CoProof: Arquitectura

David Alejandro López Torres, 22310432
Daniel Tejeda Saavedra, 22310431
Emiliano Flores Márquez, 22110044

Febrero 17, 2026

1 Arquitectura General

1.1 Modelo Arquitectónico

CoProof implementa una arquitectura hexagonal basada en microservicios distribuidos. El Backend actúa como núcleo central que orquesta las interacciones, mientras los servicios especializados se conectan como adaptadores intercambiables. Esta estructura aísla la lógica de negocio de las fluctuaciones tecnológicas en componentes externos como modelos de IA o motores de verificación formal.

1.2 Inventario de Microservicios

1.2.1 Backend (API Gateway)

Punto de entrada único del sistema. Centraliza autenticación, gestión de sesiones y enrutamiento de tráfico. Gestiona permisos de colaboración, coordina sincronización en tiempo real vía WebSockets y mantiene coherencia de datos entre usuarios. Delega tareas a servicios especializados, evitando interacción directa del cliente con infraestructura de cómputo o motores de inferencia.

1.2.2 LeanServer

Encapsula el kernel de Lean 4 en un entorno contenedorizado. Responsable exclusivo de validación matemática: recibe código o tácticas y retorna veredictos de corrección. El aislamiento protege al sistema de bloqueos o consumo excesivo de recursos por bucles infinitos en demostraciones.

1.2.3 Servicios de Inteligencia Artificial

Traductor NL2FL: Transforma lenguaje natural y LaTeX en código Lean formal.
Agentes de Sugerencia: Generan estrategias de prueba y tácticas.
Ambos operan como motores de inferencia stateless. Requieren hardware especializado (GPUs) y admiten actualización de modelos sin interrumpir el editor ni modificar lógica de negocio.

1.2.4 Cluster OHPC

Infraestructura de alto rendimiento para tareas computacionalmente intensivas: búsqueda de contraejemplos, validación masiva de lemas. Opera asincrónicamente, distribuyendo cálculos entre nodos ARM paralelos. Notifica resultados al usuario sin bloquear la interfaz de edición.

1.2.5 Git-Engine

Sistema de control de versiones dedicado. Almacena y versiona archivos de demostración (.lean) con historial completo de cambios, ramas y fusiones. Independiente del Backend, que sólo gestiona metadatos.

1.2.6 SQL Service

Base de datos relacional integrada en el Backend. Almacena metadatos de usuarios, proyectos, permisos y configuraciones. Permite consultas rápidas y transacciones ACID para garantizar consistencia.

1.3 Justificación Arquitectónica

Separación de responsabilidades: Componentes determinísticos (Lean) coexisten con componentes probabilísticos (IA) sin afectar mutuamente su estabilidad.

Escalabilidad heterogénea: Cada servicio especifica sus requisitos de hardware. IA requiere GPUs, LeanServer exige CPU de alto rendimiento single-thread, Cluster demanda concurrencia masiva.

Aislamiento de fallos: Fallos en verificación matemática se contienen en el LeanServer. Reinicio automático de contenedores sin afectar sesiones colaborativas ni acceso a datos.

Modularidad: Actualización de modelos de IA o motores sin modificar el núcleo del sistema.

2 Protocolos de Comunicación y Flujos de Datos

2.1 Estrategia de Intercomunicación

2.1.1 Comunicación Cliente-Servidor

HTTP/REST: Operaciones transaccionales estándar (autenticación, gestión de proyectos). Formato JSON para interoperabilidad entre Python, Lean y TypeScript.

WebSockets (WSS): Transmisión bidireccional de baja latencia para edición colaborativa en tiempo real.

2.1.2 Comunicación Interna

Modelo de orquestación centralizada. El API Gateway es el único coordinador. Los microservicios periféricos no establecen conexiones directas entre sí, sólo responden a solicitudes del Gateway.

Llamadas síncronas: APIs REST internas o gRPC sobre red privada. Usadas para verificación en LeanServer o consultas a IA que requieren respuesta inmediata.

Llamadas asíncronas: Colas de mensajes gestionadas por Redis y Celery. Usadas para operaciones intensivas (Cluster OHPC, Git-Engine). Desacoplan hilo principal del servidor, evitando bloqueos en interfaz de usuario.

2.2 Flujos de Datos por Caso de Uso

2.2.1 Autenticación y Creación de Proyectos

1. Cliente envía credenciales cifradas vía HTTPS POST al Gateway
2. Gateway valida identidad contra base de datos SQL

3. Si exitoso, emite token JWT
4. Al crear proyecto, Gateway ejecuta transacción distribuida:
   - Registra metadatos y permisos en PostgreSQL
   - Instruye a Git-Engine para inicializar repositorio bare
5. Proyecto queda con representación relacional (búsquedas rápidas) y física (control de versiones)

2.2.2 Edición Colaborativa en Tiempo Real

1. Cliente establece túnel WebSocket persistente con Backend
2. Usuario escribe carácter o táctica, evento se transmite como delta
3. Gateway actúa como broadcaster, propagando cambio a colaboradores conectados
4. Simultáneamente, Gateway envía estado actual a LeanServer
5. LeanServer recompila fragmento y retorna estado de prueba
6. Gateway inyecta validación en canal WebSocket
7. Todos los usuarios ven indicadores sincronizados de error/éxito

2.2.3 Traducción de Lenguaje Natural

1. Usuario envía texto (lenguaje natural o LaTeX) al Gateway
2. Gateway redirige a microservicio Traductor NL2FL
3. Traductor analiza entrada con LLMs y genera código Lean formal
4. Código generado se envía internamente a LeanServer para validación
5. Sólo si LeanServer confirma sintaxis válida, Gateway retorna respuesta al usuario
6. Flujo triangular (Usuario → Traductor → Verificador → Usuario) filtra alucinaciones antes de llegar a interfaz

2.2.4 Cómputo Intensivo Asíncrono

1. Usuario solicita operación (búsqueda de contraejemplos) al Gateway
2. Gateway valida permisos y coloca trabajo en cola de Cluster OHPC
3. Gateway responde inmediatamente con ID de trabajo, libera conexión HTTP
4. Cluster distribuye cálculos entre nodos ARM
5. Al finalizar, escribe resultados en base de datos y notifica Gateway vía Webhook
6. Gateway alerta usuario a través de WebSocket o notificación en interfaz

2.3 Orquestación con Kubernetes

Ingress Controller: Punto de entrada único desde internet. Maneja terminación SSL y distribuye tráfico hacia pods del Backend según carga actual.

Auto-reparación: Kubernetes detecta contenedores sin respuesta (liveness probe) y los reinicia automáticamente sin afectar sistema.

Escalado horizontal selectivo: Incrementa réplicas de servicios específicos según demanda. Ejemplo: aumentar contenedores de Agentes IA en nodos GPU mientras Backend permanece estable.

Gestión eficiente: Asigna hardware costoso sólo donde es necesario. Sistema escala de 50 a miles de sesiones concurrentes agregando nodos al clúster.

3 Especificación Tecnológica por Microservicio

3.1 Backend - API Gateway y Gestión de Datos

3.1.1 Stack Tecnológico

- **Framework:** Flask (Python)
- **ORM:** SQLAlchemy con PostgreSQL
- **Validación:** Marshmallow para serialización, deserialización y validación de esquemas
- **Tareas asíncronas:** Celery respaldado por Redis
- **WebSockets:** Flask-SocketIO

3.1.2 Patrones de Diseño

Repository: Abstrae consultas de base de datos. Lógica de negocio interactúa con objetos de dominio, no con SQL directo.

Data Transfer Object (DTO): Marshmallow valida contratos de entrada/salida antes de tocar base de datos o enviar al cliente.

Observer: WebSockets notifican cambios de estado a clientes conectados.

3.1.3 Responsabilidades

Centraliza lógica de negocio, autenticación, validación de datos y persistencia de metadatos. Única fuente de verdad para información estructurada del sistema.

3.2 LeanServer - Verificación Formal

3.2.1 Stack Tecnológico

- **Motor:** Lean 4 (binario oficial)
- **Wrapper:** Python para gestión de subprocesos y estandarización de E/S

- **Contenedorización:** Docker para portabilidad y aislamiento

3.2.2 Patrones de Diseño

Adapter: Transforma salida nativa del compilador Lean en estructuras JSON estandarizadas.

Object Pool: Mantiene instancias del proceso Lean precalentadas, eliminando latencia de arranque.

Sidecar: Script Python gestiona ciclo de vida del proceso Lean en deployment.

3.2.3 Responsabilidades

Recibe código fuente, ejecuta contra kernel Lean 4, retorna veredicto estructurado de validez. Operación aislada y efímera para evitar contaminación de estados.

3.3 Servicios de Inteligencia Artificial

3.3.1 Stack Tecnológico

- **Framework:** Python con cliente HTTP
- **API:** Interfaz compatible con especificación OpenAI
- **Procesamiento asíncrono:** Celery para llamadas a APIs externas

3.3.2 Patrones de Diseño

Strategy: Permite cambiar dinámicamente proveedor de IA o modelo específico (GPT-4, modelos locales) mediante configuración.

Circuit Breaker: Maneja fallos o timeouts en APIs de terceros, asegurando degradación elegante del servicio.

3.3.3 Responsabilidades

Pasarela de inferencia flexible (Bring Your Own Key). Construcción de prompts y comunicación HTTP delegando procesamiento pesado a proveedores externos.

3.4 Cluster OHPC - Cómputo Distribuido

3.4.1 Stack Tecnológico

- **Sistema Operativo:** Rocky Linux (estabilidad empresarial, compatibilidad binaria con RHEL)
- **Orquestación HPC:** OpenHPC con Slurm para gestión de colas
- **Aprovisionamiento:** Warewulf con PXE para boot-from-network
- **Comunicación:** RoCE para baja latencia
- **Integración:** Workers Celery en Python interactuando con Slurm

3.4.2 Patrones de Diseño

Master-Worker: Arquitectura de infraestructura con distribución de tareas.
Stateless: Nodos de procesamiento sin estado con boot-from-network.
Immutable Infrastructure: Cada reinicio carga imagen limpia de Rocky Linux para entorno determinista.

3.4.3 Responsabilidades

Ejecución masiva y paralela de tareas matemáticas. Escalable y stateless para búsquedas exhaustivas y validaciones intensivas.

3.5 Git-Engine - Control de Versiones

3.5.1 Stack Tecnológico

- **Lenguaje:** Python
- **Librería:** GitPython para manipulación programática de repositorios
- **Almacenamiento:** Sistema de archivos (local o compatible S3)

3.5.2 Patrones de Diseño

Bridge: Desacopla abstracción de almacenamiento de implementación física (Disco Local vs S3).
Command: Gateway envía órdenes explícitas (Commit, Checkout, Branch) que el motor ejecuta sobre sistema de archivos.

3.5.3 Responsabilidades

Almacenamiento físico y control de versiones de archivos de demostración (.lean). Historial completo de cambios, ramas y fusiones. Separado del Backend que sólo gestiona metadatos.

4 Arquitectura del Frontend

4.1 Modelo Arquitectónico

Arquitectura de componentes en capas basada en Angular. Tres estratos horizontales:
Capa de Presentación: Componentes dumb para renderizado de interfaz y captura de eventos.
Capa de Abstracción: Fachadas y gestión de estado. Controla lógica de interfaz y flujo de datos.
Capa de Core: Servicios y adaptadores. Comunicación con API Gateway y manejo de WebSockets.
Modularización: Feature Modules con lazy loading. Módulos pesados (Editor de Teoremas, Visualizador de Grafos) se cargan bajo demanda, optimizando Time-to-Interactive.

4.2 Stack Tecnológico

- **Framework:** Angular (LTS)
- **Lenguaje:** TypeScript con tipado estricto
- **Estado global:** NgRx (Redux pattern) + Angular Signals para reactividad local
- **Editor de código:** Monaco Editor con soporte LSP para autocompletado
- **Visualización de grafos:** Cytoscape.js o D3.js para manipulación interactiva de nodos
- **Comunicación reactiva:** RxJS para flujos asíncronos de WebSockets y eventos

4.3 Patrones de Diseño

4.3.1 Facade (Fachada)

Servicios de fachada (ProjectFacade, EditorFacade) actúan como interfaz simplificada entre componentes visuales y complejidad del estado.

Justificación: Componentes no conocen origen de datos (REST, WebSocket, caché local). Fachada expone Observables y métodos de acción. Desacopla vista de lógica de negocio, facilita testing y cambios futuros.

4.3.2 Observer & Observable (Paradigma Reactivo)

RxJS implementa patrón Observer para manejar asincronismo inherente del sistema.

Implementación: Servicios de comunicación exponen Subjects/Observables. Fachadas se suscriben para actualizar estado automáticamente. WebSocketService emite eventos de cambio en documento, fachadas consumen y actualizan editor sin polling.

4.3.3 Interceptor (Cadena de Responsabilidad)

HttpInterceptors de Angular para comunicación HTTP limpia.

Cadena de interceptores:

1. Inyecta token JWT en headers de cada petición
2. Maneja errores globales (401 → login, 500 → notificación)
3. Logging y métricas

Centraliza lógica transversal, evita repetición de código de manejo de errores.

4.3.4 State / Redux (Gestión de Estado)

NgRx Store implementa flujo de datos unidireccional para estado complejo del editor colaborativo.

Flujo:

1. Acciones de usuario se despachan como Actions
2. Reducers puros transforman estado actual + acción → nuevo estado inmutable
3. Effects interceptan acciones para efectos secundarios (llamadas Backend, WebSocket)
4. Estado predecible, depurable (time-travel debugging) y consistente

4.3.5 Adapter (Adaptador)

Transforma DTOs del API en modelos de dominio ricos del frontend.

Ejemplo: Array plano de nodos/aristas del API → estructura de objetos grafo navegable por librería de visualización.

4.3.6 Singleton (Instancia Única)

Servicios Angular (@Injectable({providedIn: 'root'})) son Singletons por diseño.

Aplicación crítica: WebSocketManagerService debe ser instancia única compartida. Asegura una sola conexión con servidor, gestionando heartbeats, reconexiones automáticas y distribución de mensajes.

4.4 Estrategia de Comunicación con Backend

Carga inicial y navegación: Peticiones HTTP estándar con HttpClient y Interceptors para seguridad.

Sesión de trabajo activa: Modelo orientado a eventos sobre WebSockets.

Optimistic UI: Usuario realiza acción, aplicación actualiza interfaz localmente de inmediato. Mensaje se envía a backend en segundo plano. Si hay error asíncrono, aplicación ejecuta rollback o compensación. Elimina percepción de latencia de red, proporciona fluidez en edición.

5 Consolidación de Principios de Diseño

5.1 Patrones de Gang of Four Aplicados

- **Strategy:** Selección dinámica de proveedores IA y modelos
- **Observer:** Notificaciones en tiempo real vía WebSockets y RxJS
- **Adapter:** Transformación de salidas nativas a formatos estandarizados
- **Facade:** Interfaces simplificadas para subsistemas complejos

- **Repository:** Abstracción de acceso a datos relacionales
- **Command:** Encapsulación de operaciones como objetos ejecutables
- **Singleton:** Gestión centralizada de conexiones y recursos compartidos
- **Bridge:** Desacoplamiento de abstracciones de implementaciones físicas
- **Circuit Breaker:** Resiliencia ante fallos de servicios externos

5.2 Principios SOLID

Single Responsibility: Cada microservicio tiene una responsabilidad definida.
Open/Closed: Sistema abierto a extensión (nuevos modelos IA, lenguajes) sin modificar código existente.
Liskov Substitution: Implementaciones de estrategias son intercambiables.
Interface Segregation: Interfaces específicas para cada tipo de cliente.
Dependency Inversion: Dependencia de abstracciones, no de implementaciones concretas.

5.3 Características de Arquitectura de Microservicios

- **Desacoplamiento:** Servicios independientes con contratos bien definidos
- **Escalabilidad independiente:** Recursos asignados según necesidad específica
- **Tolerancia a fallos:** Aislamiento de errores en servicios individuales
- **Deployment independiente:** Actualización de servicios sin afectar sistema completo
- **Diversidad tecnológica:** Stack optimizado por servicio
- **Observabilidad:** Logs, métricas y tracing distribuido

6 Consideraciones de Deployment

6.1 Entorno Local (Desarrollo)

Docker Compose: Orquestación de servicios en máquina de desarrollo.
Volumen: Datos de PostgreSQL y repositorios Git persistentes.
Red interna: Comunicación entre contenedores vía red Docker.

6.2 Entorno Producción (Kubernetes)

Deployment: Manifiestos de Kubernetes para cada microservicio.
Services: Descubrimiento interno y balanceo de carga.
Ingress: Enrutamiento externo con certificados TLS.
StatefulSets: Para PostgreSQL y Redis con almacenamiento persistente.
ConfigMaps/Secrets: Configuración y credenciales.
HPA: Horizontal Pod Autoscaler para escalado automático basado en métricas.
Resource Limits: CPU y memoria limitadas por pod para prevención de noisy neighbor.

6.3 Monitoreo y Observabilidad

Logging: Logs estructurados en JSON agregados vía ELK stack.
Métricas: Prometheus para recolección, Grafana para visualización.
Tracing: OpenTelemetry con Jaeger para trazabilidad de peticiones distribuidas.
Health Checks: Liveness y readiness probes para auto-reparación.
Alertas: Notificaciones automáticas basadas en umbrales de métricas críticas.

7 Seguridad

7.1 Autenticación y Autorización

JWT Tokens: Autenticación stateless con expiración configurable.
RBAC: Control de acceso basado en roles (Admin, Owner, Collaborator, Viewer).
OAuth: Integración con GitHub para autenticación federada.

7.2 Comunicación Segura

TLS/HTTPS: Todas las comunicaciones externas cifradas.
WSS: WebSockets seguros para edición colaborativa.
CORS: Configuración restrictiva permitiendo sólo orígenes autorizados.
Rate Limiting: Protección contra abuso de APIs.

7.3 Aislamiento de Ejecución

Contenedores: Procesos aislados en entornos contenedorizados.
Resource Limits: Previene DoS por consumo excesivo de recursos.
Network Policies: Comunicación restringida entre pods en Kubernetes.
Secrets Management: Credenciales almacenadas en Kubernetes Secrets o Vault.