.UBAfiuba
FACULTAD DE INGENIERÍA

Procesamiento de Lenguaje Natural III

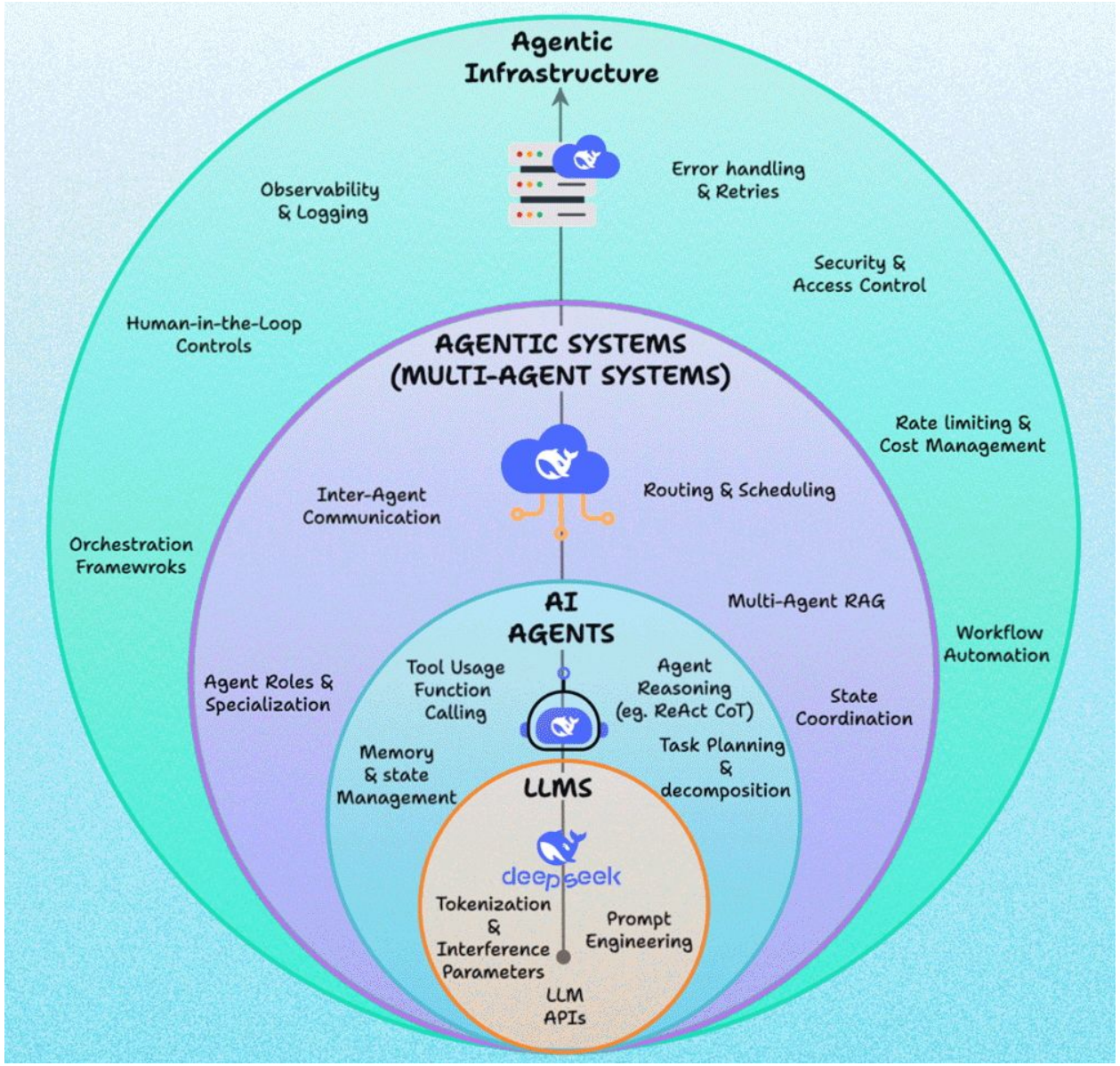
Docentes:
Esp. Abraham Rodriguez - FIUBA
Mg. Oksana Bokhonok - FIUBA

Programa de la materia

1. RAG avanzado y personalización de soluciones
2. Seguridad. Ética y alineación de modelos con valores humanos.
3. Sistemas cognitivos y agentes autónomos.
4. **Escalabilidad y Optimización.**

LLMs, agentes e infraestructura

- Los sistemas con LLMs funcionan bien en prototipos, pero escalar a producción implica **desafíos de costo, latencia, throughput, mantenimiento, desarrollo, integraciones y otros.**
- Escalabilidad debe considerarse en 3 niveles:
  1. **Modelos (LLMs)**
  2. **Agentes**
  3. **Infraestructura**
  4. **Arquitectura de sistemas**



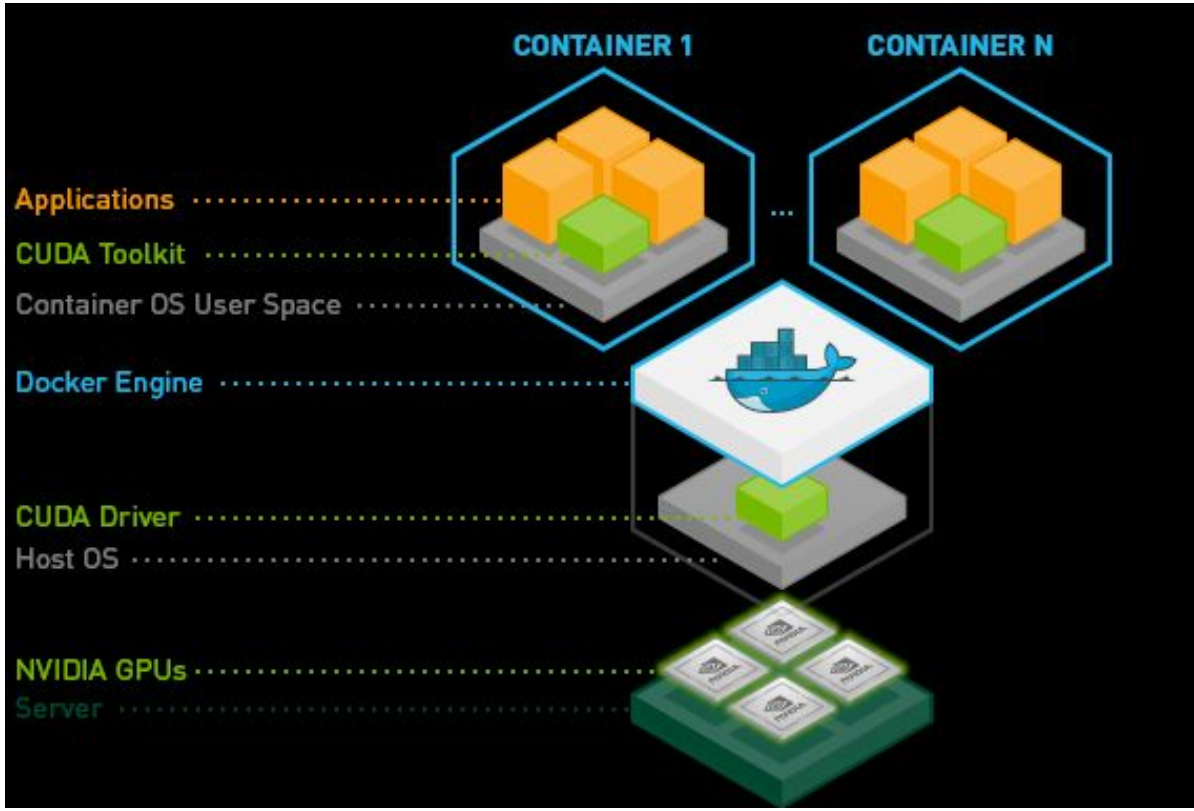
LLMs: On-premise vs Cloud

Depende del tamaño de la organización, capital aplicación a realizar o políticas organizacionales el hecho de elegir uno sobre otro.

[How to Choose the Best Deployment Model for Enterprise AI: Cloud vs On-Prem]

Factor	Qué examinar
Seguridad	¿Qué tan sensible es la información/datos/propiedad intelectual? ¿Existen regulaciones? ¿Riesgo de exposición a través del proveedor o del entrenamiento público de LLM?
Disponibilidad	Latencia (qué tan rápidas son las respuestas), tiempo en línea, confiabilidad, necesidades en tiempo real.
Personalización	Qué tanto necesitas adaptar el modelo o agente a tu dominio, tus propios datos o flujos de trabajo especiales.
Costo	Inversión inicial vs. costos operativos a largo plazo; costo de infraestructura, entrenamiento/fine-tuning, uso transaccional (llamadas a API, etc.), mantenimiento, depreciación.

LLMs: On-premise

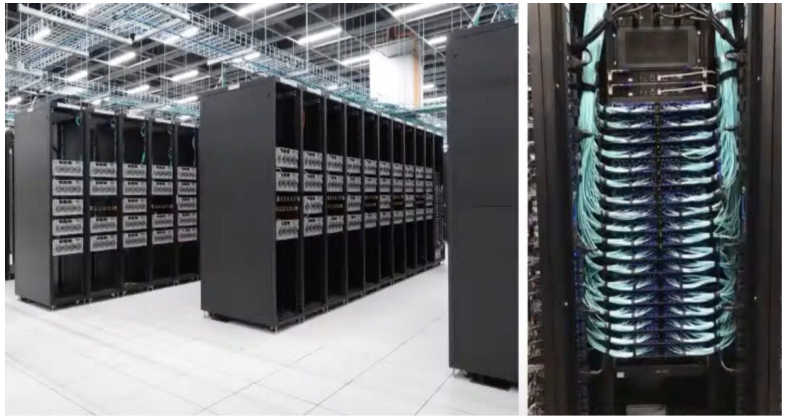


Ventajas

- **Mayor control y seguridad de datos / propiedad intelectual**: Control total, menor riesgo de filtraciones.
- **Mejor rendimiento (latencia, confiabilidad)**: Especialmente para aplicaciones críticas o en tiempo real.
- **Mayor personalización / especificidad de dominio**: Puede servir modelos fine-tuned o custom LLMs y personalizar en profundidad.
- **Ventajas de costo a escala**: Inversión inicial alta, con el tiempo (uso intensivo/grandes volúmenes de datos) los costos pueden amortizarse.

Desventajas

- **Alta inversión inicial**: Infraestructura (hardware, redes, etc.), personal, configuración.
- **Complejidad de mantenimiento**: Responsabilidad interna para actualizaciones/parches, confiabilidad, etc.
- **Escalabilidad más difícil**: Escalar puede requerir más hardware o trabajo de infraestructura.
- **Mayor tiempo para generar valor**.



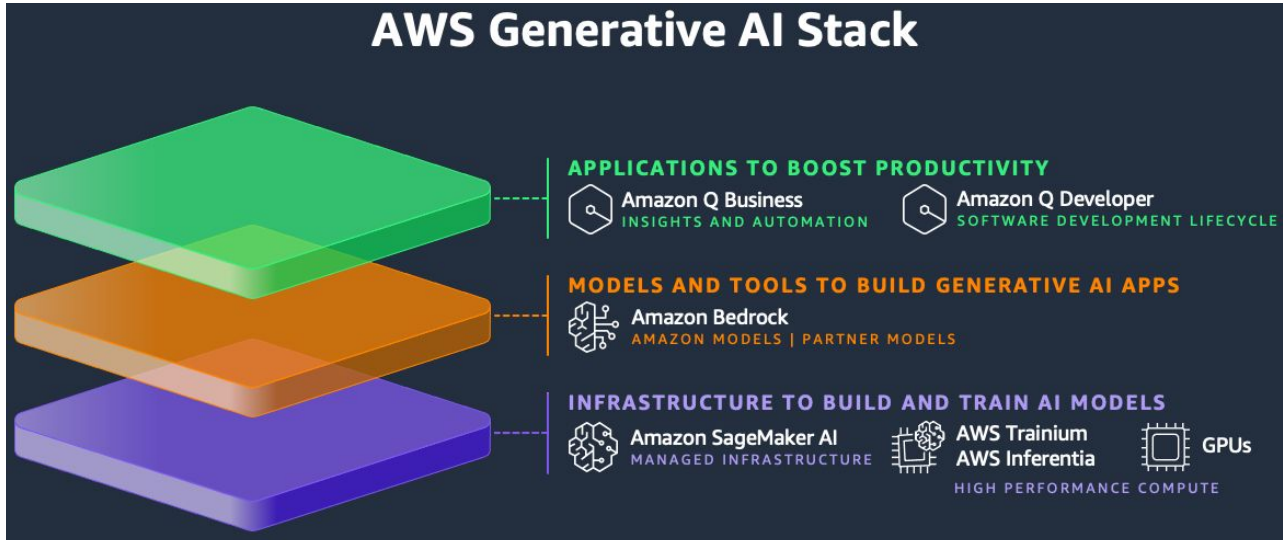
LLMs: Cloud

Ventajas

- **Despliegue rápido / bajo costo inicial**: Más fácil de poner en marcha, menor tiempo para estar en producción (APIs, etc.).
- **Menos infraestructura por gestionar**: El proveedor se encarga de muchas tareas operativas (servidores, software, confiabilidad).
- **Adecuado para experimentación, pruebas de concepto, menor escala o datos menos críticos.**

Desventajas

- **Menor control / riesgo de seguridad**: Limitado a políticas del proveedor sobre la protección de datos/propiedad intelectual;
- **Personalización limitada**: Las capacidades prediseñadas pueden no ajustarse a las necesidades..
- **Posibles problemas de latencia / disponibilidad** según la infraestructura del proveedor, la carga o la red.
- **Costos a largo plazo más altos con mucho uso**, Pay as you go; tarifas de API/uso.



aws Azure Google Cloud Platform

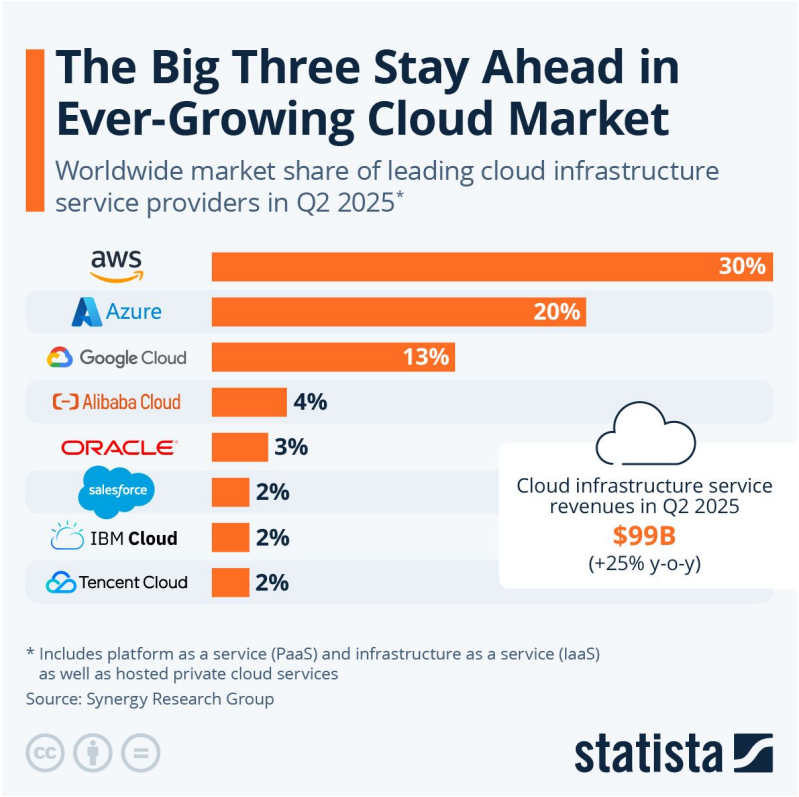
OpenAI ANTHROP\C

LLMs: Cloud, AWS

Gracias a la IA generativa, cloud obtuvo un incremento, AWS es el mayor cloud provider en general, es natural que ofrezcan servicios de IA robustos siendo más relevantes:

- **Amazon Bedrock**
- **SageMaker AI**
- **Acceso a GPUs y chips personalizados en EC2 y EKS.**

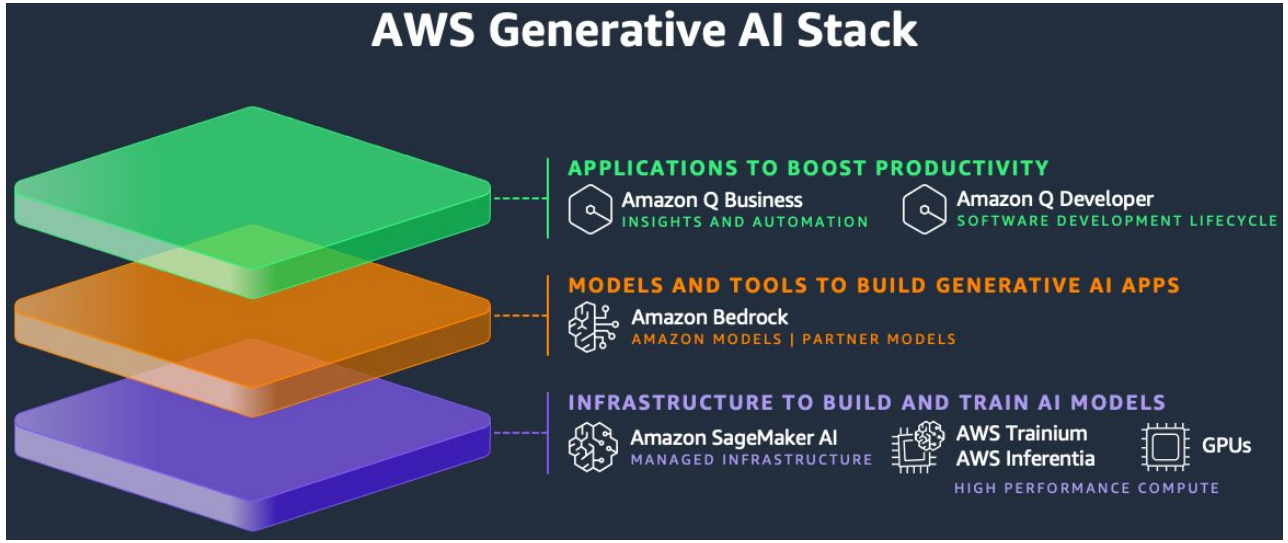
Amazon Bedrock or Sagemaker?



Sin importar qué servicio se elija las LLMs siempre deben ser **expuestas a través de una API**

LLMs: Cloud, AWS Bedrock

Bedrock es útil cuando la necesidad es crear aplicaciones con IA generativa al utilizar foundational models para **inferencia**. Se cuenta con un catálogo de modelos populares como, Claude, DeepSeek, Llama, etc.



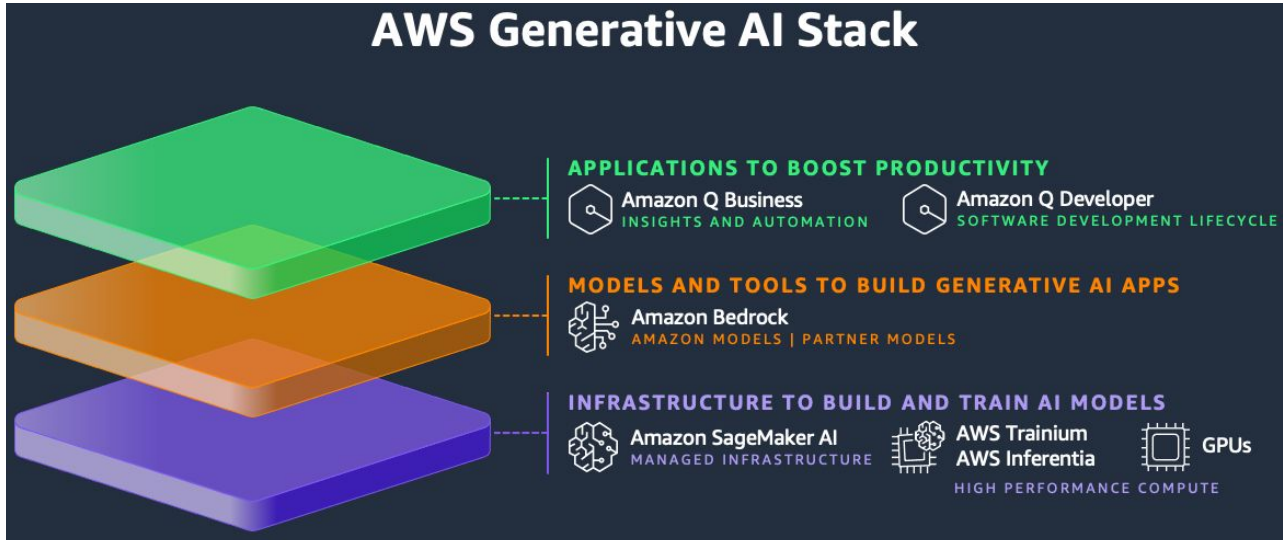
Con bedrock, las LLMs son provistas por AWS el costo on-demand de LLMs puede ser bajo a comparación de host un EC2/EKs ejemplo:

Claude Total cost incurred = 11K tokens/1000 * $0.008 + 4K tokens/1000 * $0.024 = $0.088 + $0.096 = $0.184

LLMs: Cloud, AWS SageMaker AI

SageMaker es un servicio para construir, **entrenar y desplegar** modelos a escala, esto implica crear pipelines, MLOps, profilers y notebooks.

Es útil cuando es necesario crear modelos, fine-tuning, entre otros.

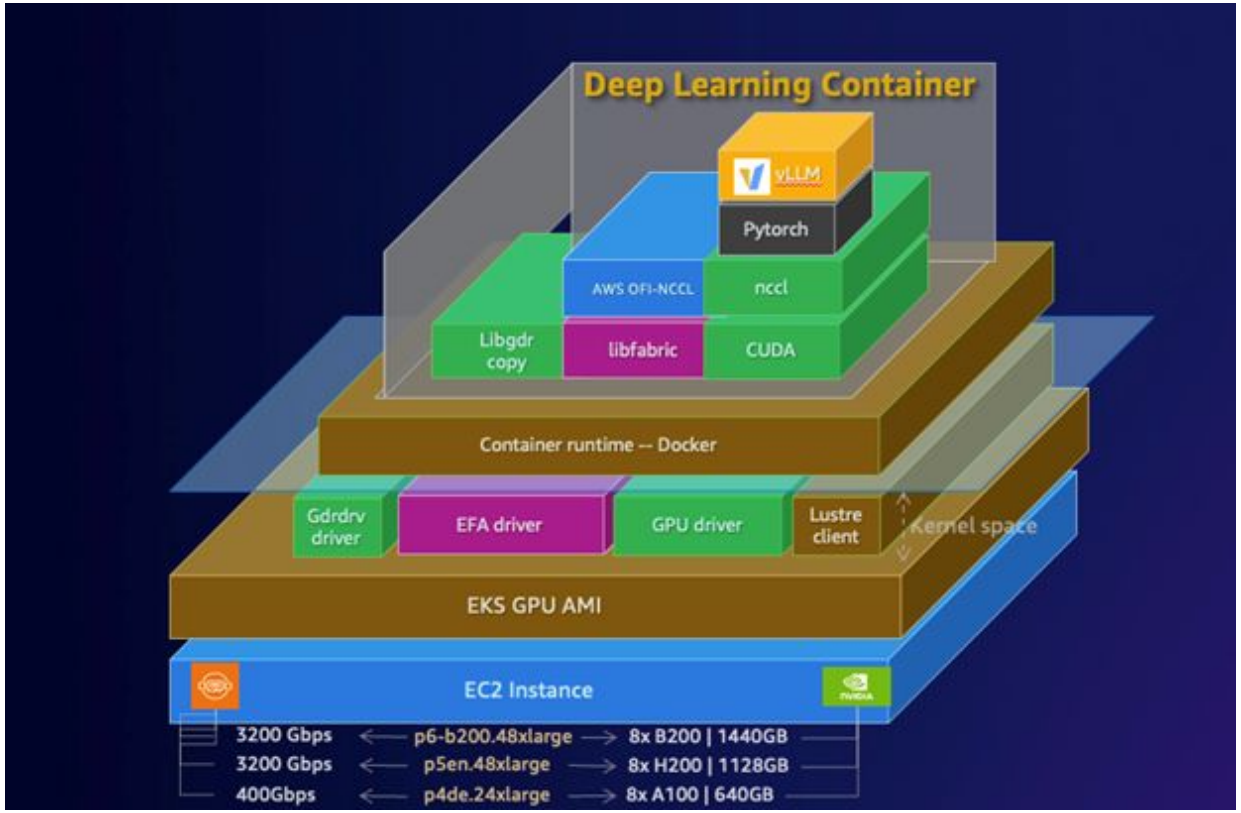


LLMs: Cloud, AWS EC2 y EKS

Elastic Compute Cloud (EC2), es el servicio estándar de instancias de cómputo en AWS, cuenta con instancias con aceleración de hardware.

Sin embargo, puede ser ineficiente ya que se es facturado por que la instancia simplemente existe lo cual no siempre es necesario, para ello se puede utilizar Elastic Kubernetes Service (EKS).

En nuestro caso, **cualquiera de los dos servicios solo deben cumplir una labor, servir LLMs mediante un inference server**, se debe procurar que sea escalable, de bajo mantenimiento y costo.



Serving vLLM with EC2

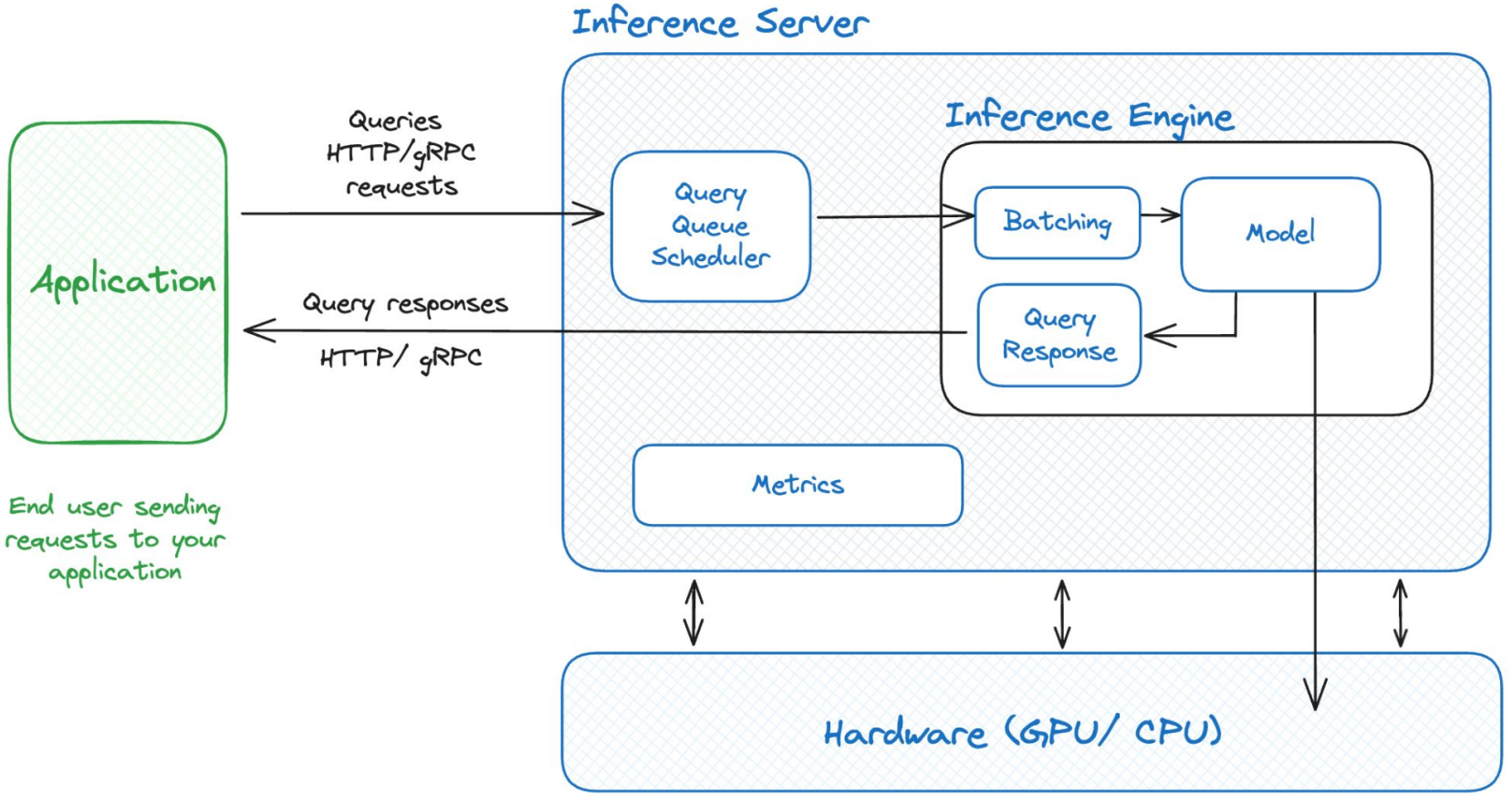
Deploy LLMs on Amazon EKS using vLLM

Serving Meta Llama with EKS

LLMs: APIs

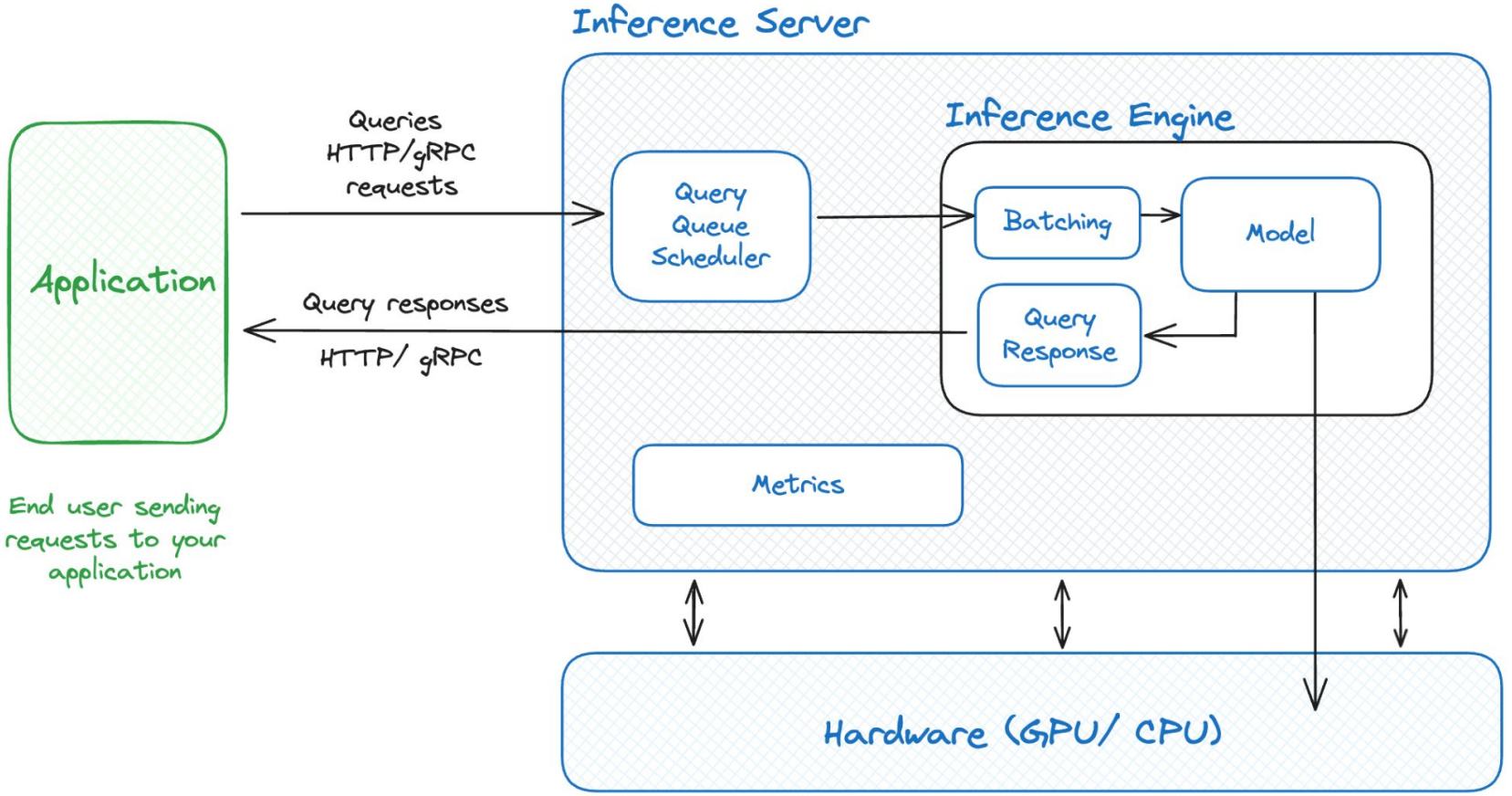
Sea on-prem o cloud, la finalidad es la misma: **exponer LLMs para crear aplicaciones.**

Sin embargo, es a nivel de inference server que se realizan diversas mejoras sobre la ejecución de las LLMs, como: **algoritmos de decodificación, KV-cache, entre otros.**



LLMs: APIs

En LLM providers closed-source es poco común que expongan algoritmos de decodificación o manipulación de parámetros del inference server, mas alla de temperatura, beam search, etc.



En el caso de vLLM, decodificación especulativa está disponible

Amazon Rufus Speculative Decoding

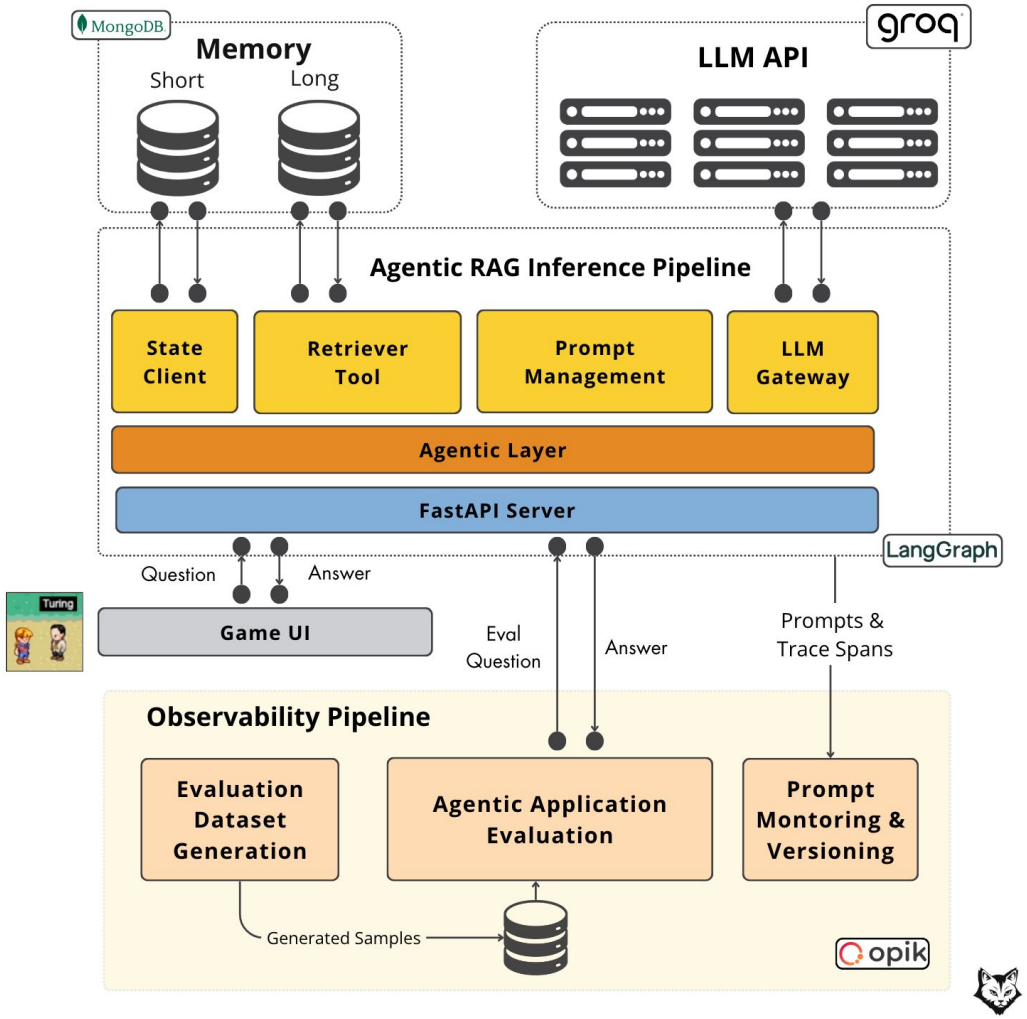
Agentes: App

Luego de tener acceso a una API para interactuar con un inference server, podemos crear un sistema de agentes, etc.

Como cualquier sistema productivo debe ser escalable, soluciones existentes como docker y kubernetes satisfacen esta necesidad.

Servicios como AWS EKS son comúnmente utilizados para agentes/LLMs productivos.

[AI on EKS]



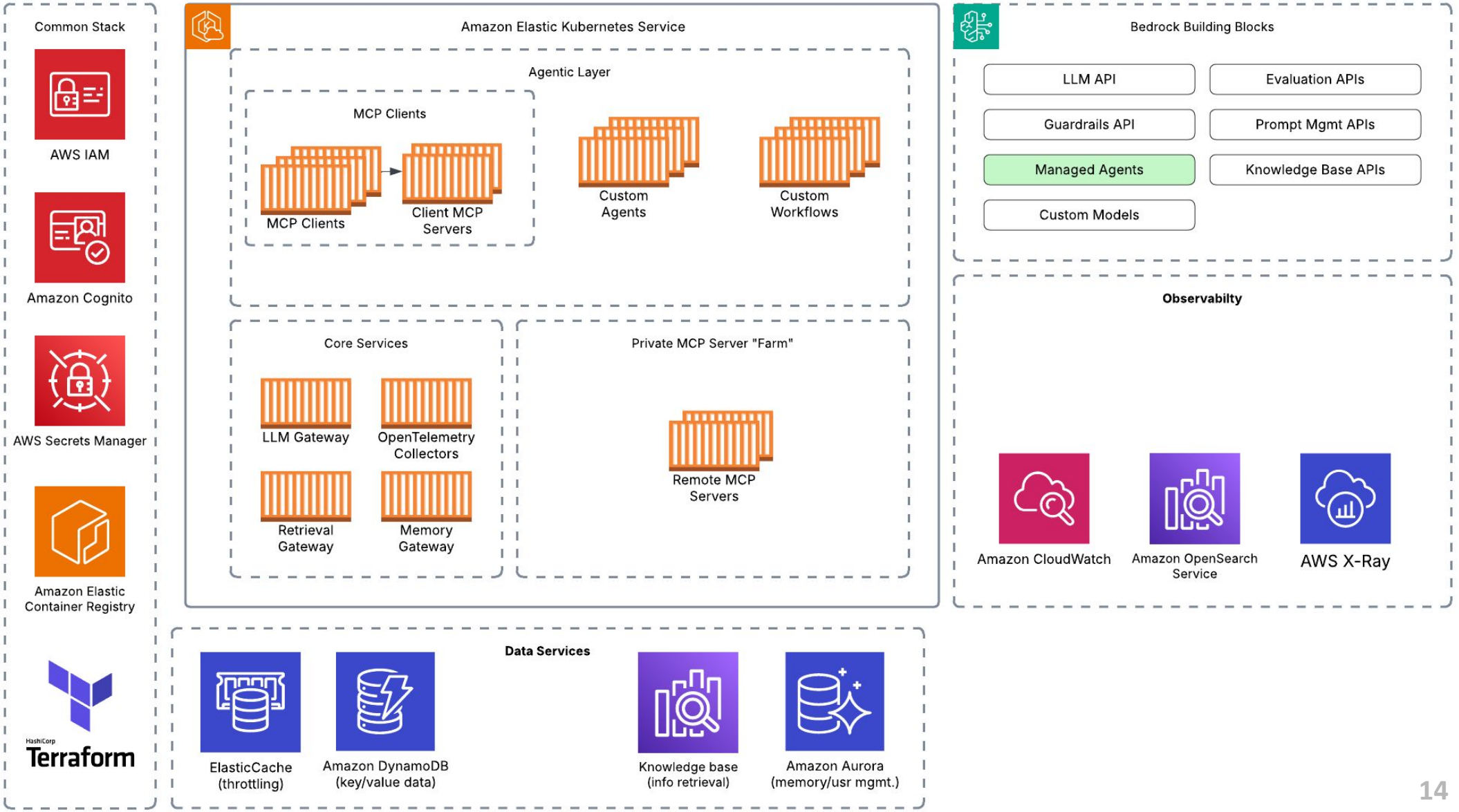
Agentes: Arquitectura e infra

Bedrock, expone un **inference server**.

Sobre este integramos nuestra **app de agentes.**

Se debe tomar en cuenta:

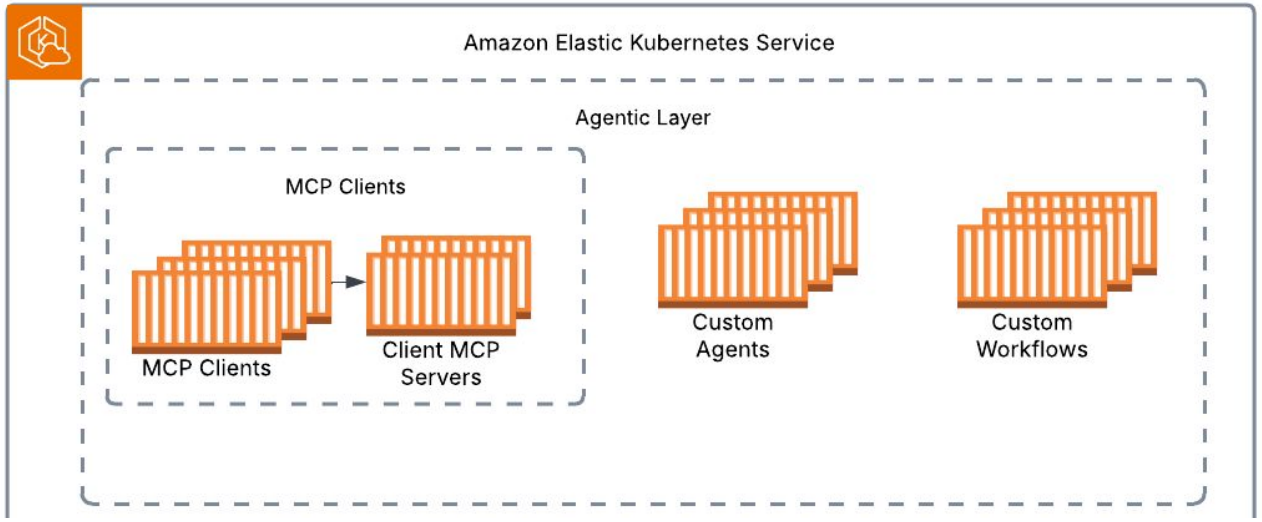
- Orquestación
- OAuth
- MCPs
- Observabilidad,
- Gateways
- DBs



Agentes: Agentic Layer Runtime

El nivel de optimización en **runtime** consiste en:

- **Selección de modelos/costo del Inference Server API.**
  - Selección de modelos pequeños.
  - Enrutamiento de modelos, ej: delegación de tareas complejas a LLMs de razonamiento.
  - Modelos cuantizados.
- Comunicacion Agent 2 Agent.
- **Uso de tokens/manejo de contexto.**
  - Compresión de conversación
- **Caching**
  - Embeddings cache
- **Ejecución async**
- **Tooling**
  - Async
  - Caching



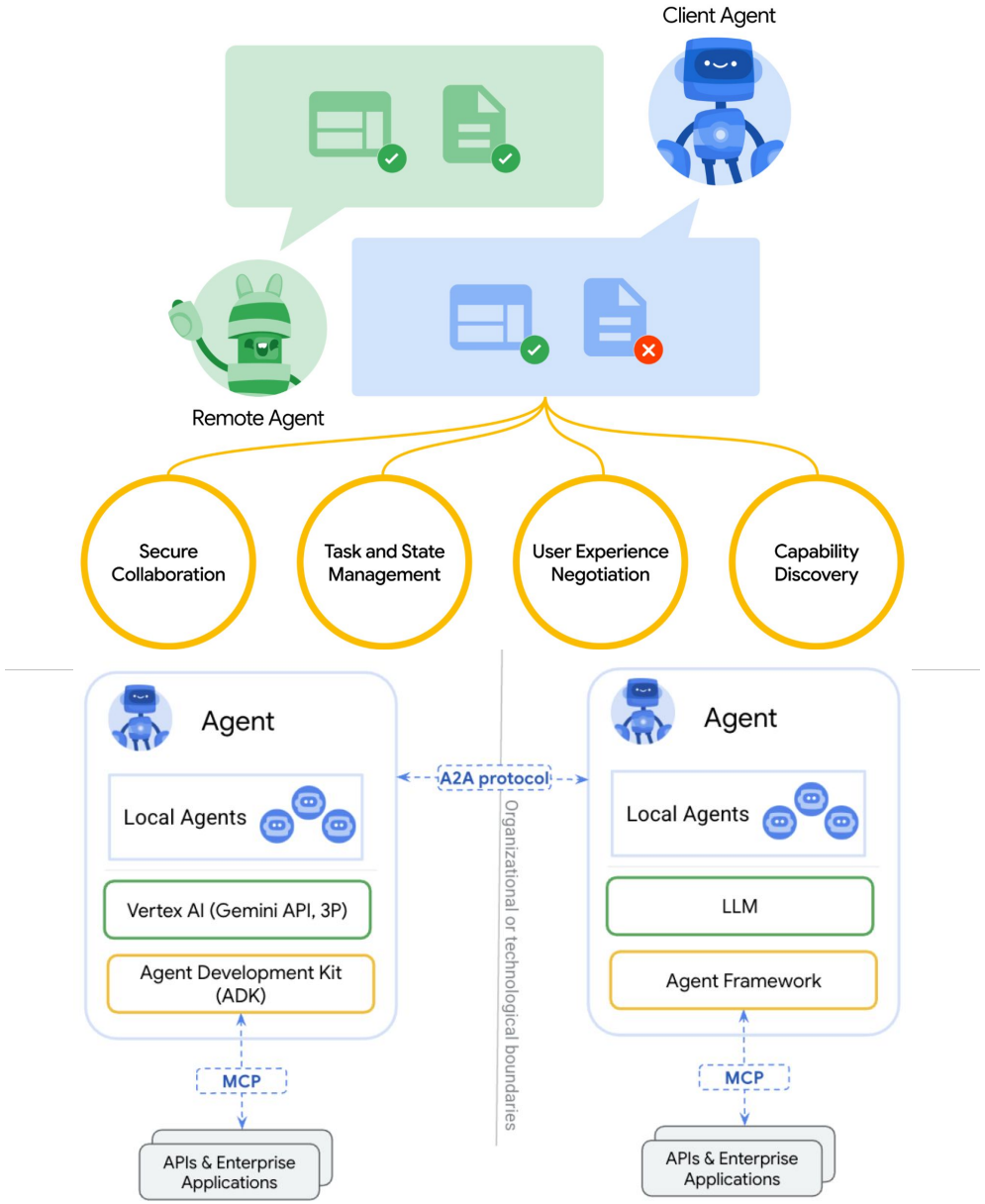
Agentes

Agentes: Agent 2 Agent Protocol

La capa de Agentes, cuenta con un nuevo protocolo que **complementa** a MCP (tooling). A2A consiste en flujos de agentes capaces de conectar con otros agentes con la finalidad de colaborar.

Es muy similar a microservices en términos de ser un sistema distribuido.

[A2A Protocol]
[A2A Protocol Site]



Agentes: Agent 2 Agent Protocol

Un sistema multiagente para planificar viajes implica orquestar agentes de:

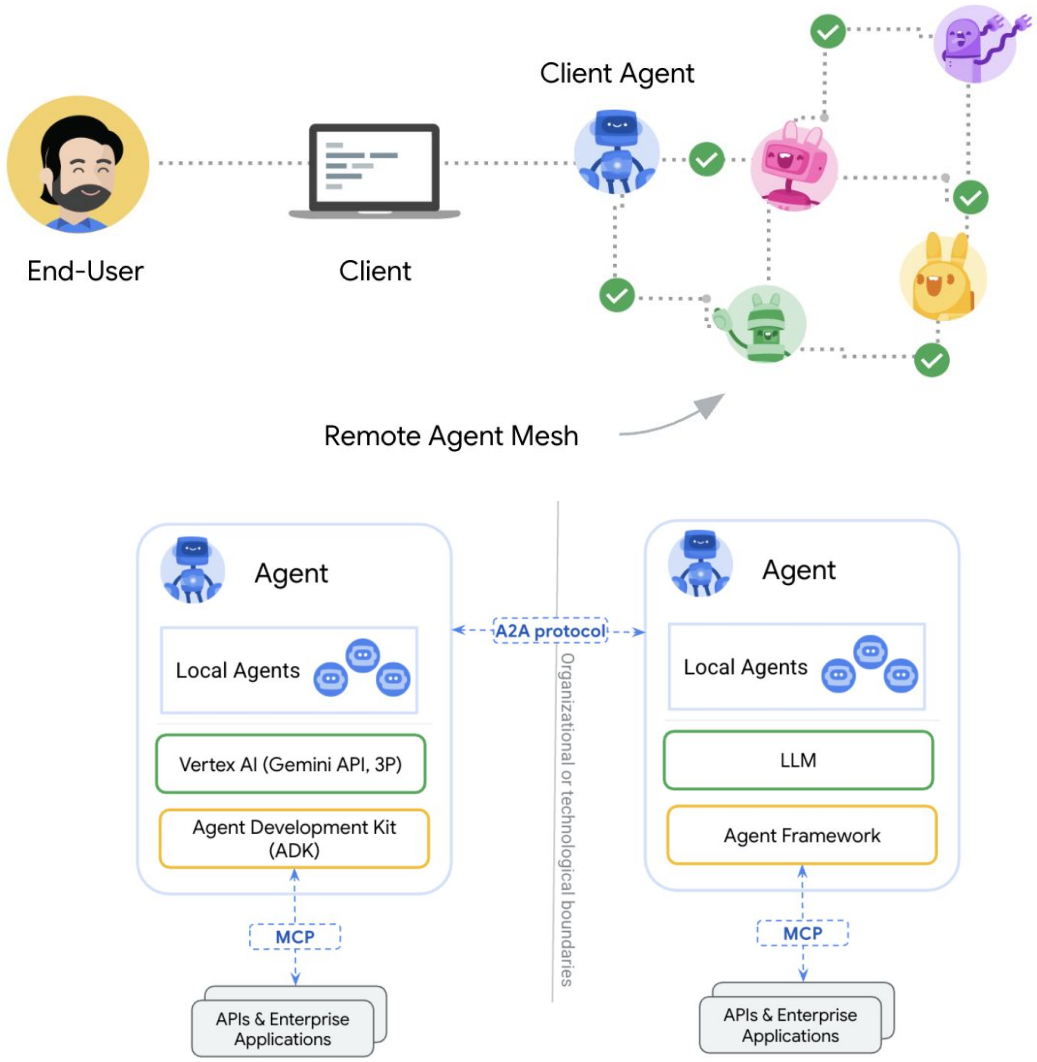
- Vuelos
- Reservaciones de hotel
- Recomendaciones de tours
- Recomendaciones de actividades
- Conversión de monedas
-

Esto conlleva a problemas de escalabilidad a medida que la cantidad de agentes aumenta.

Cada agente implica implementaciones custom/horas de ingeniería.

A2A simplemente consiste en comunicar un agente con otro sin importar su implementación (similar a microservicios).

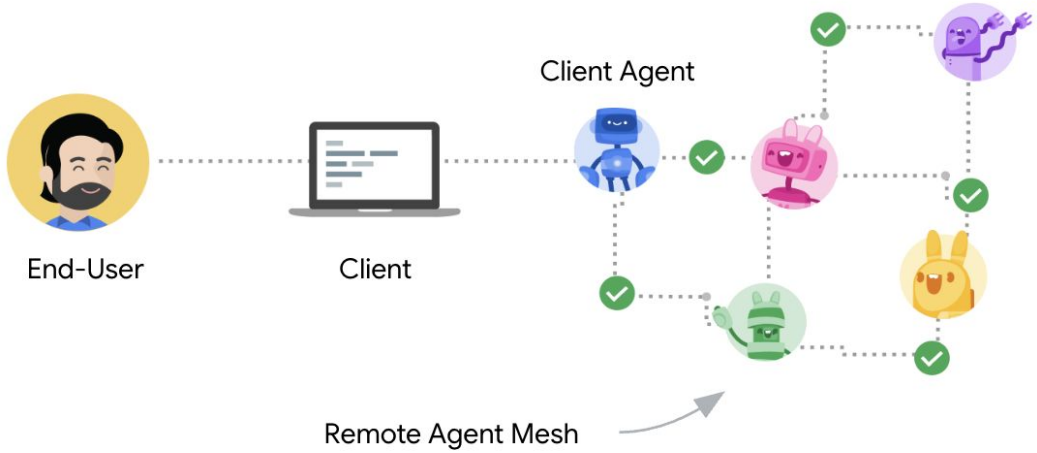
[A2A Overview]



Agentes: Agent 2 Agent Protocol

Un agente cliente (orquestador) se comunica con agentes externos, pero para conocer acerca de ellos. A2A expone Agent discovery mediante una estructura JSON llamada “Agent Card” que consiste en:

Elemento	Descripción
Identidad	Incluye nombre, descripción e información del proveedor.
Service Endpoint	Especifica la URL para el servicio A2A.
Capacidades A2A	Enumera las funciones compatibles, como *streaming* o *pushNotifications*.
Auth	Detalla los esquemas requeridos (por ejemplo, "Bearer", "OAuth2")
Habilidades	Describe las tareas del agente usando objetos “AgentSkill” siendo: *id, nombre, descripción, inputModes, outputModes y ejemplos.*



Similar a MCP, A2A utiliza HTTP, JSON-RPC y SSE asi como gRPC, REST

[LangGraph server A2A]

Agentes: Agentic Layer Build time

Parte del CI/CD pipeline, el build es uno de los procesos más pesados/tardados en realizar, es mayormente dependiente de la tecnología o lenguaje en la que está escrita la aplicación.

En el caso de Python, los builds pueden tardar considerablemente debido a las dependencias.

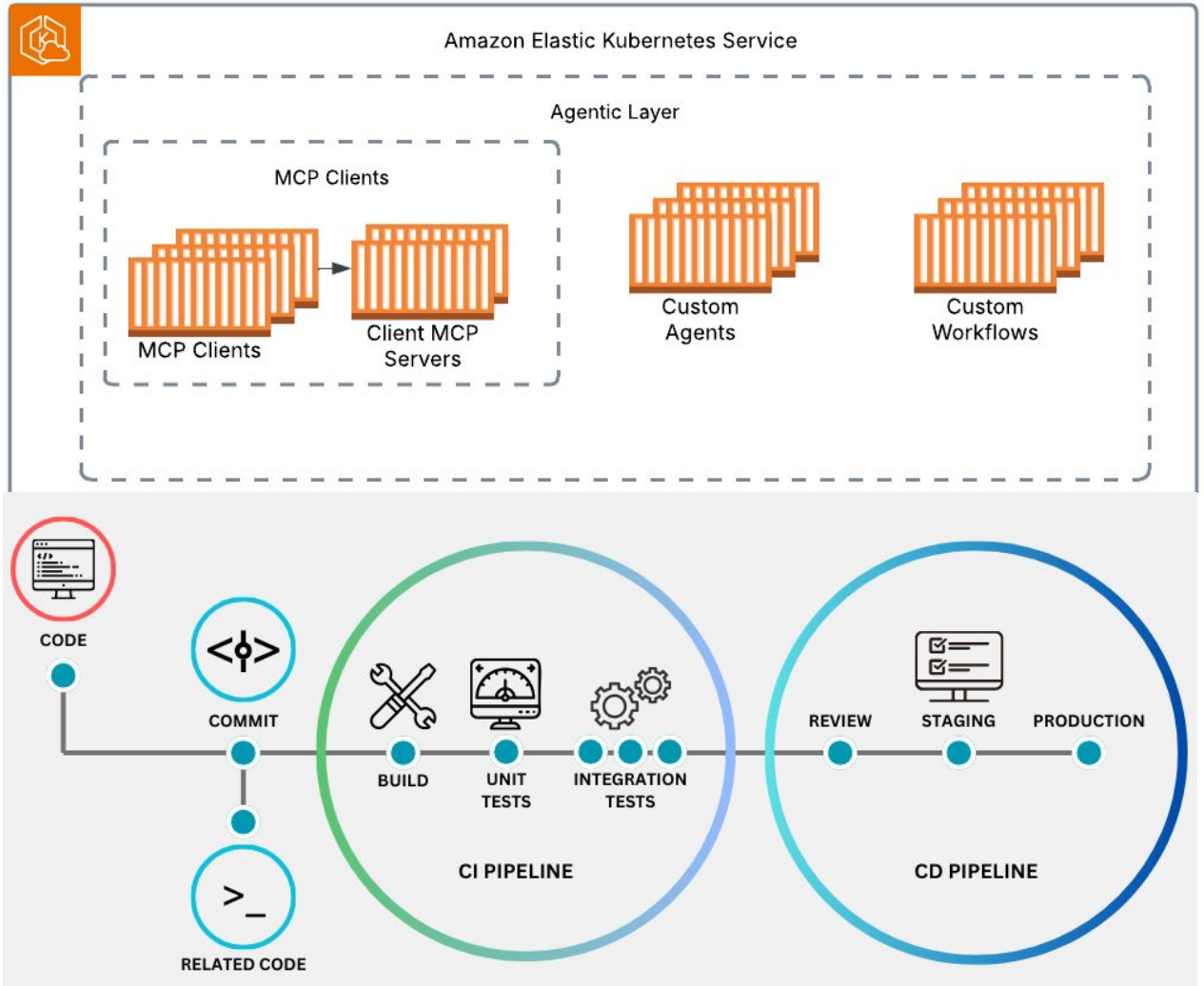
En términos de tamaño de la imagen:

Las imagenes de containers de Python son pesadas a comparacion de otras tecnologías, para ello se suele utilizar alpine o slim. Para reducir aún más se puede utilizar multi-stage builds.

Con respecto a la velocidad del build:

Pip ha sido el package manager default de Python sin embargo, es considerablemente lento por su naturaleza secuencial.
uv es un package manager escrito en **Rust** considerablemente más rápido que pip **util especialmente en build time.**

uv pip docker speedup



Agentes: Data layer

Como toda app, el data layer es la más **esencial,** aquí habitan DBs, Cache, Knowledge Bases, Message brokers, etc.

Sobre el data layer, la optimización es crucial, ej:

- **SQL queries lentos** ➜ se soluciona con índices, particionamiento, profiling.
- **Stored procedures** ➜ útiles para encapsular lógica pesada cerca de los datos, pero deben revisarse por mantenimiento y performance.
- **Cache sin TTL** ➜ riesgo de servir datos obsoletos o crecer sin control.
- **Vector DBs con mala configuración** ➜ búsquedas lentas, mezclas de dominios distintos en el mismo índice, falta de segmentación por metadata.

Agentes: SQL Queries

En aplicaciones productivas lo ideal es tratar los SQL Queries cómo tools estructuradas (similar a APIs).
Otra forma es utilizar stored procedures.

```python
from pydantic import BaseModel, Field
from langchain.tools import StructuredTool

class UserQuery(BaseModel):
    loyalty_member: bool = Field(..., description="Filter by loyalty membership")

@tool
def get_users(input: UserQuery) -> str:
    cur = conn.cursor()
    if input.loyalty_member:
        cur.execute("SELECT * FROM users WHERE loyalty_member = 1;")
    else:
        cur.execute("SELECT * FROM users;")
    rows = cur.fetchall()
    return str(rows)
```

Agentes: SQL Stored Procedures

.UBAfiuba
FACULTAD DE INGENIERÍA

Aparte de queries directos, otra forma que es de mayor performance y seguro es utilizar stored procedures.

El stored procedure es una “función” de SQL la cual se implementa **exclusivamente dentro de la DB**. Se expone el procedure de tal manera que un **usuario con permisos de ejecución** del puede llamar la función con el input estricto y output estructurado.
Esto es ideal para agentes y evita la realización de queries “on the fly”.

```sql
# SQL
CREATE PROCEDURE GetUsersByTier(IN tier_level INT)
BEGIN
  SELECT * FROM users WHERE loyalty_tier = tier_level;
END;
```

```python
from pydantic import BaseModel, Field
from langchain.tools import StructuredTool

class UserTierInput(BaseModel):
    tier_level: int = Field(..., description="Loyalty tier number (e.g., 1, 2, 3)")

@tool
def get_users_by_tier(input: UserTierInput) -> str:
    cur = conn.cursor()
    cur.callproc("GetUsersByTier", [input.tier_level])
    results = []
    for result in cur.stored_results():
        results.extend(result.fetchall())
    return str(results)
```

Agentes: SQL Stored Procedures

.UBAfiuba
FACULTAD DE INGENIERÍA

Realizar queries en DBs SQL implica realizar uso de network. En el caso de MySQL existen varios métodos definidos en el [Command Phase] el cual consiste en enviar **data packets** entre client/server. El más simple es **Text Protocol** siendo los inline queries, ej:

*Drop Table, Select * From.*

También existen los **prepared statements** (binary protocol), mas optimo debido a que reduce el parseo del texto SQL.
[MySQL Client/Server protocol]
[MariaDB Text Protocol]

El stored procedure implica 1 solo round trip.
[Stored Procedures to simplify DB Operations]

```sql
# SQL
CREATE PROCEDURE GetUsersByTier(IN tier_level INT)
BEGIN
  SELECT * FROM users WHERE loyalty_tier = tier_level;
END;
```

```python
from pydantic import BaseModel, Field
from langchain.tools import StructuredTool

class UserTierInput(BaseModel):
    tier_level: int = Field(..., description="Loyalty tier number (e.g., 1, 2, 3)")

@tool
def get_users_by_tier(input: UserTierInput) -> str:
    cur = conn.cursor()
    cur.callproc("GetUsersByTier", [input.tier_level])
    results = []
    for result in cur.stored_results():
        results.extend(result.fetchall())
    return str(results)
```

Agentes: Data layer

- **Bases de datos relacionales (SQL)**
  - PostgreSQL, MySQL, MariaDB, SQL Server, Oracle.
- **Bases de datos NoSQL**
  - MongoDB, Cassandra, DynamoDB.
  - Flexibles para documentos, key-value.
- **Data Warehouses**
  - BigQuery, Snowflake, Redshift.
- **Knowledge Bases / Vector DBs**
  - Pinecone, Weaviate, Milvus, Chroma.
  - Almacenamiento semántico para RAG.
- **Knowledge Graphs**
  - Neo4j, TigerGraph.
  - Representan relaciones explícitas entre entidades.
  - Soportan razonamiento más simbólico y navegaciones complejas.

- **Cache**
  - Redis, Memcached.
  - TTL y políticas de invalidación son clave.
- **Message Brokers / Streaming**
  - Kafka, RabbitMQ, Pulsar.
  - Sirven como backbone de eventos y comunicación asíncrona, permite reaccionar en tiempo real a cambios del entorno.
- **Data Lakes**
  - S3, HDFS, Delta Lake.
  - Repositorios masivos de datos crudos, estructurados y no estructurados.

Agentes: Data layer, Snowflake

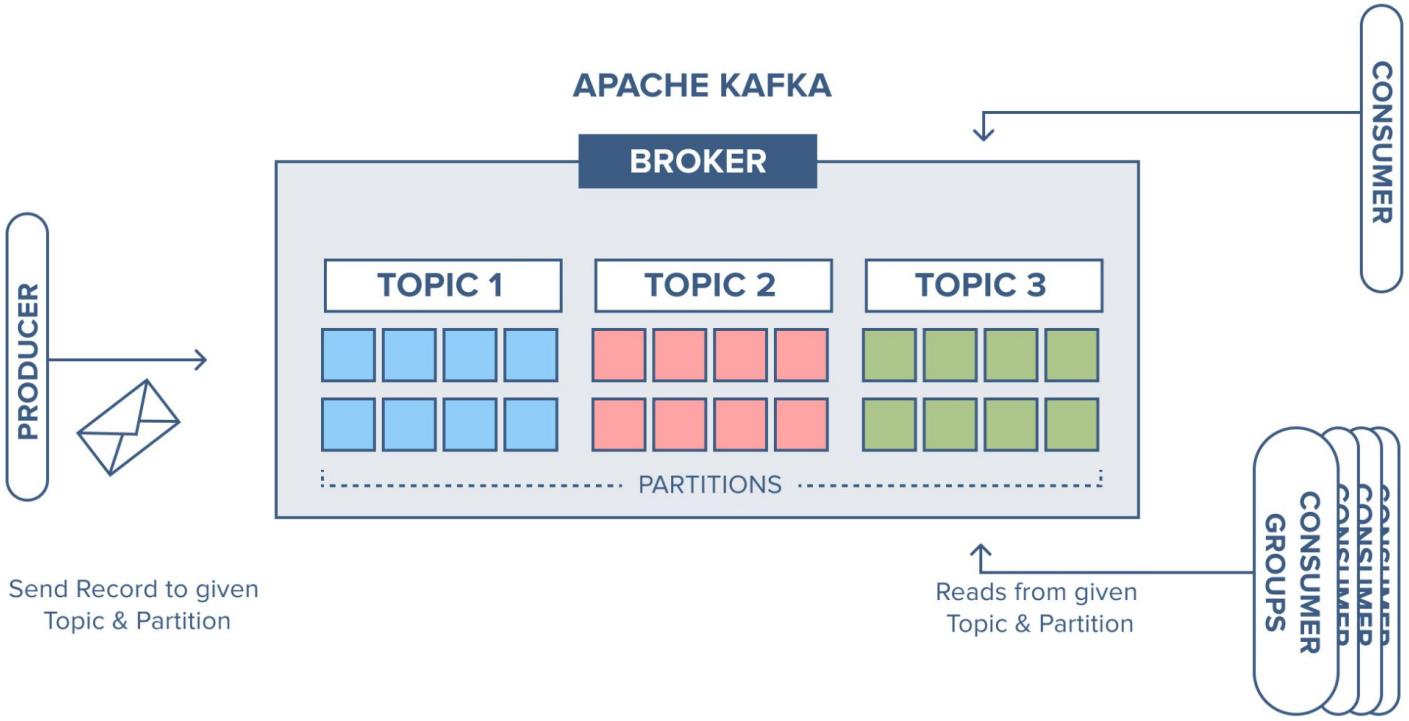
- https://www.snowflake.com/en/product/features/cortex/

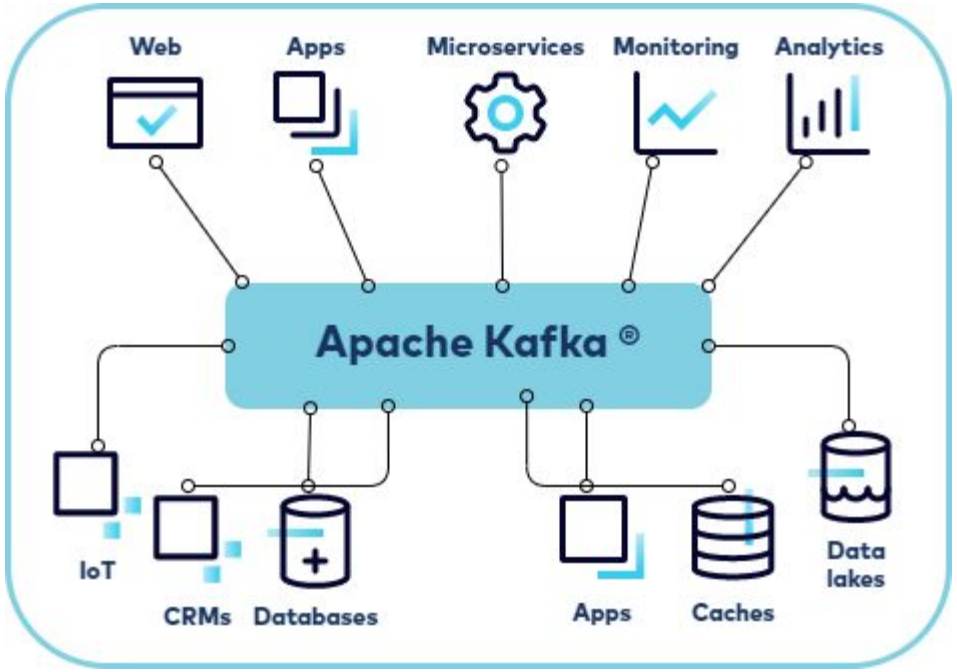
Snowflake Cortex as LangGraph Tool

Snowflake Agentic Workflow

Apache Kafka 101

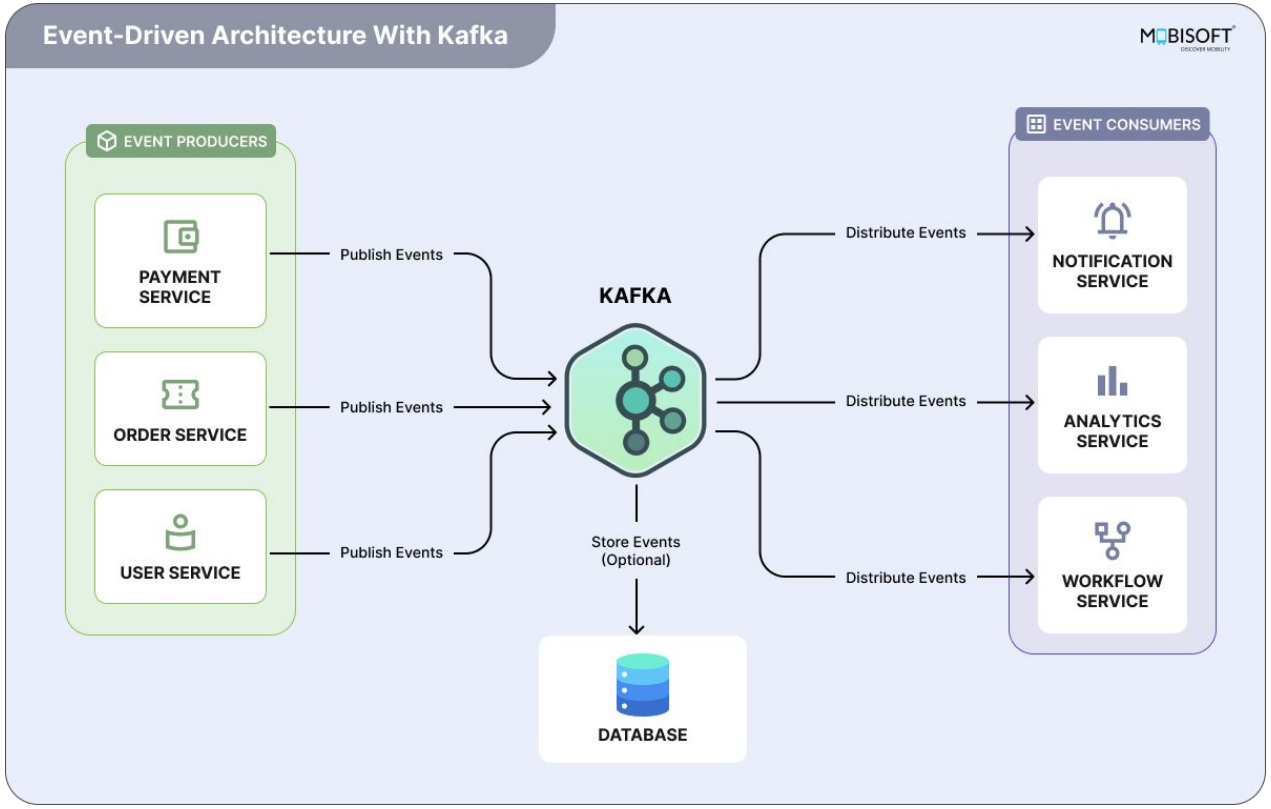
Presentado por LinkedIn en 2011, Apache Kafka empezo siendo un sistema de **procesamiento de logs**, hoy en día es una plataforma de streaming de eventos utilizado para high-performance pipelines, y aplicaciones en tiempo real.

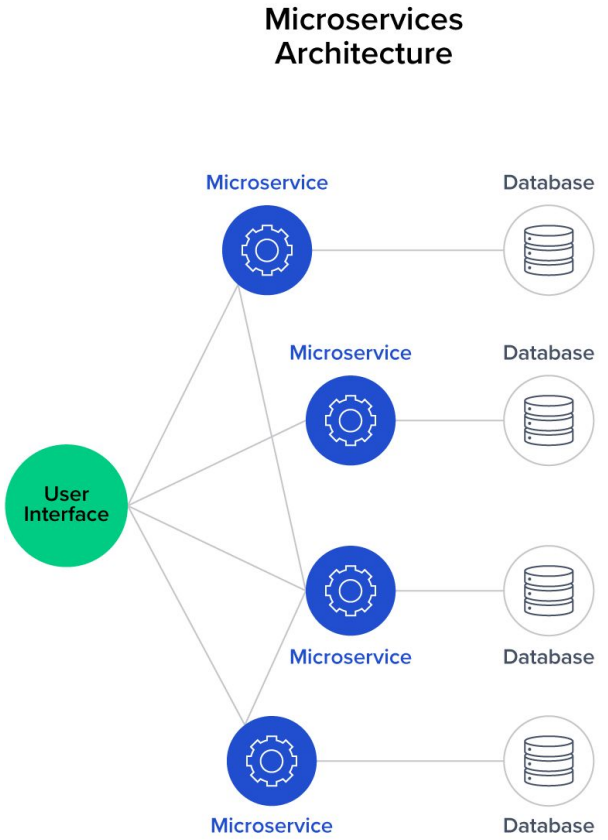


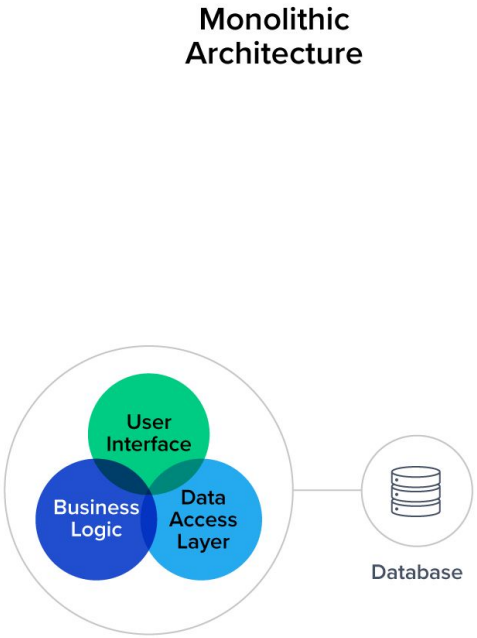


Apache Kafka 101

En sistemas backend tradicionales, la arquitectura de **microservicios,** utiliza fuertemente los message brokers, el problema consiste en sincronizar bases de datos.



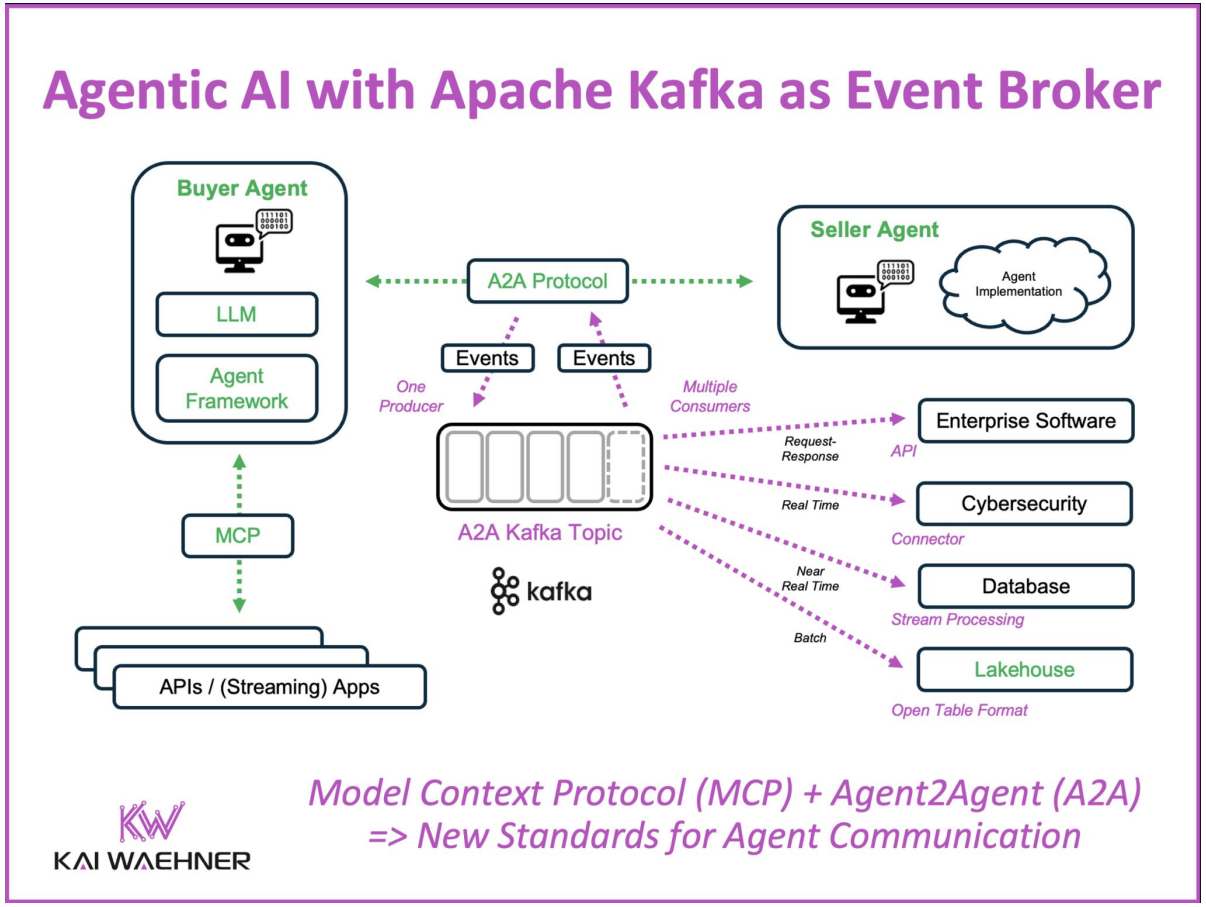




Apache Kafka 101

El message broker realiza streaming de eventos a todos aquellos suscritos a un topic.

En sistemas de Agentes, Kafka es parte del data layer, emulando microservicios mediante A2A, un agente puede comunicarse al tener acceso al topic de kafka de manera distribuida y desacoplada a otro agente



[Agentic AI in FinTech with Apache Kafka]

[Agentic AI with the A2A and MCP using Apache Kafka as Event Broker]

[Confluent Kafka MCP]

[Confluent Kafka MCP Github]

[Confluent Streaming Agents]

Agentes: Data layer, Apache Kafka (Real-time)

Research on the Application of Spark Streaming Real-Time Data Analysis System and large language model Intelligent Agents

Observabilidad: OpenTelemetry

Observabilidad es clave no solamente para agentes sino el software en general

AI Agent Observability
GCP: Instrument a LangGraph ReAct Agent with OpenTelemetry
LangSmith OTel

LangFuse Observability

LangFuse vs LangSmith

Resumen

Los sistemas de agentes implica muchos pasos pero los mayores cuellos de botella en aplicaciones de agentes están en el **agent layer y data layer siendo**:

- LLMs utilizadas,
- Arquitectura del sistema agente,
- Tooling/MCP
- Data layer

Langchain how do I speed up my AI Agent

How to Scale Your LangGraph Agents in Production From A Single User to 1,000 Coworkers

Como atacar el problema?

Empezando por la aplicación y los requerimientos, si es una organización grande con grandes cantidades de data, el **data layer es el primer paso**, esto puede implicar aportes de diversos equipos y desarrollo de software/connectors custom.

Ejemplo: microservicios gRPC, REST, DBs SQL/NoSQL, Kafka Topics, se requerirá desarrollo para exponerlos a través de MCP o por lo menos tooling de manera **segura con Agentes.**

Una vez identificado el data layer, se puede arquitectar el sistema mediante **N Agentes especializados**, A2A (en sistemas distribuidos) o arquitecturas simples pero poderosas como supervisor.

Preguntas?