**UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA SEDE
VIÑA DEL MAR - CHILE**

"Arquitectura de software para Plataforma Académica del Cuerpo de Bomberos Viña del Mar"

CRISTIAN IGNACIO MONTIEL CATALAN

cristian.montiel@usm.cl

Trabajo de Titulación en formato Tesina para optar al Título de Ingeniería en Informática

Profesor Guía: Dagoberto Cabrera
Profesor correferente: Diego Cáceres

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): **X** Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: _**Arquitectura de software para Plataforma Académica del Cuerpo de Bomberos Viña del Mar** _

Nombre del candidato(a): ___________**Cristian Ignacio Montiel Catalan**_______________________________

Carrera / Grado: __________**Ingeniería en Informática**_____________________________________

Campus: _______Sede Viña del Mar_______________ **Departamento:** _____ELINF_________________________________

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, _____DAGOBERTO CABRERA TAPIA_________, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

X El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por **(marcar una opción)**:

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

____

____

____

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: _____23-01-2026__________ **Firma:** ________

Estudiante o Candidato(a):

Fecha: _______23/01/2026_______ **Firma:** ____________________

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

Resumen

El presente trabajo de título aborda el diseño e implementación de una plataforma digital centralizada para la gestión académica del Cuerpo de Bomberos de Viña del Mar, con el objetivo de resolver las problemáticas actuales relacionadas con la dispersión de la información curricular, la falta de trazabilidad en el historial de los voluntarios y los tiempos excesivos en procesos manuales.

Para lograrlo, se diseñó una arquitectura de software desacoplada del tipo Cliente-Servidor, utilizando Angular para la interfaz de usuario y Django REST Framework para la lógica de negocio, documentando el diseño bajo el estándar de "4+1 Vistas" para asegurar su mantenibilidad. La solución se desplegó sobre una infraestructura Cloud Native en Google Cloud Platform, aprovechando la escalabilidad automática de servicios Serverless (Cloud Run) y bases de datos gestionadas, además de integrar un asistente virtual basado en Inteligencia Artificial (Gemini) para democratizar el acceso a la información técnica.

Los resultados principales, validados mediante métricas de telemetría en la nube, demuestran que la plataforma soporta la concurrencia operativa de la institución garantizando una disponibilidad 24/7, reduciendo drásticamente los tiempos de gestión y optimizando los costos mediante un modelo de pago por uso. La relevancia de este proyecto radica en la transformación digital de la academia, entregando una herramienta moderna que profesionaliza la gestión actual y establece una base tecnológica escalable para el crecimiento futuro de la institución.

Palabras clave: Gestión Académica, Arquitectura Desacoplada, Cloud Native, Serverless, Inteligencia Artificial.

Glosario

- **Autenticación**: sistema de verificación en la cual un usuario o aplicación se identifica.
- **Nube**: es una red de servidores ubicados en centros de datos remotos que ofrecen servicios como almacenamiento de archivos, potencia informática y software a través de Internet, permitiendo a los usuarios acceder a sus datos y aplicaciones desde cualquier lugar y dispositivo con conexión a la web.
- **Escalabilidad**: capacidad de un sistema de mantener un flujo de trabajo que aumenta sin presentar perdida de funcionamiento o rendimiento.
- **Métodos tradicionales**: véase todos los procesos de trabajo en el cual no se hace uso de tecnologías de herramientas digitales o el uso de aplicaciones digitales, de los cuales se destacan el respaldo en información en papel.
- **Módulos**: unidades lógicas o componentes del sistema que agrupan funcionalidades relacionadas entre sí. Se usan para organizar, dividir y estructurar el software, facilitando su mantenimiento, escalabilidad y reutilización.
- **Claims**: son declaraciones (afirmaciones) o piezas de información en formato de par clave-valor que se incluyen dentro del token que transmiten datos sobre la identidad y privilegios del usuario.
- **API (Interfaz de Programación de Aplicaciones)**: Conjunto de reglas y definiciones que permite que dos aplicaciones de software se comuniquen entre sí. En este proyecto, actúa como el "puente" que conecta la interfaz visual (Frontend) con la lógica del servidor (Backend).
- **Backend**: La parte del sistema que se ejecuta en el servidor y no es visible para el usuario final. Es el "cerebro" de la aplicación, encargado de procesar los datos, aplicar las reglas del negocio y comunicarse con la base de datos. En este proyecto, está construido con Django.
- **Cloud Computing (Computación en la Nube)**: Modelo tecnológico que permite acceder a recursos informáticos (servidores, bases de datos, almacenamiento) a través de internet bajo demanda, en lugar de tener servidores físicos locales. Se utiliza Google Cloud Platform (GCP).
- **Cloud Run**: Servicio de Google Cloud que permite ejecutar aplicaciones empaquetadas en contenedores de manera automática. Su principal ventaja es que es "Serverless" (sin servidor), lo que significa que Google administra la infraestructura y el sistema escala automáticamente según la cantidad de usuarios.
- **Contenedor (Docker)**: Una unidad de software estandarizada que empaqueta el código y todas sus dependencias (librerías, configuraciones) para que la aplicación se ejecute de manera rápida y fiable en cualquier entorno informático. Es como una "caja cerrada" que contiene todo lo necesario para que el sistema funcione.
- **CRUD**: Acrónimo de Create, Read, Update, Delete (Crear, Leer, Actualizar, Eliminar). Se refiere a las cuatro operaciones básicas que se pueden realizar sobre los datos en un sistema informático.

- **Deploy** (Despliegue): El proceso de trasladar el código de la aplicación desde el entorno de desarrollo (la computadora del programador) hacia el entorno de producción (la nube), donde estará disponible para los usuarios finales.
- **Endpoint**: Una dirección específica (URL) dentro de la API a la que el cliente puede enviar una solicitud para realizar una acción concreta, como "obtener lista de bomberos" o "crear un curso".
- **Frontend**: La parte del sistema con la que interactúa directamente el usuario (la interfaz visual, botones, formularios). Se ejecuta en el navegador web y, en este proyecto, está construida con Angular.
- **Framework**: Un esquema o estructura de trabajo predefinida que ofrece herramientas y funciones genéricas para facilitar y acelerar el desarrollo de software. Django y Angular son los frameworks utilizados.
- **JWT (JSON Web Token)**: Un estándar abierto para la creación de tokens de acceso. Funciona como una "credencial digital" segura y encriptada que el usuario presenta al servidor para demostrar su identidad y acceder a recursos protegidos sin tener que enviar su contraseña en cada petición.
- **Latencia**: El tiempo que tarda un paquete de datos en viajar desde un punto a otro. En el contexto web, es el tiempo de espera que percibe el usuario desde que hace clic hasta que el servidor responde.
- **MVC (Modelo-Vista-Controlador)**: Patrón de arquitectura de software que separa los datos (Modelo), la interfaz de usuario (Vista) y la lógica de control (Controlador). Ayuda a organizar el código y facilita el mantenimiento.
- **ORM (Mapeo Objeto-Relacional)**: Herramienta que permite a los desarrolladores interactuar con la base de datos utilizando código de programación (objetos) en lugar de escribir consultas SQL complejas. Actúa como un "traductor" entre Python y la base de datos.
- **Serverless (Sin Servidor)**: Modelo de ejecución en la nube donde el proveedor (Google) gestiona dinámicamente la asignación de recursos del servidor. El desarrollador solo se preocupa por el código, no por el mantenimiento del hardware o sistema operativo.
- **SPA (Single Page Application)**: Tipo de aplicación web que carga una sola página HTML y actualiza dinámicamente el contenido a medida que el usuario interactúa con ella, sin necesidad de recargar la página completa. Esto ofrece una experiencia más fluida, similar a una aplicación de escritorio.
- **uWSGI**: Servidor de aplicaciones que actúa como intermediario entre el servidor web y la aplicación Python (Django). Es fundamental para manejar múltiples solicitudes simultáneas de usuarios de manera eficiente.

Contenido

Resumen ... 2
Glosario ... 3
1. Introducción ... 7
  1.1 Contexto ... 7
  1.2 Problemática ... 7
  1.3 Propuesta ... 8
  1.4 Objetivo general del Proyecto ... 9
  1.5 Objetivos Específicos ... 9
  1.6 Justificación de proyecto ... 9
  1.7 Metodología ... 10
  1.8 Breve organización del informe ... 11
2. Marco teórico ... 11
  2.1 Fundamentos de la arquitectura de software ... 12
  2.2 Tecnologías empleadas ... 13
  2.3 Uso de API en la arquitectura de la plataforma ... 16
  2.4 Mecanismos de autenticación y control de acceso ... 17
3. Diseños de la Plataforma ... 18
  3.1 Arquitectura de Software (Modelo 4+1 Vistas) ... 18
    Vista Lógica (Logical View) ... 18
    Vista de Procesos (Process View) ... 21
    Vista de Desarrollo (Development View) ... 22
    Vista Física o de Despliegue (Physical View) ... 23
    Vista de Escenarios (+1) ... 23
  3.2 Diseño de modelos de datos ... 25
  3.3 Diseño de API ... 31
    Organizaciones (api/organizaciones/) ... 32
    Cursos (api/cursos/) ... 33
    Usuarios (api/users/) ... 34
    Avisos (api/avisos/) ... 34
    Integración con IA (api/chatbot/) ... 35
  3.4 Arquitectura de Infraestructura y Despliegue en la Nube (Google Cloud Platform) ... 36
    Introducción a la Infraestructura Cloud ... 36

Diseño de la Arquitectura de Despliegue 36
Implementación de Servicios de Cómputo: Cloud Run 38
Persistencia y Almacenamiento 40
Escalabilidad y Modelo de Costos 40
Elección de infraestructura 41
Análisis de Alternativas de Infraestructura: Serverless vs. IaaS (Máquinas Virtuales) 41
Conclusión de la Selección de Infraestructura 42
4. Validación de solución 43
4.1 Validación de Infraestructura y Rendimiento 43
4.2 Análisis Comparativo: Situación Actual vs. Solución Propuesta 45
4.3 Conclusiones de la Validación 46
5. Conclusión general y trabajo futuro 46
5.1 Síntesis y Cumplimiento de Objetivos 46
5.2 Aportes e Impacto en la organización 46
5.3 Alcances y Limitaciones de la Propuesta 47
5.4 Recomendaciones y Trabajo Futuro 47
5.5 Conclusiones generales 48
6. Agradecimientos 48
7. Referencias 49

1. Introducción

1.1 Contexto

La Academia del Cuerpo de bomberos de Viña del Mar presenta una serie de desafíos, de los cuales se destaca el seguimiento profesional de cada bombero que compone sus filas y trazabilidad de información acerca de ellos, entre otros. Actualmente, la organización utiliza métodos manuales o herramientas no centralizadas para el manejo de información respecto al historial educativo, lo que genera ineficiencias, errores en la administración, y dificultades para lograr acceder a datos claves de las personas que componen sus filas.

En el caso particular de las diez compañías que conforman el cuerpo de bomberos en estudio, cuentan con aproximadamente 600 voluntarios, se han identificado fallas importantes relacionadas con el conocimiento de la capacidad operativa y niveles formativos que poseen los profesionales, al almacenamiento de documentos de respaldo, la trazabilidad del alumnado y la pérdida de información. Estas problemáticas afectan directamente en la estructura curricular de la organización, comprometiendo tanto la capacitación y la disponibilidad de personal cualificado para operar.

Los intentos de solucionar esta situación han sido pocos, de los cuales se destaca la intención de utilizar un sistema implementado en Estados Unidos, iniciativa que finalmente no se concretó. Únicamente se hace uso de hojas de cálculo en Excel como herramienta TI como complemento a los métodos tradicionales.

1.2 Problemática

La administración académica del Cuerpo de Bomberos de Viña del Mar carece actualmente de una plataforma de gestión centralizada, dependiendo exclusivamente de métodos manuales y hojas de cálculo. Esta precariedad tecnológica ha derivado en la desorganización de los recursos académicos, la pérdida recurrente de documentación y la imposibilidad de realizar una trazabilidad curricular en tiempo real.

Esta situación genera un impacto negativo en tres dimensiones críticas:

- **Nivel Operativo:** Ineficiencia administrativa y descoordinación en los procesos de capacitación, lo que retrasa la formación de los voluntarios.
- **Nivel Estratégico:** Incapacidad para generar estadísticas fidedignas que apoyen la toma de decisiones y el seguimiento de competencias profesionales.
- **Nivel Institucional:** Debilitamiento de la proyección corporativa y riesgo potencial en la calidad del servicio de emergencia entregado a la comunidad, al no poder garantizar la certificación de las habilidades del personal.

En consecuencia, resulta imperativo implementar una herramienta tecnológica que no solo automatice la gestión interna, sino que garantice la transparencia, la seguridad de la información y la excelencia formativa requerida para la seguridad comunal.

1.3 Propuesta

La propuesta presentada en este trabajo se enfoca en resolver las limitaciones actuales mediante el diseño e implementación de una solución tecnológica especializada para la Academia del Cuerpo de Bomberos de Viña del Mar. Esta solución consiste en una plataforma digital centralizada, adaptable a las necesidades específicas de la organización y accesible para todos los usuarios que interactúan con el entorno académico bomberil.

La plataforma permitirá la gestión integral de la información académica, incluyendo cursos, certificados, historial de capacitaciones, documentación de respaldo y datos personales de los voluntarios. Además, se contempla la incorporación de perfiles de usuario diferenciados (administrador, instructor, alumno), cada uno con vistas y funcionalidades específicas, lo que facilitará la navegación, el acceso a la información pertinente y la ejecución de tareas según el perfil.

Desde el punto de vista técnico, se propone una **arquitectura de software desacoplada del tipo Cliente-Servidor**, la cual separa la lógica de negocio (Backend) de la interfaz de usuario (Frontend) mediante una API REST. Esta decisión responde a la necesidad de contar con una solución moderna, altamente escalable y mantenible, que aproveche las ventajas de la infraestructura en la nube para garantizar la disponibilidad y permita una evolución tecnológica ágil a futuro.

La plataforma incluirá mecanismos de autenticación segura por roles (JWT), conexión con bases de datos gestionadas en la nube, formularios dinámicos para la gestión de contenidos académicos, *dashboards* personalizados y generación de reportes automáticos. Todo esto contribuirá a mejorar la trazabilidad curricular, reducir los tiempos de respuesta, evitar la pérdida de información y fortalecer la capacidad operativa de la institución.

En definitiva, esta propuesta busca transformar digitalmente la gestión académica del Cuerpo de Bomberos de Viña del Mar, elevando los estándares de calidad, eficiencia y transparencia, y consolidando una herramienta que apoye la formación continua de sus integrantes.

1.4 Objetivo general del Proyecto

Diseñar una arquitectura de software desacoplada y escalable para una plataforma académica digital del Cuerpo de Bomberos de Viña del Mar, que permita integrar de forma segura y eficiente los procesos de gestión curricular, trazabilidad de capacitaciones y acceso segmentado a la información mediante tecnologías web, API REST y servicios de infraestructura en la nube.

1.5 Objetivos Específicos

- Diseñar la arquitectura funcional y segura de la plataforma.
- Diseñar la intercomunicación mediante API de los componentes que conforman tanto el desarrollo Backend como el Frontend.
- Seleccionar tecnologías/frameworks óptimos para el desarrollo de la plataforma.
- Diseñar la conexión de la plataforma con la base de datos y almacenamiento en alguno de los servicios de infraestructura en la nube (véase AWS, Azure, GCP, entre otros).
- Seleccionar los proveedores y servicios de la nube más adecuados para llevar a producción.
- Diseñar los modelos de datos.

1.6 Justificación de proyecto

La gestión académica actual del Cuerpo de Bomberos de Viña del Mar se ve limitada por procesos manuales y dispersos, lo que genera pérdida de información crítica y dificulta la trazabilidad curricular de sus voluntarios. Este proyecto se justifica en la necesidad urgente de modernizar dicha gestión mediante una plataforma digital centralizada que garantice la integridad y disponibilidad de los datos.

Desde una perspectiva técnica, la implementación de una **arquitectura de software del tipo Cliente-Servidor** permite optimizar los recursos administrativos y asegurar el acceso a la información en tiempo real. **Esta solución desacoplada** no solo resuelve las ineficiencias actuales, sino que establece una base tecnológica robusta preparada para evolucionar hacia microservicios, asegurando la sostenibilidad del sistema a largo plazo.

La solución aporta valor inmediato a la institución a través de los siguientes beneficios clave:

- **Eficiencia Operativa:** Centralización de cursos y certificados, eliminando la pérdida de documentación y errores manuales.

- **Seguridad y Control:** Acceso segmentado por roles (Administrador, Instructor, Alumno) y protección de datos sensibles mediante autenticación JWT.

- **Trazabilidad:** Seguimiento fidedigno y en tiempo real del historial formativo de cada voluntario.

- **Escalabilidad y Disponibilidad:** Acceso 24/7 desde cualquier dispositivo gracias a una infraestructura en la nube diseñada para soportar el crecimiento institucional.

1.7 Metodología

Para el desarrollo del proyecto se optó por la metodología ágil **Scrum**, debido a su flexibilidad y capacidad de adaptación a entornos dinámicos como el de la Academia del Cuerpo de Bomberos de Viña del Mar. Esta metodología permite una gestión eficiente del tiempo y de los recursos humanos, facilitando la entrega continua de incrementos funcionales del sistema, así como el ajuste ágil ante nuevos requerimientos o correcciones durante el ciclo de vida del desarrollo.

El equipo de trabajo se organiza en roles definidos para cubrir todas las aristas del proyecto:

- **Product Owner:** Representa los intereses del usuario final (Bomberos) y prioriza los requerimientos del negocio.
- **Scrum Master:** Facilita el proceso, elimina obstáculos y asegura el cumplimiento de la metodología.
- **Arquitecto de Software:** Define el diseño de alto nivel, la estructura de los componentes (Cliente-Servidor) y los patrones de comunicación.
- **Equipo de Desarrollo:** Compuesto por desarrolladores Backend y Frontend, encargados de la implementación técnica de la lógica de negocio y la interfaz de usuario respectivamente.

El trabajo se estructura en ciclos iterativos denominados *Sprints*, los cuales abarcan desde la planificación hasta la evaluación de funcionalidades. La hoja de ruta de implementación se definió de la siguiente manera:

- **Sprint 1 | Fundamentos y Seguridad:**
  - Configuración inicial de los proyectos (Django y Angular).
  - Diseño e implementación del modelo de datos relacional (PostgreSQL).
  - Desarrollo del sistema de autenticación vía Token (JWT), donde el Backend valida credenciales y otorga acceso seguro al Frontend.
- **Sprint 2 | Lógica de Negocio e Integración:**
  - Desarrollo de las API REST para los módulos de administración, instructores y alumnos.
  - Implementación de servicios en Angular para el consumo de datos y renderizado de vistas dinámicas.
  - Integración del módulo de Inteligencia Artificial (Chatbot).
- **Sprint 3 | Infraestructura y Despliegue en la Nube:**
  - Contenedorización de las aplicaciones (Docker).
  - Configuración de los servicios de Google Cloud Platform (Cloud Run, Cloud SQL y Cloud Storage).

- Elección y configuración de los servicios adecuados para el despliegue productivo y la gestión de la escalabilidad automática.

En cuanto a la gestión del código fuente y el trabajo colaborativo, se utiliza un repositorio en la nube (**GitHub**) bajo el flujo de trabajo estándar **Gitflow**, lo que permite mantener un control de versiones ordenado, separando las ramas de desarrollo, pruebas y producción.

1.8 Breve organización del informe

El presente informe se estructura en varios capítulos que abordan de manera progresiva y lógica el desarrollo del proyecto:

- **Capítulo 1 | Introducción:** Se presenta el contexto general del problema, la situación actual de la Academia del Cuerpo de Bomberos de Viña del Mar, la propuesta de solución tecnológica, la metodología de trabajo, los objetivos del proyecto y su justificación.
- **Capítulo 2 | Marco Teórico:** Se revisan los conceptos clave relacionados con la arquitectura de software, las tecnologías empleadas (Angular, Django, Google Cloud), fundamentos de API REST, mecanismos de autenticación (JWT) y otros estándares técnicos que sustentan el diseño de la plataforma.
- **Capítulo 3 | Diseño y Arquitectura de la Solución:** Se detallan los aspectos técnicos del sistema bajo el modelo de documentación de **"4+1 Vistas"**. Este capítulo abarca el diseño lógico de las aplicaciones, el modelado de datos, la especificación de la API y la **arquitectura de infraestructura y despliegue** en la nube (Cloud Run, Contenedores y Persistencia).
- **Capítulo 4 | Validación de Infraestructura y Rendimiento:** Se presentan los resultados de la validación empírica de la solución. Se analizan métricas de telemetría real (tráfico, latencia y costos) para demostrar el cumplimiento de los requisitos de escalabilidad, disponibilidad y eficiencia operativa.
- **Capítulo 5 | Conclusiones y Trabajo Futuro:** Se sintetiza el cumplimiento de los objetivos planteados, se discuten los alcances y limitaciones de la solución, y se proponen lineamientos para la continuidad y expansión futura del proyecto.
- **Capítulo 6 | Agradecimientos:** Se reconoce el apoyo de profesores, familia y compañeros que contribuyeron al desarrollo de esta memoria.
- **Capítulo 7 | Referencias:** Se listan las fuentes bibliográficas, estándares técnicos y documentación oficial utilizada para fundamentar el contenido del informe

2. Marco teórico

Este capítulo presenta los conceptos teóricos y técnicos fundamentales que sustentan el desarrollo de la plataforma académica propuesta para la Academia del Cuerpo de Bomberos de Viña del Mar. Se abordan temas clave como los estilos de arquitectura de software, el enfoque modular adoptado en el proyecto, el uso de API para la integración

de servicios, y los mecanismos de Autenticación para garantizar la seguridad del sistema. Estos elementos permiten comprender las decisiones de diseño tomadas y cómo estas contribuyen a la eficiencia, escalabilidad y mantenibilidad de la solución tecnológica desarrollada.

2.1 Fundamentos de la arquitectura de software

La arquitectura de software constituye la estructura fundamental de un sistema, definida por sus componentes, las relaciones entre ellos y los principios que guían su diseño y evolución. Esta abarca desde la planificación hasta la integración. Su objetivo principal es garantizar organización, escalabilidad, mantenibilidad, seguridad y rendimiento, que, en este contexto, supondría establecer principios como lo son la separación de responsabilidades de trabajo, modularidad de los componentes, el bajo acoplamiento y la alta cohesión a la plataforma académica [12].
Existen una variedad estilos arquitectónicos entre los que se encuentran la arquitectura de capas, cliente-servidor, MVC, microservicios, monolíticas (tradicionales), hexagonal y arquitecturas orientadas a eventos, según las necesidades de tiempo y recursos del proyecto. En este sentido, la arquitectura actúa como guía para el desarrollo, asegura la calidad global del sistema y permite que los equipos de trabajo mantengan una visión común y coherente durante todo el ciclo de vida del software. A continuación, se muestran **cuatro** opciones que se analizaron [12]:

Tabla 1 Comparativa de Patrones arquitectónicos
Fuente: Elaboración Propia

Arquitectura	Descripción	Beneficios	Desventajas
Monolítica tradicional	Toda la aplicación se desarrolla como una única unidad con componentes interdependientes.	- Fácil de desarrollar y desplegar - Menor complejidad inicial - Ideal para MVPs simples	- Difícil de escalar - Bajo aislamiento de errores - Cambios afectan todo el sistema
Monolítica modular	Variante del monolito que organiza el sistema en módulos independientes con interfaces definidas.	- Mejor organización interna - Facilita mantenimiento - Posible transición a microservicios	- Aún existe dependencia entre módulos - Escalabilidad limitada comparada con microservicios
Microservicios	La aplicación se divide en servicios independientes que se comunican entre sí por API.	- Alta escalabilidad - Despliegue independiente por servicio - Mejor tolerancia a fallos	-Mayor complejidad técnica -Requiere infraestructura robusta y mayor personal de desarrollo - Costos de operación más altos

Arquitectura Desacoplada (Cliente-Servidor)	El **Frontend** (cliente, ej. Angular) y el **Backend** (servidor, ej. Django) son aplicaciones **independientes**. Se comunican a través de una red, usualmente mediante una **API** (ej. REST). El Backend expone datos y el Frontend los consume y presenta.	- Flexibilidad tecnológica - Escalabilidad independiente - Desarrollo en paralelo - Reutilización	- Mayor complejidad inicial - Dependencia de la red - Gestión de la API - Seguridad

En el caso del proyecto, se optó por una **arquitectura del tipo cliente-servidor** a nivel general, debido a su flexibilidad y a la facilidad que ofrece para integrar nuevas funcionalidades sin afectar a otros componentes, lo cual resulta adecuado considerando los recursos y el tiempo disponible para desarrollar el MVP (Producto Mínimo Viable). Esta arquitectura, al separar las dos aplicaciones —llamadas en adelante "servidor" para la aplicación Backend (Django Rest) y "cliente" para la aplicación Frontend (Angular)—, contribuye a poder trabajarlas y desarrollarlas de forma independiente, sin afectar o ralentizar los avances de cada una [12].
Cabe destacar que la arquitectura final implementada corresponde estrictamente al modelo **cliente-servidor desacoplado (Angular + Django)**. La mención a arquitecturas monolíticas modulares presentada anteriormente en la Tabla 1 forma parte únicamente del análisis comparativo realizado en la etapa de diseño para descartar alternativas menos eficientes para los requerimientos de escalabilidad del proyecto.

2.2 Tecnologías empleadas

Con respecto a las tecnologías implementadas al proyecto se seleccionaron tecnologías modernas y robustas, garantizando eficiencia, seguridad y escalabilidad.

Lenguaje de programación

- **Python 3:** Se eligió lenguaje principal para el desarrollo del Backend debido a su simplicidad, legibilidad y amplia comunidad, lo que facilita el desarrollo ágil y colaborativo. Esta elección está directamente vinculada al uso del framework **Django**, que permite construir aplicaciones web de forma rápida y estructurada. Django, junto con Django REST Framework, ofrece herramientas integradas para autenticación, gestión de datos y creación de API, lo que lo convierte en una

solución ideal para implementar una arquitectura modular escalable, segura y alineada con los objetivos del proyecto.

Framework Backend

- **Django REST Framework (DRF):** Framework integrado a Django, el cual es flexible en integración y está orientado en la construcción de API webs (Interfaces de Programación de Aplicaciones) **[7]**.

Para evaluar la elección de Django como framework principal, se realizó una comparación con dos alternativas ampliamente utilizadas: Spring Boot (Java) y .NET (C#). La siguiente tabla resume las principales características de cada uno:

Tabla 2 Comparación de frameworks Backend
Fuente: Elaboración Propia

	Django (Python)	Spring Boot (Java)	.NET (C#)
Lenguaje base	Python	Java	C#
Arquitectura recomendada	Cliente-servidor	Microservicios, REST, MVC	Monolítica, Microservicios, MVC
Curva de aprendizaje	Baja	Media-Alta	Media
Velocidad de desarrollo	Alta	Media	Alta
Rendimiento	Bueno para proyectos medianos	Muy alto	Muy alto
Escalabilidad	Moderada	Excelente	Excelente
Seguridad	Buena (JWT, OAuth)	Muy buena (Spring Security)	Muy buena (Identity, JWT)
Soporte para API REST	Excelente (DRF)	Excelente (Spring REST)	Excelente (ASP.NET Web API)
Ecosistema y comunidad	Amplia comunidad	Muy grande	Muy grande
Despliegue en la nube	Fácil (Azure, AWS, GCP)	Requiere configuración	Muy integrado con Azure

Testing y mantenimiento	Sencillo	Robusto	Robusto
Licencia	Open Source (BSD)	Open Source (Apache 2.0)	Open Source (.NET Core)

La elección de Django como framework Backend se fundamenta en su capacidad para desarrollar rápidamente un Producto Mínimo Viable (MVP), su estructura modular, y su integración nativa con herramientas de autenticación, bases de datos y API REST [6]. Además, su curva de aprendizaje baja y su comunidad activa lo convierten en una opción ideal para proyectos académicos con recursos limitados y necesidad de resultados funcionales en corto plazo.

No obstante, se reconoce que frameworks como Spring Boot y .NET ofrecen ventajas significativas en términos de rendimiento, escalabilidad y soporte empresarial, por lo que podrían considerarse en futuras fases del proyecto, especialmente si se *migra* hacia una arquitectura de microservicios o se requiere integración con plataformas corporativas.

Framework Frontend

- **Angular**: Framework de código abierto, creado por la empresa tecnológica Google y desarrollado en TypeScript, que se utiliza para crear y mantener aplicaciones web de una sola página [8].

Base de datos

- **PostgreSQL:** motor relacional de código abierto utilizado para almacenar datos, en los cuales para el proyecto se utilizó para los distintos modelos. Destaca por su robustez, soporte de integridad referencial, funciones avanzadas y capacidad de escalar con grandes volúmenes de datos.

- **SQLite:** motor de base de datos relacional ligero e integrado, el cual en el proyecto se empleó específicamente para la etapa de desarrollo local. Destaca por su arquitectura *serverless* (sin servidor), su portabilidad al almacenar todo el sistema en un único archivo y su eficiencia operativa sin necesidad de configuración compleja.

  Para el entorno de desarrollo local se utiliza SQLite como motor ligero de almacenamiento, mientras que en el entorno de producción se emplea el servicio gestionado de Google Cloud denominado Cloud SQL. La elección de estos sistemas de gestión de bases de datos relacionales (RDBMS) se debe a su compatibilidad nativa con el Framework Backend seleccionado (Django). Esta decisión estratégica evita la complejidad técnica y la sobrecarga de configuración que implicaría la integración de soluciones NoSQL (como MongoDB), las cuales no son requeridas dado que la naturaleza estructurada de los datos del proyecto se ajusta idóneamente al modelo relacional [3].

Almacenamiento

- **Cloud Storage**: servicio administrado para almacenar datos no estructurados proporcionado por Google Cloud **[4]**.

Plataforma de ejecución

- **Cloud Run**: plataforma de procesamiento administrada de Google Cloud, la cual ejecuta contenedores basado en imágenes y posee la funcionalidad de escalabilidad horizontal, como también configuración vertical de los propios contenedores, en otras palabras, poder aumentar los recursos necesarios para otorgar una mayor estabilidad en la aplicación levantada **[1]**.

  Su uso está relacionado a la ejecución de los dos tipos de frameworks (Django REST y Angular) utilizado conectados entre los dos mediante las API y al uso de redes internas sin tener que acceder a conexiones públicas.

  Este servicio es esencial, debido a su integración con otros servicios propios de Google Cloud como lo son las anteriormente mencionadas Cloud SQL y Cloud Storage, esenciales para el funcionamiento de la aplicación de Django.

Autenticación y control de acceso

- **JWT (JSON Web Tokens):** estándar abierto (RFC 7519) el cual se utiliza para transmitir de forma segura información entre dos partes, como un cliente y un servidor, mediante un objeto JSON firmado criptográficamente. Este es utilizado por el sistema de autenticación basado en tokens de este tipo para garantizar protección del acceso a la plataforma como el acceso a recursos específicos por parte de los usuarios. Esto asegura sesiones sin estado (*stateless*), es decir, no almacenadas por el servidor, y la segregación por roles (administrador, instructor, bombero) **[10]**.

2.3 Uso de API en la arquitectura de la plataforma

Las **API REST (Representational State Transfer)** son un componente esencial en el diseño de sistemas modernos, ya que permiten la comunicación entre diferentes capas de una aplicación, como el Frontend y el Backend, de forma estructurada, segura y escalable **[7]**.

En el contexto de esta plataforma académica, las API cumplen funciones clave:

- **Interconexión entre componentes**: Facilitan el intercambio de datos entre el cliente (interfaz web) y el servidor (lógica de negocio), permitiendo que cada usuario acceda a la información que le corresponde según su rol (administrador, instructor, alumno).
- **Modularidad y escalabilidad**: Las API permiten que cada módulo del sistema se comunique de forma independiente, lo que favorece la evolución hacia una arquitectura de microservicios en etapas futuras.

- **Seguridad y control de acceso**: Integradas con mecanismos de autenticación como JWT, las API garantizan que solo usuarios autorizados puedan acceder a recursos específicos.
- **Facilidad de mantenimiento y expansión**: Al estar bien definidas, las API permiten agregar nuevas funcionalidades sin afectar el resto del sistema, lo que mejora la mantenibilidad del software.

Para su implementación, se utiliza el framework **Django REST Framework (DRF)**, que proporciona herramientas robustas para la creación de API en Python, incluyendo serialización de datos, control de permisos, autenticación y documentación automática.

Esta estructura basada en API REST permite que la plataforma sea accesible desde distintos dispositivos, mantenga sesiones sin estado (*stateless*) y ofrezca una experiencia de usuario fluida y segura.

2.4 Mecanismos de autenticación y control de acceso

La seguridad y el control de acceso son pilares fundamentales en el diseño de plataformas académicas, especialmente cuando se manejan datos sensibles como historiales de capacitación, certificados y documentación personal. Para este proyecto, se implementa un sistema de autenticación basado en **JWT (JSON Web Tokens)**, el cual permite validar la identidad de los usuarios y controlar el acceso a los recursos del sistema de forma segura y eficiente.

Autenticación con JWT

JWT, como se explicó en el apartado de tecnologías, es un estándar abierto (RFC 7519) que permite transmitir información entre dos partes (cliente y servidor) mediante un objeto JSON firmado criptográficamente, a lo que consideramos token. Este token contiene información sobre el usuario (claims), como su rol, ID y permisos, y se utiliza para autorizar el acceso a funcionalidades específicas dentro de la plataforma **[10]**.

Para la seguridad y la gestión de sesiones, se implementa el uso de dos tipos de tokens:

- **Access Token**: Tiene una vida útil corta y se utiliza para acceder a recursos protegidos. Es enviado con cada solicitud al servidor para validar la identidad del usuario.
- **Refresh Token**: Tiene una vida útil más larga y se utiliza para obtener nuevos access tokens sin necesidad de que el usuario vuelva a iniciar sesión. Este token se almacena de forma segura y permite mantener sesiones activas sin comprometer la seguridad.

Este enfoque permite mantener sesiones **sin estado (*stateless*)**, mejorar la experiencia del usuario y reducir el riesgo de exposición prolongada en caso de que un token sea comprometido.

Segmentación por roles

El sistema de autenticación también permite definir accesos diferenciados para los distintos tipos de usuarios: **administradores**, **instructores** y **alumnos**. Cada rol tiene

permisos específicos y acceso a funcionalidades particulares, lo que garantiza un uso controlado y seguro de la plataforma.

Integración con la Arquitectura Orientada a Servicios (API)

La elección de estos mecanismos de seguridad no es aislada, sino que responde directamente a la arquitectura desacoplada del sistema. Dado que la comunicación entre el cliente (Angular) y el servidor (Django) se realizará exclusivamente a través de una **API REST**, el protocolo de seguridad debe ser **sin estado** (*stateless*).

A diferencia de las aplicaciones web tradicionales que dependen de sesiones almacenadas en la memoria del servidor, la API diseñada para este proyecto requerirá que cada petición HTTP (ya sea para solicitar datos de un curso o registrar un bombero) porte su propia credencial de autorización (el *token*). De esta forma, la API actúa como una barrera de seguridad que valida cada interacción de forma independiente, garantizando que los principios de autenticación y control de roles definidos teóricamente se apliquen de manera técnica en cada uno de los *endpoints* o puntos de acceso del sistema. Esto es fundamental el uso del servicio **Cloud Run** o **microservicios (contenedores)** que pueden generar o deshacer instancias de contenedores de forma dinámica y automática según la demanda. En este entorno efímero, un mecanismo de sesión tradicional (almacenado en la memoria del servidor) resultaría en desconexiones constantes para el usuario cada vez que la plataforma reasigne la petición a una nueva instancia. Por el contrario, el enfoque *stateless* adoptado asegura que la validación de credenciales sea agnóstica a la infraestructura subyacente, permitiendo una escalabilidad horizontal transparente y sin interrupciones para el cliente final **[1,10]**.

3. Diseños de la Plataforma

3.1 Arquitectura de Software (Modelo 4+1 Vistas)

Para garantizar una comprensión integral de la plataforma Académica, se ha adoptado el modelo de **"4+1 Vistas"** propuesto por Philippe Kruchten. Este estándar de la industria permite descomponer la complejidad del desarrollo en cinco perspectivas simultáneas, asegurando que tanto los interesados en el negocio como los diversos profesionales relacionados con el proyecto comprendan la solución desde sus respectivos roles **[11]**.

A continuación, se detallan las vistas que componen la arquitectura del sistema, las cuales sirven de marco para el diseño detallado de datos y API presentados en las secciones subsiguientes.

Vista Lógica (Logical View)

Esta vista se centra en la organización funcional del código y los componentes que entregan valor al usuario final. Su objetivo es mostrar cómo el sistema se estructura internamente para cumplir con los requisitos funcionales definidos.

Debido a la naturaleza desacoplada del sistema por el patrón arquitectónico Cliente-servidor, la vista lógica se divide en dos grandes bloques que interactúan vía HTTP (REST):

1. **Lógica del Cliente (Frontend):** Construida sobre el framework **Angular**. La interfaz se organiza mediante una arquitectura de componentes reutilizables y servicios inyectables que gestionan la comunicación asíncrona con el servidor **[8]**.
   - Arquitectura Interna del Cliente (Angular): Patrón de Servicios
     1. Para garantizar la mantenibilidad y la escalabilidad del código en el Frontend, se implementó el Patrón de Servicios. Esta decisión de diseño permite desacoplar la lógica de presentación (Componentes) de la lógica de negocio y comunicación de datos.
     2. Componentes (Components): Son responsables únicamente de renderizar la vista y gestionar la interacción directa con el usuario. No contienen lógica compleja ni realizan llamadas HTTP directas.
     3. Servicios (Services): Actúan como una capa de abstracción para la comunicación con la API. Se implementó un servicio específico por cada entidad del dominio (ej. AuthService, CursoService, BomberoService). Estos servicios utilizan el cliente HTTP de Angular (HttpClient) para enviar peticiones asíncronas y manejan la transformación de respuestas utilizando Observables (RxJS).
     4. Inyección de Dependencias: Los servicios son inyectados en los componentes que los requieren, promoviendo la reutilización de código y facilitando las pruebas unitarias.

2. **Lógica del Servidor (Backend):** Implementada con **Django [6]**. Se utiliza una organización basada en "Aplicaciones" (Apps) modulares que encapsulan la lógica de negocio específica **[6]**.
   - Las aplicaciones principales identificadas son: users (gestión de bomberos), courses (gestión académica) y organizations (estructura institucional).
   - **Relación con Datos:** La estructura de clases y entidades que sustentan esta vista se basan en lo propuesto en la sección **3.2 Diseño de modelos de datos**.
   - Arquitectura Interna del Servidor (Django): Serializadores y Vistas
     1. En el Backend, la arquitectura se apoya en las capacidades de Django REST Framework (DRF) para estructurar la respuesta de la API de manera estandarizada. La lógica se distribuye en las siguientes capas:
     2. Modelos (Models): Definen la estructura de los datos y las reglas de integridad a nivel de base de datos (ORM).

3. Serializadores (Serializers): Son los encargados de la transformación de datos. Convierten los objetos complejos de Django (QuerySets) a tipos de datos nativos de Python (JSON) para el envío, y validan los datos entrantes antes de guardarlos. Esto asegura que solo información válida llegue a la base de datos.

4. Vistas (ViewSets): Orquestan el flujo de la petición. Reciben la solicitud HTTP, determinan qué acción realizar (listar, crear, actualizar), invocan al serializador correspondiente y devuelven la respuesta estándar.

5. Capa de Servicios (Service Layer): Para lógica de negocio compleja que no pertenece estrictamente a la vista ni al modelo (como la orquestación de la comunicación con la IA de Gemini mediante Vertex AI o almacenamiento en servicio Cloud), se crearon módulos de servicios dedicados. Esto evita la saturación de los controladores.

A continuación, la **Figura 3.1** ilustra la arquitectura lógica global del sistema, detallando cómo los módulos/App internos del servidor (como las aplicaciones de organización y formación) interactúan con la arquitectura de servicios del cliente Angular y las integraciones externas (Gemini AI y Cloud Storage):

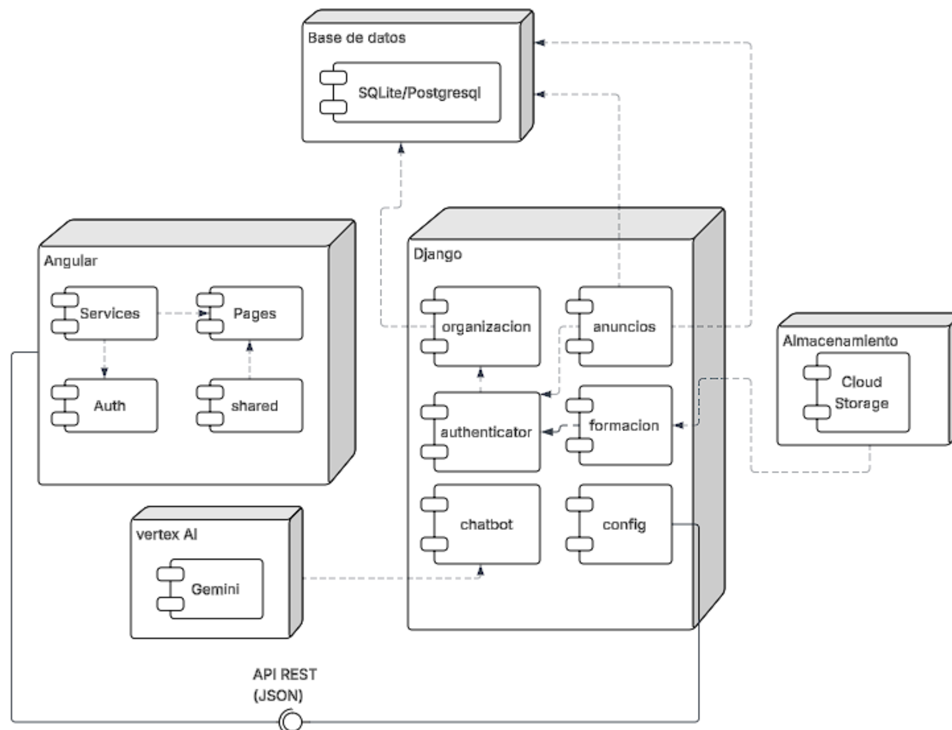

Figura 3.1 Diagrama de componentes del sistema
Fuente: elaboración propia

Vista de Procesos (Process View)

Esta vista describe el comportamiento dinámico del sistema, enfocándose en la concurrencia, el flujo de los datos y cómo se comunican los procesos en tiempo de ejecución. Es crítica para entender cómo el sistema maneja múltiples usuarios simultáneos y procesos de fondo.

Flujos Principales del Sistema:

- **Gestión de Concurrencia API:** El servidor utiliza **uWSGI** como servidor de aplicaciones. Este actúa como una interfaz entre el tráfico web y el código Python (Django Rest), gestionando múltiples procesos (*workers*) e hilos (*threads*) para atender las solicitudes concurrentes de los 600 usuarios esperados sin bloquear el hilo principal de ejecución **[9]**.
- **Integración Asíncrona con IA:** Existe un proceso de orquestación donde el Backend actúa como intermediario seguro: recibe la consulta del Frontend, construye el contexto del sistema y se comunica con la API de **Gemini**, devolviendo la respuesta procesada al usuario, tal como se detalla en la sección de diseño de API **[5]**.

A continuación, se muestra un diagrama de secuencia, en el cual se ejemplifica uno de los flujos de acción que pueden ocurrir en la plataforma:

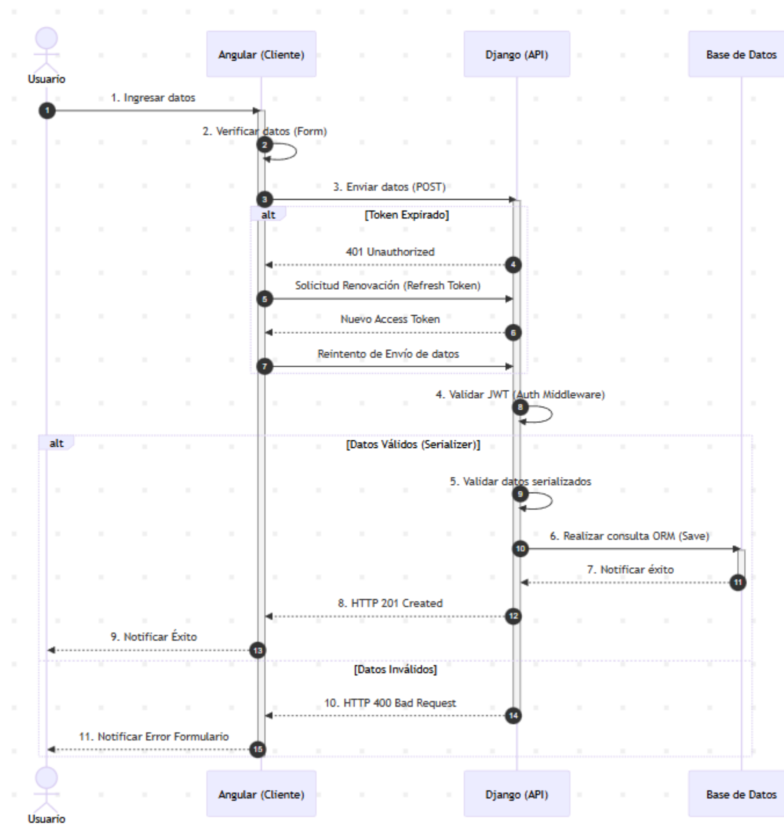

Figura 3.2 Diagrama de Secuencia - Creación de Inscripción de Curso
Fuente: elaboración propia

Análisis del Flujo de Inscripción: El diagrama detalla la interacción síncrona entre las capas del sistema durante el proceso crítico de inscripción. El flujo inicia en la capa de presentación (**Angular**), donde se realizan validaciones de formato en el formulario (paso 2) antes de consumir recursos del servidor, optimizando el tráfico de red.
Al recibir la petición, el servidor (**Django**) ejecuta una cadena de responsabilidades:

1. **Seguridad:** Se intercepta la solicitud para validar la vigencia del *JSON Web Token* (JWT). En caso de expiración, se muestra el sub-proceso de renovación del token de acceso (access token).
2. **Integridad de Datos:** El *Serializer* de Django verifica que los datos cumplan con las reglas del negocio (paso 8).
3. **Persistencia:** Mediante el ORM, se transfiere la información a la Base de Datos Relacional.

Como parte final del flujo, el sistema retorna códigos de estado HTTP estándar (**200 OK** para éxito o **400 Bad Request** para errores de validación), permitiendo que el cliente notifique el resultado al usuario final.

Vista de Desarrollo (Development View)

Esta vista se enfoca en la organización estática del software en el entorno de programación. Describe la estructura de carpetas, la gestión de paquetes y las herramientas utilizadas por el equipo de desarrollo para construir el sistema **[11]**.

- **Estructura del Proyecto**:
  - Backend (Django REST Framework):
    - La gestión de dependencias se realiza mediante pyproject.toml (estándar moderno de Python), asegurando la reproducibilidad de los entornos.
    - La estructura sigue el patrón modular de Django ("Apps"), separando claramente la lógica de negocio (cursos, authenticator) de la configuración (config) y las integraciones externas (gemini_integration).
  - Frontend (Angular):
    - Organizado bajo una arquitectura modular por características (Feature-based).
    - Se distinguen directorios clave: pages (vistas agrupadas por roles como admin/alumno), auth (seguridad y guards), @core (servicios singleton y utilidades globales) y services (comunicación HTTP).
  - Entorno y Contenedores:
    - Se utiliza Docker para orquestar los servicios en desarrollo, garantizando la paridad entre el entorno local y producción.
  - Control de Versiones:

  - Se utiliza Git para la gestión del código fuente, permitiendo el trabajo colaborativo y el control de cambios sobre los archivos de ambos entornos.

Vista Física o de Despliegue (Physical View)

Esta vista describe el mapeo del software sobre el hardware y la infraestructura de nube donde se ejecuta el sistema.

Implementación en la Nube: La arquitectura física adopta un enfoque *Cloud Native* utilizando contenedores. El sistema no reside en un servidor físico tradicional, sino que se despliega en:

- **Nodos de Cómputo:** Servicios de **Google Cloud Run** que ejecutan los contenedores Docker del Cliente y del Servidor de forma independiente **[1]**.
- **Persistencia:** Una instancia gestionada de **Cloud SQL** para la base de datos relacional **[3]**.
- **Almacenamiento:** Buckets de **Cloud Storage** para archivos estáticos **[4]**.

Nota: Los detalles técnicos, configuraciones de auto-escalado y diagramas de red de esta vista se abordan en profundidad en el ***subcapítulo 3.4 Arquitectura de Infraestructura y Despliegue****.*

Vista de Escenarios (+1)

La vista de escenarios actúa como la conexión que une los requerimientos funcionales con la arquitectura definida. Su objetivo es validar que el diseño (componentes Angular, API Django y Base de Datos) es capaz de soportar los flujos de trabajo reales de la institución.

Basado en la planificación de los ciclos de desarrollo (Sprints 1, 2 y 3), se han identificado los actores críticos y se han agrupado las historias de usuario en tres escenarios arquitectónicos.

Identificación de Actores

A partir de las historias de usuario, se consolidan tres roles principales que interactúan con el sistema:

1. **Bombero (Usuario Final):** Es el consumidor principal de la información. Su interacción valida la usabilidad, el acceso seguro y la disponibilidad de datos históricos (certificados y entrenamientos).
2. **Instructor:** Responsable de la gestión operativa de la docencia. Sus casos de uso validan la capacidad del sistema para manejar archivos (material didáctico), comunicaciones masivas y evaluaciones.
3. **Administrador / Rector Académico:** Posee el control estructural. Sus interacciones ponen a prueba la integridad referencial de la base de datos (prerrequisitos, mallas curriculares) y las integraciones avanzadas (IA).

Catálogo de Escenarios Funcionales

A continuación, se describen los casos de uso que la arquitectura debe satisfacer, clasificados por su impacto en el sistema:

A. Escenario de Gestión y Estructura Académica (Core del Negocio)

Este grupo de funcionalidades valida el modelo de datos relacional y la lógica de negocio en el Backend (Django).

- **Gestión Curricular:** El administrador debe poder crear una estructura jerárquica compleja (Niveles, Especialidades, Cursos y Módulos) y definir **prerrequisitos** para controlar la progresión académica.
- **Inscripción y Control:** Capacidad de inscribir bomberos a módulos (masiva o individualmente), validando automáticamente si cumplen con los requisitos previos y su estado en el cuerpo de bomberos.
- **Gestión de Usuarios:** Registro de nuevos bomberos, asignación de compañías y gestión de roles (RBAC) para diferenciar permisos entre alumnos e instructores.

B. Escenario de Operativa Docente y Comunicación

Valida la interacción en tiempo real y el almacenamiento de objetos (Cloud Storage).

- **Ciclo de Enseñanza:** El instructor debe poder subir **material didáctico** (archivos) por módulo y realizar la **evaluación** de los alumnos, registrando notas y asistencia.
- **Comunicaciones:** El sistema debe permitir el envío de **notificaciones** segmentadas por curso y la publicación de **anuncios globales** en el dashboard, asegurando que la información fluya desde la comandancia hacia los voluntarios.

C. Escenario de Experiencia del Usuario y Autogestión

Valida la seguridad (Auth) y la eficiencia de las consultas de lectura (Frontend Angular).

- **Acceso y Seguridad:** Autenticación segura (Login), gestión de sesiones y recuperación de contraseñas.
- **Hoja de Vida Académica:** El bombero debe poder visualizar su historial de entrenamientos, ver su progreso en el *dashboard* y, críticamente, **descargar certificados** en formato PDF de los cursos aprobados.
- **Asistencia Inteligente:** Uso de un **Chatbot con IA** (Gemini) para que los administradores realicen consultas complejas sobre alumnos o instructores usando lenguaje natural.

Para estos casos, a continuación, se presenta la figura 3.3, la cual muestra la visión general de las interacciones, destacando cómo el actor Administrador orquesta la estructura base, sobre la cual el Instructor gestiona la ejecución académica y el Alumno

consume los recursos formativos.

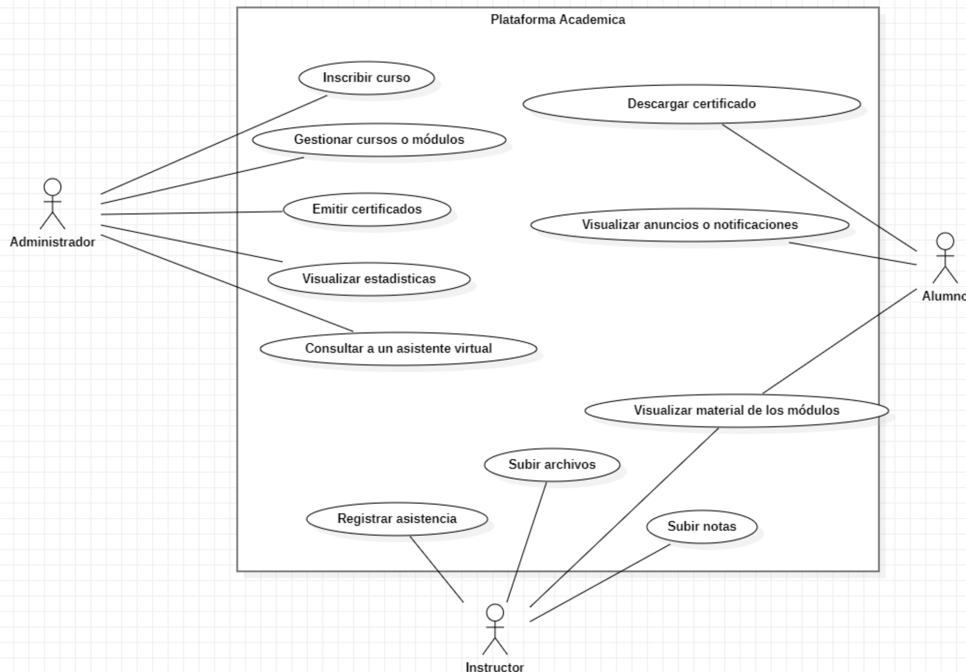

Figura 3.3 Diagrama de Casos de Uso – Visión general
Fuente: elaboración propia

3.2 Diseño de modelos de datos

La base operativa sobre la cual funciona la aplicación reside en la persistencia estructurada de la información. Para garantizar la integridad, escalabilidad y coherencia de los datos del sistema, se diseñó un **Modelo Relacional**, implementado sobre los motores de base de datos **PostgreSQL** (gestionado en producción mediante el servicio Cloud SQL) y SQLite (gestionado en local).

La definición de este esquema de datos se realizó siguiendo la metodología *Code-First* (Código Primero) proporcionada por el **ORM (Object-Relational Mapping)** de Django. Mediante este proceso, las estructuras de datos se definen inicialmente como clases de Python (Modelos) dentro de cada "Aplicación" del Backend, las cuales son traducidas y mapeadas automáticamente a tablas relacionales mediante un sistema de migraciones.

Los modelos de datos resultantes se agrupan lógicamente en cuatro módulos principales que componen el servidor:

- **Autenticación:** Tablas encargadas de la seguridad, gestión de perfiles de bomberos, roles y credenciales.

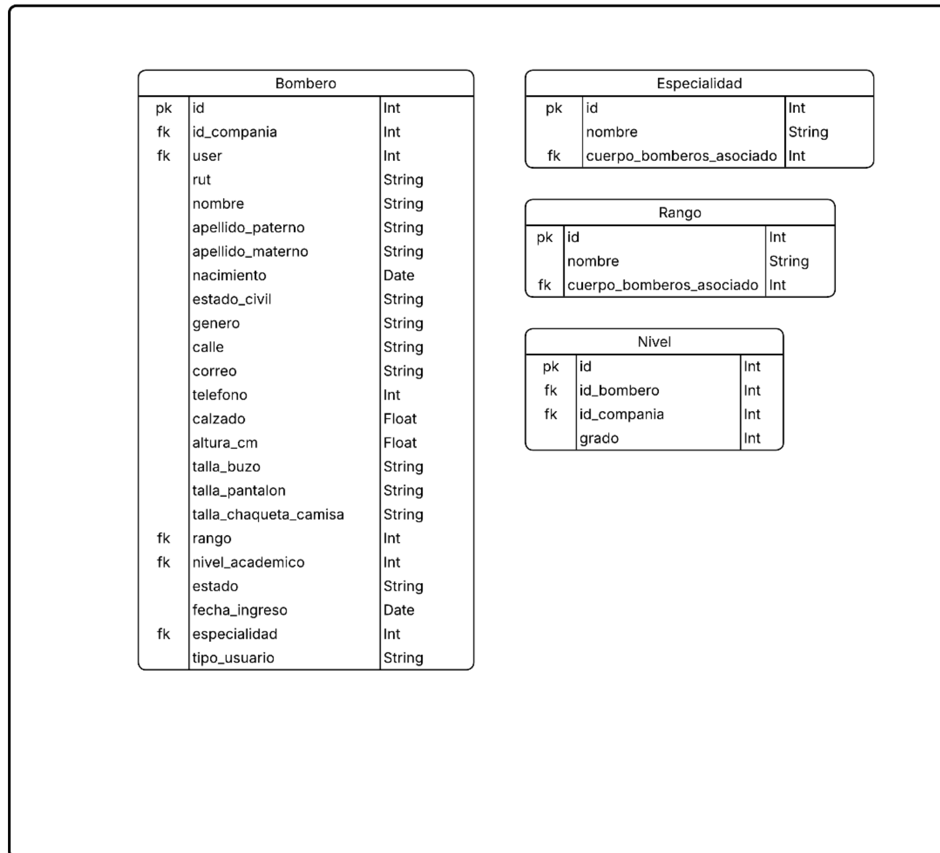

Figura 3.4 Tablas correspondientes a Autenticación
Fuente: Elaboración propia

Relaciones:

- Bombero (user) -> User[1]
- Bombero (id_compania) -> Compania
- Bombero (rango) -> Rango
- Bombero (nivel_academico) -> Nivel
- Bombero (especialidad) -> Especialidad
- Especialidad (cuerpo_bomberos_asociado) -> CuerpoBomberos
- Nivel (cuerpo_bomberos_asociado) -> CuerpoBomberos
- Rango (cuerpo_bomberos_asociado) -> CuerpoBomberos

[1] **User**: tabla proporcionada por Django, en la cual se almacenan información de la cuenta como correo, contraseña y entre otros.

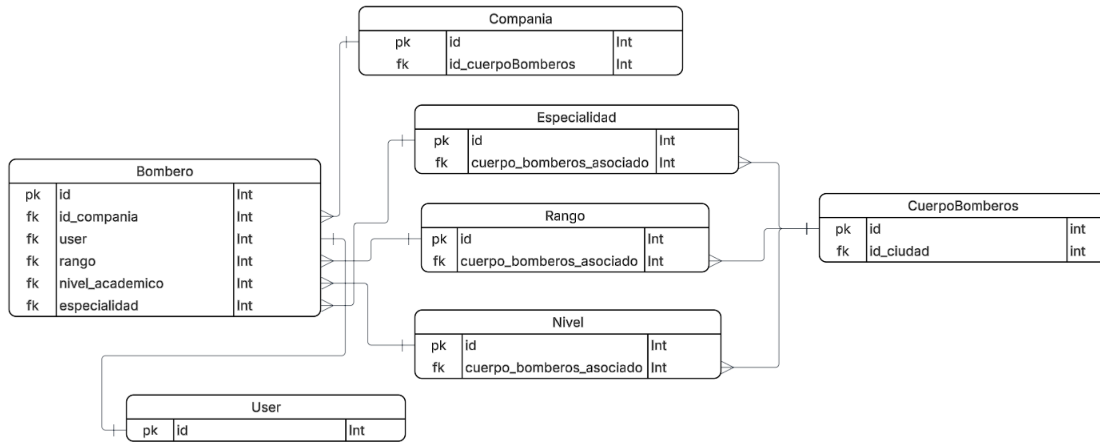

Figura 3.5 Relaciones de las tablas de Autenticación
Fuente: Elaboración propia

- **Formación:** Entidades que modelan la malla curricular, incluyendo cursos, módulos, inscripciones, certificados y archivos.

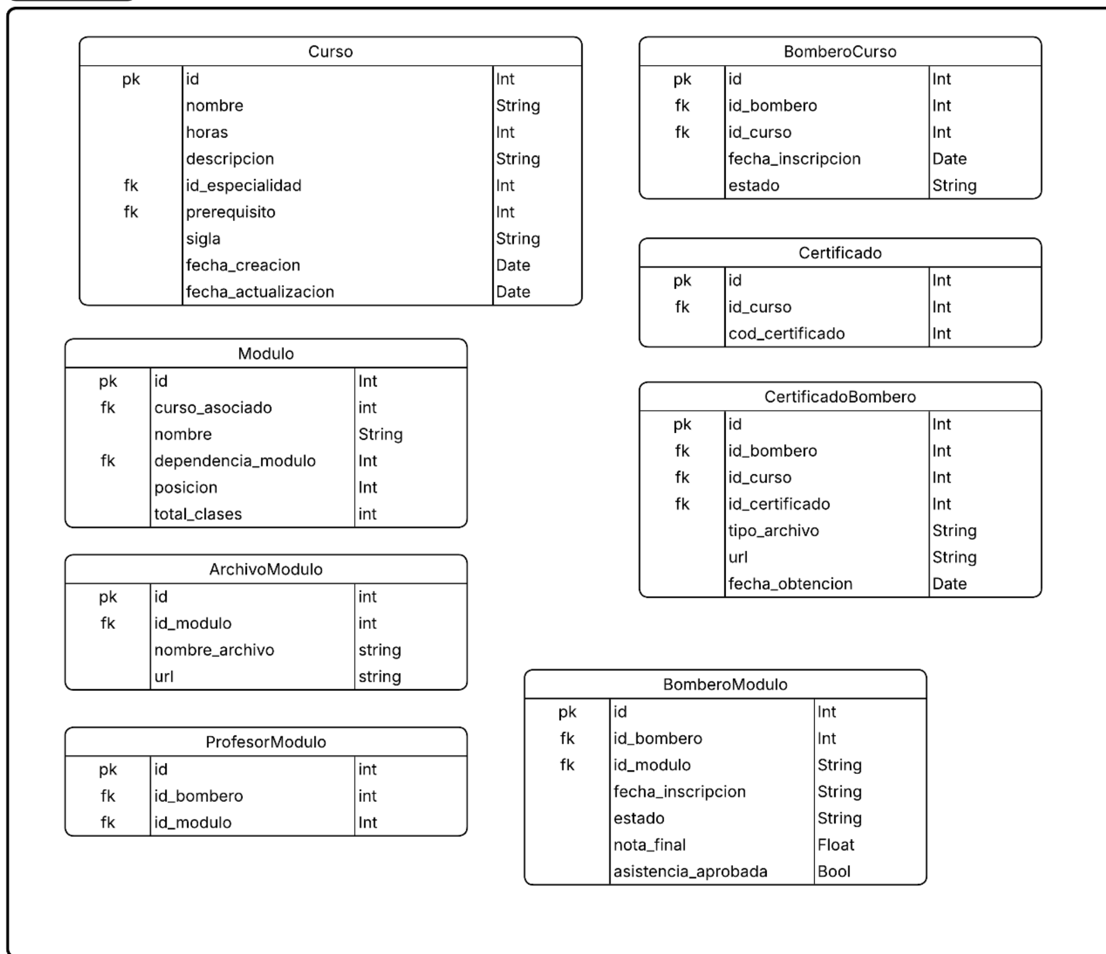

Figura 3.6 Tablas correspondientes a Formación
Fuente: Elaboración propia

Relaciones:

- Curso (nivel) -> Nivel
- Curso (prerequisito) -> Curso
- Curso (tipo_especializado) -> Especialidad
- Certificado (id_curso) -> Curso
- CertificadosBombero (id_certificado) -> Certificado
- CertificadosBombero (id_bombero) -> Bombero
- Modulo (curso_asociado) -> Curso
- Modulo (dependencia_modulo) -> Modulo
- ProfesorModulo (id_bombero) -> Bombero
- ProfesorModulo (id_modulo) -> Modulo
- BomberoCurso (id_bombero) -> Bombero
- BomberoCurso (id_curso) -> Curso
- BomberoModulo (id_bombero) -> Bombero
- BomberoModulo (id_modulo) -> Modulo
- ArchivoModulo (modulo) -> Modulo

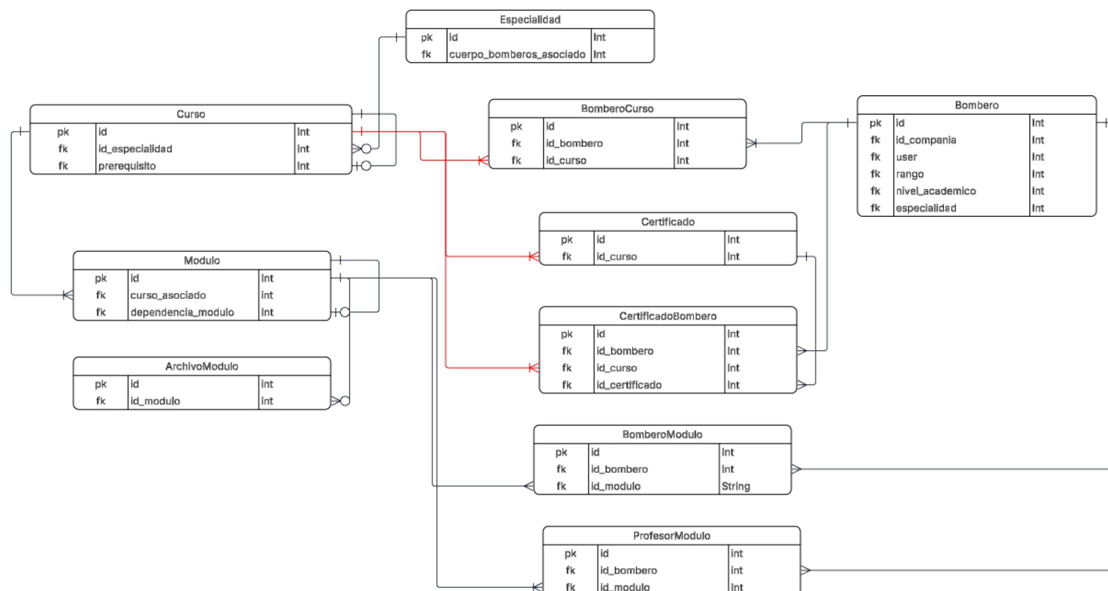

Figura 3.7 Relaciones de las tablas de Formación
Fuente: Elaboración propia

- **Organización:** Tablas que representan la distribución geográfica y jerárquica organizacional (Cuerpos de Bomberos, Compañías, Regiones).

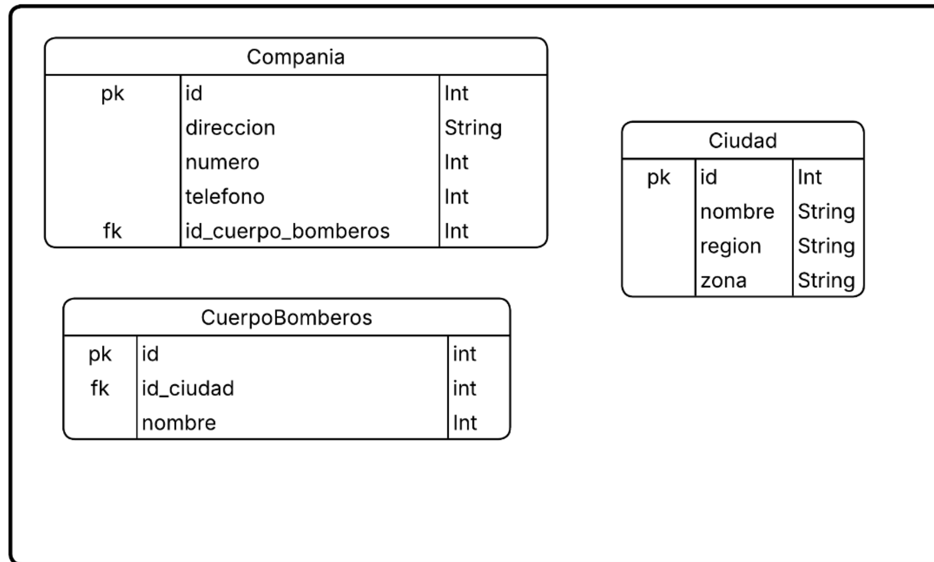

Figura 3.8 Tablas correspondientes a Organización
Fuente: Elaboración propia

Relaciones:

- CuerpoBomberos (id_ciudad) -> Ciudades
- Compania (id_cuerpo_bomberos) -> CuerpoBomberos

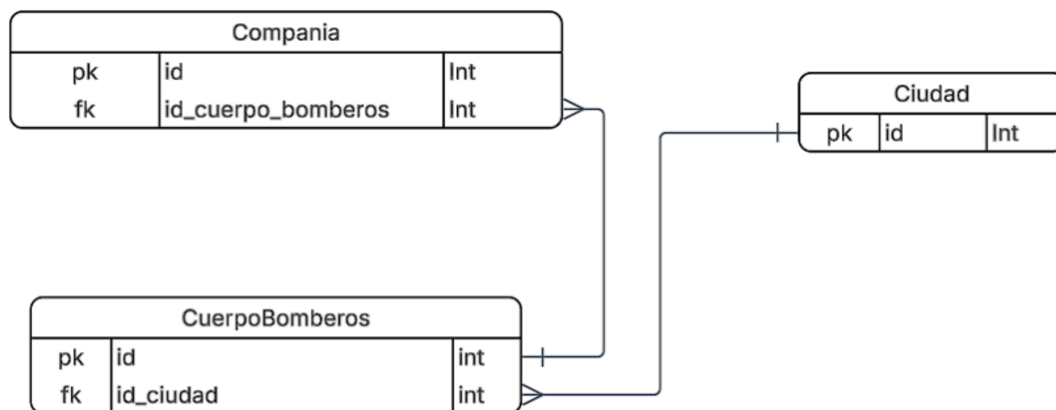

Figura 3.9 Relaciones de las tablas de Organización
Fuente: Elaboración propia

- **Avisos**: Estructuras para la gestión de notificaciones y calendario de eventos.

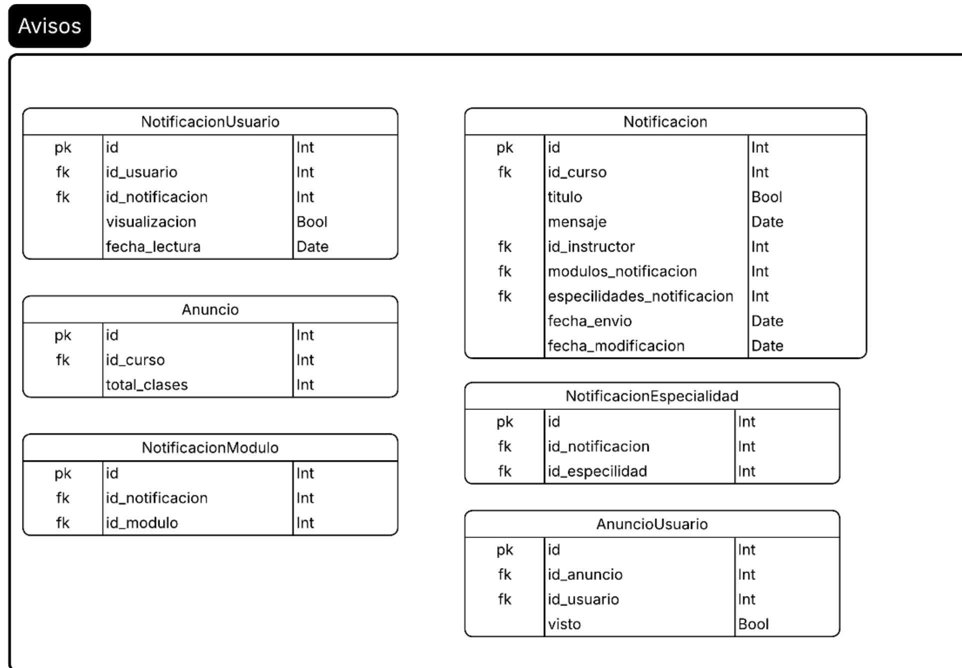

Figura 3.10 Tablas correspondientes a Avisos
Fuente: Elaboración propia

Relaciones:

- Notificacion (id_instructor) -> Bombero
- NotificacionUsuario (id_usuario) -> Bombero
- NotificacionUsuario (id_notificacion) -> Notificacion
- AnuncioUsuario (id_usuario) -> Bombero
- AnuncioUsuario (id_anuncio) -> Anuncio
- NotificacionModulos (id_notificacion) -> Notificacion
- NotificacionModulos (id_modulo) -> Modulo
- NotificacionEspecialidades (id_notificacion) -> Notificacion
- NotificacionEspecialidades (id_especialidad) -> Especialidad
- Curso (prerequisito) -> Curso

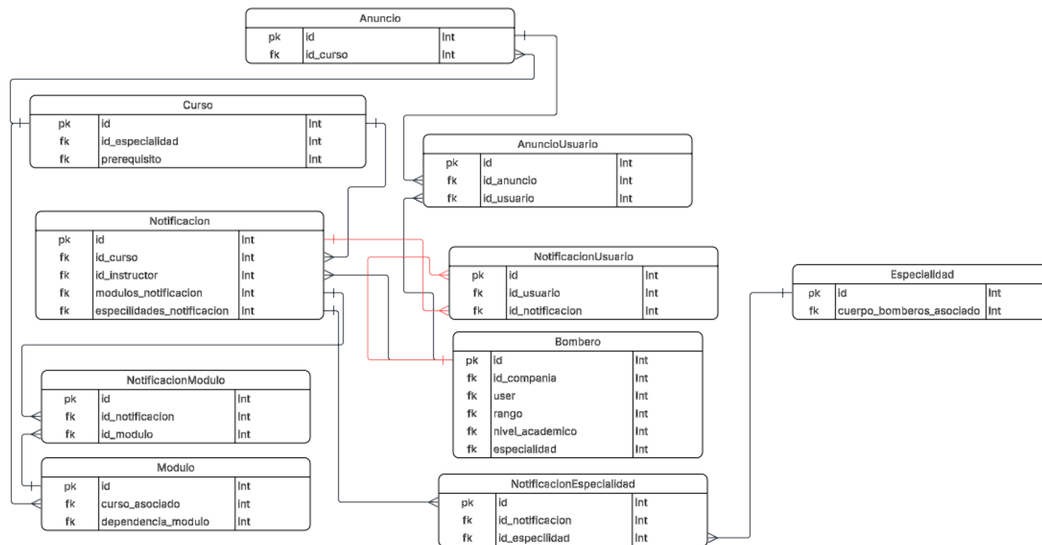

Figura 3.11 Relaciones de las tablas de Avisos
Fuente: Elaboración propia

3.3 Diseño de API

Como se ha mencionado, la **API** (Interfaz de Programación de Aplicaciones) es el componente que actúa como **puente** entre la **lógica de negocio/acceso a datos (Backend)** y la **aplicación cliente (Frontend)**, que es la interfaz final con la que interactúa el usuario.

Para este proyecto, se diseñó una **API REST** que expone un conjunto de **endpoints** (o rutas). Cada endpoint representa un **recurso** específico del sistema y permite realizar operaciones sobre él (como crear, leer, actualizar o eliminar, conocido como CRUD).

Los principales recursos y endpoints implementados son:

- **api/organizaciones/**: Gestiona la información de las entidades organizativas (ej. Cuerpos de Bomberos, Compañías, Regiones).
- **api/cursos/**: Administra todo lo relacionado con la oferta académica del Cuerpo de Bomberos (ej. creación de cursos, módulos, certificados).
- **api/users/**: Maneja la información de los usuarios (ej. Bomberos, instructores, administradores), incluyendo su autenticación y perfiles.
- **api/avisos/**: Controla la programación de notificaciones o anuncios por parte de instructores o administradores.
- **api/chatbot/**: Maneja la información entre el usuario administrador y el propio LLM (Large Language Model o Gran Modelo de Lenguaje).

Cada uno de estos endpoints, como se mencionó en el capitulo 2.4, poseen de mecanismo de seguridad el uso de un JWT, siendo este esencial para que no exista un acceso indebido de algún usuario anónimo a los datos entregados por la API.

Para mostrar de forma explícita aquello, se presenta el escenario en el cual un cliente (ya sea un usuario conectado mediante el Angular o un programa que haga petición a la API) quiera acceder a una de las endpoints enseñadas y que no posea alguna credencial, se le entrega la siguiente respuesta:

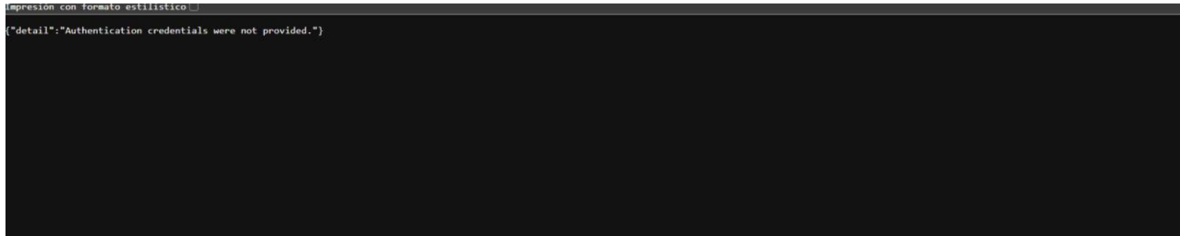

Figura 3.12 Mensaje provisto para accesos no autenticados (Navegador)
Fuente: imagen propia

Este mecanismo garantiza que solo el personal autorizado pueda interactuar con los recursos sensibles. A continuación, se detalla la estructura operativa de cada recurso expuesto por la API, definiendo los métodos HTTP permitidos y su función lógica dentro del sistema:

Organizaciones (api/organizaciones/)

Este endpoint actúa como la columna vertebral de la estructura jerárquica institucional. Permite modelar la realidad del Cuerpo de Bomberos (Compañías, Cuerpos, Regiones).

Tabla 3 Endpoints de organizaciones
Fuente: Elaboración Propia

Método HTTP	**Endpoint**	**Descripción Funcional**
GET	/api/organizaciones/compania/	Lista todas las compañías disponibles.
GET	/api/organizaciones/compania/{id}/	Obtiene detalles o capacidad de una compañía específica.
GET	/api/organizaciones/cuerpo-bomberos/	Lista los Cuerpos de Bomberos registrados.
GET	/api/organizaciones/mi-cuerpo-bomberos/	Obtiene la información del Cuerpo de Bomberos al que pertenece el usuario autenticado.

Método HTTP	**Endpoint**	**Descripción Funcional**
GET	/api/organizaciones/estadisticas-alumnos/	Devuelve métricas generales sobre los alumnos.
GET	/api/organizaciones/estadisticas-niveles/	Devuelve métricas agrupadas por nivel jerárquico o académico.
GET	/api/organizaciones/estadisticas-especialidades/	Devuelve métricas agrupadas por especialidad bomberil.
GET	/api/organizaciones/estudiantes-por-compania/	Lista o cuenta los estudiantes desglosados por compañía.
GET	/api/organizaciones/estadisticas-cursos/	Devuelve métricas sobre el rendimiento o estado de los cursos.

Cursos (api/cursos/)

Es el núcleo de la gestión académica. Este recurso no solo maneja la información básica del curso, sino que orquesta la relación entre módulos, archivos y usuarios.

Tabla 4 Endpoints de cursos
Fuente: Elaboración Propia

Método HTTP	**Endpoint**	**Descripción Funcional**
GET	/api/cursos/cursos/	Listar cursos disponibles y sus atributos correspondientes.
POST	/api/cursos/cursos/	Crear un nuevo curso (Uso en Administración).
GET, PUT, DELETE	/api/cursos/cursos/{id}/	Detalle, edición o eliminación de un curso.
GET	/api/cursos/modulos/	Listar módulos de aprendizaje.
POST	/api/cursos/modulos/	Crear un módulo dentro de un curso.
GET	/api/cursos/mis-cursos/	Cursos donde el usuario está inscrito.
POST	/api/cursos/inscripciones/	Inscribir a un alumno en un curso como también en uno o varios módulos.

Método HTTP	Endpoint	Descripción Funcional
GET	/api/cursos/material-estudio/	Listar archivos/PDFs de un módulo.
GET	/api/cursos/certificados/	Listar certificados de los usuarios.
GET	/api/cursos/certificados/{id}/pdf/	Descargar PDF del certificado.
POST	/api/cursos/evaluaciones/submit/	Enviar respuestas de una prueba.

Usuarios (api/users/)

Gestiona el ciclo de vida de los actores del sistema. Dada la naturaleza sensible de los datos personales de los bomberos, este endpoint posee las restricciones de seguridad más estrictas.

Tabla 5 Endpoints de users
Fuente: Elaboración Propia

Método HTTP	Endpoint	Descripción Funcional
POST	/api/users/login/	Autenticación (JWT Create).
POST	/api/users/refresh/	Refrescar token (JWT Refresh).
GET	/api/users/me/	Obtener perfil del usuario actual.
GET	/api/users/bomberos/	Listar todos los bomberos (Estudiantes/Instructores).
POST	/api/users/bomberos/	Registrar un nuevo bombero.
GET, PUT, PATCH	/api/users/bomberos/{id}/	Ver o editar perfil de un bombero.
GET	/api/users/instructores/	Listar instructores disponibles.
GET	/api/users/rangos/	Obtener lista de rangos.

Avisos (api/avisos/)

Funciona como un tablón de anuncios digital, permitiendo la comunicación asíncrona entre la administración y el cuerpo de bomberos o alumnos.

Tabla 6 Endpoints de avisos
Fuente: Elaboración Propia

Método HTTP	Endpoint	Descripción Funcional
GET	/api/avisos/eventos/	Listar eventos del calendario.
POST	/api/avisos/eventos/	Crear un evento en el calendario.
GET, PUT, DELETE	/api/avisos/eventos/{id}/	Gestionar un evento específico.
GET	/api/avisos/anuncios/	Listar noticias generales (Dashboard).
POST	/api/avisos/anuncios/	Publicar una noticia.
GET	/api/avisos/notificaciones/	Listar notificaciones no leídas del usuario.
POST	/api/avisos/notificaciones/marcar-leida/	Marcar notificación como vista.

Integración con IA (api/chatbot/)

A diferencia de los recursos anteriores que realizan operaciones CRUD sobre la base de datos local (PostgreSQL), el endpoint api/chatbot/ funciona como una **pasarela de integración** (Gateway) hacia servicios de Inteligencia Artificial externos.

Este diseño se implementó para proteger la clave de API (API Key) correspondiente al **LLM** y controlar el contexto de las conversaciones.

Tabla 7 Endpoints de chatbot
Fuente: Elaboración Propia

Método HTTP	Endpoint	Descripción Funcional
POST	/api/chatbot/chat/	Enviar prompt al modelo Gemini y recibir respuesta.
POST	/api/chatbot/upload-doc/	Subir documento PDF/Txt para contexto (RAG).
GET	/api/chatbot/history/	Recuperar historial de conversación previo.
DELETE	/api/chatbot/history/	Limpiar historial de chat.

El cliente (Angular) envía la consulta del usuario en formato JSON. El servidor (Django) intercepta esta solicitud y realiza las siguientes acciones antes de responder:

1. **Validación:** Verifica que el usuario tenga un JWT válido y cuota disponible para consultas.
2. **Contextualización:** Inyecta un "System Prompt" (instrucción de sistema) invisible para el usuario, que instruye al modelo a comportarse como un "Asistente Experto en Bomberos", limitando las respuestas al ámbito académico e institucional.
3. **Invocación a Gemini:** El servidor realiza una petición segura a la API de **Google Gemini Pro**, enviando el historial de conversación y la nueva pregunta.
4. **Respuesta:** Recibe la generación de texto del LLM, la procesa y la devuelve al Frontend para ser mostrada al usuario.

Este enfoque centralizado en el Backend evita exponer las credenciales en el código del cliente (navegador), manteniendo la seguridad de la infraestructura.

3.4 Arquitectura de Infraestructura y Despliegue en la Nube (Google Cloud Platform)

Introducción a la Infraestructura Cloud

Tras definir la arquitectura de software a nivel de aplicación (Cliente-Servidor), este capítulo aborda el diseño de la infraestructura tecnológica necesaria para soportar el despliegue, operación y escalabilidad del sistema. Se ha seleccionado **Google Cloud Platform (GCP)** como proveedor de servicios en la nube debido a su robustez, sus herramientas de gestión de contenedores y su modelo de facturación flexible.

La infraestructura propuesta adopta un enfoque **Serverless** (sin servidor) basado en contenedores, lo que permite abstraer la gestión del hardware subyacente y centrarse en la disponibilidad y el rendimiento de las aplicaciones.

Diseño de la Arquitectura de Despliegue

La arquitectura de despliegue replica la separación lógica del software (Frontend y Backend) en la infraestructura, utilizando servicios gestionados para cada capa del

sistema.

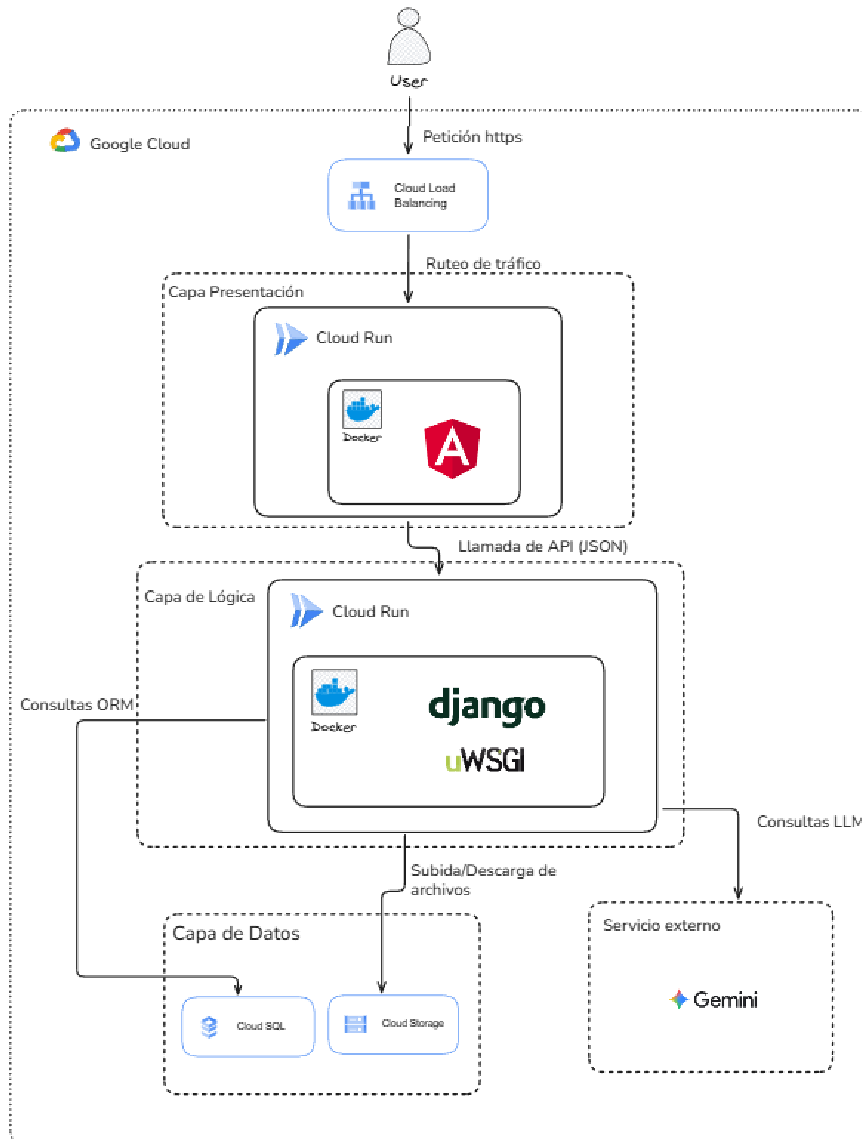

Figura 3.13 Capas existentes en la plataforma
Fuente: Elaboración propia

El flujo de comunicación y los componentes seleccionados son los siguientes:

1. **Capa de Presentación (Cliente):** Contenedor Docker con la aplicación **Angular**, servido mediante **Google Cloud Run**.
2. **Capa de Lógica y API (Servidor):** Contenedor Docker con la aplicación **Django REST Framework**, servido mediante una instancia independiente de **Google Cloud Run**.

3. **Capa de Datos (Persistencia):** Base de datos gestionada en **Google Cloud SQL** (PostgreSQL).
4. **Capa de Almacenamiento Estático:** Almacenamiento de archivos multimedia mediante **Google Cloud Storage**.

Implementación de Servicios de Cómputo: Cloud Run

El núcleo de la infraestructura se basa en **Google Cloud Run**, un servicio de computación gestionado que permite ejecutar contenedores invocables vía solicitudes HTTP.

Instancia del Cliente (Angular)

La aplicación de Angular (SPA) se "*dockeriza*" (empaqueta en un contenedor) utilizando un servidor web ligero como Nginx para servir los archivos estáticos compilados.

- **Función:** Servir la interfaz de usuario al navegador del usuario final.
- **Comunicación:** Esta instancia recibe las peticiones del navegador y, internamente, el código de Angular realiza llamadas HTTPS hacia la URL pública de la instancia del Backend.

Instancia del Servidor (Django API)

El Backend de Django se empaqueta en su propio contenedor Docker. Dado que Cloud Run es una plataforma efímera (los contenedores se crean y destruyen según la demanda), se configura para ser "stateless" (sin estado), delegando la persistencia de datos a servicios externos.

- **Función:** Procesar la lógica de negocio, autenticar usuarios y exponer los endpoints de la API REST.
- **Intercomunicación:** Actúa como el punto de entrada seguro para las solicitudes provenientes de la instancia de Cloud Run del Cliente.

Gestión de Concurrencia y Servidor de Aplicaciones (WSGI)

Aunque Django es un framework robusto, su servidor de desarrollo incorporado por defecto (runserver) no está diseñado para manejar cargas de producción ni múltiples peticiones simultáneas de manera eficiente. Para solucionar esto y maximizar la capacidad de respuesta dentro del entorno de Cloud Run, se implementó **uWSGI** como servidor de aplicaciones de producción.

El Rol de uWSGI

Django se comunica mediante el estándar **WSGI** (Web Server Gateway Interface). uWSGI actúa como un intermediario de alto rendimiento entre el tráfico HTTP entrante (gestionado por la infraestructura de Google) y el código Python de Django.

Su función crítica en esta arquitectura es la **gestión de procesos y hilos (workers and threads)**. Mientras que una instancia simple de Python solo puede atender una solicitud a la vez (bloqueando las demás), uWSGI permite configurar múltiples "trabajadores" dentro de un mismo contenedor **[9]**.

Sinergia con el Escalamiento de Cloud Run

La arquitectura logra una **doble capa de gestión de carga**, combinando la capacidad interna del contenedor con la elasticidad de la nube:

1. **Nivel Interno (Concurrencia de uWSGI):** Dentro de cada instancia de contenedor desplegada en Cloud Run, uWSGI se configura para levantar múltiples procesos (workers). Esto permite que **un solo contenedor** atienda varias solicitudes HTTP simultáneamente (multitasking) sin necesidad de crear una nueva instancia, optimizando el uso de la CPU y la memoria RAM asignada (pagando solo por lo que se usa).

2. **Nivel Externo (Escalamiento de Cloud Run):** Cloud Run monitorea el tráfico entrante. Se configuró un parámetro de **concurrencia máxima** (por ejemplo, 80 peticiones simultáneas por instancia).
   - Si llegan 50 peticiones, uWSGI las distribuye entre sus *workers* internos dentro de un solo contenedor.
   - Si la demanda supera la capacidad de concurrencia configurada (ej. llegan 200 peticiones de golpe), Cloud Run detecta la saturación y **automáticamente inicia nuevos contenedores** (replica la infraestructura) para absorber el exceso de tráfico.

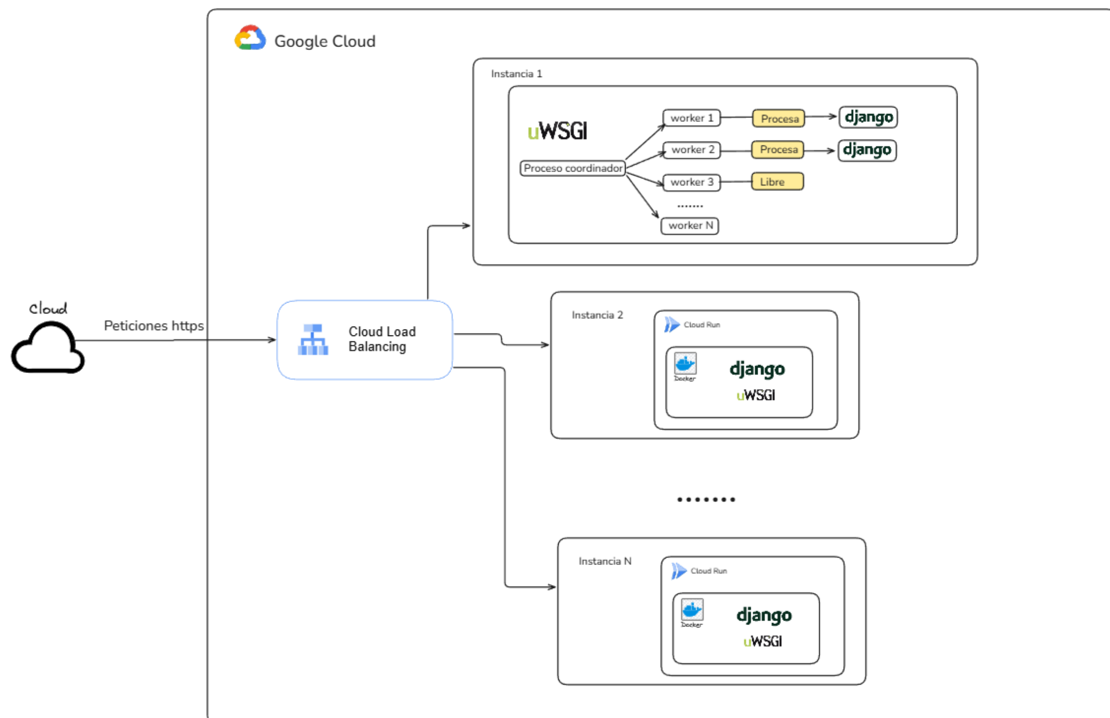

Figura 3.14 Funcionamiento sincronizado entre Cloud Run y uWSGI

Fuente: Elaboración propia

Justificación de la Estrategia

Esta combinación es ideal para el modelo de **pago por uso**. uWSGI exprime al máximo los recursos de un contenedor activo (haciendo eficiente el gasto por milisegundo de CPU), mientras que Cloud Run asegura que, si esa eficiencia interna no es suficiente ante un pico de usuarios (ej. 600 usuarios intentando loguearse a la vez), la infraestructura crezca horizontalmente sin intervención humana y sin degradar el servicio.

Persistencia y Almacenamiento

Base de Datos: Cloud SQL

Para garantizar la integridad y disponibilidad de los datos, se utiliza **Cloud SQL**, el servicio de bases de datos relacionales totalmente gestionado de Google.

- Se ha configurado una instancia de **PostgreSQL** que se conecta de forma segura con el servicio de Cloud Run del Backend.
- Esta elección elimina la necesidad de gestionar parches de seguridad, réplicas o copias de seguridad manuales, tareas que son automatizadas por la plataforma.

Archivos Estáticos y Media: Cloud Storage

Debido a que los contenedores de Cloud Run son efímeros (su sistema de archivos se reinicia cuando el contenedor se apaga), no es posible guardar archivos subidos por los usuarios (como fotos de perfil o documentos PDF) directamente en el servidor.

- Se implementó **Google Cloud Storage** (Buckets) para almacenar estos objetos.
- Django se configura mediante una función propia que hace uso de la librería de Google, para que, cuando un usuario suba un archivo, este se envíe directamente al Bucket de Cloud Storage, garantizando su persistencia.

Escalabilidad y Modelo de Costos

Uno de los requisitos no funcionales críticos del proyecto es la eficiencia de costos considerando un uso inicial estimado de **600 usuarios**, pero con la capacidad de crecer sin reingeniería.

Modelo de Pago por Uso (Pay-as-you-go)

Cloud Run opera bajo un modelo de pago por uso exacto (se cobra por cada 100 milisegundos de procesamiento de CPU y memoria asignada).

- **Ventaja Económica:** Dado que el sistema es de uso académico/administrativo, es probable que tenga picos de uso durante el día y tráfico nulo durante la noche. Cloud Run tiene la capacidad de **escalar a cero (si se configura previamente)**; es decir, si nadie está usando el sistema, no se generan costos de cómputo, lo cual es ideal para el presupuesto del proyecto.

Estrategia de Escalabilidad Automática

La infraestructura está diseñada para la elasticidad o, en otras palabras, poder escalar en función al uso requerido por la plataforma. Aunque actualmente la plataforma está desarrollada en torno a la interacción con un estimado máximo de 600 usuarios:

1. Si la demanda aumenta repentinamente (ej. durante un periodo de uso intensivo debido al aumento de peticiones por parte de varios usuarios/clientes a la vez), Cloud Run detectará el aumento de tráfico HTTP.
2. Automáticamente provisionará nuevas instancias del contenedor (réplicas) para manejar la carga, balanceando el tráfico entre ellas.
3. Una vez que la demanda baja, elimina las instancias sobrantes. Esto asegura que el sistema no se sature ni se caiga, sin necesidad de intervención manual por parte del administrador.

Elección de infraestructura

La elección de una arquitectura basada en contenedores sobre Google Cloud permite cumplir con los objetivos de modernidad y flexibilidad. La separación de los servicios de Cloud Run para el Cliente y el Servidor mantiene el desacoplamiento arquitectónico en producción, mientras que el uso de Cloud SQL y Storage garantiza la seguridad de los datos, todo bajo un esquema de costos optimizado para la escala actual y futura de la organización.

Análisis de Alternativas de Infraestructura: Serverless vs. IaaS (Máquinas Virtuales)

Para validar la elección de Google Cloud Run como plataforma de despliegue, se realizó un análisis comparativo frente al modelo tradicional de Infraestructura como Servicio (IaaS) basado en Máquinas Virtuales (VM). Esta evaluación se centró en tres ejes críticos para el proyecto: carga operativa, escalabilidad y eficiencia de costos.

Opción A: Modelo Tradicional (Máquinas Virtuales / Compute Engine)

En este esquema, el despliegue implicaría provisionar un servidor virtual (VPS) con un sistema operativo completo (ej. Ubuntu Linux), instalar las dependencias (Python, Nginx, Node.js), configurar el firewall y gestionar manualmente los servicios.

- **Gestión (Carga Operativa Alta):** El equipo de desarrollo es responsable de parchear el sistema operativo, configurar la seguridad de la red y mantener el servidor actualizado. Si el servidor falla, la recuperación es manual.
- **Escalabilidad (Lenta):** Para escalar horizontalmente, se requiere configurar "Grupos de Autoescalado" complejos que clonen la máquina virtual. El tiempo de arranque de una nueva VM es de minutos, lo cual es lento para picos repentinos de tráfico.
- **Costos (Ineficiencia):** En el modelo de VM, se paga por la capacidad reservada (CPU/RAM y almacenamiento) las 24 horas del día, independientemente de si el sistema está siendo usado. Para una plataforma con uso esporádico (como inscripciones de bomberos), esto implica pagar por tiempo ocioso ("*pagar por aire*").

Opción B: Modelo Serverless Containers (Cloud Run - Seleccionado)

En el modelo seleccionado, se abstrae totalmente la gestión del servidor. Google Cloud gestiona la infraestructura subyacente, y el desarrollador solo es responsable de entregar un contenedor Docker.

- **Gestión (Carga Operativa Nula):** No hay sistema operativo que mantener ni parches de seguridad que aplicar al servidor host. Google garantiza la disponibilidad de la plataforma.
- **Escalabilidad (Inmediata):** Cloud Run puede pasar de 0 a 100 instancias (incluso 1000 si se hacen ajustes en la consola de Google) en segundos. Esto es vital para manejar la concurrencia descrita en la sección 5.3.3 (uWSGI), donde el sistema reacciona automáticamente a la demanda.
- **Costos (Pago por Uso):** Solo se factura cuando el código está procesando una solicitud. Si nadie usa el sistema de noche, el costo es cero.

Tabla Comparativa de Decisión

A continuación, se resume el análisis que llevó a descartar el uso de Máquinas Virtuales en favor de Cloud Run:

Tabla 8 Comparativa entre Cloud Run y Máquinas virtuales
Fuente: Elaboración propia

Criterio	**Máquinas Virtuales (IaaS)**	**Cloud Run (Serverless)**	**Veredicto para el Proyecto**
Unidad de Despliegue	Servidor Completo (OS + App)	Contenedor Docker (Solo App)	Cloud Run gana por portabilidad y ligereza.
Mantenimiento	**Manual:** Actualizar OS, configurar SSH, seguridad.	**Automático:** Gestionado por Google.	Cloud Run permite al equipo centrarse en el código, no en el servidor.
Escalado	Lento (Minutos). Requiere configuración compleja.	Rápido (Segundos). Nativo y automático.	Cloud Run maneja mejor los picos de inscripción de cursos.
Modelo de Costos	**Costo Fijo:** Se paga por servidor encendido (24/7).	**Costo Variable:** Se paga por petición (100ms).	Cloud Run es más económico para una base de 600 usuarios con uso intermitente.
Configuración	Archivos de configuración dispersos en el servidor.	Inmutable (Dockerfile).	Cloud Run garantiza que Desarrollo y Producción sean idénticos.

Conclusión de la Selección de Infraestructura

El análisis demuestra que el uso de Máquinas Virtuales introduciría una carga administrativa innecesaria y costos fijos injustificados para el tamaño actual del proyecto

(+600 usuarios). La arquitectura Serverless con Cloud Run optimiza el presupuesto y garantiza que la infraestructura pueda crecer orgánicamente junto con la institución de Bomberos sin requerir reingeniería.

4. Validación de solución

4.1 Validación de Infraestructura y Rendimiento

Para validar el comportamiento de la arquitectura propuesta bajo condiciones reales de operación, se extrajeron y analizaron las métricas de telemetría generadas por la plataforma Google Cloud Run durante un periodo de actividad representativo. El objetivo de este análisis fue contrastar el comportamiento teórico del diseño *Serverless* con la evidencia empírica en tres ejes fundamentales: escalabilidad, latencia y eficiencia de costos **[1,2]**.

A. Comportamiento del Tráfico y Respuesta Elástica

El análisis de los registros de volumen de solicitudes revela un patrón de tráfico variable y no lineal, comportamiento característico de los sistemas de gestión académica donde la demanda fluctúa drásticamente según horarios y eventos administrativos.

Como se evidencia en la **Figura 4.1**, el sistema experimentó picos de demanda significativos intercalados con periodos de baja actividad. La infraestructura respondió a estos eventos instanciando automáticamente nuevos contenedores para absorber la carga, manteniendo la disponibilidad del servicio sin requerir intervención manual. No se registraron errores de saturación (códigos de estado HTTP 503) durante los aumentos repentinos, lo que valida la configuración de auto-escalado horizontal definida en la arquitectura.

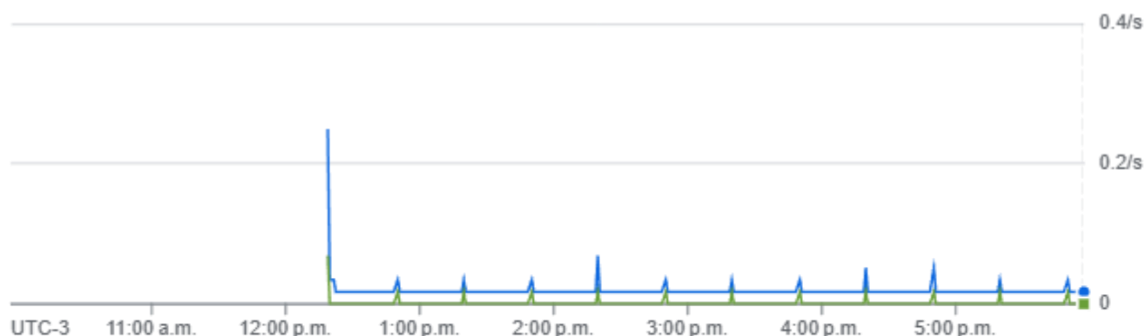

Figura 4.1 Flujo de solicitudes entrantes al servidor. Los picos representan momentos de alta concurrencia gestionados automáticamente por la plataforma.
Fuente: Consola Google Cloud

B. Latencia y Experiencia de Usuario

Para evaluar la calidad del servicio percibida por el usuario final, se realizó un desglose estadístico de los tiempos de respuesta del servidor (Backend), utilizando percentiles para identificar tanto el rendimiento estándar como los casos atípicos.

- **Rendimiento Base (p50):** La mediana de los tiempos de respuesta se mantuvo en un rango óptimo de milisegundos. Esto indica que, para la mayoría de las interacciones (como la navegación general o consultas simples), la plataforma ofrece una experiencia fluida.
- **Fenómeno de Arranque en Frío (p99):** Al analizar el 1% de las solicitudes más lentas (percentil 99), se identificaron picos de latencia que superan el promedio. El cruce de estos datos con el desglose de tiempos de procesamiento confirmó que estos eventos corresponden a *Cold Starts* (Arranques en Frío). Este fenómeno ocurre cuando la plataforma debe iniciar un contenedor desde cero tras un periodo de inactividad para atender una nueva solicitud.

Aunque esto introduce una latencia inicial mayor en momentos específicos, se considera un compromiso aceptable asociado a la tecnología *Serverless,* a cambio de la eficiencia de recursos lograda.

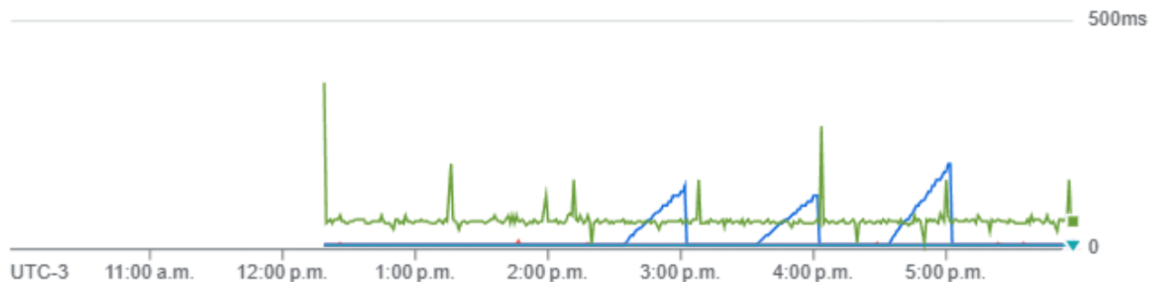

Figura 4.2 Análisis de latencia del servicio. La línea inferior (p50 de color azul) muestra la estabilidad en operaciones normales, mientras que los picos superiores (p99 de color verde) evidencian los tiempos de aprovisionamiento de nuevas instancias.
Fuente: Consola Google Cloud

C. Eficiencia de Costos y Uso de Recursos

Uno de los objetivos críticos fue optimizar el presupuesto operativo de la institución. El análisis de los tiempos de instancia facturables demuestra la efectividad del modelo de "Pago por Uso".

Al correlacionar el volumen de tráfico con el tiempo de CPU facturado, se observa un ajuste dinámico. A diferencia de una arquitectura tradicional basada en Máquinas Virtuales (donde el costo es una línea plana constante las 24 horas, independientemente del uso), la gráfica de consumo de recursos de esta solución muestra valles cercanos a cero durante los horarios de inactividad.

Esto confirma que la arquitectura propuesta elimina el desperdicio de recursos por capacidad ociosa ("*pagar por aire*"), generando costos únicamente cuando el sistema está aportando valor real al procesar transacciones.

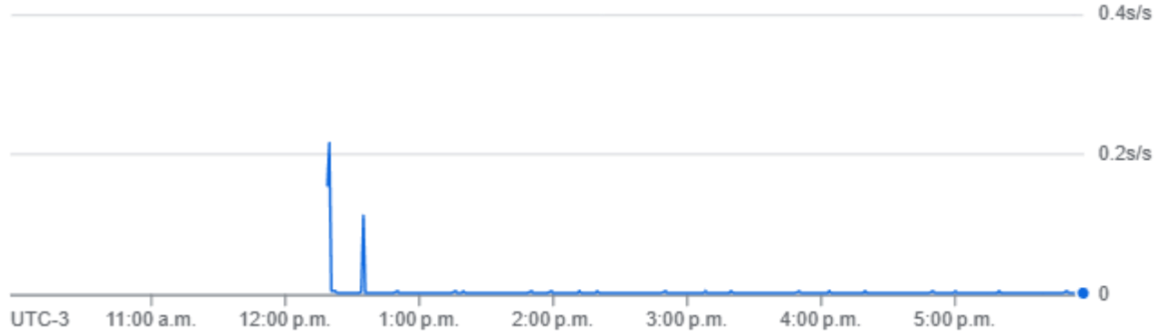

Figura 4.3 Relación de tiempo de instancia facturable. La gráfica demuestra cómo el consumo de recursos desciende automáticamente cuando disminuye la demanda, optimizando el gasto operativo **(Feria de Software)**.
Fuente: Consola Google Cloud

4.2 Análisis Comparativo: Situación Actual vs. Solución Propuesta

A continuación, se presenta un cuadro comparativo que demuestran la mejora en los procesos de gestión académica, siendo los datos correspondientes a "Situación Actual" basados en datos cualitativos y cuantitativos entregados por la organización:

Tabla 9 Comparativa de Indicadores de Rendimiento
Fuente: elaboración propia

Indicador / Proceso	Situación Actual (Manual/Legacy)	Solución Implementada (Plataforma Web)	Mejora
Tiempo de Inscripción	15 - 20 minutos (Trámite físico/email)	45 segundos (Portal Web)	**~95% de ahorro**
Disponibilidad del Servicio	Limitada a horario de oficina (L-V)	24/7 (Alta Disponibilidad en GCP)	**100% disponible**
Error en Datos de Usuario	Alto (Errores de digitación manual)	Nulo (Validación en tiempo real en Angular)	**Eliminación de error**
Obtención de Certificado	3 a 5 días hábiles o incluso no es posible	Inmediata (Descarga de PDF)	**Rápida**

4.3 Conclusiones de la Validación

Los resultados obtenidos a partir de la telemetría confirman que la arquitectura basada en contenedores y servicios gestionados (**Cloud Run** y **Cloud SQL**) satisface los requisitos no funcionales del sistema. Las pruebas de carga y el análisis de costos validan tres aspectos fundamentales:

1. **Resiliencia:** La infraestructura es capaz de absorber picos de tráfico sin degradar el servicio gracias al auto-escalado horizontal.
2. **Rendimiento:** Los tiempos de latencia se mantienen dentro de márgenes óptimos para una buena experiencia de usuario, mitigando el impacto de los arranques en frío.
3. **Sostenibilidad:** El modelo de facturación por uso demostró ser económicamente eficiente, eliminando gastos por infraestructura ociosa.

En consecuencia, la solución tecnológica se considera validada para su paso a un entorno productivo.

5. Conclusión general y trabajo futuro

5.1 Síntesis y Cumplimiento de Objetivos

El desarrollo de la Plataforma Académica ha permitido consolidar una solución tecnológica adecuada para la gestión académica del Cuerpo de Bomberos de Viña del Mar, cumpliendo con el objetivo general de diseñar una arquitectura de software moderna y escalable.

La adopción del modelo arquitectónico **Cliente-Servidor desacoplado** (Angular y Django) demostró ser la decisión correcta frente a alternativas monolíticas. Esta separación no solo facilitó la especialización del desarrollo, sino que permitió validar, mediante el modelo de documentación de **"4+1 Vistas"**, que el sistema es capaz de soportar la concurrencia esperada de 600 usuarios sin comprometer el rendimiento.

Los resultados obtenidos en la fase de validación confirman que la integración de servicios en la nube (**Google Cloud Run**) y la gestión eficiente de peticiones (**uWSGI**) resuelven las problemáticas de disponibilidad y latencia que afectaban a los procesos anteriores. Se logró transformar un flujo de trabajo manual, asíncrono y propenso a errores, en una plataforma operativa 24/7.

5.2 Aportes e Impacto en la organización

La principal contribución de este trabajo trasciende el código; radica en la **modernización de los procesos formativos** de la institución.

1. **Optimización de Recursos:** La implementación de una arquitectura *Serverless* ("sin servidor") introduce un modelo de costos basado en el uso real (*Pay-as-you-go*), eliminando gastos por infraestructura ociosa y garantizando la sostenibilidad financiera del proyecto a largo plazo.

2. **Integridad y Centralización:** Se eliminó la dispersión de información (planillas aisladas), centralizando los datos en una base relacional robusta (**Cloud SQL**) que asegura la consistencia de los historiales académicos de los voluntarios.

3. **Innovación Operativa:** La incorporación de Inteligencia Artificial (**Gemini**) como asistente virtual marca un hito tecnológico para la organización, democratizando el acceso a la información técnica y reduciendo la carga de consultas repetitivas sobre los instructores humanos.

Como consecuencia directa de estas implementaciones, la solución **transforma significativamente la operatividad de la institución**. Los indicadores cualitativos y cuantitativos observados proyectan una **reducción drástica en los tiempos administrativos** y una mejora sustancial en la fiabilidad de la información, validando la hipótesis de modernización planteada al inicio de esta memoria.

5.3 Alcances y Limitaciones de la Propuesta

Es importante reconocer las fronteras del trabajo realizado para contextualizar su alcance:

- **Dependencia de Conectividad:** Al ser una solución 100% basada en la nube (Cloud Native), la operatividad del sistema depende estrictamente de la conexión a internet de los cuarteles y usuarios, lo cual puede ser una limitante para compañías en zonas rurales extremas del país.

- **Acoplamiento con el Proveedor:** Si bien el uso de contenedores Docker ofrece portabilidad, la arquitectura aprovecha servicios gestionados específicos de Google Cloud (como Cloud Run y Cloud SQL). Una eventual migración a otro proveedor (AWS o Azure) requeriría un esfuerzo de reingeniería en la capa de infraestructura.

- **Latencia de la IA:** Aunque funcional, la respuesta del módulo de Inteligencia Artificial depende de los tiempos de procesamiento de la API externa, lo que introduce una variable de rendimiento que no está totalmente bajo el control del sistema.

5.4 Recomendaciones y Trabajo Futuro

Para dar continuidad a este proyecto y maximizar su vida útil, se proponen las siguientes líneas de acción en investigación y desarrollo:

1. **Implementación de PWA (Progressive Web App):** Evolucionar el cliente Angular hacia una PWA permitiría ofrecer funcionalidades *offline* (sin conexión), mitigando la limitante de conectividad en zonas aisladas y permitiendo a los bomberos consultar manuales básicos sin internet (si fuese necesario).

2. **Arquitectura de Microservicios:** Si la institución crece y se agregan módulos complejos (ej. Gestión de Inventario de Material Mayor o Despacho de Emergencias), se recomienda evaluar la transición desde la arquitectura actual hacia microservicios independientes para evitar crear un "monolito distribuido".

3. **Automatización DevOps (CI/CD):** Profundizar en la implementación de pipelines de despliegue continuo (con Jenkins) para automatizar las pruebas unitarias y el paso a producción, reduciendo el riesgo de error humano en las actualizaciones del sistema.

5.5 Conclusiones generales

En conclusión, este proyecto pone a disposición del Cuerpo de Bomberos de Viña del Mar una herramienta tecnológica moderna que contribuye significativamente a la modernización de su gestión académica. Más que una solución que cierra el ciclo por completo, la plataforma se presenta como un instrumento de apoyo fundamental que, en la medida de su adopción progresiva, ayudará a mitigar las ineficiencias de los procesos manuales y facilitará el control de la información formativa.

Uno de los desafíos más significativos abordados durante el desarrollo fue la complejidad de la lógica operativa de la organización. La escasez de documentación formal sobre los flujos de información internos y la particular estructura jerárquica exigieron un exhaustivo proceso de reconsiderar los requisitos y la forma de plantear el diseño de la arquitectura. Fue necesario realizar un exhaustivo levantamiento de requerimientos y la formalización de los procesos operativos de la institución para avanzar en la construcción de la plataforma. Este proceso evidenció que el éxito del software no depende únicamente de la calidad del código, sino de la capacidad de analizar, estructurar y adaptarse tanto a la realidad del usuario como a la disponibilidad efectiva de la información.

Asimismo, el proceso de construcción del sistema permitió cerrar una importante brecha tecnológica. Se demostró que es viable acercar herramientas de vanguardia (como Cloud Computing e Inteligencia Artificial) a organizaciones que tradicionalmente no han contado con estos recursos, entregándoles una base sólida para su evolución digital.

Finalmente, este trabajo deja establecidos los cimientos para que la institución pueda avanzar hacia una gestión más eficiente y transparente. La arquitectura diseñada queda disponible para que el Cuerpo de Bomberos continúe escalando sus capacidades, permitiéndoles centrar sus esfuerzos en lo más importante: la formación de profesionales para salvar vidas, con la tranquilidad de contar con un sistema que respalda su labor administrativa.

6. Agradecimientos

Quiero agradecer a todas las personas que me apoyaron en el transcurso de mi carrera, entre los cuales se encuentran mis amigos, compañeros y profesores.

Por último, y de manera muy especial, dedico este trabajo a mi familia. Gracias por ser mi pilar fundamental, por su apoyo incondicional y por creer en mí, impulsándome a seguir adelante tanto en mis mejores momentos como en los más difíciles. Este logro es también de ustedes.

7. Referencias

[1]. Google Cloud. "Cloud Run Documentation - Serverless container platform." Google Cloud. Disponible en: https://cloud.google.com/run/docs.

[2]. Google Cloud. "Precios de Cloud Run (Cloud Run Pricing)." Google Cloud. Disponible en: https://cloud.google.com/run/pricing.

[3]. Google Cloud. "Cloud SQL for PostgreSQL Documentation." Google Cloud. Disponible en: https://cloud.google.com/sql/postgresql/docs.

[4]. Google Cloud. "Cloud Storage Documentation." Google Cloud. Disponible en: https://cloud.google.com/storage/docs.

[5]. Google AI. "Gemini API Documentation - Generative AI for Developers." Google AI for Developers. Disponible en: https://ai.google.dev/docs.

[6]. Django Software Foundation. "Django Documentation." Disponible en: https://docs.djangoproject.com/.

[7]. Encode. "Django REST Framework - The official guide." Disponible en: https://www.django-rest-framework.org/.

[8]. Google. "Angular Documentation - The modern web developer's platform." Disponible en: https://angular.dev/overview.

[9]. Unbit. "The uWSGI project documentation." Disponible en: https://uwsgi-docs.readthedocs.io/.

[10]. IETF (Internet Engineering Task Force). "RFC 7519 - JSON Web Token (JWT)." Disponible en: https://datatracker.ietf.org/doc/html/rfc7519.

[11]. Kruchten, P. (1995). "Architectural Blueprints—The '4+1' View Model of Software Architecture." *IEEE Software*, 12(6), 42–50. (Fuente del modelo de vistas utilizado en el Cap. 3).

[12]. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3.ª ed.). Addison-Wesley Professional. (Fuente para definiciones de atributos de calidad).

[13]. Tapia Durán, A. I. (2014). *Arquitectura de Software para el Entorno Computacional de KUALI-BEH*. Tesis de Maestría, UNAM. (Referencia de metodológica utilizada).