clave i | Software solutions for business

# Experiencia de una Pyme en la migración de sus aplicaciones legacy al cloud

## "Descomponiendo el monolito"


**David Perona**
**Arquitecto de Software**
**en Clave Informática S.L.**
dperona@clavei.es

**Antonio de Rojas**
**Director de Tecnología**
**en Clave Informática S.L.**
aderojas@clavei.es



GOCLOUD

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

red.es

Plan de Recuperación, Transformación y Resiliencia

# 0. Abstracto

El objetivo del presente artículo es ofrecer una visión completa del camino llevado a cabo por una Pyme para evolucionar varias ERP (Enterprise Resource Planning), que dan servicio a una cartera muy amplia de clientes, en una única aplicación ERP 100% cloud, y con el propósito, además, de utilizar los servicios en la nube de algunos de los proveedores más conocidos actualmente como es el caso de Amazon AWS, Microsoft Azure o Google Cloud.

El trabajo aquí descrito supone varios desafíos que enumeramos
a continuación:

• **Tecnológico.** Partimos de aplicaciones software desarrolladas con una tecnología legacy en algunos casos no soportada por los propios fabricantes. Y se une, además, la complejidad de unificar en un solo producto el funcionamiento y conocimiento de varias ERPs que dan servicio a sectores completamente diferentes, con sus propios modelos de datos, etc.

• **Organizacional.** Debemos distribuir recursos de los equipos que están desarrollando y manteniendo las ERPs actuales al nuevo desarrollo, y además habilitarlos mediante formaciones específicas para adquirir el conocimiento necesario.
Y no sólo los equipos de desarrollo tienen que prepararse, sino también los equipos de consultoría que tendrán que dar soporte y formación a los clientes, y en etapas tempranas del desarrollo mantener reuniones con los clientes para mostrarles la evolución del producto, conocer su feedback, y en general, involucrarlos en la construcción del nuevo aplicativo. Punto que consideramos vital para el éxito del proyecto.

• **Empresarial.** Este punto a su vez se puede desglosar a su vez en otros, cada cuál con una entidad considerable:
Gestión de los costos de la nueva plataforma.
Migración de los datos de los clientes hacia el nuevo modelo de datos de tal forma que no haya pérdida de datos, que sea repetible tantas
veces sea necesario, y no suponga una parada de negocio para el cliente.

Como veremos más adelante los riesgos derivados de la ejecución de este proyecto son importantes y se hace necesario una correcta estrategia de análisis y desarrollo para inclinar la balanza hacia el éxito.

# 1.Punto de Partida

A fecha anterior al inicio de este artículo de investigación (septiembre de 2022), el punto del que partimos era el siguiente:

- Partimos de una posición en la cual tenemos un conjunto de aplicaciones empresariales (ERP) que ofrecen soluciones a distintos sectores a los que nos dirigimos, realizados todos ellos con tecnología legacy. Se trata de aplicaciones de una gran envergadura en cuanto a su conjunto de funcionalidades, que suponen millones de línea de código, con miles de tablas y que han ido complicando la estructura de datos y la trama de desarrollo a lo largo de su historia y de las progresiones versiones que se han evolucionado, dentro de un ámbito de ejecución local.

Estas aplicaciones cubren todas las necesidades de funcionamiento de las diferentes empresas, incluyendo procesos generales de gestión, almacén, producción, financieros y contables.

La instalación de estas aplicaciones se realiza de forma aislada, no concurrente, y para cada cliente individual, localizadas normalmente en sus servidores empresariales o en VPS (“Servidores Privados Virtuales”) de uso exclusivo por parte de cada uno de ellos, en los que además reside la base de datos que manejan nuestras aplicaciones para su funcionamiento.

Todas estas aplicaciones tienen modelos de datos totalmente heterogéneos entre sí, e incluso sistemas de bases de datos diferentes, y con código acoplado a dichos sistemas por el uso de procedimientos almacenados, triggers y otros mecanismos similares.

Los grandes inconvenientes que sufrimos por esta tipología de aplicaciones, repositorio de datos y despliegue en servidores locales individuales, se resumen en los siguientes aspectos:

• **Tecnología de desarrollo obsoleta,** que ha perdido el soporte oficial y las actualizaciones del fabricante.

• **Código monolítico,** complicado a lo largo de los años, por la cantidad de modificaciones y personalizaciones realizadas a la cartera de los clientes, que en muchos de los casos incluye personalización de funciones para un grupo o un único cliente. Ha llegado un momento en que su mantenimiento se ha tornado inmanejable, y en los últimos años sufrimos una tendencia ascendente a la introducción de bugs constantes en el desarrollo, lo que ha desbordado a nuestro equipo de control de calidad y ha enlentecido el desarrollo de nuevas funciones.

• **Dificultad para optimizar** el desempeño de los equipos de soporte y desarrollo, puesto que en muchos casos se trata de replicar las mismas funcionalidades en todas las herramientas. Es el caso, por ejemplo, de la introducción de cambios legales en la gestión y documentos de negocios, que pueden llegar a afectar a todas. Igualmente tampoco se pueden optimizar los recursos de los equipos de soporte puesto que se trata de aplicaciones totalmente heterogéneas entre sí y que necesitan un conocimiento muy específico.

• Dificultad en incorporar nuevas personas al equipo de desarrollo, teniendo en cuenta como hemos comentado que están desarrolladas con tecnología obsoleta, y no se encuentra disponibilidad de esos recursos para su contratación.

• Se dificulta enormemente la incorporación de nuevas tecnologías, como es el caso de los desarrollos de funciones que requieren el uso de algoritmos basados en técnicas de Inteligencia Artificial, que en la actualidad son proporcionadas en forma de servicios por las propias plataformas en la Nube.

• Imposibilidad de poder optimizar la infraestructura hardware de las empresas ante la obsolescencia de la misma, además de no poder gestionar de forma eficaz cambios en los recursos del sistema, ante picos de demanda, o determinados procesos que se necesiten ejecutar en momentos puntuales.

Ante la necesidad de abordar el proceso de migración de las aplicaciones legacy a la nube, y antes de embarcarnos en el actual proyecto, hicimos una evaluación de diferentes herramientas del mercado para valorar si era posible la adopción de alguna de ellas para facilitar y abaratar en lo posible el proceso de migración tanto de aplicaciones como de bases de datos, desde el entorno operativo local hasta la nube. Pero observamos que estaban lejos de cubrir las necesidades de los sectores a los que necesitábamos dirigirnos y para los cuales tenemos un know-how muy importante adquirido a lo largo de los años de experiencia en la comercialización y desarrollo de nuestras actuales aplicaciones.

Y por otra parte, nos preocupaba mucho el proceso de transición de los clientes de las aplicaciones legacy a la aplicación Cloud, ya que se buscaba un proceso sencillo y no traumático para el usuario final, bajo el amparo de una metodología que pudiera ser utilizada para realizar ese proceso desde cualquier aplicación legacy a la nueva aplicación Cloud a construir.

# 2. Novedades Tecnológicas

Las principales innovaciones tecnológicas que surgen de este proyecto con respecto a la situación original de la que se parte, referido a las soluciones de gestión empresarial de uso en ámbito operativo local, y teniendo en cuenta que partimos de soluciones que requieren infraestructuras on-premise (en local, tanto SW como HW), se pueden resumir en:

**- Elasticidad y Escalabilidad:** en la nube, los recursos informáticos y de almacenamiento son escalables de forma dinámica según la demanda. Esto significa que podemos aumentar o reducir los recursos de manera automática en función de las necesidades del negocio. En contraste, las soluciones on-premise a menudo requieren una inversión anticipada en hardware y capacidad que puede ser difícil de ajustar según las fluctuaciones en la carga de trabajo.

**- Disponibilidad y Resiliencia**: las soluciones en la nube ofrecen arquitecturas altamente disponibles y resistentes a caídas, gracias a las redundancias incorporadas. Los proveedores de servicios en la nube garantizan mediante SLAs (Service Level Agreements) la disponibilidad de servicios, lo que puede ser difícil de lograr con soluciones on-premise sin una inversión significativa en infraestructura redundante.

**- Flexibilidad y Agilidad:** las plataformas cloud nos están ofreciendo una serie de herramientas que nos permitirán agilizar al máximo el desarrollo software del proyecto sin la necesidad de complejos despliegues en infraestructuras on-premise.

**- Actualizaciones y Mantenimiento Automatizado:** los proveedores cloud se encargan del mantenimiento de la infraestructura subyacente, incluidas las actualizaciones de software y seguridad. Esto libera a los equipos de TI de tareas operativas rutinarias y les permite centrarse en actividades de mayor valor, como el desarrollo de aplicaciones y la optimización de procesos. Esto en lo relativo a la infraestructura, pero también cabe decir que en lo relacionado con el software la incorporación de sistemas de integración continua nos facilitará la actualización de versiones de forma concurrente y masiva para atender simultáneamente a todos los clientes.

**- Costos Operativos Optimizados:** los servicios en la nube reducen drásticamente los costes operativos y de mantenimiento. Además, poseen herramientas avanzadas de monitorización que permiten de una forma proactiva el tener controlados los costes de la infraestructura y servicios usados. Con vistas al cliente final, los beneficios son importantes puesto que no va a necesitar mantener ni evolucionar una infraestructura compleja para dar soporte a las exigencias que requieren nuestras aplicaciones on-premise actuales.

**- Acceso desde cualquier lugar y cualquier dispositivo:** al tratarse de una base de aplicaciones de tipo web los usuarios van a poder trabajar con la aplicación desde cualquier lugar del mundo accediendo a través de un navegador, y desde cualquier dispositivo dado que se trata de una App con diseño adaptativo.

- Seguridad documental con trazabilidad: el intercambio de documentos entre empresas se

hará utilizando la tecnología blockchain, lo cual permitirá la realización de contratos inteligentes para proveer de trazabilidad y garantizar el intercambio seguro de documentos.

- Asistente virtual: integración de un chatbot para facilitar a los usuarios la utilización de la aplicación y todas sus funcionalidades, que actuará el modo de un asistente virtual para facilitar la transición hacia el nuevo modelo de funcionamiento y hacia las nuevas funcionalidades incluidas en las diferentes herramientas.

Como se menciona en los puntos anteriores, además de buscar las mejoras propias de las plataformas cloud, se pretende incorporar otras innovaciones relacionadas con tecnologías como Inteligencia Artificial y Blockchain.

# 3. Dificultades y Riesgos Técnicos

La previsión de estos notables avances tecnológicos, como fruto del proyecto a realizar, es pareja a un elevado nivel de dificultades y riesgos técnicos para el proyecto, que podemos resumir en:

**- Problemas asociados a la compatibilidad:** las aplicaciones legacy utilizan ciertas tecnologías como integraciones con máquinas de producción, balanzas, etcétera, que pueden representar problemas de compatibilidad y rendimiento adecuado en la aplicación cloud. Es posible que esos problemas de compatibilidad representen inconvenientes imposibles de solucionar y que por tanto impidan la adopción de las soluciones cloud.

- **Incremento exponencial de complejidad:** la migración al cloud puede resultar compleja y requerir un esfuerzo considerable en términos de tiempo y recursos. Estamos hablando de la migración de varios modelos de datos completamente diferentes que hay que llevar hacia un modelo de datos único, así como la integración con aplicaciones de terceros mediante APIs que es necesario replantear y seguramente recodificar nuevamente. Si estamos hablando de diferentes aplicaciones, con miles de tablas cada una, que se deben unificar y migrar sin pérdida de datos desde las aplicaciones legacy.

**- Rendimiento y Escalabilidad:** es necesario migrar millones de registros de las bases de datos origen de los clientes, que compartirán infraestructura y repositorio con otras bases de datos de clientes. Si el sistema no está bien optimizado para soportar cargas de trabajo importante, puede representar problemas de rendimiento muy importantes, que incluso supongan un retraso en el funcionamiento del cliente con respecto al trabajo en la legacy. Es, por tanto importante, realizar pruebas exhaustivas de carga y rendimiento para identificar y abordar cuellos de botella y asegurar que la aplicación pueda escalar de manera efectiva para satisfacer las demandas cambiantes. El riesgo principal es que sea imposible mantener una infraestructura que dé soporte a la importante cantidad de datos que van a manejar nuestros clientes, bajo unos parámetros de rendimiento aceptable, y con unos costes asumibles.



**- Seguridad y Cumplimiento:** la migración al cloud puede plantear preocupaciones adicionales en términos de seguridad y cumplimiento, especialmente si la aplicación maneja datos sensibles o está sujeta a regulaciones específicas (como GDPR o HIPAA). Es fundamental implementar medidas de seguridad robustas, como el cifrado de datos, la gestión de identidades y accesos, y el monitoreo continuo para proteger la información confidencial y cumplir con los requisitos regulatorios.

**- Costos y Optimización:** si no se gestionan adecuadamente, los costos asociados con la migración al cloud pueden aumentar significativamente, especialmente si no se optimizan los recursos y se subestiman las necesidades de capacidad. Será importante realizar un análisis detallado de costos y adoptar prácticas de optimización, como el uso de instancias reservadas y la monitorización de recursos, para controlar los costos a largo plazo. Sumado a lo comentado en el riesgo relativo a rendimiento y escalabilidad, puede llegar a un punto en el que sea inasumible los costes derivados de una infraestructura que proporcione un rendimiento adecuado.

**- Experiencia de Usuario:** los clientes están habituados a trabajar con las aplicaciones legacy que por regla general tienen una usabilidad bastante diferente que las aplicaciones web, entre ellas por ejemplo el manejo del teclado y otros dispositivos de interacción. El desarrollo de las nuevas aplicaciones se realizará teniendo muy presente ese tipo de requisitos de usabilidad, pero su adopción puede suponer un problema para los usuarios actuales. Podemos llegar a encontrarnos con el rechazo frontal por parte de los usuarios de las aplicaciones legacy, al tener que asumir nuevos paradigmas de interacción, así como cambios sustanciales en la interfaz.

En resumen, podemos concluir que el proceso de migración de aplicaciones legacy al cloud conlleva una serie de riesgos técnicos que deben ser abordados desde una planificación cuidadosa, pruebas exhaustivas y medidas de mitigación adecuadas, pero sin perder de vista que todo el proceso tiene un nivel de incertidumbre que no podemos evitar, ya que se trata de una labor no desarrollada previamente, que implica a cientos de bases de datos diferentes, a múltiples aplicaciones de gestión empresarial y a centenares de clientes que deben mantener en marcha sus entornos operativos durante todo el proceso de migración, por lo que hay que tener en cuenta en todo momento la posibilidad de enfrentarnos a riesgos imposibles de solucionar.

Partiendo del punto inicial explicado en los apartados anteriores debemos abordar 2 puntos críticos, digamos “core”, para asegurar el resto del proyecto:

• Cómo abordar el análisis funcional del nuevo aplicativo.

• Cómo plantear la transición de los clientes actuales o de otras aplicaciones legacy al nuevo producto, resultado del proyecto, de una forma transparente y sin pérdida de información.

**Tratamos a continuación cada uno de estos puntos.**

# 4. Estrategia para la realización del Análisis Funcional

Ante la dificultad de descomponer funcionalmente varios monolitos que constituían nuestras aplicaciones legacy, con cientos de formularios, y millones de línea de código desarrolladas a lo largo de más 20 años de evolución de productos, nos preguntamos:
"¿ Sabemos cuántas de esas funcionalidades realmente aportan valor a nuestros clientes porque realmente las estén usando ?"
Y aunque contamos con un equipo sénior, tanto de analistas como de desarrolladores, la respuesta sincera fue: "No, no estamos seguros de cuál o tal funcionalidad está siendo usada".
Evidentemente, conocemos nuestros productos. Sabemos qué aporta más valor a los sectores a los que nos dirigimos, pero puestos a trasladar el conocimiento al nuevo producto, no nos podíamos quedar en una "foto a ojo de pájaro", sino que debíamos entrar en el detalle.

**Debíamos analizar con la mayor granularidad que pudiésemos:**

• Uso de pantallas.
• Uso de controles en cada pantalla, porque es posible que hubiese campos o botones que nadie usara o muy pocos usuarios.
• Parámetros de configuración que afectasen a funcionalidades o incluso a apariencia de controles en pantallas (rejillas de datos, pestañas, orden de tabulación, visibilidad de controles,….)
Alguna de esa información ya la guardábamos localmente en las tablas de nuestras aplicaciones, y otra tuvimos que incorporarla para su obtención en las aplicaciones.

El objetivo final era poder recolectar esa información de forma centralizada en nuestra infraestructura y poder realizar los análisis necesarios para la obtención de conclusiones que nos permitiera focalizar el proceso de análisis.
Para ello, lo primero que tuvimos que obtener es la autorización para la recopilación de esa información desde nuestros clientes antes de lanzar a producción la versión con la recogida de esa información.

Poco a poco, fuimos recogiendo las métricas que nos iban enviando los clientes y pudimos llegar a conclusiones como las siguientes:

• Del total de formularios de las aplicaciones, en promedio sólo se estaban usando alrededor del 60% en un 80% de nuestros clientes.
• Y cuando hablábamos de parámetros de configuración que cambiaban cierta funcionalidad o incluso apariencia de las pantallas, ese porcentaje bajaba a un 40%
• El porcentaje de utilización superaba el 80% cuando se trataba de los módulos contables y financieros. Cuestión totalmente lógica puesto que se trata de funcionalidades muy estándares y normalmente con muy poca personalización



Se muestra a continuación un gráfico que representa visualmente a lo que nos estamos refiriendo:

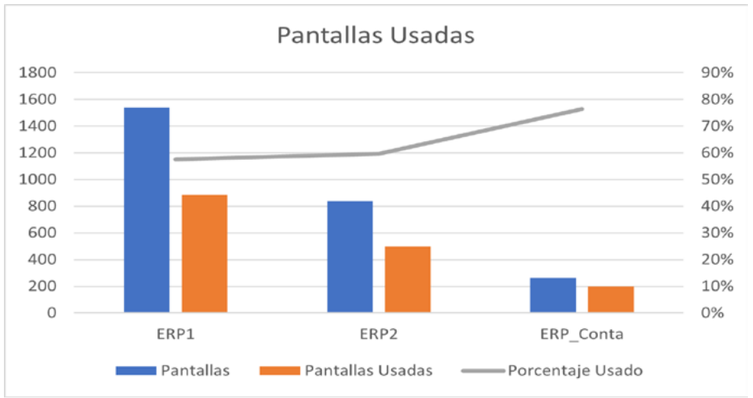



*Fig 1. Representación Métricas Utilización*

Mantuvimos los análisis durante varios meses y el estudio final nos permitió focalizarnos en realizar el análisis en aquellas funcionalidades que ahora sí, teníamos la certeza de que un porcentaje muy importante de nuestros clientes las estaban usando.

Y otra consecuencia de estos resultados fue que nos permitió realizar una primera estimación del coste de desarrollo. Cierto que la tecnología a emplear no la teníamos del todo decidida (Angular vs React, .Net Core vs Java, Azure vs AWS, SQL Server vs MySQL vs PostgreSQL,….), pero teníamos conocimientos de las diferentes alternativas y sobre todo en base a la complejidad del código seleccionado para el futuro desarrollo, establecimos una serie de parámetros para realizar los cálculos en base a ellos:

• Líneas de código a desarrollar.
• Pantallas a instanciar.
• Controles de entrada en cada pantalla.
• Objetos instanciados.
• Métodos dentro de cada pantalla.

Cada uno de estos parámetros, con un peso específico diferente, nos permitió establecer una primera aproximación al coste temporal de análisis y desarrollo. Existen muchas publicaciones al respecto de las estimaciones paramétricas, véase por ejemplo la que aquí se referencia: "Development of a Parametric Estimating Model for Technology-Driven Deployment Projects" (Watson, Young Hook Kwak) [1]

# 5. Estrategia para la migración de datos de las aplicaciones legacy a la cloud

La otra cuestión central que debíamos plantear al comienzo del proyecto era cómo permitir que los clientes de las aplicaciones legacy pudieran pasar a la aplicación cloud de una forma transparente, repetible y sin pérdida de información.

## En concreto, los requerimientos perseguidos eran los siguientes:

• Se debe ejecutar en múltiples ocasiones para el mismo cliente con el objetivo de que el cliente pueda probar nuevas funcionalidades de la aplicación, o varias pruebas de migración.
• No debe representar parada de negocio para el cliente.
• Se debe poder migrar todos los datos del cliente (maestros, pedidos, albaranes, facturas, almacén, órdenes de producción, cartera, etcétera).
• Se debe poder ejecutar bajo demanda por parte del cliente o del consultor encargado de la instalación. Pero también debería poder programarse para ejecutarse de forma desatendida.
• Debe poder configurarse el período a incluir de los datos (X meses, X años, etcétera)
• Deben poder seleccionarse las empresas del cliente a traspasar.
• Se deben poder traspasarse los datos de todas las aplicaciones legacy actuales hacia un único modelo de datos de la aplicación cloud a desarrollar.
• El proceso debe registrar mediante un log de las operaciones realizadas.
• El proceso debe notificar mediante un correo y también de forma interactiva de errores producidos. Entre otros, por ejemplo, de pérdida de información.

Atendiendo a estos requerimientos montamos una PoC (“Prueba de Concepto”) para analizar varias estrategias de migración de datos.

La primera de ellas consistió en la utilización de Apache Kafka **[2]**.

La sincronización de datos y esquemas entre sistemas con Kafka es muy potente. Además, existen una cantidad importante de conectores para poder enlazar con SQL Server, PostgreSQL, etc. Con lo cual, teniendo la infraestructura montada (clúster, nodos, zookeeper, los conectores, etc..), se consigue que los cambios que se produzcan en las tablas monitorizadas lleguen a la cola que le corresponda, pudiendo capturarlo, transformarlo, y dejarlo en otra cola que cogerá el conector destino y lo enviará donde se configure.

Kafka es un sistema muy seguro, potente, y diseñado para escenarios mucho más complejos incluso del que nosotros proponemos. Aun así, la gestión de la infraestructura es muy compleja. Configurar cada clúster, los brokers que asignemos a cada uno de ellos, las diferentes particiones de cada topic (se recomiendan 3 por topic), etcétera, es tremendamente complejo. Aunque la creación se puede automatizar, el mantenimiento implicaría un coste operacional muy alto por parte de nuestro personal.
Es por ello, que finalmente lo descartamos como una solución válida, por el volumen total de transacciones y procesamiento que estamos manejando: miles de tablas, por aplicación, y por cada una de ellas un mapeo que tiene que resolverse en tiempo de evaluación de cada mensaje, con un número de mensajes/segundo que no podemos predecir.
La última estrategia que consideramos fue la construcción de un conector personalizado de desarrollo propio o bien a través de software de terceros como es el caso de Pentaho **[3]**, y concretamente de su módulo PDI (“Pentaho Data Integration”) **[4]**
El objetivo que se proponía era construir un conector que permita volcar los datos de las aplicaciones heredadas de una forma transparente para el cliente final, sin pérdida de información, y tantas veces como sea necesario sin necesidad de parada de servicio por parte del cliente.
Nuestra idea inicial era instalar los conectores en los servidores de cada cliente con el objetivo, por cuestiones de seguridad, de no requerir abrir puertos que puedan comprometer la seguridad de la infraestructura del cliente. Funcionan en segundo plano sin provocar interrupción en el trabajo habitual de los usuarios.

## El conector estaría compuesto por los siguientes elementos:

- **Creación de vistas independientes de las aplicaciones origen** de tal forma que el conector las utiliza y cada equipo de las aplicaciones legacy, pueda codificar cada vista en función de su modelo de datos. Se abre de esta forma la posibilidad de que cualquier aplicación legacy del mercado utilizando estas vistas pueda crear sus propios scripts que sean utilizados por el resto de los procesos que a continuación se comentan, y cuyo resultado final será la migración de los datos en la aplicación cloud.

- **Proceso de transformación y migración de datos** utilizando la herramienta Spoon perteneciente a Pentaho, que nos permitiría diseñar de una forma gráfica y flexible el flujo de información. No obstante, el mapeo de los campos para su migración se automatizó mediante la herramienta que se comenta a continuación y que construimos ad-hoc para el desarrollo del proyecto.

- **Aplicación para el aprovisionamiento de nuevas empresas y automatización de mapeo de campos.** Desarrollamos una nueva herramienta en este caso en tecnología .Net que nos permitiese automatizar el mapeo de campos entre los modelos de datos de origen y destino, generando los flujos iniciales que se utilizarán desde Pentaho. Esta herramienta acelera el tiempo de desarrollo de estos mapeos, y evita posibles errores en su creación.

Esta última estrategia explicada, cumpliendo con los requisitos inicialmente establecidos, presenta una menor cantidad de riesgos y por ello fue la seleccionada para la transición de los clientes de las aplicaciones legacy al nuevo desarrollo.

# 6. La Importancia de las Herramientas de Análisis, Comunicación y Gestión de Proyectos

En este apartado nos gustaría poner en valor la importancia de cuestiones que en muchos casos se obvian o incluso ocupan un papel secundario en proyectos de desarrollo de esta envergadura.

Por una parte, se encuentra la metodología de gestión de proyectos. Basamos nuestro ciclo de desarrollo en iteraciones (sprints) de Scrum [5] cada 2 semanas que nos iban a permitir realizar entregas contínuas con funcionalidades completas para poder ser revisadas por nuestro equipo interno de analistas, como por parte de un grupo seleccionado de nuestros clientes, que formaron parte de lo que llamamos “early adopters”, y que desde etapas tempranas nos dieron feedback sobre el producto y sugerencias de mejora. Para ello además utilizando la estrategia de datos comentada en el punto anterior pudieron realizar pruebas reales de funcionamiento con sus propios datos ya migrados.

La herramienta utilizada para la gestión del proyecto ha sido **Jira [6]** de Atlassian.

**Jira** es una herramienta de gestión de proyectos y seguimiento de problemas desarrollada por la empresa Atlassian. JIRA ofrece una variedad de funciones para el seguimiento y gestión de proyectos, incluyendo seguimiento de problemas, gestión de tareas, planificación de sprints y administración de versiones.
En este caso la hemos combinado con el marco de trabajo Scrum, lo cual nos proporciona un conjunto de herramientas y flujos de trabajo específicos para implementarlo correctamente dentro del equipo y poder seguir sus principios. De esta forma el equipo ha podido planificar y realizar sprints, mediante la gestión de un “Backlog” y poder hacer seguimiento al progreso de las tareas a través de un tablero Scrum.

Para la gestión documental y actuando como repositorio de la documentación generada hemos utilizado **Confluence [7]**, también de Atlassian.

Es una plataforma de colaboración en equipo desarrollada por Atlassian que permite a los usuarios crear, organizar y compartir contenido de manera eficiente y efectiva.
Una de las características destacadas de Confluence es su capacidad como repositorio. Los usuarios pueden crear y almacenar contenido en Confluence, incluyendo documentos, planes, hojas de cálculo e imágenes. Además, el contenido se puede organizar en espacios y páginas, lo que facilita la búsqueda y recuperación de información.

Confluence también permite la colaboración en tiempo real en el contenido almacenado. Los usuarios pueden agregar comentarios, revisar y aprobar documentos, y trabajar juntos en proyectos de manera eficiente.
Y otra de las herramientas a la que atribuimos gran importancia es **Figma [8]**

Se ha utilizado la herramienta **Figma** para:

- Creación de la librería de componentes.
- Creación del sitemap por parte de los analistas.
- Creación del prototipado “High Fidelity”.

Es una herramienta de diseño colaborativo basada en la nube que permite a los equipos de diseño trabajar juntos en proyectos en tiempo real desde cualquier lugar.

Los usuarios pueden compartir diseños y prototipos con otros miembros del equipo, permitiendo comentarios y revisiones en tiempo real, obteniendo feedback de forma directa.

Ofrece una variedad de funciones para la creación de interfaces de usuario y diseño de experiencias de usuario. Con esta herramienta, se pueden crear diseños de “High Fidelity”, prototipos y especificaciones de diseño en un solo lugar. Además, cuenta con una amplia gama de herramientas de diseño, como formas, símbolos y estilos de texto, para crear diseños personalizados y visualmente atractivos.

Con Figma y Confluence hemos trabajado conjuntamente para enlazar cada uno de los procesos diseñados y especificados en Figma con su correspondiente análisis funcional desarrollado en Confluence.

# 7. Equipo y Flujo de Trabajo

El desarrollo de este proyecto implicaba la participación de muchísima gente en equipos de trabajo diferentes, cuya coordinación interna era importante, pero cuya complejidad crecía de forma considerable cuando hablamos de coordinación entre equipos altamente vinculados por la utilización y desarrollo de tecnología común. Nos referimos a que toda la parte de desarrollo de componentes tanto de la parte front como de la parte back, así como lo relativo a los adaptadores de datos, debía ser común para el desarrollo del resto de funcionalidades de los diversos módulos de las aplicaciones.

Durante todo el desarrollo del proyecto ha sido necesario la realización de guías técnicas que se iban alimentando constantemente de nuevos procedimientos que permitiese plasmar todo el conocimiento que se iba adquiriendo y que permitiese a nuevos desarrolladores el incorporarse lo más rápido a estos equipos de trabajo en caso de ser necesario. Aunque las directrices del equipo de desarrollo eran escribir código que fuese lo más auto-explicativo posible, se ha demostrado la eficacia y necesidad de estas guías de desarrollo para reducir la curva de adaptación y aprendizaje de nuevo personal incorporado en el proyecto.

Utilizamos Azure DevOps, con las herramientas que nos proporciona y la definición de framework que define, para el trabajo de los equipos de desarrollo.

A través de **Git [9]** como sistema de versionado y repositorio de código fuente implementamos el siguiente flujo de trabajo:

• Se crea una rama nueva feature/nuevo-desarrollo a partir de main.
• Cuando el desarrollo esté listo, se podrá desplegar en el entorno de desarrollo.
A través de este pipeline ejecutaremos las pruebas unitarias y de componentes y las verificaciones de calidad del código con SonarQube.

• Una vez verificado que el desarrollo cumple con los criterios de calidad y de aceptación de la funcionalidad, sincronizamos nuestro desarrollo con los cambios de main y podremos desplegar desde la misma rama al entorno de integración.
A través de este pipeline se ejecutarán de nuevo las pruebas unitarias y de componentes y las verificaciones con SonarQube.
Se desplegarán las aplicaciones en el entorno de integración.
Se ejecutarán las pruebas funcionales automatizadas (API, Interfaz)

• Si todas las validaciones y pruebas han sido satisfactorias, el siguiente paso es abrir el Merge Request o Pull Request de nuestra rama feature/nuevo-desarrollo a main.
Se ejecutará el pipeline bajo demanda y se ejecutarán de nuevo las pruebas unitarias y de componentes.

• Se desplegarán las aplicaciones en el entorno de producción.



Los Pull Request por miembros del equipo con las atribuciones necesarias para verificar el código y validar su aceptción.

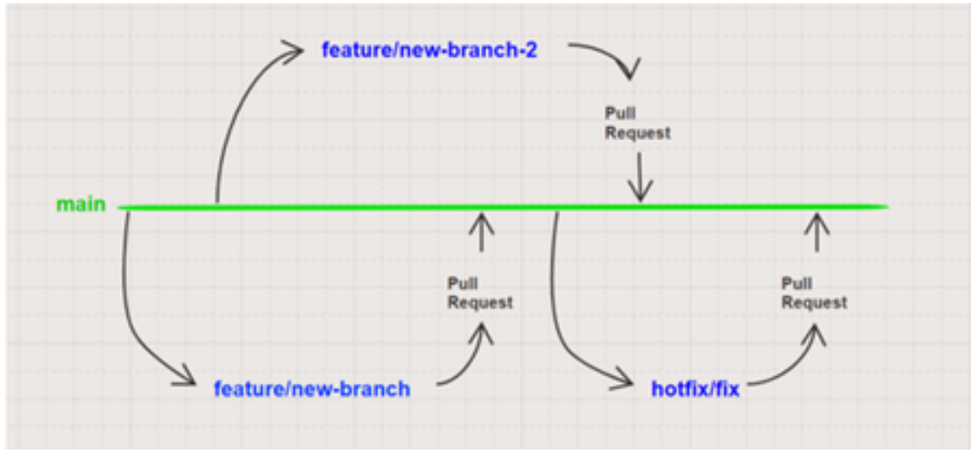



*Diagrama de flujo de ramas*

## Habilitamos 3 entornos:

• Desarrollo: su objetivo es proveer a los desarrolladores de frontend acceso a las APIS durante el desarrollo.
• Integración: entorno estable con una base de datos limpia y controlada. Su objetivo es proveer un entorno estable para poder ejecutar las pruebas automatizadas, realizar las pruebas manuales oportunas y validar los desarrollos con el cliente.
• Producción: Será el entorno final, que se expondrá a internet, con la versión validada y definitiva del desarrollo.

Cada entorno dispondrá de una base de datos en Azure, Elasticsearch [10] y Redis [11]; además de disponer de su propio fichero de configuración por app en el servicio de configuraciones.

Las aplicaciones de backend serán desplegadas en Azure Container App.
En los siguientes puntos iremos profundizando en las diferentes tecnologías específicas del desarrollo software del proyecto.

# 8. Selección de Tecnología del Back

Para el desarrollo del Back evaluamos dos alternativas. Por un lado, nos planteamos seguir una línea continuista, mantener al equipo bajo las tecnologías .NET, y hacer un desarrollo en C#. Por otro lado, totalmente opuesto, planteamos el uso del marco de desarrollo JAVA Spring Boot. Para la elección de una de estas tecnologías evaluamos los siguientes factores:

- Lenguaje de programación
- Experiencia del equipo
- Integración con la nube
- Rendimiento
- Comunidad

## Lenguaje de programación

Tanto .NET con C# como lenguaje de programación, como Spring Framework, ofrecen una suite completa de herramientas y extensiones propias orientadas al desarrollo de este tipo de aplicaciones. Ambas tecnologías disponen de equivalencias entre las distintas herramientas que serían utilizadas en el desarrollo. Cierto es que JAVA, al ser un lenguaje con muchos más años en la palestra, y con una comunidad notablemente mayor, el número de extensiones y componentes disponibles para el desarrollador, hacen que por esta parte la balanza se incline del lado de JAVA.

## Experiencia del equipo

El equipo dentro de la compañía podía agruparse en dos sectores. Aquellos que tenían una larga y consolidada experiencia desarrollando con .NET, y por otro lado, los que habían dedicado toda su carrera a programar con JAVA. No tendríamos problema para conformar el equipo que fuese a liderar el desarrollo, independientemente de la decisión sobre la tecnología. El conocimiento de lo que íbamos hacer, la experiencia de desarrollo de las ERP's legacy, recaía en mayor medida sobre el equipo JAVA, cosa que reforzaba la posible selección de la tecnología final.

## Integración con la nube

Evaluamos las dificultades que nos íbamos a encontrar, a la hora de utilizar cualquier de las dos tecnologías, para utilizar los servicios alojados en la nube. Como era de esperar, ninguna de las dos plataformas exponía ningún impedimento que nos hiciese descartarla. Igual que cuando hablamos de las herramientas disponibles para el desarrollo, la integración de cada plataforma con la nube es prácticamente equivalente entre ambas. La integración es completa por ambas partes.



## Rendimiento

.NET Core tiene un alto rendimiento, tanto si hablamos como de tiempo de ejecución, como hablando del consumo de memoria. Spring Boot no es capaz de equipararse en este aspecto a .NET, ya que siempre va a depender de la máquina virtual de JAVA, y de las limitaciones en este aspecto del propio lenguaje. En este enlace, se muestra en detalle una comparativa de rendimiento de varios aspectos en ambas plataformas. [12]

## Comunidad

Microsoft mantendría la documentación sobre .NET Core completamente actualizada y renovada, y ofrecer constantemente recursos nuevos a la comunidad.

Por su parte, Java cuenta con una de las comunidades de programadores más numerosas del mundo. Es de esta comunidad de donde podemos obtener tutoriales, extensiones y toda la información que podamos necesitar.

Tomando como referencia, por ejemplo, la actividad semanal en StackOverflow **[13]** tanto para C#, como para JAVA y Spring Boot, podemos comprobar que efectivamente son comunidades muy activas. Si bien, podemos apreciar unas métricas más elevadas en JAVA debido a que, como hemos comentado, además de contar con más años que .NET en el mercado, su comunidad es más numerosa. Una imagen similar obtenemos, si nos movemos entre los repositorios públicos alojados en GitHub. Mucho código, muchos proyectos, y en mayor número los escritos en JAVA por los motivos ya comentados.

## Conclusión

A la vista de todo lo anterior, nos encontramos con que la decisión no iba a producirse por ningún factor que nos marcase la tecnología, sino más bien por nuestra posición con respecto a ella.
Finalmente optamos por iniciar el desarrollo con JAVA y Spring Boot, primero por la experiencia y disponibilidad del equipo, y segundo por el respaldo de la comunidad.

# 9. Selección de Tecnología del Front

En lo que respecta a la selección de la tecnología a utilizar para el desarrollo del front, podemos decir que teníamos un favorito desde el inicio, y este era Angular **[14].** Aun así, analizamos también el uso de otras dos tecnologías, como son Primefaces y React.

Con Primefaces **[15]** la velocidad de desarrollo iba a ser notablemente superior, y parte del equipo de desarrollo ya contaba con conocimientos y algún proyecto realizado con esta tecnología. Esta tecnología está fuertemente acoplada al back, y uno de las restricciones del sistema es que el acoplamiento debía evitarse a toda cosa entre cualquier pieza del sistema, con lo que fue descartada desde el principio.

React **[16]**, en cambio, no tenía este problema. Era una tecnología desacoplada del back completamente, con muy buen rendimiento y que ofrecía unas características acordes al objetivo que perseguíamos. Como contrapartida, no disponíamos de perfiles con experiencia, por poca que fuese, en esta tecnología. React, además, establece una curva de aprendizaje superior a Angular, añadía complejidad a la estructura del proyecto, y en cuanto al número de desarrolladores (datos obtenidos de LinkedIn), estaba en desventaja tanto a nivel local, como nacional.

Con todo esto, vimos reforzada la elección de Angular para el desarrollo. Contábamos personal con experiencia, y la incorporación de nuevas personas nos iba a resultar más fácil que con React. Angular nos ofrecía, además, algo que era otro de los requisitos a conseguir con esta evolución. Los usuarios de las ERPs legacy, han adquirido una exigencia muy alta en lo que a usabilidad se refiere. La nueva interfaz debe aproximarse lo máximo posible a lo que están acostumbrados a utilizar, o partiremos con un rechazo por su parte.



# 10.Selección de la Base de Datos (SGBD)

Pese a que nuestra experiencia estaba de nuevo con herramientas Microsoft, y particularmente con SQL Server desde sus primeras versiones, planteamos dar un salto hacia PostgreSQL por varios motivos.

- Permite bases de datos de tamaños más que considerables.
- Rendimiento muy fluido, independientemente de estos tamaños.
- Muy estable
- Escalable
- Admite una concurrencia muy alta. El número de conexiones y peticiones por segundo que es capaz de procesar está entre las más altas.
- Código abierto. Lo que permite, además, un menor coste de licenciamiento para la aplicación, que se refleja de manera lógica en el usuario final.

Aun así, aun habiéndonos decidido por PostgreSQL, el desarrollo y la arquitectura están pensados para que no haya nada que no nos permita movernos hacia otro SGBD con un esfuerzo mínimo. Todas las piezas están diseñadas con total independencia, sin acoplamiento entre ellas, de forma que puedan replicarse con otras tecnologías con un esfuerzo mínimo.

# 11. Selección de la Plataforma Cloud

En esta parte del análisis, enfrentamos esencialmente a dos proveedores: Amazon Web Services, y Azure. Recopilamos los servicios que íbamos a utilizar, y evaluamos lo que nos ofrecía cada uno de ellos. Sabíamos que, para todos los servicios que necesitásemos, íbamos a tener la equivalencia en ambos, y que la decisión quizá se tendría que basar en otros aspectos.

Entre los servicios que necesitábamos, se encontraban los de la siguiente tabla. No aparecen todos los que son, y solo comentamos alguno de ellos. Todos estos servicios, tenían su equivalencia en ambos proveedores.

• Repositorio de código.
En este aspecto, Azure tiene toda la suite de herramientas en Azure DevOps. Este conjunto de herramientas está en una fase muy madura, y además el equipo estaba ya habituado a trabajar con ellas.
• Pipelines.
• Repositorio de imágenes de contenedor.
Elastic Container Registry vs Azure Container Registry.
• Servicio de DNS.
Azure DNS vs AWS Route 53.
• Servicio de seguridad.
• API Manager.
• Clúster para el despliegue de contenedores.
• Balanceadores de carga.
• Bases de datos relacionales..
Ambas permiten desplegar directamente sus bases de datos propietarias, como SQL Server y Aurora, o bien, levantar servidores con las bases de datos más populares.
• Bases de datos no relacionales.
Disponen de servicios de bases de datos propietarias como son CosmosDB y DynamoDB.
• Servicio de almacenamiento.
Azure Blob Storage vs Amazon S3.
• CDN.
• Caché.
• Despliegue de aplicaciones estáticas.
• Redes virtuales.
• Grupos de seguridad.
• Proveedor de identidades.
Los usuarios se identificarán de distintas formas, y una diferencia notable en este aspecto, es que AWS Cognito, en el momento del análisis de este servicio, tenía carencias en cuanto a la interfaz de usuario, con un grado de personalización bajo y sin posibilidad de gestionar traducciones.



• Repositorio de información sensible.
• Servicios de colas.

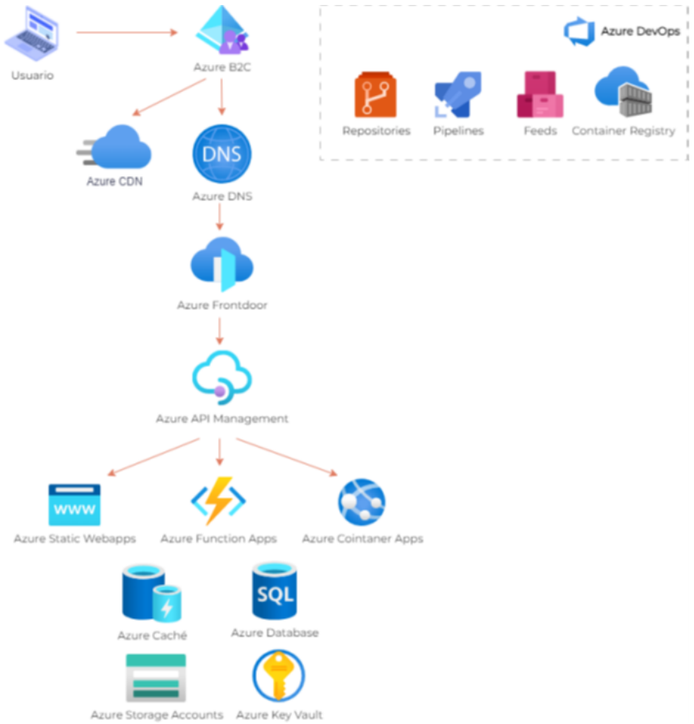
El análisis de costes tampoco nos ayudó a tomar la decisión final. Aunque si bien es cierto que pudiera haber unas diferencias notables en alguno de los servicios, el coste global una vez aprovisionada toda la arquitectura, y el coste proyectado teniendo en cuenta los salto que daría el sistema, según la elasticidad que se le exigirá, estaba muy a la par. No fue un factor decisivo este tampoco.

Finalmente, y como ya imaginamos al principio, la decisión se tomó por la experiencia y conocimiento del equipo en cuanto a las herramientas de Azure, y por el plus de personalización que ofrecía Azure B2C frente a Amazon Cognito.

# 12. Arquitectura

La arquitectura que debía tener el sistema fue apareciendo de forma natural, tras ir enumerando los requerimientos que se deben cumplir en cada uno de los distintos aspectos. Cuando hablas de disponibilidad, cuando hablas de elasticidad, seguridad, almacenamiento, etcétera… van surgiendo una serie de servicios y configuraciones, que una vez conectados entre sí, terminarán mostrando aquello que queremos conseguir.

En nuestro caso, este es el diseño de arquitectura de nuestro sistema.




Este diseño está preparado para dar cabida a lo que serán los tres grandes bloques de la aplicación. Backend, Frontend y DevOps.

## Backend

Seguimos un enfoque orientado a microservicios. Cada uno de los micros, será empaquetado y distribuido en contendores independientes, con una asignación de recursos y configuración de elasticidad diseñada a la medida de cada uno de ellos. Las imágenes de estos contenedores estarán alojadas en Azure Container Registry.

## Frontend

Siguiendo la misma filosofía, usaremos Azure Static WebApp para la parte estática del frontal, apoyado por Azure Content Delivery Network para el cacheo y optimización del servicio de información.

## DevOps

Toda la parte de CI/CD, será gestionada desde Azure DevOps. Desde la gestión de código con los repositorios de código, hasta la gestión y ejecución de Pipelines para el despliegue, estará centralizada en DevOps.
Azure DevOps, también nos proporciona Artifact Feed, que nos permitirá tener centralizada la colección de dependencias del proyecto, de modo que todo el equipo tenga un único origen de las mismas.
Con Azure DevOps automatizaremos los pipelines, que se dispararán algunos de forma manual, otros de forma automática, con las interacciones de los desarrolladores.

- **Pipeline de testeo y compilación.**

Disparado de forma automática con cada Pull Request solicitado por cada desarrollador. Integrará el código nuevo con el existente, ejecutará todos los tests y compilará todos los paquetes. En caso de que alguna de estas acciones no prospere, notificará automáticamente al desarrollador para que revise el problema.

- **Pipeline de despliegue a entornos Develop y Preproducción.**

Una vez que ha pasado el primer Pipeline, el equipo de QA se encarga de revisar el código de dicho Pull Request, para garantizar que cumple con los requisitos esperados. En caso afirmativo, este pipeline actualizará los contenedores y los publicará en los distintos entornos.

- **Pipeline de despliegue a Producción.**

Una vez que el equipo de QA da por bueno en el entorno de Preproducción el funcionamiento del sistema, manualmente lanzará este pipeline, actualizando los contenedores de los entornos de producción, y liberando así la nueva versión a todos los usuarios.

# 13. Herramientas para el Aseguramiento de la Calidad

Pruebas, pruebas y más pruebas. Cada aspecto y cada funcionalidad debe ser probada, a distintos niveles, y persiguiendo distintos objetivos con el fin de que el sistema en general, y cada despliegue particular, cumplan con los requisitos establecidos.

Hay factores que se deben probar, y cuyos resultados se comparan contra respuestas "mockeadas" con datos correctos. Existen otros que nos permiten estresar al sistema y ver cómo se comporta, ver cómo escala y desescala para mantener siempre el rendimiento al 100%, procurando un consumo de recursos ajustado. Otros que aseguran que cada método, cada función de código, cada método responde de forma adecuada... En definitiva, para el desarrollo de este proyecto contamos con herramientas que nos permite probar, y además, muy importante, automatizar el testeo de infinidad de elementos del sistema.

**Pruebas de rendimiento**
El objetivo es evaluar la capacidad del sistema de mantenerse en un rendimiento óptimo, bajo distintos escenarios y cargas de trabajo. Estas pruebas son esenciales para asegurar la calidad de la experiencia del usuario, en cualquier momento y bajo cualquier situación que pueda afectar al sistema. Las herramientas que usamos en este punto son Grafana K6 **[17]** y Apache JMeter **[18].**

Algunas de las métricas que vigilamos son:

• **Tiempo de respuesta.** De cada comunicación, del proceso de actividades completas, etc...
• **Capacidad y escalabilidad.** Nos dice cómo responde el sistema con distintas magnitudes de uso. Básico para conocer si el sistema está bien dimensionado en momentos de poco uso, momentos de uso habitual y periodos de uso extremo.
• **Estabilidad y confiabilidad.** Verifica la estabilidad del sistema durante largos periodos.
• **Uso de recursos.** Informa si algún recurso está suponiendo un cuello de botella, o, por el contrario, si existe algún recurso sobre dimensionado.
• **Tolerancia al estrés y concurrencia.** Estas métricas nos dan información sobre si en los momentos en los que el sistema se enfrenta a periodos de estrés, es capaz de auto escalar, aumentando por sí solo el número de instancias de cada componente, si es capaz de dirigir el tráfico de forma equilibrada entre elementos, y, por último, si ha sido capaz de desescalar de forma adecuada a su estado inicial, destruyendo todos los elementos que no son necesarios, y asegurando que la experiencia del usuario ha sido siempre satisfactoria.

**Pruebas de seguridad**
Estas pruebas ayudan a evaluar la capacidad del sistema de protegerse contra amenazas y vulnerabilidades, garantizando la confidencialidad, integridad y disponibilidad, tanto de los datos como del servicio. Las herramientas que usamos en este punto son OWASP ZAP **[19],** OpenVas **[20]** y Archery Sec **[21]**, y algunas de las métricas son:

• Autenticación y autorización
• Protección de datos. Protección tanto del alojamiento, como de las comunicaciones que se producen en el sistema.
• Ataques típicos al sistema. Prevención de los ataques más típicos, como los de inyección SQL, XSS, CSRF...
• Gestión de vulnerabilidades. Estos sistemas se mantienen actualizados a diario, y chequean el sistema ante todas las nuevas vulnerabilidades que aparecen, de forma que con cada despliegue cuentes con la seguridad de que el sistema está completamente actualizado.

**Pruebas de regresión manual**
Estas pruebas nos permiten comprobar que las funcionalidades ya existentes en el sistema, no se ven comprometidas por la implementación de cambios, actualización del software, etc.... Estas pruebas se hacen de forma manual tras cada implementación por parte del equipo de QA, siguiendo unos casos de uso definidos. Son ellos los que tienen la capacidad de revertir el despliegue si detectan que algo no funciona como debiera. Ni durante el proceso de despliegue, ni durante el proceso de retroceso, el usuario debe percibir ningún deterioro en el rendimiento del sistema.

Las herramientas que utilizamos para la gestión de estas pruebas son **Jira [19]** junto con XRay.

**Pruebas de regresión automáticas**
El objetivo de estas pruebas es, por un lado, comprobar que todo el código nuevo cumple con los requisitos establecidos, y por otro, garantizar que no interfiere con las funcionalidades existentes.
Este tipo de pruebas se realizan de forma automatizada y transparente para el equipo. Los tests de cada funcionalidad se escriben al mismo tiempo que el propio código, y toda funcionalidad debe estar respaldada por su batería de pruebas. Es importante este dato, pues en el momento que las pruebas no cubran el umbral establecido para el global del proyecto, el sistema CI/CD bloqueará la incorporación de cambios para todo el equipo, y mantendrá el sistema en un estado seguro. En caso de que alguna de estas pruebas obtenga un resultado no esperado, el sistema no permitirá la incorporación de este código en el sistema.

Algunos aspectos clave en este tipo de pruebas son:

• Validación de funcionalidades. Se ejecutan pruebas directas, atómicas, sobre cada función de código. Por encima de estas, se ejecutan pruebas que chequean la respuesta de funcionalidades más amplias, y que integran la respuesta de unidades más pequeñas.

• Identificar el origen de los defectos. El sistema nos marcará el punto en el que el sistema se ha comportado de forma no esperada, indicando, además, el resultado ofrecido y el esperado en cada momento.

• Cobertura de código. Como comentábamos antes, este indicador actúa de nota de corte. El código debe tener pruebas que lo validen prácticamente por completo, no permitiendo que las pruebas cubran prácticamente el código completo del sistema.

El coste de mantener esta batería de pruebas actualizada, y asegurando una cobertura prácticamente total, es alto. Tanto, que suponer una valoración adicional al coste que supondrá cada desarrollo. Pero este coste no es comparable al coste que supondría, primero, no hacer estas pruebas, y segundo, tener que hacerlas de forma manual. Obviamente, en este tipo de proyectos, ni la primera ni la segunda son opciones viables. Se busca la excelencia, la seguridad, la tranquilidad del usuario sabiendo que uno de los patrimonios de su empresa, la información, está completamente asegurada. Se busca que la experiencia del usuario se siempre la mejor, hasta el punto de que el usuario se olvide que está usando una herramienta, y pueda centrarse en la gestión de su información.

# 14. Conclusión

El proceso de construcción de una nueva aplicación en tecnología, arquitectura y plataforma moderna, a partir de uno o varios monolitos como era nuestro caso, es un proceso altamente complejo y con un riego inherente que no debemos descuidar.

La planificación inicial de todos los factores posibles a tener en cuenta constituye la base del éxito, y en ellos nos hemos centrado en este artículo.

Hemos querido no sólo centrarnos en cuestiones técnicas relativas al desarrollo como bajo nuestro punto se suelen enmarcar este tipo de publicaciones, sino también en aspectos organizacionales, uso de herramientas y coordinación de los equipos de trabajo, fundamentales para abordar proyectos de esta complejidad y envergadura.



[1] https://www.researchgate.net/publication/221527675_Development_of_a_Parametric_Estimating_Model_for_Technology-Driven_Deployment_Projects

[2] https://kafka.apache.org/

[3] https://www.hitachivantara.com/en-us/products/pentaho-plus-platform/data-integration-analytics

[4] https://keepcoding.io/blog/que-es-pentaho-data-integration/

[5]https://www.researchgate.net/publication/258023481_SCRUM_and_Productivity_in_Software_Projects_A_Systematic_Literature_Review

[6] https://www.atlassian.com/es/software/jira

[7] https://www.atlassian.com/es/software/confluence

[8] https://www.figma.com/

[9] https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git

[10] https://www.elastic.co/es/elasticsearch

[11] https://redis.io/

[12] https://iopscience.iop.org/article/10.1088/1742-6596/1933/1/012041

[13] https://stackoverflow.com/tags

[14] https://angular.io/

[15] https://www.primefaces.org/

[16] https://react.dev/

[17] https://k6.io/

[18] https://jmeter.apache.org/

[19] https://www.zaproxy.org/

[20] https://www.openvas.org/

[21] https://www.archerysec.com/

clave i

Software solutions for business