PONTIFICIA UNIVERSIDAD JAVERIANA

Arquitectura de Software
Taller presentación 1
Colas y Eventos

Grupo 7

Juan Diego Romero
Pablo Quintero
Kamilt Bejarano

Bogotá, Colombia
6/10/2025

Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025

Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025

Introducción General

Este informe presenta una investigación sobre dos arquitecturas fundamentales en los sistemas distribuidos modernos: la Arquitectura Dirigida por Eventos (Event-Driven Architecture, EDA) y la Arquitectura de Colas de Mensajes (Message Queue Architecture). Destacando su aplicación en un stack tecnológico que integra Astro (frontend), Ruby on Rails (backend), Kafka (middleware) y Cassandra (base de datos). El propósito es comprender cómo estos enfoques y tecnologías se complementan para construir sistemas escalables, desacoplados y resilientes.

1. Arquitectura Dirigida por Eventos (Event-Driven Architecture - EDA)

Definición

La Arquitectura Dirigida por Eventos (Event-Driven Architecture, EDA) es un modelo de diseño en el que los componentes del sistema se comunican y responden mediante la generación, transmisión y consumo de eventos. Un evento es un hecho significativo en el sistema, como la creación de un pedido o el inicio de sesión de un usuario. EDA promueve la asincronía y el desacoplamiento, permitiendo que los servicios reaccionen en tiempo real sin depender directamente unos de otros.

Características

- Comunicación asíncrona entre componentes.
- Desacoplamiento total entre productores y consumidores.
- Procesamiento en tiempo real mediante buses o brokers de eventos.
- Soporte para múltiples patrones: Pub/Sub, CQRS, Event Sourcing.
- Escalabilidad horizontal y tolerancia a fallos.
- Consistencia eventual y resiliencia ante errores.

Historia y evolución

La EDA surge a partir de la evolución de la Arquitectura Orientada a Servicios (SOA) y de los sistemas de mensajería empresarial (MOM). Durante los años 2000, empresas como LinkedIn y Amazon impulsaron este paradigma con la creación de plataformas como Apache Kafka y AWS SNS, capaces de manejar millones de eventos por segundo. En la actualidad, EDA es una de las bases de la arquitectura de microservicios, los sistemas IoT, y la analítica en tiempo real.

Tipos de Arquitectura EDA

Tipo	Descripción

Simple (1:1)	Un productor genera un evento consumido por un único servicio.
Publish/Subscribe (1:N)	Un productor publica un evento al que pueden reaccionar múltiples consumidores.
Event Streaming	Los eventos se procesan como un flujo continuo en tiempo real.
Event Sourcing	El estado del sistema se reconstruye a partir de la secuencia histórica de eventos.
CQRS	Se separan las operaciones de lectura y escritura para optimizar rendimiento y escalabilidad.

Ventajas y desventajas

Ventajas	Desventajas
Alta escalabilidad y flexibilidad.	Mayor complejidad para monitoreo y depuración.
Procesamiento asíncrono y en tiempo real.	Consistencia eventual en lugar de inmediata.
Desacoplamiento que facilita el mantenimiento.	Curva de aprendizaje alta.
Resiliencia ante fallos parciales.	Dependencia en infraestructura especializada.

Casos de uso

- Procesamiento de pagos y operaciones financieras en tiempo real.
- Sistemas IoT y monitoreo de sensores distribuidos.
- Analítica en tiempo real de comportamiento de usuarios.
- Plataformas de notificaciones y eventos en aplicaciones móviles.
- Sistemas distribuidos de microservicios desacoplados.

Casos de aplicación

Empresa	Aplicación

Netflix	Usa Kafka para procesar billones de eventos diarios y generar recomendaciones en tiempo real.
Uber	Coordina eventos de geolocalización, pagos y solicitudes de viajes mediante EDA y Kafka.
Amazon	Emite eventos de compras y actualizaciones de inventario que disparan flujos automatizados.
Spotify	Aplica EDA para sincronización entre dispositivos y análisis de tendencias de escucha.

Relación entre los temas asignados (cómo se integran en un stack)

El stack propuesto combina EDA + Kafka + Cassandra con un backend tradicional (Ruby on Rails) y un frontend orientado a performance (Astro). El frontend (Astro) presenta la interfaz y envía acciones o requests al backend (Rails) o publica eventos directamente. Rails puede actuar como productor de eventos (publicando en Kafka) o consumidor, mientras Kafka funge como bus de eventos centralizado. Cassandra almacena los resultados o estados derivados de estos eventos, aportando persistencia distribuida y alta disponibilidad. Este flujo representa la aplicación práctica de EDA donde cada capa cumple un rol especializado.

Qué tan común es el stack designado

El stack es relativamente común en empresas que requieren procesamiento en tiempo real y escalabilidad. Kafka y Cassandra son estándares en empresas de datos y streaming. Ruby on Rails aporta rapidez en desarrollo y Astro ofrece rendimiento en el frontend. Aunque no es una combinación masiva, representa una integración viable y eficiente para sistemas event-driven.

Matrices de Análisis

A continuación se presentan las matrices de evaluación relacionadas con los principios SOLID, atributos de calidad, tácticas, patrones y mercado laboral.

Matrices de Análisis – Arquitectura dirigida e eventos

Matriz: Principios SOLID vs Temas

	EDA	Astro	Rails	Cassandra	Kafka

S - Single Responsibility	Alta: componentes por evento con responsabilidad única.	Alta: componentes UI suelen tener responsabilidades claras.	Media-Alta: Rails promueve separation (models/controllers) pero fácil sobrecargar controllers.	Baja-Media: tablas diseñadas por query, responsabilidad más de datos que lógica.	Alta: productores/consumidores con responsabilidades bien definidas.
O - Open/Closed	Media: sistemas extensibles por nuevos consumidores sin cambiar productores.	Alta: interfaces y componentes pueden extenderse sin modificar base.	Media: extendible mediante gems y módulos, aunque cores pueden requerir cambios.	Media: esquema y modelado requieren planificación; extensibilidad por diseño.	Media-Alta: topics nuevos permiten extender sin cambiar productores existentes.
L - Liskov Substitution	Baja: no aplica directamente al nivel de arquitectura.	Media: componentes reutilizables aplican substitución.	Media: principios OOP en Ruby aplican si se diseñan correctamente.	Baja: no orientada a herencia de objetos.	Baja: principio OOP no directamente relevante.
I - Interface Segregation	Alta: consumidores se suscriben sólo a interfaces (eventos) que necesitan.	Alta: componentes pequeños y específicos favorecen este principio.	Media: Rails puede exponer APIs granularmente; depende del diseño.	Baja-Media: las interfaces son de consulta/escritura; granularidad depende del diseño.	Alta: topics y consumers permiten interfaces específicas.
D - Dependency Inversion	Media-Alta: desacoplamiento	Media: dependencias de librerías UI	Media: Rails facilita inyección limitada;	Baja: DB es una dependencia infraestruct	Media: abstracción sobre brokers posible; consumidores no

	mediante eventos favorece invertir dependencias en runtime.	gestionables; buen diseño importa.	requiere patrones para DI.	ural; inversión se realiza vía abstracción.	dependen directamente de productores.

Matriz: Atributos de Calidad vs Temas

	EDA	Astro	Rails	Cassandra	Kafka
Rendimiento	Variable: depende de diseño y latencia de eventos.	Alta: baja carga cliente si se configura bien.	Media: buen rendimiento en cargas moderadas.	Alta en throughput de escritura/lectura distribuida.	Alta throughput y baja latencia en streaming.
Escalabilidad	Alta: diseñado para escalar horizontalmente.	Media: escalado de frontend por CDN/edge.	Media: puede escalar vertical u horizontalmente con cache.	Alta: excelente escalado horizontal.	Alta: particionado y consumer groups escalables.
Disponibilidad	Alta: desacoplamiento y replicación aumentan disponibilidad.	Media: depende hosting y CDNs.	Media-Alta: depende de infra y DB.	Alta: replicación multi-dc.	Alta: replicación y tolerancia a fallos.
Consistencia	Eventual consistency común.	Fuerte para UI local, pero depende de backend.	Fuerte eventual/strong depende de DB.	Eventual/ tunable consistency.	Eventual: mensajes pueden reenviarse; orden garantizada por partición.

Mantenibilidad	Media: más servicios = más complejidad.	Alta: estructura simple facilita mantenimiento.	Alta si se siguen convenciones; puede degradar si no.	Media: operaciones y modelado requieren expertos.	Media: operativa y configuración agregan complejidad.

Matriz: Tácticas vs Temas

	EDA	Astro	Rails	Cassandra	Kafka
Caching	Sí: caches de eventos o materialized views.	Sí: SSR y CDN.	Sí: caches de página y query caching.	Limitado: caches externos suelen usarse (Redis).	Sí: layer de cache en consumidores.
Replicación	Sí: replicación de eventos entre servicios.	N/A	A nivel DB y app.	Sí: replicación multi-dc nativa.	Sí: replicación de logs entre clusters.
Particionado / Sharding	Sí para tópicos/eventos y carga.	N/A	Sí a nivel de base de datos y cargas.	Sí: particionado por key (token).	Sí: particionado por key en topics.
Backpressure	Necesario en flujos intensos.	N/A	Implementable en endpoints.	N/A (depende del consumidor).	Crítico: consumidores deben aplicar backpressure/flow control.
Event Sourcing / CQRS	Naturalmente compatible.	N/A	Compatible: Rails puede implementar patrones.	Usado como storage para proyecciones.	Soporta event sourcing (logs persistentes).

Matriz: Patrones vs Temas

	EDA	Astro	Rails	Cassandra	Kafka

Pub/Sub	Núcleo: comunicación por eventos.	Puede integrar analytics pub/sub.	Puede producir/consumir mensajes.	No aplica como broker, pero almacena estados.	Núcleo: pub/sub distribuido.
MVC	No aplica directamente.	No aplica; frontend component-based.	Sí: Rails es MVC.	No aplica.	No aplica.
CQRS	Compatible: comandos vs consultas por eventos.	N/A	Implementable para separar lectura/escritura.	Usada para proyecciones (read models).	Buena para pipelines CQRS/event sourcing.
Event Sourcing	Compatible: eventos como fuente de verdad.	N/A	Posible mediante librerías.	Puede persistir proyecciones derivadas.	Ideal: logs inmutables permiten event sourcing.
Producer-Consumer	Sí.	N/A.	Sí.	N/A (almacenamiento).	Sí: model básico.

Matriz: Mercado Laboral vs Temas

	EDA	Astro	Rails	Cassandra	Kafka
Demanda (tendencia)	Alta: arquitectos y diseñadores EDA buscados en empresas de datos.	Creciente: adopción emergente en proyectos web orientados a performance.	Media: persistente pero competitiva frente a stacks JS.	Media-Alta: demanda en empresas de datos/telemetría.	Alta: muy demandado en roles de streaming y datos.
Roles típicos	Arquitecto de software,	Frontend engineer,	Backend developer,	DBA NoSQL, Data Engineer,	Data Engineer,

	ingeniero de integración, SRE.	performance engineer.	fullstack developer, API engineer.	Backend engineer.	Streaming Engineer, Platform Engineer.
Nivel de especialización requerido	Alto: comprensión de concurrencia, consistencia y diseño distribuido.	Medio: habilidades JS y SSR/SSG.	Medio: buena praxis Ruby/Rails y patrones web.	Alto: modelado por queries y operación de clusters.	Alto: operación y tuning de clusters, particiones y retention.

Conclusión de la sección EDA

La EDA representa uno de los paradigmas más influyentes en la arquitectura moderna. Permite construir sistemas adaptativos y resilientes, capaces de reaccionar a sucesos en tiempo real. En conjunto con tecnologías como Kafka, Cassandra, Rails y Astro, permite lograr un ecosistema distribuido robusto, desacoplado y preparado para la expansión de datos y servicios.

Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025

2. Arquitectura de Colas de Mensajes (Message Queue Architecture)

1. Definición

La Arquitectura de Colas de Mensajes (Message Queue Architecture) es un estilo arquitectónico que permite la comunicación asíncrona y desacoplada entre componentes de software mediante colas intermedias de mensajes. Cada cola actúa como un buffer confiable donde los productores (emisores) envían mensajes y los consumidores (receptores) los procesan más tarde, asegurando que ninguna tarea se pierda aun cuando un servicio esté inactivo temporalmente. Su objetivo principal es garantizar la entrega confiable, controlar el flujo de trabajo y permitir escalabilidad horizontal.

2. Características

- Comunicación asíncrona y desacoplada.
- Persistencia y orden garantizado de los mensajes (FIFO, Exactly-once, etc.).
- Tolerancia a fallos mediante colas persistentes y reintentos automáticos.
- Balanceo de carga mediante múltiples consumidores (competing consumers).
- Escalabilidad horizontal al agregar consumidores.
- Desacoplamiento temporal entre productor y consumidor.
- Integración con arquitecturas EDA, SOA y microservicios.

3. Historia y evolución

Década	Evolución
1980s–1990s	Surgen los Message-Oriented Middleware (MOM) como IBM MQSeries.
2000s	SOA integra colas con buses empresariales como ActiveMQ y RabbitMQ.
2010s	Aparecen brokers distribuidos como Kafka y Pulsar con soporte de streaming.
2020s	Mensajería integrada en la nube (AWS SQS, Azure Service Bus, Google Pub/Sub).

4. Tipos o variantes de arquitecturas de colas

Existen múltiples variantes de arquitecturas de colas, que se diferencian según la topología, persistencia y forma de entrega:

Tipo	Descripción	Uso típico	Ejemplo

Point-to-Point (P2P)	Cada mensaje es enviado a una cola y consumido por un solo receptor.	Procesamiento de pedidos o tareas.	Amazon SQS, RabbitMQ
Publish/Subscribe	Un productor publica un mensaje y múltiples consumidores lo reciben.	Difusión de eventos.	Kafka, Pulsar
Work Queue	Varios consumidores compiten por mensajes de la misma cola.	Distribución de carga.	Celery, Sidekiq
Request-Reply	El emisor envía un mensaje y espera una respuesta en otra cola.	Comunicación confiable entre servicios.	ActiveMQ, ZeroMQ
Event Streaming	Los mensajes se guardan en un log inmutable y pueden reprocesarse.	Procesamiento en tiempo real.	Kafka, Redpanda
Dead Letter Queue	Cola para mensajes no procesables o fallidos.	Auditoría y recuperación de errores.	RabbitMQ DLQ, AWS SQS DLQ

5. Ventajas y desventajas

Ventajas	Desventajas
Desacopla productores y consumidores.	Complejidad operativa y monitoreo constante.
Alta tolerancia a fallos.	Latencia en el procesamiento.
Escalabilidad horizontal.	Duplicidad si no se controla la entrega.
Entrega confiable incluso ante fallos.	Depuración más compleja.

Control de flujo y reintentos automáticos.	Requiere políticas de limpieza de colas.

6. Casos de uso

- Procesamiento de tareas en segundo plano (emails, reportes, validaciones).
- Integración de microservicios con comunicación asíncrona.
- Sistemas de pedidos, pagos o inventarios.
- Ingestión de datos IoT y telemetría.
- Ejecución de funciones serverless desencadenadas por mensajes.

7. Casos de aplicación (industria)

Empresa	Implementación	Propósito
Netflix	RabbitMQ y Kafka	Orquestar operaciones y notificaciones.
Uber	Kafka y Redis Streams	Coordinar viajes, pagos y ubicación.
Spotify	Google Pub/Sub	Sincronizar listas y usuarios.
Airbnb	RabbitMQ	Procesamiento de reservas y alertas.
Amazon	SQS/SNS	Mensajería entre servicios serverless.

Qué tan común es el uso de colas en la industria

Las arquitecturas basadas en colas de mensajes son una de las prácticas más extendidas en la ingeniería de software moderna.
Casi todos los sistemas distribuidos actuales —incluyendo microservicios, plataformas cloud, soluciones IoT y aplicaciones serverless— emplean colas como mecanismo de comunicación asíncrona y desacoplada.

Empresas tecnológicas de todos los sectores utilizan colas para gestionar grandes volúmenes de datos, tareas en segundo plano y flujos de integración entre servicios.

Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025

Matrices de Análisis – Arquitectura de Colas

1. Matriz de Principios SOLID vs Tema

Principio	Nivel de Aplicación	Justificación
S – Single Responsibility	Alto	Cada cola gestiona un tipo específico de mensaje con una única responsabilidad.
O – Open/Closed	Alto	Se pueden agregar nuevos consumidores sin modificar los productores existentes.
L – Liskov Substitution	Bajo	No aplica directamente a nivel arquitectónico, ya que no hay herencia entre componentes.
I – Interface Segregation	Alto	Cada consumidor se suscribe solo a la cola o mensajes que necesita.
D – Dependency Inversion	Alto	Productores y consumidores dependen de una abstracción (la cola), no entre sí.

2. Matriz de Atributos de Calidad vs Tema

Atributo de Calidad	Nivel	Descripción
Rendimiento	Alto	Permite procesar grandes volúmenes de mensajes en paralelo con baja latencia.
Escalabilidad	Alta	Escala horizontalmente al añadir consumidores o particiones de cola.
Disponibilidad	Alta	Los brokers replicados aseguran continuidad ante fallos.

Consistencia	Eventual	Los mensajes pueden procesarse en tiempos distintos pero garantizan entrega.
Mantenibilidad	Media	Requiere monitoreo constante de colas y configuración adecuada de DLQ.
Seguridad	Media-Alta	Se asegura con autenticación, autorización y cifrado TLS en brokers.

3. Matriz de Tácticas vs Tema

Táctica	Aplicación en Colas de Mensajes
Caching	Uso de buffers en memoria o disco para acelerar el procesamiento.
Replicación	Duplicación de colas o brokers para tolerancia a fallos.
Particionado / Sharding	Distribución de mensajes por clave o tema para balancear carga.
Backpressure	Controla el flujo para evitar que los productores saturen a los consumidores.
Load Leveling	Las colas absorben picos de carga manteniendo el sistema estable.
Dead Letter Queue (DLQ)	Aísla mensajes con errores tras múltiples intentos.
Retry Policies	Definición de reintentos automáticos para errores transitorios.

4. Matriz de Patrones vs Tema

Patrón	Nivel de Aplicación	Descripción
Producer-Consumer	Núcleo	Base del funcionamiento de las colas; productores

		envían y consumidores procesan.
Competing Consumers	Alta	Múltiples consumidores procesan mensajes de una misma cola para balancear carga.
Pub/Sub	Media	Algunos brokers permiten publicar mensajes a múltiples consumidores.
Queue-based Load Leveling	Alta	Las colas estabilizan el flujo entre productores y consumidores.
Retry / Circuit Breaker	Alta	Permite resiliencia ante errores y desconexiones temporales.
Event Sourcing / CQRS	Media	Posible al combinar colas con logs persistentes.

5. Matriz de Mercado Laboral vs Tema

Aspecto	Descripción
Demanda	Alta en empresas que implementan microservicios, cloud y sistemas de mensajería.
Roles	Backend Developer, Integration Engineer, Cloud Architect, Data Engineer.
Nivel de especialización	Medio-Alto; requiere conocimiento de concurrencia y sistemas distribuidos.
Tecnologías destacadas	RabbitMQ, Kafka, ActiveMQ, AWS SQS, Azure Service Bus, Google Pub/Sub.
Tendencia	Creciente, impulsada por el auge de arquitecturas event-driven y serverless.

Salario promedio LATAM	Entre 4 y 8 millones de pesos mensuales, dependiendo del rol y experiencia.

3. Astro

Definición

Astro es un framework para páginas web Open Source basado en JavaScript y escrito en Go y Typescript por Fred K. Schott enfocado en crear páginas orientadas a contenido rápidas

Características

Astro está orientado a contenido, y es considerado el framework para páginas web más ligero y eficiente de usar gracias a la arquitectura creada por este conocido como Astro Islands.

Cuenta con soporte oficial para múltiples frameworks como React, Vue, Svelte, Cloudflare y Tailwind.

Historia y evolución

- 2021: Beta
- 2022: Versión 1.0
- 2023: Versión 2.0 y 3.0
- 2024: Versión 4.0

Ventajas y desventajas

Astro ofrece una gran flexibilidad al igual que ser eficiente y ligero. Ofrece un gran desempeño por medio de renderización de componentes en servidor.

Sin embargo, no es apto para aplicaciones dinámicas o masivas como redes sociales.

Casos de uso

Astro es un excelente framework para páginas web estáticas orientadas a contenido como foros o blogs.

Casos de éxito

Las páginas web de Porsche, el periodico britanico The Guardian y Visa están escritas en Astro.

4. Ruby

Definición

Ruby es un lenguaje de uso general inspirado en Perl, Lisp, BASIC, entre otros. El objetivo principal de su creador es que el programador sea productivo y se divierta. Es muy similar a Python tanto en funcionalidades como su facilidad de aprendizaje.

Características

Ruby soporta múltiples paradigmas de programación, tales como orientado a objetos, funcional y procesal (procedural). Su syntax es similar a Perl, mientras que la semántica es similar a Smalltalk, pero la filosofía detrás de este es muy diferente a la de Python.

Historia y evolución

- 1993: Yukihiro Matsumoto empieza a desarrollar Ruby
- 1995: Primera versión estable de Ruby es disponible en Japón
- 2000: Ruby supera en popularidad a Python en Japón
- 2003: Ruby 1.8
- 2013: Ruby 2.0, Ruby 1.8 es deprecado
- 2020: Ruby 3, también conocido como Ruby 3x3, ya que los programas son 3 veces más rápidos que con Ruby 2
- Diciembre 2024: Última versión estable de Ruby (Ruby 3.4)

Ventajas y desventajas

Ruby ofrece múltiples librerías (Gems) para diferentes funcionalidades, además de tener un costo de uso muy bajo por ser open source, y, al igual que Python, es fácil de aprender y de leer.

Sin embargo, Ruby es muy estricto con sus paradigmas, por lo que ofrece poca flexibilidad, adicionalmente, el código tiene poca documentación.

Casos de uso

En general, Ruby puede sustituir a Python en cualquier caso de desarrollo Backend. Por ende, sus casos de uso más comunes incluyen:

- Desarrollo Web
- Sistemas electrónicos
- Automatización
- e-Commerce

Casos de éxito

Los principales casos de uso de Ruby en la vida real surgen de su framework para desarrollo web Ruby on Rails. Github, Shopify y Airbnb son algunos ejemplos de empresas masivas que utilizan Ruby en sus sistemas.

5. Ruby on Rails

Definición

Ruby on Rails, o simplemente Rails, es un framework MVC para aplicaciones web para Ruby. Este framework ha influenciado una gran cantidad de frameworks para aplicaciones web para otros lenguajes, incluyendo Django para Python, Grails para Groovy, CakePHP para PHP y Sails.js para Node.js.

Características

Rails facilita el uso de estándares web como XML o Json para data y HTML, CSS y Javascript para interfaces, además de implementar paradigmas como CoC y DRY.

Rails es famoso por su integración con base de datos para creación y migración.

Historia y evolución

- 2004: Los creadores David Heinemeier Hansson publica la primera versión de Rails
- 2007: Apple incluye Rails en Mac OSx Leopard
- 2009: Rails 2.3
- 2011: Rails 3.1, esta versión incluye jQuery como la librería de JavaScript por defecto
- 2013: Rails 4.0

- 2016: Rails 5.0
- 2019: Rails 6.0
- 2021: Rails 7.0
- 2024: Última versión estable Rails 8.0

Ventajas y desventajas

Rails ofrece seguridad para sistemas backend, al igual de ser simple de usar y facilitar la implementación de lógica de negocios y ser compatible con otros frameworks.

Sin embargo, al igual que Ruby, es poco flexible, y no es tan popular como otros frameworks.

Casos de uso

Dado que está orientado a páginas web, Rails es bueno para elementos tales como APIs o Dashboards, redes sociales o plataformas de E-commerce.

Casos de éxito

Adicional a los casos de Ruby, Rails es utilizado en las páginas web de Soundcloud, Indiegogo y Hulu.

6. Kafka

Definición

Apache Kafka es una plataforma open source de envío de eventos escrita en Java y Scala,desarrollada en LinkedIn por Jay Kreps, Neha Narkhede y Jun Rao en el 2010.
Su nombre viene del autor Franz Kafka, Kreps escogió este nombre debido a que es fan del autor.

Características

- Alto throughput: Latencias tan bajas como 2ms
- Escalabilidad: Capacidad para miles de brokers, trillones de mensajes diarios y pentabytes de data
- Almacenaje permanente: Clusters distribuidos tolerantes a fallas
- Alta disponibilidad

Historia y evolución

- 2010: Jay Kreps, Neha Narkhede y Jun Rao empiezan el desarrollo de Kafka en LinkedIn
- 2011: LinkedIn implementa Kafka en sus operaciones
- 2012: Kafka se vuelve Open Source y se une a Apache Software Foundation
- 2025: Última versión estable

Ventajas y desventajas

Kafka es fácil de implementar, tiene un alto desempeño, ofrece una alta seguridad, y adicionalmente tiene una fuerte documentación.

Sin embargo, Kafka es tecnológicamente complejo y no cuenta con herramientas de desempeño o interfaz gráfica, por lo que requiere de desarrolladores experimentados en este. Adicionalmente, ya que depende de Apache Zookeeper, este hereda todas sus desventajas. Sin embargo, esto último se puede evitar en versiones más recientes de Kafka, gracias al protocolo KRaft, el cual elimina las dependencias de Zookeeper.

Casos de uso

Kafka es ideal para el procesamiento de datos en tiempo real, sistemas de mensajería y el registro de eventos y logs en aplicaciones.

Casos de éxito

El 80% de las empresas en el Fortune 100 Companies utilizan Kafka, los sectores principales sectores donde se utiliza incluye infraestructura, bancos y telecomunicaciones

7. Cassandra

Definición

Cassandra es una base de datos noSQL de la familia Apache, es altamente escalable y está diseñada para manejar grandes volúmenes de datos en múltiples servidores, pues se rige bajo los fundamentos de tolerancia a fallos y escalabilidad masiva. Su nombre viene de la profeta Cassandra en mitología Trojana.

Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025

Características

Cassandra utiliza un modelo de datos basado en columnas y maneja una arquitectura distribuida con redundancia y capacidad de recuperación ante desastres, tiene una clasificación AP, lo que la hace tener una alta disponibilidad y tolerancia a fallos, es altamente escalable y maneja la replicación de datos por nodos. Además, es compatible con Apache Hadoop.

Historia y evolución

- 2000: Creada por Avinash Lakshman y Prashant Malik para el motor de la bandeja de entrada de Facebook
- Open Source en Google Cloud
- 2009: Donada a Apache Foundation
- 2011: Versión 1.0
- 2013: Versión 2.0
- 2015: Versión 3.0
- 2021: Versión 4.0
- 2025: Versión 5.0. Última versión estable
- Actualmente: Usada en sistemas de big data y analítica en tiempo real.

Ventajas y desventajas

Al ser distribuida, Cassandra no tiene un único punto de fallo, es altamente disponible y escalable, y es muy buena para escrituras masivas de datos

Sin embargo, su consistencia eventual, no es muy fácil de administrar y no soporta consultas complejas o Joins.

Casos de uso

Cassandra es una opción viable en escenarios donde se requiera una base de datos NoSql, especialmente para sistemas de alta lectura tales como:

- IoT y telemetría en tiempo real
- analitica web
- Plataformas de streaming y redes sociales
- Sistemas financieros de alta disponibilidad

Casos de éxito

- Netflix: para recomendaciones y logs
- Spotify: para metadatos musicales
- Instagram: para mensajes y feeds

8. Estado del Stack en el mercado laboral

Toda la información fue tomada de LinkedIn para Colombia

EDA y Cola de Eventos

Puestos de desarrollador Backend y Fullstack para móvil o web.

Astro

Sin demanda para Astro en específico, pero sí para Landing Pages y SaaS que se pueden hacer en Astro.

Ruby y Rails

Puestos de desarrollador Backend. Requiere de otras tecnologías como .NET, Node.Js o JavaScript.

Kafka

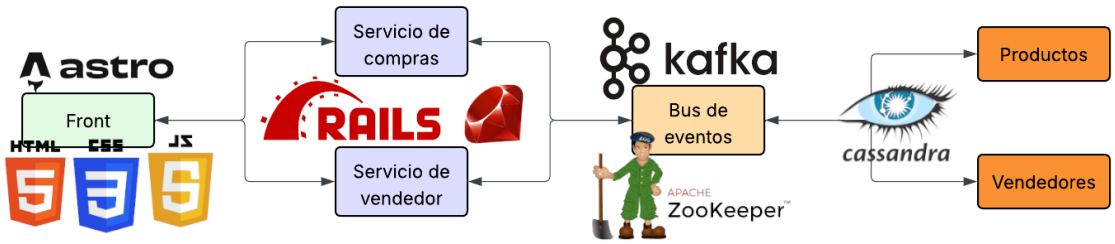
Puestos de desarrollador Backend para Java.

Cassandra

Nicho pequeño, pero existen puestos para ingeniero de datos y desarrollador Backend.

9. Ejemplo práctico y funcional

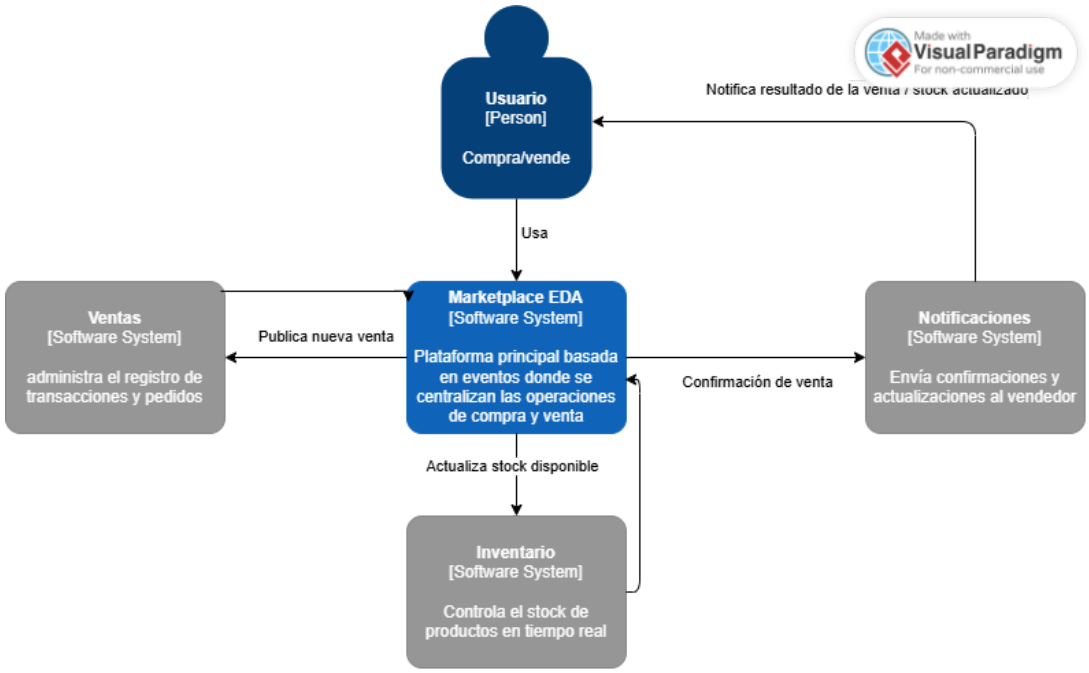
Alto nivel



Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025

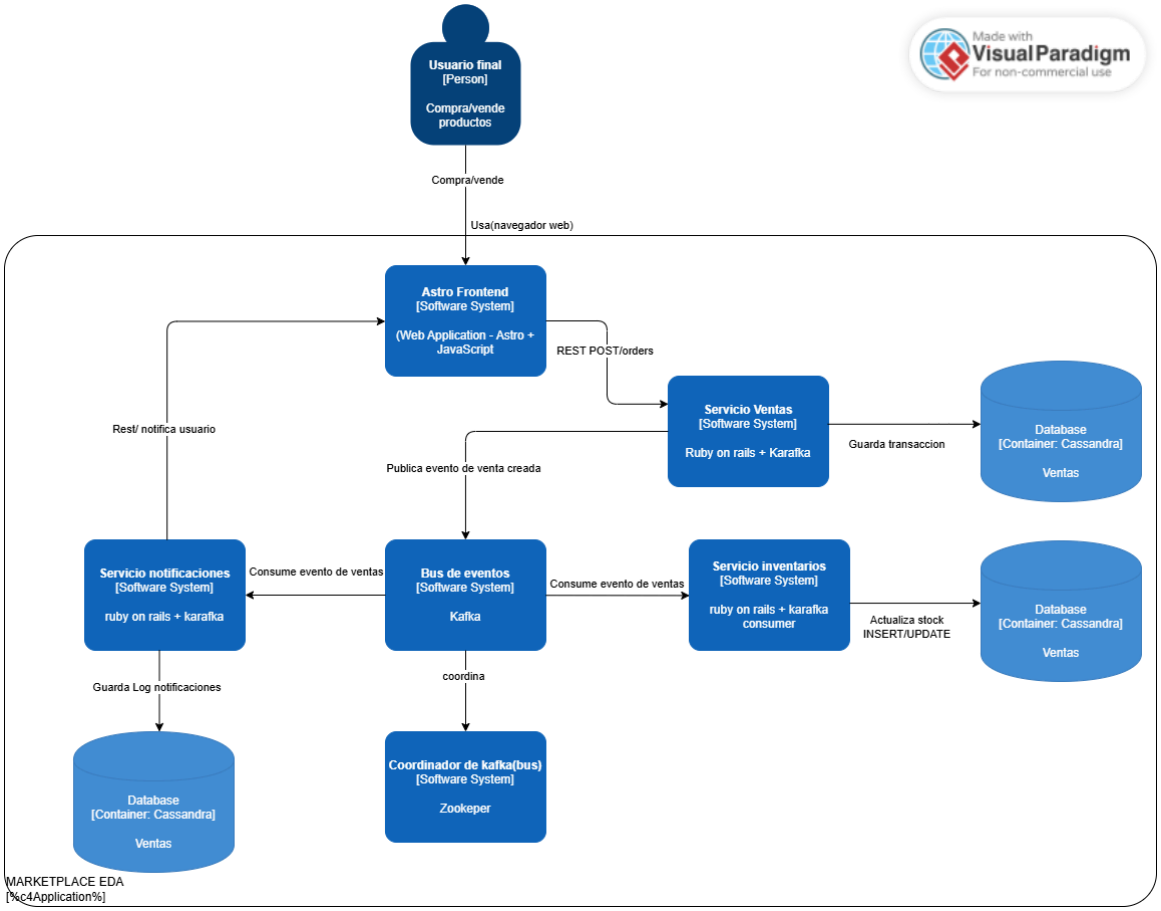
C4Model

- **C1:**

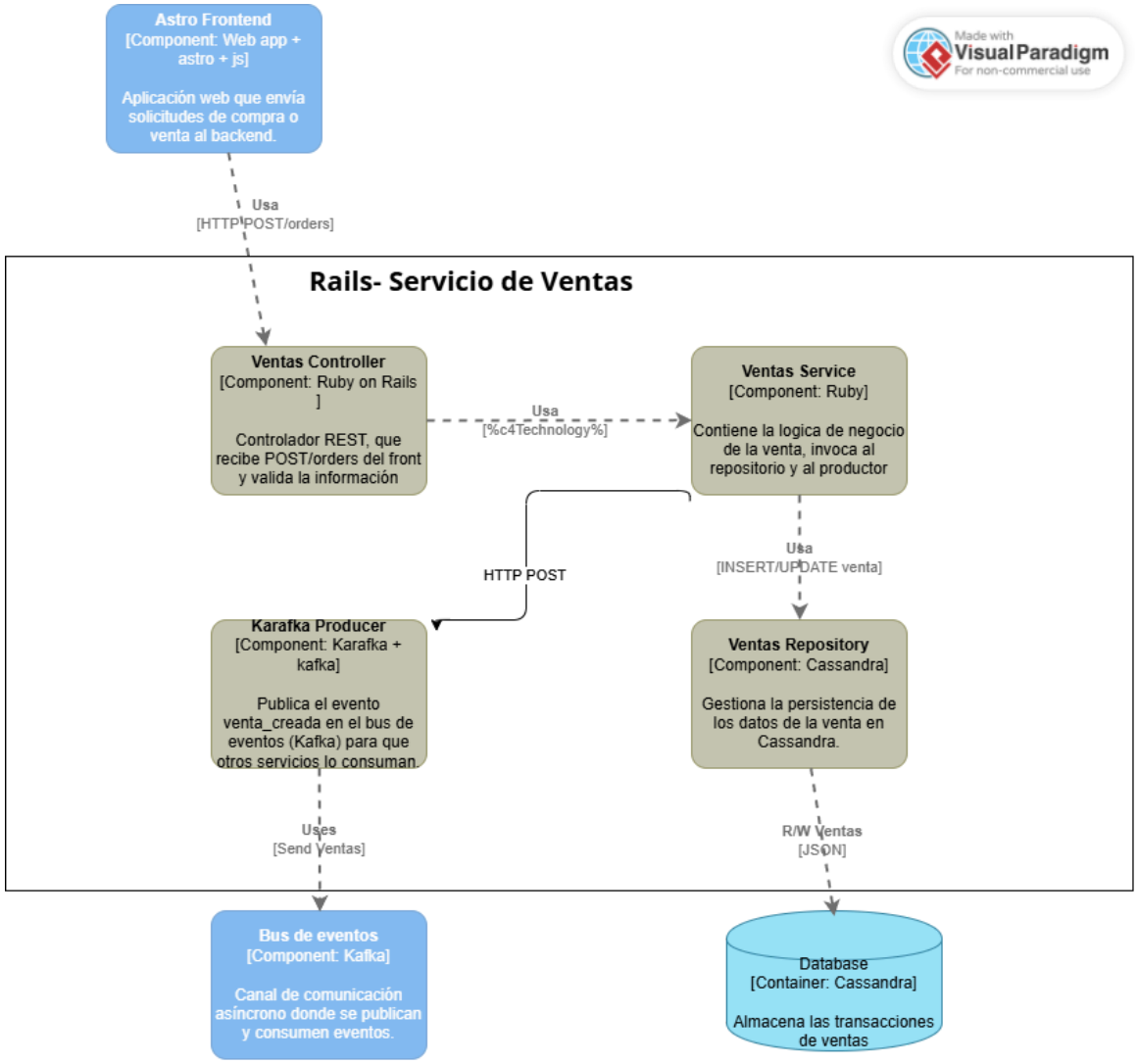


Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025

- **C2:**

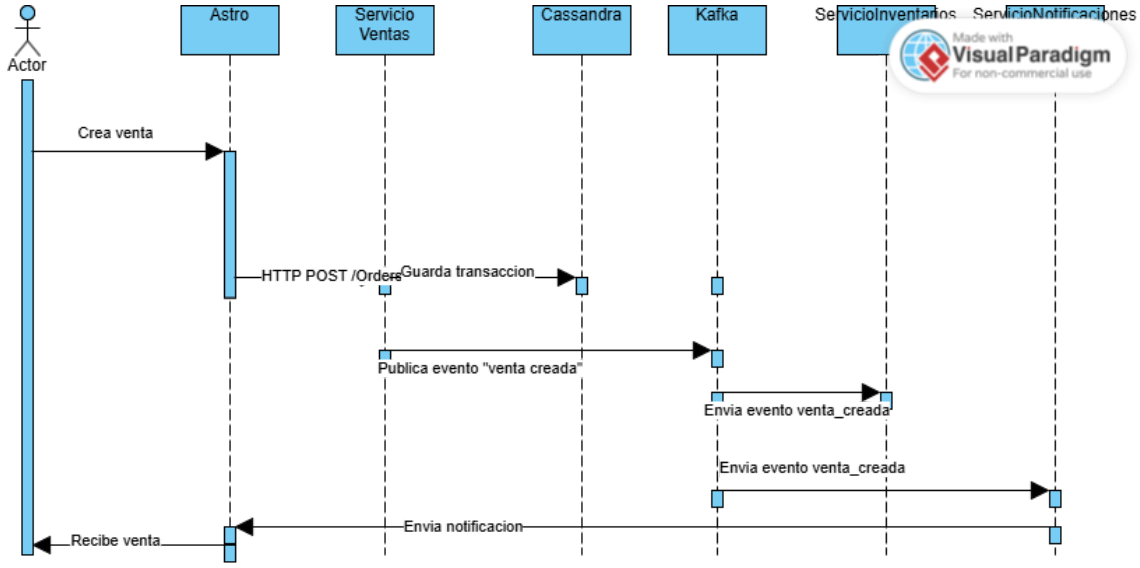


- **C3:**

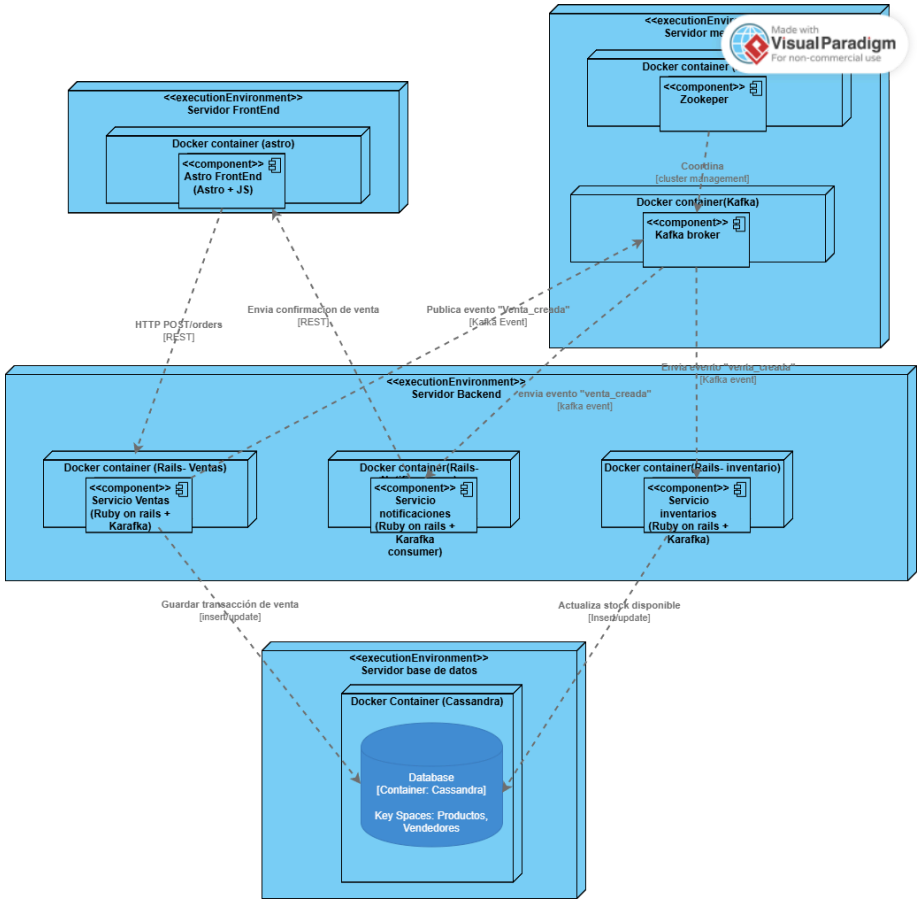


- **Diagrama Dynamic C4**

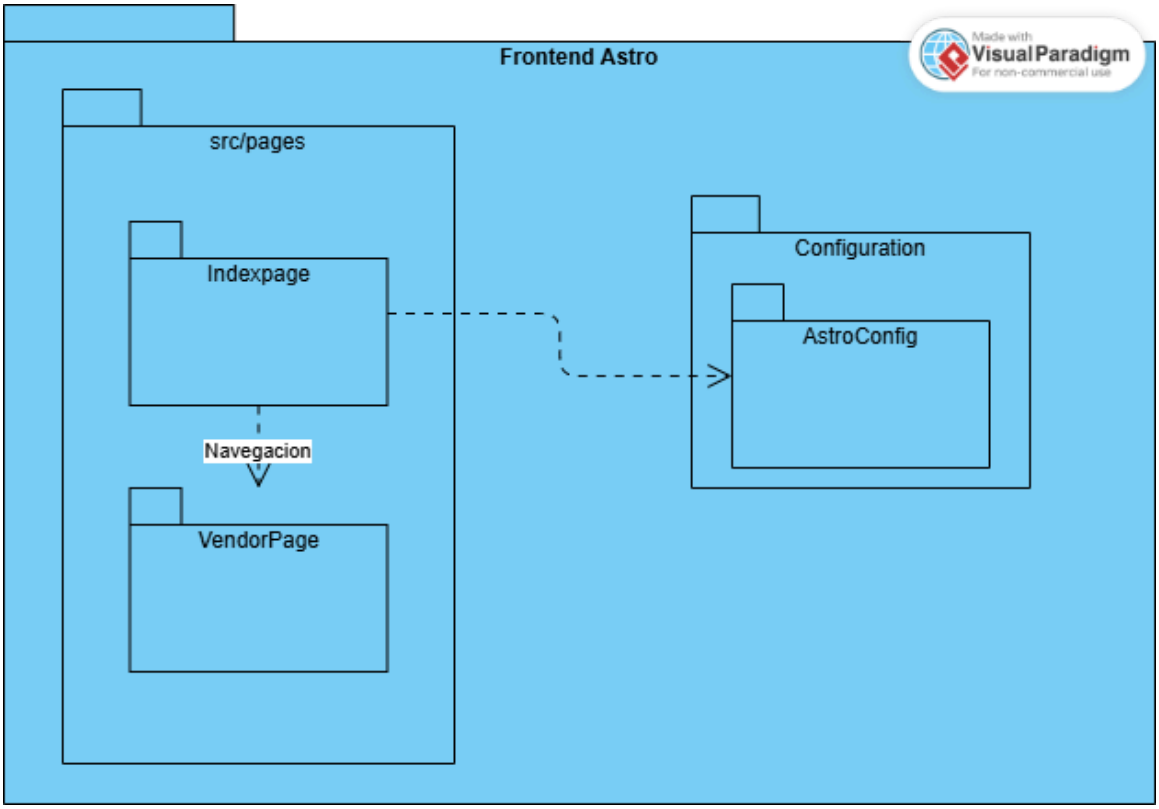
Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025



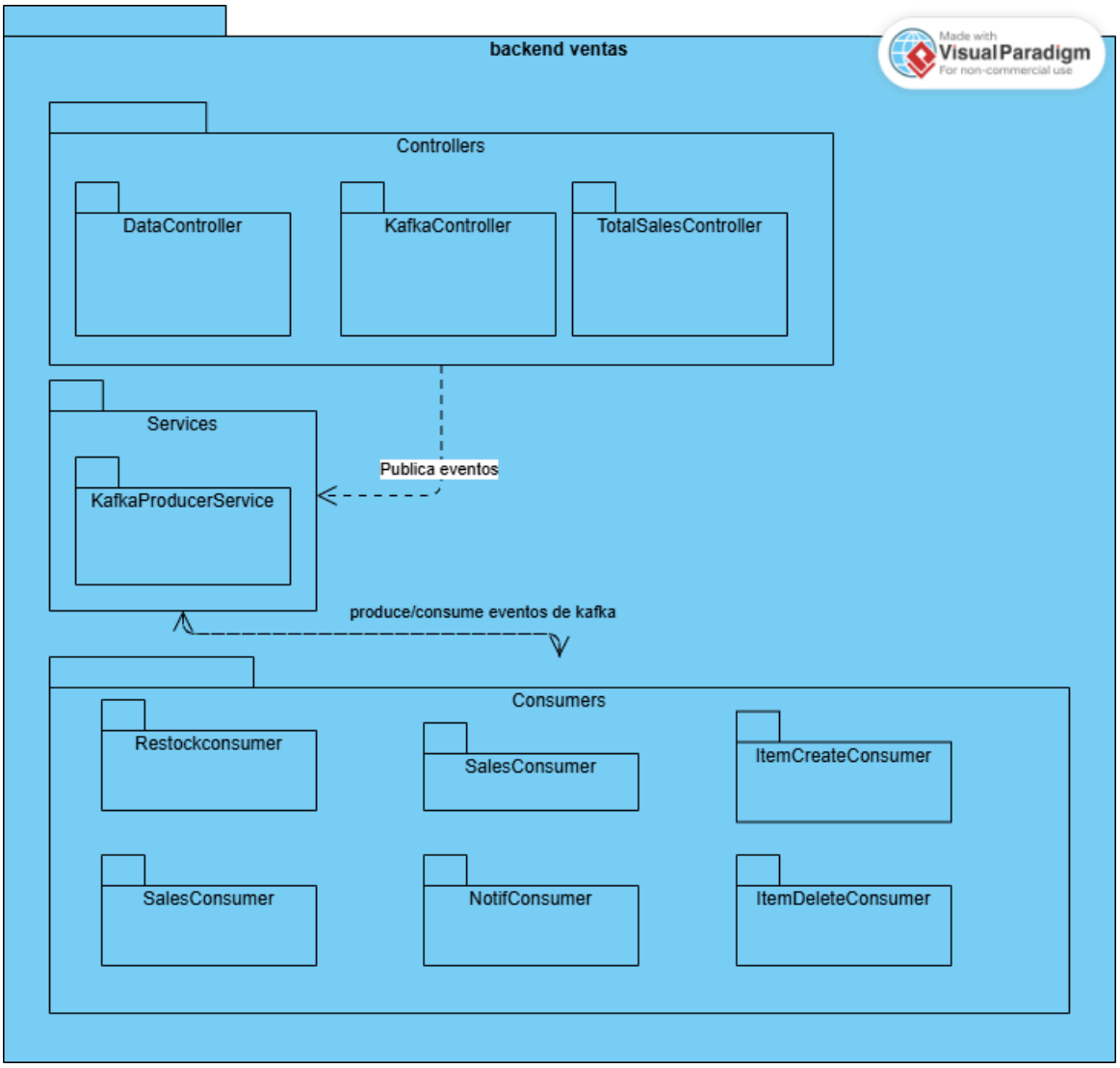
- **Diagrama Despliegue C4**



- **Diagrama de paquetes UML de cada componente**
  - **Frontend**



  - **Backend**



10. Código Fuente en repositorio público Git con Tag

https://github.com/QuinteroEP/AstroMarket

11. Referencias Bibliográficas

- https://en.wikipedia.org/wiki/Apache_Kafka
- https://kafka.apache.org/
- https://www.altexsoft.com/blog/apache-kafka-pros-cons/
- Hohpe, G., & Woolf, B. (2003). Enterprise Integration Patterns. Addison-Wesley.
- Richards, M. (2020). Software Architecture Patterns. O'Reilly Media.

Juan Diego Romero, Pablo Quintero, Kamilt Bejarano
Arquitectura de Software - Taller presentación; Colas y Eventos
Octubre 2025

- Fowler, M. (2017). Event-Driven Architecture. martinfowler.com.
- Bass, L., Clements, P., & Kazman, R. (2022). Software Architecture in Practice (4th ed.). Addison-Wesley.
- Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.
- Apache Software Foundation. (2024). Apache Kafka Documentation. https://kafka.apache.org/documentation/
- Confluent Inc. (2024). Event Streaming Platform Guide. https://www.confluent.io/what-is-event-streaming/
- Netflix TechBlog. (2023). How Netflix Uses Kafka for Real-Time Event Processing. https://netflixtechblog.com/
- Uber Engineering. (2022). Building Reliable Event-Driven Systems at Scale. https://eng.uber.com/
- Namiot, D., & Sneps-Sneppe, M. (2014). On Microservices Architecture. International Journal of Open Information Technologies, 2(9), 24–27.
- Vaughn, V. (2016). Implementing Domain-Driven Design. Addison-Wesley. (Capítulos sobre Event Sourcing y CQRS).
- Junco, R. (2024, octubre 9). Work Queues: The Simplest Form of Batch Processing. System Design Classroom. https://newsletter.systemdesignclassroom.com/p/work-queues-the-simplest-form-of
- Iberasync. (s. f.). Buses de mensajes y colas de mensajes en sistemas distribuidos cloud. Iberasync. Recuperado octubre 2025, de https://iberasync.es/buses-de-mensajes-y-colas-de-mensajes-en-sistemas-distribuidos-cloud/
- Redpanda Data. (2024, junio 18). Enterprise Messaging vs. Event Streaming. Redpanda Blog. https://www.redpanda.com/blog/enterprise-messaging-vs-event-streaming
- https://en.wikipedia.org/wiki/Ruby_(programming_language)
- https://www.geeksforgeeks.org/ruby/ruby-programming-language/
- https://pangea.ai/resources/best-practices-ruby
- https://en.wikipedia.org/wiki/Ruby_on_Rails
- https://www.codica.com/blog/pros-and-cons-of-ruby-on-rails-for-web-development/
- https://en.wikipedia.org/wiki/Apache_Cassandra
- https://stackoverflow.com/questions/2634955/when-not-to-use-cassandra
- https://medium.com/@vinciabhinav7/cassandra-part-2-use-cases-alternatives-and-drawbacks-a152dce60c6b
- https://es.wikipedia.org/wiki/Astro_(framework)
- https://astro.build/
- https://www.linkedin.com/