Trabajo Fin de Grado
en Ingeniería de las Tecnologías de Telecomunicación

GastroMatch: Sistema de recomendación personalizada de restaurantes basado en microservicios desplegados en Microsoft Azure con inteligencia artificial y ML.NET

Autor: Jesús Guerrero Martín

Tutor: María Teresa Ariza Gómez

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2026

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

GastroMatch: Sistema de recomendación personalizada de restaurantes basado en microservicios desplegados en Microsoft Azure con inteligencia artificial y ML.NET

Autor:

Jesús Guerrero Martín

Tutor:

María Teresa Ariza Gómez

Profesor titular

Dpto. de Ingeniería Telemática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2026

Trabajo fin de grado: GastroMatch: Sistema de recomendación personalizada de restaurantes basado en microservicios desplegados en Microsoft Azure con inteligencia artificial y ML.NET

Autor: Jesús Guerrero Martín

Tutor: María Teresa Ariza Gómez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2026

El Secretario del Tribunal

Agradecimientos

Después de cuatro años, no pensaba que este día llegaría, por lo que quiero agradecerme a mí mismo haber retomado este trabajo y, con mucho esfuerzo, haber conseguido terminarlo, poniendo así fin a mi etapa universitaria.

Quiero agradecer también a mis padres, que siempre insistieron en que lo retomara y gracias a quienes he podido encontrar la motivación necesaria para hacerlo.

Por último, agradecer a mi tutora, **María Teresa**, por estar siempre atenta y hacerme más fácil el camino hasta la entrega final de este proyecto.

Jesús Guerrero Martín

Sevilla, 2026

Resumen

El presente Trabajo Fin de Grado describe el diseño e implementación de *GastroMatch*, una plataforma cloud destinada a ofrecer recomendaciones personalizadas de restaurantes mediante técnicas de inteligencia artificial. El sistema se ha desarrollado siguiendo una arquitectura de microservicios desplegada en Microsoft Azure e implementada íntegramente con tecnología .NET.

El núcleo funcional del proyecto es un sistema híbrido de recomendación que combina dos enfoques complementarios: el filtrado colaborativo, basado en el análisis de patrones de interacción entre usuarios, y el filtrado basado en contenido, que utiliza las características de los restaurantes para identificar elementos similares a los que el usuario ha valorado positivamente. Para el filtrado colaborativo se emplea un modelo de factorización de matrices desarrollado con ML.NET, mientras que para el filtrado basado en contenido se aplican técnicas de vectorización y medición de distancias entre atributos.

Con el fin de garantizar la escalabilidad, modularidad y mantenibilidad del sistema, la arquitectura se ha estructurado en microservicios independientes que gestionan usuarios, restaurantes, interacciones y recomendaciones. La solución se complementa con un API Gateway, un entorno de red privada en la nube, una base de datos gestionada en Azure SQL y un sistema de despliegue continuo mediante Azure DevOps.

El resultado es una plataforma funcional, extensible y preparada para su evolución futura, capaz de generar recomendaciones personalizadas en tiempo real y de adaptarse al crecimiento del número de usuarios y restaurantes. El trabajo recoge tanto los fundamentos teóricos de los modelos utilizados como su aplicación práctica en un entorno de microservicios, así como los procesos de diseño, implementación y validación realizados durante el desarrollo.

Abstract

This Final Year Project presents the design and implementation of *GastroMatch*, a cloud-based platform aimed at providing personalized restaurant recommendations through artificial intelligence techniques. The system has been developed following a microservices architecture deployed on Microsoft Azure and implemented entirely using .NET technologies.

The core functionality of the project lies in a hybrid recommendation system that combines two complementary approaches: collaborative filtering, based on analyzing interaction patterns between users, and content-based filtering, which leverages restaurant attributes to identify items similar to those previously liked by the user. Collaborative filtering is implemented through a matrix factorization model built with ML.NET, while content-based filtering applies vectorization techniques and distance metrics to compare restaurant features.

To ensure scalability, modularity, and maintainability, the system is structured into independent microservices responsible for managing users, restaurants, interactions, and recommendations. The solution is complemented by an API Gateway, a private cloud network environment, an Azure SQL managed database, and a continuous deployment pipeline powered by Azure DevOps.

The result is a functional and extensible platform, capable of generating real-time personalized recommendations and adapting to increasing numbers of users and restaurants. This work covers both the theoretical foundations of the employed models and their practical implementation in a microservices environment, as well as the design, development, and validation processes carried out throughout the project.

Índice

Índice de Figuras **15**
1 Introducción **17**
1.1 Motivación 17
1.2 Objetivos 17
1.3 Antecedentes 18
1.4 Descripción de la solución 18
1.4.1 Funcionalidades 19
1.4.2 Arquitectura del Sistema 19
1.5 Estructura de la memoria 21
2 Recursos utilizados **22**
2.1 Recursos Hardware 22
2.1.1 Ordenador Lenovo 22
2.2 Recursos Software 22
2.2.1 Visual Studio 2022 22
2.2.2 Postman 23
2.2.3 Docker Desktop 23
2.2.4 Azure 23
2.2.5 Mermaid 24
3 Tecnologías utilizadas **25**
3.1 .Net 8 25
3.2 ML.NET 25
3.3 YARP (Yet Another Reverse Proxy) 26
3.4 RabbitMQ 26
3.5 Entity Framework Core 27
3.6 Azure Devops 27
3.7 Azure SQL Database 28
3.8 Cloudinary 28
3.9 GIT 29
4 Sistema de Recomendaciones **31**
4.1 Fundamentos del sistema de recomendación 31
4.2 Filtrado Colaborativo 32
4.2.1 Principio teórico 32
4.2.2 Implementación en ML.NET 32
4.3 Filtrado basado en contenido 33
4.3.1 Principio teórico 33
4.3.2 Implementación en ML.NET 34

4.4. Integración Híbrida de los Modelos 34
5 Implementación en .Net 36
5.1 Arquitectura del sistema y microservicios que lo conforman 36
5.1.1 API Gateway 37
5.1.2 Auth API 37
5.1.3 Users API 37
5.1.4 Restaurants API 37
5.1.5 Likes API 38
5.1.6 Recommendation API 38
5.2 Características comunes de las APIs 38
5.2.1 Arquitectura interna 39
5.2.2 Convenciones de desarrollo y respuesta 39
5.2.3 Seguridad 39
5.2.4 Documentación y pruebas 40
5.3 Comunicación entre microservicios 40
5.3.1 Comunicación síncrona (HTTP) 40
5.3.2 Comunicación asíncrona (RabbitMQ) 40
5.4 Implementación de las APIs 42
5.4.1 API Gateway 42
5.4.2 Auth API 45
5.4.3 Users API 48
5.4.4 Restaurants API 49
5.4.5 Likes API 52
5.4.6 Recommendations API 54
6 Infraestructura del sistema 60
6.1 Arquitectura general del despliegue cloud 60
6.1.1 Azure DevOps – Gestión del ciclo de vida del código 60
6.1.2 Azure Container Apps – Ejecución de microservicios 60
6.1.3 Virtual Network (VNet) – Comunicación privada y seguridad 61
6.1.4 Azure Container Registry – Almacenamiento y distribución de imágenes 61
6.1.5 Azure SQL Database – Persistencia independiente por dominio 61
Esquema general del entorno cloud 61
6.2 Azure DevOps y pipeline de CI/CD 62
6.2.1 Azure DevOps como plataforma de automatización 62
6.2.2 Pipeline de despliegue de las APIs 63
6.2.3 Pipeline de publicación de librerías NuGet 64
Esquema del flujo de despliegue 64
6.3 Azure Container Apps como plataforma de ejecución 64
6.3.1 Modelo de ejecución basado en contenedores 65
6.3.2 Escalado y versionado de los servicios 65
6.3.3 Configuración y aislamiento entre microservicios 65
6.4 Azure Container Registry 66
6.5 Virtual Network (VNet) y comunicaciones privadas 67
6.5.1 Diseño de la red virtual 67
6.5.2 Aislamiento de los microservicios 68
6.5.3 Acceso privado a Azure SQL Database 68

6.5.4 Integración con Azure Container Registry 69
Justificación del diseño de red 69
6.6 Azure SQL Database – Persistencia de datos 69
6.6.1. Modelo de persistencia por microservicio 69
6.6.2 Integración con la red privada 69
6.6.3 Justificación del uso de Azure SQL Database 70
7 Conclusiones y Futuras mejoras 71
7.1 Conclusiones del trabajo realizado 71
7.2 Evaluación de los objetivos planteados 72
7.3 Limitaciones del sistema 72
7.4 Líneas de trabajo futuro 73
7.5 Valoración personal del proyecto 74
8 Anexos 76
Anexo 1: Librería GastromatchML 76
A.1.1 Motivación y objetivos de la librería 76
A.1.2 Estructura del proyecto GastromatchML 76
A.1.3 Entrenamiento del modelo de Filtrado Colaborativo 77
A.1.4 Motor de Filtrado Colaborativo 78
A.1.5 Motor de Filtrado Basado en Contenido 80
A.1.6 Modelos de datos utilizados 82
A.1.7 Construcción de los conjuntos de entrenamiento 82
A.1.8 Integración con Recommendations API 83
A.1.9 Justificación del diseño 83
Anexo 2: Librería Gastromatch.RabbitMQ 84
A.2.1 Motivación y objetivos de la librería 84
A.2.2 Estructura del proyecto Gastromatch.RabbitMQ 84
A.2.3 Configuración y gestión de la conexión 84
A.2.4 Publicación de eventos 85
A.2.5 Consumo de eventos mediante BackgroundService 86
A.2.6 Integración con los microservicios de GastroMatch 89
A.2.7 Flujo de eventos asíncronos en la arquitectura de GastroMatch 90
A.2.8 Justificación del diseño 91
Anexo 3: Gestión y despliegue de librerías NuGet personalizadas 92
A.3.1 Objetivo del anexo 92
A.3.2 Librerías NuGet utilizadas en el proyecto 92
A.3.3 Proceso de empaquetado y publicación de las librerías NuGet 93
A.3.4 Uso de Azure Artifacts como feed privado de NuGet 94
Referencias 96

ÍNDICE DE FIGURAS

Ilustración 1: Esquema de la arquitectura del Sistema. 21
Ilustración 2: Logo Visual Studio 2022. 23
Ilustración 3: Logo Postman. 23
Ilustración 4: Logo Docker. 23
Ilustración 5: Logo Azure. 23
Ilustración 6: Logo Git. 24
Ilustración 7: Logo .Net 8. 25
Ilustración 8: Logo ML.NET. 26
Ilustración 9: Logo YARP. 26
Ilustración 10: Logo RabbitMQ. 27
Ilustración 11: Entity Framework Core. 27
Ilustración 12: Azure Devops. 28
Ilustración 13: Logo Azure SQL Database. 28
Ilustración 14: Logo Cloudinary. 28
Ilustración 15: Logo Git. 29
Ilustración 16: Esquema general de la arquitectura del Sistema. 37
Ilustración 17: Comunicación entre los servicios del sistema 42
Ilustración 18: Configuración YARP. 44
Ilustración 19: middleware propagación cabecera X-USERID. 45
Ilustración 20: Flujo completo de registro de un usuario. 47
Ilustración 21: Flujo completo de login. 47
Ilustración 22: Flujo subida de imagen de un restaurante. 51
Ilustración 24: Flujo completo de generación de recomendaciones. 58
Ilustración 25: Esquema general del entorno cloud 62
Ilustración 26: Pipeline para despliegue de las APIs. 64
Ilustración 27: Esquema del flujo de despliegue. 64
Ilustración 28: Panel de Azure Container Apps mostrando los microservicios de GastroMatch desplegados y en ejecución en el entorno cloud. 66
Ilustración 29: Topología de red completa. 68
Ilustración 30: Configuración y entrenamiento del modelo de Filtrado Colaborativo. 77
Ilustración 31: Implementación del modelo de Filtrado Colaborativo. 79
Ilustración 32: Implementación del motor de Filtrado Basado en Contenido. 81
Ilustración 33: Clase de extensión GastromatchMLExtensions. 83
Ilustración 34: Clase RabbitPublisher. 86
Ilustración 35: Inicialización del consumidor en el método ExecuteAsync. 87
Ilustración 36: Procesamiento de mensajes en el método OnReceivedAsync. 88
Ilustración 37: Ejemplo de inyección de un handler. 89
Ilustración 38: Ejemplo de implementación de un handler. 89
Ilustración 39: Clase de extensión GastromatchMLExtensions. 90

Ilustración 40: Diagrama publicador/subscriptor en Gastromatch. 91
Ilustración 41: Pipeline para publicación de NuGet propio. 94

1 INTRODUCCIÓN

La experiencia es el nombre que damos a nuestros errores.

- Oscar Wilde -

1.1 Motivación

En la actualidad, la mayoría de las decisiones sobre dónde comer o qué restaurante visitar se toman a partir de reseñas y valoraciones en plataformas digitales. Sin embargo, este modelo presenta cada vez más limitaciones: la proliferación de **reseñas falsas**, tanto positivas como negativas, y la influencia de determinados **creadores de contenido o campañas publicitarias encubiertas** han generado un entorno cada vez menos fiable para los usuarios que buscan opiniones auténticas.

A partir de esta problemática surge **GastroMatch**, una plataforma cuyo objetivo es ofrecer una alternativa más **transparente y personalizada** para descubrir nuevos restaurantes. En lugar de basarse en opiniones públicas manipulables, el sistema utiliza los **likes reales de los usuarios** como fuente de datos principal para alimentar un modelo de inteligencia artificial que aprende de sus gustos y comportamientos, generando recomendaciones objetivas y ajustadas a cada perfil.

La motivación principal de este trabajo radica en **diseñar e implementar una arquitectura moderna, escalable y mantenible**, que permita dar soporte a este enfoque más honesto de la recomendación gastronómica. Para ello, se ha optado por una solución basada en **microservicios**, desarrollada con **.NET 8**, **RabbitMQ** para la comunicación asíncrona entre servicios y **Azure Container Apps** para su despliegue en la nube.

Además, este proyecto supone una oportunidad personal para integrar diferentes áreas del desarrollo de software —arquitectura distribuida, inteligencia artificial y computación en la nube— en un caso práctico que busca no solo resolver un problema técnico, sino también **mejorar la confianza y la experiencia de los usuarios en el entorno digital gastronómico**.

1.2 Objetivos

El objetivo principal de este trabajo es **diseñar e implementar una plataforma de recomendación gastronómica inteligente**, basada en una arquitectura de **microservicios**, que permita ofrecer sugerencias personalizadas de restaurantes a los usuarios a partir de sus *likes* dentro de la aplicación.

Para lograrlo, se han planteado los siguientes objetivos específicos:

- **Diseñar e implementar una arquitectura modular y escalable**, que permita el desarrollo independiente de cada servicio y facilite su mantenimiento a largo plazo.

- **Desarrollar una API de recomendaciones** que combine diferentes enfoques de filtrado para generar sugerencias personalizadas.

- **Garantizar una comunicación eficiente y desacoplada** entre los distintos servicios del sistema mediante mensajería asíncrona.

- **Asegurar la calidad del desarrollo**, aplicando buenas prácticas de diseño, control de versiones y pruebas de funcionamiento.

- **Automatizar el proceso de integración y despliegue** de los servicios, favoreciendo la entrega continua en entornos en la nube.

1.3 Antecedentes

Previo al desarrollo de la plataforma, se llevó a cabo un análisis de las principales herramientas de recomendación y descubrimiento de restaurantes disponibles en la actualidad. En la mayoría de los casos, estas soluciones se basan en sistemas de reseñas y puntuaciones públicas que, si bien resultan útiles en sus inicios, se han visto afectados por la **falta de veracidad** en muchas valoraciones y por la **influencia de estrategias de marketing o colaboraciones pagadas**.

Esta situación ha provocado que los usuarios desconfíen progresivamente de las plataformas tradicionales, lo que motivó la búsqueda de un **nuevo enfoque centrado en la experiencia real del usuario**. A partir de esta idea inicial, GastroMatch comenzó como un **prototipo monolítico**, que integraba todas las funcionalidades —autenticación, gestión de usuarios, restaurantes, likes y recomendaciones— dentro de una única API.

Sin embargo, a medida que el proyecto crecía y se incorporaban nuevas funcionalidades, se detectaron las limitaciones de este enfoque. La dificultad para escalar, mantener y desplegar de forma independiente cada módulo llevó a adoptar una **arquitectura de microservicios**, donde cada parte del sistema puede evolucionar y escalar de forma autónoma.

1.4 Descripción de la solución

La solución desarrollada consiste en una **plataforma de recomendación gastronómica** compuesta por varios servicios independientes que trabajan de forma coordinada para ofrecer al usuario una experiencia personalizada y fiable.

El sistema se basa en una arquitectura de **microservicios desplegados en la nube**, donde cada API cumple una función específica dentro del flujo general:

- **Auth API**: gestiona la autenticación y emisión de tokens para el acceso seguro al sistema.
- **User API**: almacena y gestiona la información de los usuarios registrados.
- **Restaurant API**: se encarga de la gestión de los restaurantes y sus atributos (categoría, rango de precio, etc.).
- **Like API**: registra los *likes* de los usuarios sobre los restaurantes, que son la base de los datos utilizados por el sistema de recomendaciones.
- **Recommendation API**: implementa la lógica de inteligencia artificial para generar recomendaciones personalizadas combinando filtrado colaborativo y basado en contenido.

- **API Gateway (YARP)**: actúa como punto de entrada único para las peticiones externas, gestionando la autenticación y el enrutamiento hacia los distintos servicios.

Todos los servicios se despliegan en **Azure Container Apps**, con integración y entrega continua configuradas mediante **Azure DevOps**, lo que permite un desarrollo ágil y una rápida evolución del sistema.

1.4.1 Funcionalidades

La plataforma **GastroMatch** ofrece un conjunto de funcionalidades orientadas tanto a los usuarios finales como al funcionamiento interno del sistema, con el objetivo de ofrecer una experiencia personalizada dentro del ecosistema gastronómico digital.

Funcionalidades para el usuario

- **Registro y autenticación**: permite a los usuarios crear una cuenta y acceder de forma segura al sistema. La autenticación se realiza mediante *tokens JWT*, garantizando la protección de las operaciones dentro de la plataforma.
- **Gestión del perfil de usuario**: cada usuario dispone de un perfil personal donde puede consultar sus datos básicos y la lista de restaurantes a los que ha dado *like*.
- **Exploración de restaurantes**: el sistema ofrece un catálogo de restaurantes que pueden filtrarse por criterios como tipo de cocina y rango de precio.
- **Interacción con los restaurantes**: los usuarios pueden indicar su interés mediante *likes*, que son la base de las recomendaciones personalizadas.
- **Recomendaciones personalizadas**: la aplicación genera un conjunto de restaurantes recomendados ordenados por afinidad, combinando filtrado colaborativo y basado en contenido.

Funcionalidades internas del sistema

- **Gestión independiente de servicios**: cada área funcional del sistema (usuarios, restaurantes, likes, recomendaciones y autenticación) está implementada como un microservicio independiente, lo que facilita su mantenimiento y escalabilidad.
- **Comunicación basada en eventos**: mediante **RabbitMQ**, los microservicios intercambian información de manera asíncrona. Por ejemplo, cuando un nuevo usuario se registra, el evento **UserRegistered** es publicado por la **Auth API** y consumido por la **User API**, que se encarga de crear el usuario correspondiente en su propio dominio.
- **Generación dinámica de recomendaciones**: la **Recommendation API** combina dos enfoques de inteligencia artificial —*Collaborative Filtering* y *Content-Based Filtering*— para ofrecer resultados que se adaptan a las preferencias del usuario.
- **Centralización de peticiones mediante API Gateway**: el sistema cuenta con un **API Gateway** que actúa como punto de entrada único y se encarga de enrutar las peticiones hacia los distintos servicios, además de propagar la identidad del usuario autenticado.
- **Despliegue automatizado en la nube**: el sistema cuenta con un proceso de integración y entrega continua que permite actualizaciones rápidas y seguras de los servicios.

1.4.2 Arquitectura del Sistema

La arquitectura de **GastroMatch** se ha diseñado siguiendo un enfoque basado en **microservicios**, con el objetivo de construir una plataforma modular, escalable y fácilmente mantenible. Cada servicio representa un dominio independiente dentro del sistema y puede desplegarse, actualizarse o escalarse de forma autónoma.

El acceso de los usuarios se realiza a través de un **API Gateway desarrollado con YARP (Yet Another Reverse Proxy)**, que actúa como punto de entrada único al sistema. Este componente enruta las peticiones hacia los distintos microservicios desplegados, además de encargarse de la autenticación y la propagación del identificador de usuario en las cabeceras de las peticiones.

Cada microservicio se ejecuta dentro de su propio **contenedor**, lo que permite aislar su entorno de ejecución y gestionar sus dependencias de manera independiente. Esta estructura facilita el despliegue, la escalabilidad y el mantenimiento del sistema, además de mejorar su estabilidad al evitar interferencias entre servicios.

A continuación, se describe el papel de cada componente dentro de la arquitectura:

- **Auth API**: gestiona el registro y autenticación de los usuarios. Tras un registro exitoso, publica un evento `user.registered` en **RabbitMQ** para notificar al resto de servicios.
- **RabbitMQ**: actúa como sistema de mensajería y permite la comunicación asíncrona entre microservicios. Los eventos publicados se distribuyen sin acoplar directamente a los emisores y consumidores, lo que mejora la extensibilidad y la tolerancia a fallos.
- **Users API**: consume el evento `user.registered` publicado por la Auth API y crea el usuario correspondiente dentro de su propio dominio, manteniendo una base de datos independiente.
- **Likes API**: gestiona las acciones de *like* realizadas por los usuarios sobre los restaurantes, que sirven como base de datos de preferencias para el sistema de recomendaciones.
- **Restaurants API**: administra la información de los restaurantes, incluyendo su tipo de cocina, ciudad y rango de precio.
- **Recommendations API**: genera las recomendaciones personalizadas combinando los enfoques de **filtrado colaborativo** y **basado en contenido**, en función de los datos obtenidos de los otros servicios.
- **Bases de datos independientes**: cada servicio dispone de su propia base de datos (DB Auth, DB Users, DB Likes, DB Restaurants y DB Recs), garantizando el **principio de independencia de datos** y la **separación de responsabilidades** entre dominios.

A continuación, se muestra el esquema general de la arquitectura implementada, donde puede observarse la interacción entre los distintos microservicios y sus respectivas bases de datos:

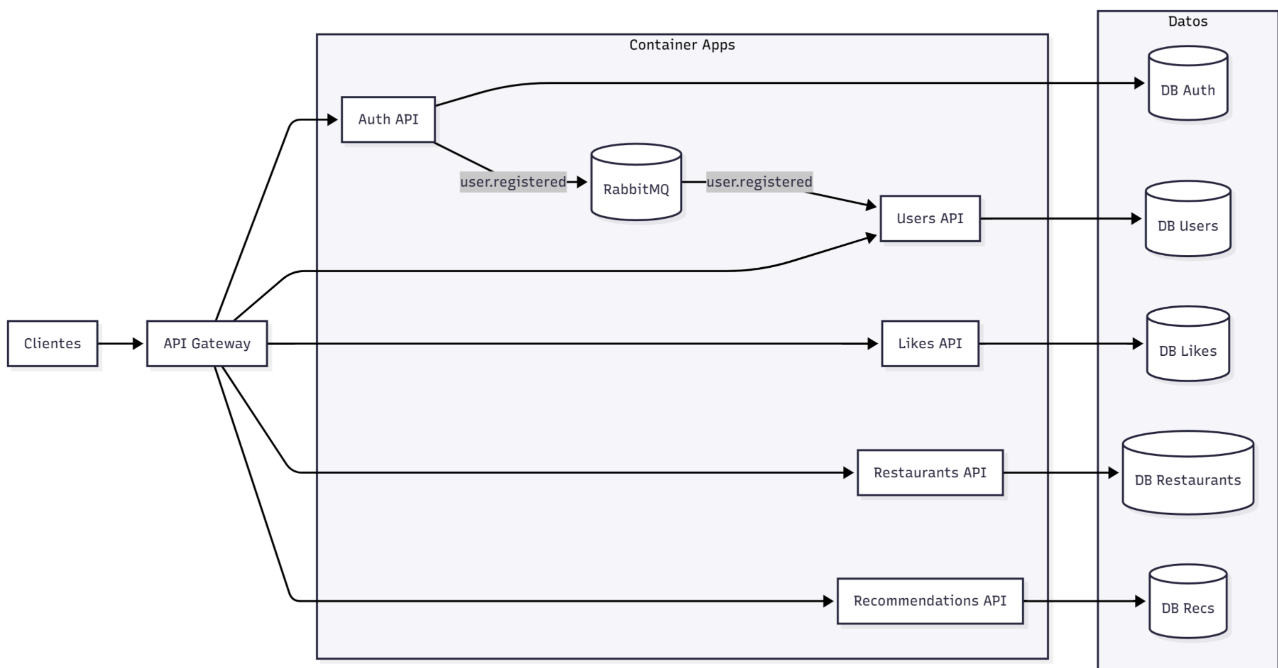

Ilustración 1: Esquema de la arquitectura del Sistema.

1.5 Estructura de la memoria

1. **Introducción**: recoge la motivación, antecedentes, objetivos, funcionalidades y arquitectura del

sistema, presentando el contexto general del proyecto y su justificación técnica.

2. **Recursos utilizados**: Detalla los recursos hardware y software empleados durante el desarrollo, incluyendo el entorno local de trabajo y los servicios en la nube utilizados para el despliegue.
3. **Tecnologías utilizadas**: Describe las tecnologías y servicios clave empleados en el proyecto, justificando su selección según las necesidades técnicas de la plataforma.
4. **Sistema de recomendaciones:** Explica los fundamentos teóricos del sistema de recomendación implementado, diferenciando los enfoques de **filtrado colaborativo** y **filtrado basado en contenido**, y cómo ambos se combinan para generar resultados personalizados.
5. **Implementación del sistema en .NET**: Describe el desarrollo de los distintos microservicios que componen la solución, la estructura de sus capas, la comunicación entre ellos y el uso de **RabbitMQ** como sistema de mensajería. También incluye la implementación práctica del sistema de recomendaciones con **ML.NET**.
6. **Azure e infraestructura del sistema**: Expone la configuración del entorno cloud en **Microsoft Azure**, detallando el despliegue de los microservicios en **Azure Container Apps**, la red virtual (VNet), la gestión de bases de datos mediante **Azure SQL Database** y la automatización del despliegue continuo con **Azure DevOps**.
7. **Conclusiones y Mejores futuras:** Este apartado analiza los resultados obtenidos, los conocimientos adquiridos y las oportunidades de mejora identificadas a lo largo del desarrollo del proyecto. También recoge las posibles líneas de evolución del proyecto, entre ellas la incorporación del evento `like.created` para el recálculo dinámico de recomendaciones, la ampliación del sistema de observabilidad con *logs* y *health checks* y el desarrollo de un frontend visual.
8. **Anexos**
   - **A. Librería GastromatchML:** Describe la librería NuGet desarrollada para encapsular la lógica de Machine Learning del sistema, incluyendo los motores de Filtrado Colaborativo y Filtrado Basado en Contenido, así como su integración con la Recommendations API.
   - **B. Librería Gastromatch.RabbitMQ:** Presenta la librería NuGet creada para estandarizar la comunicación asíncrona mediante RabbitMQ, detallando el modelo de publicación y consumo de eventos y su uso desacoplado en los microservicios.
   - **C. Gestión y despliegue de librerías NuGet personalizadas:** Explica el proceso de empaquetado, versionado y despliegue de las librerías internas de GastroMatch mediante un feed privado en Azure Artifacts.

2 RECURSOS UTILIZADOS

No es que tengamos poco tiempo, sino que perdemos mucho.

- Séneca -

En este capítulo se van a detallar cada uno de los recursos, tanto hardware como software, que han sido utilizados para la realización y desarrollo del proyecto.

2.1 Recursos Hardware

2.1.1 Ordenador Lenovo

Se ha empleado un ordenador portátil para implementar el desarrollo de la solución con las siguientes características:

- Sistema Operativo: Microsoft Windows 11
- Procesador: AMD Ryzen 7 4700U 2,00GHz
- Memoria RAM de 8GB
- Almacenamiento de 512 GB SSD

2.2 Recursos Software

2.2.1 Visual Studio 2022

Entorno de desarrollo integrado (IDE) empleado para la creación de todos los microservicios.

Permite desarrollar aplicaciones compatibles con la plataforma **.NET**, compilar contenedores Docker, y gestionar el control de versiones mediante **Git** integrado.

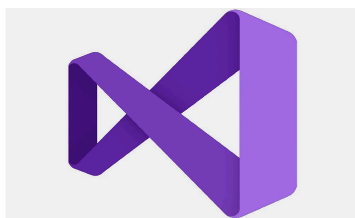

Ilustración 2: Logo Visual Studio 2022.

2.2.2 Postman

Aplicación empleada para realizar peticiones HTTP a los distintos endpoints de las APIs desarrolladas.

Se ha utilizado tanto en las fases de desarrollo como en las de validación para comprobar el correcto funcionamiento de los servicios y el intercambio de información entre microservicios.

Ilustración 3: Logo Postman.

2.2.3 Docker Desktop

Herramienta utilizada para la **contenedorización de los servicios**.

Permite ejecutar los microservicios de forma aislada dentro de contenedores, simulando el entorno de producción desde el equipo local.

Ilustración 4: Logo Docker.

2.2.4 Azure

Plataforma en la nube de Microsoft utilizada como **entorno principal de despliegue y gestión de los servicios** de GastroMatch.

Azure ha permitido orquestar los distintos componentes del sistema, proporcionando infraestructura, almacenamiento, bases de datos y herramientas de automatización que facilitan el desarrollo y la puesta en producción del proyecto.

Ilustración 5: Logo Azure.

Dentro de esta plataforma se han empleado distintos servicios, entre los que destacan:

- **Azure Container Apps**, para el despliegue de los microservicios en contenedores, garantizando la escalabilidad, el aislamiento y la disponibilidad de cada API.

- **Azure SQL Database**, como solución de base de datos relacional en la nube, que proporciona almacenamiento independiente para cada servicio del sistema (usuarios, restaurantes, likes, recomendaciones y autenticación).

- **Azure DevOps**, que se ha utilizado para la gestión del código fuente mediante **Azure Repos** y para la automatización de despliegue a través de **pipelines CI/CD**, permitiendo la compilación, prueba y publicación automática de los contenedores.

2.2.5 Mermaid

Herramienta de diagramación basada en texto que permite crear **diagramas técnicos y de arquitectura** de forma rápida y legible.

Se ha utilizado para generar los esquemas incluidos en la memoria.

Ilustración 6: Logo Git.

3 Tecnologías utilizadas

Ningún gran logro se consigue sin esfuerzo.

- Epicteto -

En este capítulo van a ser introducidas las tecnologías utilizadas durante la realización del proyecto.

3.1 .Net 8

Framework principal utilizado para el desarrollo de los microservicios del sistema.

La plataforma **.NET 8** proporciona un entorno moderno, de alto rendimiento y multiplataforma, que permite crear APIs robustas y escalables utilizando el lenguaje **C#**.

Ilustración 7: Logo .Net 8.

Entre sus ventajas destacan el soporte nativo para contenedores Docker, su compatibilidad con herramientas de inteligencia artificial como **ML.NET**, y la integración con **Entity Framework Core** para el manejo de bases de datos relacionales.

La elección de esta tecnología ha permitido mantener una arquitectura limpia y orientada a dominios, acorde con los principios de **Domain-Driven Design (DDD)** adoptados en el proyecto.

3.2 ML.NET

Biblioteca de aprendizaje automático integrada en el ecosistema .NET, utilizada para el desarrollo del sistema de **recomendaciones inteligentes** de GastroMatch.

ML.NET permite crear, entrenar y consumir modelos de machine learning directamente en C#, sin necesidad de dependencias externas.

En este proyecto se ha empleado para implementar un **modelo híbrido** que combina **filtrado colaborativo** y **filtrado basado en contenido**, ofreciendo recomendaciones personalizadas según los gustos del usuario y las características de los restaurantes.

Ilustración 8: Logo ML.NET.

3.3 YARP (Yet Another Reverse Proxy)

Framework de código abierto desarrollado por Microsoft, utilizado para implementar el **API Gateway** del sistema.

Ilustración 9: Logo YARP.

Este componente actúa como punto de entrada único a todos los microservicios, gestionando el enrutamiento de las peticiones, la autenticación y la propagación de cabeceras de usuario entre servicios.
Su elección se debe a su gran flexibilidad y facilidad de integración con entornos .NET, lo que ha permitido construir un gateway ligero, configurable y totalmente adaptado a la arquitectura del proyecto.

3.4 RabbitMQ

Sistema de mensajería basado en el protocolo AMQP, empleado para la comunicación asíncrona entre los microservicios.

Gracias a RabbitMQ, los servicios pueden intercambiar información mediante **eventos** sin depender directamente unos de otros, lo que reduce el acoplamiento y mejora la escalabilidad del sistema.

Actualmente, se utiliza para publicar y consumir el evento `user.registered`, que permite mantener sincronizados los datos de usuario entre la Auth API y la Users API.

Ilustración 10: Logo RabbitMQ.

3.5 Entity Framework Core

ORM (Object-Relational Mapper) utilizado para gestionar el acceso y la persistencia de datos en las distintas bases de datos del sistema.

Ilustración 11: Entity Framework Core.

Permite definir modelos de datos en C# y generar automáticamente las estructuras correspondientes en la base de datos relacional, facilitando las migraciones y el mantenimiento del esquema.

Su integración con .NET 8 ha simplificado la interacción con las instancias de **Azure SQL Database** empleadas por cada microservicio.

3.6 Azure Devops

Servicio de Microsoft empleado para la **gestión del ciclo de vida del desarrollo** y la **automatización del despliegue continuo** (CI/CD).

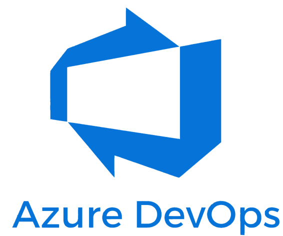

Ilustración 12: Azure Devops.

A través de **Azure Repos**, se ha gestionado el control de versiones de todos los microservicios, y mediante **pipelines**, se ha automatizado el proceso de construcción y publicación de los contenedores en el entorno de producción.

Esta integración ha permitido mantener un flujo de trabajo ágil, fiable y centralizado.

3.7 Azure SQL Database

Servicio de base de datos relacional en la nube utilizado para almacenar la información de cada microservicio de forma independiente.

Ilustración 13: Logo Azure SQL Database.

Ofrece alta disponibilidad, seguridad gestionada y escalabilidad dinámica, lo que permite que cada API disponga de su propio esquema de datos sin depender de una única instancia compartida.
Se integra de forma nativa con Entity Framework Core, simplificando la persistencia y las migraciones de datos.

3.8 Cloudinary

Servicio externo utilizado como **CDN (Content Delivery Network)** para el almacenamiento y entrega de imágenes de restaurantes.

Ilustración 14: Logo Cloudinary.

Cloudinary permite optimizar automáticamente el tamaño, formato y calidad de las imágenes, ofreciendo una

carga más rápida y eficiente desde cualquier dispositivo.
Su integración con el sistema garantiza que las imágenes se mantengan centralizadas y disponibles sin necesidad de almacenamiento local.

3.9 GIT

Sistema de control de versiones utilizado para mantener el historial del código fuente y facilitar la colaboración durante el desarrollo.

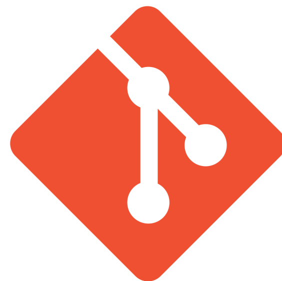

Ilustración 15: Logo Git.

Su integración con **Azure Repos** ha permitido llevar un control exacto de los cambios, gestionar ramas de desarrollo y garantizar la trazabilidad de cada modificación realizada en el proyecto.

Resumen de tecnologías utilizadas en el proyecto

Tecnología	Propósito principal	Categoría
.NET 8	Framework principal para el desarrollo de los microservicios y la lógica de negocio.	Backend / Desarrollo de APIs
ML.NET	Implementación del sistema de recomendaciones basado en aprendizaje automático.	Inteligencia Artificial / Machine Learning
YARP	Creación del API Gateway para el enrutamiento y autenticación centralizada.	Comunicación / Gateway
RabbitMQ	Comunicación asíncrona entre microservicios mediante eventos.	Mensajería / Integración
Entity Framework Core	Gestión del acceso y persistencia de datos mediante ORM.	Acceso a datos / ORM
Docker	Contenedorización y ejecución aislada de los microservicios.	Infraestructura/ Contenedores
Azure DevOps	Control de versiones, gestión de repositorios y despliegue continuo (CI/CD).	Integración y entrega continua

Azure SQL Database	Almacenamiento de datos relacional para cada servicio independiente.	Base de datos / Cloud
Cloudinary	Gestión y entrega optimizada de imágenes a través de CDN.	CDN
Git	Control de versiones distribuido e integración con Azure Repos.	Control de versiones

4 SISTEMA DE RECOMENDACIONES

La tecnología es una herramienta útil, pero nunca debe sustituir al juicio humano.

- Norbert Wiener -

El sistema de recomendaciones constituye el núcleo funcional de la plataforma **GastroMatch**, encargado de ofrecer a cada usuario una lista personalizada de restaurantes basada en sus gustos reales dentro de la aplicación.

Con el fin de obtener resultados más precisos, se ha diseñado un **modelo híbrido**, que combina dos enfoques clásicos en el ámbito de los sistemas de recomendación:

- **Filtrado colaborativo (Collaborative Filtering, CF)**: aprende patrones de comportamiento entre usuarios con gustos similares.
- **Filtrado basado en contenido (Content-Based Filtering, CB)**: analiza las características de los restaurantes para recomendar restaurantes con atributos semejantes a los que el usuario ya ha marcado con *like*.

Ambos modelos se integran dentro de la **Recommendation API**, que procesa los datos de interacción y genera la lista final de recomendaciones personalizadas.

4.1 Fundamentos del sistema de recomendación

Los sistemas de recomendación buscan predecir la afinidad entre un usuario y un conjunto de elementos, en este caso, restaurantes.
Para ello, se parte del histórico de interacciones registradas *likes* y de la información descriptiva de los restaurantes.

El enfoque híbrido empleado en **GastroMatch** permite aprovechar lo mejor de cada técnica:

- El **filtrado colaborativo** identifica patrones de comportamiento entre usuarios sin depender de los atributos explícitos del restaurante.
- El **filtrado basado en contenido** permite generar recomendaciones incluso en etapas tempranas, cuando el sistema aún dispone de pocos datos (problema conocido como *cold start*).

La combinación de ambos mejora la precisión y la cobertura de las recomendaciones, evitando sesgos y garantizando sugerencias coherentes con el perfil de cada usuario.

4.2 Filtrado Colaborativo

4.2.1 Principio teórico

El filtrado colaborativo es una técnica de recomendación basada en la premisa de que usuarios con gustos similares tenderán a coincidir en sus preferencias futuras. En lugar de analizar las características del restaurante (como su tipo de cocina o precio), el filtrado colaborativo se fija únicamente en los patrones de interacción entre usuarios y restaurantes (likes). En este trabajo se emplea feedback implícito positivo, representado mediante likes, un enfoque habitual en escenarios reales donde no se dispone de valoraciones explícitas por parte del usuario [16].

Para implementar el filtrado colaborativo en GastroMatch se utiliza la **factorización de matrices**. La factorización de matrices es una técnica ampliamente usada en sistemas de recomendación modernos. Su función es encontrar patrones ocultos en los datos de interacción.

Para entenderlo de forma intuitiva:

- Imaginamos una gran matriz donde cada fila representa un usuario y cada columna un restaurante.
- En esta matriz, el valor "1" indica que el usuario ha dado like a ese restaurante.
- La matriz está casi vacía, ya que cada usuario solo interactúa con unos pocos restaurantes.

La factorización de matrices descompone esta estructura en dos matrices más pequeñas que representan:

- Los patrones de preferencia aprendidos para cada usuario.
- Las características implícitas asociadas a cada restaurante (aunque dichas características no estén presentes explícitamente en los datos).

Estas características reciben el nombre de "factores" y representan tendencias o gustos generales. Por ejemplo:

- Preferencia por comida económica.
- Afinidad por cocina italiana.

El sistema aprende automáticamente estos factores sin que tengamos que definirlos. Cuando el modelo ya está entrenado, puede predecir qué restaurantes serían adecuados para cada usuario, incluso si nunca ha interactuado con ellos.

4.2.2 Implementación en ML.NET

Para implementar esta técnica, GastroMatch utiliza el componente `MatrixFactorizationTrainer` de ML.NET.
El modelo se entrena únicamente con los siguientes datos:

- Identificador del usuario
- Identificador del restaurante
- Like

Durante el entrenamiento, el modelo aprende patrones comunes entre usuarios y restaurantes y es capaz de predecir una **puntuación de afinidad** entre 0 y 1. Una puntuación más alta indica mayor probabilidad de que el usuario valore positivamente este restaurante.

El capítulo 5 detalla cómo se entrena, evalúa e integra este filtrado dentro de la arquitectura de microservicios.

4.3 Filtrado basado en contenido

4.3.1 Principio teórico

El filtrado basado en contenido recomienda restaurantes en función de sus características, sin tener en cuenta el comportamiento de otros usuarios [17]. La lógica es simple: si a un usuario le ha gustado un restaurante con unas características determinadas, es probable que también le gusten otros restaurantes que compartan esos atributos.

En GastroMatch se utilizan dos características principales:

- Tipo de cocina
- Rango de precio

Para que el sistema pueda compararlas matemáticamente, estas características se convierten en un vector numérico mediante *codificación one-hot.*
Esta técnica crea un conjunto de posiciones dentro del vector, colocando un 1 en la posición que corresponde al valor real y 0 en todas las demás.

Tipos de cocina incluidos:

1. Italiano
2. Japonés
3. Mexicano
4. Chino
5. Español
6. Indio
7. Mediterráneo
8. Americano
9. Vegano

Rangos de precio incluidos:

1. Económico
2. Moderado
3. Alto
4. Lujo

El tipo de cocina se codifica en un vector de 9 posiciones y el rango de precio en un vector de 4 posiciones. El vector final del restaurante es la concatenación de ambos. Por ejemplo:

- **Restaurante A**: Italiano, precio Moderado
  A = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
- **Restaurante B**: Indio, precio Alto
  B = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0]

Para implementar el filtrado basado en contenido en GastroMatch se utiliza la **distancia euclídea**. Esta distancia mide cuán separados están los vectores que representan a los restaurantes, cuanto menor es la distancia, mayor es la similitud. La distancia euclídea se calcula con la siguiente fórmula:

$$d(A, B) = \sum_{i=n}^{n} \sqrt{(A_i - B_i)^2}$$

- **A** y **B**: Son los vectores one-hot que representan los dos restaurantes.
- **Ai** y **Bi**: Son las posiciones individuales dentro del vector.
- **La suma de todos los (Ai – Bi)²**: Representa la separación total entre ambos restaurantes dentro del espacio de características.

Interpretación:

- **d(A, B) cercana a 0 → restaurantes muy similares**
- **d(A, B) grande → restaurantes diferentes**

El sistema calcula esta distancia entre cada restaurante candidato y los restaurantes que el usuario ha marcado con *like*. Después, calcula la distancia media, ordena los candidatos y selecciona los más cercanos.

4.3.2 Implementación en ML.NET

Para llevar a la práctica el filtrado basado en contenido descrito en el apartado anterior, GastroMatch utiliza un pipeline de transformación de datos de ML.NET. Este pipeline se encarga de convertir las características simbólicas de los restaurantes (tipo de cocina y rango de precio) en vectores numéricos, y de aplicar sobre ellos la métrica de distancia euclídea para medir la similitud. De este modo, la lógica teórica se traduce en un proceso automatizado e integrable en la Recommendation API.

El proceso es el siguiente:

1. ML.NET transforma las categorías y los rangos de precio en vectores one-hot.
2. Se genera un vector de características para cada restaurante, concatenando la codificación de tipo de cocina y rango de precio.
3. Se crean también vectores para los restaurantes que han recibido like por parte del usuario.
4. Para cada restaurante candidato se calcula la distancia euclídea respecto a los restaurantes con like, obteniendo una distancia media que resume su similitud global.
5. El sistema ordena los restaurantes según esa distancia media y selecciona los más cercanos para construir el conjunto de recomendaciones.

El capítulo 5 detalla cómo se evalúa e integra este filtrado dentro de la arquitectura de microservicios.

4.4. Integración Híbrida de los Modelos

El sistema híbrido combina ambos enfoques para obtener recomendaciones más robustas.

El filtrado colaborativo aporta precisión cuando existen suficientes likes e interacciones, mientras que el filtrado basado en contenido garantiza cobertura en situaciones con menos datos.

La puntuación final de recomendación se obtiene combinando las salidas de los dos modelos mediante una suma ponderada:

$$ScoreFinal = (wCF \times ScoreCF) + (wCB \times ScoreCB)$$

- **ScoreCF**: Es la puntuación generada por el filtrado colaborativo para un restaurante concreto.
- **ScoreCB**: Es la puntuación procedente del filtrado basado en contenido para un restaurante concreto.
- **wCF:** peso del filtrado colaborativo, su valor está entre 0 y 1.
- **wCB**: peso del filtrado basado en contenido, su valor está entre 0 y 1.
- La suma de **wCF** y **wCB** debe ser igual a 1.

Estos pesos se han fijado atendiendo al tamaño del conjunto de datos:

- Cuando hay poca interacción, **wCB** se establece más alto para compensar la escasez de datos del modelo colaborativo.
- Conforme el sistema acumula más likes y usuarios, **wCF** puede incrementarse para aprovechar la capacidad predictiva del modelo colaborativo.

El resultado es un sistema flexible que mantiene la calidad de las recomendaciones en todo momento.

5 Implementación en .Net

Diseñar tecnología es, en el fondo, diseñar experiencias humanas.

- Don Norman -

5.1 Arquitectura del sistema y microservicios que lo conforman

La arquitectura de **GastroMatch** se ha diseñado bajo un enfoque **modular y distribuido**, siguiendo los principios de los **microservicios** y **Clean Architecture**, tal y como se describe en la documentación técnica sobre arquitecturas de microservicios [1].
Cada API constituye una **unidad funcional independiente**, desplegada como un **contenedor aislado** dentro de **Azure Container Apps**, y responsable de un dominio concreto dentro del sistema.
De esta forma, se consigue una aplicación **altamente escalable, mantenible y extensible**, en la que cada módulo puede evolucionar sin afectar al resto.

El sistema se estructura en torno a un **API Gateway**, que actúa como punto único de entrada para los clientes, y a un conjunto de **microservicios especializados** conectados mediante **HTTP** y **RabbitMQ**.

Cada microservicio cuenta con su **propia base de datos** y modelo de datos aislado, en cumplimiento del **principio de independencia de datos**. Esto garantiza que cada dominio evolucione de forma autónoma, evitando dependencias cruzadas y manteniendo la integridad de la arquitectura.

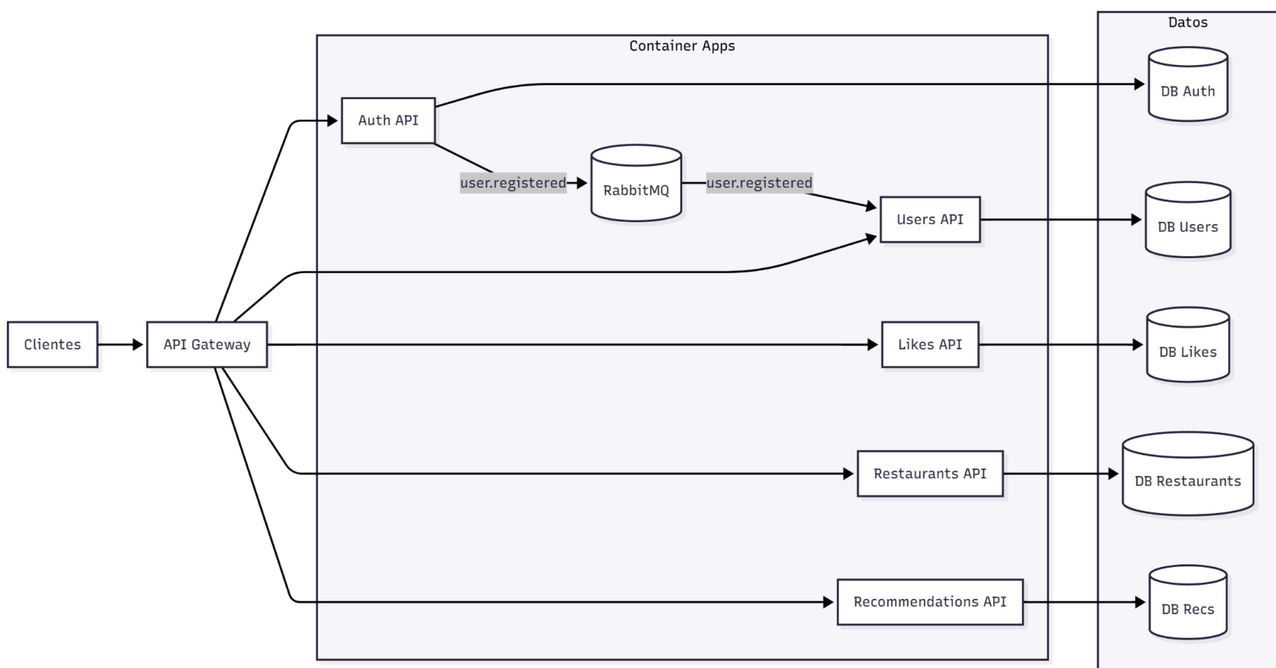

Ilustración 16: Esquema general de la arquitectura del Sistema.

En la Ilustración 16 se muestra el esquema general de la arquitectura de GastroMatch, incluyendo el API

Gateway y los microservicios que conforman el sistema.

A continuación, se describe la función de cada componente:

5.1.1 API Gateway

El **Gateway**, desarrollado con *YARP (Yet Another Reverse Proxy)*, se encarga de enrutar todas las peticiones hacia los microservicios correspondientes.
Es el **único punto público expuesto** del sistema, y se comunica internamente con el resto de servicios a través de la red privada de contenedores.

Entre sus principales responsabilidades se encuentran:

- Autenticación y validación de tokens JWT.
- Propagación del identificador de usuario a las cabeceras (X-USERID).
- Enrutamiento dinámico según prefijo de ruta (/auth, /users, /restaurants, etc.).

Esto permite desacoplar completamente el cliente (frontend) de la estructura interna de las APIs.

5.1.2 Auth API

La Auth API gestiona la autenticación del sistema y proporciona los mecanismos necesarios para que un usuario pueda registrarse y obtener un token JWT válido. Su responsabilidad se limita a validar credenciales y emitir tokens, evitando almacenar o gestionar información del perfil, que queda delegada a Users API. Tras un registro correcto, publica un evento para permitir la sincronización con los demás microservicios sin dependencias directas.

Entre sus principales responsabilidades se encuentran:

- Registrar nuevos usuarios en el sistema.
- Generar y devolver tokens JWT al validar correctamente las credenciales del usuario.
- Publicar el evento user.registered para que Users API cree su propio registro asociado.

5.1.3 Users API

La Users API es el servicio encargado de gestionar la información del perfil de los usuarios dentro de la plataforma. No participa en los procesos de autenticación, sino que actúa de forma complementaria a Auth API. Cuando un usuario se registra, recibe el evento user.registered publicado en RabbitMQ y crea automáticamente el registro correspondiente en su propia base de datos, manteniendo así un desacoplamiento total entre servicios. Además, expone endpoints para consultar los datos básicos del usuario.

Entre sus principales responsabilidades se encuentran:

- Crear el perfil del usuario al recibir el evento user.registered.
- Almacenar y gestionar la información del usuario (nombre, email y datos asociados).
- Permitir la consulta del perfil del usuario desde el frontend o servicios internos.
- Mantener la separación total entre autenticación y gestión de datos personales.

5.1.4 Restaurants API

La Restaurants API gestiona todo el catálogo de restaurantes disponible en la plataforma. Es la fuente principal de información para el usuario y un componente clave para el sistema de recomendaciones, ya que proporciona los atributos necesarios para calcular similitudes entre restaurantes. Cada registro incluye datos como nombre, tipo de cocina, rango de precio y la URL de la imagen servida mediante Cloudinary. Además,

ofrece endpoints tanto para la consulta general del catálogo como para la administración de nuevos restaurantes.

Entre sus principales responsabilidades se encuentran:

- Almacenar y gestionar los datos del catálogo de restaurantes.
- Permitir listar, filtrar y obtener el detalle de cualquier restaurante.
- Mantener atributos relevantes para las recomendaciones, como tipo de cocina y rango de precio.
- Gestionar la URL de imagen de cada restaurante a través de servicios CDN como Cloudinary.
- Consumir los eventos like.created y like.deleted para actualizar el contador total de likes de cada restaurante.

5.1.5 Likes API

La Likes API centraliza todas las interacciones de los usuarios con los restaurantes mediante el registro de likes. Actúa como la fuente principal de preferencias explícitas dentro del sistema y proporciona los datos necesarios para que el motor de recomendaciones pueda analizar comportamientos y generar sugerencias personalizadas. Además, publica eventos para mantener sincronizados a otros microservicios, como Restaurants API, permitiendo actualizar sus contadores internos sin crear dependencias directas.

Entre sus principales responsabilidades se encuentran:

- Registrar y eliminar likes asociados a usuarios y restaurantes.
- Permitir consultar los restaurantes que un usuario ha marcado con like.
- Publicar los eventos like.created y like.deleted para mantener sincronizado al resto de microservicios.

5.1.6 Recommendation API

La Recommendation API constituye el núcleo inteligente del sistema. Es el servicio encargado de procesar los datos de actividad de los usuarios y la información del catálogo de restaurantes para generar recomendaciones personalizadas. Para ello implementa un enfoque híbrido que combina Filtrado Colaborativo y Filtrado Basado en Contenido utilizando ML.NET (descritos en el punto 4). El servicio permite entrenar el modelo cuando sea necesario y expone el endpoint /feed para obtener recomendaciones adaptadas a cada usuario.

Entre sus principales responsabilidades se encuentran:

- Recopilar los datos de Likes API y Restaurants API para construir los conjuntos de entrenamiento.
- Entrenar y ejecutar el modelo de **Filtrado Colaborativo (CF)** para detectar similitudes entre usuarios.
- Entrenar y ejecutar el modelo de **Filtrado Basado en Contenido (CB)** para comparar restaurantes por sus características.
- Combinar ambos enfoques en un sistema híbrido para mejorar la precisión de las recomendaciones.
- Exponer el endpoint /feed para obtener el listado de recomendaciones personalizadas del usuario.

5.2 Características comunes de las APIs

En el apartado anterior se han descrito por separado los microservicios que forman parte de GastroMatch. Más allá de sus responsabilidades individuales, todas las APIs comparten una serie de características comunes a nivel de arquitectura, estilo de desarrollo y forma de exponer sus funcionalidades.

Este enfoque unificado facilita el mantenimiento, la extensibilidad del sistema y la colaboración entre servicios, además de ofrecer una experiencia homogénea tanto para el desarrollador como para el frontend

que las consume.

5.2.1 Arquitectura interna

Cada API se ha desarrollado sobre .NET 8 siguiendo los principios de Clean Architecture y Domain-Driven Design (DDD), según los enfoques propuestos por Robert C. Martin [2] y Eric Evans [3]. Todas ellas comparten una estructura basada en cuatro capas bien definidas:

- **Domain**: contiene las entidades y reglas de negocio que representan el núcleo del dominio de cada servicio.
- **Application**: implementa la lógica de aplicación mediante casos de uso y servicios que orquestan las operaciones del dominio.
- **Infrastructure**: gestiona la persistencia, la configuración de bases de datos y la integración con servicios externos (por ejemplo, RabbitMQ o ML.NET en los servicios que lo requieren).
- **Presentation**: expone los endpoints HTTP que interactúan con los clientes a través de controladores REST.

En la práctica, cuando llega una petición HTTP, esta entra por la capa de Presentation, se procesa en Application, se aplican las reglas de negocio definidas en Domain y, si es necesario, se accede a la base de datos o a servicios externos a través de Infrastructure.
Esta división en capas permite mantener el código desacoplado y facilita que cambios tecnológicos (por ejemplo, sustituir una base de datos o un proveedor externo) no afecten a las reglas de negocio.

5.2.2 Convenciones de desarrollo y respuesta

Todas las APIs siguen las mismas convenciones a la hora de definir y exponer sus endpoints. Las rutas se agrupan por recurso principal (por ejemplo, /users, /restaurants, /likes, /recommendations), de modo que el cliente puede identificar fácilmente qué API gestiona cada tipo de información.

Los verbos HTTP se utilizan siguiendo el estándar REST:

- GET para la lectura de recursos.
- POST para la creación.
- PUT o PATCH para la actualización.
- DELETE para la eliminación.

Del mismo modo, los códigos de estado HTTP se utilizan de forma coherente en todos los servicios (200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 404 Not Found, etc.).
Todas las APIs devuelven objetos de respuesta autodescriptivos, que incluyen únicamente la información necesaria para el cliente. De esta forma, el frontend puede trabajar con cualquier servicio de GastroMatch siguiendo siempre el mismo estilo de llamadas y de respuestas.

Este enfoque permite una comunicación sencilla, desacoplada y ampliamente compatible entre servicios y clientes, alineándose con las recomendaciones habituales para el diseño de APIs web [4].

5.2.3 Seguridad

La autenticación se gestiona de forma centralizada a través de Auth API, que se encarga de emitir los tokens JWT. El API Gateway valida estos tokens en cada petición entrante y, una vez comprobados, propaga la identidad del usuario a las APIs internas mediante la cabecera X-USERID.

Gracias a este esquema, los microservicios pueden confiar en dicha cabecera y centrarse exclusivamente en su lógica de dominio, manteniendo la seguridad desacoplada del resto de la aplicación y evitando duplicar la lógica de autenticación en cada servicio.

5.2.4 Documentación y pruebas

Cada API se documenta automáticamente mediante Swagger, lo que permite explorar los endpoints disponibles, conocer los formatos de entrada y salida y realizar pruebas manuales de forma sencilla. Esta documentación sirve tanto de referencia para el desarrollo del frontend como de ayuda durante las fases de validación y depuración.

En entornos de desarrollo se utiliza Docker Compose para levantar de forma conjunta las diferentes APIs. Esto permite replicar un entorno de ejecución similar al de producción en una única máquina, simplificando las pruebas locales e integradas del sistema.

5.3 Comunicación entre microservicios

GastroMatch combina dos formas de comunicación para coordinar sus microservicios: la comunicación síncrona mediante HTTP y la comunicación asíncrona basada en eventos.
Este enfoque permite equilibrar coherencia, rendimiento y desacoplamiento, asegurando que cada servicio interactúe únicamente con aquellos que necesita.

5.3.1 Comunicación síncrona (HTTP)

La comunicación síncrona se utiliza cuando un servicio necesita obtener información actualizada de otro en el momento de procesar una solicitud. Todas las peticiones internas se realizan a través del API Gateway, que actúa como proxy inverso y añade la cabecera X-USERID con la identidad del usuario autenticado.

Cuándo se utiliza

- Cuando un servicio necesita datos en tiempo real para construir su respuesta.
- Cuando se requiere consistencia inmediata.

Casos representativos de uso

- Recommendations API → Likes API (recuperar los likes del usuario).
- Recommendations API → Restaurants API (obtener atributos de restaurantes).
- Gateway → cualquier API interna (validación de token y propagación de usuario).

Ventajas

- Acceso inmediato a datos actualizados.
- Sencillez en la comprensión y depuración.
- Uso centralizado del Gateway para seguridad.

Limitaciones

- Ambos servicios deben estar disponibles simultáneamente.
- Introduce acoplamiento temporal entre servicios.

5.3.2 Comunicación asíncrona (RabbitMQ)

La comunicación asíncrona se utiliza para transmitir eventos que no requieren una respuesta inmediata.
En GastroMatch se implementa mediante RabbitMQ, lo que permite que los servicios permanezcan desacoplados y reaccionen a cambios del sistema de forma independiente.

Para facilitar el uso de este mecanismo y evitar la duplicación de código entre microservicios, se ha desarrollado una **librería NuGet propia** que encapsula la lógica común de publicación y consumo de

eventos. Esta librería proporciona clases base para los roles de *publisher* y *consumer*, simplificando la integración de RabbitMQ en cada servicio y garantizando un uso homogéneo de la mensajería en todo el sistema.

El diseño y la lógica interna de esta librería se detallan en el **Anexo 2**, donde se describe su estructura y funcionamiento con mayor profundidad.

Cuándo se utiliza

- Cuando un cambio en un servicio debe notificarse a otros sin necesidad de una respuesta inmediata.
- Para mantener coherencia eventual entre dominios sin crear dependencias directas.

Casos representativos de uso

- **user.registered** → publicado por Auth API y consumido por Users API.
- **like.created** y **like.deleted** → publicados por Likes API y consumidos por Restaurants API para actualizar contadores de likes totales de cada restaurante.

Ventajas

- Fuerte desacoplamiento entre microservicios.
- Resiliencia ante fallos temporales de otros servicios.
- Facilita la escalabilidad y la evolución del sistema.

Limitaciones

- La coherencia no es inmediata, sino eventual.
- Requiere infraestructura adicional y gestión de colas.

El sistema combina comunicación síncrona y asíncrona para obtener lo mejor de ambos enfoques.
Los eventos garantizan la sincronización desacoplada entre servicios, mientras que las peticiones HTTP permiten obtener la información más reciente cuando es necesario responder al usuario.

En la Ilustración 17 se representa la comunicación entre los microservicios del sistema, diferenciando las interacciones síncronas mediante HTTP y la comunicación asíncrona basada en eventos a través de RabbitMQ.

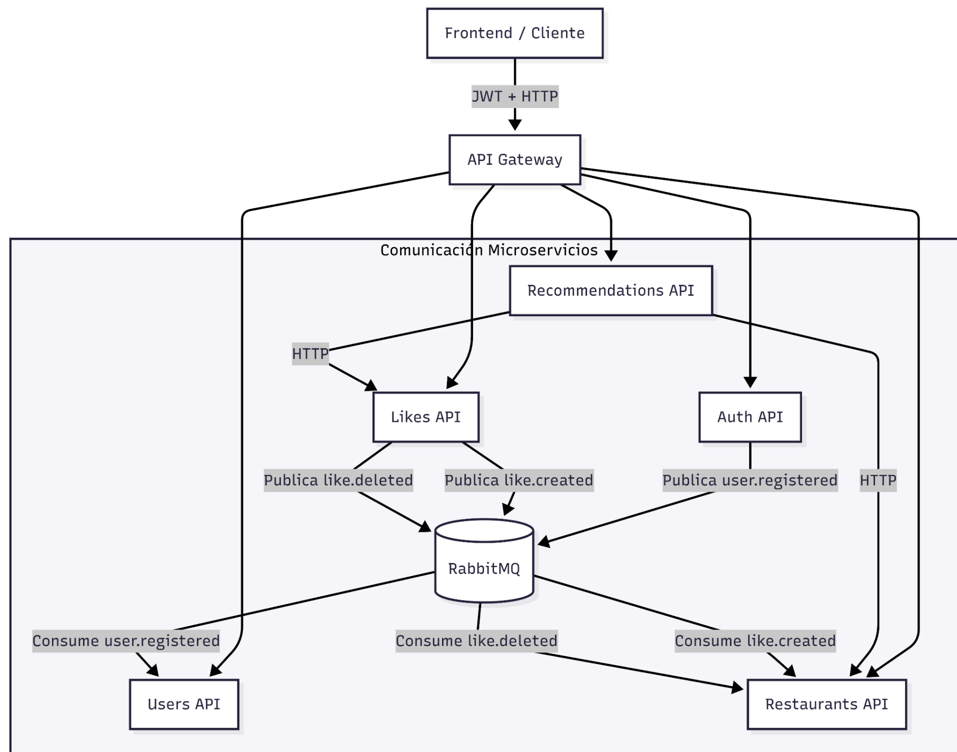

Ilustración 17: Comunicación entre los servicios del sistema

5.4 Implementación de las APIs

Cada uno de los microservicios de **GastroMatch** ha sido desarrollado como una API REST independiente, con su propio dominio, base de datos y responsabilidades específicas.
En esta sección se describen los componentes principales del sistema, su estructura de endpoints y las particularidades de su implementación.

Comenzaremos por los dos pilares de acceso y seguridad del sistema: **el API Gateway y la Auth API**.

A continuación, se presentan las APIs que gestionan los elementos centrales del dominio **Users API, Restaurants API y Likes API,** y finalmente se detalla la API que constituye el núcleo inteligente de GastroMatch, **Recommendations API**.

5.4.1 API Gateway

El API Gateway es el punto de entrada único al conjunto de microservicios de GastroMatch. Todas las peticiones provenientes del cliente pasan primero por este componente, que se encarga de validar el token JWT enviado por el usuario autenticado, extraer su identificador y propagarlo al resto de APIs mediante la cabecera X-USERID.
Además, el Gateway abstrae por completo la estructura interna del sistema: el cliente se comunica con un único endpoint público sin necesidad de conocer las direcciones reales ni la topología de los microservicios.

Su responsabilidad principal es enrutar cada solicitud al servicio correspondiente en función del prefijo de la ruta, actuando como proxy inverso y gestionando la seguridad de forma centralizada.

Tecnologías y dependencias

- **YARP (Yet Another Reverse Proxy):** middleware encargado del enrutamiento dinámico hacia los

microservicios internos.
- **JWT Bearer Authentication:** utilizado para validar los tokens emitidos por Auth API.
- **ASP.NET Core 8:** infraestructura sobre la que se construye el Gateway, proporcionando el pipeline de middlewares y el sistema de autenticación.

ASP.NET Core actúa como la base del pipeline de ejecución, mientras que YARP se integra como middleware especializado en el reenvío de peticiones.

Enrutamiento y funcionamiento de YARP

YARP funciona mediante dos elementos principales:

1. **Routes** → se encargan de identificar qué solicitudes debe interceptar el Gateway.
2. **Clusters** → definen los microservicios de destino, con una o varias direcciones internas.

Cuando una petición llega al Gateway, YARP:

1. Comprueba si la ruta coincide con alguna Route.
2. Determina el Cluster asociado a esa ruta.
3. Reenvía la petición al destino configurado (por ejemplo, http://likeapi:8080).

La configuración de YARP se realiza en appsettings.json, se muestra en la Ilustración 18, donde se define el mapeo entre los prefijos de ruta y los microservicios de destino.

```
{
  "ReverseProxy": {
    "Routes": {
      "auth": {
        "ClusterId": "auth-cluster",
        "Match": { "Path": "/auth/{**catch-all}" },
        "Transforms": [ { "PathRemovePrefix": "/auth" } ],
        "AuthorizationPolicy": "Anonymous", //permitimos entrada desde cualquier sitio
        "CorsPolicy": "frontend"
      },
      "users": {
        "ClusterId": "user-cluster",
        "Match": { "Path": "/user/{**catch-all}" },
        "Transforms": [ { "PathRemovePrefix": "/user" } ],
        "AuthorizationPolicy": "Default",
        "CorsPolicy": "frontend"
      },
      "restaurant": {
        "ClusterId": "restaurant-cluster",
        "Match": { "Path": "/restaurant/{**catch-all}" },
        "Transforms": [ { "PathRemovePrefix": "/restaurant" } ],
        "AuthorizationPolicy": "Default",
        "CorsPolicy": "frontend"
      },
      "like": {
        "ClusterId": "like-cluster",
        "Match": { "Path": "/like/{**catch-all}" },
        "Transforms": [ { "PathRemovePrefix": "/like" } ],
        "AuthorizationPolicy": "Default",
        "CorsPolicy": "frontend"
      },
      "recommendation": {
        "ClusterId": "recommendation-cluster",
        "Match": { "Path": "/recommendation/{**catch-all}" },
        "Transforms": [ { "PathRemovePrefix": "/recommendation" } ],
        "AuthorizationPolicy": "Default",
        "CorsPolicy": "frontend"
      }
    },
      "Clusters": {
        "auth-cluster": { "Destinations": { "d1": { "Address": "http://authapi/" } } },
        "user-cluster": { "Destinations": { "d1": { "Address": "http://userapi/" } } },
        "restaurant-cluster": { "Destinations": { "d1": { "Address": "http://restaurantapi/" } } },
        "like-cluster": { "Destinations": { "d1": { "Address": "http://likeapi/" } } },
        "recommendation-cluster": { "Destinations": { "d1": { "Address": "http://recommendationapi/" } } }
      }
```

Ilustración 18: Configuración YARP.

Cómo interpreta YARP esta configuración

- Todo lo que empiece por **/api/like/** se asigna al **like-cluster**.
- YARP reescribe la ruta manteniendo el resto del path.
- YARP reenvía la petición al destino interno configurado **Address**.
- La respuesta se devuelve al cliente sin que este conozca la existencia del microservicio real.

Este mecanismo permite cambiar la infraestructura interna (puertos, contenedores, instancias) sin modificar el frontend.

Flujo interno y decisiones de diseño

El procesamiento de una petición típica en el Gateway sigue los pasos siguientes:

1. **Validación del token JWT**: El Gateway utiliza AddAuthentication().AddJwtBearer() para validar la firma y el contenido del token. Si el token es incorrecto o ha expirado, la petición no se reenvía al

microservicio y se devuelve directamente un 401 Unauthorized.

Esto garantiza que las APIs internas nunca reciben solicitudes no autenticadas.

2. **Inserción de la cabecera X-USERID (middleware personalizado):** Después de validar el token, el Gateway utiliza un middleware propio para extraer el identificador del usuario (ClaimTypes.NameIdentifier) y añadirlo a la cabecera X-USERID.

```
app.Use(async (ctx, next) =>
{
    ctx.Request.Headers.Remove("X-USERID");

    if (ctx.User?.Identity?.IsAuthenticated == true)
    {
        var userId =
            ctx.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;

        if (!string.IsNullOrEmpty(userId))
            ctx.Request.Headers.Append("X-USERID", userId);
    }

    await next();
});
```

Ilustración 19: middleware propagación cabecera X-USERID.

En la Ilustración 19 se muestra el código implementado para este middleware personalizado.

Este diseño permite que todas las APIs internas confíen en esta cabecera sin necesidad de validar tokens ni conocer la estructura JWT.

3. **Enrutamiento mediante YARP:** Tras la validación y el middleware, YARP toma control del pipeline: identifica el `Cluster` correspondiente, reenvía la petición al microservicio correcto, espera la respuesta, y la devuelve al cliente.

Esto garantiza que las APIs internas nunca reciben solicitudes no autenticadas.

El Gateway concentra: la **seguridad**, el **enrutamiento dinámico**, y la **propagación de la identidad** del usuario. Gracias a esto, las APIs internas no necesitan exponer endpoints públicamente ni preocuparse por la autenticación, lo que simplifica su implementación y aumenta la mantenibilidad del sistema.

5.4.2 Auth API

La Auth API es el servicio responsable de gestionar la autenticación de los usuarios en GastroMatch. Su función se centra en dos tareas principales: registrar nuevos usuarios en el sistema y validar sus credenciales durante el inicio de sesión.
Cuando un usuario se registra correctamente, la API genera su identidad en el sistema de autenticación y publica un evento user.registered que permite que otros microservicios, como Users API, creen su propio registro interno sin necesidad de comunicarse directamente con Auth API.

Además, Auth API es el único componente encargado de emitir tokens JWT firmados, que posteriormente serán validados por el API Gateway para permitir el acceso a las APIs internas.

Tecnologías y dependencias

- **JWT Bearer Token Generation:** usado para emitir tokens con firma HMAC-SHA256.

- **Password Hashing (PBKDF2):** mecanismo empleado para almacenar contraseñas de forma segura.
- **RabbitMQ Publisher:** utilizado para publicar el evento user.registered.
- **ASP.NET Core 8:** base del pipeline y del sistema de autenticación.

Endpoints principales

Método	Ruta	Descripción
POST	/api/auth/register	Registra un nuevo usuario y publica el evento user.registered.
POST	/api/auth/login	Valida las credenciales y genera un token JWT.

Registro de usuario (`/register`)

1. El cliente envía los datos de registro (nombre, email y contraseña).
2. La contraseña se transforma mediante un algoritmo seguro de hashing para no almacenarla en texto plano.
3. Se crea el registro interno del usuario en la base de datos de Auth API.
4. La API publica un evento `user.registered` en RabbitMQ con los datos básicos del usuario (ID, nombre y email).
   Este evento permite que Users API cree su propio registro de manera asíncrona, manteniendo el desacoplamiento entre los dominios de autenticación y de gestión de usuarios.

El flujo completo del proceso de registro de un usuario se representa en la Ilustración 20, incluyendo la generación del evento user.registered y su consumo por Users API.

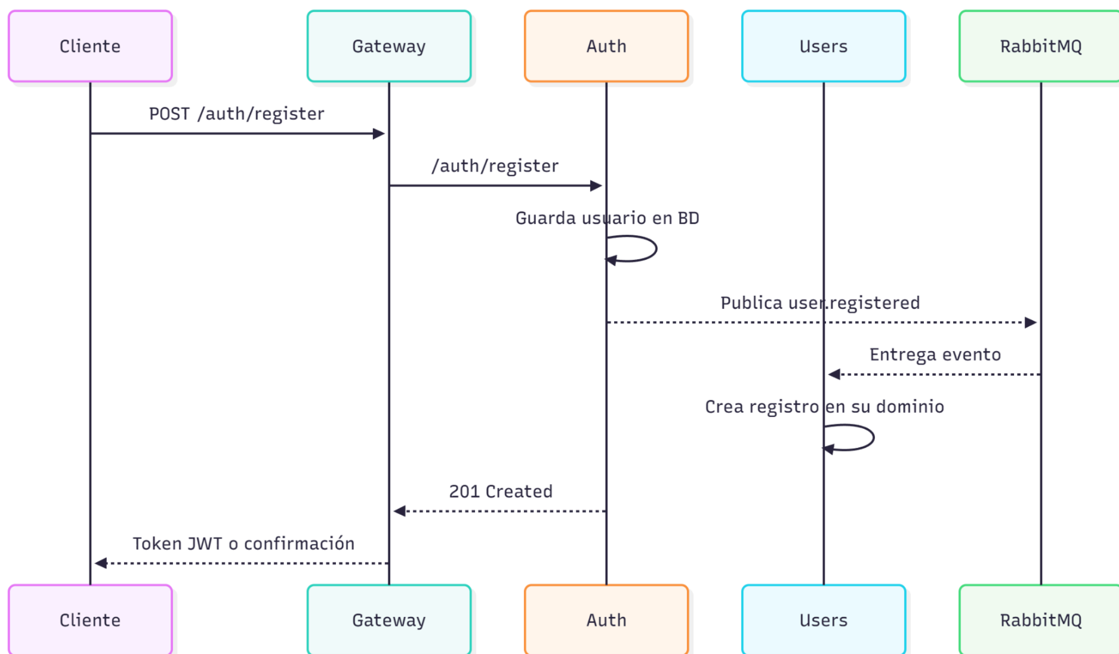

Ilustración 20: Flujo completo de registro de un usuario.

Inicio de sesión (`/login`)

1. La API recibe el email y la contraseña.
2. Comprueba que el usuario existe y valida la contraseña comparando el hash almacenado con el hash del usuario.
3. Si las credenciales son correctas, genera un **token JWT firmado**, que incluye:
   - sub (identificador del usuario),
   - nombre,
   - email,
   - fecha de expiración.
4. La respuesta incluye exclusivamente el token JWT, que posteriormente será validado por el API Gateway en cada petición.

En la Ilustración 21 se muestra el flujo completo del inicio de sesión, desde la validación de credenciales hasta la emisión del token JWT.

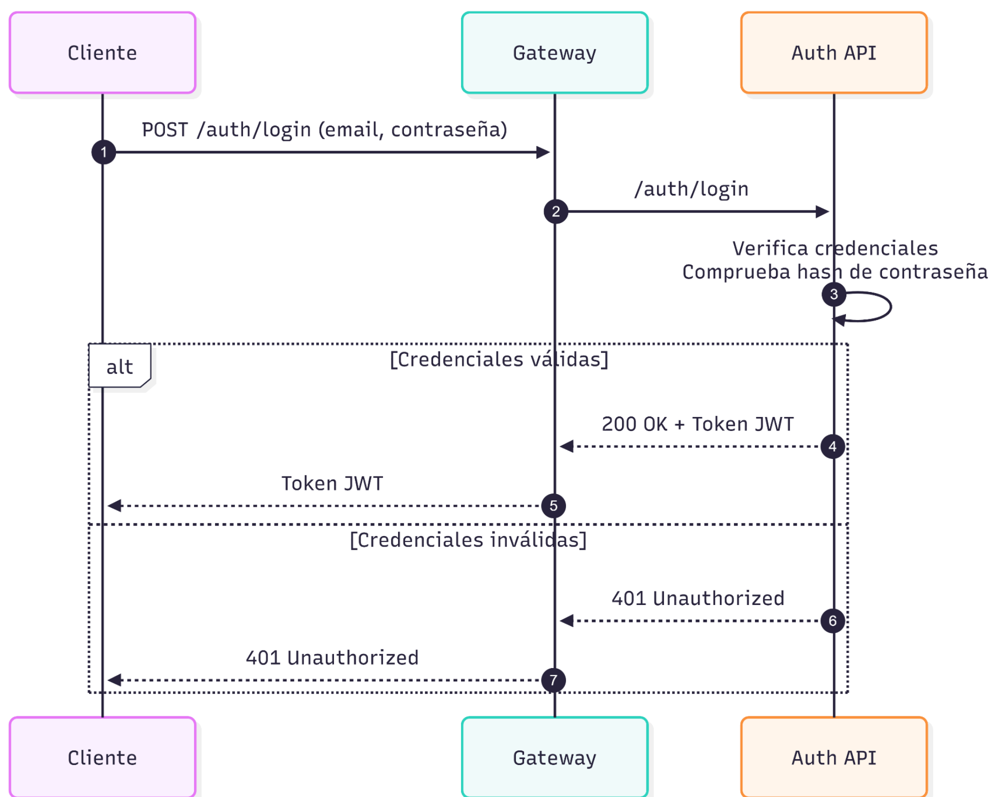

Ilustración 21: Flujo completo de login.

Generación del token JWT

Auth API utiliza una clave secreta simétrica y los parámetros configurados en appsettings.json para firmar los tokens. El proceso utiliza JwtSecurityTokenHandler, y el resultado es un token compacto listo para enviarse al cliente.
La configuración del token incluye:

- Firma HMAC-SHA256
- Expiración de 24 horas
- Claims que identifican al usuario

Este enfoque garantiza que solo el Gateway necesite validar los tokens, mientras que las APIs internas confían únicamente en la cabecera X-USERID.

Justificación del diseño

- Auth API concentra toda la lógica de autenticación, evitando duplicación en otros servicios.
- El uso de eventos desacopla la creación del usuario, permitiendo escalabilidad futura.
- La emisión de tokens JWT firmados permite autenticación sin estado, ideal para microservicios.
- La API se mantiene pequeña y especializada, siguiendo principios de Single Responsibility.

5.4.3 Users API

La Users API es el servicio encargado de gestionar la información del perfil de los usuarios dentro de GastroMatch. Su función es complementar a Auth API almacenando los datos personales del usuario (nombre y email), además de permitir su consulta desde el frontend.
Este microservicio no participa en la autenticación ni en la emisión de tokens; su responsabilidad es mantener el estado del usuario en su dominio.

La creación de registros en esta API no depende de llamadas directas desde Auth API, sino que se realiza de forma completamente desacoplada gracias al evento user.registered, garantizando independencia entre los dominios de autenticación y gestión de usuarios.

Tecnologías y dependencias

- **RabbitMQ Consumer:** utilizado para recibir y procesar el evento user.registered.
- **Base de datos SQL:** almacena los perfiles de usuario gestionados por este servicio.
- **ASP.NET Core 8:** infraestructura utilizada para exponer los endpoints REST.

Endpoints principales

Método	Ruta	Descripción
GET	/api/users/me	Devuelve el perfil del usuario autenticado.

Esta API no implementa endpoints de creación ni actualización manual, ya que la creación se realiza mediante eventos y los datos básicos no son editables desde el cliente.

Consumo del evento `user.registered`

Cuando Auth API registra un nuevo usuario, publica un evento con la información básica del usuario.
Users API está suscrita a este evento mediante RabbitMQ y ejecuta el siguiente flujo:

1. RabbitMQ entrega el mensaje a Users API.
2. El servicio valida el contenido del evento.
3. Se crea en la base de datos interna el registro correspondiente al nuevo usuario, con su Id, nombre y email.

Este diseño simplifica la API y permite separar totalmente autenticación y gestión de datos personales.

Recuperación del perfil del usuario autenticado

El endpoint /users/me utiliza la cabecera X-USERID, proporcionada por el API Gateway, para identificar al

usuario que realiza la petición. El flujo es:

1. El Gateway valida el JWT y añade X-USERID.
2. Users API recibe la petición sin necesidad de validar tokens.
3. Se consulta la base de datos interna utilizando el identificador recibido.
4. Se devuelve la información básica del perfil (nombre y email).

5.4.4 Restaurants API

La Restaurants API es el microservicio responsable de gestionar el catálogo de restaurantes de GastroMatch. En este servicio se almacena toda la información relevante de cada restaurante, como su nombre, tipo de cocina, rango de precio e imágenes asociadas.
La API expone endpoints para la consulta y administración del catálogo, permitiendo obtener listados paginados, recuperar restaurantes individuales o cargar imágenes.

Además, este servicio consume los eventos like.created y like.deleted provenientes de Likes API, lo que le permite mantener actualizado el contador de likes de cada restaurante sin necesidad de dependencias directas.

Tecnologías y dependencias

- **Base de datos SQL:** almacena los datos del catálogo de restaurantes.
- **Cloudinary (CDN):** utilizado para almacenar y servir imágenes de restaurantes.
- **RabbitMQ Consumer:** suscrito a los eventos like.created y like.deleted.
- **ASP.NET Core 8:** infraestructura utilizada para exponer los endpoints REST.

Endpoints principales

Método	Ruta	Descripción
GET	/api/restaurants/all	Obtiene un listado paginado de restaurantes mediante cursor y pageSize.
POST	/api/restaurants/allbyids	Devuelve los restaurantes cuyos identificadores se envían en el cuerpo de la solicitud.
GET	/api/restaurants/{id}	Obtiene el detalle de un restaurante concreto.
GET	/api/restaurants/top	Devuelve los restaurantes mejor valorados según categoría, filtrando opcionalmente por nuevos restaurantes.
POST	/api/restaurants/	Crea un nuevo restaurante usando un objeto JSON

		enviado en el cuerpo.
POST	/api/restaurants/{id}/image	Sube una imagen asociada a un restaurante.

Los endpoints restaurants/all y restaurants/{id} son consumidos directamente por el frontend para mostrar el catálogo y el detalle del restaurante, respectivamente.

Todos los endpoints administrativos (crear restaurante, subir imagen) están pensados para uso interno o herramientas de gestión, no para el frontend del usuario final.

Consultas paginadas mediante cursor y pageSize

El endpoint /restaurants/all utiliza **paginación basada en cursor**, una técnica más eficiente que la paginación tradicional con offset, especialmente cuando el número de registros puede escalar.

El funcionamiento es el siguiente:

1. El cliente envía un cursor (el último ID recibido) y un pageSize (tamaño de página deseado).
2. El servicio recupera únicamente los registros **posteriores al cursor**, limitando la cantidad al pageSize.
3. La respuesta incluye el nuevo cursor, permitiendo al cliente solicitar la siguiente página.

Este enfoque es ideal para listados grandes como el catálogo de restaurantes.

Subida y gestión de imágenes (Cloudinary)

El endpoint /restaurants/{id}/image acepta una solicitud **multipart/form-data**, que incluye:

- Archivo de imagen.
- Indicador (isMain) para marcar si la imagen debe ser la principal del restaurante.

El flujo es:

1. El servicio recibe el archivo y lo valida.
2. La imagen se sube a Cloudinary usando su API.
3. Se almacena la URL resultante en la base de datos del restaurante.
4. Si isMain = true, se actualiza el campo principal de imagen.

La Ilustración 22 representa el flujo de subida y gestión de imágenes de un restaurante, incluyendo la integración con el servicio externo Cloudinary.

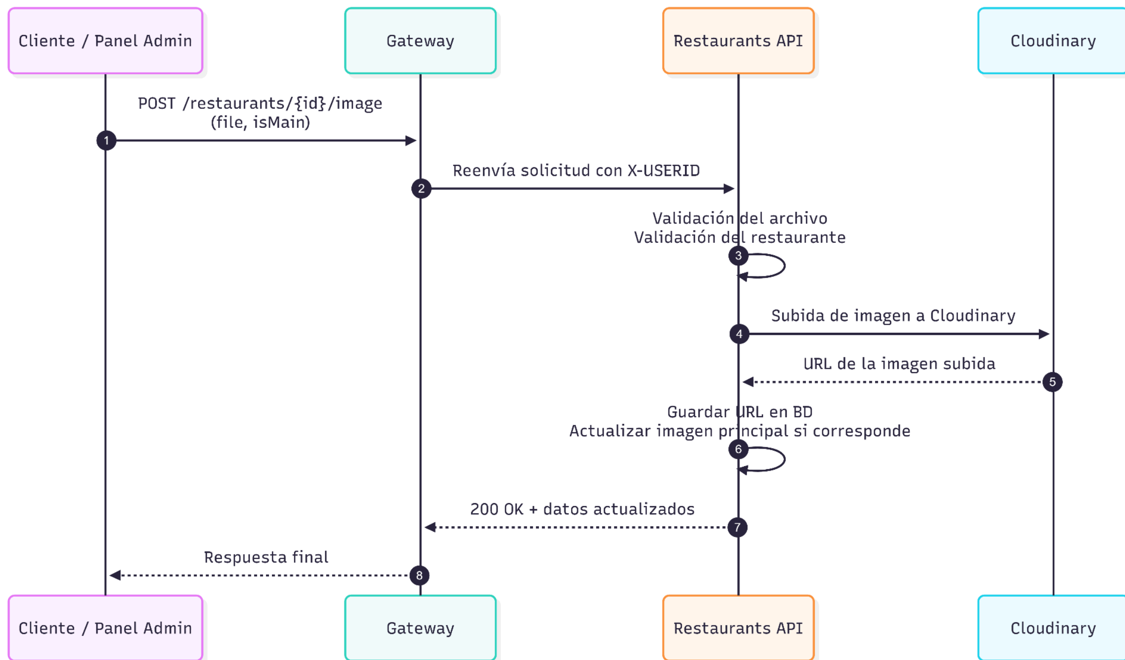

Ilustración 22: Flujo subida de imagen de un restaurante.

Consumo de los eventos like.created y like.deleted

Restaurants API está suscrita a los eventos emitidos por Likes API. Su responsabilidad es mantener actualizado el contador de likes de cada restaurante:

1. Likes API publica el evento like.created o like.deleted.
2. RabbitMQ entrega el evento a Restaurants API.
3. Incrementa o decrementa el contador de likes del restaurante en su base de datos.

En la Ilustración 23 se muestra el flujo de consumo de los eventos like.created y like.deleted por parte de Restaurants API para mantener actualizado el contador de likes.

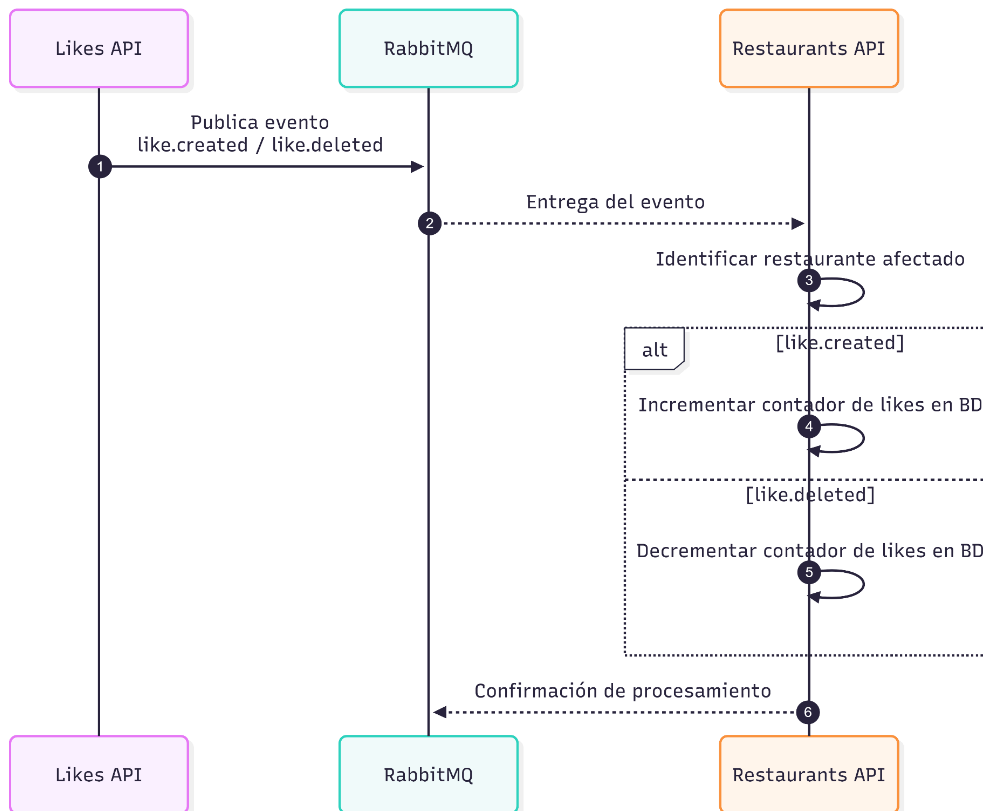

Ilustración 23: Flujo consumo de los eventos like.created y like.deleted.

Este mecanismo evita consultas innecesarias al servicio de likes y mantiene una coherencia eventual adecuada en sistemas distribuidos.

Integración con Recommendations API

Restaurants API es una de las fuentes de información esenciales para el sistema de recomendaciones.
Sin entrar en los detalles del motor esta API proporciona:

- /all → para obtener el catálogo completo de restaurantes cuando Recommendations API lo requiere.
- /allbyids → para recuperar solo los restaurantes necesarios según el modelo.
- /top → para devolver restaurantes destacados cuando Recommendations API lo requiere.

Esta integración es interna y fluye a través de llamadas HTTP entre microservicios gestionadas por el API Gateway.

5.4.5 Likes API

La Likes API es el microservicio encargado de gestionar las interacciones de los usuarios con los restaurantes mediante la acción de "like".
Su función es almacenar qué usuario ha dado like a qué restaurante y permitir consultar esta información cuando otros servicios la necesiten.
Este servicio constituye la fuente de las preferencias explícitas del usuario y es fundamental para el funcionamiento del motor de recomendaciones.

Además de registrar los likes, la API publica los eventos like.created y like.deleted en RabbitMQ, que permiten informar a Restaurants API de los cambios en el contador de likes sin necesidad de llamadas

directas entre servicios.

Tecnologías y dependencias

- **Base de datos SQL:** almacena la relación usuario–restaurante (likes).
- **RabbitMQ Publisher:** utilizado para publicar los eventos like.created y like.deleted.
- **ASP.NET Core 8:** infraestructura para exponer los endpoints REST.

Endpoints principales

Método	Ruta	Descripción
GET	/api/likes	Devuelve los likes del usuario autenticado en formato paginado (page, pageSize). Usado por el frontend.
GET	/api/likes/all	Devuelve todos los likes del sistema con paginación basada en cursor. Usado por Recommendations API.
POST	/api/likes?restaurantId={id}	Registra un like del usuario autenticado sobre un restaurante.
DELETE	/api/likes?restaurantId={id}	Elimina el like del usuario autenticado.

Obtención de likes (usuario y globales)

Likes API expone dos endpoints de lectura que responden a necesidades distintas pero comparten el mismo enfoque de diseño:

- **GET /likes** → utilizado por el frontend para mostrar los restaurantes que un usuario ha marcado con like.
- **GET /likes/all** → utilizado por Recommendations API para obtener el conjunto completo de likes del sistema.

Ambos endpoints utilizan **paginación**, siguiendo el mismo criterio ya descrito en Restaurants API, **paginación basada en cursor**, que permite procesar grandes volúmenes de datos de forma eficiente y progresiva.

De este modo, Likes API proporciona los datos necesarios sin comprometer rendimiento ni escalabilidad.

Registro y eliminación de likes

La creación y eliminación de un like representan dos acciones opuestas sobre la misma entidad, por lo que comparten prácticamente el mismo flujo interno.

Cuando un usuario añade o elimina un like:

1. El API Gateway valida el token JWT y añade la cabecera X-USERID.
2. Likes API extrae el identificador del usuario desde dicha cabecera.
3. Se valida la existencia (o no) de la relación usuario–restaurante (like).
4. Se crea o elimina el registro correspondiente en la base de datos.
5. Se publica el evento de dominio asociado:
    - like.created cuando se registra un like.
    - like.deleted cuando se elimina.

Estos eventos permiten que Restaurants API actualice el contador total de likes de cada restaurante sin necesidad de llamadas síncronas, manteniendo una coherencia eventual adecuada para un sistema distribuido.

5.4.6 Recommendations API

La Recommendations API constituye el núcleo inteligente del sistema GastroMatch. Su responsabilidad es generar un feed de restaurantes personalizado para cada usuario, teniendo en cuenta tanto sus preferencias explícitas como los patrones de comportamiento del conjunto de usuarios de la plataforma.

El **feed** hace referencia al listado principal de restaurantes recomendados que se presenta al usuario, de forma similar al feed de una red social.

Este microservicio encapsula toda la lógica relacionada con la generación de recomendaciones, manteniéndola separada del resto de APIs encargadas de la gestión de datos (usuarios, restaurantes y likes). De este modo, se evita que la complejidad asociada al motor de recomendación se propague a otros servicios del sistema.

La API no sigue un enfoque CRUD tradicional, sino que actúa como un servicio de cálculo. A partir de los datos proporcionados por otros microservicios, ejecuta distintos modelos de recomendación y combina sus resultados para construir un feed final de restaurantes adaptado a cada usuario.

Tecnologías y dependencias

- **ML.NET:** utilizado para implementar el modelo de Filtrado Colaborativo y Filtrado basado en contenido.
- **Likes API:** fuente de datos de interacciones de usuario.
- **Restaurants API:** proporciona los atributos de los restaurantes necesarios para las recomendaciones.
- **ASP.NET Core 8:** infraestructura utilizada para exponer los endpoints REST.

Endpoints principales

Método	Ruta	Descripción
GET	/api/v1/recommendations/feed	Devuelve el feed de restaurantes recomendado para el usuario autenticado.
POST	/api/v1/recommendations/train	Ejecuta el entrenamiento del modelo de Filtrado Colaborativo.

Datos de entrada al sistema de recomendaciones

La generación del feed de recomendaciones se basa exclusivamente en datos obtenidos de otros microservicios del sistema. La Recommendations API no mantiene información propia de dominio, sino que actúa como un servicio de cálculo que combina distintas fuentes de datos.

Por un lado, la API consume información procedente de **Likes API**, que representa las preferencias explícitas de los usuarios. A partir de este servicio se obtienen:

- Los restaurantes a los que el usuario ha dado like.
- El conjunto global de likes del sistema, necesario para identificar patrones de comportamiento entre usuarios.

Estos datos constituyen la base del filtrado colaborativo, ya que permiten relacionar usuarios con comportamientos similares.

Por otro lado, la Recommendations API obtiene información desde **Restaurants API**, que proporciona los atributos descriptivos de cada restaurante, como el tipo de cocina o el rango de precio. Estos datos se utilizan para analizar similitudes entre restaurantes y construir recomendaciones basadas en contenido.

De este modo, el sistema distingue claramente entre dos tipos de información:

- **Datos de comportamiento**, derivados de las interacciones de los usuarios (likes).
- **Datos de contenido**, asociados a las características propias de los restaurantes.

Esta separación permite que el motor de recomendaciones combine ambas perspectivas de forma flexible y desacoplada, sentando la base para un enfoque híbrido que se detalla en el siguiente apartado.

Estrategia de recomendación (implementación híbrida)

La Recommendations API implementa un sistema de recomendación híbrido en el que se ejecutan de forma independiente dos modelos distintos —filtrado colaborativo y filtrado basado en contenido— y posteriormente se combinan sus resultados para generar el feed final de restaurantes.

La lógica asociada a los modelos de recomendación se ha encapsulado en una librería NuGet propia denominada **GastromatchML**, con el objetivo de aislar la lógica de Machine Learning del resto del sistema y evitar que la complejidad del motor de recomendación se propague a la API. Hablamos sobre esta librería en el **Anexo 1.**

El modelo de Filtrado Colaborativo (CF) se utiliza para asignar una puntuación a cada restaurante en función del comportamiento global de los usuarios. Este modelo se entrena previamente mediante el endpoint /train y, una vez entrenado, se carga en memoria para su uso durante la generación de recomendaciones. En tiempo de ejecución, el modelo recibe como entrada el identificador del usuario y devuelve una estimación de afinidad entre el usuario y los distintos restaurantes.

El modelo de Filtrado Basado en Contenido (CB) no requiere un proceso de entrenamiento previo. En este caso, la Recommendations API calcula dinámicamente la similitud entre restaurantes a partir de sus atributos (tipo de cocina, rango de precio, etc.) y las preferencias del usuario, obtenidas a partir de los restaurantes a los que ha dado like.

Durante la generación del feed, ambos modelos producen una puntuación independiente para cada restaurante candidato. Estas puntuaciones se combinan utilizando pesos fijos, definidos a nivel de configuración, que determinan la contribución de cada modelo al resultado final. De este modo, la API obtiene una puntuación agregada que se utiliza para ordenar los restaurantes recomendados.

Modelo de Filtrado Colaborativo

El entrenamiento del modelo de Filtrado Colaborativo se realiza de forma explícita mediante el endpoint /api/recommendations/train. Este endpoint ejecuta el proceso completo de generación del modelo a partir de los likes almacenados en el sistema.

Cuando se invoca este endpoint, la Recommendations API delega el proceso en el servicio ModelTrainingService, que obtiene el conjunto completo de interacciones usuario–restaurante a través de Likes API. Estos datos se transforman en un conjunto de entrenamiento en el que cada registro representa un like, modelado como una relación positiva entre un usuario y un restaurante.

A partir de estos datos se construye un conjunto de instancias RestaurantRating, donde cada entrada contiene el identificador del usuario, el identificador del restaurante y una etiqueta fija con valor 1, ya que el sistema trabaja únicamente con feedback implícito positivo (likes).

El entrenamiento se realiza utilizando ML.NET y el algoritmo de Matrix Factorization, configurado específicamente para escenarios de recomendación con feedback implícito. Una vez entrenado, el modelo se guarda en disco en formato .zip, utilizando una ruta distinta según el entorno de ejecución (local o Azure), lo que permite su posterior carga durante la generación de recomendaciones.

Este enfoque permite separar claramente el proceso de entrenamiento del proceso de inferencia, evitando el coste computacional de entrenar el modelo en cada petición al endpoint /feed y facilitando el reentrenamiento manual o controlado del sistema cuando sea necesario.

Modelo de Filtrado Basado en Contenido

El modelo de Filtrado Basado en Contenido se implementa como un motor independiente dentro de la librería **GastromatchML**, concretamente a través del componente ContentBasedRecommendationEngine. Este motor no requiere un proceso de entrenamiento previo, ya que las recomendaciones se calculan dinámicamente en tiempo de ejecución a partir de las características de los restaurantes.

El proceso comienza con la construcción de un conjunto de características para cada restaurante candidato. A partir de los datos obtenidos desde Restaurants API, cada restaurante se transforma en una estructura RestaurantFeatures, que incluye atributos relevantes para la recomendación, como la categoría y el rango de precio. Estos atributos representan la información de contenido que define a cada restaurante.

Una vez construidos los conjuntos de datos, el motor aplica transformaciones de **one-hot encoding** sobre las variables categóricas. Estas transformaciones convierten los atributos de categoría y rango de precio en vectores numéricos que pueden ser procesados por el motor de ML.NET. Posteriormente, los vectores generados se concatenan en una única representación que define el perfil de cada restaurante.

El mismo proceso de transformación se aplica tanto a los restaurantes candidatos como a los restaurantes que el usuario ha valorado previamente con like. De este modo, ambos conjuntos quedan representados en el mismo espacio vectorial, lo que permite compararlos de forma directa.

A partir de estas representaciones, el motor calcula la similitud entre cada restaurante candidato y los restaurantes previamente valorados por el usuario utilizando **distancia euclídea**. Para cada candidato, se obtiene una puntuación media de similitud respecto al conjunto de restaurantes con like, lo que permite ordenar los resultados según su cercanía al perfil del usuario.

Finalmente, el motor devuelve un conjunto de identificadores de restaurantes cuya distancia se encuentra por debajo de un umbral definido en la implementación. Estos identificadores representan los restaurantes que el modelo considera similares desde el punto de vista de contenido y que serán utilizados posteriormente en el proceso de combinación híbrida.

Flujo de generación del feed

La generación del feed de recomendaciones se realiza a través del endpoint GET /api/v1/recommendations/feed, que delega toda la lógica en el método GetFeedByUserIdAsync.

Este proceso puede dividirse en una serie de pasos, que permiten construir el feed final:

1. Identificación del usuario

Cuando el cliente solicita el feed, el API Gateway valida previamente el token JWT y añade la cabecera X-USERID. La Recommendations API extrae este identificador desde el HttpContext y lo utiliza como entrada principal del sistema de recomendación.

Si el identificador no está presente o es inválido, la API devuelve una respuesta 401 Unauthorized, evitando ejecutar cualquier lógica adicional.

2. Obtención de restaurantes candidatos

El segundo paso del proceso consiste en obtener el conjunto de restaurantes candidatos sobre los que se aplicarán los modelos de recomendación. Para ello, la Recommendations API realiza una llamada a Restaurants API para recuperar el catálogo completo.

Este conjunto actúa como espacio de búsqueda inicial para los modelos colaborativo y basado en contenido.

3. Obtención de los likes del usuario

A continuación, la API consulta Likes API para recuperar los likes asociados al usuario autenticado.
Estos datos cumplen una doble función:

- Determinar si el usuario dispone de información suficiente para generar recomendaciones personalizadas.
- Identificar los restaurantes que deben excluirse del conjunto de candidatos (aquellos que el usuario ya ha marcado con like).

4. Gestión del caso de usuario sin likes

Si el usuario no dispone de likes registrados, el sistema no ejecuta los modelos de recomendación.
En su lugar, se aplica una estrategia que consiste en obtener un listado de restaurantes destacados mediante el endpoint /restaurants/top.

Este mecanismo permite ofrecer un feed válido incluso en escenarios de cold start, garantizando una experiencia coherente para usuarios nuevos o con poca actividad.

5. Filtrado de candidatos

En el caso de usuarios con likes, el sistema elimina del conjunto de candidatos todos aquellos restaurantes que el usuario ya ha valorado positivamente. Este filtrado evita recomendaciones redundantes y asegura que el feed contenga únicamente restaurantes nuevos para el usuario.

6. Ejecución de los motores de recomendación

Una vez filtrado el conjunto de candidatos, la Recommendations API ejecuta de forma independiente los dos motores de recomendación encapsulados en la librería **GastromatchML**:

- El **motor de Filtrado Colaborativo**.
- El **motor de Filtrado Basado en Contenido**.

Cada motor produce su propio conjunto de resultados de forma independiente.

7. Combinación de resultados (modelo híbrido)

Los resultados obtenidos por ambos modelos se combinan utilizando pesos fijos definidos en la implementación:

- El modelo colaborativo aporta un peso del 60 %.
- El modelo basado en contenido aporta un peso del 40 %.

Para cada restaurante candidato se calcula una puntuación final agregada, que se utiliza para ordenar las recomendaciones. Una vez ordenadas, se seleccionan los 10 restaurantes con mayor puntuación, que conforman el feed final.

8. Construcción y devolución del feed

Finalmente, la Recommendations API solicita a Restaurants API la información completa de los restaurantes seleccionados y construye la respuesta final que se devuelve al cliente.

El feed resultante contiene únicamente restaurantes completos, listos para ser representados directamente en el frontend sin necesidad de llamadas adicionales.

El flujo completo de generación del feed de recomendaciones se representa en la Ilustración 24, mostrando la interacción entre el API Gateway, la Recommendations API y los motores de recomendación.

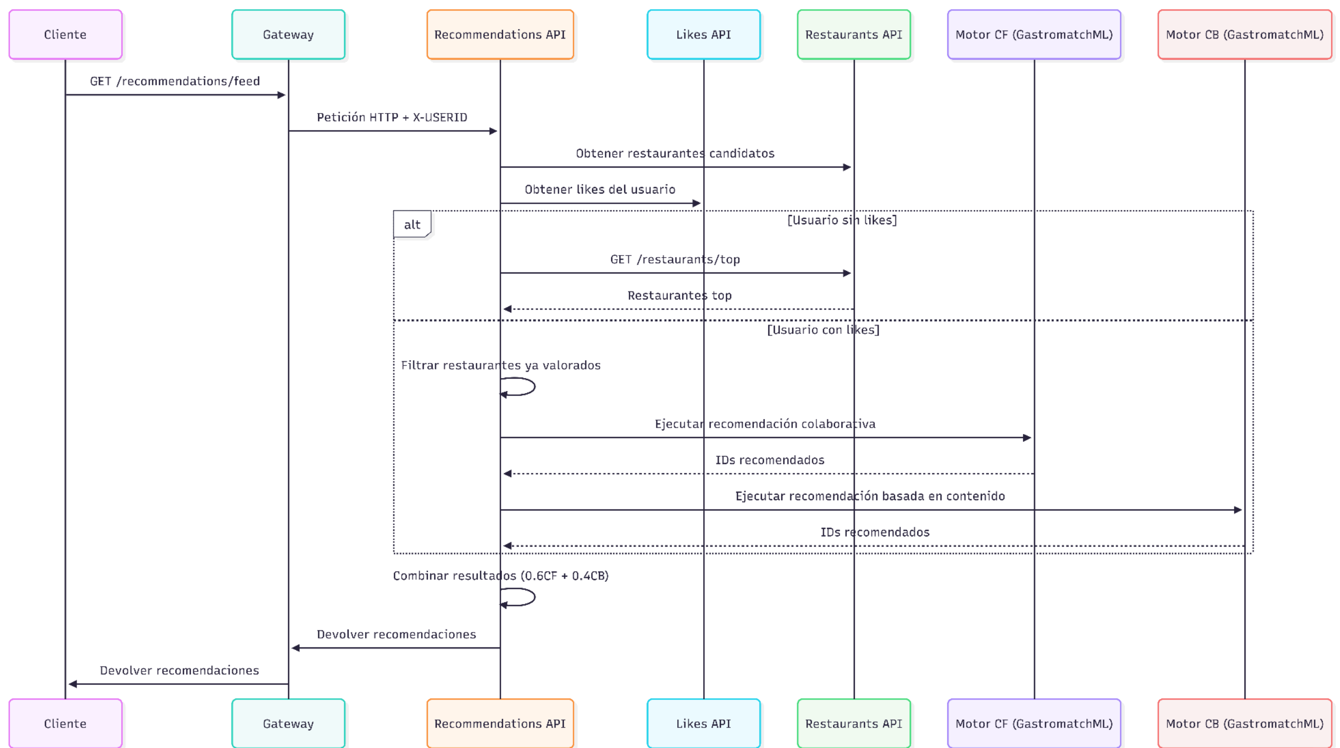

Ilustración 24: Flujo completo de generación de recomendaciones.

6 INFRAESTRUCTURA DEL SISTEMA

Si algo está en tu poder, ¿por qué no hacerlo?

- Marco Aurelio -

6.1 Arquitectura general del despliegue cloud

La infraestructura de **GastroMatch** se ha diseñado y desplegado completamente sobre **Microsoft Azure**, aprovechando sus servicios gestionados para garantizar un entorno **seguro, escalable y automatizado**, capaz de soportar una arquitectura moderna basada en **microservicios**.

El objetivo principal de este despliegue es mantener una **separación clara de responsabilidades** entre los distintos componentes del sistema, permitiendo que cada microservicio se ejecute, actualice y escale de manera independiente, pero dentro de un entorno **coordinado y controlado**.

El ecosistema en la nube de GastroMatch está compuesto por cuatro grandes pilares:

6.1.1 Azure DevOps – Gestión del ciclo de vida del código

Azure DevOps actúa como la **plataforma central de desarrollo y entrega continua (CI/CD)**.
Cada microservicio dispone de su propio **repositorio Git** dentro de **Azure Repos**, desde el cual se ejecutan **pipelines automatizados** que:

- Compilan el código.
- Construyen la imagen Docker correspondiente.
- Publican esa imagen en el **Azure Container Registry (ACR)**.
- Despliegan la nueva versión del servicio en **Azure Container Apps**.

Este flujo permite que cada cambio subido desde Visual Studio al repositorio se propague automáticamente hasta el entorno de producción en Azure sin intervención manual.

6.1.2 Azure Container Apps – Ejecución de microservicios

Cada uno de los microservicios de GastroMatch (Auth, Users, Restaurants, Likes, Recommendations y el Gateway) se ejecuta dentro de una **Container App** independiente.
Este servicio permite ejecutar contenedores Docker en la nube sin necesidad de administrar infraestructura compleja (como clusters de Kubernetes), pero manteniendo funcionalidades esenciales:

- Escalado automático en función de la carga.
- Versionado mediante revisiones.
- Configuración independiente de variables de entorno, secretos y puertos.

- Integración directa con la red virtual privada y con el Container Registry.

Esta arquitectura modular facilita la independencia operativa: si una API necesita reiniciarse, actualizarse o escalar, puede hacerlo sin afectar al resto del sistema.

6.1.3 Virtual Network (VNet) – Comunicación privada y seguridad

Todos los contenedores de la aplicación están desplegados dentro de una **red virtual privada de Azure (VNet)**.
Dentro de esta red se han definido dos tipos de subredes:

- Una **subred pública**, donde se encuentra el **API Gateway (YARP)**, único punto de acceso externo.
- Una **subred privada**, que aloja el resto de APIs y servicios (Auth, Users, Restaurants, Likes, Recommendations y SQL Databases).

El tráfico externo se recibe exclusivamente a través del Gateway, que enruta las peticiones hacia las APIs internas.

6.1.4 Azure Container Registry – Almacenamiento y distribución de imágenes

Azure Container Registry (ACR) se utiliza como repositorio central de las imágenes Docker de todos los microservicios de GastroMatch. Actúa como componente intermedio entre la fase de integración continua y la ejecución en Azure Container Apps.

En cada pipeline de CI/CD, la imagen Docker generada se publica en el registro privado de ACR, desde donde posteriormente es descargada por la Container App correspondiente para su ejecución. Este enfoque garantiza que el artefacto construido durante la fase de integración continua es exactamente el mismo que se ejecuta en producción.

El registro se configura como **privado** y se integra en la Virtual Network mediante **Private Endpoints**, de forma que el acceso a las imágenes se realiza exclusivamente desde el entorno cloud de GastroMatch.

6.1.5 Azure SQL Database – Persistencia independiente por dominio

Cada microservicio cuenta con su propia base de datos en **Azure SQL**, lo que asegura el cumplimiento del principio de **“database per service”** propio de las arquitecturas de microservicios.
Esta separación permite que cada API tenga control total sobre su modelo de datos y facilite su evolución independiente.
Los endpoints de conexión a las bases de datos están limitados a la VNet privada mediante **service endpoints**, de forma que solo las Container Apps del sistema pueden acceder a ellas.

Esquema general del entorno cloud

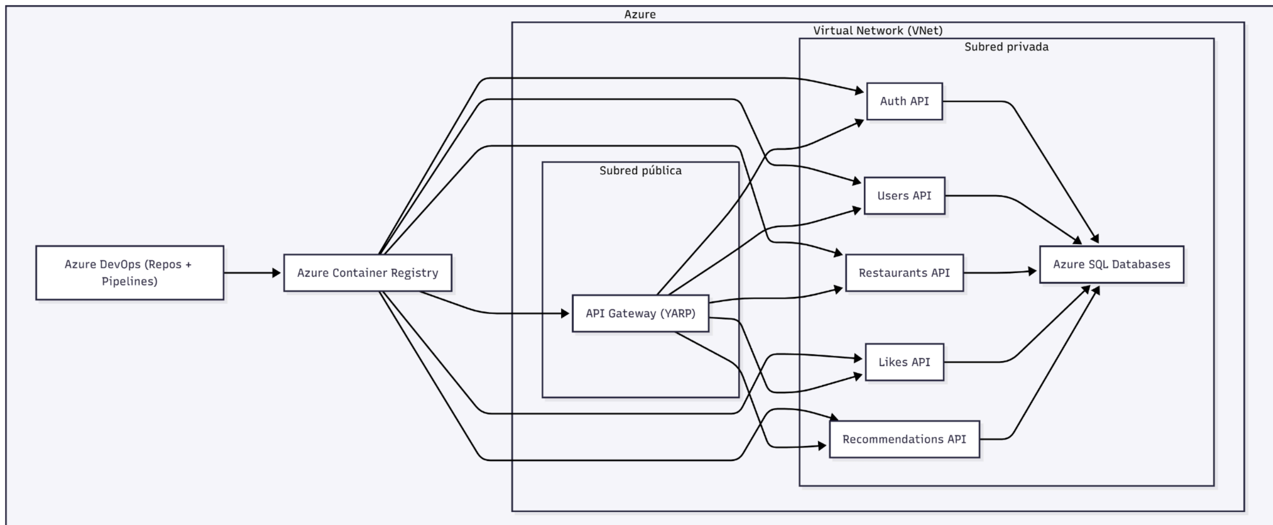

Ilustración 25: Esquema general del entorno cloud

En la Ilustración 25 se muestra el esquema general del entorno cloud de GastroMatch, incluyendo el API Gateway como punto de acceso público, los microservicios desplegados en Azure Container Apps, la red virtual privada y los servicios de persistencia.

La configuración en Azure se basa en los principios de **infraestructura moderna en microservicios**:

- **Desacoplamiento total:** cada API tiene su propio contenedor, base de datos y configuración.
- **Seguridad en capas:** acceso público restringido al Gateway, y comunicaciones internas en red privada.
- **Escalabilidad independiente:** cada servicio puede escalar según su carga sin impactar a los demás.
- **Automatización completa:** cada commit de código genera un nuevo despliegue automático mediante CI/CD.
- **Bajo mantenimiento:** Azure gestiona la infraestructura subyacente (balanceo, escalado, red, etc.).

6.2 Azure DevOps y pipeline de CI/CD

El despliegue de GastroMatch se apoya en un enfoque de **integración y despliegue continuos (CI/CD)**, cuyo objetivo es automatizar el paso del código fuente a producción de forma fiable y repetible. Este enfoque resulta especialmente adecuado en arquitecturas basadas en microservicios, donde múltiples componentes evolucionan de forma independiente y requieren despliegues frecuentes y controlados.

La integración continua permite que cada cambio realizado en el código sea compilado y validado de forma automática, mientras que el despliegue continuo garantiza que las nuevas versiones de los servicios se publiquen en el entorno cloud sin intervención manual. Este modelo reduce errores humanos y acelera el ciclo de desarrollo, tal y como se describe en la documentación oficial de Azure DevOps [4].

En GastroMatch, esta estrategia se implementa mediante **Azure DevOps**, que actúa como plataforma central para la gestión del código y la ejecución de los pipelines de automatización.

6.2.1 Azure DevOps como plataforma de automatización

Cada microservicio del sistema dispone de su propio repositorio Git en **Azure Repos**, siguiendo el principio de independencia entre servicios. De este modo, cada API puede evolucionar, versionarse y desplegarse sin afectar al resto del sistema.

Azure DevOps se utiliza como motor de pipelines, encargándose de ejecutar de forma automática las tareas necesarias para compilar, contenerizar y desplegar cada microservicio en el entorno cloud. Todos los pipelines comparten una estructura común, variando únicamente los parámetros asociados al servicio de destino, como el nombre de la imagen Docker o la Container App asociada.

6.2.2 Pipeline de despliegue de las APIs

El pipeline de despliegue de las APIs se activa automáticamente ante cualquier cambio en la rama principal del repositorio. El flujo seguido es el siguiente:

1. **Obtención del código fuente** desde Azure Repos.
2. **Construcción del proyecto** utilizando el SDK de .NET.
3. **Generación de la imagen Docker** mediante un Dockerfile multistage, separando la fase de compilación de la imagen final de ejecución.
4. **Publicación de la imagen** en Azure Container Registry (ACR).
5. **Despliegue automático** de la nueva versión en Azure Container Apps, asociada al microservicio correspondiente.

Este proceso permite que cada API (Auth, Users, Restaurants, Likes, Recommendations y Gateway) se despliegue de forma independiente, manteniendo una infraestructura modular y fácilmente escalable.

En la Ilustración 26 se muestra un fragmento del pipeline utilizado para el despliegue de las APIs, donde puede observarse la tarea encargada de construir y publicar la imagen Docker en el registro y desplegarla en la Container App de destino.

```
 5  trigger:
 6  - main
 7
 8  resources:
 9  - repo: self
10
11  pool: Local
12
13  variables:
14    # Container registry service connection established during pipeline creation
15    azureSubscription: 'fbfe21cb-f68c-41ad-9284-0dc23b585941'
16    dockerRegistryServiceConnection: 'gastromatch'
17    imageRepository: 'restaurantapi'
18    resourceGroup: 'Gastromatch'
19    containerAppName: 'restaurantapi'
20    containerRegistry: 'gastromatch.azurecr.io'
21    appSourcePath: '$(Build.SourcesDirectory)/src'
22    tag: '$(Build.BuildId)'
23
24  stages:
25  - stage: Build
26    displayName: Build and push stage
27    jobs:
28    - job: Build
29      displayName: Build & Deploy
30      steps:
31      - task: AzureContainerApps@1
32        displayName: Build and push an image to container registry
33        inputs:
34          azureSubscription: '$(azureSubscription)'
35          acrName: '$(dockerRegistryServiceConnection)'
36          acrUsername: 'gastromatch'
37          acrPassword: '0rwEd4c2/EJL6UY79C5LVov1Nov3erpoz6wSGg5lpl+ACRBhc+0C'
38          appSourcePath: '$(appSourcePath)'
39          containerAppName: '$(containerAppName)'
40          resourceGroup: '$(resourceGroup)'
41          imageToBuild: '$(containerRegistry)/$(imageRepository):$(tag)'
42          imageToDeploy: '$(containerRegistry)/$(containerAppName):$(tag)'
```

Ilustración 26: Pipeline para despliegue de las APIs.

6.2.3 Pipeline de publicación de librerías NuGet

Además de las APIs, GastroMatch cuenta con librerías NuGet propias, como **GastromatchML** y la librería de integración con RabbitMQ, que encapsulan lógica común utilizada por distintos microservicios.

Estas librerías disponen de pipelines independientes que automatizan su compilación y publicación, permitiendo que cualquier microservicio pueda consumir nuevas versiones sin necesidad de duplicar código. Este enfoque refuerza la reutilización, el desacoplamiento y la coherencia técnica del sistema.

Feed privado de Azure Artifacts

Azure Artifacts proporciona un **feed privado de paquetes** que permite almacenar y distribuir librerías internas de forma controlada dentro de una organización. En GastroMatch, este mecanismo se utiliza para publicar y versionar las librerías NuGet propias del sistema, que posteriormente son consumidas por los distintos microservicios.

El uso de un feed privado garantiza el control de versiones y el acceso restringido a los paquetes, evitando dependencias externas innecesarias y facilitando la reutilización de código entre servicios. La configuración y el funcionamiento detallado de este feed, así como su integración en los pipelines de CI/CD, se describen con mayor profundidad en el **Anexo 3**.

Esquema del flujo de despliegue

Para facilitar la comprensión del proceso completo, en la **Ilustración 27** se representa el flujo de subida de código desde el entorno de desarrollo hasta su despliegue en producción, mostrando la relación entre Visual Studio, Azure DevOps, Azure Container Registry y Azure Container Apps.

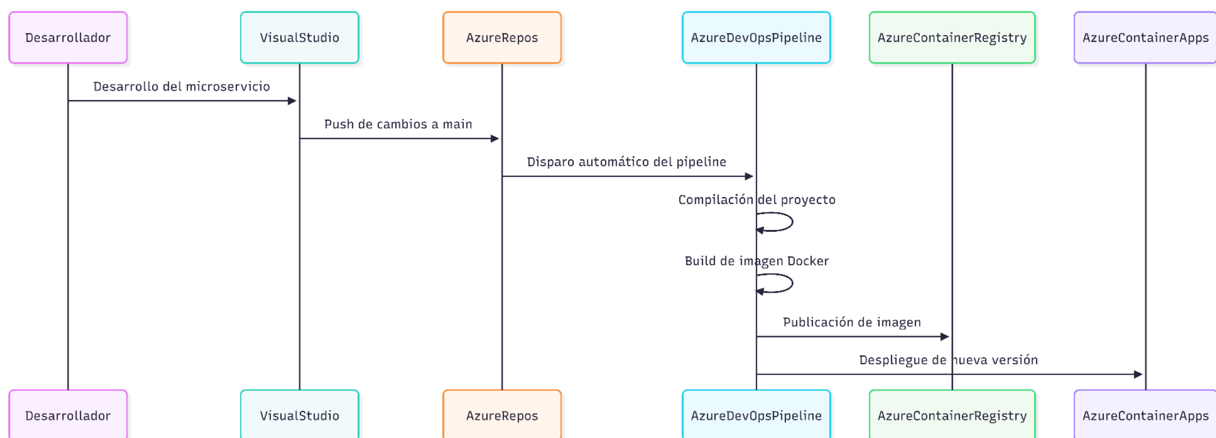

Ilustración 27: Esquema del flujo de despliegue.

6.3 Azure Container Apps como plataforma de ejecución

Azure Container Apps es el servicio utilizado para la **ejecución de todos los microservicios de GastroMatch en el entorno cloud**. Esta plataforma permite desplegar y ejecutar contenedores Docker de forma gestionada, sin necesidad de administrar directamente infraestructura compleja como clústeres de Kubernetes, pero manteniendo características clave para arquitecturas modernas basadas en microservicios [6].

En GastroMatch, cada microservicio (Auth, Users, Restaurants, Likes, Recommendations y API Gateway) se despliega como una **Container App independiente**, lo que refuerza el principio de aislamiento entre servicios y permite su evolución y escalado de forma autónoma.

6.3.1 Modelo de ejecución basado en contenedores

Cada Container App ejecuta una imagen Docker previamente generada y publicada en Azure Container Registry mediante los pipelines de CI/CD descritos en el apartado anterior. Este enfoque garantiza que el mismo artefacto construido durante la fase de integración continua es el que se ejecuta finalmente en producción, eliminando discrepancias entre entornos.

Las Container Apps actúan como unidades de despliegue autosuficientes, configuradas con:

- Su propia imagen Docker.
- Variables de entorno específicas.
- Puertos de exposición.
- Dependencias externas.

Este modelo simplifica la gestión del sistema y permite tratar cada microservicio como un componente completamente independiente.

6.3.2 Escalado y versionado de los servicios

Azure Container Apps proporciona mecanismos de **escalado automático**, permitiendo que cada microservicio ajuste dinámicamente el número de instancias en función de la carga. Aunque en el contexto de este proyecto el escalado no es el foco principal, su disponibilidad garantiza que la arquitectura está preparada para escenarios de crecimiento futuro.

Además, el servicio gestiona el **versionado mediante revisiones**, de modo que cada despliegue genera una nueva versión del microservicio. Esto permite:

- Desplegar nuevas versiones sin interrumpir el servicio.
- Realizar rollbacks controlados si fuera necesario.
- Mantener un historial de cambios por servicio.

6.3.3 Configuración y aislamiento entre microservicios

Cada Container App dispone de su propia configuración independiente, incluyendo variables de entorno y secretos. Esta separación evita configuraciones compartidas y refuerza el desacoplamiento entre microservicios.

Desde el punto de vista de red, las Container Apps de GastroMatch se integran dentro de la **Virtual Network privada**, lo que permite la comunicación interna segura entre servicios y restringe el acceso externo únicamente al API Gateway, tal y como se describirá en el apartado siguiente.

Este aislamiento garantiza que:

- Los microservicios internos no están expuestos directamente a Internet.
- La comunicación entre servicios se produce exclusivamente dentro del entorno controlado.
- Cada servicio puede actualizarse o reiniciarse sin afectar al resto del sistema.

Tal y como se muestra en la **Ilustración 28**, el sistema GastroMatch se encuentra desplegado en Azure mediante múltiples Container Apps, donde cada microservicio se ejecuta como una aplicación contenedora independiente dentro de un mismo entorno gestionado. Esta vista permite comprobar que todos los servicios que componen la arquitectura se encuentran correctamente desplegados y operativos en el entorno cloud.

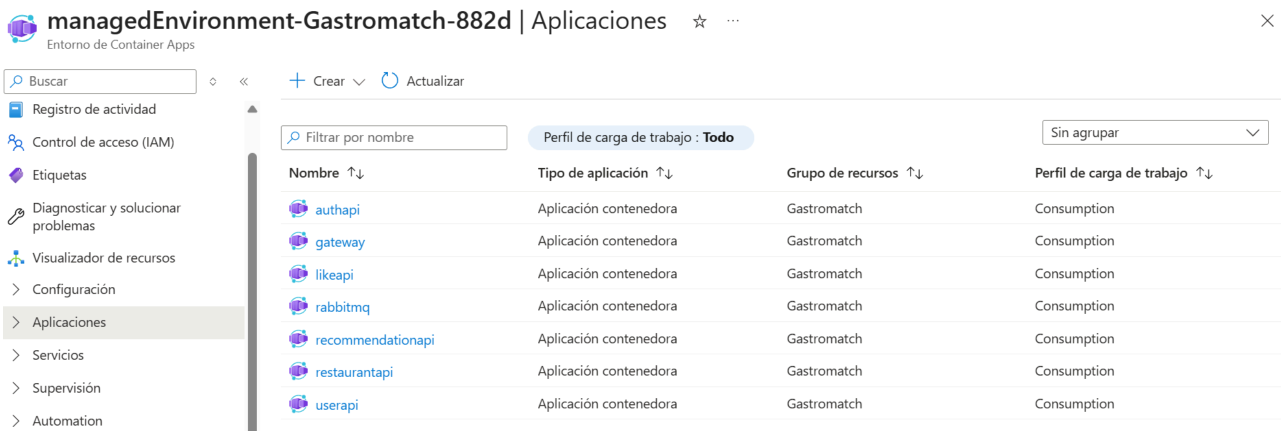

Ilustración 28: Panel de Azure Container Apps mostrando los microservicios de GastroMatch desplegados y en ejecución en el entorno cloud.

6.4 Azure Container Registry

Azure Container Registry (ACR) es el servicio utilizado en GastroMatch para almacenar y distribuir las imágenes Docker de todos los microservicios del sistema. Actúa como repositorio central de artefactos de despliegue y constituye el nexo entre la fase de integración continua y la fase de ejecución en Azure Container Apps.

En el flujo de despliegue de GastroMatch, cada pipeline de CI/CD construye la imagen Docker correspondiente a un microservicio y la publica en el registro privado de ACR. Posteriormente, Azure Container Apps obtiene estas imágenes directamente desde el registro para ejecutar cada servicio en el entorno cloud.

Este enfoque garantiza que:

- Las imágenes ejecutadas en producción son exactamente las mismas que se han construido durante el proceso de integración continua.
- Se mantiene una trazabilidad completa entre código fuente, imagen Docker y despliegue.
- No se introducen dependencias manuales ni artefactos intermedios.

Registro privado y control de acceso

El Azure Container Registry de GastroMatch se configura como un **registro privado**, lo que implica que las imágenes no son accesibles públicamente. El acceso al registro queda restringido a:

- Los pipelines de Azure DevOps encargados de publicar las imágenes.
- Las Container Apps autorizadas para descargar y ejecutar dichas imágenes.

Integración con la red virtual

Tal y como se detalla en el apartado de red, el acceso al Azure Container Registry se realiza mediante un **Private Endpoint** integrado en la Virtual Network del sistema. Esto permite que tanto los procesos de despliegue como las Container Apps descarguen imágenes sin salir de la red privada, eliminando tráfico hacia Internet y reduciendo la superficie de ataque.

De este modo, Azure Container Registry se integra de forma transparente en la arquitectura segura de GastroMatch, actuando como componente intermedio entre el sistema de CI/CD y la plataforma de ejecución.

6.5 Virtual Network (VNet) y comunicaciones privadas

La infraestructura de red de GastroMatch se apoya en una **Virtual Network (VNet)** de Azure, diseñada para garantizar la comunicación privada entre los distintos componentes del sistema y restringir al máximo la exposición a Internet de los servicios internos [7].

Tal y como se ha introducido en la arquitectura general del despliegue (Ilustración 26), todos los recursos críticos del sistema se integran dentro de una misma red virtual, denominada `vnet-gastromatch`, que actúa como perímetro de seguridad principal del entorno cloud.

6.5.1 Diseño de la red virtual

La red virtual `vnet-gastromatch` se ha configurado con un **espacio de direcciones IPv4 10.40.0.0/16**, lo que proporciona un rango amplio para el crecimiento futuro del sistema sin necesidad de rediseñar la topología de red.

Dentro de esta VNet se han definido **dos subredes claramente diferenciadas**, cada una con una responsabilidad concreta:

- **Subred pública (`vnet-aca`, 10.40.1.0/24):** Destinada exclusivamente a los componentes que requieren exposición controlada, en concreto:
  - API Gateway (YARP)
  - Los puntos de conexión privados (Private Endpoints) utilizados para acceder a servicios PaaS como Azure Container Registry y Azure SQL Database.
- **Subred privada:** Reservada para el resto de microservicios del sistema:
  - Auth API
  - Users API
  - Restaurants API
  - Likes API
  - Recommendations API

Esta separación permite aplicar una política de **seguridad en capas**, donde únicamente el Gateway actúa como punto de entrada al sistema.

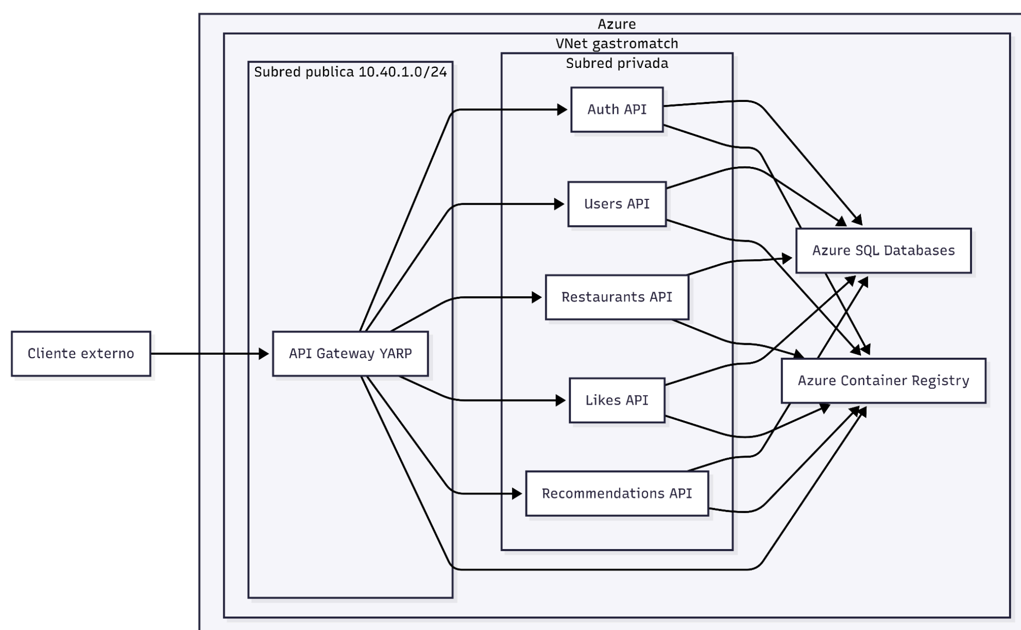

Ilustración 29: Topología de red completa.

En la Ilustración 29 se muestra la topología de red completa del sistema, donde puede observarse la separación entre subred pública y privada, el API Gateway como único punto de acceso externo y el uso de endpoints privados para la comunicación con Azure SQL Database y Azure Container Registry.

6.5.2 Aislamiento de los microservicios

Todos los microservicios de GastroMatch se ejecutan como **Azure Container Apps integradas en la VNet**, lo que implica que:

- No disponen de direcciones IP públicas.
- Solo pueden comunicarse entre sí a través de la red privada.
- El acceso externo queda completamente bloqueado, salvo a través del API Gateway.

Este enfoque reduce drásticamente la superficie de ataque del sistema y asegura que ningún microservicio pueda ser invocado directamente desde Internet.

6.5.3 Acceso privado a Azure SQL Database

Cada microservicio cuenta con su propia base de datos en **Azure SQL Database**, cumpliendo el principio de *database per service*. El acceso a estas bases de datos se ha configurado mediante **Private Endpoints [8]**, lo que permite que la comunicación entre las Container Apps y Azure SQL se realice exclusivamente dentro de la VNet.

Mediante este mecanismo, cada base de datos queda asociada a una **interfaz de red privada con dirección IP interna**, integrada en la VNet de GastroMatch. Como consecuencia, el servicio de base de datos deja de exponer endpoints públicos accesibles desde Internet y pasa a comportarse, desde el punto de vista de red, como un recurso interno del sistema.

Este mecanismo garantiza que:

- Las bases de datos no exponen endpoints públicos.
- Solo los recursos desplegados dentro de `vnet-gastromatch` pueden acceder a ellas.
- Se elimina la necesidad de reglas de firewall basadas en IP públicas.

6.5.4 Integración con Azure Container Registry

El **Azure Container Registry (ACR)** utilizado para almacenar las imágenes Docker también se ha integrado en la VNet mediante un **Private Endpoint**.
De este modo, tanto los pipelines de despliegue como las Container Apps pueden acceder al registro de imágenes sin salir de la red privada.

Este diseño evita la exposición del registro a Internet y asegura que únicamente los recursos autorizados dentro del entorno de GastroMatch puedan descargar imágenes.

Justificación del diseño de red

La configuración de la red virtual responde a los siguientes objetivos arquitectónicos:

- **Seguridad**: eliminación de accesos públicos innecesarios y aislamiento de los servicios internos.
- **Control del tráfico**: todo el tráfico entrante pasa por el API Gateway.
- **Cumplimiento de buenas prácticas cloud**: uso de Private Endpoints y comunicación interna por VNet.

Este diseño permite que GastroMatch opere como un sistema distribuido seguro, donde cada microservicio se comunica únicamente con los componentes estrictamente necesarios, manteniendo un alto nivel de control y robustez en el despliegue cloud.

6.6 Azure SQL Database – Persistencia de datos

Azure SQL Database es el servicio utilizado en GastroMatch para la persistencia de los datos de cada microservicio. Se trata de una base de datos relacional gestionada que permite almacenar información de forma segura y escalable sin necesidad de administrar directamente servidores o infraestructura subyacente.

El uso de un servicio PaaS para la capa de persistencia permite centrar el desarrollo en la lógica de negocio, delegando en Azure aspectos como la disponibilidad, las copias de seguridad, el parcheado y la monitorización básica del sistema de bases de datos.

6.6.1. Modelo de persistencia por microservicio

Siguiendo los principios de las arquitecturas de microservicios, GastroMatch aplica el patrón **database per service**, de modo que cada microservicio dispone de su propia base de datos independiente en Azure SQL.
Esta separación implica que:

- Cada API es la única responsable de su modelo de datos.
- No existen accesos directos a bases de datos entre microservicios.
- La evolución del esquema de una base de datos no afecta al resto del sistema.

Este enfoque refuerza el desacoplamiento entre dominios y evita dependencias cruzadas a nivel de persistencia, favoreciendo la mantenibilidad y la escalabilidad del sistema.

6.6.2 Integración con la red privada

El acceso a Azure SQL Database se realiza exclusivamente a través de **Private Endpoints integrados en la Virtual Network** del sistema.
Esto implica que las bases de datos no exponen endpoints públicos accesibles desde Internet y que únicamente los recursos desplegados dentro de la VNet de GastroMatch pueden establecer conexión con ellas.

Tal y como se describe en el apartado de red, cada base de datos queda asociada a una interfaz de red con dirección IP privada dentro del entorno cloud, garantizando que toda la comunicación entre las Container Apps y la capa de persistencia se realiza de forma interna y segura.

Este diseño elimina la necesidad de reglas de firewall basadas en direcciones IP públicas y refuerza el aislamiento del sistema frente a accesos no autorizados.

6.6.3 Justificación del uso de Azure SQL Database

La elección de Azure SQL Database frente a otras alternativas responde a varios factores clave:

- **Compatibilidad con el modelo relacional**, adecuado para los dominios gestionados (usuarios, restaurantes, likes).
- **Alta disponibilidad y fiabilidad**, gestionadas de forma automática por la plataforma.
- **Integración nativa con Azure**, especialmente con Virtual Network y Private Endpoints.

Gracias a esta solución, GastroMatch dispone de una capa de persistencia robusta, segura y alineada con el enfoque distribuido del sistema, completando la arquitectura cloud definida en este capítulo.

7 Conclusiones y Futuras mejoras

El verdadero enemigo no es la máquina, sino la mente que deja de pensar.

- George Orwell -

7.1 Conclusiones del trabajo realizado

En este Trabajo Fin de Grado se ha diseñado e implementado un sistema completo de recomendación de restaurantes, denominado **GastroMatch**, que integra una arquitectura moderna basada en microservicios con técnicas de aprendizaje automático aplicadas a un entorno real de ejecución en la nube.

El sistema desarrollado no se limita a un prototipo aislado, sino que constituye una solución funcional de extremo a extremo, en la que se han definido y desplegado múltiples microservicios independientes, cada uno con responsabilidades bien delimitadas y su propia persistencia de datos. Esta aproximación permite validar en la práctica los principios teóricos de las arquitecturas distribuidas, demostrando su aplicabilidad en un escenario realista.

A nivel arquitectónico, el proyecto pone en práctica conceptos clave como **Clean Architecture** y **Domain-Driven Design**, asegurando una separación clara entre la lógica de dominio, la lógica de aplicación y los detalles de infraestructura. Esta estructura facilita la mantenibilidad del código, la evolución independiente de los servicios y la comprensión global del sistema. Asimismo, la incorporación de un API Gateway como punto único de entrada refuerza la seguridad y el desacoplamiento entre el cliente y la topología interna de las APIs.

Desde el punto de vista de la comunicación entre servicios, GastroMatch combina de forma coherente mecanismos síncronos y asíncronos. El uso de peticiones HTTP permite obtener información actualizada cuando es necesario responder al usuario, mientras que la comunicación basada en eventos favorece el desacoplamiento y la coherencia eventual entre dominios.

Uno de los aspectos más relevantes del trabajo es la integración de un **motor de recomendaciones híbrido**, que combina técnicas de Filtrado Colaborativo y Filtrado Basado en Contenido. Lejos de limitarse a una descripción teórica, el proyecto implementa ambos enfoques de forma práctica mediante la librería ML.NET, separando claramente los procesos de entrenamiento e inferencia y encapsulando la lógica de aprendizaje automático en componentes reutilizables. Este diseño permite que el sistema genere recomendaciones personalizadas de manera eficiente y extensible.

En cuanto al despliegue, el proyecto demuestra la viabilidad de ejecutar una arquitectura distribuida en un entorno cloud gestionado, utilizando contenedores Docker, pipelines de integración y despliegue continuo y una infraestructura de red privada. El uso de servicios como Azure Container Apps, Azure Container Registry y Azure SQL Database permite validar que la solución es escalable, segura y preparada para su evolución futura, reduciendo al mismo tiempo la carga operativa asociada a la gestión de la infraestructura.

En conjunto, GastroMatch demuestra que es posible integrar de forma coherente conceptos avanzados de ingeniería de software, computación en la nube y aprendizaje automático en una solución realista y funcional. El trabajo realizado valida que los conocimientos adquiridos a lo largo del grado pueden aplicarse de manera efectiva en el diseño y desarrollo de sistemas complejos, alineados con las prácticas y tecnologías actualmente utilizadas en la industria del software.

7.2 Evaluación de los objetivos planteados

Al inicio de este Trabajo Fin de Grado se definieron una serie de objetivos generales y específicos orientados al diseño y desarrollo de un sistema de recomendación de restaurantes basado en una arquitectura moderna y escalable. A continuación, se evalúa el grado de cumplimiento de dichos objetivos a la luz del trabajo realizado.

El objetivo principal del proyecto consistía en diseñar e implementar una plataforma capaz de generar recomendaciones personalizadas de restaurantes mediante el uso de técnicas de aprendizaje automático, integradas dentro de una arquitectura de microservicios. Este objetivo se ha cumplido satisfactoriamente, ya que el sistema desarrollado permite generar un feed personalizado para cada usuario combinando información de comportamiento y características de contenido, todo ello integrado en un entorno distribuido y funcional.

En cuanto a los objetivos relacionados con la arquitectura del sistema, se planteó la adopción de un enfoque basado en microservicios que favoreciera el desacoplamiento, la escalabilidad y la mantenibilidad. La implementación de APIs independientes, cada una con su propio dominio y persistencia de datos, junto con el uso de un API Gateway y mecanismos de comunicación síncrona y asíncrona, demuestra el cumplimiento de este objetivo y valida la viabilidad de este enfoque en un caso práctico.

Otro de los objetivos relevantes era aplicar principios de diseño como Clean Architecture y Domain-Driven Design para estructurar el código de forma clara y sostenible. La organización interna de las APIs en capas bien definidas y la separación entre lógica de negocio e infraestructura evidencian que este objetivo se ha alcanzado, facilitando la comprensión del sistema y su evolución futura.

Asimismo, se planteó como objetivo desplegar el sistema en un entorno cloud real, utilizando servicios gestionados que permitieran automatizar el ciclo de vida de las aplicaciones. La utilización de pipelines de CI/CD, contenedores Docker, Azure Container Apps, red privada y bases de datos gestionadas confirma que el sistema no solo es funcional a nivel local, sino que está preparado para su ejecución en un entorno de producción.

Por último, el proyecto tenía como objetivo servir como ejercicio integrador de los conocimientos adquiridos durante el grado, combinando desarrollo backend, arquitectura de software, computación en la nube y aprendizaje automático. El resultado final demuestra que estos conocimientos han sido aplicados de forma conjunta y coherente, dando lugar a una solución completa y alineada con prácticas actuales de la industria.

En conclusión, los objetivos planteados al inicio del proyecto han sido alcanzados de manera satisfactoria, sentando una base sólida tanto a nivel técnico como conceptual para futuras ampliaciones y mejoras del sistema.

7.3 Limitaciones del sistema

A pesar de que el sistema desarrollado cumple los objetivos planteados y constituye una solución funcional y coherente, es importante identificar una serie de limitaciones derivadas tanto del alcance del proyecto como de las decisiones de diseño adoptadas durante su desarrollo. Estas limitaciones no invalidan la solución propuesta, sino que delimitan su contexto de aplicación y permiten identificar oportunidades claras de mejora.

Una de las principales limitaciones del sistema se encuentra en el **proceso de entrenamiento del modelo de Filtrado Colaborativo**. Actualmente, el entrenamiento se realiza de forma explícita mediante un endpoint dedicado, lo que implica que el modelo no se actualiza automáticamente ante nuevas interacciones de los usuarios. Esta decisión se ha tomado para simplificar el control del proceso de entrenamiento y evitar el coste computacional asociado a reentrenamientos frecuentes, pero introduce una dependencia manual que podría no ser adecuada en entornos con alta variabilidad de datos.

Otra limitación relevante está relacionada con la **estrategia de combinación del modelo híbrido**. En la implementación actual, los resultados del Filtrado Colaborativo y del Filtrado Basado en Contenido se combinan utilizando pesos fijos. Aunque este enfoque resulta sencillo y efectivo para validar el funcionamiento del sistema, no permite adaptar dinámicamente la importancia de cada modelo en función del perfil del usuario, del volumen de datos disponibles o del contexto de uso.

Desde el punto de vista de la evaluación del sistema de recomendaciones, el proyecto no incorpora **métricas cuantitativas de calidad** tales como precisión, recall o métricas de ranking. La validación del sistema se ha centrado en su correcto funcionamiento y coherencia lógica, dejando fuera un análisis empírico más profundo que requeriría conjuntos de datos más amplios y escenarios de uso prolongados.

En cuanto a la observabilidad del sistema, la solución no incluye actualmente **mecanismos avanzados de monitorización y trazabilidad**, como sistemas centralizados de logging, métricas de rendimiento o health checks específicos por servicio. Aunque la arquitectura está preparada para integrar este tipo de herramientas, su implementación queda fuera del alcance del presente trabajo.

Por último, el proyecto se ha centrado principalmente en el desarrollo del backend y de la infraestructura cloud, por lo que **no se ha implementado un frontend visual completo** que permita interactuar con el sistema de forma directa. La validación de las APIs se ha realizado mediante herramientas de prueba y clientes REST, lo que resulta suficiente a nivel técnico, pero limita la evaluación de la experiencia de usuario final.

En conjunto, estas limitaciones responden a decisiones conscientes orientadas a acotar el alcance del proyecto y priorizar el diseño arquitectónico y la integración del motor de recomendaciones. No obstante, todas ellas abren la puerta a futuras mejoras que permitirían evolucionar el sistema hacia un entorno más automatizado, evaluable y orientado a producción.

7.4 Líneas de trabajo futuro

A partir del sistema desarrollado y de las limitaciones identificadas, se abren diversas líneas de trabajo futuro que permitirían evolucionar GastroMatch hacia una solución más completa, automatizada y cercana a un entorno de producción real.

Una primera línea de evolución se centra en la **automatización del proceso de entrenamiento del modelo de Filtrado Colaborativo**. Actualmente, el reentrenamiento del modelo se realiza de forma manual mediante un endpoint específico. Como mejora futura, podría incorporarse un mecanismo basado en eventos, de modo que la creación de nuevas interacciones (`like.created`) desencadene de forma controlada el reentrenamiento o la actualización incremental del modelo. Este enfoque permitiría adaptar las recomendaciones de manera más dinámica al comportamiento de los usuarios, manteniendo la calidad del sistema a medida que crece el volumen de datos.

Otra línea de mejora relevante está relacionada con la **optimización del modelo híbrido de recomendaciones**. En lugar de emplear pesos fijos para combinar los resultados del Filtrado Colaborativo y del Filtrado Basado en Contenido, el sistema podría evolucionar hacia una estrategia de ponderación dinámica. Esta permitiría ajustar automáticamente la contribución de cada modelo en función de factores como la cantidad de interacciones del usuario, su antigüedad en la plataforma o la diversidad de sus preferencias, mejorando la personalización del feed.

Desde el punto de vista de la evaluación del sistema, una mejora futura consistiría en la **incorporación de métricas cuantitativas** para medir la calidad de las recomendaciones. La inclusión de métricas como precisión, recall o métricas de ranking permitiría analizar el rendimiento del sistema de forma objetiva y comparar distintas configuraciones del modelo, facilitando una mejora continua basada en datos.

En relación con la infraestructura y la operación del sistema, otra línea de trabajo futuro sería la **implantación de mecanismos avanzados de observabilidad**. La integración de sistemas de logging centralizado, métricas de rendimiento y health checks por microservicio permitiría supervisar el comportamiento del sistema en tiempo real, detectar incidencias de forma temprana y facilitar tareas de mantenimiento y escalado.

Asimismo, el desarrollo de un **frontend visual completo** representa una línea de evolución natural del proyecto. La implementación de una interfaz web o móvil permitiría evaluar la experiencia de usuario final, validar la utilidad real de las recomendaciones generadas y cerrar el ciclo completo de interacción entre usuarios y sistema.

Por último, el sistema podría ampliarse incorporando **nuevas fuentes de datos y señales de preferencia**, como valoraciones numéricas, historial de búsquedas o interacciones implícitas adicionales. Estas mejoras enriquecerían el conjunto de información disponible para el motor de recomendaciones y permitirían generar sugerencias aún más precisas y contextualizadas.

En conjunto, estas líneas de trabajo futuro reflejan que GastroMatch constituye una base sólida y extensible, preparada para evolucionar progresivamente hacia un sistema más avanzado, automatizado y orientado a un uso real en producción.

7.5 Valoración personal del proyecto

El desarrollo de este Trabajo Fin de Grado ha supuesto una experiencia especialmente enriquecedora desde el punto de vista técnico y formativo, al permitir integrar de manera práctica conocimientos adquiridos a lo largo del grado en un proyecto coherente y de alcance realista.

Uno de los principales aprendizajes derivados del proyecto ha sido la **importancia del diseño arquitectónico previo**. La definición temprana de una arquitectura basada en microservicios, junto con la adopción de principios como Clean Architecture y Domain-Driven Design, ha resultado clave para mantener el control del sistema a medida que aumentaba su complejidad. Este enfoque ha facilitado la organización del código, la separación de responsabilidades y la toma de decisiones fundamentadas durante el desarrollo.

Asimismo, el proyecto ha permitido profundizar en el **despliegue de sistemas distribuidos en la nube**, abordando aspectos que van más allá del desarrollo puramente funcional, como la automatización del ciclo de vida del software, la seguridad en red, la gestión de contenedores y la integración de servicios gestionados. La utilización de un entorno cloud real ha aportado una visión más cercana a la práctica profesional y ha puesto de manifiesto la relevancia de considerar la infraestructura como parte integral del diseño del sistema.

Otro aspecto especialmente relevante ha sido la **implementación práctica de un sistema de recomendaciones**. La integración de técnicas de aprendizaje automático dentro de una arquitectura distribuida ha permitido comprender mejor las implicaciones reales de estos modelos en términos de entrenamiento, inferencia, rendimiento y mantenimiento. Este enfoque ha reforzado la idea de que el valor del machine learning no reside únicamente en el modelo en sí, sino en su correcta integración dentro de un sistema de software bien diseñado.

Desde una perspectiva personal, el desarrollo de GastroMatch ha supuesto un ejercicio de consolidación de conocimientos y de adquisición de nuevas competencias, especialmente en lo relativo a la toma de decisiones técnicas, la resolución de problemas complejos y la evaluación crítica de las propias soluciones. El proyecto ha requerido un equilibrio constante entre ambición y viabilidad, obligando a priorizar objetivos y a justificar

conscientemente las limitaciones asumidas.

En conjunto, este Trabajo Fin de Grado ha cumplido su función como proyecto integrador, permitiendo aplicar de forma práctica conceptos avanzados de ingeniería de software, computación en la nube y aprendizaje automático. La experiencia adquirida sienta una base sólida para afrontar futuros proyectos profesionales o académicos relacionados con el diseño y desarrollo de sistemas distribuidos y escalables.

8 ANEXOS

Anexo 1: Librería GastromatchML

A.1.1 Motivación y objetivos de la librería

Durante el desarrollo de GastroMatch se identificó la necesidad de **aislar la lógica de Machine Learning** del resto de la aplicación, evitando que la complejidad asociada a los modelos de recomendación se propagase a la Recommendations API o al dominio del sistema.

Con este objetivo se desarrolló una librería NuGet propia denominada **GastromatchML**, que encapsula toda la lógica relacionada con:

- Entrenamiento del modelo de Filtrado Colaborativo.
- Ejecución del motor de Filtrado Colaborativo.
- Ejecución del motor de Filtrado Basado en Contenido.
- Definición de los modelos de datos utilizados por ML.NET.
- Construcción de los conjuntos de entrenamiento a partir de datos de dominio.

Este enfoque permite:

- Mantener la **separación de responsabilidades** entre cálculo y orquestación.
- Facilitar la **reutilización** de la lógica de ML.
- Reducir el acoplamiento entre la Recommendations API y ML.NET.
- Mejorar la mantenibilidad y extensibilidad del sistema.

La Recommendations API actúa únicamente como **consumidora** de esta librería, delegando en ella la lógica de inferencia y entrenamiento.

A.1.2 Estructura del proyecto GastromatchML

La librería se ha desarrollado como un proyecto **.NET Standard 2.1**, lo que permite su reutilización en diferentes tipos de aplicaciones .NET.

A continuación se describen las principales **clases** que componen este módulo:

- **CollaborativeFilteringTrainer:** Responsable del entrenamiento del modelo de Filtrado Colaborativo.
- **CollaborativeRecommendationEngine:** Encargado de realizar inferencias a partir del modelo entrenado.
- **ContentBasedRecommendationEngine:** Implementa el motor de Filtrado Basado en Contenido.
- **TrainingDataBuilder:** Clase de soporte para transformar datos de dominio en estructuras compatibles con ML.NET.
- **Modelos de datos**:
    - RestaurantRating
    - RestaurantRatingPrediction
    - RestaurantFeatures
    - RestaurantVector
- **GastromatchMLExtensions:** Extensión para facilitar la integración de la librería mediante inyección de dependencias.

A.1.3 Entrenamiento del modelo de Filtrado Colaborativo

El entrenamiento del modelo de Filtrado Colaborativo se encapsula en la clase **CollaborativeFilteringTrainer**. Este componente recibe un conjunto de interacciones usuario–restaurante y genera un modelo entrenado utilizando ML.NET.

El proceso sigue los siguientes pasos:

1. Carga de los datos de entrenamiento en un `IDataView`.
2. Configuración del algoritmo de **Matrix Factorization** con feedback implícito.
3. Entrenamiento del modelo a partir de las interacciones disponibles.
4. Persistencia del modelo entrenado en disco para su reutilización posterior.

El sistema trabaja exclusivamente con **feedback implícito positivo**, representado por los likes de los usuarios. Por este motivo, cada interacción se modela con una etiqueta fija de valor 1, indicando únicamente la existencia de afinidad entre un usuario y un restaurante.

Tal y como se muestra en la **Ilustración 30**, el entrenamiento se realiza mediante la configuración explícita de las opciones del algoritmo `MatrixFactorizationTrainer` proporcionado por [ML.NET](ML.NET).

```
public static class CollaborativeFilteringTrainer
{
    private static readonly MLContext _mlContext = new MLContext();

    public static void Train(IEnumerable<RestaurantRating> data, string modelPath)
    {
        var trainingData = _mlContext.Data.LoadFromEnumerable(data);

        var options = new MatrixFactorizationTrainer.Options
        {
            MatrixColumnIndexColumnName = nameof(RestaurantRating.UserId),
            MatrixRowIndexColumnName = nameof(RestaurantRating.RestaurantId),
            LabelColumnName = nameof(RestaurantRating.Label),
            LossFunction = MatrixFactorizationTrainer.LossFunctionType.SquareLossOneClass,
            Alpha = 0.01,
            Lambda = 0.1,
            NumberOfIterations = 30,
            ApproximationRank = 20
        };

        var pipeline = _mlContext.Recommendation().Trainers.MatrixFactorization(options);
        var model = pipeline.Fit(trainingData);

        _mlContext.Model.Save(model, trainingData.Schema, modelPath);
    }
}
```

Ilustración 30: Configuración y entrenamiento del modelo de Filtrado Colaborativo.

El algoritmo de Matrix Factorization se configura a través del objeto `MatrixFactorizationTrainer.Options`, cuyos parámetros determinan el comportamiento del modelo entrenado:

- **MatrixColumnIndexColumnName:** Indica la columna que representa el identificador del usuario. En este caso, se utiliza `UserId`, que identifica de forma única a cada usuario del sistema.
- **MatrixRowIndexColumnName:** Define la columna que representa el identificador del restaurante.

Se utiliza `RestaurantId`, permitiendo modelar la relación usuario–restaurante.

- **LabelColumnName:** Especifica la columna que contiene la etiqueta de entrenamiento. En este sistema, la etiqueta es siempre 1, ya que se trabaja únicamente con feedback implícito positivo (likes).
- **LossFunction:** Se utiliza la función de pérdida `SquareLossOneClass`, diseñada específicamente para escenarios de recomendación con feedback implícito, donde solo se dispone de interacciones positivas y no de valoraciones negativas explícitas.
- **Alpha:** Parámetro de confianza que controla el peso de las interacciones positivas frente a las no observadas. Un valor bajo como 0.01 permite suavizar la influencia de cada like y evitar sobreajuste.
- **Lambda:** Término de regularización que penaliza modelos excesivamente complejos. Su objetivo es reducir el sobreajuste y mejorar la capacidad de generalización del modelo.
- **NumberOfIterations:** Número de iteraciones del proceso de entrenamiento. En este caso se ha establecido en 30, buscando un equilibrio entre calidad del modelo y coste computacional.
- **ApproximationRank:** Dimensión del espacio latente utilizado para representar usuarios y restaurantes. Un valor de 20 permite capturar patrones relevantes de afinidad sin incrementar en exceso la complejidad del modelo.

Una vez configurado el algoritmo, se construye el pipeline de entrenamiento y se ajusta el modelo a los datos disponibles. El modelo resultante se almacena en disco en formato `.zip`, permitiendo su posterior carga durante la fase de inferencia sin necesidad de reentrenar en cada ejecución.

Este enfoque se apoya en el modelo de recomendación por factorización matricial implementado por ML.NET, ampliamente utilizado en sistemas de recomendación modernos [9], y permite separar claramente el proceso de entrenamiento del proceso de generación de recomendaciones en tiempo de ejecución.

A.1.4 Motor de Filtrado Colaborativo

La ejecución de recomendaciones basadas en filtrado colaborativo se encapsula en la clase `CollaborativeRecommendationEngine`. Este componente es responsable de **cargar el modelo previamente entrenado** y **realizar inferencias en tiempo de ejecución**, a partir de los identificadores de usuario y de los restaurantes candidatos.

Tal y como se muestra en la **Ilustración 31**, el motor se inicializa una única vez y reutiliza el modelo cargado en memoria para generar múltiples predicciones, siguiendo las recomendaciones oficiales de ML.NET para escenarios de inferencia [10].

```
public class CollaborativeRecommendationEngine
{
    private readonly MLContext _mlContext;
    private readonly PredictionEngine<RestaurantRating, RestaurantRatingPrediction> _predictionEngine;

    public CollaborativeRecommendationEngine()
    {
        _mlContext = new MLContext();
        var isAzureApp = Environment.GetEnvironmentVariable("WEBSITE_INSTANCE_ID") != null;
        var model = isAzureApp ? _mlContext.Model.Load(Path.Combine(Path.GetTempPath(), "CollaborativeModel.zip"), out _) :
            _mlContext.Model.Load(Directory.GetCurrentDirectory() + "/CollaborativeModel.zip", out _);
        _predictionEngine = _mlContext.Model.CreatePredictionEngine<RestaurantRating, RestaurantRatingPrediction>(model);
    }

    public List<int> Recommend(int userId, IEnumerable<int> restaurantIds)
    {
        var results = new List<(int Id, float Score)>();

        foreach (var restaurantId in restaurantIds)
        {
            RestaurantRatingPrediction prediction = _predictionEngine.Predict(new RestaurantRating
            {
                UserId = (uint) userId,
                RestaurantId = (uint) restaurantId
            });

            if (prediction.Score > 0.4f)
                results.Add((restaurantId, prediction.Score));
        }

        return results.OrderByDescending(x => x.Score)
                      .Select(x => (int) x.Id)
                      .ToList();
    }
}
```

Ilustración 31: Implementación del modelo de Filtrado Colaborativo.

Inicialización y carga del modelo

La inicialización del motor se realiza en el constructor de la clase y consta de los siguientes pasos:

- **Inicialización del contexto de ML.NET:** Se crea una instancia de `MLContext`, que actúa como punto central para todas las operaciones de Machine Learning, tanto de carga del modelo como de inferencia.

- **Detección del entorno de ejecución:** El motor distingue entre ejecución local y ejecución en Azure mediante la variable de entorno `WEBSITE_INSTANCE_ID`. Esta comprobación permite adaptar dinámicamente la ruta desde la que se carga el modelo entrenado.
- **Carga del modelo entrenado desde disco:**
  - En entorno local, el modelo se carga desde el directorio de trabajo de la aplicación.
  - En entorno Azure, el modelo se carga desde el directorio temporal del sistema (`Path.GetTempPath()`), que es el mecanismo recomendado para almacenar artefactos generados dinámicamente en entornos cloud.
- **Creación del PredictionEngine:** Una vez cargado el modelo, se crea una instancia de `PredictionEngine<RestaurantRating, RestaurantRatingPrediction>`, que permite realizar predicciones individuales de forma eficiente.

Este diseño permite que el modelo se cargue una única vez durante el ciclo de vida del servicio y se reutilice para todas las solicitudes de recomendación posteriores, evitando costes innecesarios de inicialización.

Generación de recomendaciones

La lógica principal de inferencia se implementa en el método `Recommend`, cuyo funcionamiento se detalla a continuación:

- **Entrada de datos:**
  - El identificador del usuario autenticado.
  - Un conjunto de identificadores de restaurantes candidatos sobre los que se desea calcular la afinidad.
- **Cálculo de la puntuación de afinidad:** Para cada restaurante candidato, se construye una instancia de `RestaurantRating` que contiene:
  - El identificador del usuario.
  - El identificador del restaurante.
- Esta estructura se pasa al `PredictionEngine`, que devuelve una predicción con una puntuación (`Score`) que representa el grado estimado de afinidad entre el usuario y el restaurante.
- **Filtrado por umbral mínimo:** Para evitar recomendaciones con baja relevancia, se aplica un umbral mínimo de score (`Score > 0.4`). Este valor actúa como filtro de calidad, eliminando resultados cuya afinidad estimada no resulta suficientemente significativa.
- **Ordenación de resultados:** Los restaurantes que superan el umbral se ordenan de forma descendente según su puntuación, devolviendo finalmente un listado de identificadores ordenados por relevancia.

Este enfoque permite generar recomendaciones personalizadas de forma eficiente y controlada, garantizando que únicamente se incluyan en el feed aquellos restaurantes con una afinidad estimada suficientemente alta.

A.1.5 Motor de Filtrado Basado en Contenido

El motor de Filtrado Basado en Contenido se implementa en la clase `ContentBasedRecommendationEngine`. A diferencia del modelo de filtrado colaborativo, este enfoque **no requiere un proceso de entrenamiento previo**, ya que las recomendaciones se calculan dinámicamente en tiempo de ejecución a partir de las características propias de los restaurantes.

Tal y como se muestra en la **Ilustración 32**, el motor utiliza transformaciones y operaciones proporcionadas por ML.NET para construir representaciones vectoriales de los restaurantes y calcular su similitud respecto a las preferencias del usuario.

```
public class ContentBasedRecommendationEngine
{
    private readonly MLContext _mlContext;

    public ContentBasedRecommendationEngine()
    {
        _mlContext = new MLContext();
    }

    public IEnumerable<int> Recommend(IEnumerable<RestaurantFeatures> candidates, IEnumerable<RestaurantFeatures> likedRestaurants)
    {
        var dataView = _mlContext.Data.LoadFromEnumerable(candidates);

        var pipeline = _mlContext.Transforms.Categorical.OneHotEncoding("CategoryEncoded", nameof(RestaurantFeatures.Category))
            .Append(_mlContext.Transforms.Categorical.OneHotEncoding("PriceEncoded", nameof(RestaurantFeatures.PriceRange)))
            .Append(_mlContext.Transforms.Concatenate("Features", "CategoryEncoded", "PriceEncoded"))
            .AppendCacheCheckpoint(_mlContext);

        ITransformer transformer = pipeline.Fit(dataView);
        var transformedCandidates = transformer.Transform(dataView);

        // Obtener vectores de restaurantes con like
        IDataView likedView = _mlContext.Data.LoadFromEnumerable(likedRestaurants);
        var transformedLikes = transformer.Transform(likedView);

        var candidateEnumerator = _mlContext.Data.CreateEnumerable<RestaurantVector>(transformedCandidates, reuseRowObject: false).ToList();
        var likedVectors = _mlContext.Data.CreateEnumerable<RestaurantVector>(transformedLikes, reuseRowObject: false).ToList();

        // Calcular similitudes (distancia Euclídea)
        var ranked = candidateEnumerator
            .Select(c => new
            {
                RestaurantId = c.RestaurantId,
                Score = likedVectors.Average(l => EuclideanDistance(c.Features, l.Features))
            })
            .OrderBy(s => s.Score)
            .ToList();

        return ranked.Where(r => r.Score < 1.425).Select(r => r.RestaurantId).ToList();
    }

    private static float EuclideanDistance(float[] a, float[] b)
    {
        float sum = 0;
        for (int i = 0; i < a.Length; i++)
            sum += (a[i] - b[i]) * (a[i] - b[i]);
        return (float)Math.Sqrt(sum);
    }
}
```

Ilustración 32: Implementación del motor de Filtrado Basado en Contenido.

Inicialización del motor

El constructor de la clase se limita a inicializar una instancia de `MLContext`, que se utiliza para todas las operaciones de transformación y cálculo de similitudes. Dado que no existe un modelo entrenado persistente, no es necesario cargar artefactos adicionales desde disco.

Construcción de la representación de contenido

El método `Recommend` recibe como entrada dos conjuntos de datos:

- El conjunto de restaurantes candidatos.
- El conjunto de restaurantes previamente valorados con like por el usuario.

A partir de estos datos, el motor ejecuta el siguiente proceso:

1. **Carga de datos en ML.NET:** Los restaurantes candidatos se cargan en un `IDataView`, que es el formato interno utilizado por ML.NET para el procesamiento de datos.

2. **Transformación de atributos categóricos:** Se aplican transformaciones de *one-hot encoding* sobre los atributos categóricos:
    - Categoría del restaurante.
    - Rango de precio.
3. Estas transformaciones convierten valores categóricos en vectores numéricos binarios, permitiendo su posterior tratamiento matemático.

4. **Construcción del vector de características:** Los vectores generados se concatenan en una única columna denominada `Features`, que representa el perfil completo de cada restaurante desde el punto de vista del contenido.
5. **Aplicación del pipeline de transformación:** El pipeline se ajusta (`Fit`) sobre los restaurantes candidatos y se reutiliza para transformar tanto:
    - Los restaurantes candidatos.
    - Los restaurantes con like del usuario.
6. De este modo, ambos conjuntos quedan representados en el mismo espacio vectorial.

Cálculo de similitudes

Una vez obtenidas las representaciones vectoriales:

- Se extraen los vectores de características de los restaurantes candidatos y de los restaurantes con like.
- Para cada restaurante candidato, se calcula la **distancia euclídea media** respecto al conjunto de restaurantes valorados positivamente por el usuario.

La distancia euclídea se calcula de forma explícita mediante el método `EuclideanDistance`, que mide la separación entre dos vectores de características en el espacio multidimensional.

Este enfoque permite cuantificar el grado de similitud entre restaurantes en función de sus atributos de contenido.

Filtrado y selección de resultados

Tras el cálculo de similitudes, los restaurantes candidatos se ordenan por distancia ascendente. Finalmente, se seleccionan únicamente aquellos cuya distancia media se encuentra por debajo de un **umbral fijo (1.425)**, que actúa como criterio de similitud mínima aceptable.

El resultado final del motor es un conjunto de identificadores de restaurantes que presentan una alta similitud de contenido con aquellos previamente valorados por el usuario.

A.1.6 Modelos de datos utilizados

La librería define explícitamente los modelos de datos necesarios para trabajar con ML.NET:

- **RestaurantRating:** Representa una interacción usuario–restaurante con feedback implícito positivo.
- **RestaurantRatingPrediction:** Contiene la puntuación estimada por el modelo colaborativo.
- **RestaurantFeatures:** Agrupa los atributos relevantes del restaurante para el filtrado basado en contenido.
- **RestaurantVector:** Representa la versión vectorial de un restaurante tras las transformaciones.

Estos modelos permiten desacoplar completamente los modelos de ML de las entidades de dominio de las APIs.

A.1.7 Construcción de los conjuntos de entrenamiento

La clase **TrainingDataBuilder** actúa como adaptador entre los datos de dominio y los modelos de ML.

Sus responsabilidades incluyen:

- Transformar los likes obtenidos desde Likes API en instancias de `RestaurantRating`.
- Transformar los restaurantes obtenidos desde Restaurants API en instancias de `RestaurantFeatures`.

Este diseño evita que la lógica de transformación de datos se replique dentro de la Recommendations API.

A.1.8 Integración con Recommendations API

La integración de la librería GastromatchML en la Recommendations API se realiza mediante un método de extensión definido en la clase `GastromatchMLExtensions`. Este método encapsula el registro de los componentes de la librería en el contenedor de inyección de dependencias de ASP.NET Core, simplificando su uso desde las APIs consumidoras.

La librería expone el siguiente punto de entrada:

```
services.AddGastromatchMl();
```

Tal y como se muestra en la **Ilustración 33**, este método registra los motores de recomendación colaborativo y basado en contenido como servicios inyectables, permitiendo que la Recommendations API los utilice directamente sin necesidad de conocer los detalles internos de su implementación.

Este enfoque permite:

- Registrar los motores de recomendación de forma centralizada.
- Inyectarlos directamente en los servicios de aplicación.
- Mantener una integración limpia y desacoplada entre la API y la lógica de Machine Learning.

Gracias a esta estrategia, la Recommendations API actúa exclusivamente como un **orquestador del flujo de recomendaciones**, mientras que toda la lógica relacionada con el entrenamiento e inferencia de modelos queda encapsulada dentro de la librería GastromatchML.

```
public static class GastromatchMLExtensions
{
    public static IServiceCollection AddGastromatchMl(this IServiceCollection services)
    {
        services.AddScoped<CollaborativeRecommendationEngine>();
        services.AddScoped<ContentBasedRecommendationEngine>();

        return services;
    }
}
```

Ilustración 33: Clase de extensión GastromatchMLExtensions.

A.1.9 Justificación del diseño

La creación de la librería GastromatchML aporta las siguientes ventajas al sistema:

- Aislamiento de la complejidad de Machine Learning.
- Facilidad de mantenimiento y evolución.
- Posibilidad de reutilización en otros proyectos o servicios.

Por estos motivos, la lógica de recomendación se ha documentado de forma detallada en este anexo, manteniendo el cuerpo principal de la memoria centrado en la arquitectura y el funcionamiento del sistema.

Anexo 2: Librería Gastromatch.RabbitMQ

A.2.1 Motivación y objetivos de la librería

Durante el desarrollo de la arquitectura de GastroMatch se identificó la necesidad de implementar un mecanismo de comunicación asíncrona entre microservicios que permitiera desacoplar dominios, evitar dependencias directas y mejorar la escalabilidad del sistema. Para ello se decidió utilizar RabbitMQ como broker de mensajería.

Sin embargo, la integración directa del cliente de RabbitMQ en cada microservicio implicaba la repetición de código relacionado con:

- Gestión de conexiones y canales.
- Declaración de exchanges y colas.
- Publicación y consumo de mensajes.
- Manejo manual de acknowledgements.
- Integración con el ciclo de vida de ASP.NET Core.

Con el objetivo de evitar esta duplicación y mantener una arquitectura limpia, se desarrolló una librería NuGet propia denominada **Gastromatch.RabbitMQ**, que encapsula toda la lógica de mensajería y proporciona una API homogénea para publicar y consumir eventos dentro del ecosistema de microservicios.

La librería se basa en el modelo de mensajería *publish–subscribe* proporcionado por RabbitMQ, utilizando exchanges de tipo *topic* para el enrutamiento flexible de eventos entre microservicios [12].

A.2.2 Estructura del proyecto Gastromatch.RabbitMQ

La librería se ha desarrollado como un proyecto **.NET Standard 2.1**, permitiendo su reutilización en cualquier microservicio del sistema independientemente de su tipo.

A continuación se describen las principales **clases** que componen este módulo:

- **RabbitConnection**: gestiona la conexión persistente con el broker RabbitMQ.
- **RabbitPublisher**: implementa la lógica de publicación de eventos.
- **RabbitConsumerHostedService**: servicio en segundo plano encargado de consumir mensajes.
- **RabbitMessageContext**: encapsula metadatos asociados al mensaje recibido.
- **RabbitOptions**: configuración centralizada de conexión y mensajería.
- **RoutingKeys**: definición centralizada de las claves de enrutamiento.
- **Eventos de dominio**:
    - `UserRegistered`
    - `UserLikedRestaurant`
    - `UserUnLikedRestaurant`
    - `RestaurantCreated`
- **Interfaces**:
    - `IRabbitPublisher`
    - `IRabbitHandler<T>`

Esta organización permite separar claramente responsabilidades técnicas, contratos y eventos de dominio.

A.2.3 Configuración y gestión de la conexión

La clase `RabbitConnection` es responsable de establecer y mantener la conexión con el broker RabbitMQ. Utiliza la configuración definida en `RabbitOptions`, que incluye parámetros como:

- HostName
- Puerto
- Usuario y contraseña
- Exchange principal del sistema

La conexión se crea una única vez y se registra como **singleton** en el contenedor de dependencias, garantizando un uso eficiente de recursos y evitando múltiples conexiones innecesarias al broker.

A.2.4 Publicación de eventos

La publicación de eventos se realiza a través de la interfaz `IRabbitPublisher`, cuya implementación concreta es la clase `RabbitPublisher`.

Tal y como se muestra en la **Ilustración 34**, el publicador se encarga de preparar y enviar los eventos al broker de mensajería mediante las siguientes acciones:

- Declarar el exchange de tipo `topic`.
- Serializar el mensaje en formato JSON.
- Configurar propiedades del mensaje:
  - Persistencia.
  - ContentType.
  - CorrelationId.
- Publicar el mensaje con la routing key correspondiente.

El uso de un exchange de tipo *topic* permite una gran flexibilidad en el enrutamiento de eventos, facilitando que múltiples consumidores puedan reaccionar a un mismo evento sin acoplamiento directo.

```
public class RabbitPublisher : IRabbitPublisher
{
    private readonly RabbitConnection _conn;
    private readonly RabbitOptions _opt;

    public RabbitPublisher(RabbitConnection conn, IOptions<RabbitOptions> opt)
    {
        _conn = conn;
        _opt = opt.Value;
    }

    public async Task<bool> PublishAsync<T>(T message, string routingKey = null,
                                            string correlationId = null, CancellationToken ct = default)
    {
        using var ch = await _conn.CreateChannelAsync();
        //Declarar exchange
        await ch.ExchangeDeclareAsync(_opt.Exchange, ExchangeType.Topic, true, false);
        //Serializar el mensaje
        var body = Encoding.UTF8.GetBytes(JsonSerializer.Serialize(message));
        //Configuramos propiedades del mensaje
        var props = new BasicProperties();
        props.Persistent = true;
        props.ContentType = "application/json";
        props.CorrelationId = string.IsNullOrWhiteSpace(correlationId)
            ? Guid.NewGuid().ToString()
            : correlationId;

        var rk = routingKey ?? typeof(T).Name;
        //Publicar el mensaje
        await ch.BasicPublishAsync(_opt.Exchange, rk, false, props, body);

        return true;
    }
}
```

Ilustración 34: Clase RabbitPublisher.

A.2.5 Consumo de eventos mediante BackgroundService

El consumo de eventos en la librería Gastromatch.RabbitMQ se implementa mediante la clase genérica `RabbitConsumerHostedService<TMessage, THandler>`, que hereda de `BackgroundService`.

Este enfoque permite ejecutar consumidores de RabbitMQ como **procesos en segundo plano** dentro del ciclo de vida del microservicio, integrándose de forma natural con la infraestructura de ASP.NET Core.

El uso de `BackgroundService` resulta especialmente adecuado en este contexto, ya que permite mantener un proceso activo y persistente encargado de escuchar mensajes de RabbitMQ sin bloquear el arranque ni la ejecución principal de la API [13].

La clase divide claramente el consumo de eventos en **dos fases diferenciadas**, reflejadas directamente en su implementación:

- Inicialización del consumidor (`ExecuteAsync`)
- Procesamiento de mensajes (`OnReceivedAsync`)

Inicialización del consumidor y método ExecuteAsync

El método `ExecuteAsync` constituye el punto de entrada del `BackgroundService` y se ejecuta automáticamente cuando el microservicio se inicia. Su responsabilidad **no es procesar mensajes**, sino **configurar toda la infraestructura necesaria para el consumo**.

Tal y como se muestra en la **Ilustración 35**, en este método se realizan las siguientes operaciones clave:

- Creación del canal de comunicación con RabbitMQ a través de `RabbitConnection`.
- Declaración del exchange de tipo `topic`, compartido por todos los eventos del sistema.
- Configuración de la política de calidad de servicio (QoS) mediante `BasicQos`, limitando el número de mensajes pendientes de confirmación para evitar la sobrecarga del consumidor.
- Declaración explícita de la cola asociada al microservicio.
- Vinculación de la cola al `routing key` correspondiente al tipo de evento.
- Creación de un consumidor asíncrono (`AsyncEventingBasicConsumer`) y registro del callback `OnReceivedAsync`.
- Inicio del consumo con `autoAck = false`, delegando el control total de confirmaciones al consumidor.

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    _channel = await _conn.CreateChannelAsync();

    await _channel.ExchangeDeclareAsync(_opt.Exchange, ExchangeType.Topic, true, false);

    //Bloquea el trafico si hay más de 1000 mensajes en cola hasta que haga ACK
    await _channel.BasicQosAsync(0, 1000, global: false);
    //Declara su cola de mensajes propia ejemplo: users.auth.createdUsers
    await _channel.QueueDeclareAsync(_queue, durable: true, exclusive: false, autoDelete: false);

    //Aseguramos que la cola esta conectada al routingkey correspondiente
    await _channel.QueueBindAsync(_queue, _opt.Exchange, _routingKey);

    //Esta es la clase que escucha los mensajes que llegan a la cola
    var consumer = new AsyncEventingBasicConsumer(_channel);
    //Enlazamos a nuestro callback
    consumer.ReceivedAsync += OnReceivedAsync;

    //Indicamos al broker que nos entregue mensajes de esta cola, autoack a false para nosotros mismos hacer el ack y evitamos perder mensajes
    await _channel.BasicConsumeAsync(_queue, autoAck: false, consumer);

    await Task.Delay(Timeout.Infinite, stoppingToken);
}
```

Ilustración 35: Inicialización del consumidor en el método ExecuteAsync.

Una vez completada esta configuración, el método entra en un estado de espera indefinida mediante `Task.Delay`, manteniendo el consumidor activo mientras el microservicio se encuentre en ejecución.

Este diseño separa de forma explícita la **fase de arranque del consumidor** de la **fase de procesamiento de eventos**, mejorando la claridad y mantenibilidad del sistema.

Procesamiento de mensajes y método OnReceivedAsync

El método `OnReceivedAsync` se ejecuta cada vez que RabbitMQ entrega un mensaje a la cola del consumidor. Su responsabilidad es **procesar el evento recibido de forma segura y controlada**, delegando la lógica de negocio en el handler correspondiente.

Tal y como se muestra en la **Ilustración 36**, el flujo de procesamiento es el siguiente:

- Deserialización del mensaje recibido desde JSON al tipo genérico `TMessage`.
- Construcción de un objeto `RabbitMessageContext` que encapsula metadatos del mensaje, como el `CorrelationId` y las cabeceras.

- Creación de un *scope* de inyección de dependencias para resolver el handler correspondiente (`THandler`) sin romper el ciclo de vida *scoped*.
- Delegación del procesamiento de la lógica de negocio al handler mediante el método `HandleAsync`.
- Confirmación manual del mensaje (`ACK`) si el procesamiento finaliza correctamente.
- Rechazo del mensaje (`NACK`) en caso de error, evitando su reprocesado.

```
//se lanza cuando rabbitmq envia un mensaje
private async Task OnReceivedAsync(object sender, BasicDeliverEventArgs ea)
{
    try
    {
        var json = Encoding.UTF8.GetString(ea.Body.Span);
        var message = JsonSerializer.Deserialize<TMessage>(json)!;
        var context = new RabbitMessageContext
        {
            DeliveryTag = ea.DeliveryTag.ToString(),
            CorrelationId = ea.BasicProperties?.CorrelationId,
            Headers = ea.BasicProperties?.Headers ?? new Dictionary<string, object>()
        };

        //Inyectamos aqui el handler para evitar romper el scoped del handler con el singleton del backgroundService
        using var scope = _sp.CreateScope();
        var handler = scope.ServiceProvider.GetRequiredService<THandler>();

        //procesar el mensaje con el handler correspondiente a este tipo de mensaje
        await handler.HandleAsync(message, context, CancellationToken.None);

        //si todo fue bien enviamos ack y rabbitmq eliminara el mensaje de la cola
        await _channel!.BasicAckAsync(ea.DeliveryTag, multiple: false);
    }
    catch
    {
        // En caso de error simplemente rechazamos el mensaje (sin DLQ)
        await _channel!.BasicNackAsync(ea.DeliveryTag, multiple: false, requeue: false);
    }
}
```

Ilustración 36: Procesamiento de mensajes en el método OnReceivedAsync.

El uso de confirmaciones manuales (ACK/NACK) garantiza que los mensajes solo se eliminan de la cola cuando han sido procesados correctamente, siguiendo el modelo de consumo fiable recomendado por RabbitMQ [14].

Este enfoque proporciona tolerancia a fallos y evita la pérdida de mensajes, incluso en escenarios de error durante el procesamiento.

Modelo de handlers y aislamiento del dominio

La librería Gastromatch.RabbitMQ no contiene lógica de negocio asociada al procesamiento de eventos. En su lugar, define un **modelo genérico de handlers** que permite delegar completamente el tratamiento de cada evento en los microservicios consumidores.

Cada tipo de evento se asocia a un handler específico que implementa la interfaz genérica:

`IRabbitHandler<TMessage>`

Estos handlers **no forman parte de la librería**, sino que son implementados directamente en las APIs que consumen los eventos (por ejemplo, Users API, Restaurants API o Likes API). De este modo, cada microservicio define su propia reacción ante un evento, manteniendo el control total sobre su lógica de dominio.

Tal y como se muestra en la **Ilustración 37 y 38**, el handler recibe el mensaje tipado, resuelve sus

dependencias mediante inyección de dependencias y ejecuta la lógica correspondiente dentro del dominio del microservicio.

```
builder.Services.AddRabbitConsumer<UserLikedRestaurant, UserLikedRestaurantHandler>(
        queueName: "restaurant.userlikedrestaurant.q", //cola de esta api
        routingKey: RoutingKeys.Likes.Created);
```

Ilustración 37: Ejemplo de inyección de un handler.

```
public class UserLikedRestaurantHandler : IRabbitHandler<UserLikedRestaurant>
{
    private readonly IRestaurantService _restaurantService;

    public UserLikedRestaurantHandler(IRestaurantService restaurantService)
    {
        _restaurantService = restaurantService;
    }

    public async Task HandleAsync(UserLikedRestaurant message, RabbitMessageContext context, CancellationToken ct)
    {
        await _restaurantService.UpdateLikesAsync(message.RestaurantId, 1);
    }
}
```

Ilustración 38: Ejemplo de implementación de un handler.

A.2.6 Integración con los microservicios de GastroMatch

La integración de la librería en los microservicios se realiza mediante métodos de extensión definidos en `RabbitMqServiceCollectionExtensions`.

Tal y como se muestra en la **Ilustración 39**, la configuración típica incluye:

- Registro de la conexión RabbitMQ.
- Registro del publisher.
- Registro de consumidores específicos por evento.

Este enfoque permite que cada microservicio consuma únicamente los eventos que le conciernen, manteniendo el desacoplamiento entre dominios y favoreciendo una arquitectura basada en eventos.

```
public static class RabbitMqServiceCollectionExtensions
{
    public static IServiceCollection AddRabbitMq(this IServiceCollection services, Action<RabbitOptions> configure)
    {
        services.Configure(configure);
        services.AddSingleton<RabbitConnection>();
        return services;
    }

    public static IServiceCollection AddRabbitPublisher(this IServiceCollection services)
    {
        services.AddSingleton<IRabbitPublisher, RabbitPublisher>();
        return services;
    }

    public static IServiceCollection AddRabbitConsumer<TMessage, THandler>(
        this IServiceCollection services, string queueName, string routingKey = null)
        where THandler : class, IRabbitHandler<TMessage>
    {
        services.AddScoped<THandler>();
        services.AddHostedService(sp =>
        {
            var opt = sp.GetRequiredService<IOptions<RabbitOptions>>();
            var conn = sp.GetRequiredService<RabbitConnection>();
            return new RabbitConsumerHostedService<TMessage, THandler>(
                sp, conn, opt, queueName, routingKey);
        });
        return services;
    }
}
```

Ilustración 39: Clase de extensión GastromatchMLExtensions.

A.2.7 Flujo de eventos asíncronos en la arquitectura de GastroMatch

En la **Ilustración 39** se muestra el modelo completo de comunicación *publisher–subscriber* implementado en GastroMatch mediante RabbitMQ.

Las APIs productoras de eventos, como **Auth API** o **Likes API**, publican mensajes utilizando la interfaz `IRabbitPublisher`, encapsulada en la librería **Gastromatch.RabbitMQ**. Estos eventos se envían a un *exchange* de tipo `topic`, que actúa como punto central de distribución.

RabbitMQ se encarga de enrutar los mensajes hacia las colas correspondientes en función del *routing key*, permitiendo que cada microservicio consumidor disponga de su propia cola independiente.

Las APIs consumidoras, como **Users API** o **Restaurants API**, utilizan la clase `RabbitConsumerHostedService` para escuchar los eventos de su cola y delegar el procesamiento en handlers específicos que implementan la interfaz `IRabbitHandler<T>`.

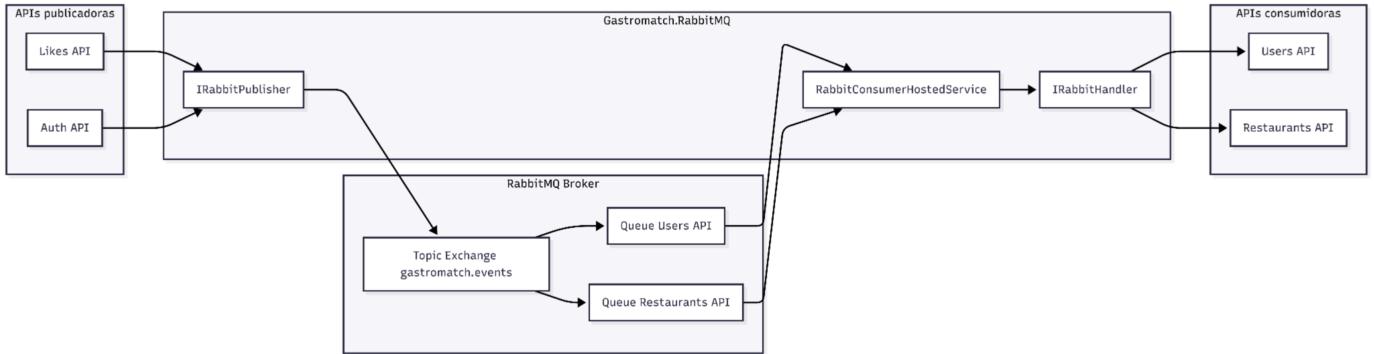

Ilustración 40: Diagrama publicador/subscriptor en Gastromatch.

A.2.8 Justificación del diseño

La creación de la librería **Gastromatch.RabbitMQ** aporta las siguientes ventajas al sistema:

- Eliminación de código duplicado en los microservicios.
- Centralización de la lógica de mensajería.
- Facilita la evolución futura del sistema de eventos.

Por estos motivos, la lógica de comunicación asíncrona se ha documentado en este anexo, manteniendo el cuerpo principal de la memoria centrado en la arquitectura y el comportamiento funcional del sistema.

Anexo 3: Gestión y despliegue de librerías NuGet personalizadas

A.3.1 Objetivo del anexo

Este anexo tiene como objetivo describir el **proceso de empaquetado, publicación y consumo** de las librerías NuGet personalizadas desarrolladas en el proyecto GastroMatch, así como su integración dentro de la arquitectura cloud y del pipeline de integración continua.

Las **motivaciones funcionales y arquitectónicas** que justifican la creación de estas librerías ya han sido detalladas en los anexos anteriores:

- En el **Anexo 1**, para la librería **GastromatchML**, donde se explica la necesidad de aislar la lógica de Machine Learning.
- En el **Anexo 2**, para la librería **Gastromatch.RabbitMQ**, donde se describe la abstracción del sistema de mensajería y la gestión de eventos.

Por tanto, este anexo no profundiza en la lógica interna de dichas librerías, sino que se centra en:

- Cómo se **empaquetan** como paquetes NuGet.
- Cómo se **publican** en un repositorio privado.
- Cómo se **consumen** desde los distintos microservicios del sistema.
- Cómo se integran dentro del flujo de **CI/CD** definido para el proyecto.

Este enfoque permite completar la visión técnica del sistema mostrando no solo cómo se diseñan las librerías, sino también cómo se **distribuyen y mantienen** de forma coherente dentro de una arquitectura basada en microservicios.

A.3.2 Librerías NuGet utilizadas en el proyecto

Durante el desarrollo de GastroMatch se han creado y utilizado **librerías NuGet personalizadas** con el objetivo de centralizar funcionalidades transversales y facilitar su reutilización entre los distintos microservicios del sistema.

Estas librerías ya han sido descritas desde el punto de vista funcional y arquitectónico en los anexos anteriores, por lo que en este apartado se presentan únicamente desde una **perspectiva de despliegue y consumo**, indicando su propósito general dentro del ecosistema del proyecto.

GastromatchML

Desde el punto de vista de despliegue, GastromatchML se distribuye como un paquete NuGet independiente y es consumida exclusivamente por la **Recommendations API**, que actúa como orquestador del flujo de recomendaciones.

Gastromatch.RabbitMQ

Desde el punto de vista de despliegue, esta librería es utilizada por los microservicios que publican o consumen eventos, como **Auth API**, **Users API**, **Restaurants API** y **Likes API**, garantizando un uso homogéneo del sistema de mensajería en todo el proyecto.

Ventajas del enfoque basado en librerías NuGet

El uso de librerías NuGet propias aporta varias ventajas clave:

- Centraliza la lógica transversal y evita duplicación de código.
- Facilita la evolución independiente de cada componente.

- Simplifica la integración en los microservicios mediante inyección de dependencias.
- Permite versionar y desplegar mejoras sin modificar directamente las APIs consumidoras.

En los siguientes apartados se describe cómo estas librerías se **empaquetan, publican y consumen** utilizando Azure DevOps y Azure Artifacts, completando así el ciclo de vida de su despliegue.

A.3.3 Proceso de empaquetado y publicación de las librerías NuGet

El empaquetado y publicación de las librerías **GastromatchML** y **Gastromatch.RabbitMQ** se realiza mediante un proceso automatizado de **Integración Continua (CI)** utilizando Azure DevOps. Este proceso garantiza que cada cambio en el código de las librerías genere una nueva versión distribuible de forma controlada y reproducible.

Tal y como se ha descrito previamente en el **Capítulo 6**, el proyecto utiliza Azure DevOps como plataforma central de CI/CD. En el caso de las librerías NuGet, el pipeline se orienta exclusivamente a la **generación y publicación de paquetes**, sin realizar tareas de despliegue en entornos de ejecución.

Tal y como se muestra en la Ilustración 41, el pipeline definido para las librerías sigue los siguientes pasos:

1. **Activación automática del pipeline:** El pipeline se dispara automáticamente ante cambios en la rama principal (`main`), garantizando que solo el código estable genera nuevas versiones del paquete.
2. **Autenticación contra Azure Artifacts:** Se utiliza la tarea `NuGetAuthenticate` para establecer la conexión segura con el feed privado de Azure Artifacts, evitando el uso de credenciales explícitas en el código.
3. **Restauración de dependencias:** El pipeline ejecuta la restauración de dependencias NuGet necesarias para compilar la librería, utilizando el feed privado del proyecto.
4. **Empaquetado del proyecto:** Mediante la tarea `dotnet pack`, se genera el archivo `.nupkg` correspondiente a la librería, utilizando la configuración `Release`. La versión del paquete se gestiona directamente desde el archivo `.csproj`, permitiendo un control explícito del versionado.
5. **Publicación del paquete:** Finalmente, el paquete generado se publica en el feed privado de Azure Artifacts mediante la tarea `dotnet push`, quedando disponible para su consumo por el resto de microservicios.

```yaml
trigger:
- main

pool: Local

steps:
- task: NuGetAuthenticate@1

- task: DotNetCoreCLI@2
  inputs:
    command: restore
    projects: '**/*.csproj'
    feedsToUse: select
    vstsFeed: 'Gastromatch/gastromatch'

- task: DotNetCoreCLI@2
  inputs:
    command: pack
    packagesToPack: 'GastromatchML/GastromatchML.csproj'
    configuration: Release

- task: DotNetCoreCLI@2
  inputs:
    command: push
    publishVstsFeed: 'Gastromatch/gastromatch'
    packagesToPush: '$(Build.ArtifactStagingDirectory)/**/*.nupkg'
```

Ilustración 41: Pipeline para publicación de NuGet propio.

A.3.4 Uso de Azure Artifacts como feed privado de NuGet

Para la distribución interna de las librerías **GastromatchML** y **Gastromatch.RabbitMQ**, el proyecto utiliza **Azure Artifacts** como **feed privado de paquetes NuGet**. Este mecanismo permite almacenar, versionar y consumir librerías propias de forma segura dentro del ecosistema de Azure DevOps, sin necesidad de publicarlas en repositorios públicos.

Azure Artifacts actúa como un repositorio centralizado de dependencias, accesible únicamente para los proyectos autorizados dentro de la organización. Este enfoque resulta especialmente adecuado en arquitecturas basadas en microservicios, donde múltiples aplicaciones comparten componentes comunes, pero no deben exponerlos públicamente.

Integración con los pipelines de CI

Tal y como se ha descrito en el punto anterior, los pipelines de las librerías publican automáticamente los paquetes generados en el feed privado de Azure Artifacts. Posteriormente, los microservicios consumidores declaran estas librerías como dependencias estándar en sus archivos de proyecto (`.csproj`), resolviéndose la descarga de los paquetes durante la fase de restauración (`dotnet restore`).

Este proceso se integra de forma transparente en los pipelines de las APIs, de modo que:

- No se requiere configuración manual adicional en los entornos de despliegue.
- Las versiones de las librerías se controlan explícitamente desde los proyectos consumidores.

Ventajas del feed privado en GastroMatch

El uso de Azure Artifacts como feed privado aporta una serie de ventajas clave al proyecto:

- **Seguridad**: las librerías no son accesibles públicamente.
- **Control de versiones**: cada paquete se versiona explícitamente, evitando dependencias implícitas o inconsistentes.

- **Reutilización**: múltiples microservicios pueden consumir las mismas librerías sin duplicar código.
- **Escalabilidad**: el sistema permite incorporar nuevas librerías internas sin modificar la arquitectura existente.

La gestión de paquetes mediante Azure Artifacts se apoya en la infraestructura estándar de Azure DevOps, tal y como se describe en la documentación oficial de Microsoft [15].

Referencias

[1] Microsoft, “Microservices architecture design”, https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

[2] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Prentice Hall, 2018.

[3] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.

[4] Microsoft, “RESTful web API design”, https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

[5] Microsoft, “*CI/CD pipelines*”, https://learn.microsoft.com/en-us/azure/devops/pipelines/

[6] Microsoft, “Azure Container Apps”, https://learn.microsoft.com/en-us/azure/container-apps/

[7] Microsoft, “Azure Virtual Network”, https://learn.microsoft.com/en-us/azure/virtual-network/

[8] Microsoft, “Azure Private Endpoint”, https://learn.microsoft.com/en-us/azure/private-link/private-endpoint-overview

[9] Microsoft, “ML.NET”, https://learn.microsoft.com/en-us/dotnet/machine-learning/

[10] Microsoft, “Train a recommendation model in ML.NET”, https://learn.microsoft.com/en-us/dotnet/machine-learning/how-to-guides/train-recommender

[11] Aggarwal, C. C., “Content-Based Recommender Systems,” Recommender Systems, Springer, 2016.

[12] RabbitMQ, “RabbitMQ Official”, https://www.rabbitmq.com/documentation.html

[13] Microsoft, “Background services in ASP.NET Core”, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services

[14] RabbitMQ, “Reliable Message Delivery”, https://www.rabbitmq.com/confirms.html

[15] Microsoft, “Azure Artifacts”, https://learn.microsoft.com/en-us/azure/devops/artifacts/

[16] Hu, Y., Koren, Y., Volinsky, C. *Collaborative Filtering for Implicit Feedback Datasets*. IEEE International Conference on Data Mining (ICDM), 2008.

[17] Lops, P., Gemmis, M., Semeraro, G. *Content-based Recommender Systems: State of the Art and Trends*. Recommender Systems Handbook, Springer, 2011.