
This is the **published version** of the bachelor thesis:

De Urrengoechea Cantavenera, Alejo. *Diseño e implementación de una plataforma logística en la nube integrada con servicios externos en tiempo real*.
Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026
(Enginyeria Informàtica)

This version is available at <https://ddd.uab.cat/record/334427>

under the terms of the  license.

Diseño e Implementación de una Plataforma Logística en la Nube Integrada con Servicios Externos en Tiempo Real

Alejo De Urrengoechea Cantavenera

Resumen—Este artículo presenta el diseño e implementación de una plataforma logística desarrollada sobre el ecosistema Salesforce, orientada a centralizar la gestión de pedidos y automatizar la sincronización de datos con un proveedor de transporte externo en tiempo real. El sistema combina un modelo de datos híbrido, que extiende los objetos estándar de Salesforce con entidades personalizadas, con un framework de integración asíncrona basado en Apex Queueable capaz de comunicarse con la API REST de Shippo para generar etiquetas y números de seguimiento. La automatización del ciclo de vida del pedido se completa mediante triggers, validaciones de negocio y un endpoint webhook que recibe actualizaciones de tracking en tiempo real. Por último, se ha desarrollado una capa de presentación mediante Lightning Web Components que permite visualizar y gestionar el estado logístico de los pedidos desde un panel operativo centralizado.

Palabras clave—Salesforce, Apex, integración de APIs, computación en la nube, logística, Lightning Web Components, Shippo.

Abstract—This paper presents the design and implementation of a logistics platform built on the Salesforce ecosystem, aimed at centralizing order management and automating real-time data synchronization with an external shipping provider. The system combines a hybrid data model, extending Salesforce standard objects with custom entities, with an asynchronous integration framework based on Apex Queueable capable of communicating with the Shippo REST API to generate shipping labels and tracking numbers. The automation of the order lifecycle is completed through triggers, business validations and a webhook endpoint that receives real-time tracking updates. Finally, a presentation layer has been developed using Lightning Web Components, allowing users to visualize and manage the logistics status of orders from a centralized operations dashboard.

Index Terms—Salesforce, Apex, API integration, cloud computing, logistics, Lightning Web Components, Shippo

1. INTRODUCCIÓN

La gestión logística constituye uno de los procesos más críticos dentro de cualquier organización que comercializa productos físicos, ya que cualquier ineficiencia en la sincronización de información entre los sistemas de venta y los proveedores de transporte repercute directamente en la experiencia del cliente final. Este trabajo aborda dicha problemática mediante el diseño e implementación de una plataforma logística desarrollada sobre Salesforce, una de las plataformas CRM en la nube más extendidas a nivel empresarial.

La motivación de este proyecto combina, por un lado, la necesidad real del sector logístico de contar con sistemas capaces de centralizar información que habitualmente se encuentra dispersa entre múltiples aplicaciones, y por otro, la experiencia profesional previa del autor como desarrollador en el ecosistema Salesforce, lo que ha permitido aplicar conocimientos ya consolidados a un caso de uso de mayor complejidad arquitectónica, profundizando en patrones de integración y automatización de lógica de negocio.

- E-mail de contacto: 1569298@uab.cat
- Menció: Tecnologies de la Informació
- Trabajo tutorizado por: Marc Ortega Gil (Departamento de Ciencias de la Computación)
- Curso 2025/2026
- Junio de 2026, Escola d'Enginyeria (UAB)

1.1 Objetivos

El objetivo principal de este trabajo es el diseño, desarrollo e implementación de una plataforma logística sobre Salesforce que permita centralizar la gestión de pedidos y automatizar la sincronización de datos con

servicios externos en tiempo real. De este objetivo general se derivan los siguientes objetivos específicos:

- Diseñar un modelo de datos híbrido que combine los objetos estándar de Salesforce (Order, Account, Product2) con entidades personalizadas que cubran el ciclo de vida logístico completo.
- Desarrollar un framework de integración asíncrona en Apex capaz de comunicarse con la API REST de un proveedor de transporte externo para la generación de etiquetas y números de seguimiento.
- Automatizar la lógica de negocio del ciclo de vida del pedido mediante triggers, validaciones y un mecanismo de recepción de eventos de seguimiento en tiempo real.
- Implementar una capa de presentación mediante Lightning Web Components que facilite la toma de decisiones operativas por parte del usuario final.
- Aplicar las mejores prácticas de la plataforma en materia de seguridad, escalabilidad y mantenibilidad del código.

El alcance de este trabajo se centra en el flujo correspondiente a un único almacén de origen y a pedidos de tipo expedición (Outbound), si bien el modelo de datos y la arquitectura de integración se han diseñado de forma que la incorporación de múltiples almacenes o de flujos de

aprovisionamiento (Inbound) no requiera cambios estructurales, tal y como se detalla en la Sección 4.1.

Los usuarios finales de la plataforma se corresponden con dos perfiles diferenciados: por un lado, el personal de gestión comercial, responsable de la creación y configuración de los pedidos; y por otro, el personal de operaciones de almacén, responsable de priorizar, preparar y confirmar los envíos a través del panel operativo descrito en la Sección 4.4. Esta distinción de perfiles ha condicionado el diseño de las dos interfaces desarrolladas: una integrada en el propio registro del pedido, orientada a la consulta puntual, y otra en forma de panel independiente, orientada a la gestión del volumen diario de pedidos pendientes.

1.2 Organización del documento

El resto del documento se estructura como sigue: la Sección 2 describe el estado del arte en torno a los patrones de integración logística sobre plataformas CRM en la nube. La Sección 3 detalla la metodología de trabajo seguida. La Sección 4 presenta los resultados obtenidos, incluyendo el modelo de datos, la arquitectura de integración, la automatización de la lógica de negocio y el desarrollo de la interfaz de usuario. La Sección 5 recoge las conclusiones del trabajo y las líneas de continuación. Finalmente, se detalla el uso realizado de herramientas de inteligencia artificial generativa y se incluye la bibliografía y el material de apéndice.

2. ESTADO DEL ARTE

En el ámbito de la integración entre sistemas de venta y proveedores de transporte existen actualmente dos grandes enfoques. El primero consiste en el uso de plataformas de integración especializadas (iPaaS), como MuleSoft, Boomi o Zapier, que actúan como capa intermedia entre el CRM y los servicios externos, ofreciendo conectores prediseñados a cambio de un coste de licencia adicional y de una capa extra de infraestructura a mantener. El segundo enfoque consiste en el desarrollo de integraciones nativas directamente sobre la plataforma CRM, aprovechando sus capacidades de programación interna.

Dentro del ecosistema Salesforce, la integración nativa mediante Apex es una alternativa consolidada para escenarios en los que no se requiere la complejidad ni el coste de un middleware dedicado. La plataforma ofrece mecanismos como Queueable Apex, Platform Events y Named Credentials que permiten construir integraciones asíncronas robustas sin infraestructura adicional, lo que resulta especialmente adecuado para organizaciones de tamaño pequeño o medio que ya operan sobre Salesforce.

Cabe comparar esta aproximación con el coste y la complejidad que supondría adoptar una plataforma de integración como MuleSoft, también propiedad de Salesforce desde 2018 y ampliamente utilizada en proyectos de integración empresarial de mayor envergadura. MuleSoft requiere una licencia adicional independiente de la del propio CRM, así como la configuración de un entorno de ejecución (runtime) propio —ya sea en la nube de Anypoint Platform o de forma local—, lo que introduce una infraestructura completa a desplegar, monitorizar y mantener. Para un proyecto del alcance de este trabajo, en el que la integración se limita a un único proveedor logístico y a un flujo de datos

relativamente acotado, esta complejidad adicional no estaría justificada: el coste total de propiedad de una solución MuleSoft superaría con creces el beneficio obtenido frente a una integración nativa en Apex, que no requiere licencias adicionales ni infraestructura externa a la propia organización de Salesforce. MuleSoft resulta una opción más adecuada en escenarios donde es necesario integrar un número elevado de sistemas heterogéneos de forma centralizada, o donde varios equipos de desarrollo deben reutilizar los mismos conectores entre distintos proyectos.

En cuanto a los proveedores de servicios de transporte, el mercado ofrece APIs de agregación de carriers como EasyPost, Shippo, ShipStation o Easyship, que exponen una interfaz unificada sobre múltiples transportistas (USPS, UPS, FedEx, Correos, etc.). Estas plataformas simplifican notablemente la generación de etiquetas y el seguimiento de envíos, aunque introducen particularidades en su contrato de API, como flujos de varias llamadas o limitaciones en sus entornos de pruebas, que deben tenerse en cuenta durante el diseño de la integración.

Respecto a los patrones de integración, la documentación oficial de Salesforce recomienda el uso de procesamiento asíncrono (Queueable o Batch Apex) para cualquier operación que implique llamadas HTTP desde triggers, evitando así el bloqueo de la interfaz de usuario y el incumplimiento de los límites de ejecución de la plataforma. De forma complementaria, los endpoints REST expuestos mediante Apex permiten implementar el patrón webhook para la recepción de eventos en tiempo real desde sistemas externos.

Este trabajo se posiciona dentro del segundo enfoque descrito: una integración nativa sobre Salesforce, sin middleware externo, que combina los patrones de Queueable Apex y REST webhooks para lograr una sincronización en tiempo real con el proveedor logístico Shippo, priorizando la simplicidad arquitectónica y la ausencia de costes de licencia adicionales frente a soluciones de integración de mayor complejidad.

Desde el punto de vista académico, este trabajo se enmarca dentro de los proyectos de integración de sistemas empresariales sobre plataformas de tipo Platform-as-a-Service (PaaS), un área de creciente interés dada la consolidación de Salesforce como una de las plataformas CRM con mayor cuota de mercado y demanda profesional en el sector de las tecnologías de la información.

Entre las distintas opciones de agregadores de transporte evaluadas, EasyPost y Shippo destacan por ofrecer un modelo de API REST muy similar entre sí —ambas requieren un flujo de creación de envío seguido de la compra de una tarifa—, pero difieren notablemente en la estabilidad de sus entornos de pruebas y en la cobertura de transportistas disponibles en modo sandbox. ShipStation, por su parte, está más orientado a la gestión de pedidos multicanal procedentes de marketplaces, por lo que su modelo de integración resulta menos directo para un escenario en el que el propio CRM actúa como sistema de origen del pedido. Esta comparación, realizada de forma empírica durante la fase de investigación inicial, fue uno de los factores que condicionaron la decisión final de proveedor, descrita con más detalle en la Sección 4.5.

Más allá del patrón webhook implementado en este trabajo, Salesforce ofrece también Platform Events como mecanismo de comunicación basado en publicación-suscripción para la integración de sistemas en tiempo real. Si bien este mecanismo resulta especialmente útil cuando múltiples consumidores internos deben reaccionar a un mismo evento, su uso introduce una capa adicional de configuración que no aporta valor en un escenario como el de este proyecto, donde el consumidor del evento de tracking es siempre el mismo Order asociado al envío. Por este motivo, se ha optado por el endpoint REST expuesto mediante Apex como mecanismo de recepción de eventos, resultando en una solución más directa y con menor sobrecarga de configuración.

3. METODOLOGÍA

El desarrollo del proyecto se ha llevado a cabo siguiendo una metodología ágil, concretamente Scrum, con el objetivo de garantizar flexibilidad y entregas incrementales de valor a lo largo del proceso. La estructura del trabajo se ha basado en sprints de dos a tres semanas, durante los cuales se han desarrollado, revisado y validado funcionalidades de forma iterativa.

Cada sprint ha incluido las siguientes fases: planificación, a partir de un backlog priorizado según el valor aportado al proyecto; desarrollo e implementación de las funcionalidades seleccionadas; revisión y testing mediante ejecución de Anonymous Apex y análisis de logs de depuración; y retrospectiva, para identificar mejoras en el proceso de trabajo de cara al siguiente sprint. El equipo de trabajo ha estado formado por el autor, como desarrollador de todas las funcionalidades, y por el tutor del proyecto, Marc Ortega Gil.

La planificación inicial se estructuró en cinco bloques principales: investigación inicial y análisis de requisitos; diseño de la arquitectura y del modelo de datos; un primer sprint centrado en la creación de los objetos personalizados; un segundo sprint dedicado a la integración con la API externa mediante Apex; un tercer sprint enfocado en la automatización de la lógica de negocio mediante triggers y un endpoint webhook; y un cuarto sprint orientado al desarrollo de la capa de presentación mediante Lightning Web Components.

Todo el código se ha desplegado a la organización de desarrollo de Salesforce mediante Salesforce CLI, siguiendo un flujo incremental en el que cada clase se ha desplegado y probado de forma individual antes de continuar con la siguiente. Esta forma de trabajo ha permitido detectar errores de forma temprana y mantener en todo momento una organización de desarrollo en estado funcional.

Como herramientas de soporte al desarrollo se ha utilizado Visual Studio Code, con las extensiones de Salesforce para la edición de código Apex y de Lightning Web Components, y la Developer Console de Salesforce para la ejecución de scripts de Anonymous Apex y la consulta de los logs de depuración generados durante las pruebas. Cada conjunto de clases relacionado con una misma funcionalidad se ha desplegado y validado de forma conjunta.

4. RESULTADOS

4.1 Diseño del modelo de datos

El modelo de datos de la plataforma se ha diseñado siguiendo un enfoque híbrido, combinando los objetos estándar que ofrece Salesforce con un conjunto de entidades personalizadas que cubren las necesidades específicas del ciclo logístico. El modelo se estructura en cuatro capas funcionales.

La capa comercial y de preventa se apoya en los objetos estándar Account, Opportunity y Quote, dejando preparada la plataforma para incorporar funcionalidades comerciales adicionales, como la estimación de costes de transporte previa a la confirmación de la venta.

La capa de operaciones y ejecución gira alrededor del objeto Order y de sus líneas de detalle (OrderItem). Sobre estos objetos se ha incorporado el uso de tipos de registro para diferenciar los flujos de expedición de ventas de los procesos de aprovisionamiento, así como el campo personalizado Priority__c para la gestión de prioridades según la urgencia operativa de cada pedido.

La capa de inventario y almacén utiliza los objetos estándar Location y ProductItem para gestionar la disponibilidad de stock distribuido en múltiples almacenes. A nivel de catálogo (Product2) se han incorporado los campos físicos necesarios para el cálculo de tarifas de transporte: Weight_kg__c y, tras un proceso de rediseño que se detalla en la Sección 4.5, los campos numéricos independientes Length_cm__c, Width_cm__c y Height_cm__c.

La capa core de integración logística introduce dos entidades personalizadas. Shipment__c, vinculada al Order mediante una relación de tipo Master-Detail, actúa como contenedor de los datos devueltos por la API de transporte: número de seguimiento, URL de la etiqueta y transportista asignado. Tracking_Event__c, relacionada con Shipment__c, almacena el historial completo de eventos de seguimiento recibidos, evitando la pérdida de información histórica que se produciría al sobrescribir un único campo de estado.

Adicionalmente, se ha definido el Custom Metadata Type Shippo_Config__mdt, que centraliza los parámetros de configuración de la integración —dirección de origen del envío y unidades de medida— permitiendo modificar dicha configuración sin necesidad de redespargar código.

El uso de tipos de registro en el objeto Order permite mantener una única estructura de datos para los dos grandes flujos contemplados por el modelo: los pedidos de expedición hacia el cliente final (Outbound) y los pedidos de aprovisionamiento desde proveedores (Inbound), evitando así la necesidad de duplicar objetos o de introducir lógica condicional adicional en el framework de integración. De manera similar, la tipificación de determinados campos como identificadores externos (External IDs), tanto en los almacenes (Location) como en los pedidos, resulta clave de cara a la programación en Apex, ya que permite utilizar la instrucción upsert al sincronizar datos con sistemas externos, garantizando la integridad referencial y evitando la generación de registros duplicados durante sincronizaciones repetidas.

Para cerrar el ciclo de vida completo del producto, el modelo incorpora además el objeto estándar `Asset` para la serialización individual de cada unidad vendida, así como soporte para `ReturnOrder` de cara a la logística inversa. Si bien estas entidades no han sido el foco principal de la automatización desarrollada en este trabajo, su inclusión en el modelo de datos deja la plataforma preparada para incorporar dichos procesos en futuras iteraciones sin necesidad de modificar la estructura base.

4.2 Arquitectura de integración con Shippo

La comunicación con el proveedor logístico externo se ha implementado mediante un framework de integración asíncrona basado en la interfaz `Queueable` de Apex combinada con `Database.AllowsCallouts`. Esta decisión permite desacoplar la transacción del trigger del proceso de integración HTTP, evitando bloquear la interfaz del usuario y respetando los límites de ejecución de la plataforma.

El flujo de integración sigue una arquitectura en capas: el trigger del Order detecta el cambio de estado a «Ready to Ship» y delega en un handler, que a su vez invoca a la capa de servicio correspondiente. Este servicio filtra los registros cualificados y encola una única instancia de la clase `Queueable` para toda la transacción, garantizando así la bulkificación del proceso.

Durante la implementación se detectó que la API de Shippo requiere un flujo de dos llamadas HTTP secuenciales para obtener el número de seguimiento y la etiqueta de envío. La primera llamada (`POST /shipments`) crea el envío con los datos del remitente, destinatario y paquete, y devuelve un listado de tarifas disponibles de los distintos transportistas. La segunda llamada (`POST /transactions`) compra la tarifa seleccionada —priorizando la marcada como más económica— y devuelve el número de seguimiento junto con la URL de la etiqueta lista para imprimir. Ambas llamadas se ejecutan de forma síncrona dentro del mismo job asíncrono, dado que el resultado de la primera es necesario para realizar la segunda.

En materia de autenticación, la integración se apoya en la funcionalidad estándar de `Named Credentials`, delegando a la propia plataforma la gestión segura del endpoint y de las cabeceras de autorización, evitando así codificar credenciales en texto plano dentro de las clases Apex.

A nivel de implementación, la serialización del payload hacia la API de Shippo se ha encapsulado en una clase de `Data Transfer Objects` (`ShippoDTOs`), que define la estructura de direcciones, paquetes y envíos de forma independiente del resto del framework. Esta separación permite que, ante un cambio en el contrato de la API o ante una futura migración a otro proveedor, las modificaciones queden acotadas a esta clase, sin afectar a los servicios ni a los selectores que consumen sus resultados. Los datos físicos del paquete —peso y dimensiones— se obtienen dinámicamente a partir de los registros de `Product2` asociados a las líneas del pedido, evitando cualquier valor codificado de forma fija en el código.

En cuanto a la tolerancia a fallos, la clase de integración captura las excepciones producidas durante las llamadas HTTP mediante bloques `try-catch`, registrando el error correspondiente sin interrumpir el resto de la transacción. La inserción del registro `Shipment__c` se realiza mediante la

instrucción `upsert` sobre el identificador externo del pedido, de modo que un reintento posterior del proceso no genere registros duplicados, cumpliendo así con el principio de idempotencia definido durante la fase de diseño.

4.3 Automatización de la lógica de negocio

Para garantizar la calidad de los datos antes de iniciar el proceso de envío, se ha implementado un contexto `before update` en el trigger del Order que valida que el pedido dispone de todos los datos necesarios antes de permitir el cambio de estado a «Ready to Ship». Las reglas implementadas verifican que el pedido tiene una cuenta de envío configurada, que contiene al menos una línea de producto y que todos los productos tienen sus datos físicos completos. En caso de incumplimiento, el sistema bloquea el guardado mediante el método `addError()`, mostrando al usuario un mensaje descriptivo del dato que falta.

Cada una de estas tres reglas se ha implementado como una comprobación independiente dentro del método de validación, de forma que un mismo pedido pueda mostrar simultáneamente varios mensajes de error si incumple más de una condición. Por ejemplo, si un pedido no tiene ninguna línea de producto asociada, el sistema bloquea el cambio de estado con un mensaje del tipo «No se puede marcar como Ready to Ship: el pedido no tiene líneas de producto», mientras que si algún producto del pedido no tiene completados los campos de peso o dimensiones, el mensaje indica explícitamente qué producto y qué dato concreto falta por rellenar. Este nivel de detalle en los mensajes de validación resulta especialmente relevante en el panel de operaciones descrito en la Sección 4.4, donde el operario interactúa con el pedido sin necesidad de abrir su registro completo, por lo que disponer de información precisa sobre el motivo del bloqueo evita tener que navegar hasta el detalle del Order para identificar el problema.

Una vez completado el envío, un trigger `after insert` sobre el objeto `Shipment__c` actualiza automáticamente el campo de estado logístico del Order padre a «Shipped», eliminando la necesidad de intervención manual y garantizando la consistencia entre el pedido comercial y el envío físico.

Para cerrar el ciclo de trazabilidad, se ha desarrollado un endpoint REST en Apex, mediante la anotación `@RestResource`, que actúa como receptor de los webhooks de tracking enviados por Shippo cuando el estado del paquete cambia durante el tránsito. Cada evento recibido genera un registro de `Tracking_Event__c`, y cuando el estado corresponde a la entrega del paquete, el sistema actualiza automáticamente el estado logístico del Order a «Delivered», cerrando así el ciclo completo: Draft, Ready to Ship, Shipped y Delivered.

4.4 Desarrollo de la interfaz de usuario

La capa de presentación se ha desarrollado mediante `Lightning Web Components`, integrados directamente en las páginas de registro y en una `App Page` de operaciones.

El componente `orderShipmentTracker` se integra en la página de registro del Order y muestra en tiempo real la información logística del envío asociado: transportista, número de seguimiento —enlazado directamente a la web de seguimiento del transportista correspondiente—, un botón de acceso a la etiqueta de envío, un indicador del estado actual

del envío y un historial cronológico de los eventos de tracking, cada uno representado con un icono específico según su estado. El componente gestiona de forma explícita tres estados de la interfaz: carga, vacío —con un mensaje adaptado al estado logístico del pedido— y con datos. La Figura 2 muestra el componente mostrando el estado actual de un envío entregado.

Sobre la versión inicial del componente se han incorporado cuatro mejoras de experiencia de usuario. En primer lugar, cada evento del historial de tracking se representa con un icono específico según su código de estado —entrega, en tránsito, fuera de reparto, error o devolución— mediante un mapa de equivalencias entre los códigos devueltos por Shippo y los iconos estándar de Lightning. En segundo lugar, el estado más reciente del envío se destaca mediante una insignia de color, cuya combinación de fondo y texto se aplica mediante clases CSS propias, dado que las variables de estilo de los componentes base de Lightning no son accesibles desde fuera del componente debido al uso de Shadow DOM. En tercer lugar, el número de seguimiento se muestra como un enlace que redirige directamente a la página de seguimiento del transportista correspondiente, construyendo la URL de forma dinámica según el valor del campo `Carrier_c`. Por último, el estado vacío del componente presenta un mensaje distinto en función del estado logístico del pedido, obtenido mediante el adaptador `getRecord` del módulo `uiRecordApi`, de manera que el usuario entienda en todo momento si el envío está pendiente de procesarse o si el pedido se encuentra todavía en borrador.

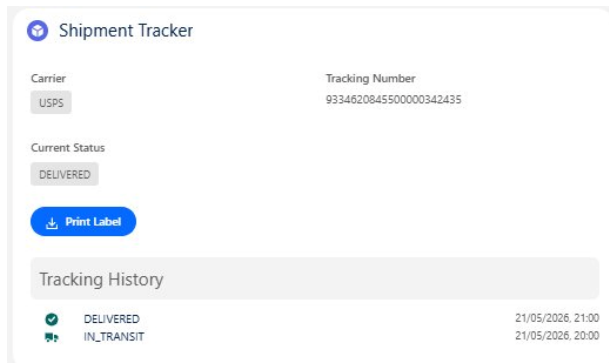


Fig. 2. Componente `orderShipmentTracker` mostrando el estado de un envío entregado, con badge de estado, icono de evento y número de seguimiento enlazado.

El componente `orderPriorityDashboard` se ha desarrollado como una App Page independiente que ofrece una vista global de los pedidos pendientes de gestión logística, ordenados por prioridad y fecha. Desde esta vista, el operario puede marcar un pedido como «Ready to Ship» directamente, lo que dispara de forma transparente todo el flujo de integración descrito en la Sección 4.2. La Figura 3 muestra el panel de operaciones con varios pedidos en distintos estados logísticos.

 The image shows a screenshot of the 'Operations Dashboard' component. It displays a table with columns for 'Order ID', 'Order To', 'Priority', 'Status', 'Date', 'Amount', and 'Action'. The table contains several rows of data, each representing a different order. The 'Status' column shows various states like 'Ready to Ship', 'In Transit', 'Out of Stock', 'Error', and 'Return'. The 'Action' column contains buttons for 'Ready to Ship', 'In Transit', 'Out of Stock', 'Error', and 'Return'.

Fig. 3. Panel de operaciones `orderPriorityDashboard` con pedidos ordenados por prioridad y estado logístico.

La acción de marcar un pedido como «Ready to Ship» desde el panel se ha implementado mediante una llamada Apex imperativa, en lugar de reactiva, dado que se trata de una operación con efectos secundarios sobre la base de datos. Tras la actualización, el componente refresca automáticamente sus datos mediante la función `refreshApex`, y notifica al usuario el resultado de la operación mediante el sistema de notificaciones de Lightning. Si la actualización incumple alguna de las validaciones descritas en la Sección 4.3, la excepción de validación se captura en el controlador y se relanza como una excepción manejable por la interfaz, de forma que el usuario recibe el mensaje descriptivo exacto que indica qué dato falta en el pedido, sin necesidad de abandonar el panel de operaciones.

4.5 Dificultades técnicas y decisiones de diseño

Durante el desarrollo se presentaron varias dificultades que condicionaron decisiones de diseño relevantes.

- Cambio de proveedor logístico: el diseño inicial contemplaba EasyPost como proveedor externo, pero se encontraron problemas persistentes de autenticación que impedían establecer una conexión estable mediante Named Credentials. Tras evaluar alternativas, se migró a Shippo, sin que ello afectara a la arquitectura de integración diseñada previamente.
- Flujo de dos llamadas: como se ha descrito en la Sección 4.2, la necesidad de encadenar dos llamadas HTTP (creación del envío y compra de la tarifa) no estaba contemplada en el diseño inicial y obligó a adaptar el flujo de la clase `Queueable`.
- Rediseño del campo de dimensiones: en el diseño inicial se contempló un único campo de texto para almacenar las tres dimensiones del producto en formato «Largo x Ancho x Alto», lo que requería un parseo manual en Apex y resultaba frágil ante errores de formato. Se sustituyó por tres campos numéricos independientes, eliminando dicha fragilidad y permitiendo enviar directamente los valores al contrato JSON de la API.
- Palabras reservadas en Apex: ciertos campos del contrato JSON de Shippo utilizan palabras reservadas del lenguaje Apex, como `async` o `from`. Para resolverlo se adoptó el patrón de construir el JSON mediante un mapa serializado, que permite usar cadenas arbitrarias como claves sin las restricciones del compilador.
- Limitaciones del entorno de pruebas: al utilizar direcciones de origen en España, el único transportista disponible en el entorno de test de Shippo era Correos, cuya integración no generaba correctamente etiquetas ni números de seguimiento. Para validar el flujo completo se configuró temporalmente una dirección de origen en Estados Unidos utilizando USPS, el único transportista

que Shippo garantiza en su entorno de pruebas. Esta limitación no afecta a la arquitectura de la solución: en un entorno productivo bastaría con actualizar el registro de configuración con los datos reales del almacén.

4.6 Validación funcional del flujo completo

La validación del sistema se ha realizado de forma incremental, desplegando y probando cada clase de forma individual mediante ejecución de Anonymous Apex y análisis de los logs de depuración generados por la plataforma. Esta forma de trabajo ha permitido detectar errores de configuración o de sintaxis de forma temprana, sin necesidad de esperar a tener el flujo completo implementado.

Para validar el flujo end-to-end, se ha simulado el ciclo de vida completo de un pedido: la creación de un Order con sus líneas de producto, el cambio de estado a «Ready to Ship» —que dispara la integración con Shippo y genera un número de seguimiento real—, la actualización automática a «Shipped» tras la creación del envío, y finalmente la simulación de los webhooks de tracking mediante una petición REST construida con Anonymous Apex que reproduce el formato del payload enviado por Shippo, verificando que el sistema crea correctamente los registros de `Tracking_Event__c` y actualiza el pedido a «Delivered» cuando corresponde.

El número de seguimiento generado durante estas pruebas se ha verificado de forma externa en la web de seguimiento de USPS, confirmando que la etiqueta y el número devueltos por Shippo corresponden a un envío real dentro de su entorno de pruebas, lo que valida el correcto funcionamiento del flujo de dos llamadas descrito en la Sección 4.2.

Si bien la validación funcional descrita ha permitido comprobar el correcto funcionamiento del flujo completo, queda pendiente la implementación de clases de test Apex que automaticen estas comprobaciones y que proporcionen la cobertura de código mínima exigida por Salesforce para el despliegue en un entorno de producción. Esta tarea, junto con la creación de datos de prueba mediante clases de tipo test data factory, se ha planificado como parte de los ajustes finales del proyecto.

4.7 Seguridad y calidad del código

El desarrollo de todas las clases Apex se ha realizado siguiendo de forma estricta las recomendaciones de seguridad y calidad de Salesforce. Todas las clases del framework se han declarado con el modificador `with sharing`, de manera que las consultas y operaciones respeten en todo momento el modelo de reparto de registros configurado en la organización.

Para evitar la duplicación de comprobaciones de seguridad a lo largo de las distintas clases selectoras, se ha desarrollado una clase utilitaria, `SObjectAccessUtil`, que centraliza la verificación de permisos de lectura sobre cada objeto antes de ejecutar cualquier consulta SOQL. De este modo, si un usuario no dispone de acceso a un objeto determinado, el selector correspondiente devuelve una lista vacía en lugar de propagar una excepción, evitando exponer información sobre la estructura interna de la organización.

Todas las consultas SOQL del proyecto se han construido utilizando variables de enlace, evitando por completo la concatenación de cadenas en su construcción y, con ello, cualquier riesgo de inyección SOQL. Asimismo, ninguna

operación de DML ni ninguna consulta se ejecuta dentro de un bucle: las operaciones se agrupan siempre en colecciones y se procesan de forma conjunta, tanto en los triggers como en los servicios y en la clase de integración asíncrona, garantizando la bulkificación del código y el cumplimiento de los límites de gobernanza de la plataforma incluso ante procesos masivos.

La lógica de negocio se mantiene completamente fuera de los triggers, que se limitan a delegar en sus respectivos handlers, siguiendo el patrón `One Trigger Per Object`. Esta separación de responsabilidades entre triggers, handlers, servicios y selectores facilita las pruebas unitarias y reduce el acoplamiento entre la lógica de automatización y el acceso a datos. De forma complementaria, los valores de los distintos estados del ciclo logístico y de prioridad se han centralizado en una clase de constantes (`AppConstants`), evitando la repetición de literales de tipo texto a lo largo del código y reduciendo el riesgo de errores al comparar o asignar valores de listas de selección.

4.8 Consideraciones de escalabilidad y mantenibilidad

El diseño del framework de integración se ha planteado teniendo en cuenta los límites de gobernanza de Salesforce desde la fase inicial. El uso de `Queueable Apex` en lugar de llamadas síncronas permite que, ante un incremento del volumen de pedidos, el procesamiento de los envíos no comprometa el tiempo de respuesta percibido por el usuario al guardar un pedido. No obstante, dado que Salesforce limita el número de jobs `Queueable` encolables desde un mismo contexto de ejecución, una evolución natural de la arquitectura hacia volúmenes de procesamiento masivo pasaría por sustituir la clase `Queueable` actual por una implementación de `Batch Apex`, manteniendo la misma capa de servicio y de integración sin necesidad de modificar el resto del framework.

Desde el punto de vista de la mantenibilidad, la separación estricta entre selectores, servicios, handlers y la capa de integración permite que cualquier cambio futuro —ya sea la incorporación de un nuevo transportista, la adición de un nuevo tipo de evento de tracking o la extensión del modelo de datos a nuevos almacenes— quede acotado a la capa correspondiente, sin que sea necesario revisar de forma transversal el resto del código. Esta organización, unida al uso de `Custom Metadata` para la configuración y de una clase de constantes para los valores del ciclo logístico, reduce significativamente el coste de mantenimiento a medio plazo de la plataforma.

En relación con la observabilidad del sistema, los errores capturados durante las llamadas a la API de Shippo quedan registrados mediante `System.debug()` en los logs de la plataforma, lo que resulta suficiente durante la fase de desarrollo. De cara a producción, sería recomendable sustituir este mecanismo por un objeto personalizado de tipo `Integration_Log__c`, que almacenara de forma persistente cada error junto con el pedido afectado y la respuesta recibida de la API. Esto permitiría implementar un proceso programado (`Scheduled Apex`) que reintentara automáticamente los envíos fallidos, sin depender de la revisión manual de los logs de depuración por parte de un administrador.

4.9 Comparativa con la planificación inicial

Comparando los resultados obtenidos con la propuesta inicial del proyecto se observa que tanto el modelo de datos como la arquitectura de integración se han mantenido fieles al planteamiento original en sus líneas generales: arquitectura híbrida de datos, integración asíncrona mediante Queueable Apex y automatización mediante triggers y servicios. Las principales desviaciones respecto al plan inicial han sido de carácter técnico y no estructural, como se ha detallado en la Sección 4.5, lo que confirma que el esfuerzo invertido en la fase de diseño previo al desarrollo ha resultado proporcional al grado de estabilidad alcanzado por la arquitectura.

En cuanto a la planificación temporal, recogida en el diagrama de Gantt del Apéndice (Figura A4), el desarrollo de los Sprints 2 y 3 se ha completado dentro de los plazos previstos, mientras que el Sprint 4, correspondiente al desarrollo de la interfaz de usuario, se ha extendido ligeramente respecto a la estimación inicial debido a la incorporación de las mejoras de experiencia de usuario descritas en la Sección 4.4, que no estaban contempladas con ese nivel de detalle en la planificación original. Esta desviación se ha absorbido sin afectar a la fecha de entrega de este informe, gracias al margen incluido en la fase de validación técnica y ajustes finales.

Desde el punto de vista de los requisitos funcionales y no funcionales definidos en el informe inicial, puede afirmarse que los relativos al modelo de datos híbrido, a la arquitectura de integración avanzada en Apex y a la automatización de la lógica de negocio se han cubierto en su totalidad. El requisito de optimización del seguimiento y trazabilidad se ha cubierto mediante el objeto `Tracking_Event__c` y el componente `orderShipmentTracker`, mientras que el requisito de desarrollo de interfaz dinámica se ha cubierto mediante los dos componentes Lightning Web Components descritos en la Sección 4.4. Los requisitos no funcionales de seguridad, integridad de datos y rendimiento en tiempo real se han abordado de forma transversal mediante las decisiones recogidas en la Sección 4.7.

5. CONCLUSIONES Y TRABAJO FUTURO

El trabajo desarrollado hasta el momento ha permitido alcanzar los objetivos principales planteados: se ha diseñado un modelo de datos híbrido que extiende el estándar de Salesforce con entidades propias del dominio logístico, se ha construido un framework de integración asíncrona robusto y escalable con un proveedor de transporte externo, se ha automatizado el ciclo de vida completo del pedido mediante triggers y un mecanismo de recepción de eventos en tiempo real, y se ha desarrollado una capa de presentación que permite gestionar la operativa logística desde dos componentes Lightning Web Components complementarios.

Las dificultades encontradas durante el desarrollo, en particular el cambio de proveedor de integración, el rediseño del modelo de datos y el descubrimiento del flujo de dos llamadas de la API de Shippo, han supuesto retos técnicos que han contribuido a consolidar las decisiones arquitectónicas y a comprobar la flexibilidad del diseño inicial frente a cambios no previstos.

En conjunto, los resultados obtenidos demuestran que es posible construir, sobre la plataforma Salesforce y sin recurrir

a middleware externo, una integración logística en tiempo real que cumple con los principios de seguridad, escalabilidad y mantenibilidad exigidos a nivel profesional. La arquitectura en capas adoptada ha demostrado además su flexibilidad ante cambios no previstos, como el cambio de proveedor logístico o la incorporación de un segundo componente de interfaz, sin que ello haya requerido modificar las capas de servicio o de integración ya existentes.

A nivel personal, este trabajo ha permitido profundizar en patrones de integración asíncrona y en el desarrollo de interfaces mediante Lightning Web Components, dos áreas en las que la experiencia profesional previa del autor era más limitada que en el desarrollo de lógica de negocio mediante Apex. La necesidad de adaptar el diseño ante los cambios no previstos descritos en la Sección 4.5 ha resultado, en este sentido, una de las partes más formativas del proceso.

De cara a una futura puesta en producción, sería conveniente definir un conjunto de métricas que permitan evaluar el impacto real de la plataforma: el tiempo medio transcurrido entre la confirmación de un pedido y la generación de su etiqueta de envío, el porcentaje de pedidos que completan el ciclo logístico sin intervención manual, o el número de incidencias detectadas mediante las validaciones implementadas frente a las que se habrían producido sin ellas. Estas métricas permitirían cuantificar de forma objetiva la mejora operativa introducida por el sistema.

Como trabajo futuro queda pendiente la implementación de los tests unitarios Apex necesarios para garantizar la cobertura de código requerida por la plataforma, así como la validación técnica final del conjunto de la solución y los ajustes que de ella se deriven antes de la entrega del proyecto. Adicionalmente, se considera de interés explorar la incorporación de notificaciones proactivas al cliente final a partir de los eventos de tracking ya recibidos por el sistema.

USO DE LA INTELIGENCIA ARTIFICIAL GENERATIVA

A lo largo del desarrollo de este trabajo se ha utilizado IA generativa como herramienta de apoyo en tareas concretas, sin que ello haya sustituido el trabajo de diseño y desarrollo propio del autor.

En las fases iniciales del proyecto se utilizó como apoyo para estructurar la documentación, organizar la jerarquía de los informes de progreso y comprobar que se incluían todos los apartados necesarios, así como para el refinamiento lingüístico de los documentos entregados, corrigiendo errores gramaticales y mejorando la coherencia del texto.

Durante las fases de desarrollo, se ha empleado como herramienta de apoyo para revisar fragmentos de código Apex, identificar posibles mejoras de legibilidad y verificar que la estructura de algunas clases seguía los estándares de la plataforma, así como complemento a la documentación oficial de Salesforce para resolver dudas puntuales sobre la sintaxis de métodos estándar y las mejores prácticas de desarrollo.

Estas consultas se concentraron principalmente en los Sprints 2 y 3, en los que la curva de aprendizaje asociada a la API externa y a determinadas particularidades del lenguaje

Apex era mayor, reduciéndose de forma natural a medida que el desarrollo avanzaba hacia las fases de interfaz de usuario.

AGRADECIMIENTOS

El autor desea agradecer a Marc Ortega Gil, tutor de este trabajo, su seguimiento y orientación a lo largo de todo el proyecto.

BIBLIOGRAFÍA

- [1] Salesforce, "Apex Developer Guide", Salesforce Inc. [En línea]. Disponible: <https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/> [Consultado: mayo 2026]
- [2] Salesforce, "Asynchronous Apex", Salesforce Inc. [En línea]. Disponible: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_async_overview.htm [Consultado: mayo 2026]
- [3] Salesforce, "Lightning Web Components Developer Guide", Salesforce Inc. [En línea]. Disponible: <https://developer.salesforce.com/docs/component-library/documentation/en/lwc> [Consultado: mayo 2026]
- [4] Shippo, "Shippo API Reference", Shippo Inc. [En línea]. Disponible: <https://goshippo.com/docs/reference> [Consultado: junio 2026]
- [5] K. Schwaber y J. Sutherland, "The Scrum Guide", 2020. [En línea]. Disponible: <https://scrumguides.org> [Consultado: junio 2026]
- [6] Salesforce, "Named Credentials and External Credentials", Salesforce Inc. [En línea]. Disponible: https://developer.salesforce.com/docs/atlas.en-us.platform_connect_apis.meta/platform_connect_apis/ [Consultado: junio 2026]
- [7] Salesforce, "SOQL and SOSL Reference", Salesforce Inc. [En línea]. Disponible: https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/ [Consultado: junio 2026]
- [8] Salesforce, "Apex REST Callouts", Salesforce Inc. [En línea]. Disponible: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_callouts.htm [Consultado: junio 2026]

APÉNDICE

A1. Arquitectura de clases Apex

La Figura A1 muestra la organización en capas del framework Apex desarrollado, siguiendo el patrón One Trigger Per Object y los principios SOLID: los puntos de entrada (triggers, endpoint REST y controladores LWC) delegan en handlers, estos en servicios con la lógica de negocio, y estos a su vez en selectores —que centralizan las consultas SOQL— y en la capa de integración asíncrona con la API externa.

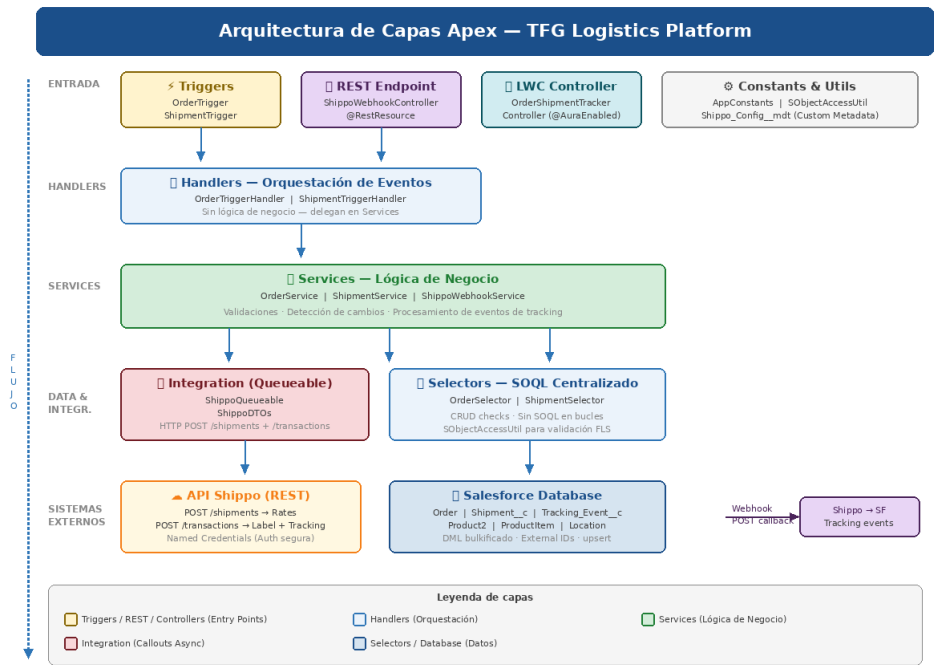


Fig. A1. Arquitectura de capas del framework Apex desarrollado.

A2. Modelo de datos (Diagrama Entidad-Relación)

La Figura A2 muestra el diagrama Entidad-Relación del modelo de datos híbrido descrito en la Sección 4.1, donde se observan las cuatro capas funcionales —comercial, operaciones, inventario e integración logística— y las relaciones entre los objetos estándar y personalizados de la plataforma.

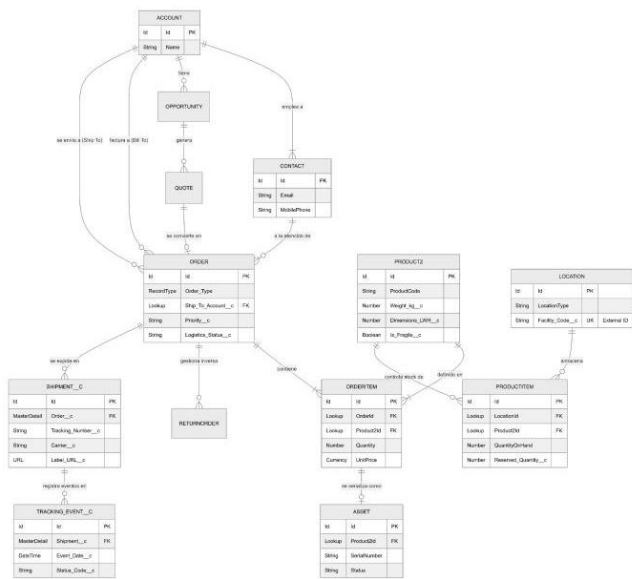


Fig. A2. Diagrama Entidad-Relación del modelo de datos de la plataforma.

A3. Diagrama de secuencia de la integración

La Figura A3 representa el diagrama de secuencia del flujo de integración asíncrona descrito en la Sección 4.2, mostrando la interacción entre el trigger del Order, la clase Queueable y la API externa. El diagrama refleja el patrón arquitectónico original del proyecto, el cual se mantuvo sin cambios tras la migración del proveedor logístico de EasyPost a Shippo.

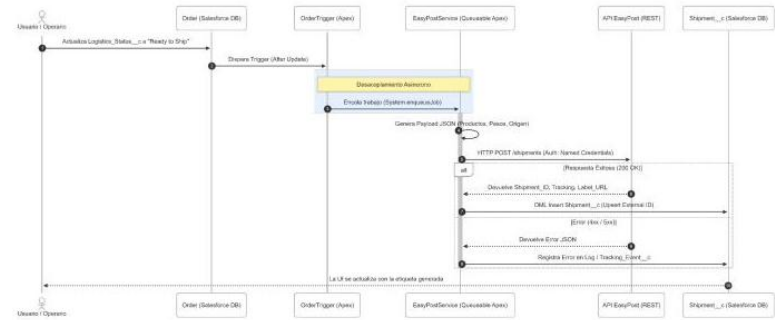


Fig. A3. Diagrama de secuencia del flujo de integración asíncrona con el proveedor logístico.

A4. Planificación del proyecto

La Figura A4 muestra la planificación inicial del proyecto en forma de diagrama de Gantt, estructurada en sprints de dos a tres semanas según la metodología descrita en la Sección 3.

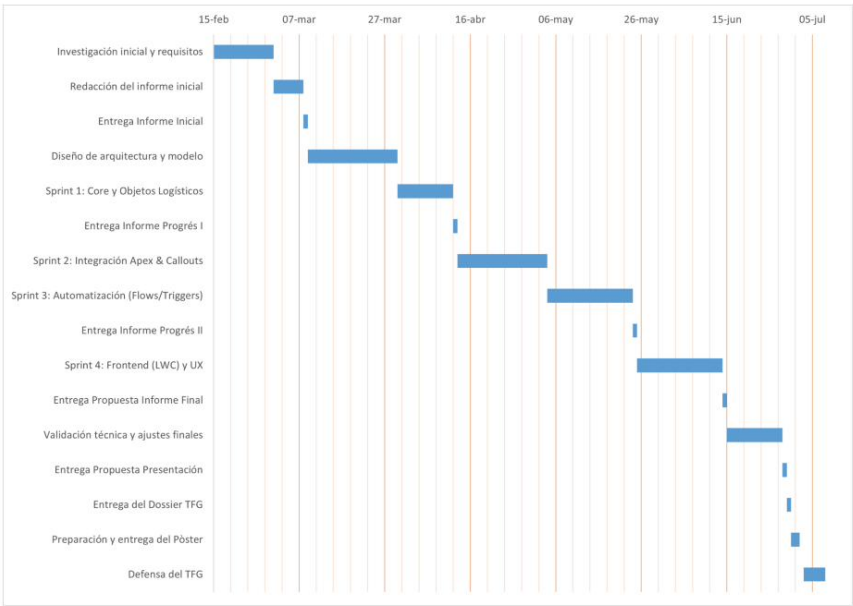


Fig. A4. Diagrama de Gantt de la planificación inicial del proyecto.