Optimización de costos para transacciones de alto volumen

Franchesco Romero

AWS Community Builder &
AWS User Group Leader

Intro

- Explorar estrategias no *tan* convencionales de optimización de costos en AWS

La Economía Oculta de las API Calls

- Arquitecturas serverless:
  - Más llamadas API = mayor costo
  - Elección de tecnología

La Economía Oculta de las API Calls

- 📌 Caso ineficiente (Polling con API Gateway y Lambda)
  - Una aplicación consulta cada 5 segundos API para verificar nuevas órdenes en DynamoDB.
  - Problema: N cantidad de requests REST innecesarias generan costos elevados en API Gateway y Lambda.
- ✅ Optimización (EventBridge + DynamoDB Streams)
  - Configurar DynamoDB Streams para detectar cambios y enviarlos a EventBridge filtra y transforma el evento antes de enviarlo a SNS, Step Functions o WebSockets.

La Economía Oculta de las API Calls

- Transformaciones vs. Lambda
  - 📌 Caso ineficiente:
    - Un frontend envía peticiones JSON a API Gateway, y este siempre ejecuta una Lambda para validar datos y enviarla a DynamoDB.
    - Problema: Cada llamada a Lambda tiene un costo adicional.
  - ✅ Optimización:
    - Usar Mapping Templates en API Gateway para hacer validaciones y transformar la petición sin necesidad de Lambda.
    - Ahorro: Se eliminan cientos o miles de invocaciones a Lambda.

La Economía Oculta de las API Calls

- Límites internos de AWS:
  - 📌 Caso ineficiente (Límite de Throughput en DynamoDB)
    - Una aplicación escribe datos en DynamoDB a alta velocidad.
    - Cuando se alcanza el límite de WCU (Write Capacity Units), AWS aplica backoff exponencial y genera reintentos automáticos.
  - ✅ Optimización (Uso de Amazon SQS como Buffer)
    - En lugar de escribir directamente en DynamoDB, los datos se envían primero a Amazon SQS -> Lambda -> DynamoDB

Optimizaciones para Servicios Serverless

- Lambda: Empaquetado avanzado y optimización de dependencias
- Compilación de dependencias nativas para ARM
- Step Functions: Standard vs. Express
- API Gateway: Regional vs. Edge-optimized para tráfico global
- EventBridge: filtros avanzados vs. procesamiento en Lambda

Data Transfer

- VPC Endpoint vs. NAT Gateway
  - 📌 Caso ineficiente (Usar NAT Gateway para acceder a S3)
    - Una aplicación en EC2 dentro de una VPC privada necesita acceder a Amazon S3 para descargar archivos.
    - NAT Gateway cobra $0.045/GB por salida de datos.
    - Para 5TB/mes → $225 adicionales solo por NAT Gateway.
  - ✅ Optimización (Usar VPC Endpoint para S3)
    - El tráfico a S3 ahora fluye directamente dentro de la red de AWS, sin costo por transferencia de datos.

Data Transfer

- Estrategias de Caché regional y CloudFront para APIs
  - 📌 Caso ineficiente:
    - Un sitio web consulta una API con datos de productos en API Gateway + Lambda.
  - ✅ Optimización: CloudFront con API Gateway
    - Configurar reglas para que CloudFront almacene respuestas comunes por 5 minutos.

Data Transfer

- LLM-Proxying y Data Gravity para cargas de GenAI
  - 📌 Caso ineficiente:
    - Un chatbot en AWS envía cada consulta de usuario a un modelo LLM alojado externamente (ej. OpenAI, Hugging Face Inference API).
    - Cada request requiere transferencia de datos entre AWS y el proveedor externo.
  - ✅ Optimización: Usar un LLM-Proxy en AWS
    - En lugar de hacer consultas directas a un modelo externo, se puede alojar un LLM Proxy en Amazon Bedrock o SageMaker.

La Economía del Almacenamiento

- Compresión de datos
- 📌 Caso ineficiente:
  - Logs de transacciones se almacenan en Redshift/EMR/Glue en formato GZIP.
  - GZIP tiene alta latencia de descompresión en Redshift.
  - GZIP usa más espacio que Zstandard.

- ✅ Optimización: Usar Compresión Zstandard
  - Usar compresión Zstandard en nivel 3 para mejor eficiencia de almacenamiento y procesamiento. (Reducción +30%)

La Economía del Almacenamiento

- Zero-copy data sharing entre cuentas
  - 📌 Caso ineficiente:
    - Usar AWS Glue Data Catalog para registrar datos en S3 y copiarlos entre cuentas para que cada equipo acceda a su propia copia.
  - ✅ Optimización:
    - Usar AWS Lake Formation con Glue Catalog Cross-Account
    - Se configura Lake Formation para que otras cuentas puedan acceder a las tablas registradas en AWS Glue sin copiarlas.

Computación Selectiva

- CPU-gravity vs. Memory-gravity vs. IO-gravity
  - 📌 Caso ineficiente
    - Una base de datos en Amazon RDS MySQL usa instancias t3.medium con 8GB RAM.
    - La RAM se llena rápidamente → swapping a disco (mayor latencia).
    - CPU no es el cuello de botella, sino la memoria.
  - ✅ Optimización: Usar Instancias R6g (ARM) o X2gd
    - Instancias R6g (Graviton2) y X2gd ofrecen 50% más memoria por dólar que T3.
    - ARM (Graviton) vs. x86 / ARM -> Microservicios, crypto -> x86

Computación Selectiva

- 📌 Caso ineficiente:
  - Una Lambda que procesa imágenes usa 128MB de RAM y tarda 6 segundos en ejecutarse.
  - Poca RAM = Poca CPU = Ejecución más lenta = Más costo por tiempo de ejecución.
- ✅ Optimización: Aumentar la Memoria para Obtener Más CPU
  - 128MB RAM → 1/6 vCPU → Tarda 6s
  - 1024MB RAM → 1 vCPU → Tarda 0.8s (7 veces más rápido)
- PowerTools

Computación Selectiva

- 📌 Caso ineficiente:
  - Un clúster de Elasticsearch en EC2 usa almacenamiento estándar en EBS gp3.
  - Alto uso de IOPS → Límite de rendimiento en EBS gp3.
  - Más latencia en consultas y búsquedas.

- ✅ Optimización:
  - Migrar a Instancias I4i o Usar EBS io2 Block Express
  - Instancias I4i con almacenamiento NVMe ofrecen 60% menos latencia que EBS gp3.

Contenedores

- 📌 Caso ineficiente:
  - En un clúster de Amazon EKS, cada microservicio se ejecuta en su propio pod con un límite de CPU y memoria dedicado.
  - Mayor número de pods = más nodos en el clúster = más costos en EC2 o Fargate.

- ✅ Optimización:
  - Consolidar Servicios en Pods Compartidos (Multi-Tenant Pods)
  - En lugar de usar un pod por servicio, se agrupan servicios relacionados en un solo pod.

Estrategias de Bases de Datos

- 📌 Caso ineficiente:
  - Una base de datos en Amazon Aurora con 5TB
  - Crecimiento automático de almacenamiento genera costos imprevistos.
  - Backups innecesarios ocupan más espacio.
- ✅ Optimización: Configurar Auto-Tiering y Aurora I/O-Optimized
  - Paso 1: Usar Aurora I/O-Optimized para cargas intensivas en lectura/escritura.
  - Paso 2: Configurar auto-tiering para mover datos inactivos a almacenamiento más barato.
  - Paso 3: Habilitar Aurora Backtrack para reducir snapshots innecesarios.

Estrategias de Bases de Datos

- 📌 Caso ineficiente:
  - Logs de transacciones en una base de datos PostgreSQL en RDS.
  - Con el tiempo, la tabla crece a millones de registros, volviendo las consultas lentas.
- ✅ Optimización:
  - Sharding por Intervalos de Tiempo con Amazon TimeStream o PostgreSQL Partitioning
  - En lugar de usar una sola tabla gigante, dividir los datos en particiones por mes o día.

Estrategias de Bases de Datos

- 📌 Caso ineficiente:
  - Una base de datos en Amazon RDS MySQL maneja tanto escrituras como lecturas con una sola instancia.
  - Costo alto en una sola instancia escalada verticalmente.
- ✅ Optimización: Separar Cargas de Escritura y Lectura
  - Usar Amazon Aurora con Writer Node dedicado.
  - Optimizar índices y uso de batch inserts.
  - Habilitar Aurora Replicas o Read Replicas en RDS.
-

Estrategias de Bases de Datos

- DynamoDB: DAX, TTL y Streams como herramientas de optimización

Estrategias en Networking para Optimización

- Transit Gateway vs. PrivateLink
- Arquitecturas de "locality-first" para reducir costos de networking
- Elastic IP vs. DNS-routing dinámico

Arquitecturas Multi-Cuenta para Optimización de Costos

- Estrategias avanzadas de Reserved Instance sharing
- VPC Sharing vs. VPC Peering
- Consolidación de NAT Gateways y servicios compartidos

Tecnologías Alternativas

- ElastiCache Redis vs. MemoryDB
- EventBridge Pipes vs. Step Functions
- RDS Proxies
- ALB vs. CloudFront Functions para routing de APIs

Conclusiones

Monitoreo de Costos

- Anomaly Detection para patrones de costos no lineales
- Budget alerts

Anti-Patterns: Lo que NO Debes Hacer

- Over-engineering en arquitecturas "event-centric"
- El mito de "serverless siempre es más económico"
- Database denormalization solo por performance
- Monitoreo excesivo y logs sin filtrar
- Sacrificar operabilidad por ahorros marginales

Estrategias Anti-Patterns: Lo que NO Debes Hacer

- Las optimizaciones no convencionales pueden aportar 15-30% adicional de ahorro
- La optimización es un proceso continuo, no un proyecto
- Crear cultura de cost awareness a nivel de arquitectura y desarrollo

Gracias!

Franchesco Romero

linkedin.com/in/elchesco