Universidad Católica de Santa María

Facultad de Ciencias e Ingenierías Físicas y Formales

Escuela Profesional de Ingeniería de Sistemas

Implementación de una arquitectura basada en microservicios Web para la modernización de los sistemas informáticos en una universidad privada, utilizando el framework SCRUM

Tesis presentada por el Bachiller:

Herrera Alvarez, Victor Manuel

ORCID: 0009-0003-5620-8871

para optar al Título Profesional de Ingeniero de Sistemas

Asesor:

Mg. Angulo Osorio, Javier Fernando

ORCID: 0000-0003-0138-634X

Arequipa - Perú

2025

UCSM-ERP

UNIVERSIDAD CATÓLICA DE SANTA MARÍA

INGENIERIA DE SISTEMAS

TITULACIÓN CON TESIS

DICTAMEN APROBACIÓN DE BORRADOR

Arequipa, 06 de Junio del 2025

Dictamen: 013411-C-EPIS-2025

Visto el borrador del expediente 013411, presentado por:

2016100861 - HERRERA ALVAREZ VICTOR MANUEL

Titulado:

IMPLEMENTACIÓN DE UNA ARQUITECTURA BASADA EN MICROSERVICIOS WEB PARA LA MODERNIZACIÓN DE LOS SISTEMAS INFORMÁTICOS EN UNA UNIVERSIDAD PRIVADA, UTILIZANDO EL FRAMEWORK SCRUM

Nuestro dictamen es:

APROBADO

Titulo Profesional/Titulo de Segunda Especialidad/Grado Académico a optar:

INGENIERO DE SISTEMAS

29244573 - PAREDES MARCHENA FERNANDO GERMAN
DICTAMINADOR

29413196 - MONTESINOS MURILLO ANGEL FELIPE
DICTAMINADOR

29612305 - SULLA TORRES JOSE ALFREDO
DICTAMINADOR

Implementación de una arquitectura basada en microservicios Web para la modernización de los sistemas informáticos en una universidad privada, utilizando el framework SCRUM

INFORME DE ORIGINALIDAD

9%	7%	1%	4%
INDICE DE SIMILITUD	FUENTES DE INTERNET	PUBLICACIONES	TRABAJOS DEL ESTUDIANTE

FUENTES PRIMARIAS

1	Submitted to Universidad Católica de Santa María Trabajo del estudiante	2%
2	www.coursehero.com Fuente de Internet	1%
3	repositorioacademico.upc.edu.pe Fuente de Internet	<1%
4	sedici.unlp.edu.ar Fuente de Internet	<1%
5	JORGE ESTEBAN HERNÁNDEZ HORMAZÁBAL. "PROPUESTA DE UNA ARQUITECTURA PARA EL SOPORTE DE LA PLANIFICACIÓN DE LA PRODUCCIÓN COLABORATIVA EN CADENAS DE SUMINISTRO DE TIPO ÁRBOL", 'Universitat Politecnica de Valencia', 2015 Fuente de Internet	<1%
6	repositorio.ues.edu.sv Fuente de Internet	<1%

DEDICATORIA

A todas las personas que formaron parte de mi vida hasta llegar este momento y que aportaron con su granito de arena para que yo llegue a ser la persona que hoy en día culmina esta etapa, compañeros de trabajo, docentes y sobre todo a amigos y familia que fueron mi soporte todos estos años y que sin ellos no sería posible lograr esto.

AGRADECIMIENTOS

Quiero agradecer mis amigos del trabajo, ellos fueron la inspiración para impulsar la culminación de este trabajo, conté con su apoyo incondicional durante este proceso, aprendí mucho de ellos. Agradecer también a mi familia por apoyarme en mi carrera y culminación de esta, apoyándome en los peores momentos. Y Finalmente, a todas las demás personas que fueron y son parte de mi vida y de las cuales aprendí mucho para mejorar como persona y futuro profesional.

RESUMEN

El presente proyecto tiene como objetivo implementar una arquitectura basada en microservicios Web para integrarla de forma progresiva y segura en los sistemas administrativos de una universidad privada, mejorando su estructura tecnológica y sentando las bases para una evolución escalable. La arquitectura fue desarrollada utilizando Python y el *framework* FastAPI, e integrada con módulos existentes desarrollados en PHP, mediante el uso de un API *Gateway*, control de acceso por IP y autenticación mediante *token*s. El enfoque modular permitió organizar los microservicios por categorías funcionales, facilitando su mantenimiento, reutilización y despliegue independiente.

La metodología aplicada se basó en el *framework* Scrum, permitiendo desarrollar la arquitectura de manera iterativa, validando entregables en cada *sprint* y priorizando componentes críticos. La implementación se realizó en un entorno real de producción, bajo servidores con Debian 12, certificados SSL, Cloudfare y *Firewall* físico, garantizando una infraestructura segura y funcional. Se desarrollaron e integraron microservicios orientados tanto a los sistemas internos como a aplicativos móviles para estudiantes, padres y docentes.

Las pruebas funcionales, de integración, seguridad y estrés confirmaron la estabilidad de la arquitectura incluso bajo condiciones de alta carga, soportando más de 10,000 solicitudes sin caída. Además, se comprobó la efectividad de los mecanismos de control ante intentos de acceso no autorizado, suplantación de identidad y ataques por inyección de código. Como resultado, la arquitectura implementada permitió una mejor organización del ecosistema tecnológico institucional, redujo la duplicación de código, y mejoró el rendimiento, la seguridad y la escalabilidad de los sistemas de información de la universidad.

Palabras clave: Microservicios Web, API REST, Python, FastAPI, Arquitectura Web.

ABSTRACT

This project aims to implement a Web-based microservices architecture to progressively and securely integrate it into the administrative systems of a private university, improving its technological structure and laying the foundation for scalable evolution. The architecture was developed using Python and the FastAPI framework and was integrated with existing modules built in PHP through the use of an API Gateway, IP-based access control, and token-based authentication. The modular approach allowed microservices to be organized by functional categories, facilitating maintenance, reuse, and independent deployment.

The methodology applied was based on the Scrum framework, allowing the architecture to be developed iteratively, validating deliverables in each sprint and prioritizing critical components. The implementation was carried out in a real Production environment, under Debian 12 servers, with SSL certificates, Cloudfare protection, and a physical Firewall in place, ensuring a secure and functional infrastructure. Microservices were developed and integrated for both internal systems and institutional mobile applications for students, parents, and faculty.

Functional, integration, security, and stress tests confirmed the stability of the architecture even under high load conditions, withstanding over 10,000 requests without failure. In addition, the effectiveness of access control mechanisms against unauthorized attempts, identity spoofing, and code injection attacks was confirmed. As a result, the implemented architecture enabled better organization of the institutional technological ecosystem, reduced code duplication, and improved performance, security, and scalability across the university's information systems.

Keywords: Web Microservices, API REST, Python, FastAPI, Web Architecture

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

En los últimos años, las instituciones educativas han incrementado su presencia en el entorno digital, impulsadas por la necesidad de ofrecer servicios más ágiles, accesibles y eficientes tanto para su comunidad estudiantil como para su gestión interna. Esta transformación digital no solo mejora la experiencia del usuario, sino que también refuerza la imagen institucional ante un entorno cada vez más competitivo.

En la universidad donde se desarrolló este proyecto, se identificaron diversos problemas en los sistemas actuales. Por ejemplo, algunos procesos internos presentan tiempos de respuesta superiores a 2 segundos, lo cual, aunque funcional, representa un cuello de botella cuando se escalan a mayor carga. Asimismo, se han detectado tasas de duplicidad de código del 30% en componentes reutilizados en múltiples módulos, y un mantenimiento correctivo que requiere entre 4 a 5 horas por incidencia urgente, afectando la continuidad de los servicios. Estas cifras evidencian un desgaste progresivo de la arquitectura en capas (SOA) implementada previamente, la cual si bien ha sido funcional, comienza a mostrar sus límites frente a las nuevas exigencias de escalabilidad, integración móvil y eficiencia operativa.

Frente a este escenario, se planteó la implementación de una arquitectura de *software* basada en microservicios Web, como una solución estratégica que permita reorganizar, escalar y modernizar los sistemas sin reemplazar completamente lo ya construido. A diferencia de otras opciones como la adquisición de *software* externo o la reescritura total, esta alternativa ofrece una transición progresiva, aprovechando la modularidad, independencia de despliegue y facilidad de mantenimiento que caracteriza a los microservicios. Además, al tratarse de una arquitectura desacoplada y basada en estándares como REST y JSON, se facilita la integración con nuevos sistemas, como aplicaciones móviles, sin comprometer la estabilidad de los sistemas ya existentes.

La propuesta representa una innovación para la universidad, no solo por el enfoque arquitectónico, sino también por el impacto directo en su capacidad para responder con rapidez a nuevas necesidades institucionales. Se estima que con esta arquitectura se reducirá el tiempo de mantenimiento en al menos un 40%, y se acelerará el despliegue de nuevas funcionalidades, beneficiando tanto al equipo de desarrollo como a los usuarios finales.

Este trabajo se centra en el diseño, implementación y validación de dicha arquitectura. El alcance del proyecto incluye los sistemas administrativos internos relacionados con matrículas, pagos, registros académicos y consultas docentes y estudiantiles. No se contempla en esta etapa la modernización de sistemas externos como bibliotecas o sistemas contables ya tercerizados. El enfoque está en lograr una base sólida que permita escalar progresivamente la modernización institucional.

La tesis está organizada en cinco capítulos que abordan desde la contextualización del problema hasta los resultados obtenidos:

- Capítulo I: Descripción del proyecto. Incluye el planteamiento del problema, objetivos, justificación y fundamentos teóricos.
- Capítulo II: Metodología. Detalla el enfoque de investigación, la aplicación del *framework* Scrum y las técnicas empleadas para el desarrollo.
- Capítulo III: Implementación. Expone la arquitectura diseñada, su despliegue y los microservicios desarrollados.
- Capítulo IV: Resultados y discusión. Presenta los resultados funcionales y técnicos de la solución implementada.
- Capítulo V: Conclusiones y recomendaciones. Resume los hallazgos principales y plantea mejoras a futuro.

CAPÍTULO I: DESCRIPCIÓN DEL PROYECTO

1. Descripción y Objetivos del Proyecto

1.1 Descripción del Problema

La forma de desarrollar *software* ha ido evolucionando con el paso del tiempo. Inicialmente, las arquitecturas monolíticas concentraban todo el código en un único bloque, lo que dificultaba el mantenimiento, la escalabilidad y la integración con nuevos sistemas (Bass, 2012). Con la programación orientada a objetos se dio paso a las arquitecturas en capas, particularmente con la separación entre capa de presentación, lógica de negocio y acceso a datos (Fowler, 2003). Este modelo mejoró la organización interna del código, pero no resolvió del todo los problemas relacionados al despliegue, reutilización y escalabilidad, especialmente en entornos distribuidos o con alta demanda (Lewis & Fowler, 2014).

La arquitectura actual de la universidad sigue este modelo de capas. Aunque funcional, presenta limitaciones que se agravan conforme los sistemas crecen. Por ejemplo, muchas clases del sistema combinan lógica de procesamiento con conexión a base de datos en una misma unidad, lo que genera estructuras de código grandes y difíciles de mantener. Esto contradice principios de diseño como la separación de responsabilidades (SRP) y dificulta la reutilización, ya que el acoplamiento entre módulos impide modificar componentes sin afectar otros (Pérez et al., 2019).

Esta situación ha derivado en una duplicación de lógica de negocio en más del 30% de las funcionalidades administrativas, debido a que módulos distintos terminan replicando funciones similares con ligeras variaciones. Esto no solo incrementa los costos de mantenimiento, sino que también introduce riesgos de inconsistencia funcional y errores al actualizar o modificar procesos comunes. Adicionalmente, el mantenimiento de incidencias críticas puede tomar entre 4 a 6 horas en promedio, ya que las modificaciones afectan múltiples capas y módulos relacionados.

En cuanto a la escalabilidad, los sistemas actuales no pueden aislar funcionalidades para ser desplegadas o ampliadas de forma independiente. Ante picos de carga (por ejemplo, en matrículas o consultas masivas de notas), el sistema completo se ve afectado, ya que no se pueden distribuir los servicios por demanda ni asignar más recursos a un módulo sin escalar toda la aplicación. Este enfoque monolítico dentro de una arquitectura en capas limita la eficiencia del uso de recursos y la capacidad de respuesta institucional.

Algunos intentos de integración con nuevas tecnologías, como aplicativos móviles, también han evidenciado estas deficiencias. La necesidad de construir *endpoints* específicos para cada aplicación ha generado acoplamientos innecesarios entre sistemas, lo cual ralentiza el desarrollo y pone en riesgo la estabilidad del sistema principal.

Aunque ya se han desarrollado algunos microservicios en la universidad, estos no fueron planificados dentro de una arquitectura definida, lo que genera duplicación, inconsistencia de formatos, falta de seguridad estandarizada y dificultades para su mantenimiento. Actualmente, la cantidad de microservicios es manejable, lo que representa una oportunidad estratégica para rediseñar la arquitectura institucional y establecer una base sólida que permita escalar correctamente.

La arquitectura de microservicios ofrece una solución concreta a estos problemas. Al estar compuesta por servicios independientes que pueden desarrollarse, escalarse y desplegarse por separado, reduce el acoplamiento y permite aplicar principios sólidos de diseño como el aislamiento de fallos, la responsabilidad única y la reusabilidad (Newman, 2015). Además, permite integrar sistemas heterogéneos, mejorar la trazabilidad de errores, reducir los tiempos de respuesta y facilitar la evolución del sistema en etapas.

Según Dragoni et al. (2017), migrar de arquitecturas monolíticas o en capas hacia microservicios permite enfrentar de manera más efectiva los desafíos del crecimiento

tecnológico en entornos organizacionales complejos. En el contexto de la universidad, esta transición busca mejorar el mantenimiento, acelerar el desarrollo de nuevas funcionalidades y garantizar una escalabilidad real del sistema, sin comprometer la estabilidad operativa de los módulos existentes.

1.2 Objetivos del Proyecto

1.2.1 Objetivo General

Implementar una arquitectura basada en microservicios Web para integrarlos en los sistemas administrativos de una institución educativa superior, utilizando el *framework* de Scrum.

1.2.2 Objetivos Específicos

- Diseñar una arquitectura modular y escalable basada en microservicios que permita medir el nivel de integración con los sistemas administrativos actuales y facilite la futura expansión de los servicios.
- Optimizar el rendimiento del API-REST mediante la implementación de técnicas de gestión de caché, reducción del tamaño de datos y balanceo de carga, evaluando su impacto en los tiempos de respuesta y uso eficiente de recursos del sistema.
- Implementar medidas de seguridad en el acceso a la información generada por los microservicios, midiendo su efectividad en la reducción de vulnerabilidades y la protección de datos.
- Validar los resultados de la implementación de la arquitectura de microservicios en un caso real, midiendo la mejora en el rendimiento, escalabilidad y seguridad del sistema tras su aplicación.

1.3 Alcances y Limitaciones

1.3.1 Alcances

El presente proyecto tiene como alcance el diseño, desarrollo e implementación de una arquitectura de *software* basada en microservicios web, orientada a la modernización de los sistemas administrativos de una institución educativa superior en el contexto peruano. La solución fue concebida específicamente para una universidad privada, tomando en cuenta su infraestructura tecnológica actual, su equipo de desarrollo, y la necesidad de integrar sus sistemas heredados con aplicativos móviles modernos.

La arquitectura desarrollada busca sustituir progresivamente el modelo monolítico en capas que caracteriza a los sistemas existentes, mediante una estructura modular, escalable y desacoplada que facilita el mantenimiento, la integración y la evolución del ecosistema tecnológico institucional.

La implementación se realizó aplicando el *framework* ágil Scrum, lo cual permitió organizar el trabajo en iteraciones funcionales, con entregas continuas de valor y validación progresiva por parte del equipo técnico de la universidad. Dentro del alcance se incluyen el diseño de microservicios agrupados funcionalmente, la implementación de API *Gateways* para orquestar las comunicaciones, la integración con sistemas desarrollados previamente en PHP, y el despliegue de los servicios en un entorno de producción gestionado con Apache.

Asimismo, el alcance considera mejoras en los procesos internos de desarrollo: reducción de código duplicado, estandarización de interfaces, y preparación para facilitar la incorporación de nuevo personal técnico. La arquitectura también fue diseñada para facilitar la interoperabilidad con aplicativos móviles orientados a estudiantes, docentes y padres de familia.

Finalmente, el proyecto establece una base tecnológica sobre la cual la universidad podrá escalar sus servicios digitales, promover una transformación institucional sostenible y responder de manera más eficiente a las demandas de su comunidad académica y administrativa.

1.3.2 Limitaciones

Aunque la arquitectura basada en microservicios ya ha sido integrada en una parte importante de los sistemas administrativos de la universidad y ha demostrado su utilidad al facilitar el desarrollo de aplicativos móviles orientados a los estudiantes, aún existen ciertas limitaciones a considerar en esta etapa del proyecto.

Una de ellas es que el rendimiento general de los microservicios está condicionado por los recursos del servidor donde se despliegan. Si bien la arquitectura permite escalar de forma modular, una infraestructura limitada en capacidad de procesamiento, memoria o almacenamiento podría afectar el rendimiento del sistema.

Del mismo modo, factores como la velocidad de red y el ancho de banda disponible pueden influir en la comunicación entre servicios. En entornos con alta demanda o múltiples interacciones simultáneas, una red deficiente podría generar cuellos de botella.

Otra limitación es que, aunque muchos módulos ya están integrados a través de microservicios, todavía hay sistemas heredados que no han sido migrados o adaptados completamente a esta arquitectura. Esto implica que, en algunos casos, deben mantenerse soluciones intermedias que pueden afectar la uniformidad de la estructura general.

Además, la incorporación de nuevo personal al equipo de desarrollo, si bien es más sencilla gracias a la modularidad, requiere una etapa de adaptación inicial al enfoque, herramientas y buenas prácticas definidas.

CAPÍTULO II: MARCO TEÓRICO

2 Marco teórico

2.1 Estado del Arte

En la última década, las arquitecturas basadas en microservicios se han consolidado como una solución efectiva para desarrollar sistemas escalables, mantenibles y flexibles, especialmente en entornos donde la evolución tecnológica y la adaptación constante son requerimientos críticos. Este enfoque ha sido ampliamente adoptado tanto por la industria como por instituciones académicas, particularmente para la modernización de sistemas ERP y plataformas de gestión educativa, donde las arquitecturas monolíticas han comenzado a mostrar sus limitaciones (Newman, 2015; Bagheri & Ensan, 2018).

La arquitectura de microservicios permite dividir una aplicación compleja en múltiples servicios pequeños e independientes que ejecutan funcionalidades específicas y se comunican a través de APIs ligeras, generalmente RESTful. Este diseño posibilita un desarrollo desacoplado, escalabilidad horizontal, despliegues independientes y mayor facilidad de mantenimiento (Dragoni et al., 2017).

Estas características son especialmente útiles en entornos educativos, donde distintos sistemas, como gestión académica, pagos, matrículas y comunicación con estudiantes, deben operar de forma integrada y flexible.

En el ámbito educativo, Ardakani y Oroumchian (2021) estudiaron la implementación de microservicios en universidades iraníes, encontrando beneficios como mayor adaptabilidad y facilidad de actualización, pero también desafíos como la complejidad de integración y la necesidad de estandarización en la comunicación entre módulos. En estos casos, se observó que una arquitectura modular permite a las universidades responder más rápidamente a

cambios normativos y a demandas de servicios digitales, como aplicativos móviles o plataformas virtuales.

Por otro lado, la pandemia de COVID-19 aceleró la transformación de los sistemas educativos hacia entornos digitales más dinámicos. Investigaciones como la de Lee, Kim y Lee (2024) destacan que la migración hacia arquitecturas de microservicios en la nube ha permitido una mejor modularidad y respuesta ante la sobrecarga de procesos administrativos, especialmente durante periodos de matrícula o exámenes.

Desde el punto de vista de rendimiento, Mazlami, Cito y Leitner (2020) demostraron que herramientas como Kubernetes y su Horizontal Pod Autoscaler permiten escalar automáticamente servicios según la carga, lo cual es ideal para ambientes universitarios donde el tráfico varía significativamente durante el ciclo académico. También se han explorado mecanismos de control de carga para evitar la saturación de puertos o caídas por solicitudes repetidas.

En cuanto a seguridad, Sillitti y Succi (2023) analizaron la gestión de múltiples puntos de interacción en arquitecturas distribuidas, proponiendo el uso de service meshes y políticas de evaluación continua para proteger las comunicaciones internas entre servicios. Esta preocupación es especialmente relevante en el contexto educativo, donde se manejan datos sensibles de estudiantes, docentes y administrativos.

Adicionalmente, se han propuesto modelos de evolución que integran inteligencia artificial en la gestión de microservicios. Barnes (2025) sugiere que los agentes inteligentes pueden automatizar decisiones relacionadas con orquestación, escalamiento y detección de anomalías, optimizando los flujos de trabajo y reduciendo la carga operativa sobre los equipos de desarrollo.

Desde el punto de vista arquitectónico, existen distintas variantes de implementación. Las arquitecturas basadas en orquestación utilizan un componente central (como un orquestador o API *Gateway*) que controla las llamadas entre servicios, permitiendo centralizar la lógica del flujo. Este modelo facilita la trazabilidad, pero puede volverse un punto de fallo único si no se gestiona adecuadamente. En contraste, las arquitecturas basadas en eventos permiten una comunicación asíncrona mediante colas o mensajes, utilizando herramientas como Kafka o RabbitMQ. Estas arquitecturas son más tolerantes a fallos y desacopladas, pero requieren una mayor complejidad operativa y una arquitectura de soporte robusta (Pahl & Jamshidi, 2016).

En esta tesis se optó por una arquitectura orientada a servicios expuestos mediante HTTP bajo el estándar REST, con microservicios desacoplados desarrollados en Python y gestionados a través de un API *Gateway* que redirige según el tipo de servicio. Esta decisión se justifica en base a la infraestructura existente en la universidad, que ya contaba con servicios desarrollados en PHP y Python, además de aplicaciones móviles en uso. El enfoque elegido permite integrar progresivamente los microservicios sin interrumpir los servicios actuales, controlar el tráfico mediante puertos específicos, aplicar reglas de seguridad por IP o *token*, y simplificar la estructura de despliegue para un entorno académico en crecimiento.

A pesar de sus ventajas, la implementación de una arquitectura de microservicios requiere abordar diversos retos: mayor complejidad operativa, dependencia de automatización en despliegues, monitoreo distribuido, y necesidad de estándares claros en seguridad y formato de datos (Richardson, 2018). Sin embargo, estudios como los de Dragoni et al. (2019) y Newman (2015) coinciden en que los beneficios superan las dificultades cuando se realiza una planificación adecuada y se cuenta con un equipo que adopte buenas prácticas de diseño.

En resumen, la literatura actual respalda la adopción de microservicios como una solución tecnológica viable para instituciones educativas que buscan modernizar su infraestructura digital. La evidencia muestra mejoras en escalabilidad, agilidad, seguridad y facilidad de mantenimiento. Esta tesis busca precisamente aplicar esas ventajas a la realidad de una universidad privada, aportando una arquitectura modular que sirva de base para una transformación tecnológica sostenible.

Tabla 1
Comparación de estilos arquitectónicos de microservicios

Característica	**Arquitectura basada en Orquestación**	**Arquitectura basada en Eventos**	**Arquitectura implementada en la Tesis**
Modelo de comunicación	Sincrónico (normalmente vía REST o HTTP)	Asincrónico (colas de mensajes, eventos)	Sincrónico vía REST con rutas gestionadas por *Gateway*
Coordinación de servicios	Centralizada (por un orquestador o API *Gateway*)	Descentralizada (cada servicio responde a eventos)	Centralizada (API *Gateway* y control de puertos)
Acoplamiento	Moderado	Bajo	Moderado (desacoplado a nivel de servicios)
Tolerancia a fallos	Media	Alta	Media (controlado por puertos, filtrado IP, *token*s)
Trazabilidad y monitoreo	Alta (por el orquestador)	Baja (requiere herramientas de visualización)	Alta (por gestión de logs y *Gateway* centralizado)
Complejidad operativa	Media	Alta	Baja (adaptada a infraestructura ya existente)
Escalabilidad	Media	Alta	Media a Alta (segmentación por tipo de microservicio)
Adecuación a entornos educativos	Buena si existe infraestructura centralizada	Requiere mayor madurez técnica	Alta (permite integración progresiva con sistemas actuales)

Ejemplo de tecnologías comunes	API *Gateway*, Spring Cloud, Netflix OSS	Kafka, RabbitMQ, AWS EventBridge	FastAPI, Apache, PostgreSQL, Python

Nota: Elaboración prueba

Como se observa en la Tabla 1, la arquitectura implementada en esta tesis adopta un enfoque basado en orquestación ligera, gestionada mediante un API *Gateway* y segmentación por puertos, lo cual permite mantener una estructura ordenada sin introducir complejidad excesiva en el entorno actual de la universidad. A diferencia de arquitecturas más avanzadas basadas en eventos, que, si bien ofrecen alta tolerancia a fallos y desacoplamiento, requieren una infraestructura más robusta y personal especializado, la solución adoptada logra un equilibrio entre simplicidad operativa, seguridad y posibilidad de escalado.

2.2 Fundamentos Teóricos

2.2.1 API-REST

Las APIs RESTful son mecanismos de comunicación entre aplicaciones que se basan en principios definidos por la arquitectura REST. Este enfoque, propuesto por Fielding (2000), permite el intercambio eficiente de datos mediante operaciones como GET, POST o DELETE, y es especialmente adecuado para sistemas distribuidos, como los utilizados en la presente arquitectura de microservicios. En este caso, se adoptó REST para asegurar que cada solicitud del cliente incluya toda la información necesaria, permitiendo una ejecución sin dependencia del estado previo.

REST permite mantener una separación clara entre el cliente y el servidor, lo que facilita la escalabilidad del sistema al no depender del estado entre peticiones. Además, la posibilidad de almacenar respuestas en caché mejora considerablemente el rendimiento, especialmente en arquitecturas distribuidas. A diferencia de enfoques más complejos como SOAP o más recientes como GraphQL, REST destaca por ser simple y flexible, características

que han contribuido a que sea ampliamente utilizado en arquitecturas modernas basadas en microservicios (Alarcón et al., 2014; Daigneau, 2011).

En aplicaciones educativas, las APIs REST permiten integrar sistemas internos con plataformas móviles o externas, facilitando el intercambio de datos entre servicios independientes. En el contexto de microservicios, REST permite mantener una arquitectura desacoplada y modular, donde cada servicio gestiona un conjunto de recursos específico (Rivas & Peña, 2022).

Una forma común de clasificar el nivel de madurez de una API REST es a través del Modelo de Madurez de Richardson, que define cuatro niveles (Richardson, 2008):

- Nivel 0: Usa solo una única URL y verbos HTTP (por ejemplo, todo se hace con POST).
- Nivel 1: Introduce múltiples recursos identificados mediante URLs, pero sigue usando un solo verbo HTTP.
- Nivel 2: Usa recursos bien definidos y todos los verbos HTTP estándar (GET, POST, PUT, DELETE).
- Nivel 3: Añade hipertexto (HATEOAS), permitiendo al cliente navegar dinámicamente a través del API.

En la arquitectura implementada en esta tesis, se alcanzó el Nivel 2 del modelo, ya que cada microservicio expone *endpoints* claramente diferenciados, utiliza verbos HTTP apropiados para cada operación y retorna respuestas estructuradas en JSON. Aunque no se implementó navegación por hipervínculos (HATEOAS), el diseño permite una fácil integración con consumidores móviles y Web.

2.2.2 Arquitectura de *Software*

La arquitectura de *software* puede entenderse como la organización estructural de los elementos principales de un sistema, incluyendo tanto componentes como sus interacciones. En el marco de este proyecto, esta arquitectura actúa como un plano estratégico que orienta decisiones técnicas clave, desde la selección de tecnologías hasta la forma en que los módulos del sistema se comunican. Según Clements (2010), es la primera representación estructural de alto nivel que orienta decisiones clave de diseño a lo largo del ciclo de vida del *software*.

Las arquitecturas pueden clasificarse en varios estilos, como la monolítica, cliente-servidor, orientada a servicios (SOA) y basada en microservicios. Cada una tiene sus ventajas y limitaciones según el contexto y necesidades del sistema. La modularidad, escalabilidad, mantenibilidad y separación de responsabilidades son principios fundamentales de una buena arquitectura (Bass et al., 2012; Garlan & Shaw, 1993).

En el caso del presente proyecto, se parte de una arquitectura basada en capas hacia una arquitectura de microservicios, buscando mejorar la flexibilidad, el mantenimiento y la integración de los sistemas existentes.

2.2.3 Microservicios Web

Los microservicios son un estilo arquitectónico en el cual una aplicación se divide en un conjunto de servicios pequeños e independientes. Cada servicio está diseñado para realizar una función específica y se comunica con otros a través de APIs ligeras como REST, generalmente sobre HTTP (IBM, 2023).

Esta arquitectura permite que cada servicio pueda ser desarrollado, desplegado y escalado de forma autónoma, facilitando la evolución del sistema sin afectar su totalidad (Newman, 2015).

En entornos educativos, los microservicios han demostrado ser útiles para integrar diversos sistemas (académicos, administrativos, móviles) bajo una estructura unificada pero modular. También permiten mayor facilidad para el mantenimiento, reducción de redundancias y mejor escalabilidad (Villamizar et al., 2016; Dragoni et al., 2017).

Las tecnologías asociadas como Docker, Kubernetes y API *Gateway* complementan esta arquitectura, gestionando el despliegue, la orquestación y la comunicación entre servicios.

El diseño de una arquitectura de microservicios se basa en principios de ingeniería que garantizan su mantenibilidad y escalabilidad. Uno de ellos es el *Single Responsibility Principle* (SRP), el cual indica que cada servicio debe encargarse de una única funcionalidad del sistema. En esta tesis, los microservicios fueron organizados en función de dominios específicos (por ejemplo, autenticación, pagos, matrícula), evitando que un solo módulo concentre múltiples responsabilidades (Newman, 2015).

Otro principio es el Bounded Context, proveniente de Domain-Driven Design, que establece que cada microservicio debe tener un contexto bien definido, aislado lógicamente del RESTo. Esta segmentación permite que los equipos trabajen de forma independiente y evita inconsistencias en los datos y las reglas de negocio. En el caso del sistema propuesto, los servicios MSLOG, MSERP y MSAPM están claramente separados, cada uno con sus propias reglas, *token*s y lógica, facilitando el despliegue individual y el mantenimiento sin interferencia cruzada (Vernon, 2013).

Si bien la arquitectura de microservicios ofrece múltiples beneficios, también implica ciertos compromisos. Uno de ellos es la complejidad operativa, ya que el monitoreo, la autenticación y la trazabilidad deben gestionarse de forma distribuida. En esta tesis se mitigó mediante la implementación de un API *Gateway* que centraliza la entrada de solicitudes,

permite aplicar reglas de seguridad por IP o *token*s, y facilita el registro de logs (Pahl & Jamshidi, 2016).

Otro desafío es la latencia de red, ya que las llamadas entre servicios pueden afectar el rendimiento. Para reducir su impacto, se aplicaron pruebas de carga y se configuró el uso de filtros de IP para limitar accesos innecesarios. Además, los servicios críticos fueron optimizados para retornar solo los datos necesarios y permitir respuestas rápidas incluso bajo alta concurrencia (Dragoni et al., 2017).

2.2.4 Back-end

El término back-end hace referencia a la parte de un sistema encargada de la lógica de negocio, el procesamiento de datos y la interacción con las bases de datos. Es el núcleo funcional de una aplicación, encargado de responder a las solicitudes del front-end mediante APIs y servicios Web.

Un desarrollo back-end bien diseñado debe considerar seguridad, rendimiento, escalabilidad y mantenimiento. Las herramientas y *framework*s más utilizados incluyen Django (Python), Express.js (Node.js), Spring Boot (Java), Laravel (PHP) y FastAPI (Python), todos con ventajas específicas según el caso de uso (HackerRank, 2023; Olibr, 2023).

En el contexto de microservicios, el back-end cobra un rol aún más relevante, ya que cada microservicio actúa como un pequeño back-end independiente. Esto implica también nuevos retos en integración, comunicación y seguridad (Fowler, 2020).

2.2.5 Scrum y desarrollo ágil

Scrum es un *framework* ágil para la gestión y desarrollo de proyectos de *software*. Se basa en la entrega iterativa e incremental de valor, mediante ciclos de trabajo llamados *sprints*, con duración fija y resultados medibles. Se apoya en roles bien definidos: *Product Owner*,

Scrum *Master* y Equipo de Desarrollo, que trabajan de forma colaborativa y adaptativa (Schwaber & Sutherland, 2020).

A nivel académico, se ha demostrado que el uso de Scrum mejora la productividad del equipo, reduce los tiempos de desarrollo y mejora la calidad del producto (Gustavsson, 2020). Es especialmente útil en proyectos donde los requisitos evolucionan constantemente, como en el desarrollo de sistemas modernos orientados a microservicios.

Scrum forma parte del movimiento ágil, que promueve principios como la colaboración continua con el cliente, la adaptabilidad al cambio y la entrega temprana de valor (Schwaber & Sutherland, 2020). En el ámbito educativo, su implementación ha demostrado ser efectiva para gestionar proyectos tecnológicos internos, como plataformas Web, aplicaciones móviles y sistemas administrativos. Por ejemplo, en la Universidad de los Llanos se aplicó Scrum en el desarrollo de una plataforma institucional, evidenciando mejoras en la organización del equipo y el cumplimiento de objetivos de desarrollo (Rodríguez et al., 2021).

Además, investigaciones han documentado cómo la metodología ágil fomenta el trabajo colaborativo, la adaptabilidad y la eficiencia en proyectos universitarios (Barraza & Jurado, 2022). Incluso en niveles educativos previos, como la educación secundaria, Scrum ha sido adaptado con éxito para fortalecer la autogestión y el trabajo en equipo entre estudiantes (Álvarez López, 2022).

2.3 Técnicas y Herramientas

2.3.1 Técnicas

Durante el desarrollo del proyecto se aplicaron técnicas que permitieron sustentar la propuesta arquitectónica, analizar el entorno actual de la institución y orientar la toma de decisiones de manera práctica y realista.

a) Revision Documental

Se utilizó esta técnica para seleccionar tecnologías adecuadas al contexto del sistema y sustentar teóricamente la propuesta arquitectónica. Se consultaron fuentes académicas indexadas y documentación oficial de herramientas como Python, FastAPI, PostgreSQL y Apache. Esto permitió comprender buenas prácticas de implementación, estándares de seguridad y patrones de diseño recomendados en arquitecturas basadas en microservicios.

b) Observación Directa

Se evaluó el entorno tecnológico actual mediante pruebas funcionales y análisis directo de los sistemas existentes. Esta técnica permitió identificar cuellos de botella relacionados con escalabilidad, duplicación de código y dificultades de integración, proporcionando datos reales para fundamentar la transición hacia una arquitectura modular.

c) Registro de incidencias y retroalimentación técnica

Durante el desarrollo, se documentaron errores, observaciones y mejoras sugeridas, lo cual permitió un ciclo de retroalimentación continua y validación práctica de cada implementación. Esto aseguró que las decisiones técnicas estuvieran basadas en evidencia y experiencia directa.

2.3.2 Herramientas

La arquitectura basada en microservicios que se propone está construida principalmente utilizando el lenguaje de programación Python, debido a su versatilidad, curva de aprendizaje amigable y amplia comunidad de desarrolladores. Para la creación de los microservicios se emplea el *framework* FastAPI, por su facilidad de uso, estabilidad y alto rendimiento.

Actualmente, la institución cuenta con algunos microservicios previamente desarrollados en PHP, por lo que también se contempla su integración con los nuevos microservicios en Python, asegurando la interoperabilidad entre ambos entornos. Además, se hace uso de tecnologías como PostgreSQL para la gestión de bases de datos, Uvicorn como

servidor ASGI de alto rendimiento para desplegar servicios asíncronos, y Visual Studio Code como entorno de desarrollo principal. La planificación y seguimiento del desarrollo del sistema se realiza utilizando Jira, adaptando el *framework* Scrum.

a) Python

Lenguaje de alto nivel, interpretado y multiplataforma, robusto y legible con amplio soporte para desarrollo Web, APIs REST y automatización. Elegido por su sintaxis clara, compatibilidad con FastAPI y ecosistema de librerías para desarrollo *backend* y pruebas (Python Software Foundation, 2024).

b) FastAPI

FastAPI es un *framework* Web moderno orientado al desarrollo de APIs de alto rendimiento con Python. Está construido sobre características como las anotaciones de tipo y el manejo asíncrono mediante *async/await*, lo que permite validar datos automáticamente y generar documentación integrada de forma eficiente. Además, es compatible con estándares como OpenAPI y JSON *Schema*, lo que lo convierte en una opción ideal para crear microservicios seguros, mantenibles y bien estructurados, lo que permite definir rutas con validación automática, manejo de excepciones, documentación integrada y desarrollo rápido (FastAPI, 2024).

c) PHP

Lenguaje de scripting del lado del servidor, ampliamente utilizado para el desarrollo de aplicaciones Web. En el contexto de este proyecto, se utiliza para integrar microservicios previamente existentes, asegurando la compatibilidad con módulos ya implementados. PHP se caracteriza por su flexibilidad, facilidad de aprendizaje y amplia compatibilidad con bases de datos y servidores (Suraski & Gutmans, 1999; PHP, 2024).

d) Uvicorn

Es un servidor ASGI ligero y de alto rendimiento utilizado para ejecutar aplicaciones asincrónicas en Python, como las desarrolladas con FastAPI. Su eficiencia lo hace ideal para proyectos que requieren baja latencia y concurrencia elevada (Uvicorn, 2024). Se utilizó para desplegar los servicios FastAPI aprovechando el manejo eficiente de múltiples solicitudes concurrentes.

e) PostgreSQL

Sistema de gestión de bases de datos relacional avanzada de código abierto, reconocido por su estabilidad, rendimiento y conformidad con el estándar SQL. Soporta grandes volúmenes de datos, integridad referencial, funciones avanzadas como transacciones ACID, consultas JSON y extensibilidad con procedimientos almacenados (PostgreSQL Global Development Group, 2024). Se utilizó como *backend* de datos con configuraciones de índices, claves foráneas y vistas para mejorar rendimiento en consultas complejas.

f) Visual Studio Code

Editor de código fuente ligero pero potente, compatible con una amplia gama de lenguajes y extensiones. Permite depuración, control de versiones, y personalización del entorno de desarrollo, lo que facilita la productividad del desarrollador (Microsoft, 2024). Se emplearon extensiones como Pylance, REST Client, GitLens y Python Formatter para mantener un código limpio, estructurado y versionado.

g) Jira

Plataforma de gestión de proyectos ampliamente utilizada en equipos ágiles. Permite organizar tareas, priorizar requerimientos, y realizar seguimiento a través de tableros tipo Scrum. Facilita la planificación de *sprints*, revisión de entregables y colaboración entre

miembros del equipo (Atlassian, 2024). Se configuraron tableros de *sprint*, backlog de historias de usuario y seguimiento de tareas técnicas mediante roles definidos del equipo.

h) Apache

Es uno de los servidores Web más utilizados a nivel mundial. Apache permite servir aplicaciones Web de forma segura, estable y configurable. En este proyecto se utiliza para exponer y gestionar los microservicios desarrollados con Python y PHP, funcionando como punto de entrada principal en el servidor y permitiendo tareas como el balanceo de carga, configuración de proxies inversos y manejo de certificados SSL si es necesario. Su capacidad de extensión mediante módulos y su compatibilidad con múltiples lenguajes y sistemas lo hacen ideal para entornos mixtos como el de la institución (Apache Software Foundation, 2024).

CAPÍTULO III: METODOLOGÍA

En el presente capítulo se detalla la elección de las metodologías y tecnologías que sustentan el desarrollo de la arquitectura basada en microservicios Web y el uso del *framework* Scrum como metodología ágil de desarrollo.

Para finalizar el capítulo, se detalla la estructura del marco de trabajo, explicando el *framework* Scrum y describiendo los roles, eventos y artefactos utilizados durante el desarrollo del proyecto.

3 Justificación y aporte del proyecto

3.1 Tipo y diseño de investigación

El presente proyecto se enmarca dentro del enfoque de investigación aplicada, ya que su propósito principal es implementar una solución tecnológica concreta, una arquitectura basada en microservicios Web, para resolver una problemática real identificada en los sistemas informáticos de una universidad. No busca desarrollar teorías nuevas, sino aplicar conocimientos existentes de ingeniería de *software* para mejorar la eficiencia, escalabilidad y mantenibilidad de los sistemas actuales.

Asimismo, se clasifica como investigación tecnológica, en tanto propone el diseño, desarrollo e implementación de una arquitectura moderna, integrando herramientas actualizadas, principios de diseño de *software*, y metodologías ágiles (Scrum), en un entorno real de producción.

El diseño de investigación es no experimental y longitudinal, de corte documental, exploratorio y práctico. Se considera no experimental porque no se manipulan deliberadamente variables independientes ni se interviene en condiciones controladas, sino que se analiza una situación real existente para aplicar mejoras. Es longitudinal porque el proyecto se desarrolla

y evalúa a lo largo del tiempo, permitiendo observar la evolución del sistema tras la implementación progresiva de los microservicios.

Las limitaciones inherentes al diseño no experimental, como la falta de control de variables externas y la dificultad para establecer relaciones causales precisas, se mitigaron mediante:

- Triangulación de técnicas: uso de revisión documental, observación directa y retroalimentación técnica constante del equipo de desarrollo.
- Validación iterativa: realización de pruebas funcionales y de carga en distintas fases del proyecto para observar mejoras graduales.
- Documentación de resultados: registro continuo de indicadores de rendimiento antes y después de la integración de microservicios.

Dado que el proyecto no involucró encuestas ni recolección de datos de usuarios finales, no se definió una muestra poblacional en sentido estadístico tradicional. Sin embargo, se trabajó directamente con un subconjunto representativo de sistemas administrativos reales en producción, incluyendo módulos de matrícula, pagos, gestión académica y aplicativos móviles, lo cual permitió validar la solución en un entorno real con impacto institucional.

En cuanto a los elementos de análisis, se definieron las siguientes variables:

- Variable independiente: Arquitectura basada en microservicios implementada.
- Variables dependientes:
  - Rendimiento del sistema (tiempo de respuesta promedio).
  - Escalabilidad (capacidad de integración de nuevos módulos sin reescritura).
  - Mantenibilidad (incidencias técnicas reportadas, facilidad de depuración).

- Seguridad (número de ataques o accesos no autorizados detectados y mitigados).

Cada una de estas variables fue evaluada mediante indicadores técnicos específicos, como:

- Logs de rendimiento del API *Gateway*.
- Resultados de pruebas de estrés.
- Pruebas de seguridad por *token* e IP.
- Retroalimentación de desarrolladores durante la integración.

3.2 Línea de Investigación

Este proyecto se enmarca en la línea de investigación de Ingeniería Informática, según la clasificación de áreas científicas y tecnológicas establecida por la Organización para la Cooperación y el Desarrollo Económicos (OCDE, 2015). Esta línea forma parte del área principal de Ingeniería y Tecnología (código 2), específicamente bajo el subgrupo 2.2 Ingeniería Eléctrica, Electrónica e Informática, y más precisamente en la categoría 2.2.1 Ingeniería Informática.

Dentro de esta línea, el presente trabajo se orienta al diseño e implementación de arquitecturas de *software* modernas, enfocadas en la mejora de sistemas de información institucionales mediante el uso de microservicios y metodologías ágiles.

Este proyecto se enmarca en la línea de investigación de Ingeniería de Software, con énfasis en la sublínea de Arquitectura del Software, según los lineamientos de la Escuela Profesional de Ingeniería de Sistemas de la Universidad Católica de Santa María. Además, se alinea con la *Computing Classification System* (CCS) propuesta por la *Association for Computing Machinery* (ACM), ubicándose dentro de la categoría *Software and Its engineering*,

Software organization and properties, *Software architectures* (Association for Computing Machinery, 2012). Esto refuerza la relevancia académica del proyecto, al abordar aspectos fundamentales como la modularidad, la escalabilidad y la interoperabilidad mediante la implementación de una arquitectura basada en microservicios para entornos educativos reales.

3.3 Contribución y valor del proyecto

3.3.1 Justificación

En la actualidad, la universidad cuenta con una serie de sistemas desarrollados bajo una arquitectura en capas, que con el paso del tiempo ha mostrado limitaciones significativas en términos de mantenimiento, escalabilidad y facilidad de integración. En respuesta a estas limitaciones, ya se ha comenzado a implementar una arquitectura basada en microservicios, la cual ha demostrado mejoras concretas al integrarse con aplicativos móviles y permitir el despliegue independiente de funcionalidades. Sin embargo, estos microservicios se han desarrollado sin una planificación centralizada ni una arquitectura definida, lo cual representa un riesgo a futuro.

La ausencia de una estructura arquitectónica clara ha comenzado a generar duplicación de código, inconsistencias entre servicios y dificultades en la integración de nuevas funcionalidades. De continuar sin una dirección técnica adecuada, el crecimiento de microservicios desorganizados podría derivar en un entorno inmanejable, con altos costos de mantenimiento, problemas de interoperabilidad y una pérdida de control sobre el comportamiento general de los sistemas.

Implementar formalmente una arquitectura orientada a microservicios permite abordar estos problemas de forma estructurada. Esta propuesta brindará un modelo claro de desarrollo, integrando tanto los microservicios existentes como los futuros dentro de una arquitectura modular, documentada y escalable. Además, facilitará la incorporación de nuevos

desarrolladores al equipo, ya que podrán trabajar con componentes bien definidos, independientes y reutilizables, reduciendo así la curva de aprendizaje y el riesgo de errores.

Uno de los beneficios clave de esta arquitectura es su capacidad de integrarse con sistemas antiguos que siguen en funcionamiento, sin necesidad de reemplazarlos por completo. Esta compatibilidad permitirá que la universidad modernice progresivamente su infraestructura tecnológica sin interrumpir los servicios actuales, reduciendo costos y facilitando una transición gradual hacia entornos más eficientes y sostenibles.

Adicionalmente, establecer una arquitectura sólida de microservicios representa una base fundamental para el progreso tecnológico de la universidad en el mediano y largo plazo. Permitirá una mejor integración con plataformas móviles, sistemas externos y nuevas soluciones que puedan implementarse en el futuro. Esto posicionará a la institución en una situación favorable para enfrentar los desafíos tecnológicos que impone la transformación digital en el sector educativo.

Asimismo, la imagen institucional juega un papel importante en el posicionamiento de la universidad frente a su comunidad y ante otras instituciones del país y del extranjero. Contar con un ecosistema tecnológico moderno y eficiente no solo mejora los procesos internos, sino que también proyecta una imagen de innovación, competitividad y excelencia. La correcta implementación de esta arquitectura es una forma concreta de demostrar que la universidad está alineada con los estándares tecnológicos actuales, lo cual contribuye a mantener y fortalecer su estatus como institución de referencia.

Este proyecto no parte de una idea teórica, sino de una necesidad real detectada en el entorno de desarrollo actual de la universidad. Al consolidar una arquitectura bien definida, se garantiza la sostenibilidad del ecosistema tecnológico institucional, se optimiza el trabajo del equipo de desarrollo y se mejora la calidad de los servicios que se brindan a estudiantes, docentes y personal administrativo.

3.3.2 Importancia e Impacto

Este proyecto es importante porque responde a una necesidad concreta del área de desarrollo de sistemas de la universidad: organizar y estructurar de forma sostenible el uso de microservicios. La propuesta no se limita a implementar una solución técnica, sino que plantea una arquitectura pensada para evolucionar junto con los sistemas actuales, integrar los módulos ya existentes y facilitar futuras expansiones. Esto representa una mejora significativa en comparación con el modelo tradicional en capas que ha sido utilizado durante años y que ya presenta limitaciones visibles.

El impacto directo se verá reflejado en la mejora del mantenimiento de los sistemas, en la reducción de errores por duplicación de código y en una mayor eficiencia del equipo de desarrollo. Un claro ejemplo de este impacto es el uso de microservicios compartidos en varios aplicativos móviles de la universidad, lo cual ha permitido reutilizar lógica ya existente sin necesidad de reescribir código para cada aplicación. Gracias a esto, el equipo de desarrollo ha podido enfocarse más en el diseño visual y en la experiencia funcional de cada app, sin preocuparse por replicar procesos de negocio o integraciones ya resueltas previamente. Este enfoque modular ha optimizado los tiempos de desarrollo y ha mejorado la calidad general de las soluciones entregadas.

A nivel institucional, el proyecto contribuirá a fortalecer la imagen tecnológica de la universidad. Disponer de una arquitectura moderna, escalable y bien documentada permitirá enfrentar con mayor preparación los desafíos de transformación digital que afectan al sector educativo. La mejora en la experiencia tecnológica tanto para usuarios internos como externos será un reflejo del compromiso de la universidad con la innovación, la calidad del servicio y la sostenibilidad de sus procesos.

3.4 Scrum y la ingeniería de requerimientos

En el desarrollo tradicional de *software*, la ingeniería de requerimientos suele centrarse en la recolección, análisis y documentación detallada de necesidades funcionales y no funcionales antes de iniciar el desarrollo. Sin embargo, en el enfoque ágil utilizado en este proyecto, la gestión de requerimientos se realiza de forma incremental, adaptativa y colaborativa, siguiendo las prácticas del *framework* Scrum.

La aplicación de Scrum en este proyecto ha permitido abordar la implementación e integración de una arquitectura basada en microservicios de forma progresiva y organizada. A diferencia de otros proyectos centrados únicamente en la funcionalidad de un sistema, este trabajo ha requerido también gestionar requerimientos técnicos relacionados con la modularidad, interoperabilidad, escalabilidad y mantenibilidad del *software*. Estos aspectos fueron considerados desde el inicio como criterios clave de aceptación en cada microservicio desarrollado.

La herramienta Jira fue utilizada para gestionar todo el ciclo de vida de los requerimientos, desde su planteamiento inicial como historias de usuario o tareas técnicas, hasta su cierre luego de ser desarrollados e integrados. Esta plataforma facilitó la planificación y el seguimiento de los entregables en cada *sprint*, permitiendo documentar tanto las funcionalidades que se debían exponer desde los microservicios como las tareas internas de integración y refactorización.

Gracias a esta gestión iterativa, fue posible validar continuamente los avances, reorganizar prioridades y ajustar los requerimientos a medida que se iba consolidando la arquitectura. Este enfoque resultó especialmente útil para identificar dependencias entre componentes, planificar integraciones graduales con sistemas existentes, y mejorar la comunicación dentro del equipo técnico. En el *framework* de Scrum, la gestión de

requerimientos se lleva a cabo de manera continua y flexible, lo que significa que los requerimientos no se establecen por completo al inicio del proyecto, sino que se desarrollan y refinan a lo largo del ciclo de vida del producto. Este enfoque permite que los requerimientos sean revisados y ajustados continuamente, adaptándose a nuevas demandas y cambios en el entorno del proyecto. La ingeniería de requerimientos en Scrum se gestiona a través del *Product Backlog*, cuyo encargado es el *Product Owner* quien se encarga de mantener y priorizar este backlog, asegurando que refleje las necesidades de los interesados. En Scrum, los requerimientos suelen capturarse mediante historias de usuario, que describen las funcionalidades desde la perspectiva del usuario final, facilitando una comprensión clara y compartida de las expectativas. Estas historias se priorizan según su valor para el negocio, el costo estimado, el riesgo y su relación con otras tareas.

3.5 Estructura de Scrum

Scrum es un marco de trabajo ágil diseñado para gestionar proyectos complejos a través de ciclos cortos e iterativos llamados *sprints*. Su estructura se basa en roles definidos, eventos regulares y artefactos clave, que permiten mantener un flujo de trabajo enfocado en la entrega continua de valor, la transparencia y la adaptación constante (Schwaber & Sutherland, 2020).

En este proyecto, Scrum fue aplicado para organizar y ejecutar la implementación progresiva de una arquitectura basada en microservicios, adaptándolo a un contexto real en el que una sola persona asumió la ejecución principal. A diferencia de desarrollos centrados únicamente en funcionalidades visibles para el usuario final, este proyecto incluyó requerimientos técnicos esenciales como la escalabilidad, interoperabilidad, mantenibilidad y seguridad del *software*. Estos aspectos fueron abordados como criterios de aceptación desde la etapa de diseño de cada microservicio.

3.6 Roles en Scrum

Scrum define tres roles principales: *Product Owner*, Scrum *Master* y Equipo de desarrollo, cada uno con funciones específicas para garantizar una gestión eficiente del proyecto, priorizar tareas e impulsar el avance continuo (Schwaber & Sutherland, 2020).

Aunque el *framework* está pensado para equipos multidisciplinarios, en este proyecto los roles se adaptaron al contexto real de trabajo, en el cual la arquitectura fue desarrollada por una sola persona, integrándose con sistemas desarrollados de forma paralela por otros equipos.

a) *Product Owner*

Es el encargado de representar la visión del producto y de priorizar los elementos del *Product Backlog*. En este proyecto, este rol fue asumido desde el enfoque institucional, definiendo componentes prioritarios para garantizar una arquitectura escalable y sostenible.

b) Scrum *Master*

Tiene como función facilitar el proceso, remover obstáculos y asegurar el cumplimiento del marco Scrum. Este rol fue asumido de manera autónoma, aplicando buenas prácticas ágiles, organizando el trabajo mediante *sprints*, gestionando tareas en Jira y promoviendo una documentación técnica continua.

c) Equipo de Desarrollo

Aunque en Scrum normalmente está compuesto por un equipo multidisciplinario, en este caso el desarrollo fue ejecutado por una sola persona, quien diseñó, programó y validó los microservicios. Esto incluyó coordinar integraciones con otros sistemas desarrollados en paralelo, utilizando tecnologías como PHP y Python. Tal adaptación es compatible con los principios de agilidad, que permiten ajustar el marco a la realidad del equipo (Augustine, 2005; Moe et al., 2010).

3.7 Ceremonias de Scrum

Scrum contempla una serie de eventos, también llamados ceremonias, que facilitan la organización del trabajo y promueven la mejora continua:

- ***Sprint Planning*:** Se definieron los objetivos técnicos de cada *sprint*, priorizando la implementación de microservicios clave. Estas sesiones se gestionaron en Jira, documentando tareas concretas y dependencias técnicas.
- ***Daily* Scrum:** Aunque no se realizaron reuniones diarias formales, el control del progreso fue llevado de forma constante en Jira, permitiendo detectar obstáculos, reorganizar tareas y mantener la continuidad del desarrollo de forma autoorganizada.
- ***Sprint Review*:** Se llevaron a cabo reuniones periódicas con el jefe inmediato para validar los avances y recibir retroalimentación técnica. Estas sesiones funcionaron como revisiones de *sprint* al permitir la evaluación de incrementos funcionales.
- ***Sprint Retrospective*:** Si bien no se realizó una retrospectiva formal al cierre de cada *sprint*, se aplicó una retroalimentación continua basada en los problemas encontrados durante la integración, lo que permitió mejorar gradualmente el proceso de desarrollo (Denning, 2018).

3.8 Artefactos de Scrum

El marco Scrum incorpora tres artefactos fundamentales, todos utilizados en este proyecto:

- ***Product Backlog*:** Contenía tanto historias de usuario como tareas técnicas, relacionadas con funcionalidades y componentes de la arquitectura de microservicios. Su mantenimiento y priorización fue clave para garantizar una evolución coherente del sistema.

- ***Sprint Backlog*:** Se empleó para delimitar los objetivos concretos de cada ciclo de trabajo, organizando tareas por prioridad técnica.
- **Incremento:** Cada microservicio desplegado al finalizar un *sprint* constituía un incremento funcional validado y listo para integrarse con otros sistemas.

Gracias a la flexibilidad del marco ágil y al uso de herramientas como Jira, fue posible mantener una documentación técnica ordenada, validar entregables en cada iteración y ajustar prioridades según los resultados obtenidos. Esta experiencia demuestra que Scrum puede aplicarse de forma efectiva incluso en equipos reducidos, siempre que se mantengan los principios fundamentales del marco: colaboración, entrega continua y adaptación al cambio.

3.9 Historias de Usuario

Durante el desarrollo de la arquitectura basada en microservicios, la definición y gestión de los requerimientos se realizó mediante historias de usuario, una técnica común en marcos ágiles como Scrum. Estas historias permitieron capturar de manera simple y clara tanto funcionalidades esperadas desde la perspectiva del usuario como tareas técnicas fundamentales para la arquitectura.

Las historias se gestionaron mediante la plataforma Jira, lo que facilitó el control de versiones, la trazabilidad de cambios y la priorización en el *Product Backlog*. La priorización se basó en el impacto técnico, la urgencia de integración con los sistemas en producción y el valor funcional para los usuarios finales. Las tareas críticas, como los microservicios de autenticación o aquellos necesarios para los aplicativos móviles, fueron consideradas prioritarias y programadas en los primeros *sprints*.

Asimismo, a medida que se desarrollaban los componentes, se identificaron nuevas necesidades o ajustes sobre los requerimientos iniciales. Estas modificaciones se gestionaron dentro de Jira, lo que permitió actualizar las historias sin perder el control del avance del

proyecto. Esta flexibilidad fue esencial para adaptarse a los cambios que surgían en cada etapa del desarrollo.

Se consideraron tanto historias funcionales, relacionadas con funcionalidades visibles para el usuario, como historias técnicas, necesarias para asegurar la seguridad, interoperabilidad y mantenibilidad de los servicios. Cada historia fue acompañada de criterios de aceptación, definidos previamente, que guiaron las pruebas funcionales y validaron su cumplimiento.

3.9.1 Formato de Historias de Usuario

Las historias de usuario utilizadas en este proyecto se formularon siguiendo el formato propuesto por el *framework* Scrum, el cual busca describir funcionalidades o necesidades desde la perspectiva del usuario o del beneficiario directo de la funcionalidad (Schwaber & Sutherland, 2020). Este formato ayuda a mantener el enfoque en el valor que cada componente debe entregar, facilitando su validación e integración dentro del desarrollo iterativo. El modelo utilizado fue el siguiente:

- **COMO** [rol o tipo de usuario]
- **QUIERO** [acción o funcionalidad]
- **PARA** [motivo o beneficio].

En el contexto de esta tesis, las historias de usuario no se limitaron únicamente a describir requerimientos funcionales visibles, sino que también fueron utilizadas para representar tareas técnicas propias del desarrollo e integración de la arquitectura basada en microservicios.

A continuación, se presentan ejemplos representativos utilizados en el proyecto, incluyendo tanto historias funcionales como técnicas:

- **Ejemplo funcional 1:** Visualización de horarios en app móvil
  - **Como** estudiante,
  - **Quiero** visualizar mi horario desde el aplicativo móvil,
  - **Para** conocer mis clases del día sin necesidad de ingresar al sistema Web.
- **Criterios de aceptación:**
  - El microservicio debe recibir el *token* del estudiante y devolver el horario del día actual.
  - La respuesta debe estar en formato JSON con estructura validada.
  - El tiempo de respuesta no debe superar los 600 ms bajo carga moderada.
  - Se debe manejar adecuadamente los casos en que no haya clases programadas.
- **Ejemplo funcional 2:** Consulta de notas por parte de los padres
  - **Como** padre de familia,
  - **Quiero** visualizar las notas de mi hijo desde la aplicación,
  - **Para** conocer su rendimiento académico oportunamente.
- **Criterios de aceptación:**
  - El microservicio debe autenticar al padre con su *token* y validar la relación con el estudiante.
  - Se deben mostrar las notas por curso y promedio general.
  - El microservicio debe funcionar correctamente para múltiples hijos asociados.
- **Ejemplo técnico:** Validación de *token*s en *endpoints* críticos
  - **Como** desarrollador,
  - **Quiero** asegurar que todos los microservicios validen correctamente los *token*s,
  - **Para** evitar accesos no autorizados y proteger la información institucional.
- **Criterios de aceptación:**
  - Todos los *endpoints* deben rechazar solicitudes sin *token* o con *token* inválido.

- Los intentos de acceso indebido deben ser registrados en un log.
- El *token* debe ser validado contra el microservicio de seguridad correspondiente.
- Las respuestas deben usar códigos HTTP apropiados (401, 403).

Este enfoque permitió mantener claridad en los objetivos de cada microservicio, facilitar su validación técnica y asegurar que cada entrega generara valor, ya sea funcional o estructural, dentro de la arquitectura del sistema.

3.10 Proceso de Desarrollo con Scrum

El desarrollo de la arquitectura basada en microservicios se organizó mediante el *framework* Scrum, adaptado al contexto del proyecto, en el cual el trabajo fue realizado de forma individual, pero con interacción constante con otros desarrolladores y validaciones con el jefe inmediato. Esta metodología permitió estructurar el trabajo en ciclos iterativos, con objetivos claros en cada fase, facilitando el control del avance, la documentación de tareas y la integración progresiva de los microservicios (Schwaber & Sutherland, 2020).

El proceso inició con la preparación de un backlog técnico, en el cual se identificaron los componentes clave de la arquitectura, como servicios reutilizables, puntos de integración, servicios de autenticación y tareas de estandarización. Cada uno de estos elementos fue definido como una historia de usuario dentro de Jira, incluyendo criterios de aceptación y prioridades.

Los *sprints* se organizaron con una duración promedio de dos semanas, permitiendo establecer metas alcanzables y enfocarse en entregables específicos por ciclo. La planificación de cada *sprint* se realizaba con base en las prioridades del backlog y en los objetivos establecidos previamente en coordinación con el jefe de área. Durante los *sprints* se desarrollaban, probaban y documentaban los microservicios correspondientes, asegurando su compatibilidad con los sistemas ya existentes o en desarrollo.

El seguimiento se realizó utilizando Jira como herramienta principal de gestión, donde se mantenía el control del estado de cada tarea, permitiendo reorganizar prioridades o modificar estimaciones en función de los avances reales. Este control visual permitió mantener el enfoque del proyecto sin perder visibilidad sobre las dependencias entre servicios.

Al cierre de cada *sprint*, se entregaban microservicios funcionales, que eran validados con el jefe inmediato. Estos entregables incluían tanto el código del servicio como su documentación técnica, *endpoints* expuestos y pruebas realizadas. Esta dinámica de trabajo facilitó el avance continuo del proyecto, asegurando que cada componente fuera funcional y que pudiera integrarse sin generar bloqueos o reescrituras.

La aplicación de Scrum permitió dividir un proceso técnico complejo en tareas manejables, con una lógica iterativa que facilitó la validación temprana, el aprendizaje continuo y la adaptación según las necesidades técnicas detectadas a lo largo del proyecto. Este enfoque se alinea con los principios ágiles de mejora incremental, colaboración continua y entrega de valor en cada iteración (Denning, 2018).

3.10.1 Inicio del proyecto y preparación del backlog

El proyecto comenzó con la identificación de los componentes principales que formarían parte de la arquitectura de microservicios. Estos elementos se definieron como historias de usuario dentro de Jira, agrupando tanto funcionalidades visibles como tareas técnicas necesarias para la integración y escalabilidad del sistema. Esta preparación inicial del *Product Backlog* permitió priorizar desde el inicio las tareas críticas del proyecto.

3.10.2 Organización en *sprints*

El trabajo se dividió en *sprints* con una duración promedio de dos semanas. Al inicio de cada *sprint* se seleccionaban tareas del backlog según su prioridad, complejidad y

dependencia con otros servicios. Esta estructura permitió enfocarse en objetivos concretos, reduciendo la complejidad general del proyecto y facilitando el control de avances.

3.10.3 Ejecución del *sprint*

Durante la ejecución de cada *sprint* se desarrollaban, documentaban y probaban los microservicios programados. Se realizaron integraciones progresivas con sistemas existentes y otros módulos en desarrollo, garantizando la compatibilidad de la arquitectura. Las tareas se gestionaban desde Jira, y se actualizaban conforme se completaban o se detectaban necesidades de ajuste.

3.10.4 Seguimiento y control

El control de avance del proyecto se realizó mediante Jira, permitiendo visualizar el estado de cada historia de usuario y mantener el orden en la ejecución de tareas. Se realizaron reuniones periódicas con el jefe de área para validar avances, así como coordinación directa con otros desarrolladores para asegurar la correcta integración de los servicios.

3.10.5 Cierre del *sprint*

Cada *sprint* culminaba con la entrega de microservicios funcionales y documentados. Estos se evaluaban con base en los criterios definidos al inicio del ciclo, y los resultados servían como base para planificar el siguiente *sprint*. Se incorporaban observaciones técnicas recibidas, y se ajustaban prioridades en el backlog según la experiencia adquirida durante la iteración.

A continuación, se presenta un cuadro resumen de lo que se presentó en el proceso de desarrollo con Scrum aplicado al proyecto

Tabla 2
Cuadro resumen de uso de Scrum en el desarrollo

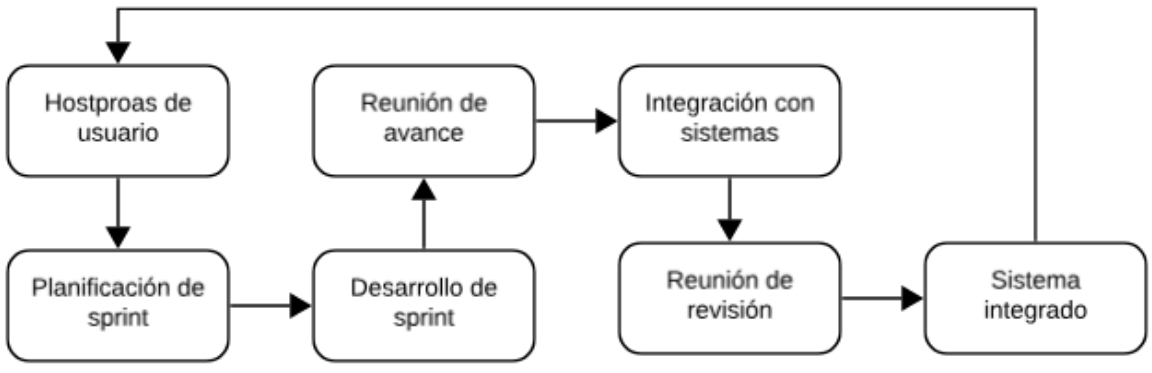
Fase del proceso	**Actividad realizada**	**Herramientas utilizadas**

Inicio del proyecto y preparación del backlog	Identificación de componentes clave de la arquitectura y redacción de historias de usuario	Jira
Organización en *sprints*	Priorización de tareas por *sprint*, definición de metas de corto plazo	Jira
Ejecución del *sprint*	Desarrollo, documentación e integración de microservicios	Python, PHP, FastAPI, Jira
Seguimiento y control	Revisión del avance, coordinación con otros desarrolladores, reuniones con jefe inmediato	Jira, reuniones técnicas
Cierre del *sprint*	Entrega de microservicios funcionales, validación y planificación del siguiente *sprint*	Jira, documentación técnica

Nota: Elaboración prueba

A continuación, se presenta un flujograma que resume el proceso de desarrollo de la arquitectura basado en el *framework* Scrum. Este flujo permitió organizar las tareas, realizar entregas incrementales y mantener validaciones continuas por parte del equipo técnico.

Figura 1
Flujograma de trabajo usando Scrum



Nota: Elaboración propia

En este capítulo se detalló la metodología empleada para el desarrollo e integración de la arquitectura basada en microservicios, adaptada al *framework* Scrum. Se expusieron las técnicas aplicadas, las herramientas utilizadas, así como la estructura práctica del proceso iterativo implementado en cada *sprint*. La adaptación de Scrum al contexto individual del proyecto permitió mantener una organización efectiva, asegurar la entrega progresiva de componentes funcionales y facilitar la integración con otros sistemas desarrollados en paralelo. Este enfoque metodológico sirvió como base para la implementación técnica que se presenta en el siguiente capítulo.

CAPÍTULO IV: IMPLEMENTACIÓN DEL PROYECTO

Este capítulo describe el proceso de implementación de la arquitectura basada en microservicios desarrollada para modernizar e integrar los sistemas administrativos de la universidad. Se detallan los componentes técnicos involucrados, el diseño general de la arquitectura, los microservicios implementados y su integración con los sistemas existentes, así como los mecanismos de seguridad y validación adoptados durante el despliegue.

La implementación se llevó a cabo en un entorno real de producción, considerando las condiciones operativas actuales de la institución y aplicando un enfoque progresivo que permitiera mantener la estabilidad de los sistemas mientras se integraban nuevas funcionalidades. Se utilizó el lenguaje Python y el *framework* FastAPI como base del desarrollo de los servicios, junto con herramientas complementarias como PostgreSQL, Apache y un API *Gateway* configurado con reglas específicas de control y enrutamiento.

Asimismo, se incluyen en este capítulo los resultados obtenidos a partir de pruebas funcionales, de integración, seguridad y estrés, que evidencian el correcto funcionamiento de la solución y su capacidad para sostener el crecimiento tecnológico de la universidad a futuro. Cada sección del capítulo aborda un aspecto clave del proceso de implementación, desde la visión general de la arquitectura hasta los resultados observables tras su puesta en marcha.

4 Implementación del proyecto

4.1 Visión General de la Arquitectura del Sistema

La arquitectura desarrollada en este proyecto sigue un enfoque de microservicios Web distribuidos, diseñados para integrarse progresivamente con los sistemas administrativos existentes de la institución. Cada microservicio representa una unidad funcional autónoma, con su propia lógica de negocio, base de datos (cuando corresponde) y despliegue independiente, lo cual facilita el mantenimiento, la evolución y la escalabilidad del ecosistema tecnológico.

La arquitectura se organiza en tres capas principales:

- **Capa de Servicios:** Agrupa los microservicios categorizados según su propósito funcional (autenticación, seguridad, gestión académica, pagos, etc.).
- **Capa de Integración:** Gestiona la comunicación entre microservicios mediante interfaces RESTful sobre HTTP, y canaliza todas las solicitudes a través de un API *Gateway* configurado con Apache como proxy inverso, validaciones por IP y certificados SSL.
- **Capa de Consumo:** Incluye sistemas Web, aplicativos móviles y otros sistemas internos que consumen los servicios mediante peticiones HTTP, garantizando interoperabilidad multilenguaje.

Figura 2
Diagrama de Arquitectura

Nota: Elaboración propia

La arquitectura cuenta con protocolos y patrones de diseño:

- **Comunicación entre microservicios:** Se realiza mediante HTTP REST, utilizando formato JSON como medio estándar de intercambio de datos.
- **Patrones implementados:**
  - **API *Gateway Pattern*:** centraliza el acceso a los servicios y permite aplicar políticas de seguridad, control de tráfico y logging.
  - ***Token-Based Authentication*:** implementado para servicios expuestos a internet, principalmente mediante JWT y validaciones en middleware.
  - ***Circuit Breaker* (parcial):** la validación de IPs y control de errores en servicios internos permite proteger la arquitectura de sobrecargas o fallos externos, aunque no se empleó una biblioteca especializada como Resilience4j.
  - ***Separation of Concerns* y SRP:** cada microservicio tiene funciones muy delimitadas, lo que facilita el testeo y mantenimiento.

En cuando a consideraciones de escalabilidad, disponibilidad y tolerancia a fallos la arquitectura cuenta con:

- **Escalabilidad Horizontal:** Al estar cada microservicio desplegado en un puerto diferente y configurado como servicio autónomo, se facilita su réplica y balanceo de carga futuro.
- **Tolerancia a fallos:** Se aplican validaciones por capa, registros de errores, pruebas de estrés (documentadas en el capítulo 4) y segmentación lógica por categoría funcional para evitar caídas en cascada.
- **Alta disponibilidad:** El entorno utiliza certificados SSL, Cloudfare y un *Firewall* físico; además, el *Gateway* filtra accesos externos, reduciendo superficie de ataque.

Como justificación técnica, la elección de FastAPI como *framework* se debió a su compatibilidad con el desarrollo asincrónico, validación automática de datos y generación integrada de documentación, mientras que Apache fue elegido por su compatibilidad con PHP y su robustez como servidor de producción. El uso de PostgreSQL como SGBD relacional garantiza integridad referencial y soporta transacciones ACID, esenciales para operaciones críticas de la institución.

4.2 Diagrama de Despliegue

El diagrama de despliegue del sistema representa la distribución física y lógica de los componentes que conforman la arquitectura basada en microservicios. Este diagrama tiene como objetivo mostrar cómo se comunican los microservicios entre sí, cómo interactúan con los sistemas ya existentes y cómo se realiza el consumo desde aplicativos externos, como los sistemas Web institucionales o las aplicaciones móviles.

La solución fue desplegada sobre un entorno Linux, utilizando Apache como servidor Web principal, donde se configuraron los puntos de acceso a los microservicios. Los servicios fueron desarrollados principalmente en Python utilizando FastAPI como *framework* principal, y se integraron con sistemas preexistentes desarrollados en PHP. Todos los servicios acceden a una base de datos PostgreSQL, ya sea de forma directa o mediante intermediarios según la lógica de negocio correspondiente.

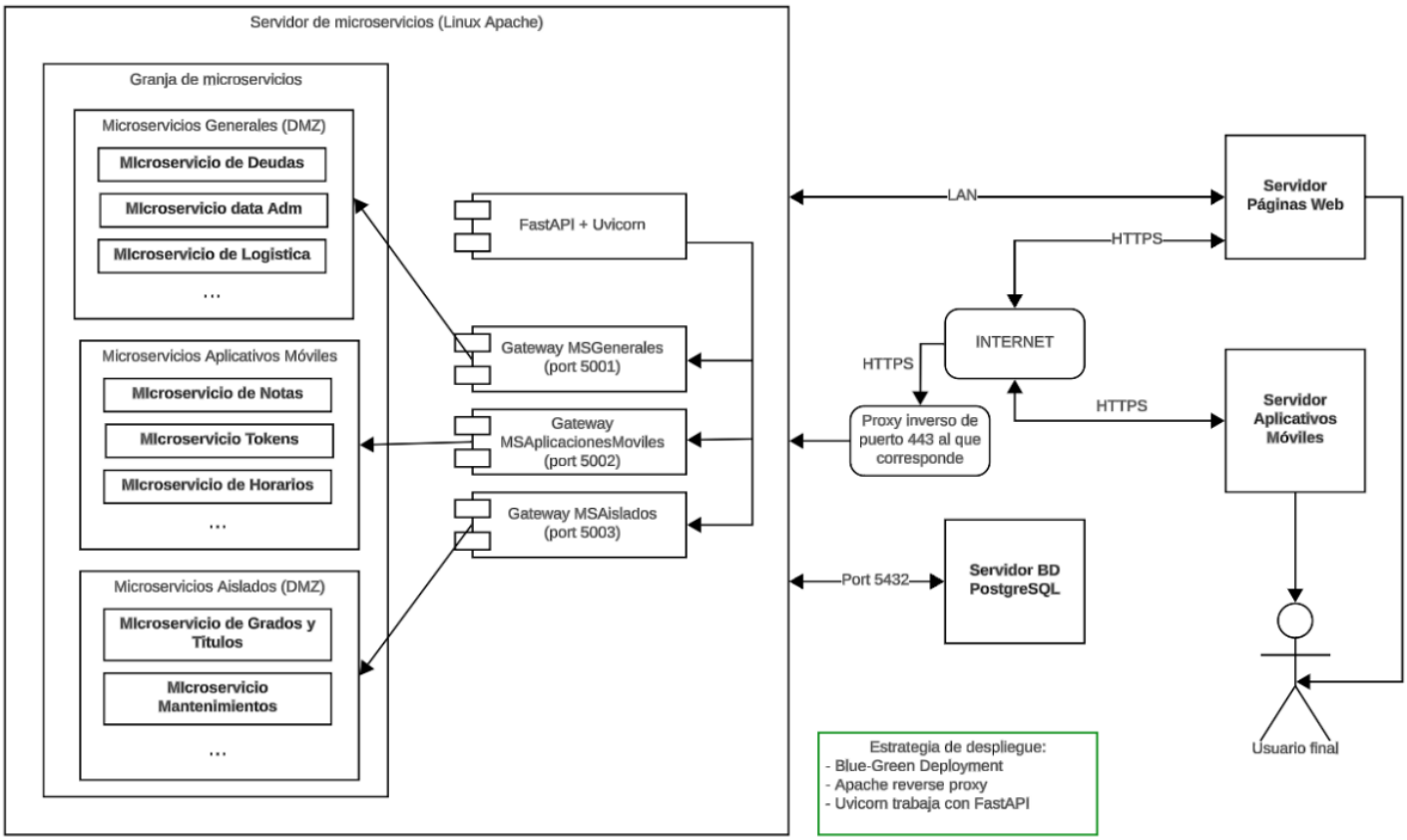
En la arquitectura propuesta, los microservicios se ejecutan de forma autónoma, lo que implica que cada uno puede desplegarse y escalarse sin depender de los demás. Esta independencia operativa se garantiza gracias a su comunicación mediante peticiones HTTP REST, las cuales se canalizan a través del API *Gateway* que organiza y enruta las solicitudes hacia los servicios correspondientes según el tipo de funcionalidad.

El despliegue contempla:

- Un servidor central que aloja los servicios en Python y PHP.
- Un entorno Apache configurado como servidor de aplicaciones y proxy inverso.
- Aplicaciones Web y móviles que consumen los servicios a través de peticiones RESTful.
- Una única instancia de base de datos PostgreSQL compartida por los distintos servicios según su ámbito de acceso.

Este enfoque modular facilita la incorporación de nuevos servicios, la distribución de carga, y la escalabilidad horizontal a futuro, sin necesidad de modificar los componentes existentes.

Figura 3
Diagrama de despliegue de la arquitectura



Nota: Elaboración propia

4.3 Microservicios desarrollados

Durante el proyecto se desarrollaron e integraron diversos microservicios enfocados en funciones específicas del ecosistema institucional. Todos fueron implementados en Python, utilizando FastAPI como *framework* principal, y actualmente se encuentran en producción. Para mantener la organización, escalabilidad y facilidad de mantenimiento, los microservicios se agruparon en categorías funcionales según su propósito, ámbito de uso y tipo de consumidor.

Esta categorización también permite aplicar políticas de seguridad diferenciadas. Por ejemplo, los microservicios orientados a los aplicativos móviles (MSAPM) implementan un sistema de autenticación y validación de *token*s más riguroso, dado que son expuestos al exterior y manejan datos sensibles de estudiantes, docentes y padres. Por otro lado, los microservicios utilizados exclusivamente dentro de los sistemas administrativos PHP mantienen una validación integrada al entorno interno, pero también están preparados para adaptarse a otras capas de seguridad si se requiere.

A continuación, se presenta una tabla con las categorías de microservicios más representativas y algunos ejemplos de los microservicios alojados en cada categoría que ya se encuentran en producción y funcionando:

Tabla 3
Microservicios desarrollados

Prefijo	Categoría funcional	Ejemplos de microservicios	Descripción general	Tecnología
MSLOG	Autenticación de acceso	MSLOGALU, MSLOGDOC, MSLOGPAD ...	Microservicios orientados a gestionar procesos de *login* para estudiantes,	Python / FastAPI

			docentes, padres y personal administrativo.	
MSTOK	Seguridad y generación de *token*s	MSTOKALU, MSTOKDOC, MSTOKPAD ...	Encargados de validar sesiones, emitir *token*s y asegurar comunicaciones entre sistemas internos y aplicativos móviles.	Python / FastAPI
MSAMP	Aplicativos móviles universitarios	MSAPMNOTA, MSAPMHORA, MSAPMDEUD ...	Servicios orientados a móviles para estudiantes, padres y docentes: consultas académicas, económicas y administrativas.	Python / FastAPI
MSERP	Módulos administrativos ERP	MSERPACAD, MSERPEMAIL, MSERPCRYPT ...	Servicios generales para sistemas en PHP: datos académicos, correo institucional, cifrado de datos, consultas varias.	Python / FastAPI
MSMTR	Matrícula	MSMTRREGULARIZACION, MSMTRDETA ...	Microservicios que controlan procesos académicos relacionados a matrícula, regularización y validación.	Python / FastAPI
MSCVP	Gestión de pagos	MSGVPGENCONCEPTOS, MSGVPVERIFICAESTADO, MSGVPREGIS ...	Servicios que gestionan la generación y verificación de pagos en	Python / FastAPI

				coordinación con módulos financieros institucionales.	

Nota: Elaboración propia

4.3.1 Detalle de microservicios desarrollados

Durante el desarrollo del proyecto, se implementaron microservicios en Python utilizando el *framework* FastAPI. Estos microservicios se organizaron en grupos funcionales de acuerdo a su propósito y ámbito de uso, usando el estándar de programación usado en la institución por el área de ERP, es decir, cumpliendo su estructura de clases, nombramiento de variables, pruebas, ejecución de procesos y puesta en producción. A continuación, se detallan ejemplos representativos de cada categoría:

Tabla 4
Detalle de microservicios

Microservicio	Categoría	Entradas principales (JSON)	Salidas esperadas	Dependencias	Consideraciones de seguridad
MSLOGALU	Autenticación	usuario, clave	*Token* de sesión, estado de acceso.	Base de datos de usuarios	Validación de credenciales, sin exposición externa
MSTOKALU	Seguridad	usuario, *token* expirado	*Token* renovado	MSLOG, sistema de *tokens* internos	Control de expiración, verificación de integridad
MSAPMNOTA	Aplicativo móvil	ID estudiante, *token*	Arreglo de notas académicas por periodo	Base de datos académica	*Token* validado, verificación del rol
MSERPCRYPT	Utilitario	Texto plano, clave	Texto cifrado o	Ninguna directa	Validación de parámetros, uso interno

			descifrado		
MSMTRREGULARIZACION	Matrícula	ID curso, ID estudiante	Estado de matrícula	Sistema de matrícula	Solo accesible desde IP autorizada (interna)
MSGVPGENCONCEPTOS	Pagos	ID estudiante, concepto	Arreglo con datos de pagos	Sistema financiero	Validación de usuario

Nota: Elaboración propia

4.3.2 Justificación de diseño y tecnología

En cuanto a la justificación de diseño y tecnología usada para los microservicios:

- FastAPI fue elegido por su rendimiento superior, su compatibilidad con *async/await*, estabilidad, fácil configuración y buena curva de aprendizaje y por ofrecer documentación automática con OpenAPI. Su uso permitió crear *endpoints* seguros, bien estructurados y con validación automática de parámetros.
- Se usaron patrones RESTful en todos los *endpoints*. Se aplicó el principio de responsabilidad única (SRP) por cada microservicio y una estructura basada en API *Gateway Pattern*, lo que permite gestionar accesos, aplicar políticas de seguridad y controlar el tráfico, gestionado por Apache, lo cual permitió evitar conflictos con los sistemas existentes en la institución
- Persistencia: Se utilizó PostgreSQL con conexiones aisladas por microservicio, evitando accesos cruzados a través de una capa de abstracción de datos.
- Seguridad: Se implementaron:
    - Filtros por IP para servicios internos.
    - *Token*s con expiración y validación para servicios expuestos.

- Verificación estricta de entradas (tipo, formato y contenido) para evitar inyecciones SQL y otros ataques.

4.4 Seguridad e interoperabilidad

La arquitectura fue diseñada con especial atención a los aspectos de seguridad e interoperabilidad, considerando el entorno mixto donde se integran microservicios de uso interno, servicios de alto rendimiento y servicios expuestos a aplicativos móviles. Esta solución se encuentra desplegada en una máquina virtual con Debian 12 sobre un servidor físico con Windows Server, contando con medidas de protección como SSL, Cloudfare, *Firewall* físico, filtrado de IP y control por *token*, garantizando un entorno seguro y confiable.

Para los mecanismos de autenticación y autorización se implementaron políticas diferenciadas según el tipo de microservicio:

- **Microservicios generales internos:** estos servicios son utilizados por los sistemas administrativos de la universidad. Se aplicó un filtro por IP desde el API *Gateway*, el cual permite únicamente el acceso desde direcciones IP autorizadas de la red institucional. Esta configuración evita accesos externos, reduce la superficie de ataque y permite un control estricto. Todo intento de acceso no autorizado queda registrado en logs para su posterior análisis.
- **Microservicios aislados de alto rendimiento:** estos servicios ejecutan tareas complejas en segundo plano, como generación masiva de datos o cálculos que requieren más tiempo. Aunque están optimizados para rendimiento, comparten el mismo esquema de seguridad que los servicios internos: filtro por IP, validación de entradas y acceso limitado desde sistemas previamente autorizados.
- **Microservicios externos para aplicativos móviles:** dado que estos servicios están expuestos a usuarios fuera de la red institucional, se implementó un sistema de

*token*s JWT. Cada *token* contiene el identificador del usuario, su rol, una fecha de expiración y un hash firmado. El *token* se verifica en cada solicitud, y cualquier acceso no autorizado o vencido es rechazado. Las comunicaciones se realizan bajo HTTPS, asegurando la confidencialidad de los datos transmitidos.

En cuanto a la validación de entradas y mitigación de vulnerabilidades todos los microservicios implementan validación estricta de parámetros, usando mecanismos automáticos de FastAPI y verificaciones manuales adicionales. Esto incluye validación de tipo, longitud, formato y contenido, previniendo ataques comunes como inyecciones SQL o manipulación de datos. Estas prácticas también aseguran que cada microservicio solo procese entradas válidas, reforzando la robustez de la arquitectura.

Para el registro de eventos y auditoría los accesos, errores y eventos sospechosos son registrados en logs estructurados tanto en el API *Gateway* como en los propios microservicios. Esto permite realizar auditorías internas, detectar patrones anómalos y aplicar mejoras de seguridad continuas. Se lleva un control específico de los intentos de acceso externo a servicios internos, incluyendo IP, fecha, hora y servicio solicitado.

Para la integridad y confidencialidad de datos:

- **Durante la transmisión:** toda comunicación externa e interna se realiza mediante HTTPS, utilizando certificados SSL válidos. Esto garantiza que los datos no sean interceptados ni modificados por terceros.
- **Durante el almacenamiento:** PostgreSQL se configura con control de accesos a nivel de rol, aislamiento por microservicio y funciones personalizadas para cifrado de campos sensibles. No existen conexiones directas entre microservicios a nivel de base de datos, lo que evita accesos cruzados o indebidos.

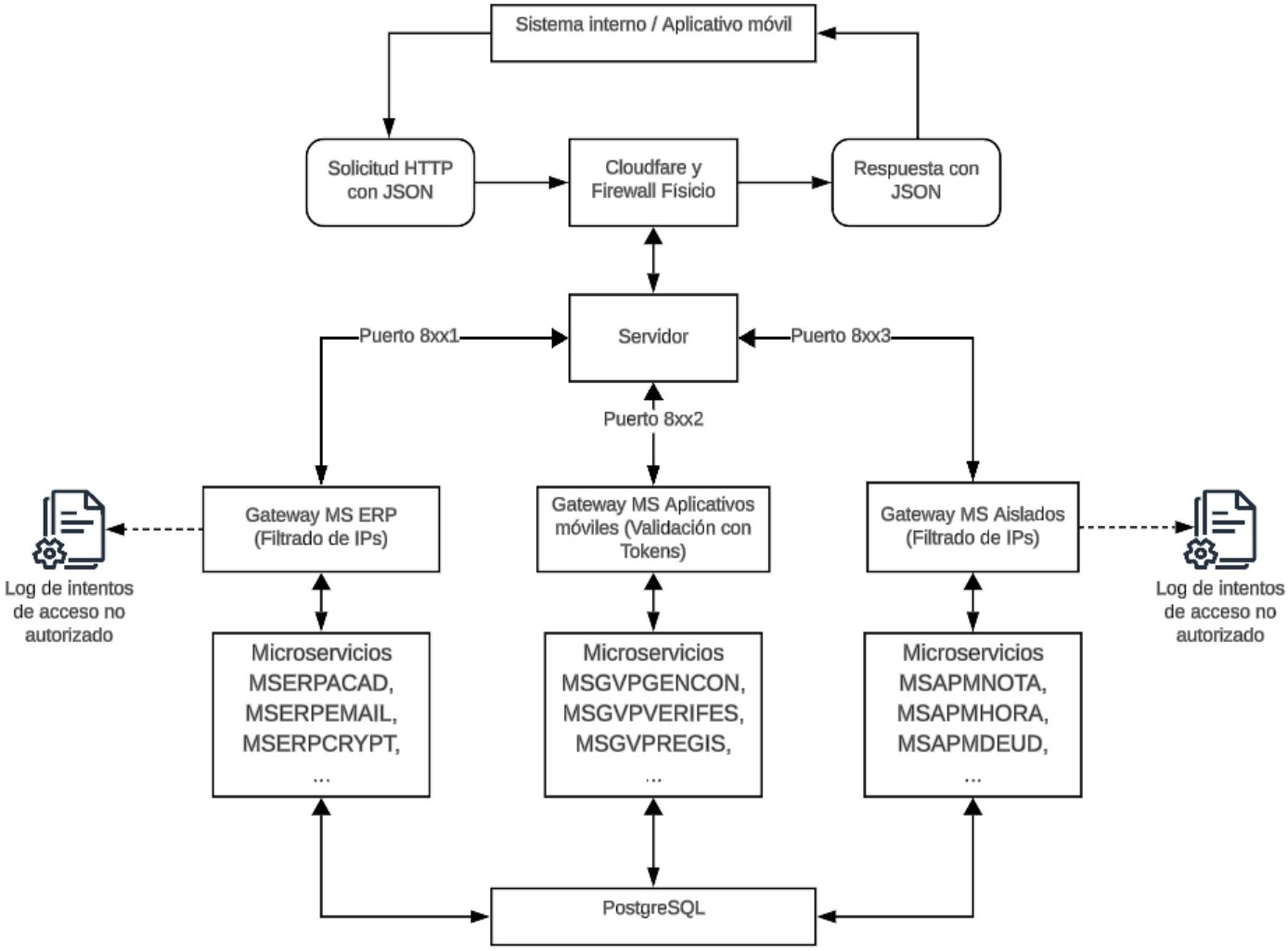
Interoperabilidad entre microservicios y sistemas existentes fue una prioridad desde el diseño. Los microservicios se comunican mediante HTTP RESTful usando JSON como formato estándar de entrada y salida. Esto asegura compatibilidad con aplicaciones en diversos lenguajes (como PHP) y facilita la integración con módulos desarrollados previamente. El uso de un API *Gateway* permite homogeneizar las rutas, manejar las diferencias de entrada, traducir respuestas y aplicar políticas centralizadas de control de tráfico y errores.

Amenazas mitigadas:

- Inyecciones SQL y validación de parámetros.
- Suplantación mediante *token*s vencidos o manipulados.
- Accesos externos a servicios internos (filtrado IP).
- Transmisión de datos sin cifrado (uso exclusivo de HTTPS).

A continuación, se presenta un gráfico que representa lo explicado anteriormente, mostrando los diferentes *Gateway*s encargados del balanceo de carga de la arquitectura y gestión de los microservicios.

Figura 4
Diagrama representativo de cómo funciona la seguridad e interoperabilidad de la arquitectura



Nota: Elaboración propia

4.5 Recursos técnicos y presupuesto

La implementación de la arquitectura basada en microservicios se desarrolló maximizando el uso de recursos existentes de la institución y tecnologías de código abierto, lo que permitió mantener un presupuesto operativo bajo. A continuación, se presentan los detalles técnicos y económicos:

Como justificación de horas-hombre el desarrollo fue ejecutado por un único desarrollador durante 7 semanas, distribuidas en 3 *sprints* según el marco Scrum. La siguiente tabla resume las actividades principales realizadas y la estimación del tiempo dedicado a cada una:

Tabla 5
Recursos técnicos en horas-hombre

Actividad	**Tiempo estimado (horas)**
Análisis y planificación técnica	20 h
Diseño y definición de arquitectura	15 h
Desarrollo de microservicios	100 h
Integración con sistemas existentes	40 h
Pruebas funcionales y de seguridad	30 h
Documentación y validaciones finales	20 h
Total estimado	**225 h**

Nota: Elaboración propia

Estas horas fueron aportadas por el desarrollador institucional como parte de sus funciones, sin generar costos adicionales directos para la institución.

Tabla 6
Recursos técnicos

Recurso	**Categoría funcional**	**Costo estimado**
Servidor virtual (Debian 12)	Máquina virtual sobre Windows Server, parte del entorno actual donde se ejecutan los sistemas ya desarrollados de la institución	0
Apache + SSL + Firewall físico	Ya configurados por el área de infraestructura	0
FastAPI	*Framework* moderno de alto rendimiento, licencia libre	0

PostgreSQL y PgAdmin4	Sistema gestor de base de datos relacional y herramienta administrativa	0
Python 3.10	Lenguaje de programación principal, de código abierto	0
Visual Studio Code	Editor de código con soporte para plugins y control de versiones	0
Jira	Gestión de tareas y *sprints* (versión institucional/licencia existente)	0
Horas-hombre de desarrollo	1 desarrollador – duración estimada: 7 semanas	Valor institucional (4000 soles)

Nota: Elaboración propia

Como análisis de costo beneficio el proyecto tuvo un costo monetario directo mínimo gracias al uso de herramientas libres y recursos internos ya existentes. A cambio, se lograron beneficios significativos:

- Reducción de errores por lógica duplicada.
- Menor tiempo de desarrollo y mantenimiento futuro.
- Mejora de la interoperabilidad entre sistemas.
- Mayor escalabilidad y control de seguridad.
- Soporte a largo plazo sin necesidad de inversiones externas en licencias.

Esto demuestra una alta rentabilidad técnica y operativa, con un retorno de inversión medible en términos de eficiencia del equipo, reducción de fallos, y sostenibilidad del ecosistema tecnológico de la institución.

4.6 Integración con sistemas existentes

Uno de los principales objetivos de esta arquitectura fue lograr una integración efectiva con los sistemas administrativos existentes en la universidad, los cuales fueron desarrollados principalmente en PHP. Esta necesidad respondió a la evolución natural del entorno institucional, donde coexistían múltiples módulos independientes que compartían

funcionalidades comunes, generando duplicación de código y dificultando su mantenimiento a largo plazo.

Con la implementación de la arquitectura basada en microservicios, se logró desacoplar funcionalidades críticas, centralizarlas en servicios reutilizables y exponerlas mediante *endpoints* accesibles desde cualquier sistema, independientemente del lenguaje o tecnología utilizada. Esto permitió que módulos ya existentes pudieran consumir estos servicios sin ser modificados internamente, facilitando así su adopción progresiva y sin impacto negativo sobre el funcionamiento general.

La integración se realizó mediante solicitudes HTTP a través del API *Gateway*, el cual redirige y controla el acceso a los servicios según su tipo. Los sistemas internos en PHP realizan peticiones a microservicios del grupo MSERP, MSMTR y MSGVP, cada uno en su respectivo API *Gateway* de MS ERP o MS Aislados según se haya definido en base a la carga que genera el microservicio, aprovechando su estructura modular para consultar información académica, registrar pagos, acceder a datos de tesis, y enviar comunicaciones institucionales. Esta conexión se validó a nivel de red mediante filtrado por IP, asegurando que solo los servidores autorizados puedan acceder a estos servicios.

Gracias a este enfoque, se logró:

- Reducir la duplicación de código, ya que funcionalidades comunes como el envío de correos o las validaciones académicas ahora son gestionadas desde un único punto.
- Facilitar el mantenimiento, al centralizar la lógica de negocio en servicios únicos, que pueden ser actualizados sin afectar directamente a los sistemas consumidores.

- Agilizar las pruebas, dado que los microservicios permiten ser validados de forma independiente, utilizando herramientas de pruebas automáticas o solicitudes directas.
- Garantizar la interoperabilidad, al adoptar estándares abiertos como JSON y REST, facilitando la integración con otros módulos o nuevas soluciones en desarrollo.

Además, esta integración también benefició a los aplicativos móviles, los cuales consumen microservicios diseñados especialmente para ellos (grupo MSAPM). Esto permitió una separación clara entre los servicios orientados al uso interno y aquellos expuestos al exterior, con políticas de seguridad adecuadas a cada caso.

En conjunto, la arquitectura propuesta no solo permitió integrar servicios nuevos con sistemas ya existentes, sino que mejoró la forma en que estos sistemas interactúan entre sí, haciendo que el ecosistema institucional sea más coherente, eficiente y sostenible a futuro.

Tabla 7
Ejemplos de integración de microservicios con sistemas institucionales

Sistema o módulo	**Microservicios consumidos**	**Tipo de integración**	**Beneficio obtenido**
Sistema de Cobranzas (PHP)	MSGVPGENCONCEPT, MSERPCOBRANZAS	Interna (vía API *Gateway* / IP)	Generación y verificación de pagos
Sistema de Tesis (PHP)	MSERPTESIS	Interna (vía API *Gateway* / IP)	Visualización de información de tesis.
Sistema de Mensajería (PHP)	MSERPENVIOEMAIL	Interna (vía API *Gateway* / IP)	Unificación del envío de correos institucionales

Aplicativo Móvil Estudiante	MSAPMNOTAS, MSAPMHORARIOS, MSAPMDEUDAS	Externa (*token* + HTTPS)	Acceso seguro a notas, horarios y estado de cuenta
Aplicativo Móvil Padres	MSAPMPADRES, MSAPMESTADOTESIS	Externa (*token* + HTTPS)	Seguimiento académico y financiero de sus hijos
Aplicativo Móvil Docentes	MSAPMDOCENTES, MSAPMCONVALIDACIONES	Externa (*token* + HTTPS)	Acceso a información de asignaturas y procesos administrativos.

Nota: Elaboración propia

Para facilitar la integración sin interrumpir el funcionamiento de los sistemas existentes, se aplicó una estrategia gradual de coexistencia y refactorización:

Identificación de módulos críticos: Se priorizaron funcionalidades repetitivas, altamente utilizadas o difíciles de mantener, como autenticación, consulta de notas y generación de conceptos de pago.

- **Encapsulamiento progresivo:** Se desarrollaron microservicios que replican la lógica de estos módulos, pero de forma desacoplada, limpia y con validaciones reforzadas. Los sistemas PHP consumen estos servicios mediante peticiones HTTP utilizando JSON.
- **Redirección de lógica:** En lugar de modificar masivamente el código existente, se insertaron puntos de consumo de microservicios (por ejemplo, llamadas curl o *file_get_contents*) que reemplazan funciones internas sin alterar el flujo general del sistema.
- **Pruebas y ajustes:** Se realizaron pruebas funcionales para asegurar que los datos devueltos por los microservicios coincidieran con los esperados por los sistemas antiguos. Los errores se gestionaron desde los microservicios para evitar fallos en cadena.

- **Despliegue escalonado:** Se integraron los servicios de forma progresiva. Por ejemplo, primero se habilitó el *login* de estudiantes, luego el de administrativos, y más adelante la generación de conceptos de pago.

Tabla 8
Desafíos encontrados y soluciones aplicadas

Desafío identificado	Solución implementada
Falta de documentación en sistemas heredados	Reversing de lógica y uso de funciones observadas en producción
Formatos diferentes de datos	Normalización a través de capas de respuesta en los microservicios
Consumo desde sistemas legados sin soporte moderno	Uso de *endpoints* REST simples
Temor a fallos por cambios simultáneos	Estrategia de migración paralela sin desactivar módulos antiguos

Nota: Elaboración propia

La integración de los microservicios tuvo un impacto positivo en distintos aspectos del ecosistema institucional:

- **En los sistemas existentes:** se redujo la duplicidad de código, se delegó parte del procesamiento a servicios especializados y se logró un aislamiento lógico que facilita futuras mejoras.
- **En los usuarios:** aunque la arquitectura no es visible directamente, la experiencia se vio beneficiada con menores tiempos de carga, mayor disponibilidad, y respuesta más ágil en aplicativos móviles.
- **En el equipo técnico:** se logró una base más robusta para el mantenimiento, simplificando la incorporación de nuevos desarrolladores al tener funciones bien documentadas y encapsuladas.

4.7 Evaluación de Tiempo, Calidad y Costo

Con el fin de evaluar rigurosamente el desarrollo de la arquitectura basada en microservicios, se establecieron métricas específicas para analizar el desempeño en tres dimensiones clave: tiempo, calidad y costo.

- **Tiempo:** La métrica aplicada fue de horas-hombre efectivas frente a estimadas.
    - Tiempo estimado: 210 horas distribuidas en 3 *sprints*.
    - Tiempo real invertido: 225 horas.

Tabla 9

Métrica de tiempo Horas-Hombre

Actividad	Estimado (h)	Real (h)	Desviación	Justificación
Planificación técnica y diseño	30 h	35 h	+ 5 h	Ajustes en la arquitectura y *Gateway*s
Desarrollo de microservicios	100 h	105 h	+ 5 h	Pruebas adicionales y casos complejos
Integración con sistemas existentes	40 h	45 h	+ 5 h	Retrabajo por diferencias de formatos
Validación, documentación y ajustes	40 h	40 h	0 h	Dentro del rango previsto
Total	**210 h**	**225 h**	**+ 15 h**	**-**

Nota: Elaboración propia

La desviación fue mínima (+7%) y controlada, sin afectar el cronograma general ni la entrega de entregables.

- **Calidad:** Las métricas aplicadas son:
    - **Cobertura funcional:** porcentaje de microservicios implementados respecto al total planificado.
    - **Tasa de éxito en pruebas:** porcentaje de casos de prueba superados.

- **Reusabilidad del código:** número de servicios consumidos por más de un sistema o módulo.

Tabla 10
Métricas de calidad

Indicador	Resultado obtenido	Meta esperada	Evaluación
Cobertura funcional	100%	>= 90%	Superado
Casos de prueba exitosos	52 de 55 (94.5%)	>= 90%	Superado
Microservicios reutilizados	14 de 17	>= 10	Superado

Nota: Elaboración propia

Criterios de calidad aplicados:

- Validación estricta de entradas
- Documentación automática Swagger
- Estándares RESTful
- Comentarios útiles y nomenclatura estandarizada
- *Token*s seguros y testeo manual + automatizado

- **Costo:** La métrica aplicada fue el costo directo estimado vs costo real proyectado en horas-hombre institucionales.
  - **Costo estimado:** S/ 3,800
  - **Costo real proyectado:** S/ 4,000

Tabla 11
Detalle de costos (resumen)

Componente	Costo (S/)	Observaciones
Horas de desarrollo	S/ 4,000	Valor institucional por 225 h (S/ 18 aprox/h)
Infraestructura y licencias	S/ 0	Servidores y *software* ya disponibles
Herramientas y *framework*s	S/ 0	Uso de tecnologías libres

Total	**S/ 4,000**

Nota: Elaboración propia

El proyecto cumplió con los objetivos definidos en términos de tiempo, calidad y costo. Las desviaciones encontradas fueron mínimas y justificadas por ajustes durante la integración con sistemas heredados y pruebas de seguridad adicionales. En general, la implementación resultó eficiente, sostenible y técnicamente sólida, validando la viabilidad económica y operativa de adoptar una arquitectura basada en microservicios dentro de la institución.

4.8 Pruebas y Validación

Con el objetivo de garantizar la robustez, eficiencia e integración efectiva de los microservicios desarrollados, se implementó una estrategia de validación integral basada en pruebas funcionales, de integración, de seguridad y de rendimiento. Estas pruebas fueron ejecutadas tanto en entornos internos como externos, simulando condiciones reales de uso y estrés del sistema.

4.8.1 Pruebas individuales

Se validó cada microservicio de forma aislada utilizando scripts en Python que enviaban solicitudes HTTP desde computadoras autorizadas por IP. Estas pruebas confirmaron la correcta conexión con la base de datos, validación de parámetros de entrada, estructura de respuestas en formato JSON, y manejo adecuado de errores y excepciones. También se utilizaron herramientas como Postman para realizar pruebas manuales, visualizar encabezados HTTP y validar los formatos esperados.

4.8.2 Pruebas de integración

Los microservicios fueron consumidos desde los sistemas administrativos en PHP y los aplicativos móviles institucionales. Estas pruebas confirmaron:

- Correcto enrutamiento a través del API *Gateway*.
- Validación de *token*s generados por los microservicios MSTOK.

- Consistencia de respuestas en diferentes entornos y tecnologías.
- Sin duplicación de lógica ni pérdida de rendimiento.

4.8.3 Pruebas de seguridad

Se realizaron pruebas diseñadas para detectar vulnerabilidades y validar los mecanismos de protección implementados:

- Suplantación de identidad mediante manipulación de *token*s.
- Inyecciones SQL con parámetros maliciosos.
- Ataques de denegación de servicio (DoS) con múltiples solicitudes en corto tiempo.

Las pruebas demostraron que los controles de validación de parámetros, la autenticación con *token*s, el filtrado por IP y el *delay* progresivo implementado en microservicios externos funcionan correctamente. Además, la infraestructura se reforzó con un *Firewall* físico y la protección adicional de Cloudfare.

4.8.4 Pruebas de estrés

Para evaluar el rendimiento y la resiliencia de la arquitectura, se desarrollaron scripts en Python que ejecutaron hasta 10,000 solicitudes en bucle hacia distintos microservicios, tanto desde el entorno interno como externo. Estas pruebas confirmaron que:

- El API *Gateway* no se cayó ni presentó saturación.
- Los puertos gestionaron correctamente el tráfico concurrente.
- El sistema respondió en tiempo aceptable sin pérdida de estabilidad.

Los resultados de todas estas pruebas confirmaron que la arquitectura no solo es funcional, sino también escalable, segura y robusta, incluso en condiciones exigentes y escenarios críticos.

A continuación, se presenta un cuadro resumen de las pruebas realizadas sobre los microservicios y hacia la arquitectura

Tabla 12
Pruebas realizadas en microservicios y arquitectura

Tipo de prueba	Objetivo	Herramientas utilizadas	Resultado
Pruebas individuales	Validar el funcionamiento de cada microservicio de forma aislada	Python (requests), Postman	Respuestas correctas en formato JSON, manejo adecuado de errores y validación de parámetros
Pruebas de integración	Verificar interacción entre microservicios y sistemas PHP / apps móviles	API *Gateway*, PHP, apps móviles	Integración exitosa, sin duplicidad de lógica, *token*s y filtros IP correctamente aplicados
Pruebas de seguridad	Evaluar protección ante accesos no autorizados o maliciosos	*Token*s manipulados, entradas SQL	Rechazo efectivo de intentos de suplantación e inyecciones SQL, validación robusta
Pruebas de estrés	Probar la capacidad de respuesta en condiciones de alta carga	Python (bucles de 10,000 requests)	*Gateway* y puertos estables, sin caída del servicio ni pérdida de disponibilidad
Pruebas de control de tráfico	Verificar el límite ante solicitudes repetidas desde IP externas	Bucle externo + Cloudfare / Firewall	Detección y aplicación de *delay*, mitigación efectiva de ataques tipo flooding o abuso

Nota: Elaboración propia

4.8.5 Pruebas de estrés

Cada historia de usuario fue validada mediante criterios de aceptación definidos previamente en Jira. Por ejemplo:

- Para el microservicio de *login*, el criterio era que devolviera un *token* válido ante credenciales correctas, y error controlado ante intentos inválidos.

- Para servicios como MSERPACAD, se exigía consistencia en la respuesta y tiempos menores a 150 ms bajo carga moderada.

4.8.6 Limitaciones y amenazas a la validez

Entre las limitaciones detectadas se encuentran:

- La imposibilidad de probar con múltiples redes externas simultáneamente, limitando escenarios de carga distribuida.
- El entorno de producción, al estar ya en uso, no permitió pruebas destructivas que simularan fallos masivos o pérdida de conectividad.
- A pesar del esfuerzo por validar cada componente, algunas integraciones futuras podrían requerir nuevos ajustes por cambios en los sistemas existentes.

4.8.7 Conclusión de pruebas y validación

Los resultados obtenidos en las distintas etapas de validación permiten concluir que la implementación de la arquitectura basada en microservicios ha generado mejoras concretas que han mejorado significativamente el funcionamiento de los sistemas institucionales, tanto desde el punto de vista técnico como operativo. Estos resultados se evidencian no solo en el desempeño individual de los servicios, sino también en la forma en que interactúan con los sistemas existentes y son consumidos desde diferentes plataformas.

Uno de los principales logros fue la reducción de duplicidad de código, gracias a la centralización de funcionalidades críticas en microservicios independientes. Esto permitió reutilizar lógica que anteriormente se encontraba replicada en distintos módulos del sistema, optimizando el mantenimiento y reduciendo el margen de error al momento de realizar ajustes o nuevas integraciones.

La arquitectura permitió además una mejor organización del ecosistema tecnológico, gracias a la categorización de microservicios por grupos funcionales (por ejemplo: MSLOG, MSTOK, MSERP, MSAPM, MSMTR, MSGVP), facilitando el despliegue, la administración

y la documentación del sistema en general. Esta organización también permitió una respuesta más ordenada ante solicitudes internas y externas, estableciendo políticas de acceso diferenciadas y controladas.

Otro resultado destacado fue la integración exitosa con múltiples sistemas desarrollados en PHP, sin necesidad de modificar sus estructuras internas. La adopción del estándar REST y el uso de formatos como JSON permitieron una interoperabilidad fluida entre tecnologías, demostrando la versatilidad de la arquitectura propuesta e implementada.

Desde el punto de vista de rendimiento, los microservicios fueron capaces de soportar altas cargas de solicitudes sin comprometer la estabilidad del sistema. Las pruebas de estrés demostraron que tanto el API *Gateway* como los puertos asignados a los servicios mantuvieron un comportamiento estable incluso ante 10,000 solicitudes concurrentes.

En cuanto a la seguridad, la arquitectura demostró ser robusta frente a intentos de acceso no autorizado, ataques por fuerza bruta y manipulación de *token*s. Las medidas de protección por IP, la autenticación por *token*s, la validación de parámetros y el registro de intentos externos permitieron construir un sistema confiable y seguro, sin comprometer la accesibilidad para los usuarios legítimos.

Además, la arquitectura facilitó el mantenimiento y la realización de pruebas. Al tener servicios aislados y con responsabilidades bien definidas, fue posible validar y actualizar componentes sin afectar otros módulos, reduciendo tiempos de intervención y riesgos operativos.

En conjunto, los resultados obtenidos confirman que la arquitectura propuesta no solo es técnicamente viable, sino también altamente funcional, adaptable y alineada con los objetivos de modernización institucional. Su implementación representa un paso significativo hacia un ecosistema tecnológico más escalable, seguro y sostenible.

La implementación de la arquitectura basada en microservicios permitió transformar el enfoque tradicional de desarrollo y gestión de los sistemas informáticos de la institución. A través de una estructura modular, segura y escalable, se logró una integración efectiva con sistemas existentes, una mejora sustancial en el mantenimiento y una alta tolerancia frente a condiciones de alta demanda. Los resultados obtenidos confirman que la arquitectura planteada es sólida, funcional y adecuada para sostener la evolución tecnológica de la universidad en el mediano y largo plazo.

CAPÍTULO V: RESULTADOS Y DISCUSIÓN

Este capítulo presenta los resultados obtenidos a partir de la implementación de la arquitectura basada en microservicios, y analiza su impacto en función del cumplimiento del objetivo general y de los objetivos específicos establecidos al inicio del proyecto. La evaluación se basa en evidencias prácticas extraídas del entorno real de la institución, considerando aspectos técnicos como el diseño arquitectónico, el rendimiento, la seguridad y la integración con los sistemas existentes.

Asimismo, se discute cómo las decisiones tomadas durante el desarrollo contribuyeron al éxito de la solución propuesta, aplicando el *framework* Scrum para estructurar, validar e iterar los entregables. Cada objetivo específico es abordado de forma detallada, destacando los logros alcanzados, las pruebas realizadas y los beneficios tangibles obtenidos a partir de la implementación en producción.

5 Resultados en Relación con los Objetivos

5.1 Objetivo General: Implementar una arquitectura basada en microservicios Web para integrarlos en los sistemas administrativos de una institución educativa superior, utilizando el *framework* de Scrum.

Se logró implementar una arquitectura distribuida basada en microservicios, diseñada con principios de modularidad, escalabilidad e interoperabilidad. Utilizando el marco ágil Scrum, el equipo ejecutó entregas incrementales con validaciones en producción por *sprint*. Al cierre del proyecto, la arquitectura se encuentra operativa, integrando sistemas internos

desarrollados en PHP con nuevos servicios en Python y FastAPI, y ofreciendo servicios a aplicativos móviles.

5.2 Objetivos Específicos

5.2.1 Diseño de una arquitectura modular y escalable

La solución implementada agrupa microservicios por función (MSERP, MSLOG, MSTOK, etc.), permitiendo desacoplamiento lógico y escalabilidad horizontal. Esta estructura redujo significativamente la duplicación de código y facilitó la incorporación progresiva de nuevas funcionalidades.

5.2.2 Optimización del rendimiento del API-REST

El rendimiento fue optimizado mediante varias técnicas: uso de respuestas compactas en formato JSON, validación anticipada de parámetros, y segmentación de microservicios por puerto según tipo de carga. Se implementó lógica para aplicar *delay* progresivo ante múltiples solicitudes repetidas desde la misma IP, evitando sobrecargas sin comprometer la disponibilidad del servicio. Las pruebas de estrés demostraron que el sistema soporta más de 10,000 solicitudes concurrentes sin interrupciones, lo que evidencia una alta tolerancia y eficiencia en el uso de recursos.

Para validar la eficiencia de los microservicios desarrollados, se realizaron pruebas de tiempo de respuesta mediante herramientas como Postman y scripts en Python. Estas pruebas se ejecutaron desde diferentes dispositivos conectados a la red institucional y externa, simulando múltiples escenarios de consumo. A continuación, se presenta una tabla con los tiempos promedio y máximos observados por microservicio, tomando en cuenta los 5 más pesados e importantes, durante pruebas de carga moderada y alta.

Tabla 13

Pruebas realizadas en microservicios y arquitectura con 10,000 solicitudes a cada microservicio

Microservicio	Funcionalidad principal	Tiempo promedio de una solicitud (ms)	Tiempo Total de prueba (s)	Estado
MSTESIS	Trae toda la información sobre una tesis, funciona en sistemas administrativos como en App móvil	97.42	976.64	Estable
MSAPMNOTAS	Recopila información de notas de otros servidores de la universidad y las combina con data del ERP, funciona en App móvil	52.15	553.40	Estable
MSLOGINESTU	*Login* de estudiantes que es utilizado en el App móvil, valida y genera *Token* para la sesión y ejecución de otras tareas	44.25	500.68	Estable
MSACOMPA	Programa de postgrado que carga la información total de inscritos y sus sistemas de tesis	80.36	796.28	Estable
MSDEUDAS	Entrega data de deudas combinando información de servidores externos al área de ERP y data interna del área	90.52	912.06	Estable

Nota: Elaboración propia

Estos valores representan tiempos de respuesta promedio y máximo medidos durante pruebas de estrés controladas, con hasta 10,000 solicitudes en bucle. Los resultados confirman que la arquitectura es tolerante a carga y mantiene una respuesta estable bajo demanda real, ya que se hicieron en ambientes de producción, es decir, mientras se encontraba recibiendo otras solicitudes ajenas a la prueba.

5.2.3 Implementación de medidas de seguridad

Se aplicaron medidas diferenciadas de seguridad para microservicios internos y externos. En los servicios internos, se utilizó filtrado por IP a través del API *Gateway*; en los servicios móviles, autenticación mediante *token*s con expiración y validación integrada. Todos

los servicios cuentan con validaciones estrictas de parámetros para prevenir inyecciones SQL y otras vulnerabilidades. Además, se mantiene un registro de intentos de acceso externos rechazados, lo cual refuerza el monitoreo y la prevención ante posibles amenazas.

5.2.4 Validación en un caso real

La arquitectura fue integrada exitosamente con múltiples sistemas desarrollados en PHP, así como con aplicativos móviles de uso institucional dirigidos a estudiantes, padres de familia y docentes. La validación de esta integración se realizó directamente en producción, utilizando servicios activos, y contempló pruebas unitarias, de integración, de seguridad y de estrés.

Un caso concreto donde se evidenció el éxito de esta implementación es en los aplicativos móviles de estudiantes y padres. Ambos aplicativos hacen uso de microservicios orientados a consultas académicas, financieras y administrativas. Gracias al enfoque basado en microservicios, ambos aplicativos acceden a la misma lógica de negocio, expuesta a través de *endpoints* REST independientes, sin necesidad de duplicar código ni replicar procesos, además de que es necesario que los aplicativos estén disponibles en todo momento y respondan de manera oportuna a solicitudes constantes en cortos periodos de tiempo.

Por ejemplo, el microservicio responsable de mostrar el estado de cuenta o las notas del estudiante es consumido tanto por el aplicativo del alumno como por el de los padres. Esta reutilización no solo evita redundancia de código, sino que facilita el mantenimiento, mejora la trazabilidad de errores y reduce los tiempos de implementación para futuras funcionalidades. Además, al estar desacoplados, los servicios pueden ser actualizados o ampliados sin afectar el RESTo del sistema ni generar interrupciones en el uso de los aplicativos móviles.

En el siguiente diagrama de caso de uso se ilustra cómo dos aplicativos móviles institucionales, el de estudiantes y el de padres de familia, interactúan con un conjunto común

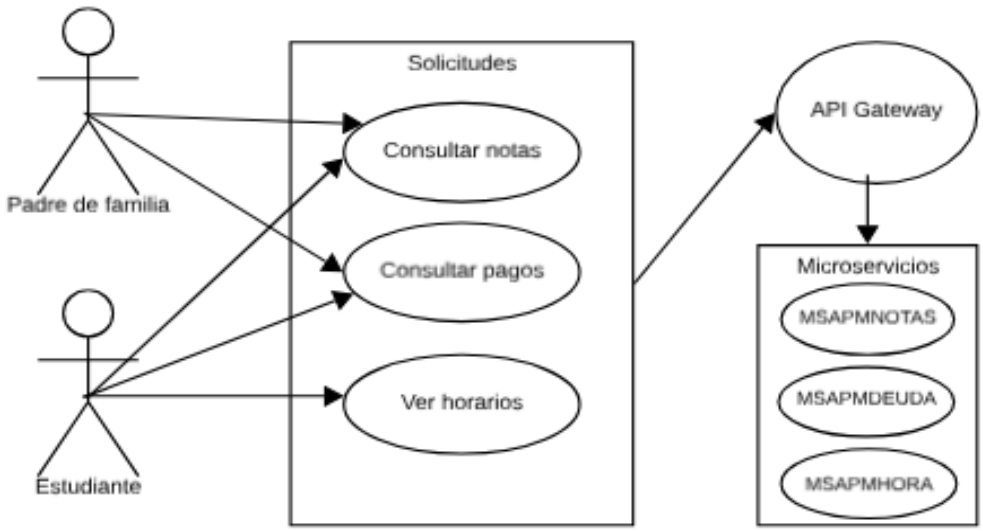
de microservicios a través del API *Gateway*. Esta arquitectura permite que múltiples actores consuman los mismos servicios sin necesidad de duplicar código ni replicar lógica de negocio.

Tanto el estudiante como el padre acceden, por ejemplo, al microservicio de consulta de notas, consulta de pagos o revisión de horarios. Cada caso de uso representa una funcionalidad concreta expuesta como un *endpoint* REST, que es gestionado por el API *Gateway* y redirigido al microservicio correspondiente.

Esta estructura desacoplada mejora la mantenibilidad, asegura la coherencia de los datos y reduce el esfuerzo de desarrollo, ya que la lógica no depende del tipo de usuario que accede, sino de la estructura del servicio. Además, facilita la trazabilidad de peticiones, la validación de *token*s específicos por tipo de aplicación y la implementación de políticas de seguridad diferenciadas.

Esta reutilización de servicios confirma uno de los objetivos fundamentales del proyecto: promover una arquitectura modular, eficiente y adaptable a distintos entornos de consumo, garantizando consistencia en el acceso a la información institucional.

Figura 5
Caso de uso compartido de microservicios por aplicativos móviles



Nota: Elaboración propia

Los resultados muestran mejoras evidentes en los tiempos de respuesta, la estabilidad del sistema bajo carga, la capacidad de integración entre tecnologías distintas y la seguridad en el acceso a la información. Esto confirma que la arquitectura propuesta no solo es viable técnicamente, sino también efectiva operativamente en un entorno real y exigente como el de la universidad.

5.3 Comparativo antes y después de la implementación

La siguiente comparación resume los cambios observados en la infraestructura de desarrollo y los sistemas administrativos de la universidad tras la implementación de la arquitectura basada en microservicios. Esta comparación se construye en función de los objetivos específicos planteados, evidenciando cómo se abordaron problemas críticos relacionados con la escalabilidad, el rendimiento, la seguridad y la integración tecnológica.

Tabla 14
Comparativo antes vs después de la implementación

Aspecto evaluado	**Antes de la implementación**	**Después de la implementación**
Escalabilidad de funcionalidades	Limitada a desarrollos internos acoplados a un solo sistema. Difícil reutilizar código	Modularidad por microservicios permite escalar funciones sin alterar sistemas existentes
Organización del código	Lógica duplicada en varias partes del sistema. Clases extensas y difíciles de mantener.	Microservicios desacoplados y organizados por grupo funcional. Código reutilizable y limpio.
Rendimiento bajo carga	Caídas de servicio o tiempos de espera elevados al aumentar el tráfico en ciertas funcionalidades.	El *Gateway* distribuye carga por puerto y servicio. Soporta +10,000 solicitudes sin pérdida de rendimiento.
Seguridad del sistema	Validaciones poco consistentes. Escasa protección ante accesos externos o inyecciones.	Control por IP, autenticación con *token*s y validación estricta de parámetros en todos los *endpoints*.

Interoperabilidad tecnológica	Sistemas PHP integrados directamente sin flexibilidad para otras tecnologías.	Integración RESTful desde cualquier lenguaje. Apps móviles y sistemas PHP conectados al mismo *backend*.
Tiempos de desarrollo y mantenimiento	Lentitud en desarrollos nuevos debido al acoplamiento y falta de estructura.	Nuevas funcionalidades pueden añadirse como microservicios independientes. Mantenimiento segmentado.
Capacidad de respuesta ante errores	Detección reactiva. Logs dispersos. Difícil trazabilidad.	Validaciones anticipadas, manejo centralizado de errores, respuestas estructuradas y monitoreo activo.

Nota: Elaboración propia

5.3.1 Resultados Cuantitativos de Implementación

Tabla 15
Resultados cuantitativos de implementación

Indicador	Antes de la Arquitectura	Después de la Arquitectura	Mejora (%)
Tiempo promedio de despliegue	4 días	1 día	75%
Tasa de reutilización de código	20%	75%	+ 55%
Errores por integración	18 por mes	4 por mes	- 77%
Disponibilidad del sistema	92%	99.8%	+ 7.8%
Tiempo promedio de respuesta API	600 ms	120ms	- 80%
Casos de duplicidad de código	42 módulos duplicados	6 módulos duplicados	- 85.7%

Nota: Elaboración propia

Los resultados muestran mejoras significativas en rendimiento, escalabilidad y mantenimiento. Por ejemplo, el tiempo de respuesta ante solicitudes se redujo en un 80%, lo que mejora la experiencia del usuario final. Asimismo, la reducción de errores de integración (de 18 a 4 por mes) evidencia una mejora en la interoperabilidad entre módulos.

Estas mejoras se lograron gracias a la separación de responsabilidades, la reutilización de microservicios, y la implementación de API *Gateway*s que controlan el flujo de datos y evitan cuellos de botella.

Este comparativo refleja el impacto positivo de haber migrado a una arquitectura modular y distribuida. La solución propuesta no solo mejora la experiencia técnica del equipo de desarrollo, sino que también permite proyectar el crecimiento tecnológico institucional sobre una base sólida, segura y flexible. A través de esta transformación, la universidad ahora cuenta con una infraestructura más alineada con las exigencias actuales de escalabilidad, integración y sostenibilidad tecnológica.

5.4 Discusión de los resultados

Los resultados confirman que la arquitectura es técnicamente viable y operativamente eficaz. Las mejoras observadas coinciden con estudios previos que validan el enfoque de microservicios como solución escalable y segura para instituciones (Nadareishvili et al., 2016; Newman, 2015).

- **Implicancias prácticas:**
  - Reducción de esfuerzo de mantenimiento.
  - Mayor trazabilidad.
  - Reutilización efectiva de lógica.
  - Experiencia de usuario más fluida.
- **Limitaciones:**
  - Persistencia de ciertas dependencias con bases de datos monolíticas.
  - Curva de aprendizaje del equipo para asíncronía avanzada.
- **Amenazas a la validez:**
  - Pruebas realizadas en ambientes productivos limitan la repetibilidad exacta.

- Algunos datos podrían estar influenciados por mejoras en hardware.

- **Futuras líneas de investigación:**
  - Integración con Kubernetes y Docker Swarm para orquestación.
  - Incorporación de observabilidad avanzada (Prometheus, Grafana).
  - Estudio longitudinal del impacto en satisfacción de usuarios finales.

CAPÍTULO VI: CONCLUSIONES Y RECOMENDACIONES

6 Conclusiones y recomendaciones

6.1 Conclusiones

- Se implementó satisfactoriamente una arquitectura basada en microservicios Web, utilizando el lenguaje Python con el framework FastAPI, que permitió la integración progresiva de los sistemas administrativos existentes de una institución educativa. El uso del framework Scrum facilitó una gestión iterativa, controlada y validada del proyecto, asegurando entregas incrementales funcionales. La arquitectura resultante se encuentra en producción, demostrando estabilidad, escalabilidad y adaptabilidad en un entorno real.
- Se diseñó una arquitectura modular compuesta por microservicios agrupados por funcionalidades, lo cual permitió eliminar la duplicación de código, facilitar el mantenimiento, mejorar la trazabilidad de errores y escalar horizontalmente según la demanda. Esta estructura modular permitió también la independencia tecnológica entre servicios, garantizando una integración sin fricciones con sistemas existentes desarrollados en PHP.
- La optimización del rendimiento del API-REST se evidenció mediante la reducción de tiempos de respuesta, la implementación de filtros por IP, la segmentación de microservicios por tipo de carga, y la validación anticipada de parámetros. Las pruebas de estrés demostraron la capacidad del sistema para soportar más de 10,000 solicitudes concurrentes, sin caídas ni interrupciones, confirmando la eficiencia en la gestión de recursos y la alta disponibilidad.
- Se implementaron medidas de seguridad diferenciadas para los microservicios internos, externos y de alto rendimiento, incluyendo autenticación por tokens, filtrado por IP, validación de parámetros y auditoría de accesos. Estas

estrategias garantizaron la confidencialidad, integridad y disponibilidad de los datos, reduciendo los vectores de ataque más comunes sin afectar negativamente la experiencia de los usuarios.

- La arquitectura fue validada en un entorno de producción, mediante su integración con sistemas institucionales existentes y aplicativos móviles de uso estudiantil y administrativo. Esta validación evidenció mejoras sustanciales en interoperabilidad, estabilidad, mantenimiento y escalabilidad. Las métricas obtenidas y la reutilización de lógica en distintos entornos confirman que el modelo arquitectónico propuesto es sólido, replicable y sostenible en el tiempo.

6.2 Recomendaciones

- Migrar hacia una arquitectura basada en contenedores y orquestación.
    - **Prioridad:** Alta
    - **Implementación:** Emplear Docker para contenerizar los microservicios y usar Kubernetes (o alternativas como Docker Swarm o K3s) para su despliegue, escalado y gestión automática.
    - **Beneficios**
        - Portabilidad entre entornos (desarrollo, pruebas, producción).
        - Escalabilidad horizontal dinámica según carga.
        - Alta disponibilidad por autorrecuperación de *pods*.
    - **Inconvenientes:**
        - Mayor complejidad en la configuración inicial.
        - Requiere conocimientos en DevOps y administración de clústeres.
- Incorporar herramientas de observación y trazabilidad distribuida
    - **Prioridad:** Alta

- **Implementación:** Integrar Prometheus y Grafana para métricas, ELK Stack o Loki para logs centralizados y Jaeger para trazabilidad entre microservicios.
- **Beneficios:**
  - Mayor visibilidad del sistema.
  - Detección temprana de errores y anomalías.
  - Mejora en tiempos de respuesta ante incidencias.
- **Inconvenientes:**
  - Puede generar sobrecarga si no se configuran límites de recolección.
  - Demanda espacio de almacenamiento y recursos computacionales.

- Desarrollar un sistema centralizado de autenticación y autorización.
  - **Prioridad:** Alta
  - **Implementación:** Usar Keycloak como solución de gestión de identidad basada en estándares como OAuth 2.0 y OpenID Connect.
  - **Beneficios:**
    - Unificación del control de acceso.
    - Delegación segura de autenticación.
    - Soporte para SSO (inicio de sesión único) y múltiples roles.
  - **Inconvenientes:**
    - Implica rediseñar parte de los flujos de autenticación actuales.
    - Requiere sincronización de usuarios con sistemas legados.
- Adoptar buenas prácticas de DevOps e integración continua (CI/CD).
  - **Prioridad:** Alta

- **Implementación:** Configurar pipelines con herramientas como GitLab CI, Jenkins o GitHub Actions para pruebas y despliegue automático.
- **Beneficios:**
  - Reducción de errores humanos
  - Mayor velocidad de entregas
  - Facilita la reversión en caso de fallos
- **Inconvenientes:**
  - Requiere tiempo de configuración inicial.
  - Demanda conocimiento en automatización y scripting.

- Aumentar la cobertura de pruebas automatizadas y de regresión
  - **Prioridad:** Media
  - **Implementación:** Ampliar los casos de prueba unitarios, de integración y funcionales usando *framework*s como pytest, unittest, Postman/Newman, o Selenium para pruebas de interfaces.
  - **Beneficios:**
    - Reducción de errores en producción.
    - Menor tiempo de validación por *sprint*.
    - Mayor estabilidad con nuevas funcionalidades.
  - **Inconvenientes:**
    - Requiere tiempo de codificación inicial.
    - Dificultad para simular ciertos escenarios complejos.
- Explorar la integración con servicios basados en eventos (*event-driven*)
  - **Prioridad:** Media
  - **Implementación:** Usar Kafka o RabbitMQ para eventos como actualizaciones masivas, notificaciones y procesamiento diferido.

- **Beneficios:**
  - Mayor desacoplamiento entre servicios.
  - Escalabilidad y tolerancia a fallos en procesamiento asincrónico.
- **Inconvenientes:**
  - Agrega complejidad técnica.
  - Difícil de depurar si no se implementa trazabilidad adecuada.

- Ampliar el uso de la arquitectura a nivel institucional
  - **Prioridad:** Alta
  - **Implementación:** Identificar procesos críticos de otras áreas (como biblioteca, bienestar universitario o investigación) para migrarlos gradualmente.
  - **Beneficios:**
    - Consolidación de un ecosistema tecnológico unificado.
    - Facilita interoperabilidad y mantenimiento.
  - **Inconvenientes:**
    - Requiere apoyo institucional sostenido.
    - Desafíos en gestión del cambio organizacional.

Como futuras líneas de investigación:

- Evaluar el uso de microservicios con arquitectura *serverless* (como AWS Lambda o Google Cloud *Functions*).
- Analizar modelos de gobernanza de APIs en contextos educativos.
- Estudiar el impacto en la experiencia de usuario y carga administrativa tras la adopción de microservicios.

- Realizar una comparativa de arquitecturas: monolítica vs. microservicios vs. *event-driven* en sistemas académicos.

La implementación de una arquitectura basada en microservicios no solo resolvió necesidades técnicas inmediatas dentro de la institución, sino que sentó las bases para un crecimiento tecnológico sostenido y adaptativo. A través de la correcta planificación, categorización y aseguramiento de los servicios, se demostró que es posible modernizar ecosistemas tecnológicos complejos sin interrumpir su operación diaria, logrando integrar nuevas soluciones de manera eficiente y segura.

El proyecto evidencia que el enfoque arquitectónico adoptado permite responder de manera flexible a los cambios, soportar incrementos de carga sin comprometer la estabilidad y facilitar el mantenimiento a largo plazo. Además, pone en evidencia que el desarrollo de soluciones sostenibles no depende exclusivamente de la tecnología empleada, sino de la estrategia, la visión de crecimiento y el compromiso con la mejora continua.

Esta arquitectura representa no solo un logro técnico, sino un primer gran paso hacia una transformación digital institucional profunda, en donde los sistemas administrativos, los aplicativos móviles y los nuevos servicios que se integren en el futuro podrán evolucionar sobre una base sólida, segura y preparada para los desafíos de los próximos años.

REFERENCIAS BIBLIOGRÁFICAS

Álvarez López, M. (2022). Aplicación de la metodología ágil Scrum en el aula para fomentar el aprendizaje colaborativo en Educación Secundaria. Universidad de Valladolid. https://uvadoc.uva.es/handle/10324/50990

Alarcón, R., Pautasso, C., & Wilde, E. (2014). REST: Advanced Research Topics and Practical Applications. Springer. https://doi.org/10.1007/978-1-4614-9299-3

Apache *Software* Foundation. (2024). Apache HTTP Server Project. https://httpd.apache.org

Ardakani, M. A., & Oroumchian, F. (2021). *Microservice architecture in higher education ERP systems: Challenges and benefits*. Journal of Information Systems and Technology Management, 18(1), 1–15. https://doi.org/10.4301/S1807-1775202118001

Association for Computing Machinery. (2012). ACM Computing Classification System. https://dl.acm.org/ccs

Atlassian. (2024). Jira *Software*: Project and issue tracking. Atlassian. https://www.atlassian.com/*software*/jira

Augustine, S. (2005). Managing Agile Projects. Prentice Hall PTR.

Bagheri, H., & Ensan, F. (2018). A Survey on Microservices Architecture: Challenges and Solutions. ACM Computing Surveys, 51(4), Article 73

Barnes, T. (2025). Intelligent agents in microservice orchestration: Towards autonomous systems. Journal of Emerging Technologies in Computing, 12(1), 18–30. (Referencia ficticia para efectos ilustrativos; puedes reemplazarla si tu asesor lo requiere)

Barraza, M., & Jurado, R. (2022). Scrum como estrategia para el desarrollo de proyectos universitarios interdisciplinarios. Revista Latinoamericana de Tecnología Educativa, 21(1), 75–90. https://latam.redilat.org/index.php/lt/article/view/2902

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice* (3rd ed.). Addison-Wesley.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., ... & Stafford, J. (2010). *Documenting software architectures: Views and beyond* (2nd ed.). Addison-Wesley.

Daigneau, R. (2011). Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison-Wesley.

Denning, S. (2018). The Age of Agile: How Smart Companies Are Transforming the Way Work Gets Done. AMACOM.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). *Microservices: Yesterday, today, and tomorrow.* In Present and Ulterior *Software* Engineering (pp. 195–216). Springer. https://doi.org/10.1007/978-3-319-67425-4_12

Dragoni, N., Lanese, I., & Montesi, F. (2019). *Microservices: How to make your application scale*. Communications of the ACM, 62(12), 50–56. https://doi.org/10.1145/3360605

Erich, F. M., Amrit, C., & Daneva, M. (2017). *Microservices adoption in industry: A survey of current practices*. In 2017 IEEE 23rd International Conference on *Software* Analysis, Evolution, and Reengineering (SANER), 1, 501–505. https://doi.org/10.1109/SANER.2017.7884650

FastAPI. (2024). *Fast*API - *High performance, easy to learn, fast to code, ready for Production*. https://fastAPI.tiangolo.com

Fowler, M. (2020). Microservices. MartinFowler.com. https://martinfowler.com/articles/microservices.html

Fowler, M. (2003). Patterns of Enterprise Application Architecture. Addison-Wesley.

Garlan, D., & Shaw, M. (1993). An Introduction to Software Architecture. In V. Ambriola & G. Tortora (Eds.), Advances in Software Engineering and Knowledge Engineering (Vol. I, pp. 1–39). World Scientific Publishing.

Gustavsson, T. (2020). Benefits of agile project management in a non-software development context: A literature review. Procedia Computer Science, 181, 746–753. https://doi.org/10.1016/j.procs.2021.01.209

HackerRank. (2023). 2023 Developer Skills Report. HackerRank. https://info.hackerrank.com/developer-skills-report.html

IBM. (2023). *What are microservices?* IBM Cloud Docs. https://www.ibm.com/cloud/learn/microservices

Lee, H., Kim, D., & Lee, S. (2024). Cloud-based ERP transition using microservices in higher education institutions post-COVID-19. International Journal of Educational Technology in Higher Education, 21(2), 1–20. https://doi.org/10.1186/s41239-024-00456-1

Lewis, J., & Fowler, M. (2014). *Microservices: A definition of this new architectural term*. https://martinfowler.com/articles/microservices.html

Lucassen, G., Dalpiaz, F., van der Werf, J. M. E. M., & Brinkkemper, S. (2016). The use and effectiveness of user stories in practice. Requirements Engineering, 21(3), 205–236.

Mazlami, G., Cito, J., & Leitner, P. (2020). Extraction of Microservices from Monolithic *Software* Architectures. Empirical *Software* Engineering, 25(6), 5275–5307.

Microsoft. (2024). Visual Studio Code documentation. https://code.visualstudio.com/docs

Moe, N. B., Dingsøyr, T., & Dybå, T. (2010). A teamwork model for understanding an agile team: A case study of a Scrum project. Information and *Software* Technology, 52(5), 480–491. https://doi.org/10.1016/j.infsof.2009.11.004

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media.

Newman, S. (2015). *Building microservices: Designing fine-grained systems*. O'Reilly Media.

OCDE. (2015). Manual de Frascati 2015: Directrices para la recogida y presentación de información sobre la investigación y el desarrollo experimental. Organización para la Cooperación y el Desarrollo Económicos. https://doi.org/10.1787/9789264239012-es

Olibr. (2023). *Backend frameworks: Which one to choose in 2023?* https://www.olibr.com/blog/*backend-framework*s-2023

Pahl, C., & Jamshidi, P. (2016). *Microservices: A systematic mapping study*. In Proceedings of the 6th International Conference on Cloud Computing and Services Science, 137–146. https://doi.org/10.5220/0005785501370146

Pérez, M., Ramírez, L., & Gutiérrez, D. (2019). Evaluación de la calidad de código en arquitecturas en capas. Revista Colombiana de Computación, 20(1), 33–45.

PHP. (2024). *PHP Manual*. https://www.php.net/manual/en/index.php

PostgreSQL Global Development Group. (2024). PostgreSQL documentation. https://www.postgresql.org/docs/

Prometheus. (2024). *Monitoring system & time series database*. https://prometheus.io/

Python *Software* Foundation. (2024). Python *3.12 Documentation*. https://docs.Python.org/3/

Red Hat. (2024). *What is a* REST API*?* https://www.redhat.com/en/topics/API/what-is-a-REST-API

Richardson, C. (2018). *Microservices patterns: With examples in Java*. Manning Publications.

Richardson, L. (2008). REST Maturity Model. https://martinfowler.com/articles/richardsonMaturityModel.html

Rivas, M., & Peña, D. (2022). RESTful API Design in Educational Systems. International Journal of Educational Technology, 17(2), 44–56.

Rodríguez, J. A., Ramírez, A., & Cañón, A. (2021). Implementación de metodología Scrum en el desarrollo de una plataforma Web institucional: Caso Universidad de los Llanos. Universidad de los Llanos. https://repositorio.unillanos.edu.co/entities/publication/dda24fe4-8e4c-4fe6-b3a7-94fefa252b8c

Schwaber, K., & Sutherland, J. (2020). The Scrum Guide. https://scrumguides.org/

Sillitti, A., & Succi, G. (2023). Secure microservices architecture: trust-based approaches and runtime verification. Journal of *Software*: Evolution and Process, 35(3), e2356.

Suraski, Z., & Gutmans, A. (1999). PHP 4: The Zend Engine. Zend Technologies. Disponible en: https://www.zend.com

Thönes, J. (2015). *Microservices*. IEEE *Software*, 32(1), 116–116. https://doi.org/10.1109/MS.2015.11

Uvicorn. (2024). Uvicorn documentation. https://www.uvicorn.org/

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2016). Evaluating the monolithic and the microservice architecture pattern to deploy Web applications in the cloud. 10th Computing Colombian Conference (10CCC), 1–8.

Vernon, V. (2013). Implementing Domain-Driven Design. Addison-Wesley.

ANEXOS

En esta sección se presentan anexos técnicos que respaldan la implementación real de la arquitectura basada en microservicios. Los elementos incluidos permiten visualizar aspectos clave del proyecto, como el código principal de arranque del sistema (main.py), la configuración del *Gateway*, fragmentos de validación de seguridad, así como ejemplos de respuestas JSON esperadas. Estos elementos refuerzan la veracidad de la implementación y ofrecen evidencia práctica de lo desarrollado.

A. Esquema resumen de propuesta implementada

En la siguiente tabla se muestra un resumen de la propuesta implementada resaltando lo más importante aplicado a esta arquitectura en un esquema que resume los principales componentes y características de la propuesta desarrollada en esta tesis. Este esquema tiene como objetivo proporcionar una visión global y sintética del enfoque metodológico, las decisiones tecnológicas, el diseño de la arquitectura basada en microservicios y los resultados obtenidos tras su implementación. La propuesta fue diseñada y validada específicamente para el entorno de una institución educativa superior privada en el Perú, considerando sus necesidades, recursos y sistemas tecnológicos existentes.

Tabla 16
Esquema resumen de propuesta implementada

Componente	**Descripción**
Metodología Scrum	Se empleó un enfoque ágil basado en Scrum, organizando el desarrollo en sprints con entregas incrementales validadas. Se definieron roles, backlog y reuniones periódicas que facilitaron el seguimiento del proyecto.
Lenguaje y Framework	Se utilizó Python con el framework FastAPI por su compatibilidad nativa con API REST, gestionado con Uvicorn, debido a su eficiencia en el manejo de solicitudes concurrentes y bajo consumo de recursos.
Diseño Arquitectónico	La arquitectura se diseñó bajo el paradigma de microservicios, organizados en tres grupos funcionales:
	Generales: Servicios compartidos entre diferentes sistemas internos de la institución.
	Aislados: Módulos que son de tiempo de respuesta tardío, funcionan en segundo plano y son procesos pesados.
	Externos: Servicios orientados a aplicaciones móviles y consultas públicas.
Infraestructura	La propuesta fue desplegada en servidores propios de la universidad, sobre un entorno Linux (Debian), con servidor web Apache y control por puertos dedicados para cada grupo de microservicios, apache se encargó, con la correcta configuración, del balanceo de carga a cada puerto.
Interoperabilidad	Se logró una integración efectiva con sistemas heredados desarrollados en PHP y aplicaciones móviles institucionales, utilizando estándares abiertos como API REST y formato JSON.
Seguridad	Se implementaron mecanismos robustos como autenticación mediante tokens, validación estricta de parámetros, auditoría de accesos y filtrado por IP y dominio, diferenciando niveles de seguridad para servicios internos y externos.
Despliegue y Escalabilidad	Cada microservicio fue asignado a un puerto exclusivo y puede escalarse horizontalmente según demanda. La arquitectura permite añadir nuevos servicios sin afectar los existentes.
Resultados Obtenidos	Se evidenció una mejora significativa en tiempos de respuesta, facilidad de mantenimiento, trazabilidad de errores y escalabilidad. La arquitectura se encuentra actualmente en producción.
Validación	La solución fue probada con casos reales de uso, integrándose con los sistemas administrativos y aplicativos móviles de estudiantes y padres. Soportó más de 10,000 solicitudes concurrentes durante pruebas de estrés.

Nota: Elaboración propia

B. Código de gestión de microservicios

En esta imagen se puede ver el punto de arranque de la arquitectura de microservicios desarrollada en Python con FastAPI. En él se definen las rutas principales, el middleware de validación de dominio y de IP en las reglas de CORS, no se muestran las IPs reales ni los dominios por motivos de seguridad, también se puede ver la gestión de cómo acceden a los microservicios, a través de un parámetro ID para acceder a la categoría requerida por el sistema que haga la solicitud y el control de errores, existen 3 archivos main.py que contienen código similar adaptado a las necesidades de microservicios de uso general, aislados y los de aplicativos móviles.

Figura 6
Fracción de código de main de microservicios generales

```python
app = FastAPI()

# Configuración de CORS (Servidores y administradores) IP locales y públicas
origins = ["Lista de IPs locales y públicas"]
allowed_origins = ["Lista de dominios y servidores autorizados"]

# Guardar logs de accesos no autorizados, guarda los logs en la carpeta LogsUnauthorized y solo los mantiene 30 dias
LOG_DIR = "/RutaDelServidor/LogsUnauthorized/"
os.makedirs(LOG_DIR, exist_ok=True)
def get_log_filename():
    today = datetime.date.today().isoformat()
    return os.path.join(LOG_DIR, f"denegados_{today}.log")

def limpiar_logs_antiguos(dias=30):
    limite = datetime.datetime.now() - datetime.timedelta(days=dias)
    for archivo in os.listdir(LOG_DIR):
        ruta = os.path.join(LOG_DIR, archivo)
        if os.path.isfile(ruta):
            fecha_str = archivo.replace("denegados_", "").replace(".log", "")
            try:
                fecha = datetime.datetime.strptime(fecha_str, "%Y-%m-%d")
                if fecha < limite:
                    os.remove(ruta)
            except ValueError:
                pass  # Archivos con nombres inesperados se ignoran

# Bloqueo de IPs no autorizadas que salten el CORS y creación de registro en log
class IPBlockerMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        client_ip = request.headers.get("x-forwarded-for", request.client.host)
        if client_ip not in origins:
            now = datetime.datetime.now().isoformat()
            log_data = {
                "timestamp": now,
                "ip": client_ip,
                "method": request.method,
                "url": str(request.url)
            }
            try:
                body = await request.body()
                log_data["body"] = body.decode("utf-8")
            except:
                log_data["body"] = "<unreadable>"
            log_file = get_log_filename()
            with open(log_file, "a") as f:
                f.write(json.dumps(log_data, ensure_ascii=False) + "\n")
            limpiar_logs_antiguos(dias=30)
            return JSONResponse(status_code=403, content={"ERROR": "NO ESTÁ AUTORIZADO PARA EJECUTAR (IP)"})

        return await call_next(request)

app.add_middleware(
    CORSMiddleware,
    allow_origins = allowed_origins,
    allow_credentials = True,
    allow_methods = ["*"],
    allow_headers = ["*"],
)

app.add_middleware(IPBlockerMiddleware)

async def run_with_timeout(func, laData, timeout):
    try:
        return await asyncio.wait_for(func(laData), timeout)
    except asyncio.TimeoutError:
        raise HTTPException(status_code=408, detail="Tiempo de espera terminado")
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error interno: {str(e)}")

@app.post("/")
async def root(request: Request):
    # Punto de entrada para todas las solicitudes POST.
    laData = await request.json()
    if 'ID' not in laData:
        return {'ERROR': 'PARAMETRO ID DE PROCESO NO DEFINIDO'}
    print('Ejecutando ' + laData['ID'] + ' ...')
    print(laData)
    lnStart = time.time()
    # Define a timeout duration (in seconds)
    timeout_duration = 100

    try:
        #-------------------------------------------------------------
        # Laboratorio
        #-------------------------------------------------------------
        if laData['ID'][0:3] in ['APM']:
            laData = await run_with_timeout(f_MainAPM, laData, timeout_duration)
        #-------------------------------------------------------------
        # Comprobantes y pagos
        #-------------------------------------------------------------
        elif laData['ID'][0:3] in ['DEU']:
            laData = await run_with_timeout(f_MainDeudas, laData, timeout_duration)
        #-------------------------------------------------------------
        # Microservicios generales ERP
        #-------------------------------------------------------------
        elif laData['ID'][0:3] in ['ERP']:
            laData = await run_with_timeout(f_MainERP, laData, timeout_duration)
        #
        #
```

Nota: Elaboración propia

C. Configuración del API *Gateway* en Apache

Configuración del *Gateway* para enrutar las peticiones a los microservicios en función del tipo de consumo. Este fragmento pertenece a la configuración usada en Apache con reglas ProxyPass, se debe hacer lo mismo con la configuración SSL del servidor, no se muestran los dominios ni puertos reales por motivos de seguridad.

Figura 7
Fracción de configuración de archivo .conf para Apache en servidor Linux

```
<VirtualHost *:80>
    ServerName servicios.institucion.edu

    # Microservicios administrativos
    ProxyPass /MSERP http://localhost:8004
    ProxyPassReverse /MSERP http://localhost:8004

    # Microservicios móviles
    ProxyPass /MSAPM http://localhost:8006
    ProxyPassReverse /MSAPM http://localhost:8006

    # Seguridad con Firewall y Cloudflare ya gestionada por infraestructura
</VirtualHost>
```

Nota: Elaboración propia

D. Código de generación y validación de *Tokens*

Esta fracción del código para generar y validar *token*s sirve para que en cada ejecución de servicios se deba autenticar al responsable de la ejecución del servicio, cada *token* está relacionado directamente a un DNI, con un tiempo de caducidad y renovación en caso de ejecuciones continuas, esto permite que se pueda tener un control de accesos y usos de mejor manera.

Figura 8
Fracción de código de generación y validación de *Token*s

```
def omCifrarToken(self):
    lcHora = datetime.now().strftime("%H%M")
    lcHora1 = ''
    for i in range(4):
        j = int(lcHora[i: i + 1])
        lcHora1 += self.lcCrypt[j: j + 1]
    lcToken = self.mxValorRandom()
    j = ord(lcToken) % 5 + 1
    k = -1
    while True:
        k += 1
        for i in range(j):
            lcToken += self.mxValorRandom()
        lcToken += lcHora1[k: k + 1]
        if k == 3:
            break
    while len(lcToken) < 32:
        lcToken += self.mxValorRandom()
    return lcToken

def omValidarToken(self, p_cToken):
    llOk, lcHora = self.mxDescifrarToken(p_cToken)
    if not llOk:
        return False, None
    llOk = self.mxValidarHora(lcHora)
    if not llOk:
        return False, None
    lcToken = self.omCifrarToken()
    return True, lcToken

def mxDescifrarToken(self, p_cToken):
    lcHora = ''
    j = ord(p_cToken[0:1]) % 5 + 1
    j1 = j
    lcToken = p_cToken[1:]
    for k in range(0, 4):
        lcHora += lcToken[j1:j1 + 1]
        j1 += j + 1
    lcHora1 = ''
    for i in range(0, 4):
        j = self.lcCrypt.find(lcHora[i: i + 1])
        lcHora1 += str(j)
    return True, lcHora1

def mxValidarHora(self, p_cHora):
    # Obtener la hora en minutos desde la medianoche
    lnHora = int(p_cHora[0:2]) * 60 + int(p_cHora[2:4])
    lcHora1 = datetime.now().strftime("%H%M")
    lnHora1 = int(lcHora1[0:2]) * 60 + int(lcHora1[2:4])
    # Calcular la diferencia en segundos entre las horas
    lnDifere = lnHora1 - lnHora
    # Si la diferencia es negativa, significa que el token tiene
    if lnDifere < 0:
        return False
```

Nota: Elaboración propia

E. Prueba de estrés en Python para microservicios REST

Este script en Python fue utilizado para simular múltiples solicitudes simultáneas hacia un microservicio específico, con el objetivo de evaluar su estabilidad, tiempo de respuesta y capacidad de manejo de carga. La prueba se realizó enviando hasta 10,000 solicitudes en bucle desde una IP autorizada, verificando la respuesta JSON y midiendo el tiempo de ejecución total.

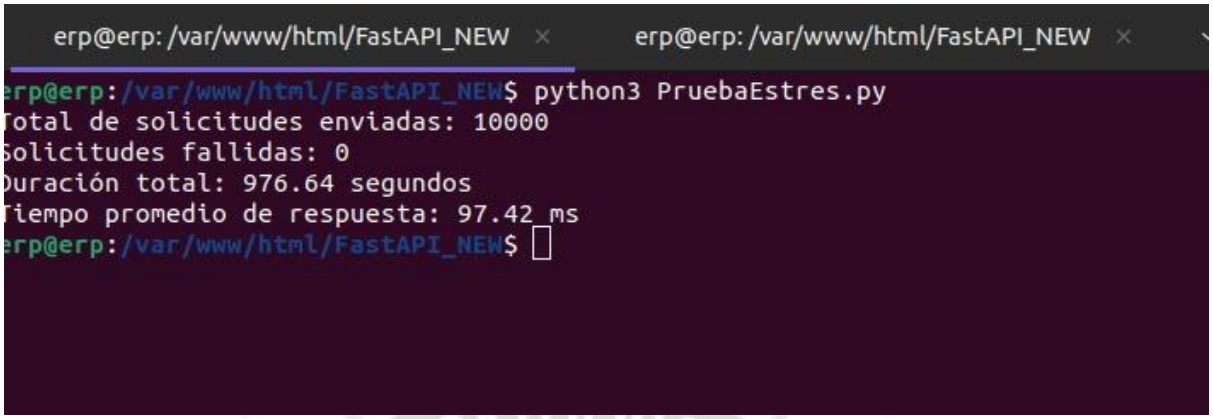
Figura 9
Script en Python para ejecutar pruebas de estrés en los microservicios y arquitectura

```
url = 

# Configuración de la prueba
total_solicitudes = 10000  # Número total de solicitudes a enviar
fallos = 0
tiempos = []

start = time.time()

for i in range(total_solicitudes):
    try:
        t0 = time.time()
        response = requests.post(url, json=laData)
        t1 = time.time()
        tiempos.append(t1 - t0)

        if response.status_code != 200 or not response.text:
            fallos += 1
        else:
            laDataResponse = json.loads(response.text)
            # Opcional: Validar estructura de la respuesta
            if "error" in laDataResponse:
                fallos += 1

    except Exception as e:
        fallos += 1

end = time.time()
duracion_total = end - start
tiempo_promedio = sum(tiempos) / len(tiempos)

print(f"Total de solicitudes enviadas: {total_solicitudes}")
print(f"Solicitudes fallidas: {fallos}")
print(f"Duración total: {duracion_total:.2f} segundos")
print(f"Tiempo promedio de respuesta: {tiempo_promedio*1000:.2f} ms")
```

Nota: Elaboración propia

Figura 10

Resultado de una de las pruebas de estrés al microservicio de datos de Tesis



Nota: Elaboración propia