Preface:

- Why move to cloud? (1)
  - → The advantages
  - → Can we reap the benefits directly?
  - → hence migration.
- Migration strategies (6 R's) (2)
- Re-architecturing in detail (3)
  - → Case study of re-arch.
  - → what is "Cloud-native"?
- Challenges of re-arch (4)
- Results (5)

Why move to cloud? (1)

We've seen all the advantages of cloud:

• Cost • Scaling • Access • Integration • Security

- By simply running your application do you think you can get ALL these benefits? PS. Answer is No
- Let's understand why by taking examples (3 examples)

i) Cost: Say you wrote all your modules in ONE project. You now have no choice But to get 1 very large instance, this gets even worse when you have to distribute. (Diag 1)

(Diagram 2) shows a Microservice architecture

ii) Security

iii) Scaling

# of instances for distribution

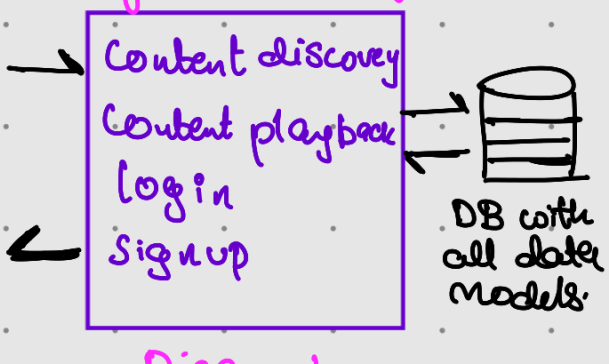


Diagram1

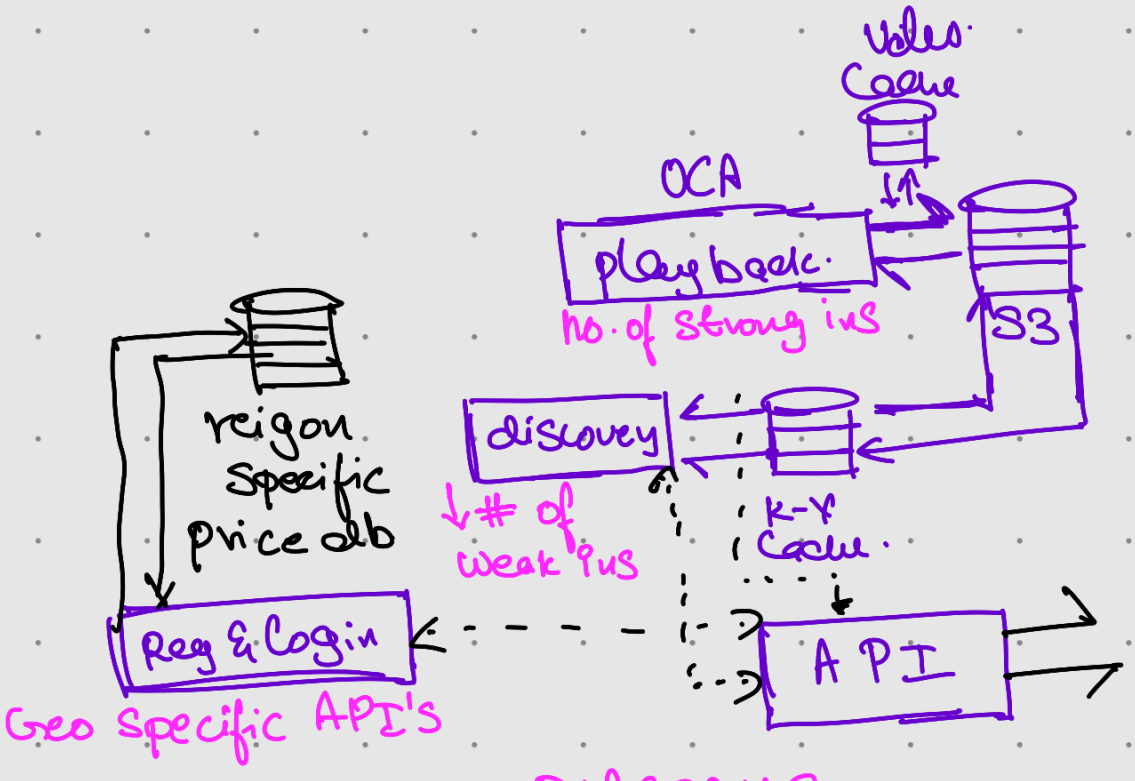


DIAGRAM 2

Migration is the process of moving services from On-premise to co-located cloud providers while taking benefits of cloud.

(PS. Migration can also be moving from one cloud env to another.)

Migration strategies (2) (6-R's)

i) Re-host (Lift & Shift)

ii) Re-platform (Lift-tinker & Shift)

➡ No changes to architecture, only some optimisations like using cloud services (DB as a Services)

Netflix themselves started moving away from internal solutions to Spring boots. This cost Sdey because of better documentation & API abstractions.

iii) Repurchase :. moving away from on lisences to SAAS

Akamai uses Salesforces instead of using internal CRM with a ton of lisences.

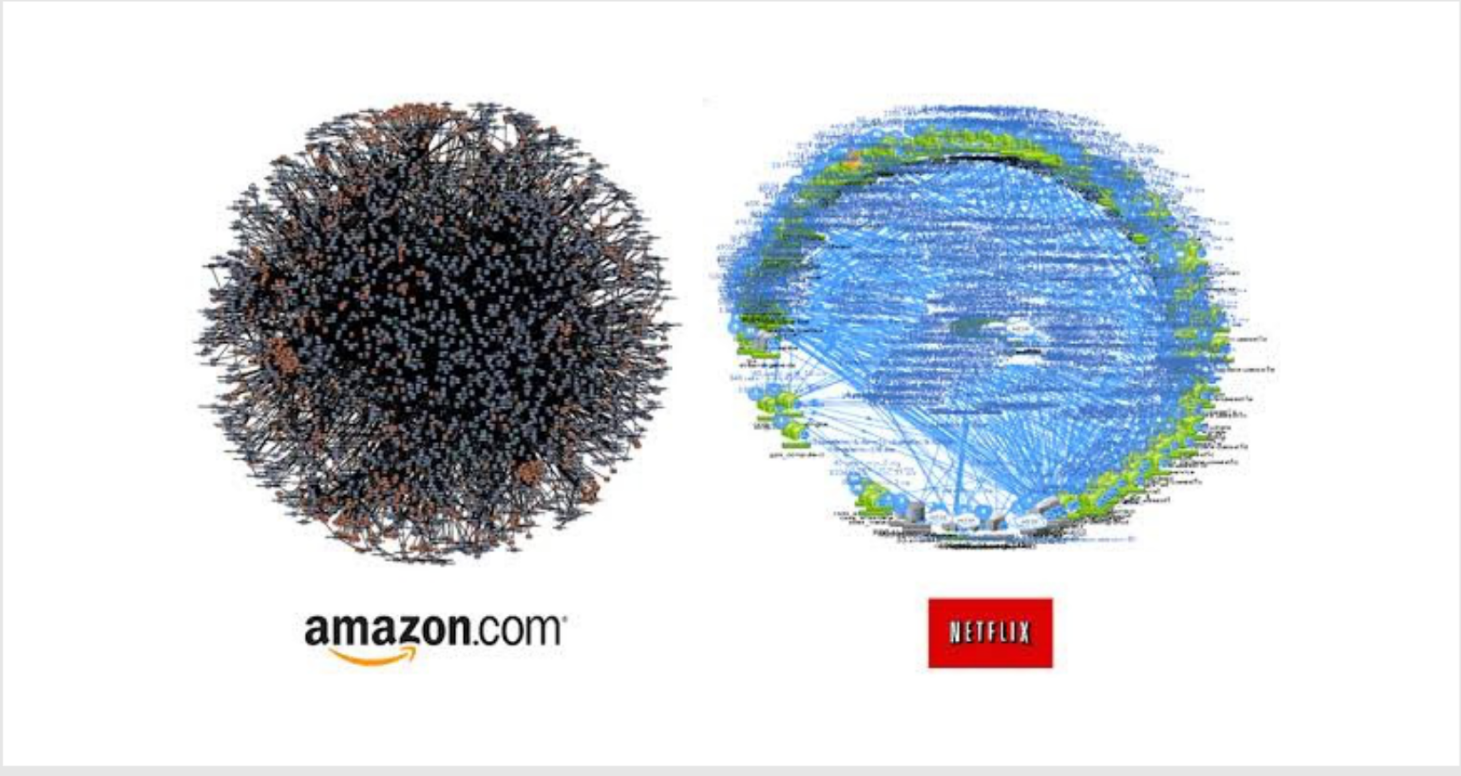
iv & v) Retire & Retain : These are post migration techniques.

vi) Re-arch : We'll see this in detail.

Rearchitecting (3)

There are 3 architectures:
i) Monoliths ii) SOA iii) Microservices.

- We've seen the advantages of μservices. But converting a monolith to μservice is no easy process.
- Why even build monoliths in the first place? Well, monoliths are a better choice when application is to be up & running quickly.

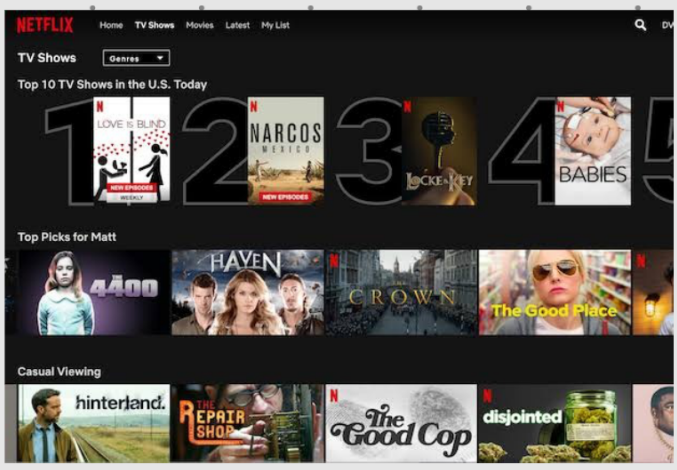


- Why μService then? Netflix serves 4 million users every second spread through 190 countries. At this scale, maintainability & scaling are MUST.

How Netflix works: the (hugely simplified) complex stuff that ha...
Not long ago, House of Cards came back for the fifth season, finally ending a long wait...
medium.com

AWS re:Invent 2015: A Day in the Life of a Netflix Engineer (DVO...
Watch Dave Hahn, a Senior Engineer at Netflix, explain what operating at Netflix and s...
youtu.be

- LOLOMO: list of list of movies a service the netflix handles, what if this was a monolith? well, one could have never pointed out. But as it is a μservice, they have simple routed a "basic" fall back that fits right in with other services:



Well, I am all for μservices, let's migrate!!... Not quite, it's hard.

> "cloud-native" applications are those that make full use of the cloud env. Hence all μservices such are cloud native.

Challenges of Re-Arch

i) Decomposition: finding those services that can work as a μservice. This is a very large task, but these help:
- → stop adding more features to monoliths (Strangler fig arch)
- → find "Seams". Monoliths modules that are loosely coupled.
- → Identify those components for which business want to add more features (ROI approach)

> But decomp add a lot more complexity to Infra, Config, deploy & monitor... hence plan on using containerisation & CI/CD piplines before hand (GCP)

Decomposition

How to decompose an application into services?

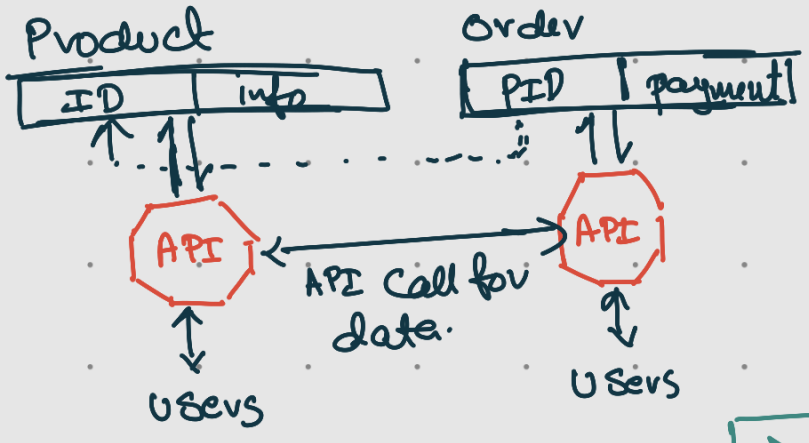
- Decompose by business capability - define services corresponding to business capabilities
- Decompose by subdomain - define services corresponding to DDD subdomains
- Self-contained Service - design services to handle synchronous requests without waiting for other services to respond new

ii) DB decomp / persistence

A truly μService can will also decompose it's DB. But often one service will need data from another (r/w)

(*) → Data refrences / table refrence (read only)

Consider these 2 schemas:



often "Join" two these 2 table are to be done for "features"

Soln 1: API calls for data

Soln 2: replicate DB. challenging.

> Consistency is lost on replication.

(DB / service)

(*) Shared mutable data (r/w)

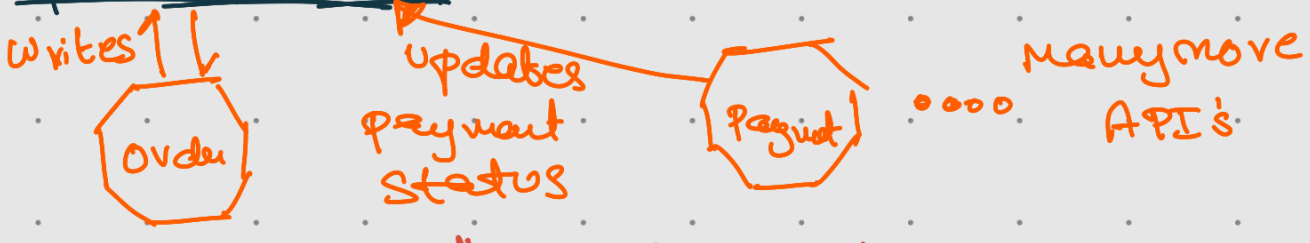
Consider the following schemas:

Order:

OID	Pay	PID	USer
1	Done	.	.
2	Done	.	
3	Pend	.	
4	Pend	.	

Payment:

Pay ID	OID	USer
23143	3	.



Soln: model the "Order" data base as a seperate μService & create an API through which others can write. → (Shared DB)

*) Shared table: a tables has an attr that is needed by 2 or more services:

iii) Transaction Boundaries / Distributed transactions

After decomp, local transactions may now be spread across service (user creation & personalisation). This makes it hard to promise ACID properties.

In μService Consider a multistep transaction. If any 1 of the Services, all previous must be rolled back. This gets even worse where failures are not imm. This means we need some way for Services to Comm. to report failures, we'll see these next.

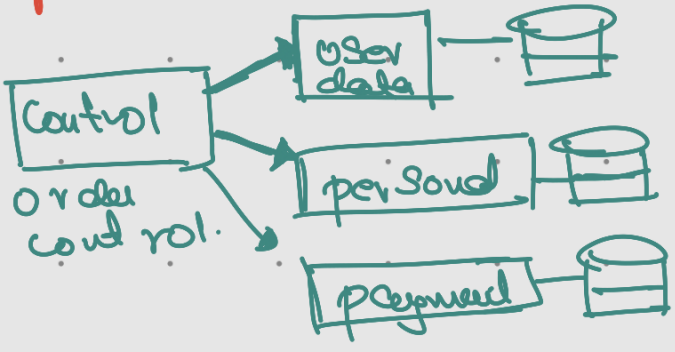
Soln 1: for distributed trans: 2-phase Commit

There exist a Controller initiating the phases:

phase 1: preparing phase ⇒ Controller asks all nodes if ready to commit periodically. (logical errors)

Phase 2: Commits to all DB simultaneously; if any failure. Roll back in all of them. (DB errors)

if payment failed, phase 1 will return an abort



Soln2 for distributed transaction: "Compensating trans"

SAGA trans.

SAGA is a sequence where each Service performs its action & publishes an event. Some Services wait for events. If any fail event SAGA executes roll backs (Compensations)

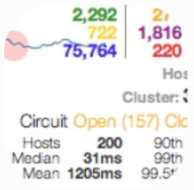
iv) Performance: The transition to µservice may have caused a slow down, hence make use of the increased observability to ...

⇒ deploy more service instance where needed.

⇒ detect problems

⇒ dedicated thread pools & deploy **circuit breakers.**

Netflix hysterix lib does this, like in LOLOMO.



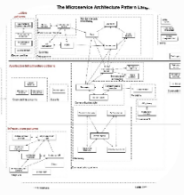
Making the Netflix API More Resilient

Maintaining high availability and resiliency for a system that handles a billion requests ...

netflixtechblog.com

- most µservices go a step ahead & apply many logging patterns:

- Log aggregation - aggregate application logs
- Application metrics - instrument a service's code to gather statistics about operations
- Audit logging - record user activity in a database
- Distributed tracing - instrument services with code that assigns each external request an unique identifier that is passed between services. Record information (e.g. start time, end time) about the work (e.g. service requests) performed when handling the external request in a centralized service
- Exception tracking - report all exceptions to a centralized exception tracking service that aggregates and tracks exceptions and notifies developers.
- Health check API - service API (e.g. HTTP endpoint) that returns the health of the service and is intended to be pinged, for example, by a monitoring service



Microservices Pattern: A pattern language for microservices

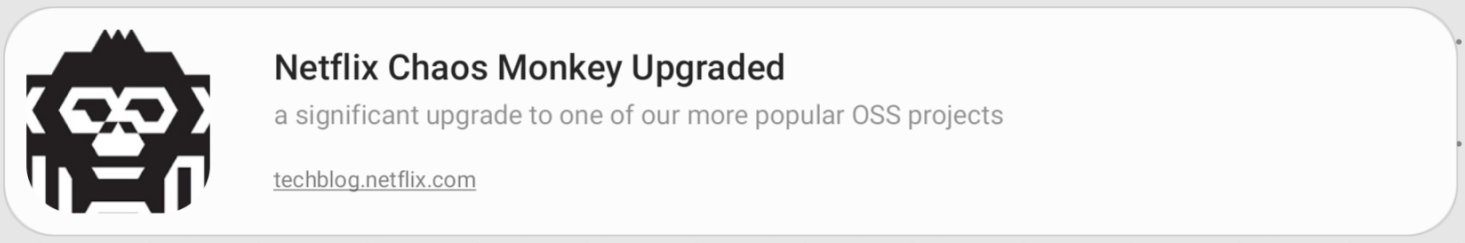
A pattern language for microservices Microservice Architecture Supported by Kong Pat..

microservices.io

excellent resource to learn µservice architecture

V) Testing : testing μservice is inherity harder. In particular those integration test. Hence adopting various testing using CI/CD is good.

eg: Chaos monkey



vi) Inter-service Comms:

Some IPC techs are: REST, gRPC OR AMQP

↳ Synchronous ↳ Async.

One can also choose from msg formats → JSON, XML, protobufs

Communication patterns

Style

Which communication mechanisms do services use to communicate with each other and their external clients?

- Remote Procedure Invocation - use an RPI-based protocol for inter-service communication
- Messaging - use asynchronous messaging for inter-service communication
- Domain-specific protocol - use a domain-specific protocol
- Idempotent Consumer - ensure that message consumers can cope with being invoked multiple times with the same message

External API

How do external clients communicate with the services?

- API gateway - a service that provides each client with unified interface to services
- Backend for front-end - a separate API gateway for each kind of client

Service discovery

How does the client of an RPI-based service discover the network location of a service instance?

- Client-side discovery - client queries a service registry to discover the locations of service instances
- Server-side discovery - router queries a service registry to discover the locations of service instances
- Service registry - a database of service instance locations
- Self registration - service instance registers itself with the service registry
- 3rd party registration - a 3rd party registers a service instance with the service registry

Reliability

How to prevent a network or service failure from cascading to other services?

- Circuit Breaker - invoke a remote service via a proxy that fails immediately when the failure rate of the remote call exceeds a threshold

Results from Re-Architecturing (5)

- Some metrics to check is Re-Arch is working:
  i) Perf ii) Scaling iii) Cloud nativeness iv) loadtime
- It is critical to note that not all applications & business needs complete microservice architecture.

Suitability	Rehosting	Replatforming	Rearchitecting
Application suitability	• Web compatible UI • MVC architecture • Flask, Symphony, .NET can be rehosted without complications • Applications not built with web frameworks but have a web-compatible UI (HTML, PHP) not ideal	• Web-compatible UI and MVC architecture • Must allow easy connection to the database server • Traditional apps using Xampp-like servers are suitable	• Monolithic architecture, but are web-compatible are suited • Large codebases • Resource intensive • Desktop applications built using frameworks are not suited
Business suitability	• Higher network speed • Low technical debt and cost of maintenance. • Small apps that require a temporary boost in performance	• Database hosting most common • Improve DB scalability • Boosts fault tolerance • Data is more resilient.	• Looking for large improvements in availability, testability, continuous delivery, reusability and lower infrastructure costs