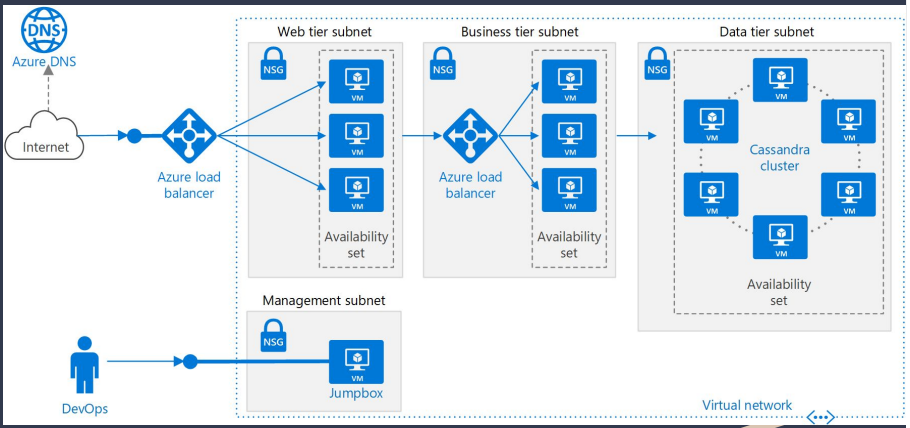
Cloud Computing Usage Patterns

El punto de vista del cloud consumer

1er cuatrimestre 2023

Cloudgenda



- **Repaso intro a cloud**
- **Ejemplo Cloud Provider: AWS**
- **Estilos de Arquitectura habituales**
  - N-tier
  - Web-Queue-Worker
  - Microservices
  - Event-Driven
- **Usage Patterns**
  - Auto-Scaling
  - Health Endpoint
  - Competing Consumers
  - Queue-Based Load Leveling
  - Priority Queue
  - Throttling
  - Retry
  - Circuit Breaker
  - Sharding
- **Algunas arquitecturas de referencia AWS**
  - Web App
  - Batch
  - Media sharing

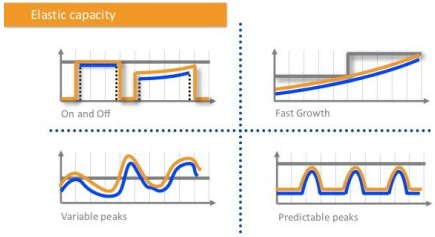
Repaso

Definición

*Forma especializada de **computación distribuída**, que introduce **modelos de utilización** para la **provisión remota** de **recursos escalables y medibles***

Características

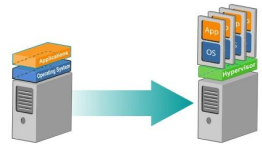
1. On-Demand usage
2. Ubiquitous Access
3. Multitenancy
4. Elasticity
5. Measured Usage
6. Resiliency

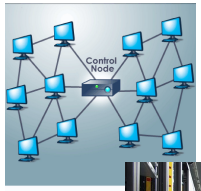


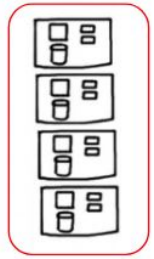
Delivery Models

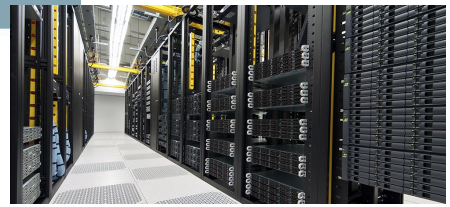
Tecnologías

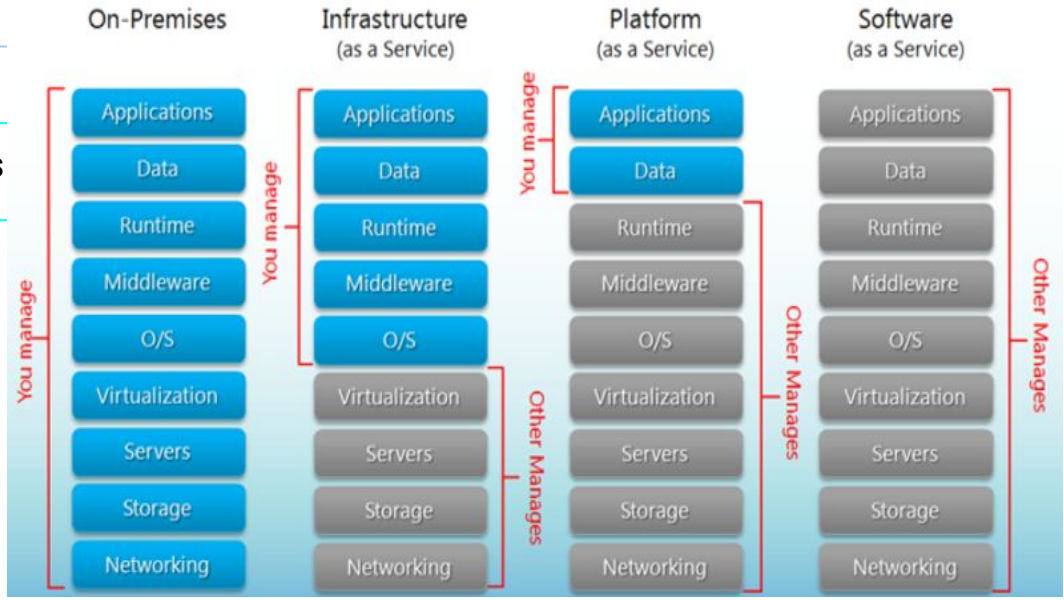
1. Virtualización
2. Clustering
3. Grid Computing
4. Internet
5. Web
6. Data Centers
7. Multitenant
8. Services











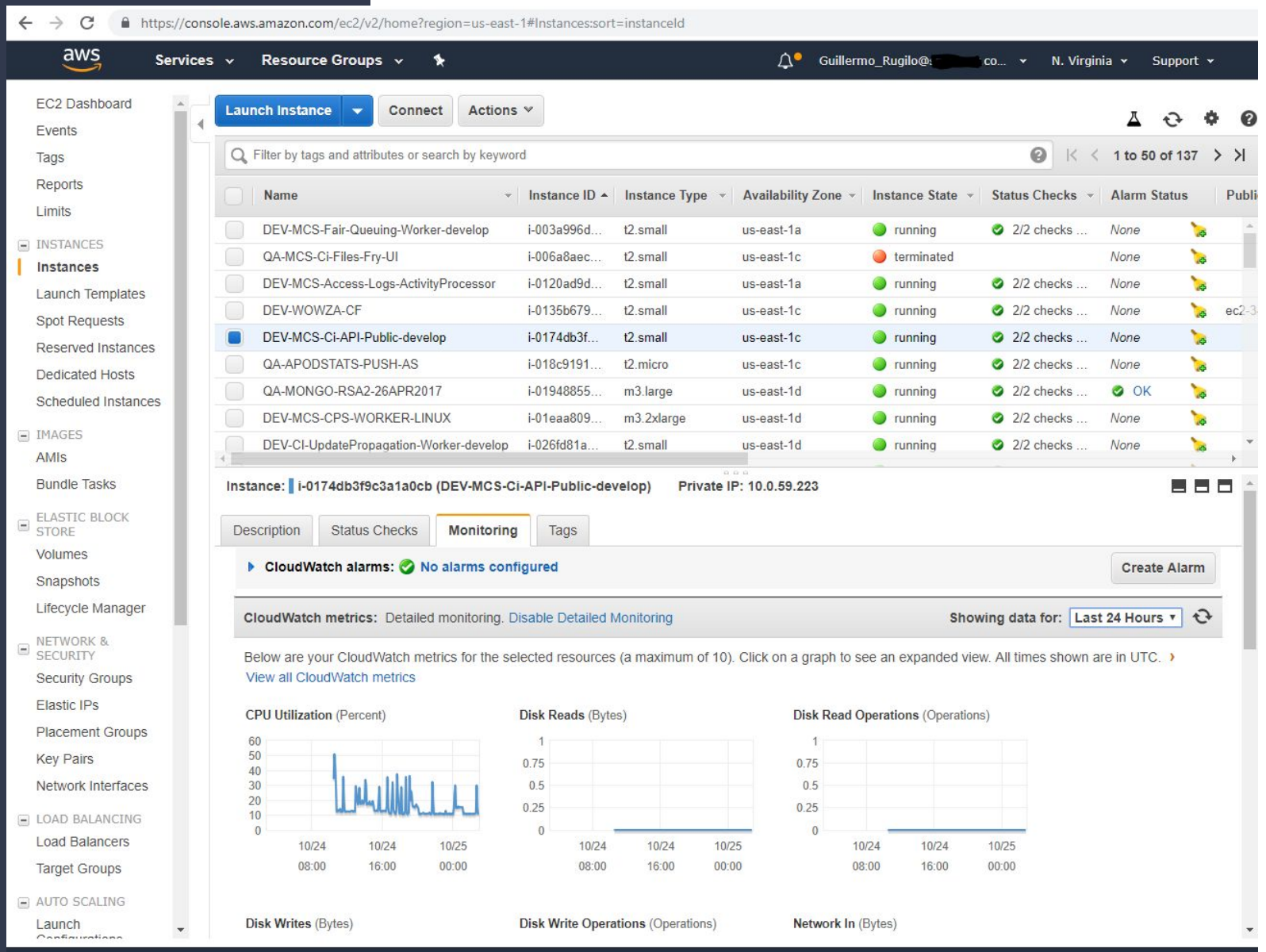
Ejemplo de Cloud Provider:

AWS - Amazon Web Services

AWS

EC2 Console

Provisión remota de recursos escalables y medibles



AWS EC2 On-Demand Pricing

Modelos de utilización

Linux | RHEL | SLES | Windows | Windows with SQL Standard | Windows with SQL Web

Windows with SQL Enterprise | Linux with SQL Standard | Linux with SQL Web | Linux with SQL Enterprise

Region: US East (N. Virginia)

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
t3.nano	2	Variable	0.5 GiB	EBS Only	$0.0052 per Hour
t3.micro	2	Variable	1 GiB	EBS Only	$0.0104 per Hour
t3.small	2	Variable	2 GiB	EBS Only	$0.0208 per Hour
t3.medium	2	Variable	4 GiB	EBS Only	$0.0416 per Hour
t3.large	2	Variable	8 GiB	EBS Only	$0.0832 per Hour
t3.xlarge	4	Variable	16 GiB	EBS Only	$0.1664 per Hour
t3.2xlarge	8	Variable	32 GiB	EBS Only	$0.3328 per Hour
t2.nano	1	Variable	0.5 GiB	EBS Only	$0.0058 per Hour
t2.micro	1	Variable	1 GiB	EBS Only	$0.0116 per Hour
t2.small	1	Variable	2 GiB	EBS Only	$0.023 per Hour
t2.medium	2	Variable	4 GiB	EBS Only	$0.0464 per Hour
t2.large	2	Variable	8 GiB	EBS Only	$0.0928 per Hour
t2.xlarge	4	Variable	16 GiB	EBS Only	$0.1856 per Hour
t2.2xlarge	8	Variable	32 GiB	EBS Only	$0.3712 per Hour
m5.large	2	8	8 GiB	EBS Only	$0.096 per Hour
m5.xlarge	4	16	16 GiB	EBS Only	$0.192 per Hour

AWS

Muchísimos cloud services

Screenshot tomado 31/10/2019

AWS Billing Console

Modelos de utilización

Monthly Spend

Welcome to the AWS Account Billing console. Your current monthly balance appears below. The accompanying graph shows the proportion of costs spent for each service you use.

Current month-to-date balance for December 2013

$3,493.57

$69.03
Previous month bill

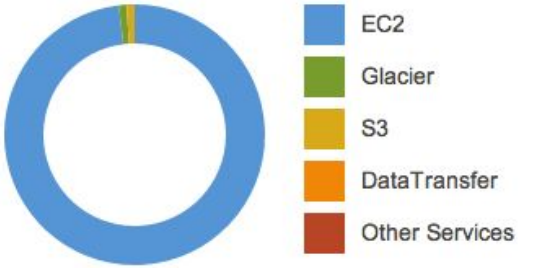
Include Subscription Charges

Alerts & Notifications

- Welcome to the redesigned Billing Console. The previous Account Activity and DevPay page can be accessed here.
- Monitor your estimated charges. Enable Now to begin setting billing alerts that automatically e-mail you when charges reach a threshold you define.
- IAM access to your account's billing information is not enabled. You can enable it on the Account Information page.

By Service

Bill Details



Top Services	Amount
EC2	$3,493.02
Glacier	$0.49
S3	$0.06
DataTransfer	$0.00
Other Services	$0.00
Tax	$0.00
Total	**$3,493.57**

Estilos de arquitectura habituales en Cloud

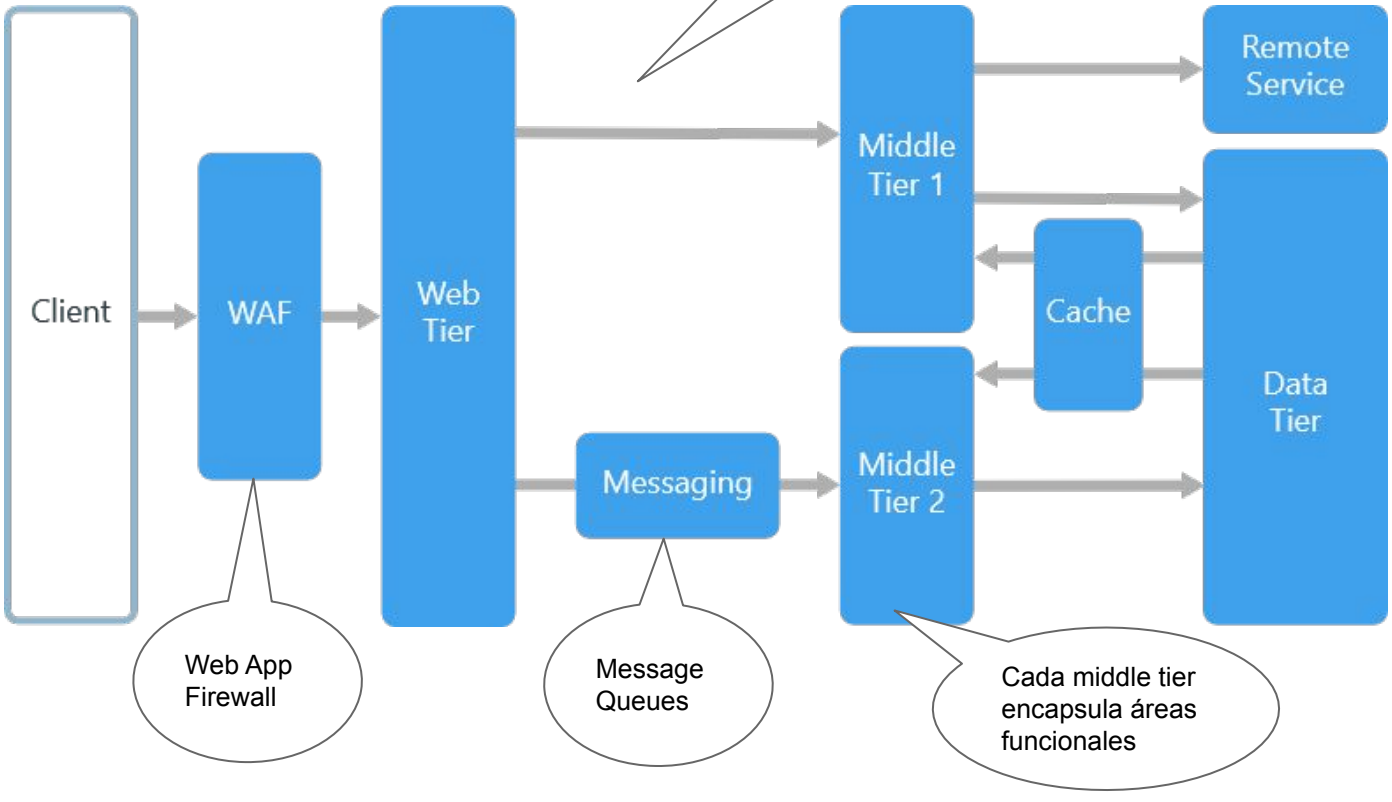
N-tier

Las tiers son capas físicamente separadas

Las tiers pueden ser grupos de VMs manejadas por nosotros (IaaS) o servicios del cloud provider

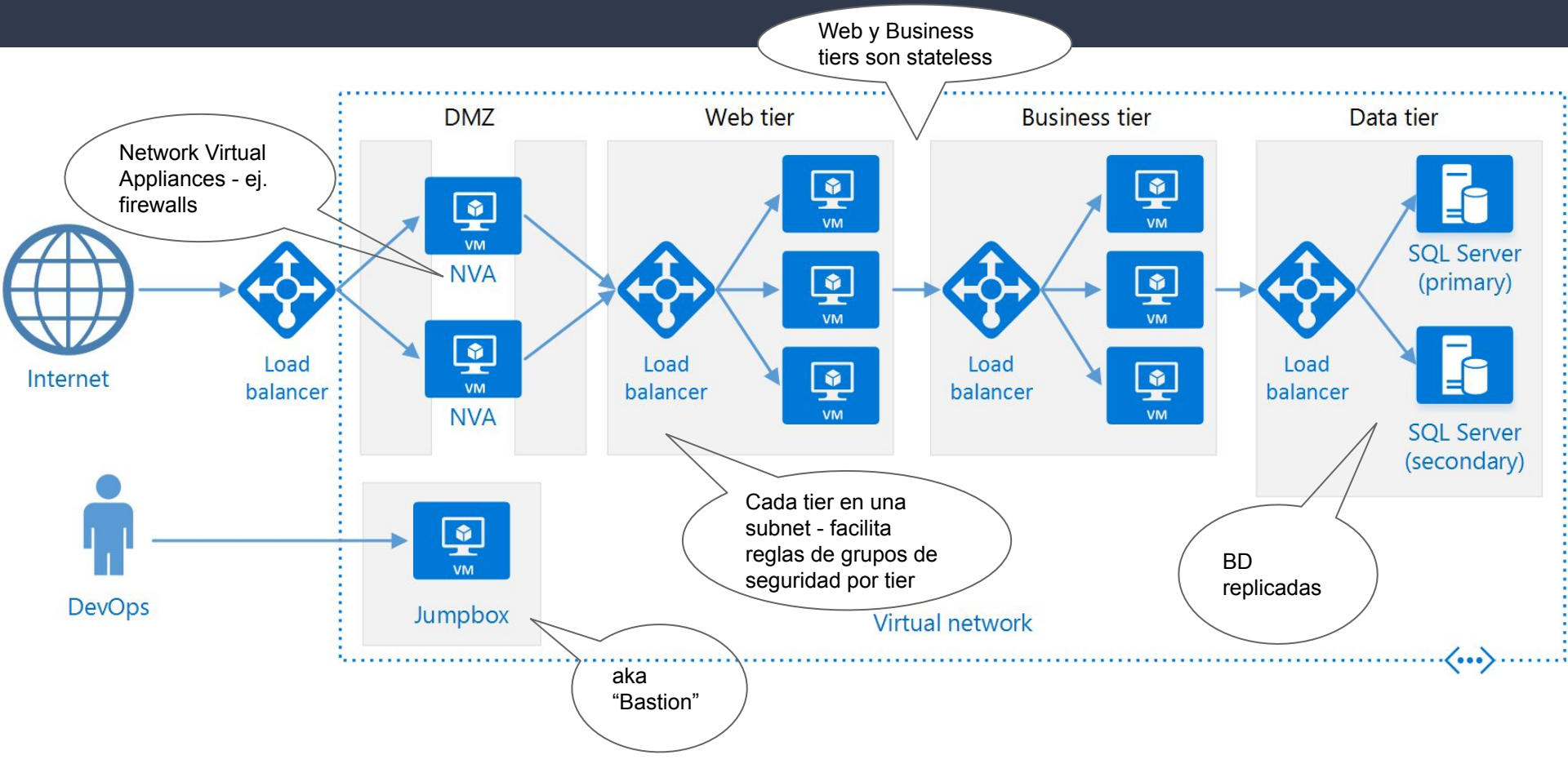
Layered System

Cada tier se puede **escalar horizontalmente**



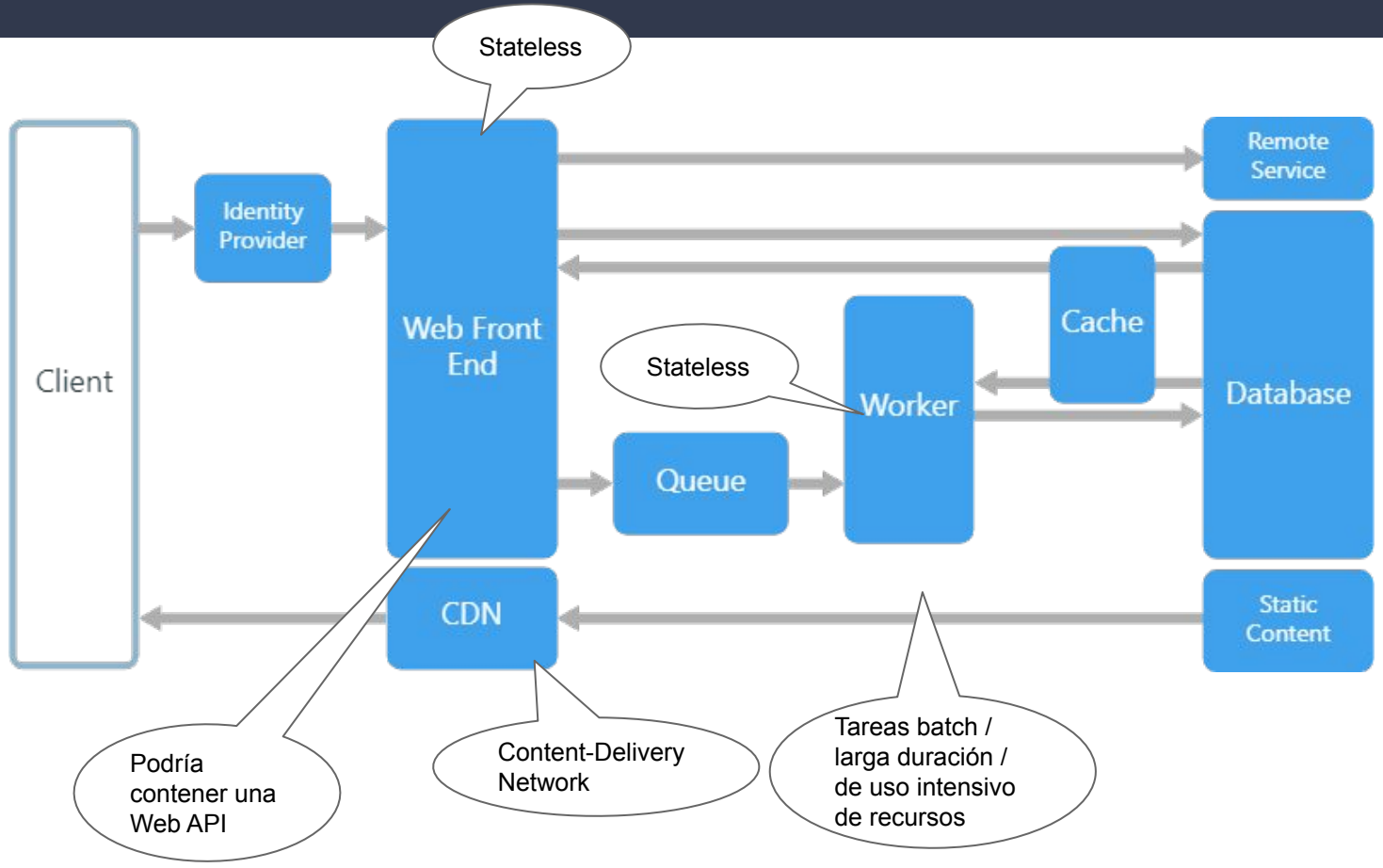
→

N-tier usando IaaS en Azure

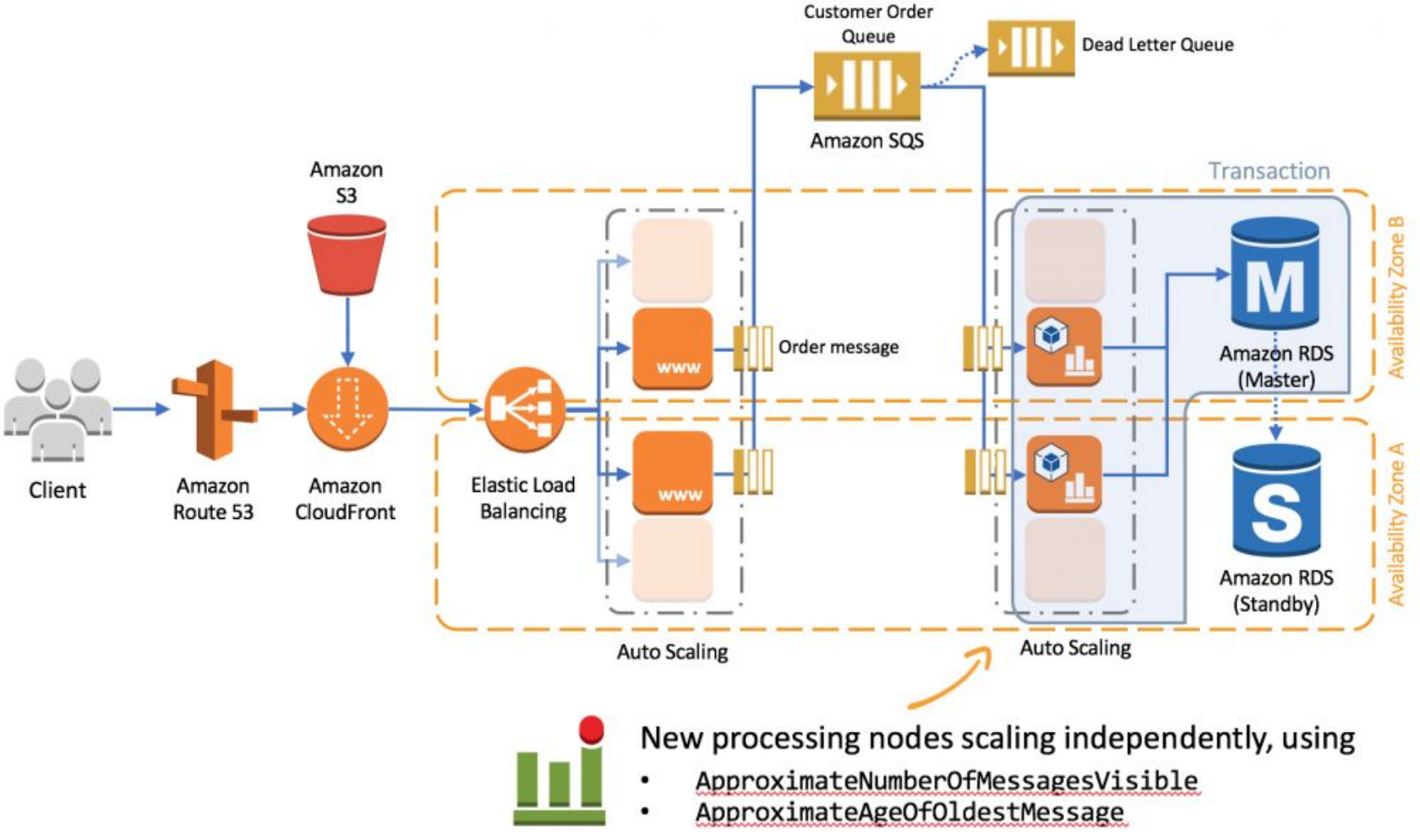


Web-Queue-Worker

Front end y worker se podrían **escalar horizontalmente**



AWS Web-Queue-Worker App



Event-driven

Event Based Integration (EBI)

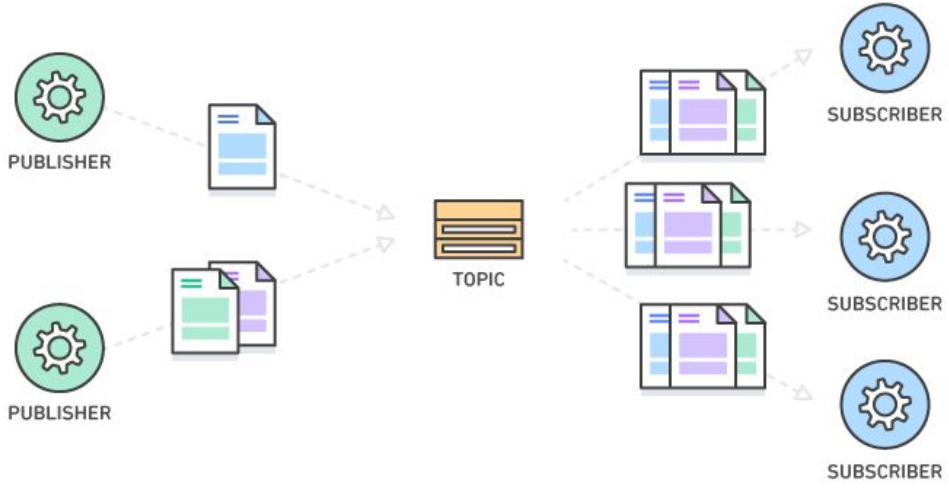
Producers desacoplados de consumers

Consumers desacoplados entre sí

Cada consumer ve todos los eventos

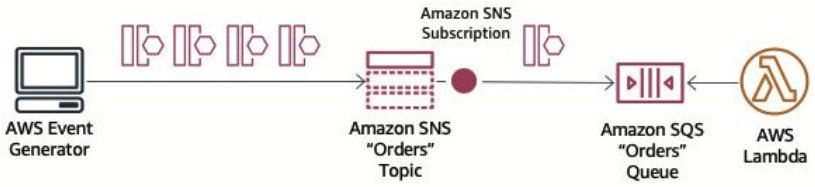
Modelos

- **Pub/sub**: eventos no se pueden volver a reproducir (replay). Nuevos subscriptores no ven eventos anteriores.
- **Event Streaming**: se escriben en un log, ordenados, y persisten. Consumers leen de cualquier parte del stream. Hay replay.

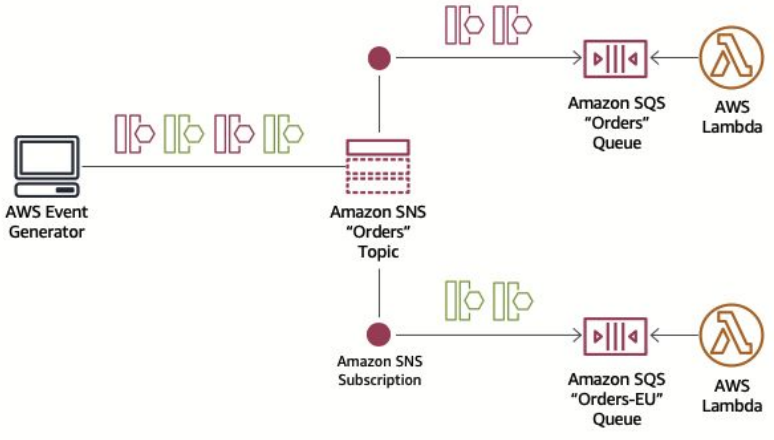


Ejemplos AWS: Pub/Sub

Simple Pub/Sub

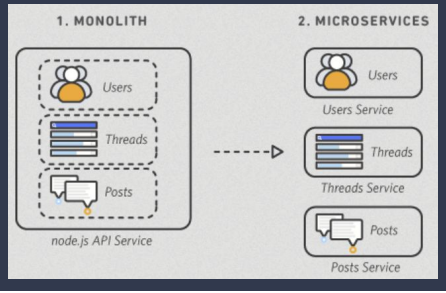


Pub/Sub con filtrado de mensajes



Microservices

Servicios autónomos, pequeños, y desacoplados, cada cual implementa un área funcional de negocio



Cada microservice suele
- tener su propio codebase separado
- ser manejado por un pequeño grupo de dev
- ser deployado de forma independiente
- encargarse de su propio storage

Los microservices se comunican a través de APIs

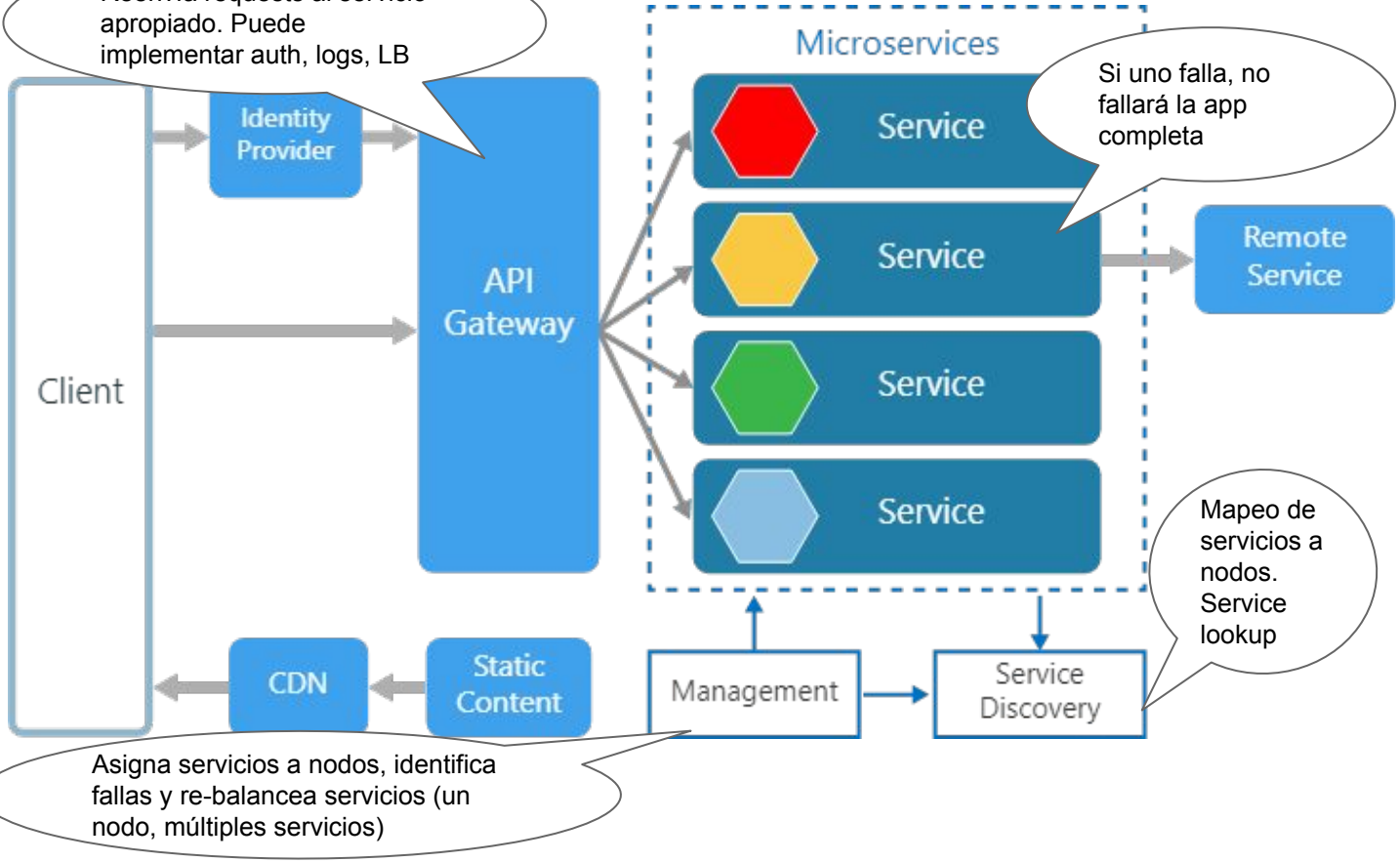
No necesariamente están implementados con la misma tecnología

Reenvía requests al servicio apropiado. Puede implementar auth, logs, LB

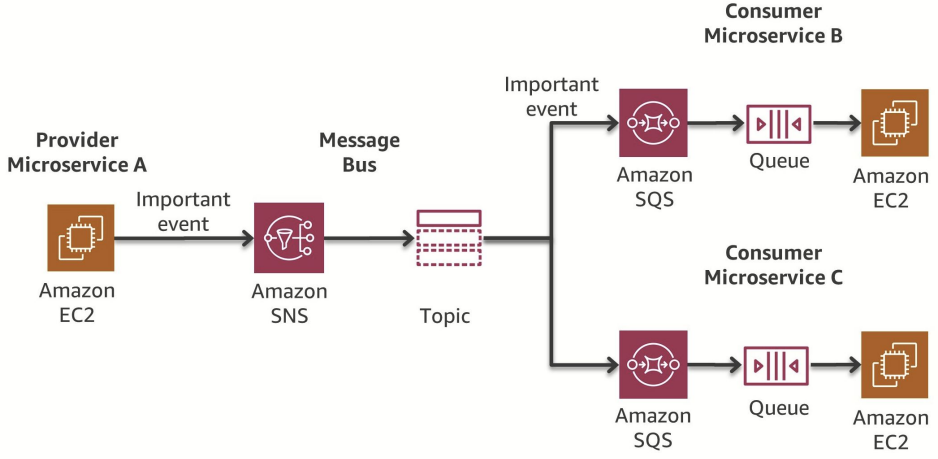
Si uno falla, no fallará la app completa

Mapeo de servicios a nodos. Service lookup

Asigna servicios a nodos, identifica fallas y re-balancea servicios (un nodo, múltiples servicios)



Ejemplo AWS: comunicación async entre Microservices



Comunicación entre Microservices

1. Sincrónica (llamados directos a APIs)
2. Asincrónica
   - Queues
   - EBI (aka “Message Bus pattern”)

- Servicios desacoplados
- No es necesario un Service Discovery (aka Service Directory)

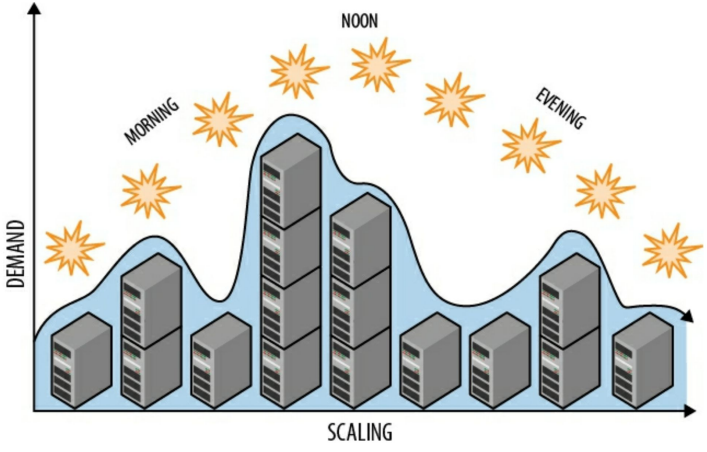
Cloud usage Patterns

¿Cómo implementar “Match Strategy” de manera de optimizar el costo lo máximo posible y cumplir con la demanda?

Auto-Scaling

Escalar manualmente usando la consola del cloud provider es posible y útil, pero:

- Conlleva esfuerzo manual rutinario
- Requiere administrador disponible
- El costo podría optimizarse más con una escalabilidad automática



Disponer un proceso de asignación dinámica de recursos (nodos, storage, queues, etc.) para implementar “match strategy”

- Por lo general, hablamos de **Horizontal Scaling**
- Reglas para eventos conocidos
  - *A las 7PM, 2 web servers.*
  - *A las 7AM, 10 web servers.*
  - *A las 7PM de los días viernes, 1 web server.*
- Reglas para reaccionar a eventos del ambiente

*“Si el **% promedio de CPU** utilizado **durante la última hora** es **mayor a 50**, **incrementar en 1** la cantidad de instancias **web server** del pool”*

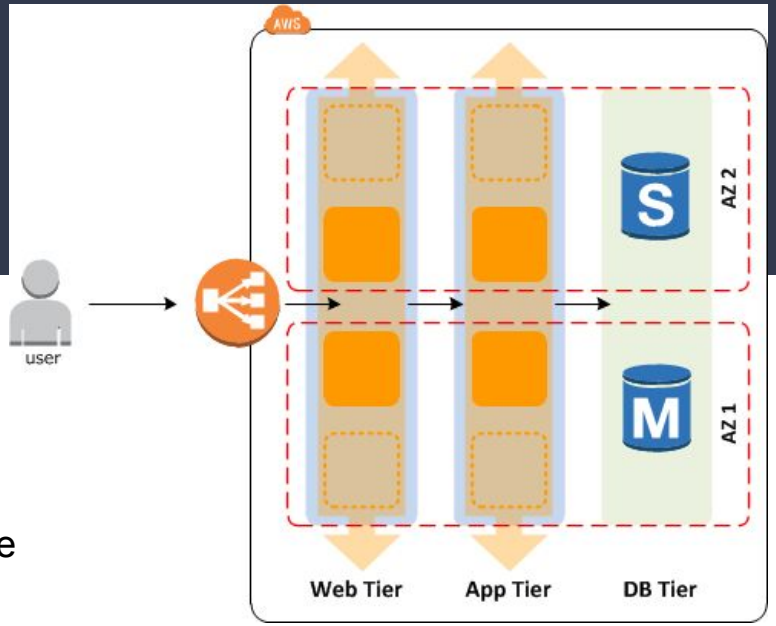
*“Si la cantidad **promedio de mensajes en la queue** en los **últimos 20 minutos** es **menor a 5**, **decrementar en 1** la cantidad de **workers**”*

*“Si el **tiempo promedio para confirmar la orden de un cliente** en la **última hora** es **mayor a 10 minutos**, **incrementar en 1** la cantidad de **workers**”*

→

Auto-Scaling

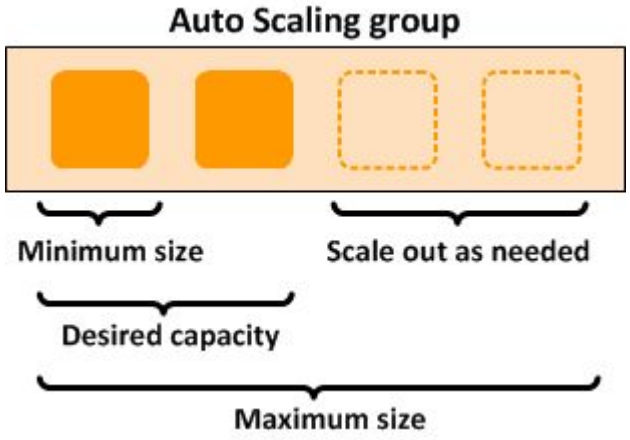
Consideraciones para la implementación



El sistema debe ser escalable horizontalmente
Evitar instance affinity, implementar Stateless

Qué necesitamos

- Scale out/in APIs del cloud provider
- Sistemas de monitoreo que capturen las métricas clave
- Lógica que evalúe las métricas respecto a las reglas predefinidas, y decidan si escalar
- Testing y tuning de la estrategia de auto-scaling

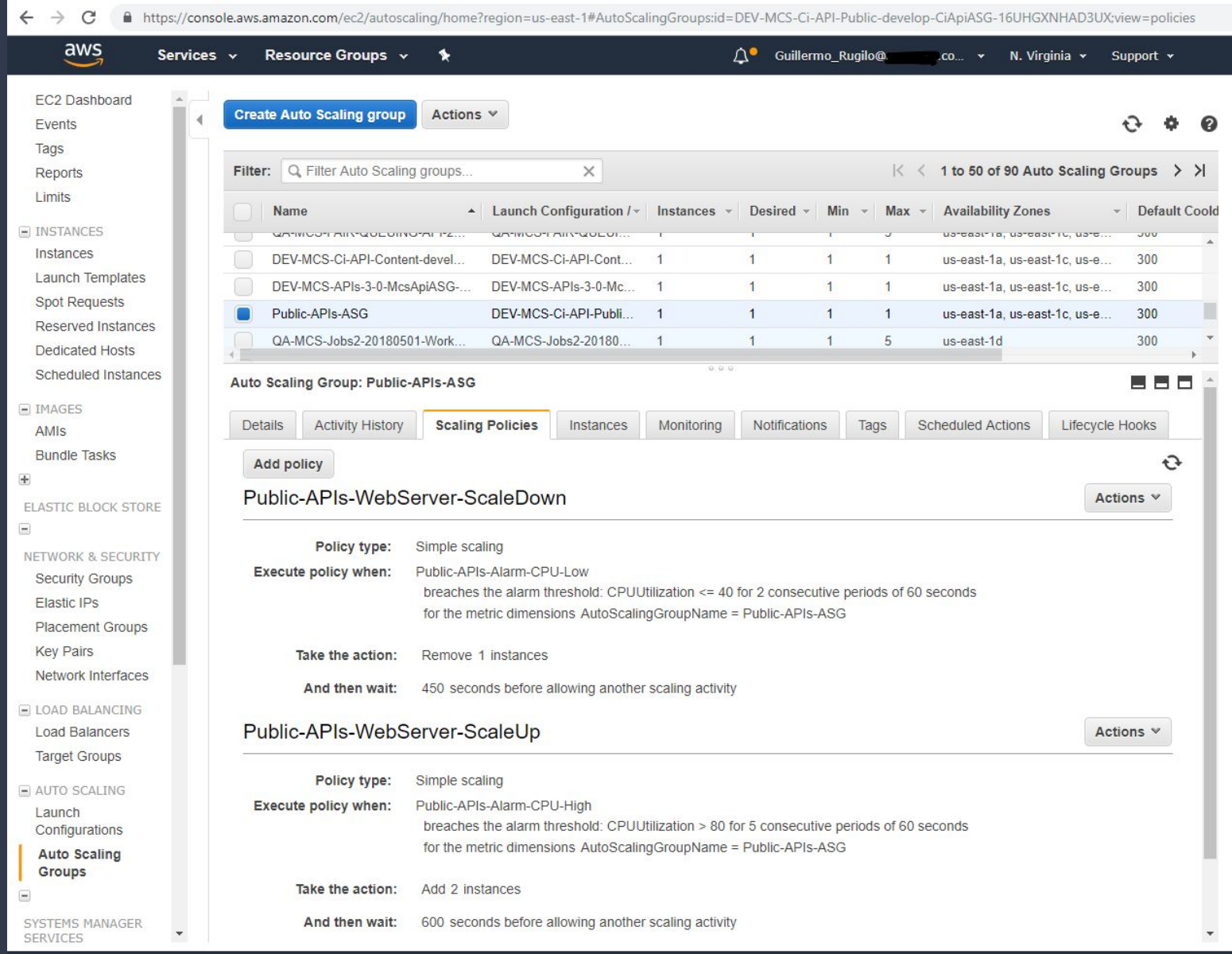


Buenas prácticas

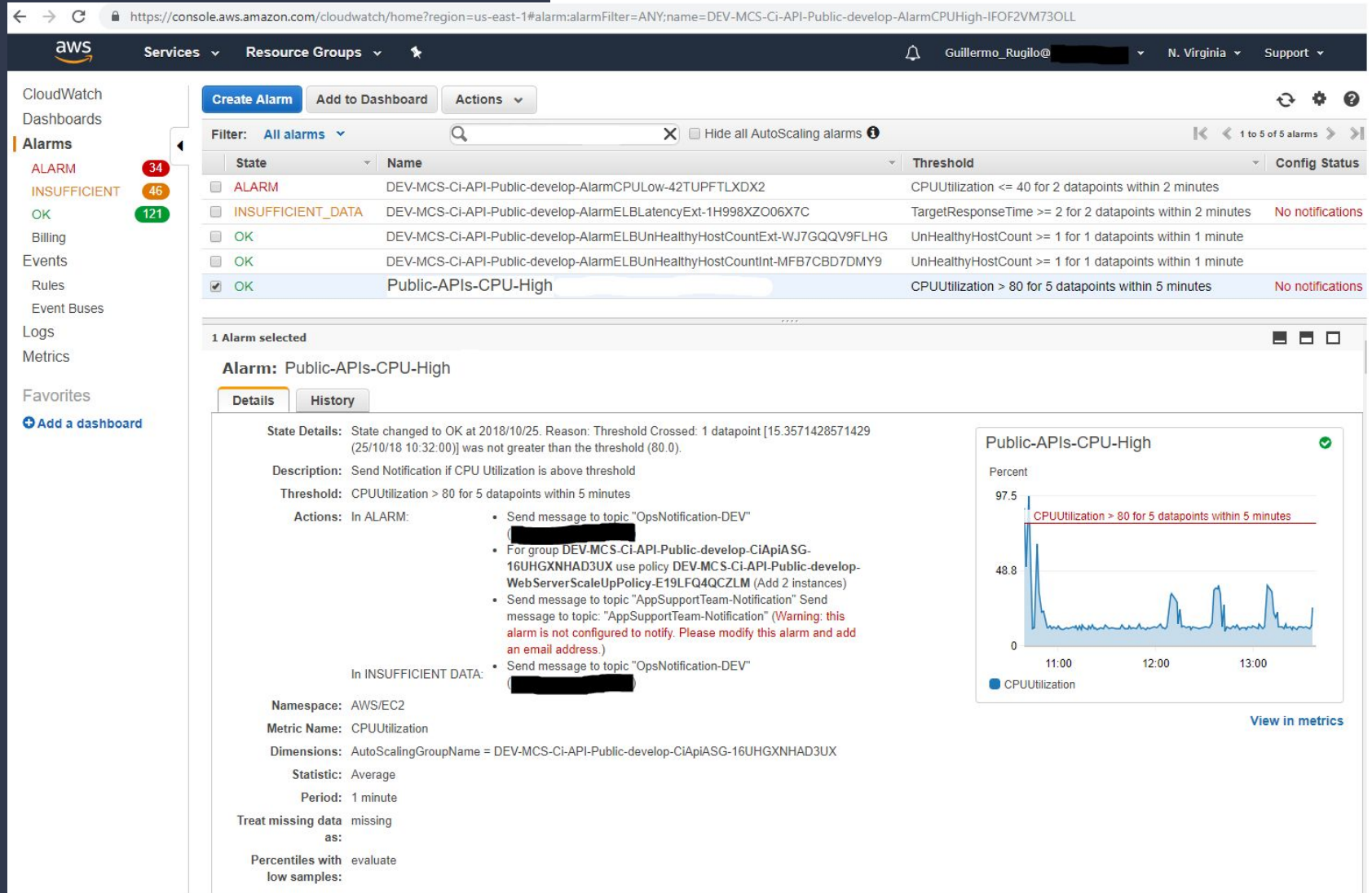
- Utilizar las funcionalidades de auto-scaling del cloud provider; solución custom requiere esfuerzo significativo
- Escalar cada componente / subsistema de forma independiente
- Poner **límites inferiores y superiores** en las reglas
- Long-running tasks pueden causar problemas
  - Refactorizar hacia Pipes and Filters
  - Usar checkpoints en almacenamiento externo

AWS Auto Scaling Groups

Reglas de Auto Scaling

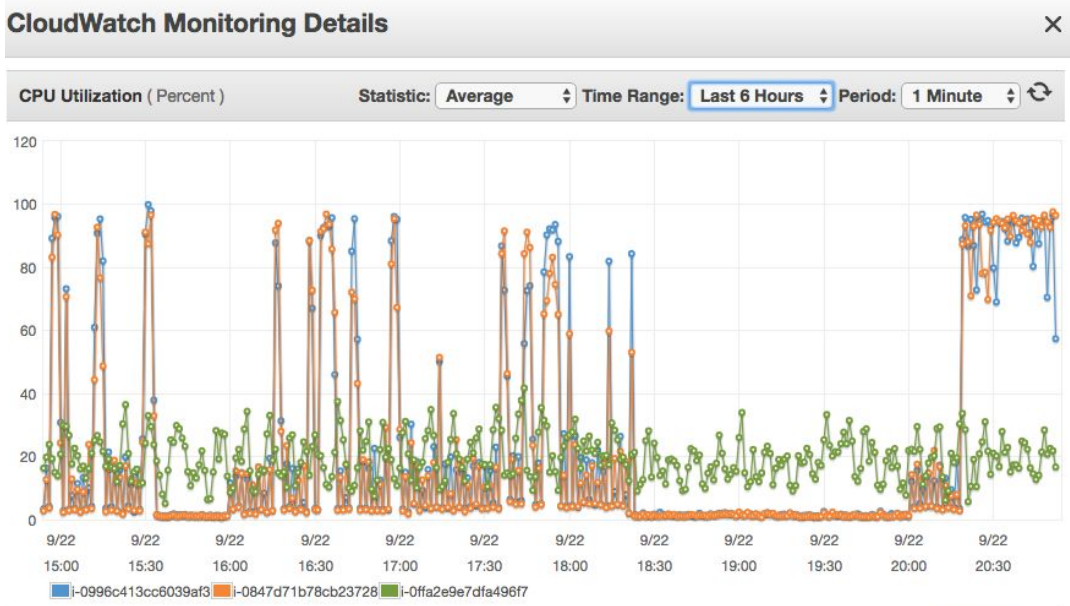


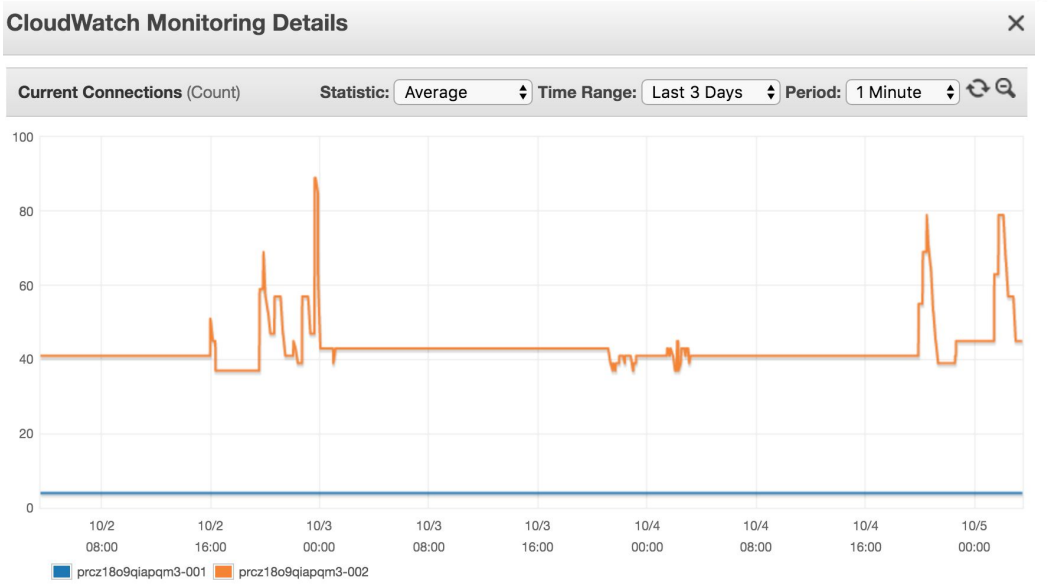
AWS CloudWatch



AWS CloudWatch

Análisis de métricas



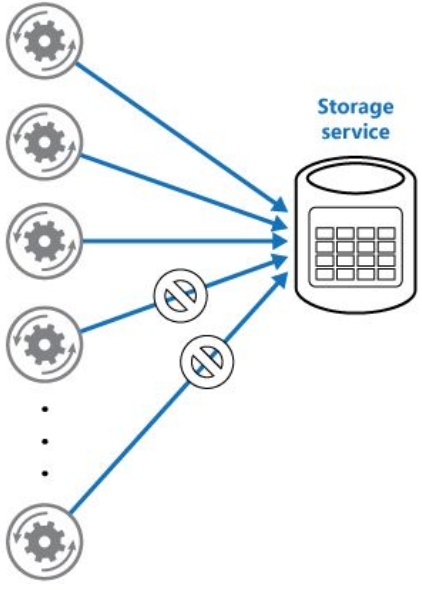


Availability, Performance, Scalability

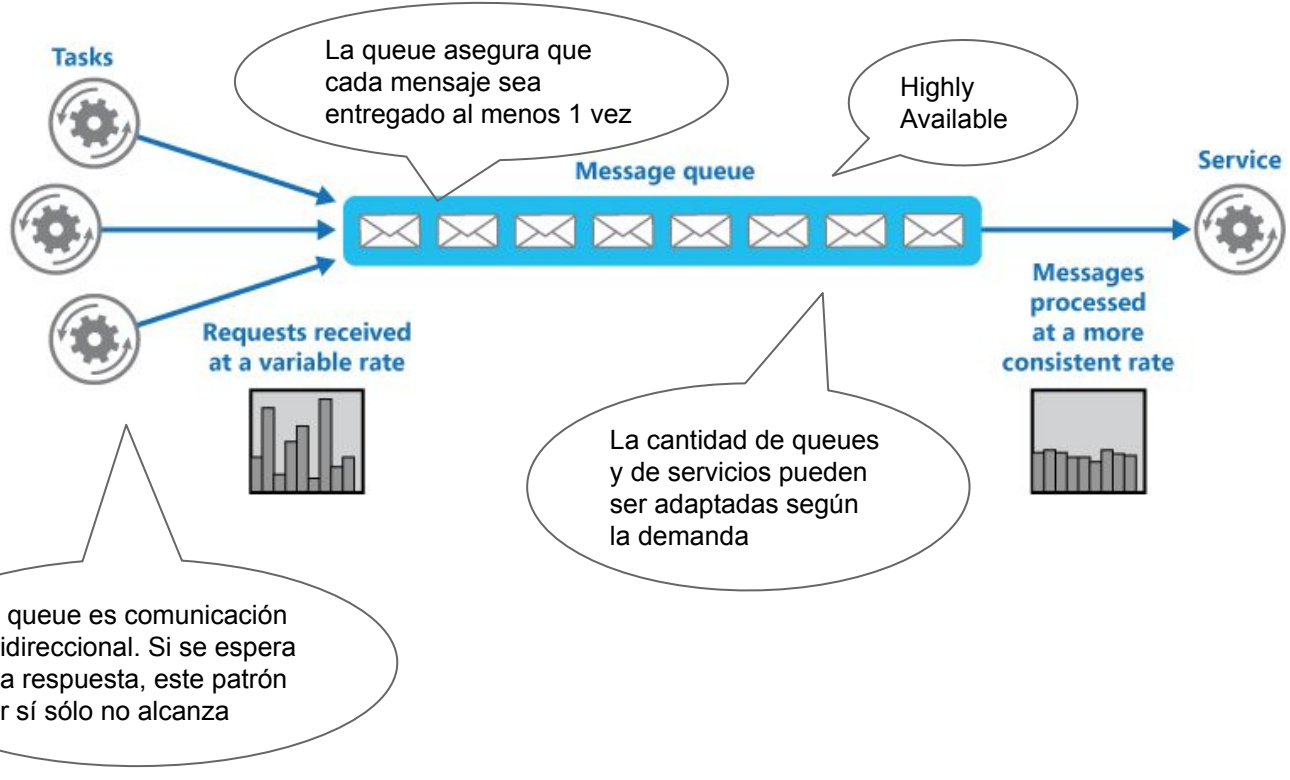
¿Cómo evitar la sobrecarga de un servicio sujeto a cargas grandes e **intermitentes**?

Queue-Based Load Leveling

Picos en la demanda, muchos requests concurrentes



Introducir una cola de mensajes (usualmente FIFO) entre los clientes del servicio y el servicio, actuando como buffer



Message Queues

Consideraciones

El orden de los mensajes no suele estar garantizado

En modelo transaccional
- Algoritmo del consumer
  1) Dequeue (get message)
  2) Process
  3) Delete message from queue

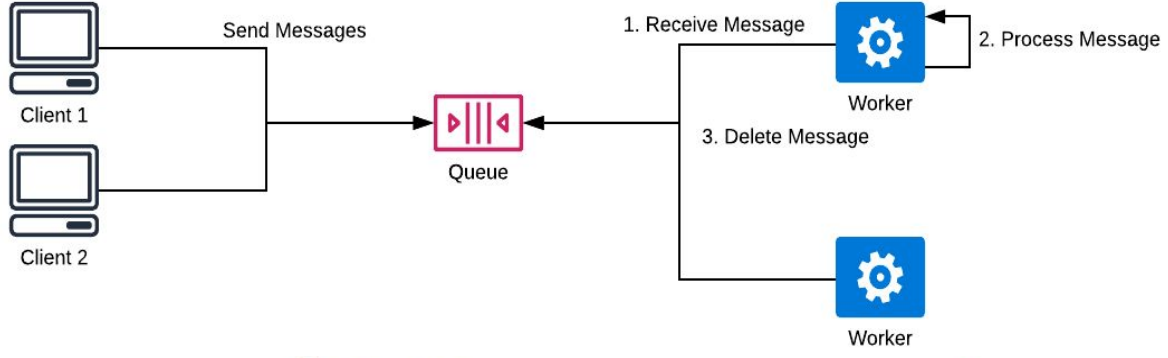
- Se usa **ventana de invisibilidad**

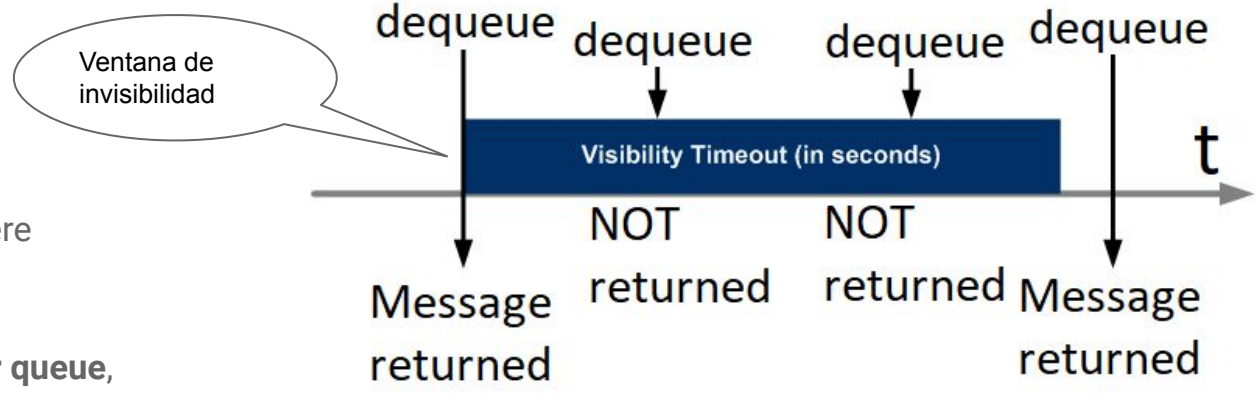
Un mensaje podría ser entregado **más de 1 vez** (ej. si un worker falla)

→ implementar **idempotencia**

Poison messages: mal formado, o requiere accesos denegados.

→ Guardar el mensaje en una **dead letter queue**, para posterior análisis





Availability, Performance, Scalability

¿Cómo lograr que múltiples componentes "consumers" (workers) procesen mensajes de múltiples "producers" (apps) de forma concurrente?

Competing Consumers

Gran número de requests

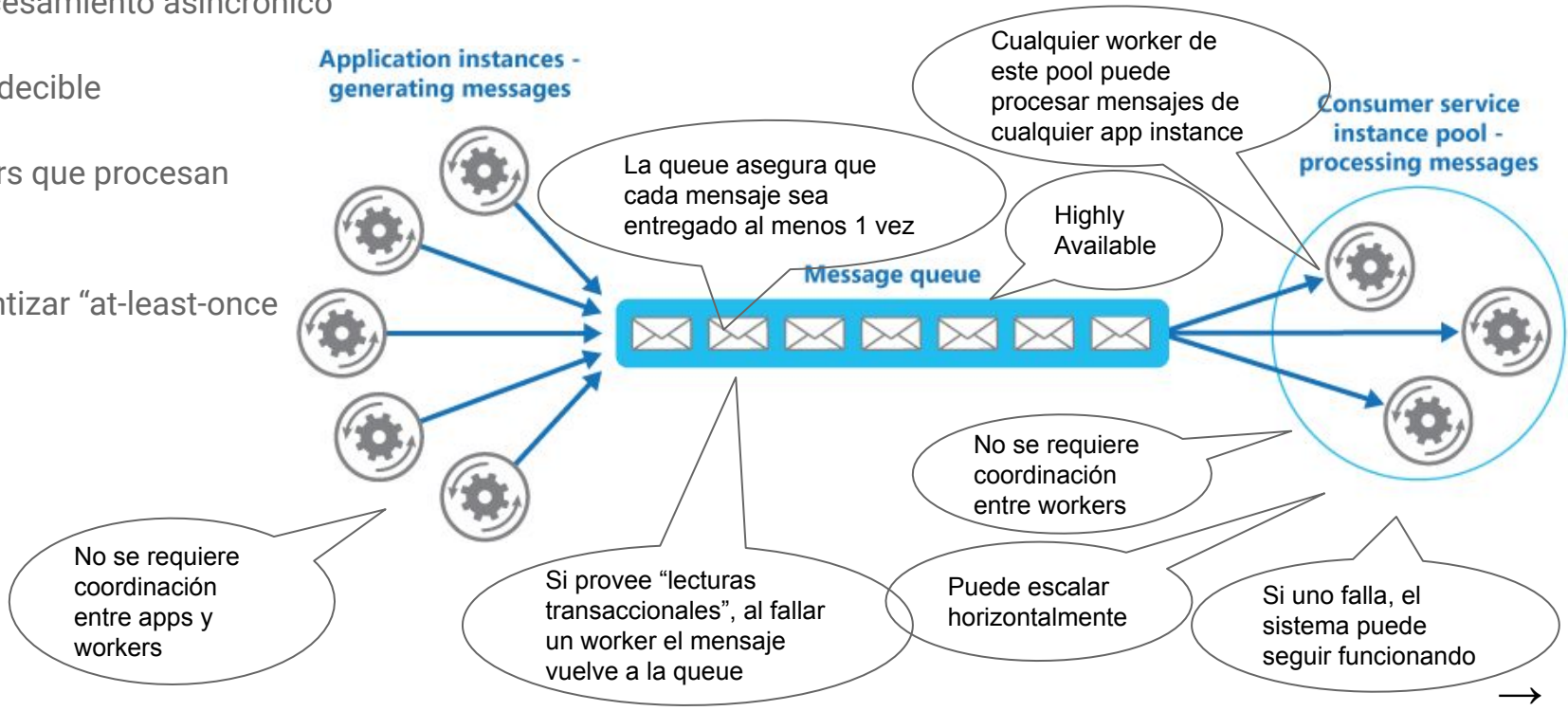
Se requiere procesamiento asincrónico

Demanda impredecible

Múltiples workers que procesan requests

Queremos garantizar "at-least-once processing"

Introducir una cola de mensajes entre las instancias de la aplicación y las instancias de los consumers

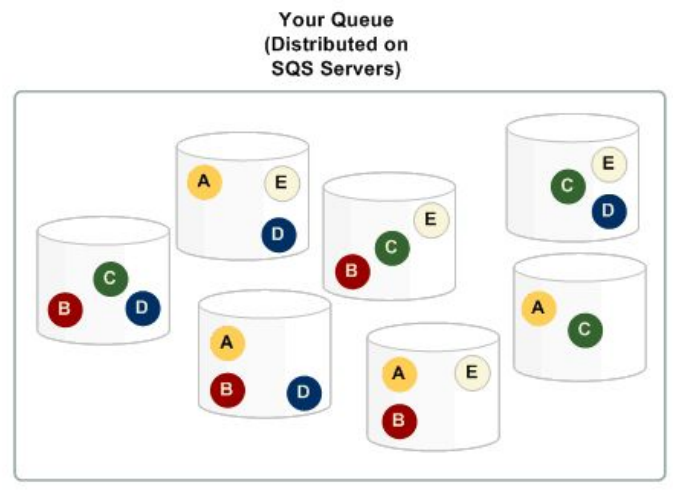


→

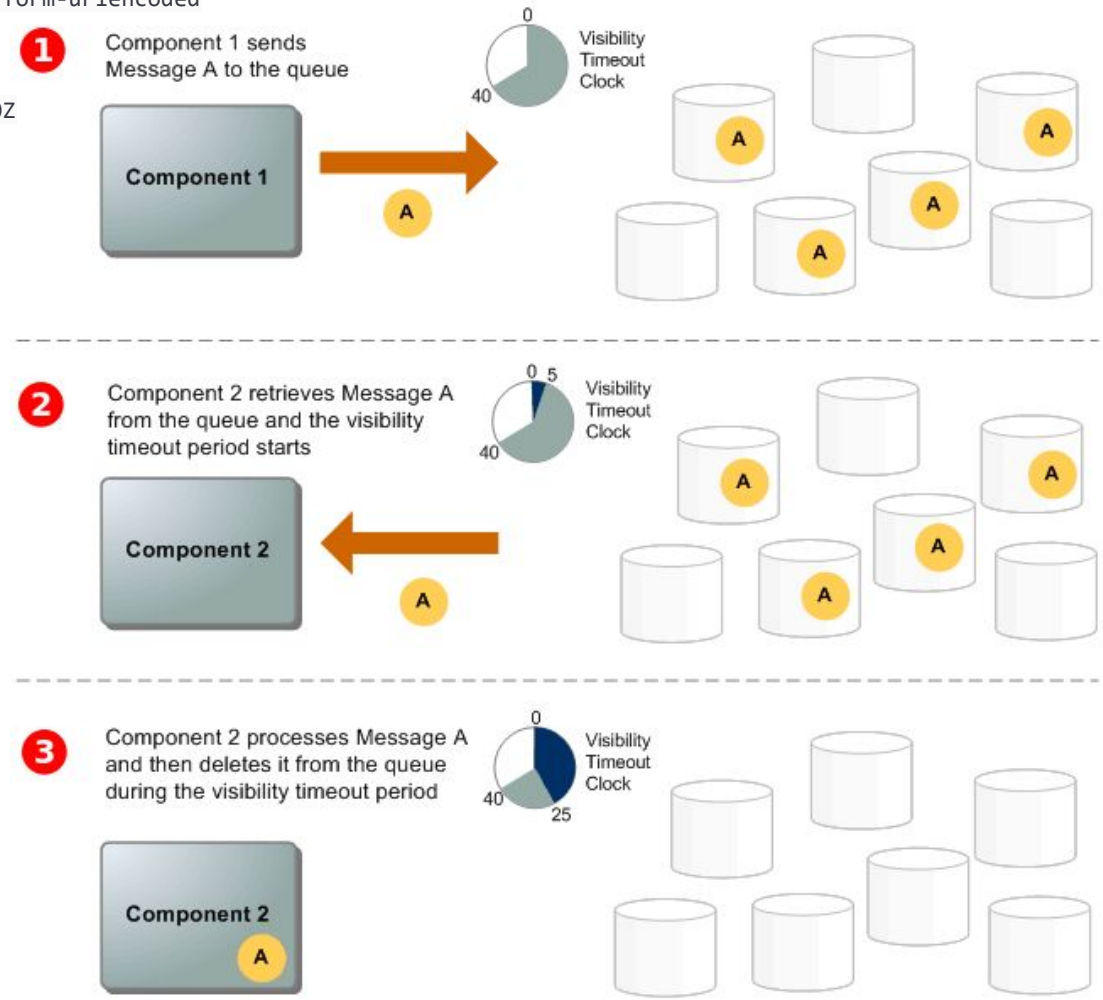
AWS SQS

```
POST /MyQueue HTTP/1.1
Host: sqs.us-east-2.amazonaws.com
Content-Type: application/x-www-form-urlencoded

Action=SendMessage
&MessageBody=Your+Message+Text
&Expires=2020-10-15T12%3A00%3A00Z
&Version=2012-11-05
```

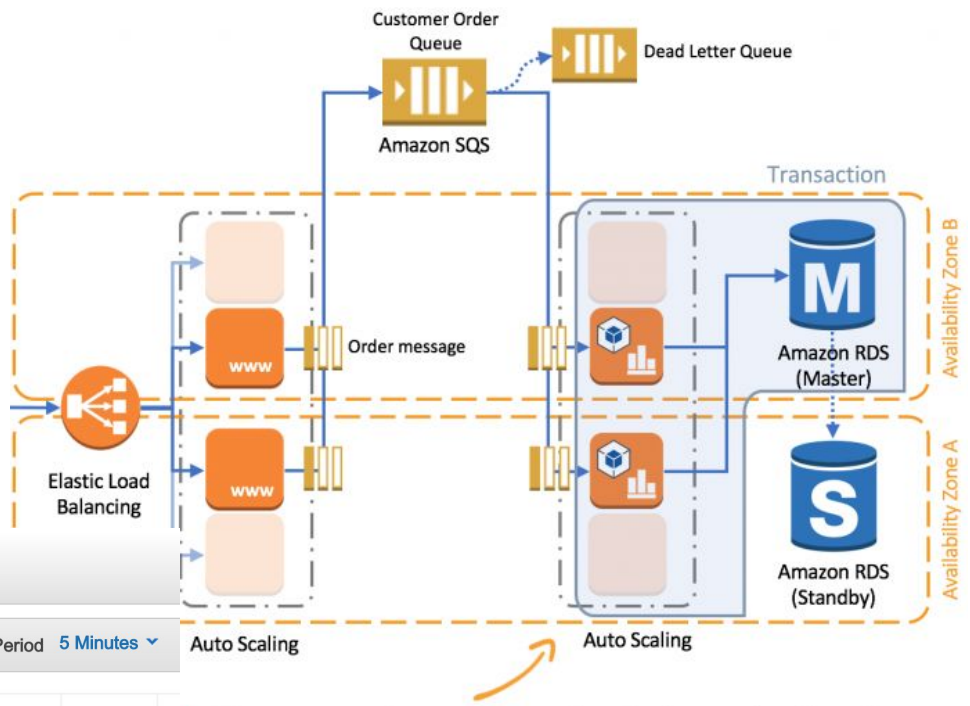


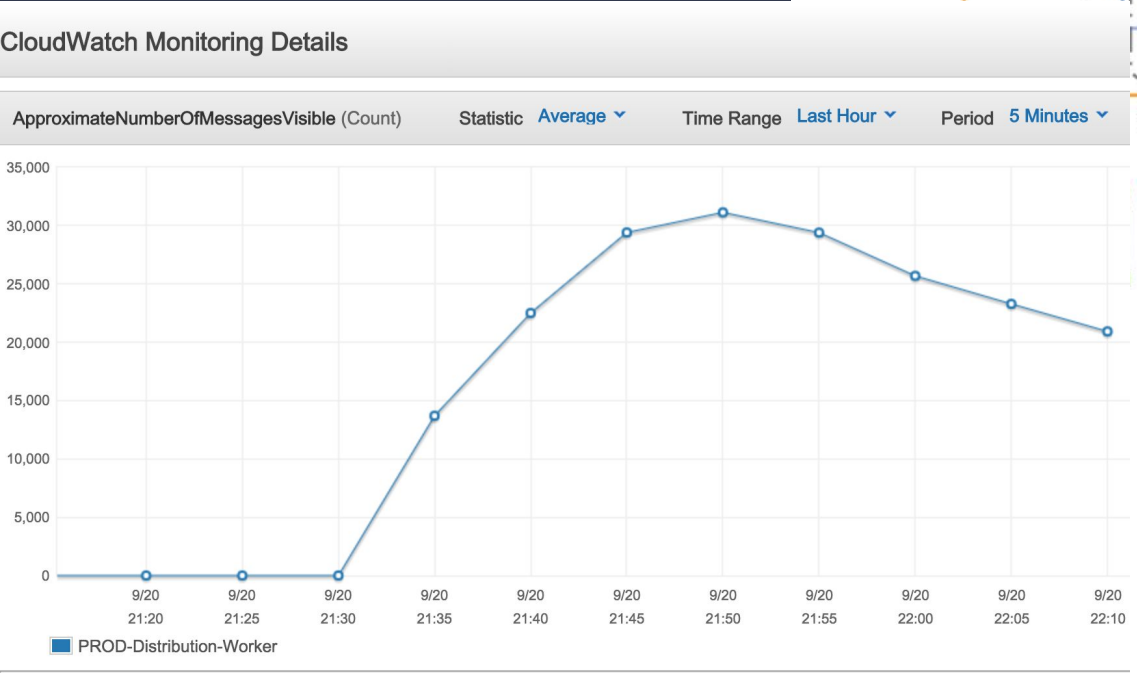
DB servers distribuidos con replicación



AWS SQS

Auto-Scaling basado en métricas de las queues





New processing nodes scaling independently, using

- `ApproximateNumberOfMessagesVisible`
- `ApproximateAgeOfOldestMessage`

Performance, Scalability

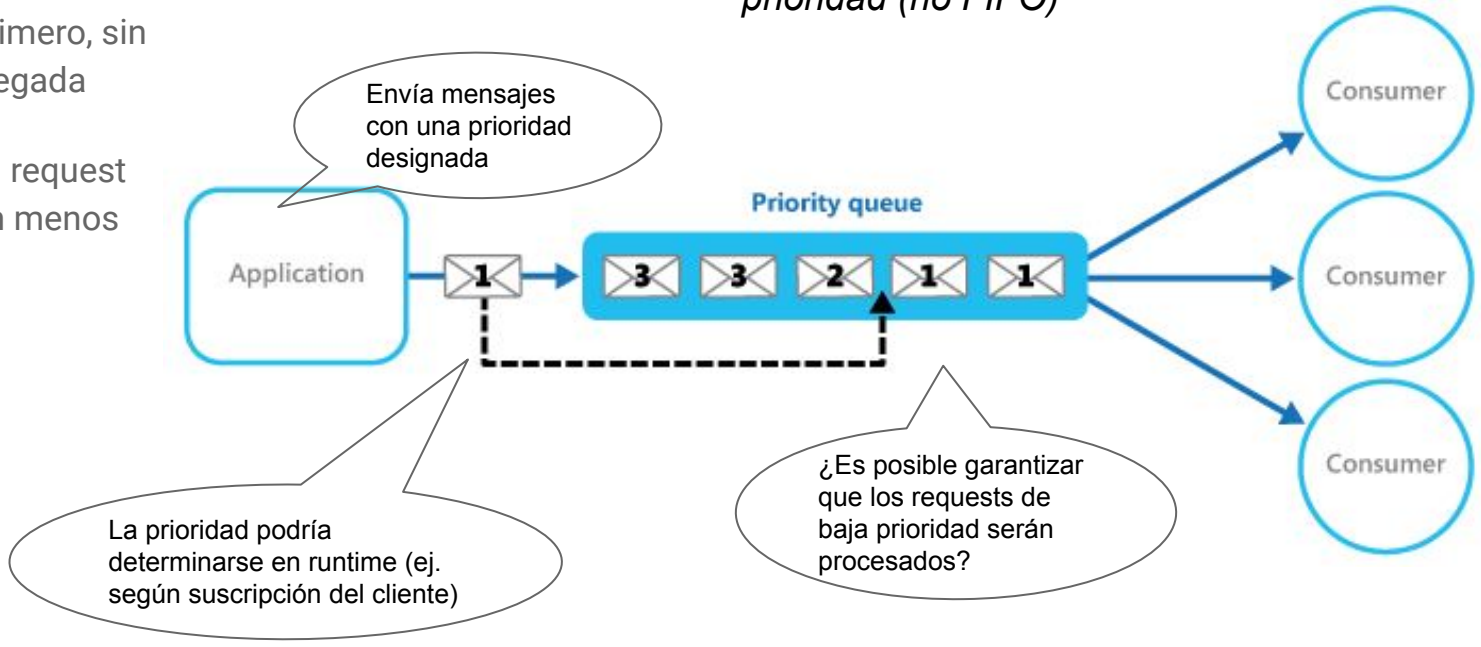
¿Cómo priorizar requests para respetar distintos SLAs para clientes particulares?

Priority Queue

Los requests de “alta prioridad” deberían ejecutarse primero, sin importar el orden de llegada

Ej.: “Alta prioridad” = el request debe ser procesado en menos de 5 segundos

Introducir una cola de prioridad que ordene los mensajes según prioridad (no FIFO)



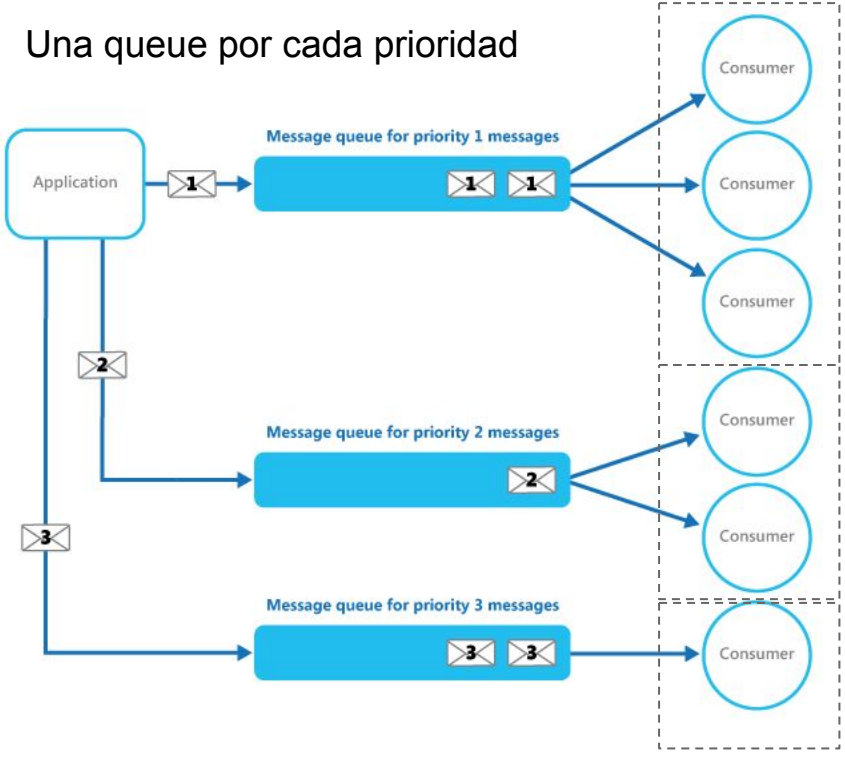
Ah sí, muy lindo, ¿y qué hago si no tengo una “Priority Queue” mágica?

→

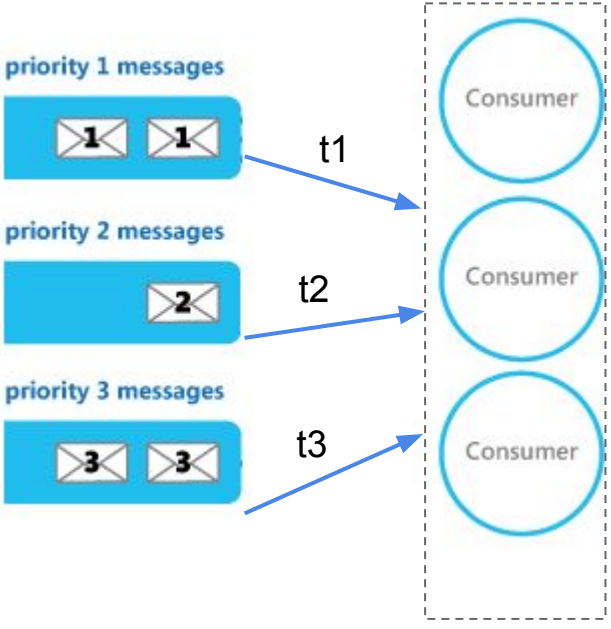
Variantes de implementación

Priority Queue

Una queue por cada prioridad

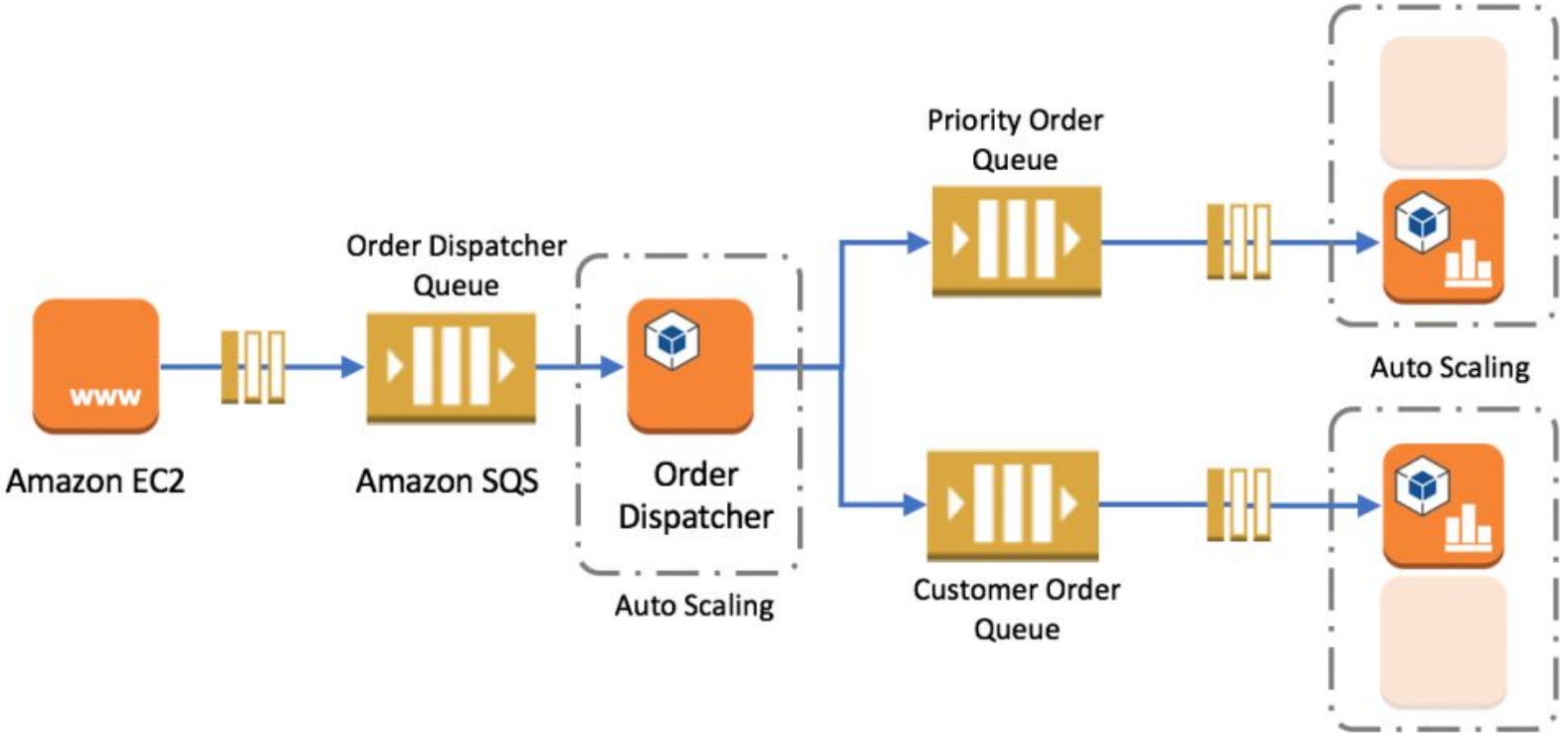


Un único pool de consumers. Sólo se procesan los P2 si no hay mensajes P1



Con 1 sólo pool de consumers no hay garantía de procesamiento de P2 y P3

Priority Queue en AWS



Availability, Performance

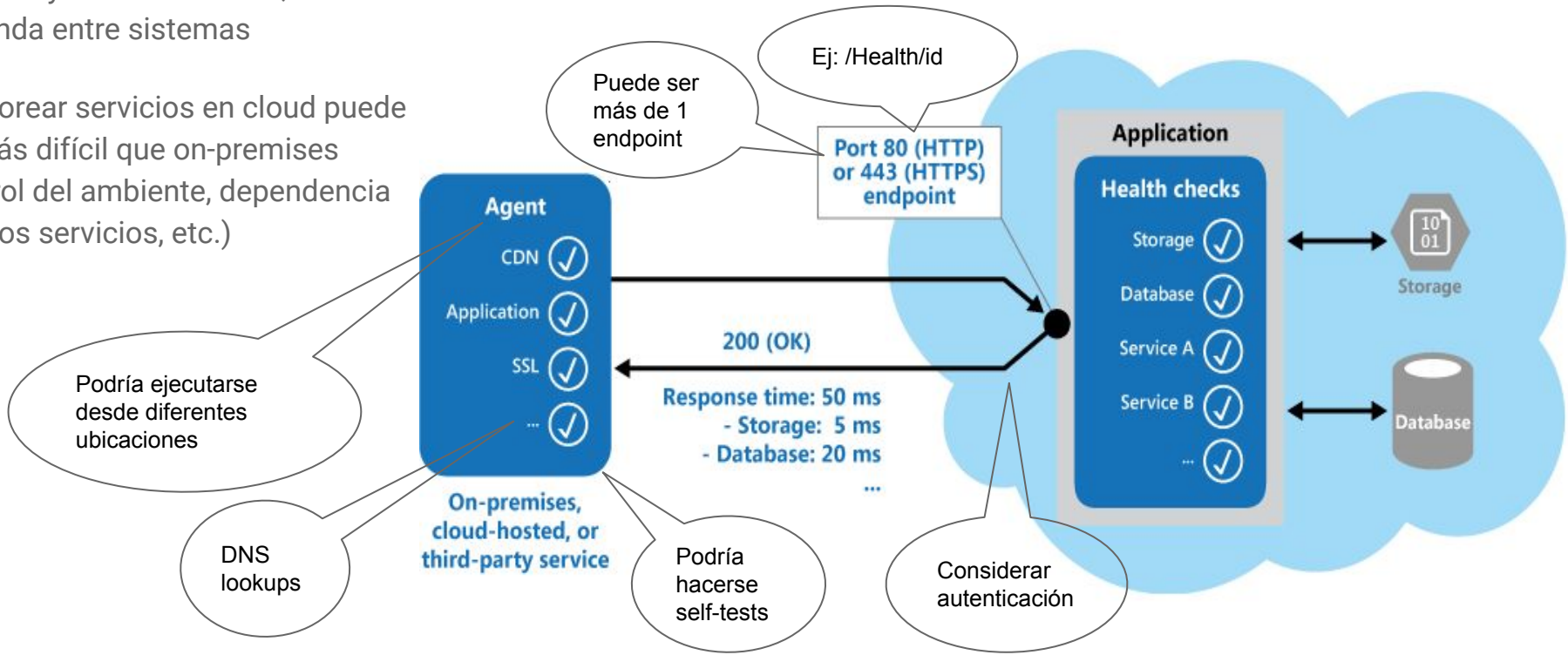
Health Endpoint Monitoring

¿Cómo verificar si un servicio está cumpliendo con el nivel requerido de availability y performance?

Factores que afectan la app: Latencia de red, performance / availability de subsistemas, ancho de banda entre sistemas

Monitorear servicios en cloud puede ser más difícil que on-premises (control del ambiente, dependencia de otros servicios, etc.)

Implementar un endpoint de "chequeo de salud" en el servicio, y disponer de un componente que lo consuma y analice resultados

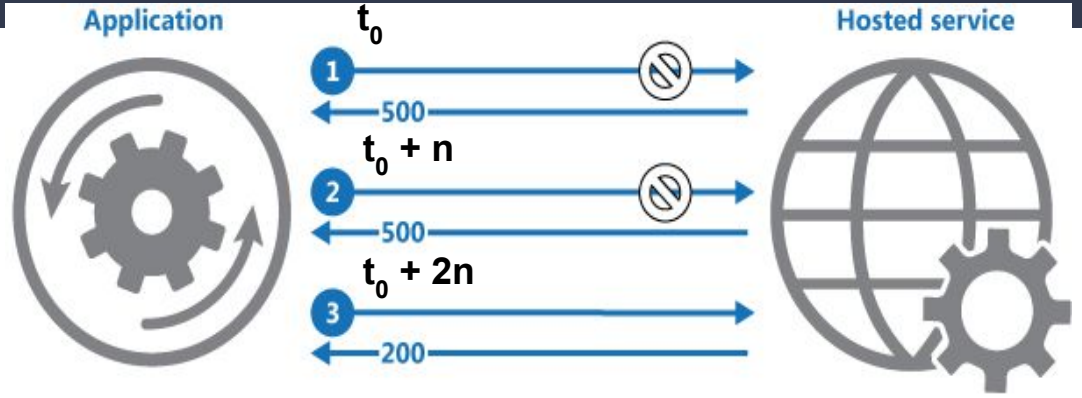


¿Cómo lograr que las aplicaciones manejen fallas temporales de los servicios que consumen?

Retry

En ambientes cloud, fallas temporales son comunes: conectividad, servicios no disponibles, timeouts por servicio ocupado

Esas fallas suelen corregirse “solas” (ej. throttling)



Implementar lógica de reintentos (retry) que consideren un lapso de tiempo (delay) entre cada reintento

- Puede haber muchas instancias de la app reintentando
  - Se debe tratar de distribuir los reintentos lo más uniformemente posible

- La política de retry (cantidad, delays) debe ser escogida según la naturaleza de la interacción client-server
- Políticas agresivas pueden degradar más el servicio
- Considerar si la operación es **idempotente**: la falla puede darse al enviar la respuesta
- Mantener logs de los intentos fallidos

Circuit Breaker

Availability

¿Cómo evitar que las aplicaciones invoquen servicios que están caídos, y que luego detecten cuando el problema ya está resuelto?

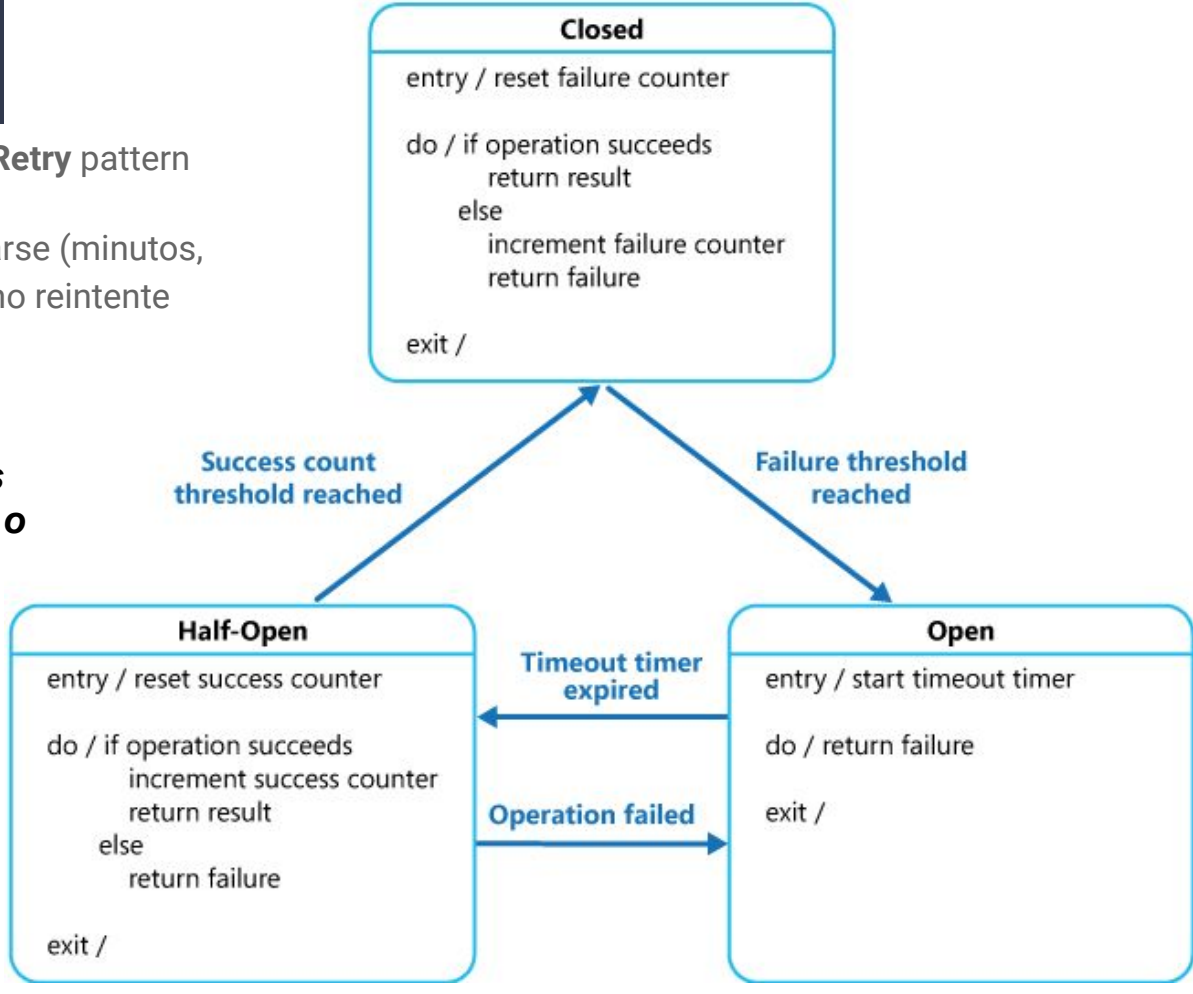
Las fallas temporales pueden manejarse con **Retry** pattern

Si la falla toma tiempo considerable en arreglarse (minutos, horas), sería ideal que la app acepte la falla y no reintente

Implementar un proxy que registre la cantidad de fallas recientes y use esos datos para decidir si invocar el servicio o si fallar inmediatamente

Estados del Circuit Breaker

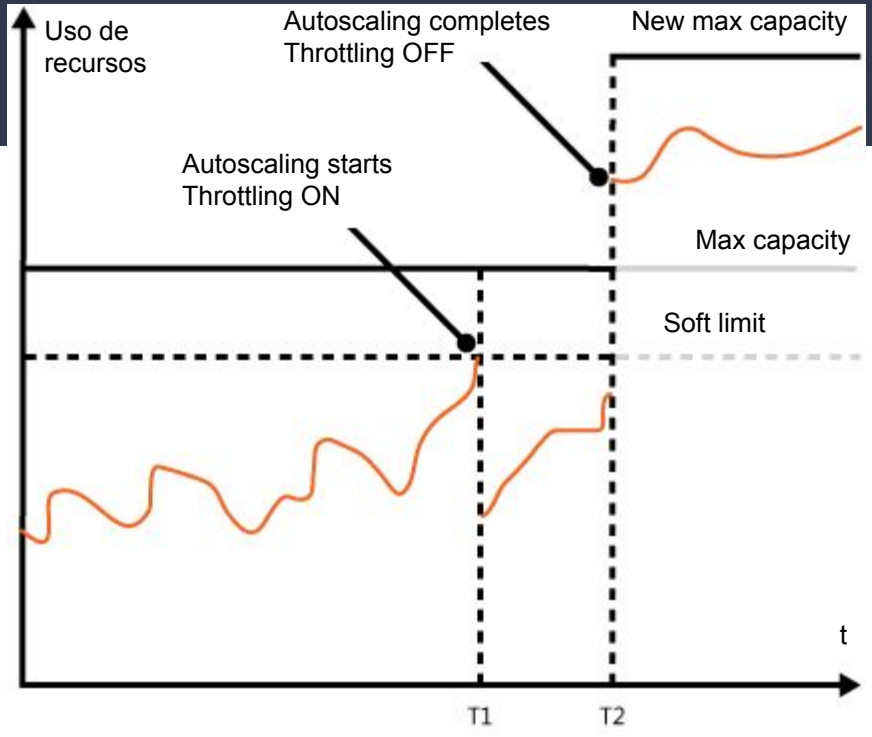
- **Closed**: pasan todos los requests
- **Open**: devuelve excepción inmediatamente
- **Half-Open**: un número limitado de requests pasan, los otros fallan



Availability, Performance

Throttling

¿Cómo controlar el uso de recursos para cumplir con SLAs sin utilizar únicamente autoscaling?



Un único cliente / app / servicio podría consumir todos los recursos con un burst de requests

Gran cantidad súbita de requests

Autoscaling no es inmediato, y aumenta los costos (respecto a no usarlo)

Imponer un límite al uso de recursos, que al alcanzarse, el sistema comenzará a regular los requests de los clientes

- Rechazar requests de clients que ya usaron la API más de N veces por segundo durante un lapso de t
- Degradar o deshabilitar funcionalidades no esenciales
- Priorizar requests según SLAs

- La implementación puede tener gran impacto - en general es difícil implementar tarde en el desarrollo
- Devolver códigos de error específicos para throttling
- Debe poder ser ejecutado y detenido rápidamente

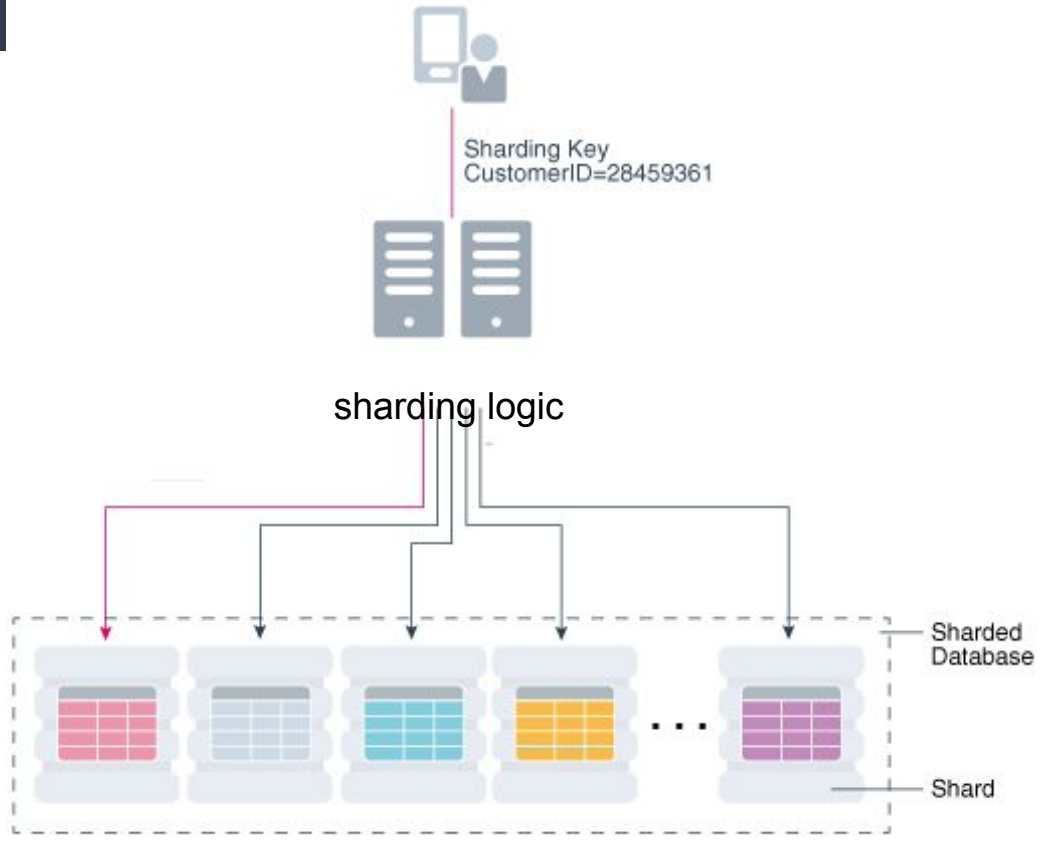
Availability, Performance, Scalability

¿Cómo superar las limitaciones de espacio, recursos, ancho de banda y geográficas de un almacenamiento de datos en 1 server?

Sharding

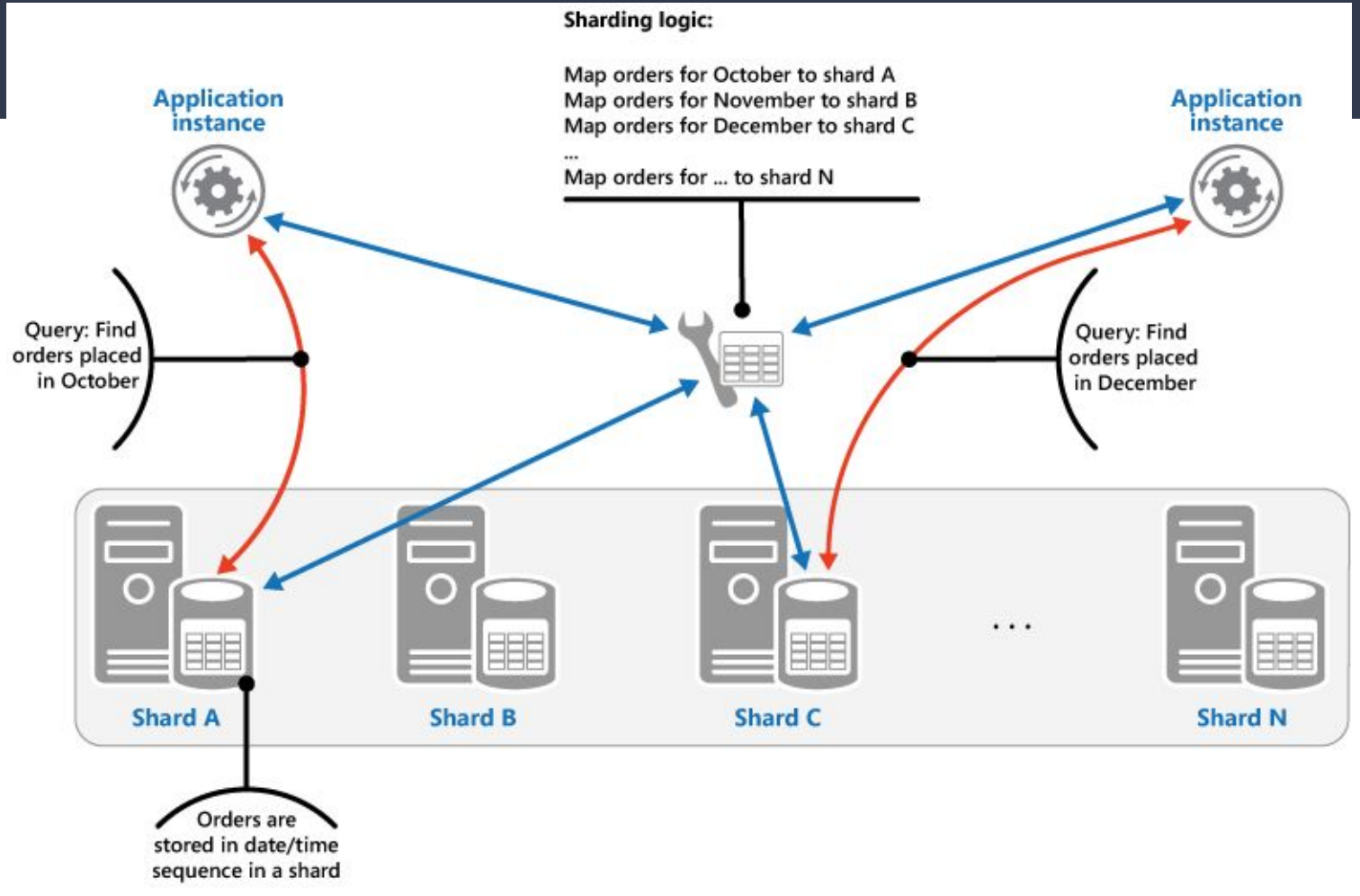
Dividir el almacenamiento de datos en particiones "horizontales" (shards)

- Cada shard corre en su propio server y es independiente del resto
- Se puede escalar horizontalmente
- Se puede balancear la carga entre shards
- Los shards pueden ubicarse geográficamente cerca de los usuarios que lo usarán
- Cada shard contiene items que pertenecen a un rango, el cual es determinado por una shard key (o partition key)
    - Lookup strategy: mapa shardkey-shard
    - Range strategy: agrupar relacionados
    - Hash strategy: Hash(4) => shard B
- Debe haber lógica de sharding, ya sea en el sistema de almacenamiento, o en la app



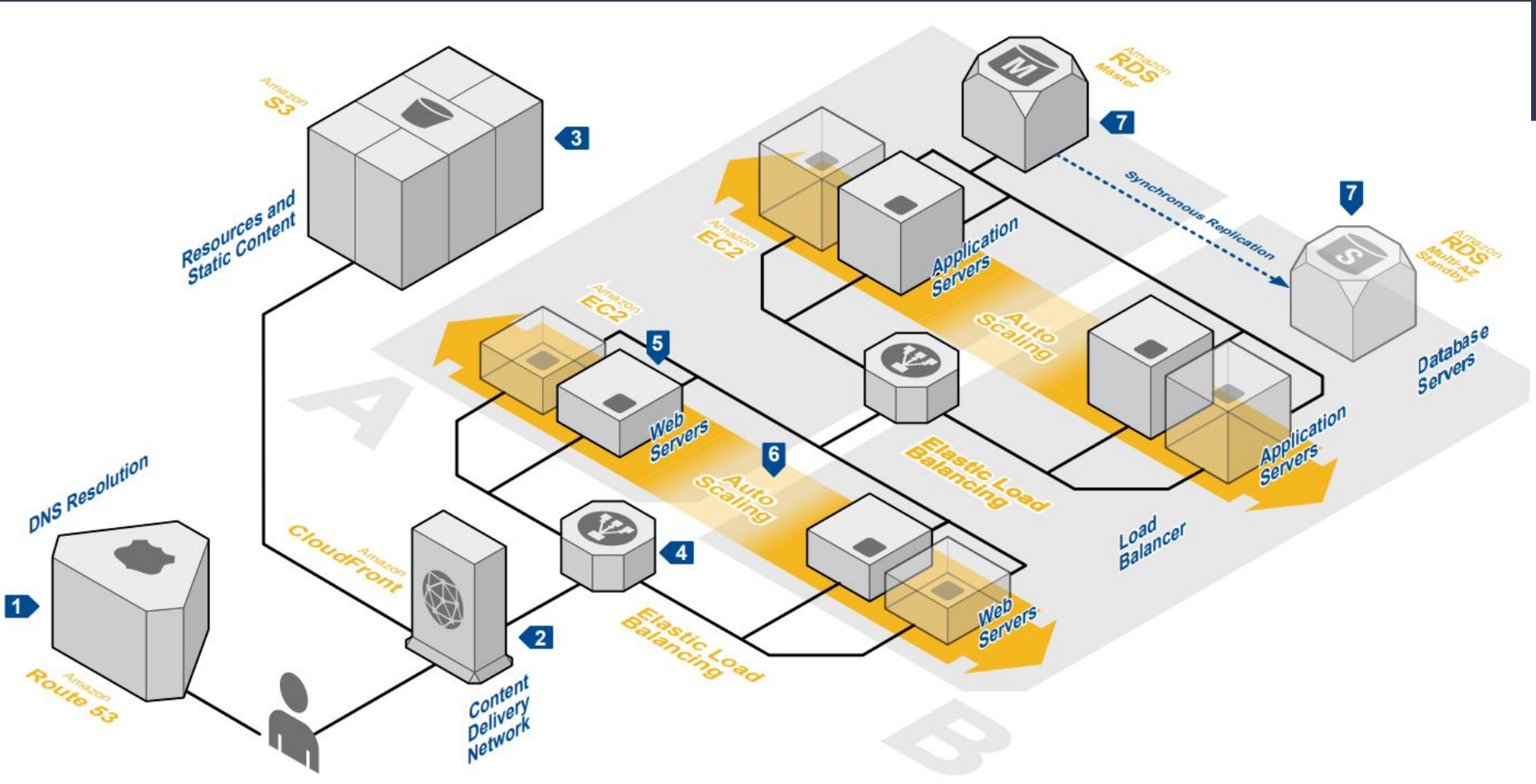
→

Sharding

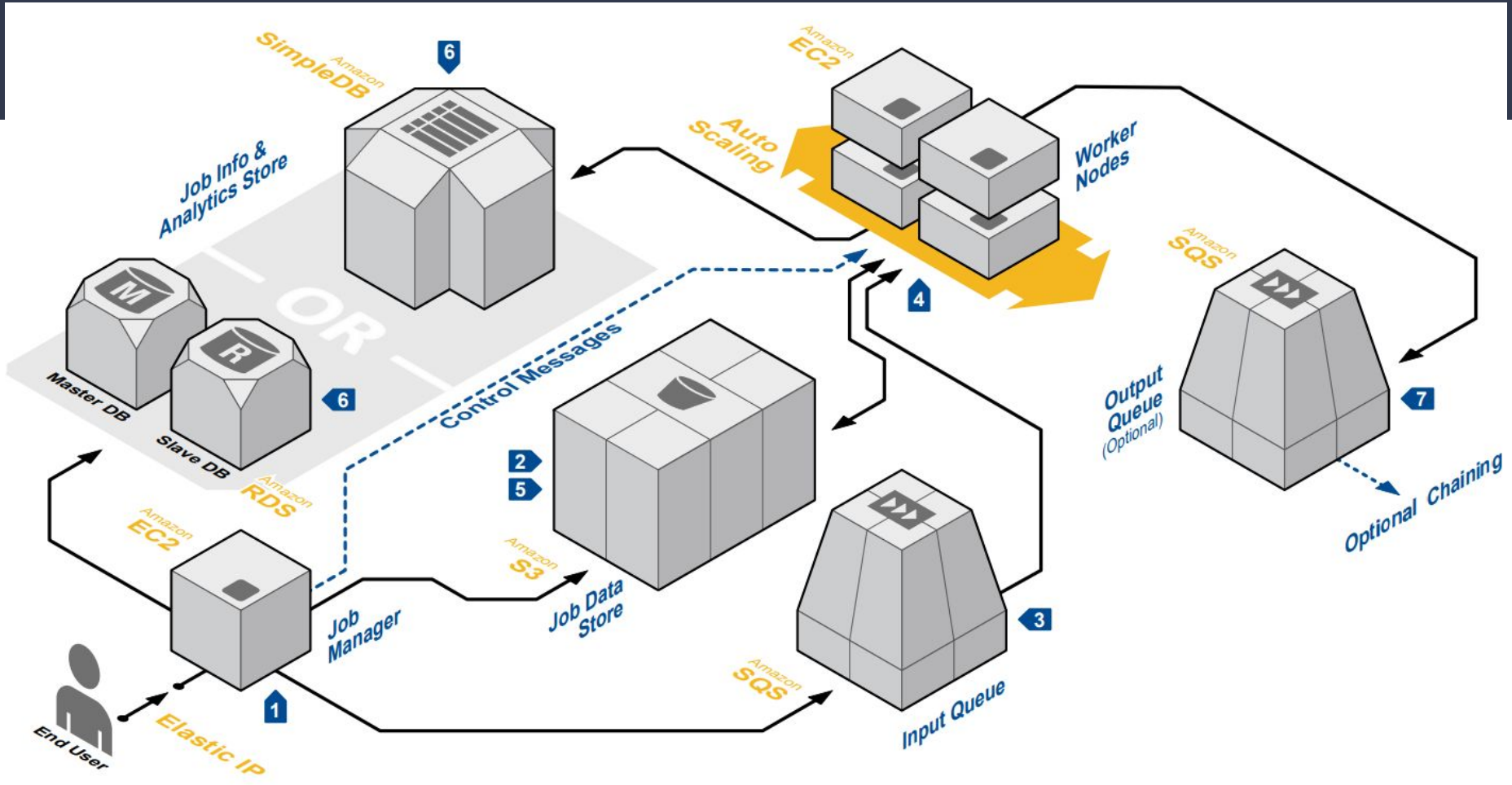

Sharding logic:
Map orders for October to shard A
Map orders for November to shard B
Map orders for December to shard C
...
Map orders for ... to shard N
Application instance
Application instance
Query: Find orders placed in October
Query: Find orders placed in December
Shard A
Shard B
Shard C
Shard N
Orders are stored in date/time sequence in a shard

Arquitecturas de referencia

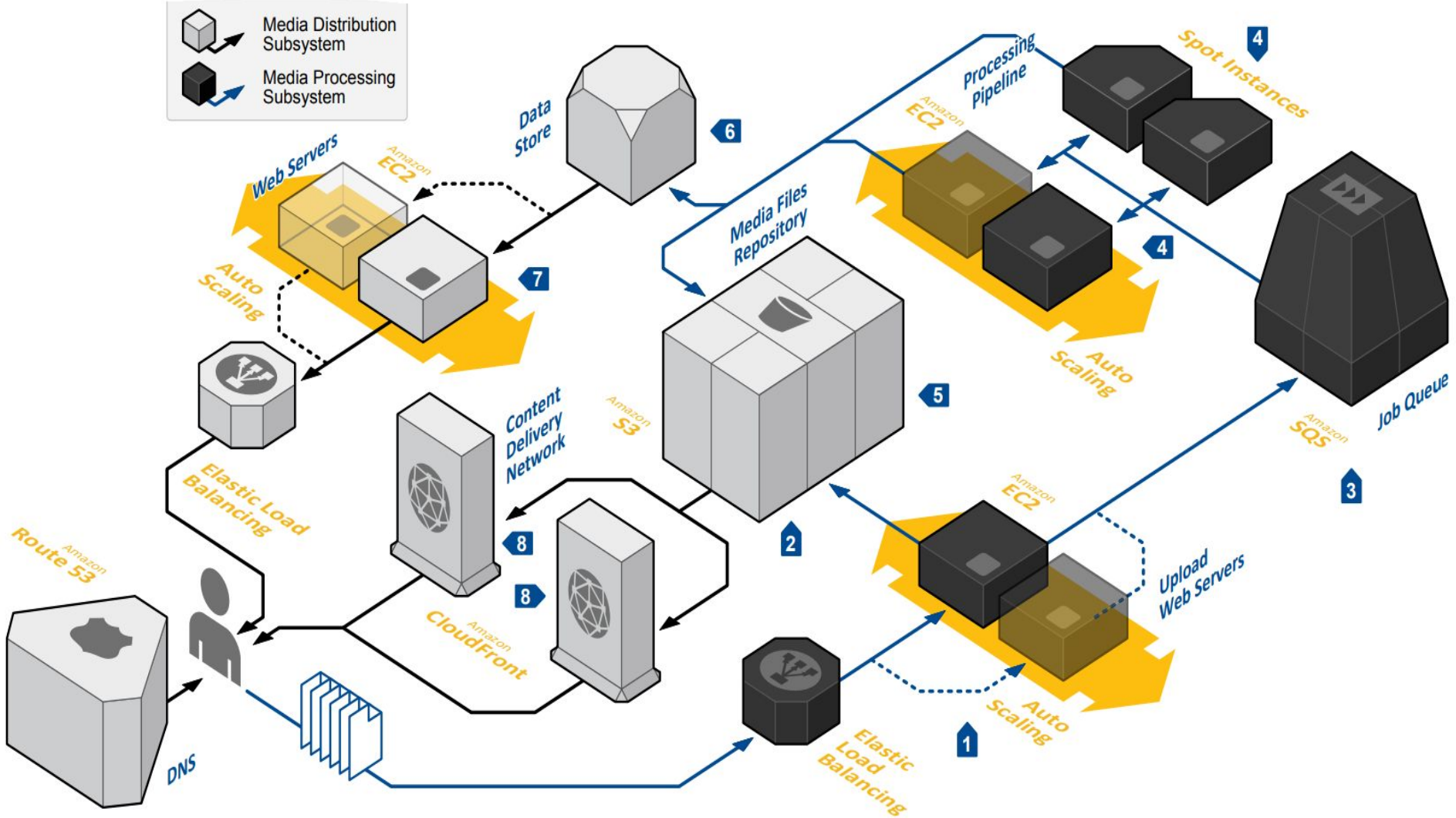
Web App


Amazon S3
Resources and Static Content
3
DNS Resolution
1
Amazon Route 53
Amazon CloudFront
2
Content Delivery Network
4
Elastic Load Balancing
Amazon EC2
Web Servers
5
Auto Scaling
6
Amazon RDS Master
7
Synchronous Replication
Amazon RDS Multi-AZ Standby
Database Servers
Amazon EC2
Application Servers
Auto Scaling
Elastic Load Balancing
Load Balancer
Application Servers
Web Servers
A
B

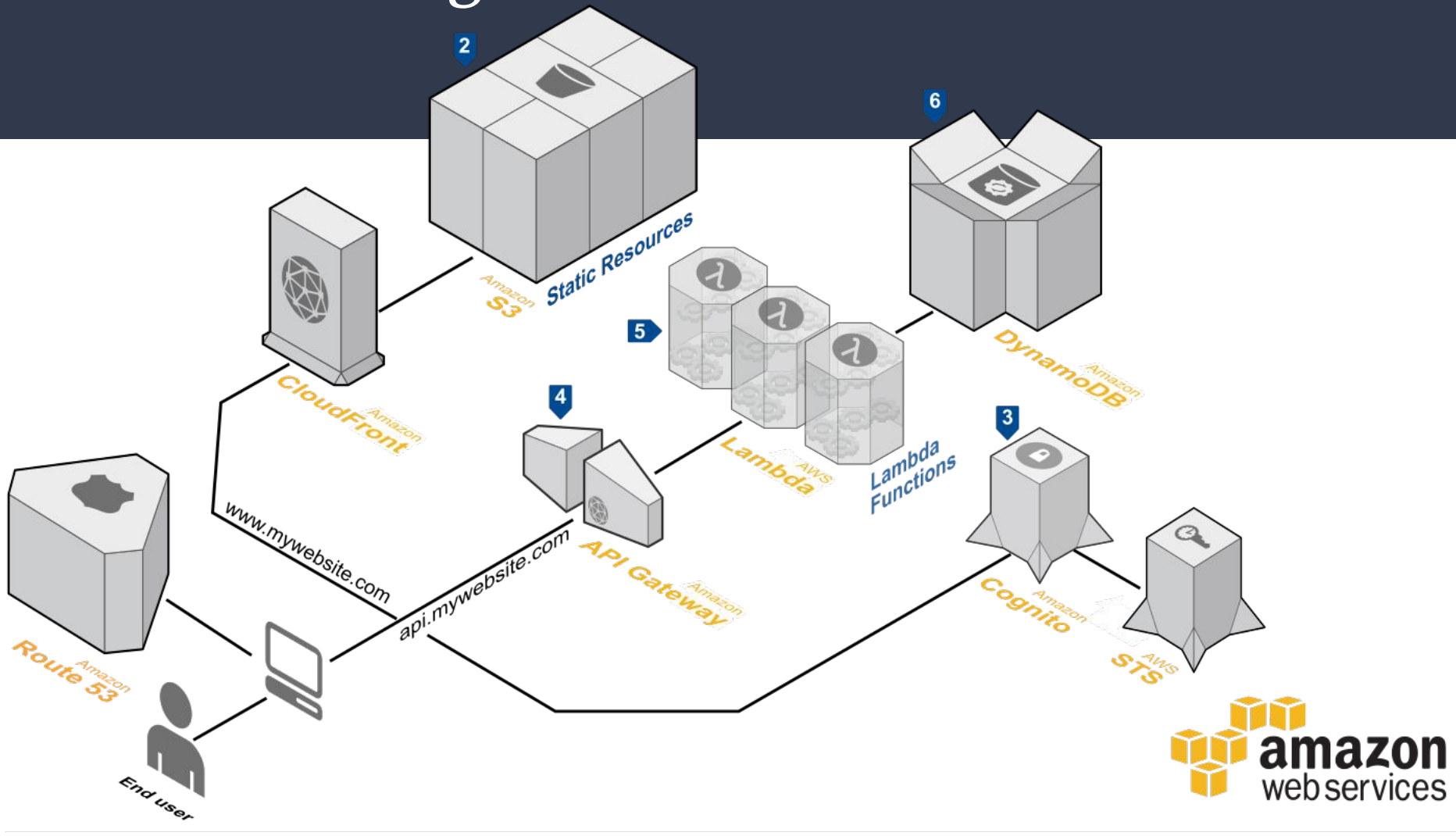
Batch Processing



Media Sharing



Serverless blog



¿Consultas?

Feedback

https://goo.gl/forms/NvrORS12kuuBitpE3

Guillermo Rugilo

grugilo@fi.uba.ar

¡Gracias!