Documento de Arquitectura para la Kata: Revisa Tu Tarea

Valeria Rudas Ruiz

Mariana Cruz Gomez

Lilian Estefanía Maradiago Correa

Materia: Arquitectura de software

Profesor: Julio Robles



Universidad San Buenaventura

Facultad de Ingeniería

Programa Académico Ingeniería de Sistemas

Santiago de Cali, Colombia

2025

Contenido

[1. Contexto 2](#_Toc1822740451)

[2. Identificar los Interesados 3](#_Toc48555088)

[3. Requerimientos Funcionales 3](#_Toc1463296772)

[4. Requerimientos No Funcionales 4](#_Toc1078732578)

[5. Atributos de Calidad 5](#_Toc280334889)

[6. Interesados VS Requerimientos No Funcionales (Ponderación) 5](#_Toc1096947258)

[7. Los Atributos de Calidad más Importantes (Drivers Arquitectónicos) MAX=5 6](#_Toc745439211)

[8. Identificar las tácticas para atacar los atributos de calidad 6](#_Toc507190373)

[9. Los Patrones Arquitectónicos que se van a Utilizar 7](#_Toc2008660084)

[9.1 Estado de colas (State Queues) 8](#_Toc1693541875)

[9.2 Tabla de patrones y tácticas aplicables al proyecto 9](#_Toc1134221857)

[10. Estrategia arquitectónica basada en tácticas y patrones seleccionados 9](#_Toc1957582481)

[11. Generar las Vistas 11](#_Toc1306366215)

[11.1. Escenarios Identificados 11](#_Toc782216483)

[11.2. Vista Lógica 11](#_Toc260577858)

[11.3. Vista de Implementación 11](#_Toc1286943423)

[11.4. Vista de Procesos 11](#_Toc1597570761)

[11.5. Vista Física 11](#_Toc1619039722)

# Contexto

Kata: Revisa tu tarea

Una universidad que ha ampliado su oferta de cursos en ciencias de la computación busca automatizar la calificación de tareas de programación sencillas para atender a más de 300 estudiantes por año. El sistema debe permitir la subida, ejecución y calificación de código fuente de forma persistente y auditable. Además, debe incluir detección de plagio mediante comparación entre entregas y el servicio externo de TurnItIn, integrarse con el sistema de gestión del aprendizaje (LMS) de la universidad, y respetar fechas límite definidas por los profesores. Todo esto debe desarrollarse en un contexto con bajo presupuesto para el área de sistemas y con un LMS basado en un mainframe difícil de modificar.

# Identificar los Interesados

* **Estudiantes**

Son los usuarios principales del sistema. Suben sus tareas de programación, reciben calificaciones automáticas y pueden enviar múltiples intentos antes de la fecha límite.

* **Profesores**

Definen las tareas, criterios de evaluación y fechas de entrega. Usan el sistema para revisar los resultados, configurar pruebas automáticas y detectar plagio.

* **Personal administrativo**

Se encarga de la auditoría de notas y reportes académicos. Necesita acceso a los historiales de calificaciones y entregas para cumplir con regulaciones estatales.

* **Equipo TI**

Encargado del desarrollo, mantenimiento e integración del sistema con la infraestructura tecnológica existente, incluyendo el LMS basado en mainframe.

* **Entidades estatales de auditoría**

Revisan anualmente las calificaciones y procesos académicos. No usan directamente el sistema, pero dependen de que este sea auditable y confiable.

# Requerimientos Funcionales

Esta tabla recoge las funcionalidades esenciales que el sistema debe ofrecer desde la perspectiva del usuario. Se detallan las acciones principales que el sistema debe permitir, como subir tareas, ejecutar código, calificar automáticamente, consultar resultados y generar reportes. Cada requisito se estructura para garantizar trazabilidad y alineación con los objetivos del sistema.

|  |  |  |  |
| --- | --- | --- | --- |
| Código | Título | Detalle | Interesados |
| RF01 | Subida de código | El estudiante debe poder subir su código fuente para una tarea específica. | Estudiantes |
| RF02 | Ejecución del código | El sistema debe ejecutar el código en un entorno controlado y mostrar los resultados. | Estudiantes, Profesores |
| RF03 | Calificación automática | El sistema debe calificar el código según criterios definidos (pruebas, métricas). | Profesores, Estudiantes |
| RF04 | Historial de envíos | Cada envío debe guardarse con fecha, hora, código enviado y resultado. | Estudiantes, Profesores, Administrativos |
| RF05 | Múltiples intentos | El estudiante puede enviar varias versiones de su tarea antes de la fecha límite. | Estudiantes |
| RF06 | Gestión de fechas límite | El profesor puede definir una fecha y hora límite para cada tarea. | Profesores |
| RF07 | Rechazo de envíos fuera de plazo | El sistema debe rechazar entregas después de la fecha límite. | Estudiantes, Profesores |
| RF08 | Comparación entre entregas (plagio) | El sistema debe comparar entregas entre estudiantes para detectar similitudes. | Profesores, Administrativos |
| RF09 | Integración con TurnItIn | El sistema debe conectarse con TurnItIn para verificar plagio externo. | Profesores |
| RF10 | Integración con LMS | El sistema debe enviar calificaciones y datos al LMS de la universidad. | Profesores, Administrativos, Sistemas |
| RF11 | Configuración de criterios de nota | El profesor puede definir pruebas o métricas como parte de la calificación. | Profesores |
| RF12 | Consulta de notas | Los estudiantes y profesores deben poder ver las notas y su historial. | Estudiantes, Profesores |
| RF13 | Generación de reportes auditables | El sistema debe generar reportes que permitan auditoría anual de notas y entregas. | Administrativos, Entidades estatales |

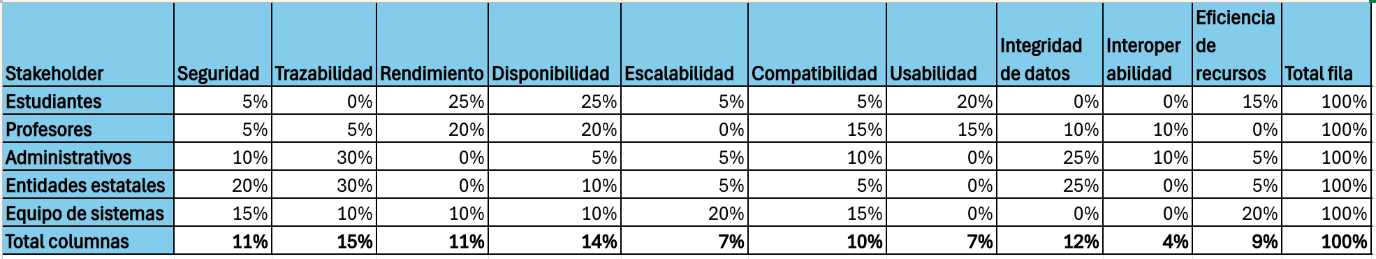
# Requerimientos No Funcionales

Esta tabla define las condiciones de calidad bajo las cuales deben operar las funcionalidades del sistema. Incluye aspectos como seguridad, rendimiento, disponibilidad, escalabilidad e interoperabilidad. Cada requisito no funcional se describe con su métrica de evaluación y los interesados afectados, estableciendo una base sólida para decisiones arquitectónicas.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **DESCRIPCIÓN** | **DETALLE** | **MÉTRICA** | **INTERESADOS** |
| RNF01 | Seguridad de ejecución | El código enviado por estudiantes debe ejecutarse en un entorno aislado (sandbox). | Ninguna ejecución debe afectar el servidor o acceder a archivos externos. | Equipo de sistemas, Estudiantes |
| RNF02 | Trazabilidad | Cada acción (envío, calificación, modificación) debe registrarse con fecha y usuario. | 100 % de acciones relevantes registradas en logs. | Administrativos, Entidades estatales |
| RNF03 | Rendimiento del sistema | El sistema debe procesar y calificar entregas de forma eficiente, incluso en picos de uso. | Tiempo de respuesta promedio < 5 s por envío. | Estudiantes, Profesores |
| RNF04 | Alta disponibilidad | El sistema debe estar disponible durante los periodos de entrega de tareas. | Uptime ≥ 99 % durante semanas de entrega. | Estudiantes, Profesores |
| RNF05 | Escalabilidad | El sistema debe soportar el crecimiento anual de estudiantes sin pérdida de rendimiento. | Soporte de al menos 500 usuarios concurrentes sin fallos. | Equipo de sistemas |
| RNF06 | Compatibilidad con LMS | La integración con el LMS debe funcionar sin necesidad de modificar el sistema heredado. | 100 % de datos sincronizados sin intervención manual. | Equipo de sistemas, Administrativos |
| RNF07 | Usabilidad | La interfaz debe ser simple y clara para usuarios no técnicos. | ≥ 85 % de satisfacción en pruebas de usabilidad. | Estudiantes, Profesores |
| RNF08 | Integridad de datos | Ninguna entrega ni calificación debe perderse o alterarse sin trazabilidad. | 0 % de pérdida o modificación no autorizada de datos. | Administrativos, Entidades estatales |
| RNF09 | Interoperabilidad externa | El sistema debe comunicarse correctamente con TurnItIn y otros servicios externos. | 100 % de envíos compatibles y respuestas válidas de servicios externos. | Profesores, Administrativos, Equipo de sistemas |
| RNF10 | Bajo consumo de recursos | El sistema debe operar con recursos limitados del área de TI. | CPU y memoria dentro del 70 % de uso promedio en servidores. | Equipo de sistemas |

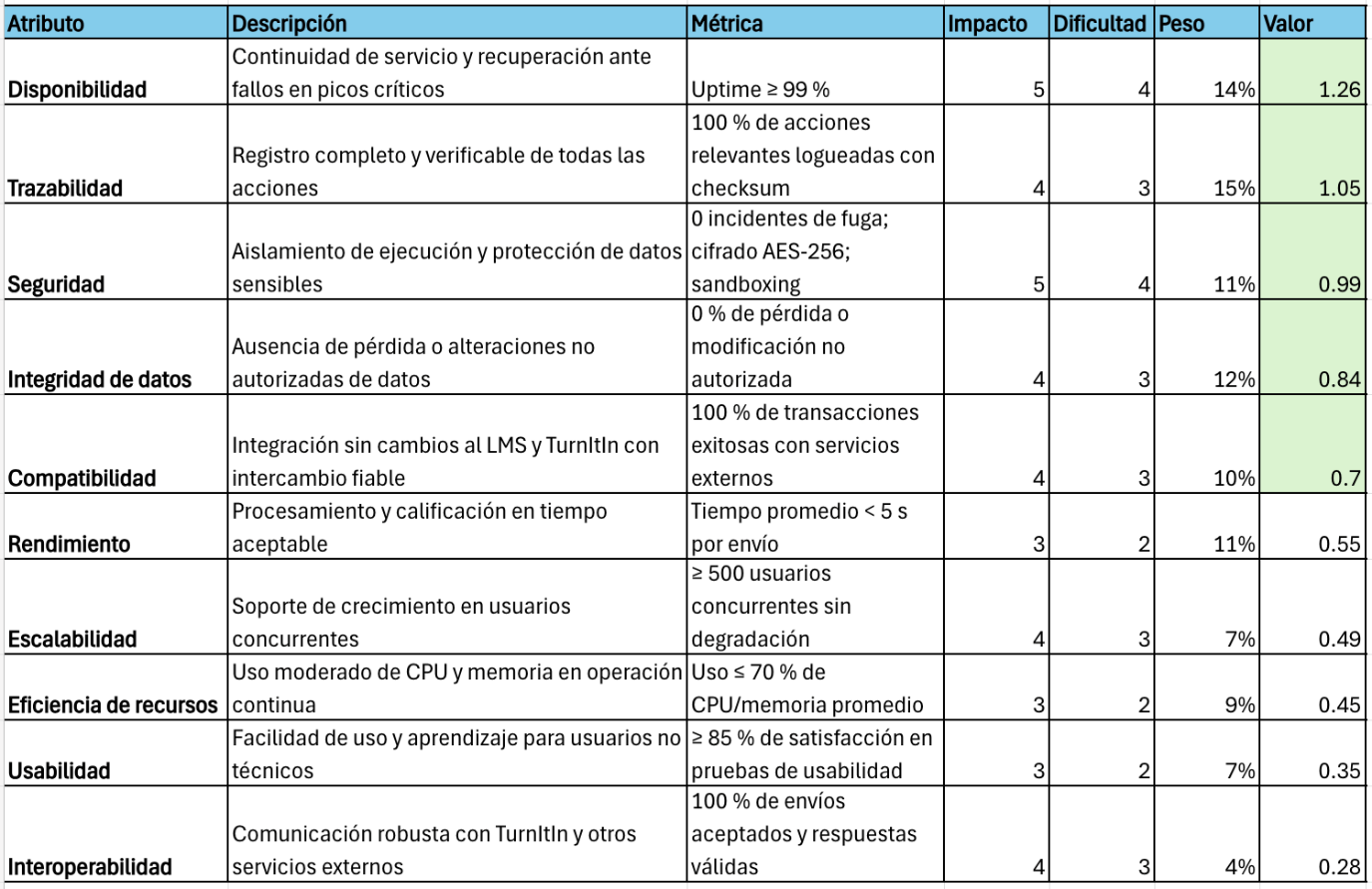
# Atributos de Calidad

En esta tabla se relacionan los diferentes atributos de calidad definidos por la norma ISO 25010 con los distintos grupos de interesados del sistema (estudiantes, profesores, administrativos, etc.). Permite visualizar qué atributos son más relevantes para cada parte interesada, facilitando una priorización informada en fases posteriores.



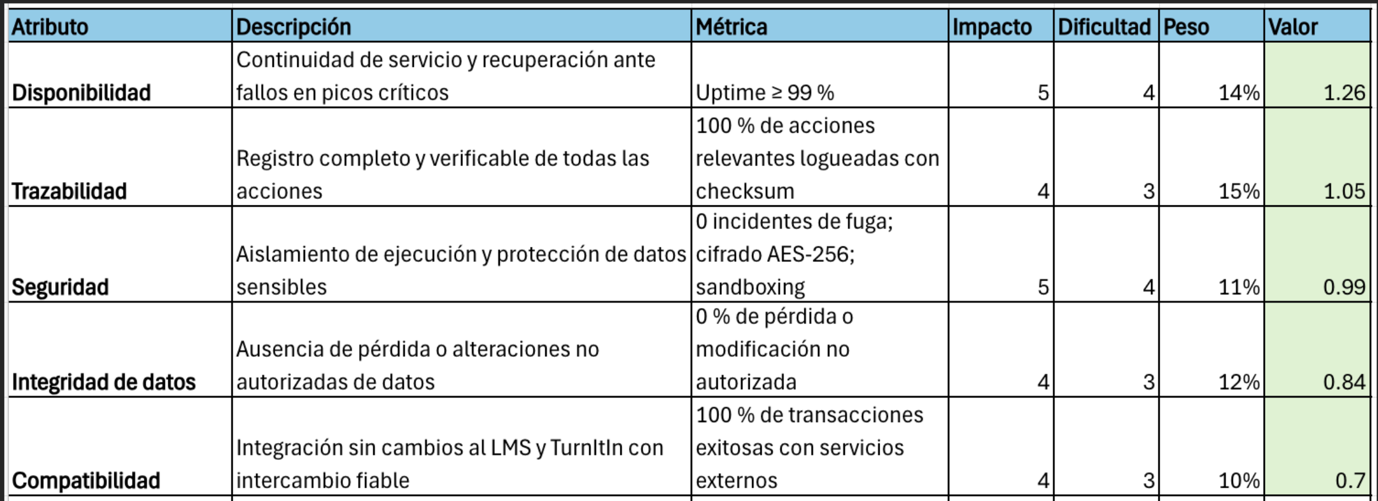
# Interesados VS Requerimientos No Funcionales (Ponderación)

Esta tabla extiende la anterior asignando un peso porcentual a cada atributo según su importancia relativa para cada interesado. Las filas suman 100 %, reflejando la prioridad interna de cada stakeholder, y la fila de totales permite identificar cuáles atributos son prioritarios a nivel general del sistema. Esta información es clave para definir los drivers arquitectónicos.



# Los Atributos de Calidad más Importantes (Drivers Arquitectónicos) MAX=5

Aquí se presentan los cinco atributos de calidad con mayor ponderación, que se convierten en los drivers arquitectónicos del sistema. Estos atributos (por ejemplo, disponibilidad, trazabilidad, seguridad) representan los ejes críticos de diseño, ya que impactan directamente en la experiencia del usuario, la viabilidad técnica y el cumplimiento institucional.



# Identificar las tácticas para atacar los atributos de calidad

En esta sección se selecciona **una táctica específica por cada atributo de calidad**, considerando el contexto del sistema: ejecución segura de código estudiantil, integraciones externas, auditoría, restricciones presupuestarias y escalabilidad. La táctica seleccionada por atributo representa la mejor opción respecto a alternativas, por su capacidad de respuesta directa, eficiencia y coherencia con la arquitectura general del sistema.

|  |  |  |
| --- | --- | --- |
| **Atributo** | **Táctica seleccionada** | **Justificación** |
| **Disponibilidad** | *Failover automático + redundancia de base de datos* | Asegura continuidad del servicio en semanas pico de entregas. Mejora sobre el simple monitoreo. |
| **Trazabilidad** | *Logging estructurado + checksum (hash SHA-256)* | Proporciona trazabilidad verificable para entidades estatales. Mejor que logs planos o audit tables. |
| **Seguridad** | *Sandboxing + RBAC + cifrado TLS/AES-256* | Aísla código ejecutado y protege los datos. Más robusto que filtros de entrada o WAF básicos. |
| **Integridad de datos** | *Control de versiones + validación cruzada* | Asegura que no se pierdan o manipulen datos; permite auditoría post mortem. |
| **Compatibilidad** | *Adaptadores (Adapter Pattern) con formatos legacy* | Permite conexión al LMS sin modificar su estructura. Más viable que reemplazar o replicar su lógica. |
| **Rendimiento** | *Ejecución asíncrona + caché temporal* | Mejora la velocidad en procesamiento masivo, especialmente durante picos de carga. |
| **Escalabilidad** | *Microservicios desacoplados por colas de mensajes* | Permite crecimiento modular y paralelo. Preferible a una arquitectura monolítica escalada verticalmente. |
| **Eficiencia de recursos** | *Autoscaling + límites de recursos por contenedor* | Controla el consumo en ambientes limitados. Mejora frente a entornos sin aislamiento ni monitoreo. |
| **Usabilidad** | *Diseño centrado en el usuario (UX) + validación proactiva* | Asegura accesibilidad sin capacitación técnica. Más eficaz que manuales o soporte pasivo. |
| **Interoperabilidad** | *Intercambio basado en API REST + colas resilientes (RabbitMQ)* | Asegura comunicación robusta y desacoplada con TurnItIn. Mejora sobre polling o cron. |

# Los Patrones Arquitectónicos que se van a Utilizar

Patrón principal: **Arquitectura orientada a eventos con microservicios (Event-Driven Microservices)**

**Descripción:**  
 Este patrón se basa en dividir el sistema en servicios independientes que se comunican entre sí a través de **eventos asíncronos** (por ejemplo, mediante colas de mensajes). Cada microservicio escucha ciertos eventos, realiza una tarea específica y publica nuevos eventos como respuesta.   
 **Ideal para:** sistemas distribuidos que requieren **desacoplamiento, escalabilidad y resiliencia**.

**Justificación:**

Este patrón es el más adecuado para el sistema por las siguientes razones:

* **Aislamiento funcional**: permite que componentes como el calificador, el sandbox, la auditoría o la integración LMS funcionen como servicios independientes.
* **Escalabilidad y resiliencia**: los servicios pueden escalarse horizontalmente de forma independiente. Por ejemplo, en fechas de entrega, se puede escalar solo el evaluador automático.
* **Integración externa confiable**: mediante colas de mensajes (RabbitMQ), se desacopla la interacción con sistemas como TurnItIn, mejorando la tolerancia a fallos.
* **Observabilidad y trazabilidad**: la arquitectura event-driven facilita el monitoreo de flujos de estado (entregado, evaluado, auditado, sincronizado...).

**Alternativas descartadas:**

* **Monolito por capas**: menos costoso pero rígido, difícil de escalar por partes o responder a cambios modulares.
* **Microkernel**: útil para plugin de ejecución, pero no para toda la orquestación.
* **ESB (Enterprise Service Bus)**: potente pero innecesariamente complejo y costoso para un proyecto con restricciones presupuestarias.

## Estado de colas (State Queues)

Para gestionar el procesamiento de envíos de código y coordinación entre componentes (sandbox, calificador automático, TurnItIn, sincronización LMS), proponemos un mecanismo de colas de estados:

1. **Definición de estados**
   1. RECIBIDO: el envío acaba de llegar y está en cola.
   2. EN\_SANDBOX: en proceso de aislamiento y ejecución.
   3. EN\_CALIFICACIÓN: ejecución de pruebas y métricas.
   4. EN\_PLAGIO: comparación interna y en TurnItIn.
   5. SINCRONIZANDO: envío de resultados al LMS.
   6. COMPLETADO o ERROR: fin del workflow.
2. **Implementación**
   1. Usar RabbitMQ (o similar) con colas dedicadas por estado.
   2. Cada microservicio “consumer” toma mensajes de la cola de su estado asignado, procesa y publica el mensaje en la siguiente cola de estado.
   3. Permite escalabilidad horizontal (más consumidores para picos), retry automático, circuit-breaker y monitoring de cada etapa.
3. **Beneficios**
   1. Visibilidad clara de dónde está cada envío.
   2. Gestión tolerante a fallos: un mensaje que no se procesa se reintenta o va a “dead letter queue”.
   3. Facilita métricas por etapa (tiempos, cuellos de botella).

## Tabla de patrones y tácticas aplicables al proyecto

Esta tabla resume los patrones arquitectónicos y tácticas seleccionadas para abordar los atributos de calidad priorizados. Se explican las ventajas específicas en el contexto del sistema y las razones por las cuales se descartaron otras opciones posibles. Esto permite justificar las decisiones arquitectónicas desde una perspectiva técnica y estratégica.

|  |  |  |
| --- | --- | --- |
| **Patrón / Táctica** | **Ventajas** | **Desventajas** |
| **Event-Driven Microservices** | - Escalabilidad fina por componente- Desacoplamiento fuerte- Resiliencia y retry integrado | - Complejidad operativa (monitoring, despliegue)- Requiere cultura DevOps madura |
| **Circuit Breaker** | - Previene cascadas de fallos- Mejora tolerancia de integraciones externas | - Configuración de umbrales puede ser delicada- Introduce latencia extra en checks |
| **Adapter Pattern** | - Integración sin tocar sistemas legacy- Bajo coste de desarrollo- Código claro y mantenible | - Cada nueva versión externa requiere actualización de adaptador |
| **Sandboxing con Contenedores** | - Aislamiento completo de ejecución- Control granular de recursos- Seguridad en tiempo de ejecución | - Overhead de instanciar contenedores- Latencia inicial en «spin-up» |
| **Logging estructurado + Hash** | - Búsqueda y análisis de logs más sencillo- Garantiza integridad de registros | - Aumenta tamaño de logs- Requiere infraestructura de almacenamiento y visualización |
| **Caching de resultados** | - Reduce uso de CPU/memoria en repeticiones- Mejora tiempos de respuesta | - Consistencia de cache- Posible complejidad invalidación |

# Estrategia arquitectónica basada en tácticas y patrones seleccionados

La arquitectura del sistema se define a partir de una **estrategia integral** que combina tácticas específicas y patrones arquitectónicos, alineados con los **drivers arquitectónicos priorizados**, conforme a la ponderación realizada. A continuación, se describe cómo se implementa dicha estrategia.

**Driver 1: Disponibilidad**

**Táctica**: *Failover automático + redundancia*

**Patrón**: *Event-Driven Microservices*

**Aplicación en el sistema**:

* + Los servicios críticos como sandbox, calificador automático y sincronizador LMS están distribuidos en microservicios independientes.
  + Cada uno puede desplegarse en nodos redundantes y balancearse dinámicamente.
  + El uso de colas garantiza que ningún mensaje se pierda si un servicio falla; al reanudarse, el servicio puede continuar desde donde se detuvo.

**Driver 2: Trazabilidad**

**Táctica**: *Logging estructurado + checksum SHA-256*

**Patrón**: *Adapter Pattern*

**Aplicación en el sistema**:

* + Todos los eventos importantes (envíos, resultados, decisiones del sistema) se registran en logs JSON con hash de verificación.
  + El adaptador LMS también registra cada sincronización con identificación única.
  + Esto permite generar trazabilidad completa para auditores y entes reguladores.

**Driver 3: Seguridad**

**Táctica**: *Sandboxing + control de acceso (RBAC) + cifrado*

**Patrón**: *Microkernel Pattern (con plugin de ejecución)*

**Aplicación en el sistema**:

* + El código del estudiante se ejecuta en contenedores efímeros aislados (Docker), protegidos del sistema anfitrión.
  + El microkernel orquesta la ejecución como un “plugin”, garantizando que no se ejecute código malicioso directamente.
  + Las credenciales y archivos se manejan con cifrado y control de acceso por rol.

**Driver 4: Integridad de datos**

**Táctica**: *Control de versiones + validación cruzada*

**Patrón**: *Event-Driven Microservices*

**Aplicación en el sistema**:

* + Cada evento de evaluación o sincronización LMS se almacena con una versión y marca de tiempo.
  + Si ocurre un error o pérdida, el sistema puede comparar estados pasados y recuperar la información más confiable.
  + Las colas aseguran que no haya sobrescritura fuera de orden.

**Driver 5: Compatibilidad**

**Táctica**: *Adaptador de integración (API Adapter)*

**Patrón**: *Adapter Pattern*

**Aplicación en el sistema**:

* + El sistema se conecta al LMS y a TurnItIn mediante adaptadores dedicados que traducen los formatos heredados a un modelo interno moderno.
  + Esto evita modificar el sistema legacy, reduciendo riesgos y permitiendo evolución futura del backend sin romper la interoperabilidad.

**Síntesis estratégica**

La arquitectura se estructura bajo el **patrón principal de Event-Driven Microservices**, complementado con **adaptadores** para compatibilidad y **microkernel** para ejecución segura. Cada uno de los cinco principales atributos de calidad se refuerza con **tácticas específicas que priorizan la resiliencia, trazabilidad, seguridad, confiabilidad e interoperabilidad** del sistema.

Esta estrategia asegura que el sistema no solo sea funcional, sino también **sostenible, auditable, seguro y escalable** en el contexto educativo y regulatorio que lo rodea.

# Generar las Vistas

Los diagramas presentados a continuación están disponibles en el siguiente enlace: <https://drive.google.com/file/d/1-pc4hQOwrsD3WC5pLYZY5oVhVPrAOtKL/view?usp=sharing>

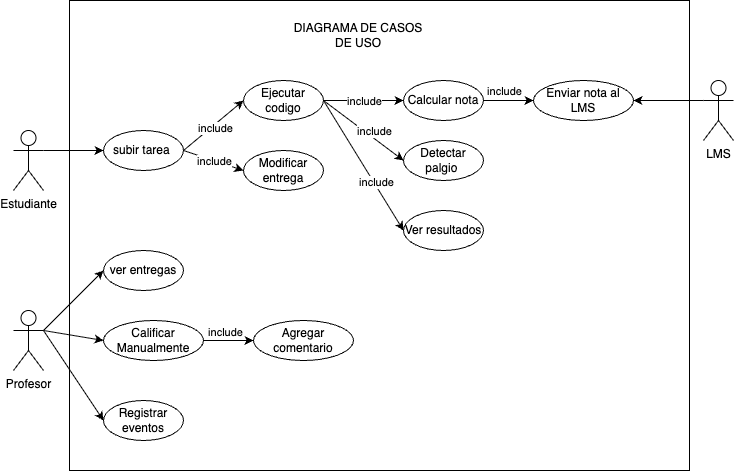
## Escenarios Identificados

**Descripción:** Esta vista captura las interacciones clave entre los actores principales (estudiantes, profesores) y el sistema, representadas como casos de uso. Muestra funcionalidades esenciales como la subida de tareas, ejecución de código, detección de plagio y sincronización con el LMS, junto con acciones secundarias como modificaciones de entregas y calificación manual. Las relaciones de inclusión ("include") destacan flujos compuestos, como la ejecución de código dentro del proceso de subida de tareas. Esta vista alinea directamente con los requerimientos funcionales del documento (RF01-RF13), ilustrando cómo los usuarios finales operarán el sistema y cómo este responde a sus acciones, proporcionando una comprensión clara de las capacidades principales desde una perspectiva de usuario.

* Estudiantes: Subir tarea, modificar entrega, ver resultados.
* Profesores: Calificar manualmente, agregar comentarios, detectar plagio.
* Sistema: Ejecutar código, calcular nota, enviar notas al LMS.

**Elementos Clave:**

* Actores: Estudiante, Profesor, LMS.
* Relaciones: include para acciones compuestas (ej: "Ejecutar código" es parte de "Subir tarea").
* Flujo básico: Muestra las funcionalidades esenciales (RF01-RF13 del documento).



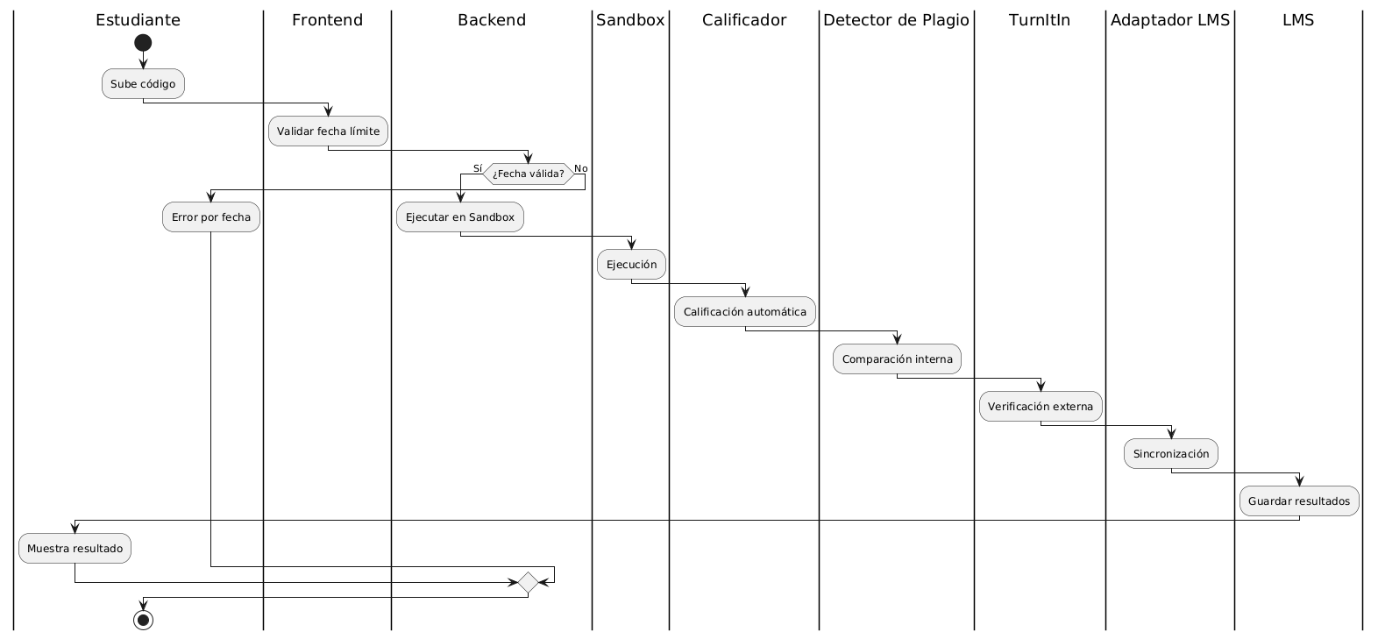
## Vista Lógica

**Descripción:** El diagrama describe el flujo lógico que sigue una tarea entregada por un estudiante, desde el momento en que se sube el código hasta que se obtiene el resultado final, pasando por todas las etapas de validación, ejecución, evaluación y almacenamiento de resultados.

* Estudiante sube código → Validación de fecha.
* Ejecución en Sandbox → Calificación → Detección de plagio (TurnItIn).
* Sincronización con LMS → Guardado de resultados.

**Elementos Clave:**

* Estados: Transiciones claras
* Servicios involucrados: Sandbox, TurnItIn, LMS.



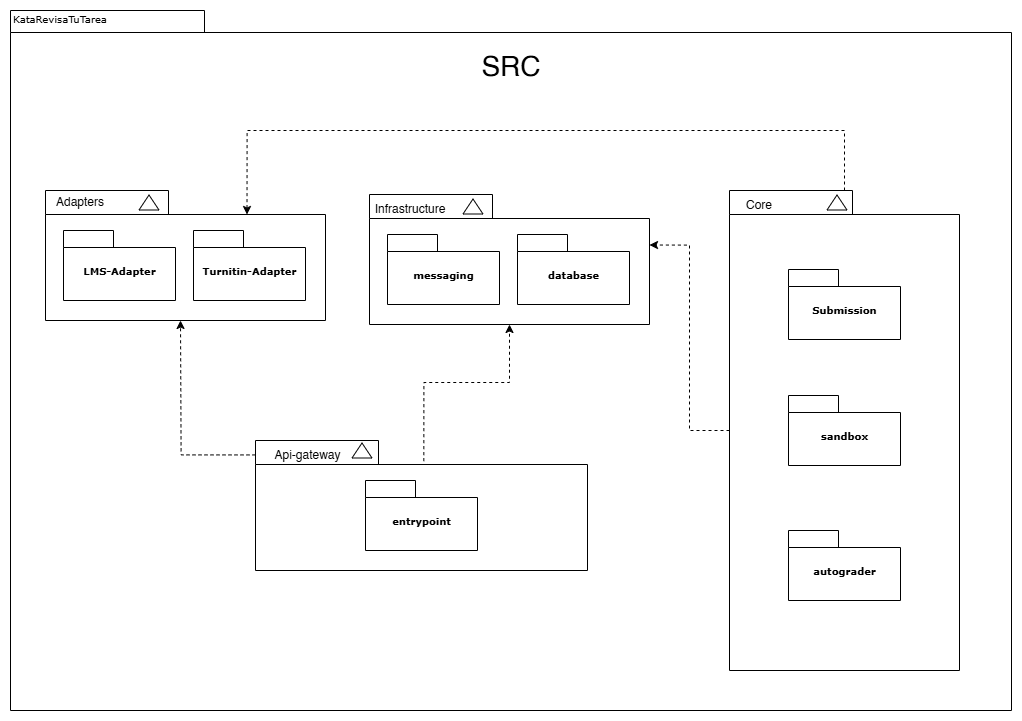
## Vista de Implementación

**Descripción:** Este diagrama presenta la organización modular del sistema, agrupando los componentes de software en paquetes según sus responsabilidades funcionales y de implementación. La estructura modular facilita la mantenibilidad, escalabilidad y reutilización del sistema.

* Adapters: Conexión con sistemas externos (LMS, TurnItIn).
* Core: Lógica de negocio (submission, sandbox, autograder).
* Infrastructure: Servicios transversales (mensajería, base de datos).
* App-gateway: Punto único de entrada.

**Elementos Clave:**

* Modularidad: Separación clara entre paquetes.
* Patrones: Uso de Adapter Pattern (para integraciones) y Microservicios.
* Relación con Documento Original:
* Refleja la arquitectura event-driven (p. 8) y la vista lógica (sección 11.2).



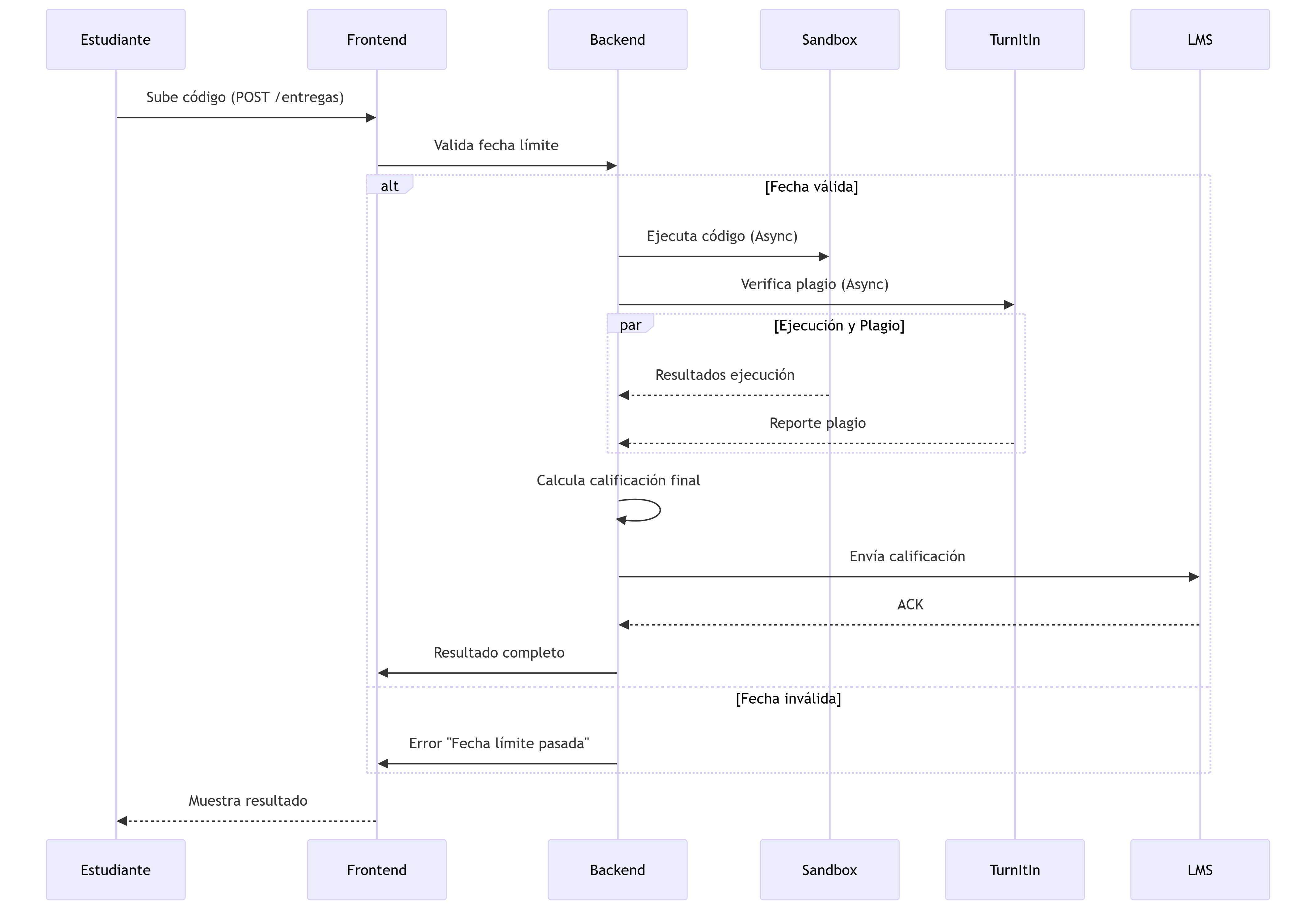
## Vista de Procesos

**Descripción:** Este diagrama de secuencia representa el flujo dinámico de mensajes e interacciones entre los principales componentes y actores involucrados en el proceso de envío y evaluación de una tarea desde la perspectiva del sistema.

* Estudiante → Frontend → API Gateway → Servicio Envíos.
* Procesamiento en Sandbox → Calificador → TurnItIn → LMS.
* Almacenamiento en base de datos.
* Concurrencia: Uso de colas (SQS) para comunicación asíncrona.

**Elementos Clave:**

* Temporalidad: Orden exacto de interacciones.
* Tolerancia a fallos: Reintentos y dead-letter queues implícitos.



## Vista Física

**Descripción:** ste diagrama representa la arquitectura física del sistema desplegado en la nube, utilizando los servicios de AWS para garantizar alta disponibilidad, escalabilidad y seguridad. Se describen los componentes físicos y virtuales que soportan la aplicación, así como la comunicación y flujo de datos entre ellos.

**Componentes:**

* Frontend: CloudFront + S3 (hosting estático).
* Backend: API Gateway + Lambda (procesamiento).
* Datos: RDS (PostgreSQL), DynamoDB (NoSQL), SQS (colas).
* Seguridad: Secrets Manager, CloudTrail (logs).
* Integraciones: TurnItIn y LMS Legacy.

**Elementos Clave:**

* Escalabilidad: Uso de Lambda y auto-scaling.
* Conectividad: API REST entre servicios.

