Microservices & Servless Computing

Programmazione Distribuita - A.A. 2020/2021

Biagio Cosenza
Dipartimento di Informatica
Università di Salerno
http://cosenza.eu/
bcosenza@unisa.it

Organizzazione della Lezione

- La morte del Big software
- Microservices
- Serverless Computing
- Conclusioni

La "morte" del Big Software

- Big software
  - software di grande dimensione, progettati da aziende di grandi dimensioni

La "morte" del Big Software

- Big software
  - software di grande dimensione, progettati da aziende di grandi dimensioni
- Tipicamente, uno dei testbed più importanti delle metolodogie e tecnologie di design e sviluppo
- Storie dell' "orrore" nello sviluppo
  - 64.1% dei progetti ERP richiedono più tempo del previsto
  - 74% dei progetti ERP richiedono più budget del previsto
  - 40% dei progetti ERP, quando vanno in produzione live, causano problemi operativi di grande entità

Progetti software

MODERN RESOLUTION FOR ALL PROJECTS

	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

The Modern Resolution (OnTime, OnBudget, with a satisfactory result) of all software projects from FY2011-2015 within the new CHAOS database. Please note that for the rest of this report CHAOS Resolution will refer to the Modern Resolution definition not the Traditional Resolution definition.

Progetti software e dimensioni

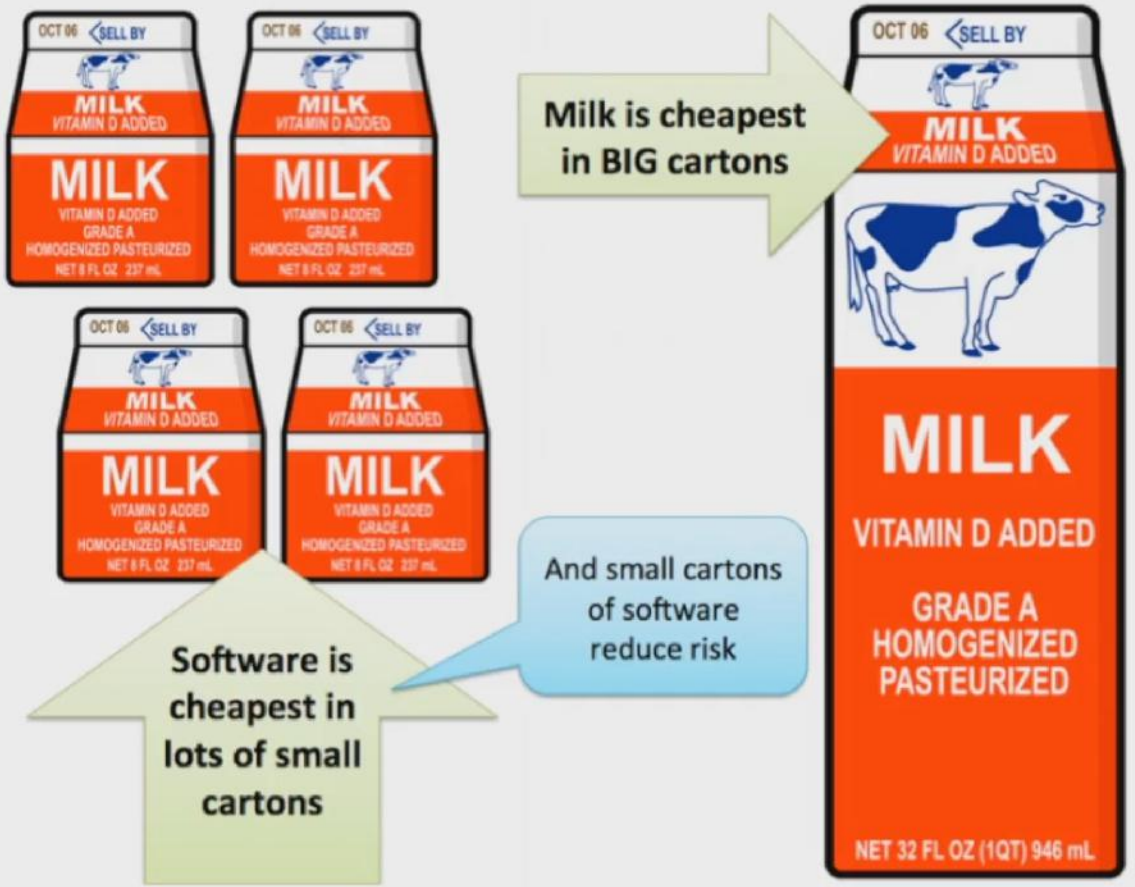
- Large vs Small

CHAOS RESOLUTION BY PROJECT SIZE

	SUCCESSFUL	CHALLENGED	FAILED
Grand	2%	7%	17%
Large	6%	17%	24%
Medium	9%	26%	31%
Moderate	21%	32%	17%
Small	62%	16%	11%
TOTAL	**100%**	**100%**	**100%**

The resolution of all software projects by size from FY2011–2015 within the new CHAOS database.

Lo sviluppo software e' un economia di scala?



Progetti – le metodologie agili

- Agile vs Waterfall

CHAOS RESOLUTION BY AGILE VERSUS WATERFALL

SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

The resolution of all software projects from FY2011–2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000

Progetti software e le ragioni del successo

- Le ragioni…

FACTORS OF SUCCESS	POINTS
Executive Sponsorship	15
Emotional Maturity	15
User Involvement	15
Optimization	15
Skilled Resources	10
Standard Architecture	8
Agile Process	7
Modest Execution	6
Project Management Expertise	5
Clear Business Objectives	4

La morte del Big Software

- Perdita del controllo del business da parte della azienda
  - in favore dei vendors della soluzione
  - vantaggio della standardizzazione, ma scarsa flessibilità
- Gestione del prodotto *one-size-fits-all*
  - uguale per tutti = adattamento necessario = maggiori costi
- Cloud: total cost of ownership
  - che aumentano con software di grande dimensione

- Altri fattori critici
  - Capacità dei team e dei progettisti
  - Costo di mantenere una singola architettura monolitica

Dalle ceneri sorge il software "small"

- Il software monolitico è rigido e poco flessibile
  - difficile da mantenere (interconnessioni strette tra moduli)
- Soluzione? Software "*small*"
- Composto da piccoli servizi che sono
  - indipendentemente sviluppati e mantenuti
  - senza dipendenze
  - comunicano con meccanismi di comunicazione leggeri e indipendenti da piattaforma
  - senza infrastruttura centralizzata
- Verso i microservices!

I vantaggi di un approccio a microservices

- Deployment continuo
- Cambi, modifiche, sostituzioni di parti delle applicazione sono ora possibili senza influenzare il resto
- Possibile il loro delivery su container (Docker)
- Gestione della applicazione in mano al cliente e non al fornitore
  - possibile rimpiazzare microservizi

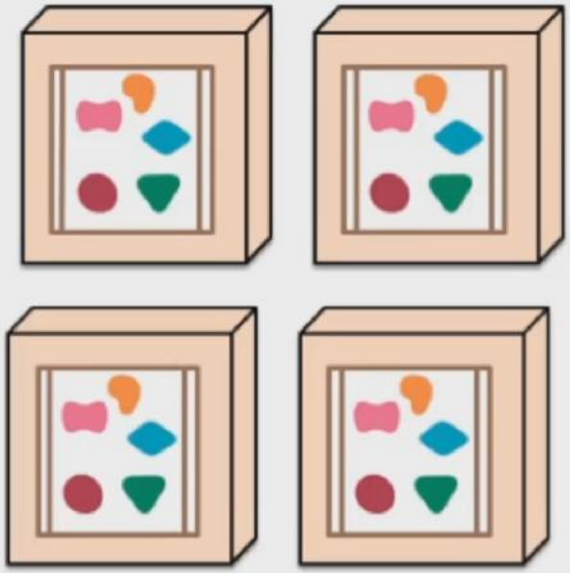
Microservices: una definizione di Fowler

In short, the microservice architectural style [1] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

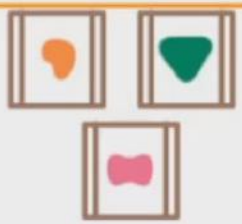
Monolithic vs Microservices

A monolithic application puts all its functionality into a single process...

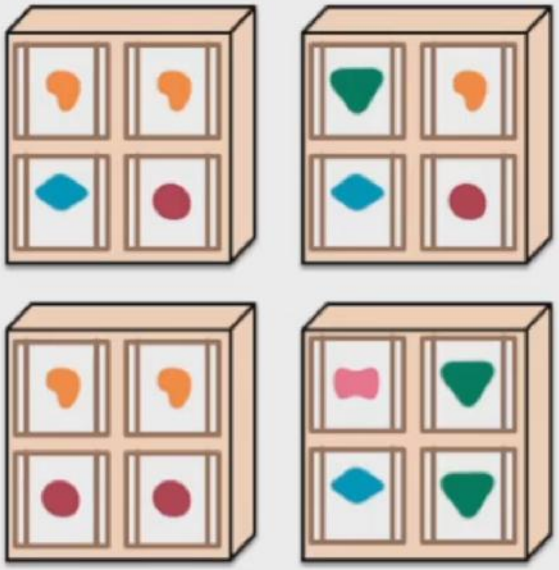
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...

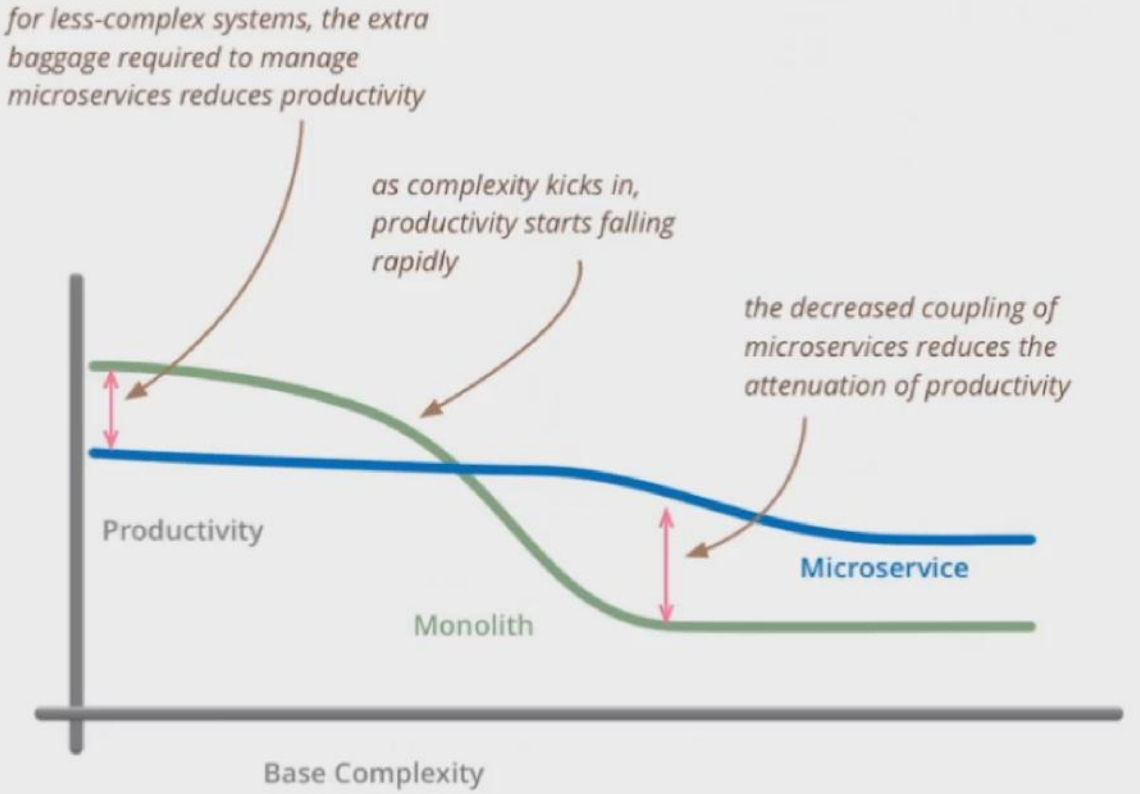


... and scales by distributing these services across servers, replicating as needed.



Il vantaggio dei microservices

- Piu produttivita per progetti piccoli



but remember the skill of the team will outweigh any monolith/microservice choice

Caratteristiche dei microservices

1. Strutturazione a componenti con servizi
2. Organizzati attorno alle competenze di Business
3. Prodotti non progetti
4. Smart endpoints e dumb pipes
5. Gestione decentralizzata del calcolo e dei dati
6. Automatizzazione del deployment
7. Progettati per i malfunzionamenti
8. Progettazione evolutiva

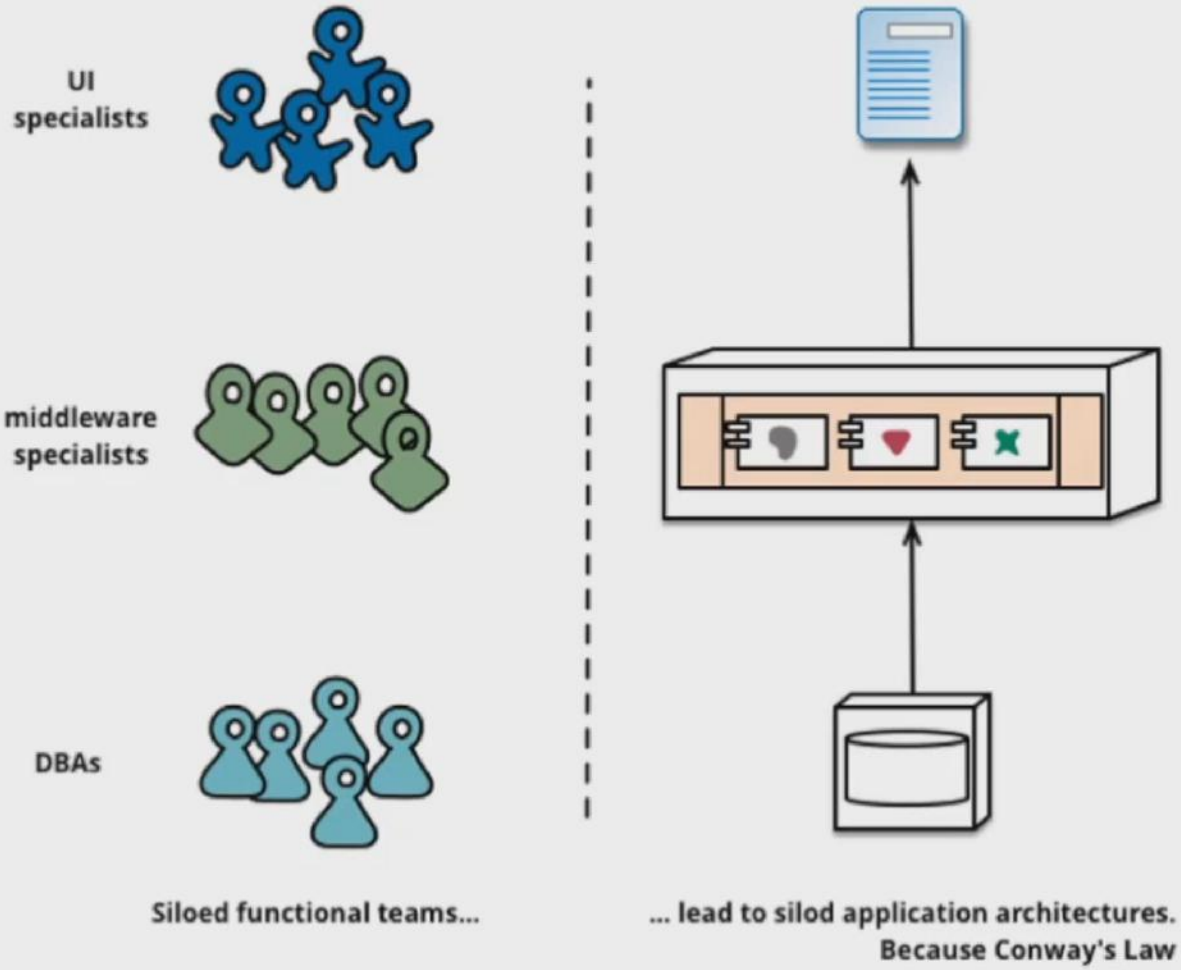
1. Componenti come servizi

- I servizi (rispetto alle componenti) sono indipendentemente deployable
- La decomposizione aiuta a eliminare la coesione
  - al minimo, almeno
- Esplicitazione dell'interfaccia dei servizi
  - Interface Definition Language
- Svantaggi
  - grana grossa delle invocazioni (per ammortizzare i tempi di comunicazione) con conseguente problemi nel partizionamento

La legge di Conway

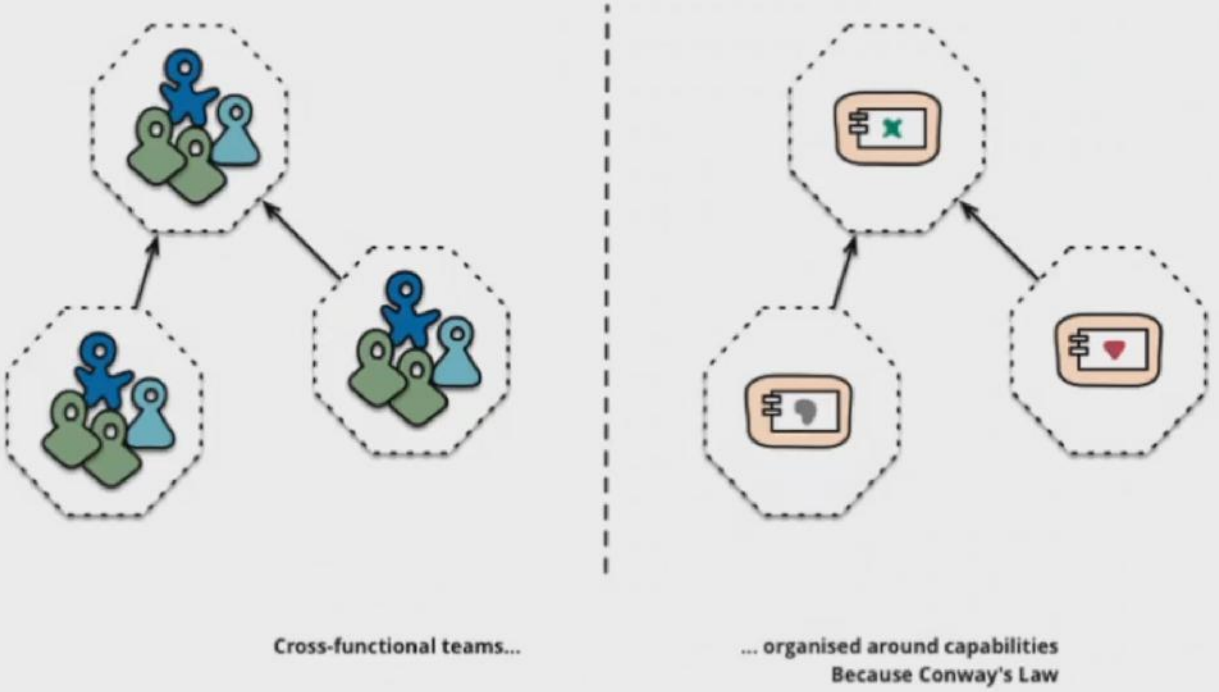
Un'organizzazione che progetta un sistema, produrrà un progetto la cui struttura è una copia della struttura di comunicazione dell'organizzazione

(Conway, 1967)



2. Microservices sulle competenze di business

- Team cross-functional
  - User experience
  - Database
  - Project management
- Multisciplinari
- Indipendenti
- Piccoli: da Amazon
  - "Two pizzas team"



Cross-functional teams...

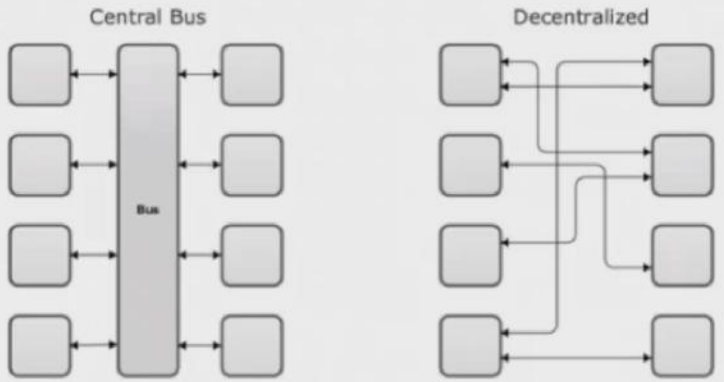
... organised around capabilities
Because Conway's Law

3. Prodotti non progetti: "You build it, you run it"

- Il team prende responsabilità completa del prodotto
- Anche in produzione
  - il team è in costante contatto con l'uso in produzione
  - migliorando il rapporto con gli utenti
- Mentalità a "prodotto"
  - non solo un insieme di funzionalità
  - ma un software che assiste gli utenti a migliorare il loro business
- Possibile anche con software monolitico
  - ma molto più agevole con i microservices

4. Smart endpoints e dumb pipes

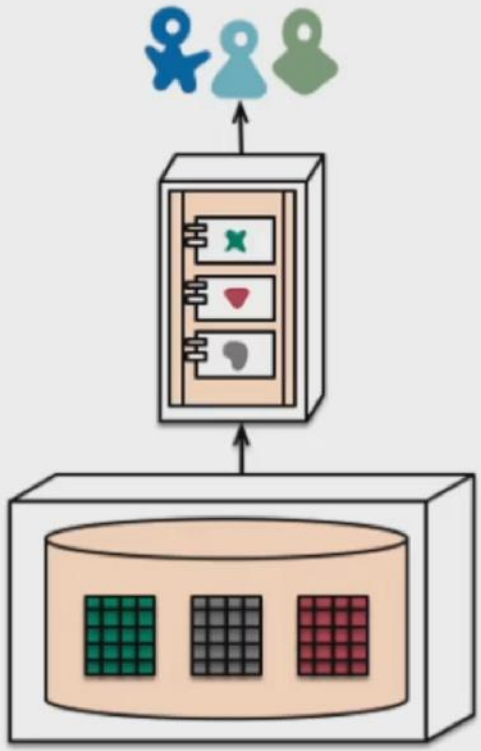
- Disaccoppiamento e coesione
  - Necessità che sembrano contraddittorie
  - Microservices devono mantenere la propria logica e agire da filtro
    - nel senso della pipe di UNIX
  - Ciclo: richiesta, applicazione della logica di business, risposta
  - Coreografia dei servizi
    - Per costruire servizi più complessi
  - HTTP come protocollo lightweight
    - ma anche uso di un Message bus (Message Oriented Middlewas)
    - la trama "asincrona" della computazione



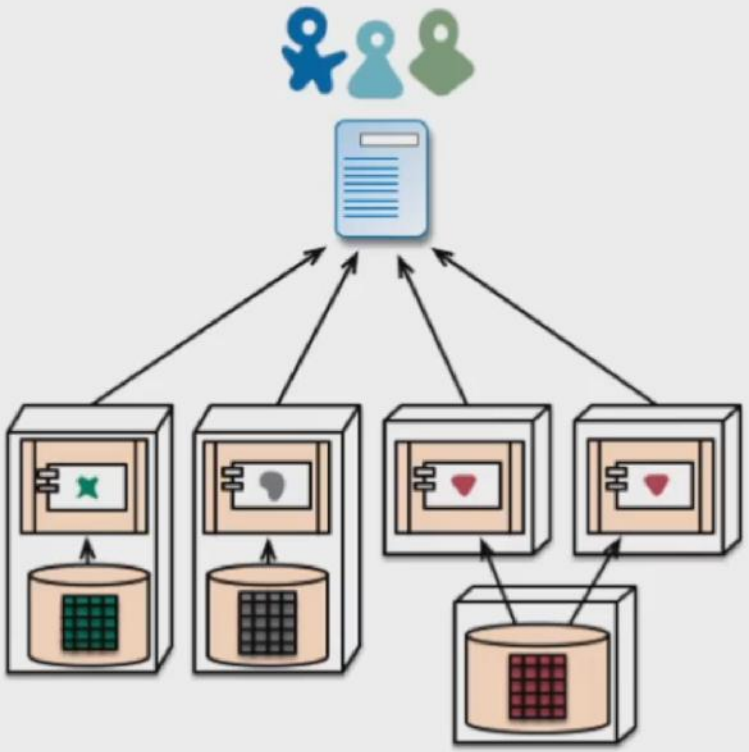
Standardizzazione? No grazie

- "Non esageriamo!"
- Invece di concentrarsi su un unico ambiente, usare l'ambiente giusto per ciascun microservice aiuta di molto
- Importante definire dei "contratti" (le interfacce)
  - La gestione dei contratti è però libera per chi implementa
- Amazon come esempio:
  - Ogni team è responsabile del software che costruiscono, compreso il 24/7
  - *"Essere svegliati alle 3 del mattino per un malfunzionamento, sicuramente aumenta la qualità del tuo codice"*

5. Gestione decentralizzata dei dati



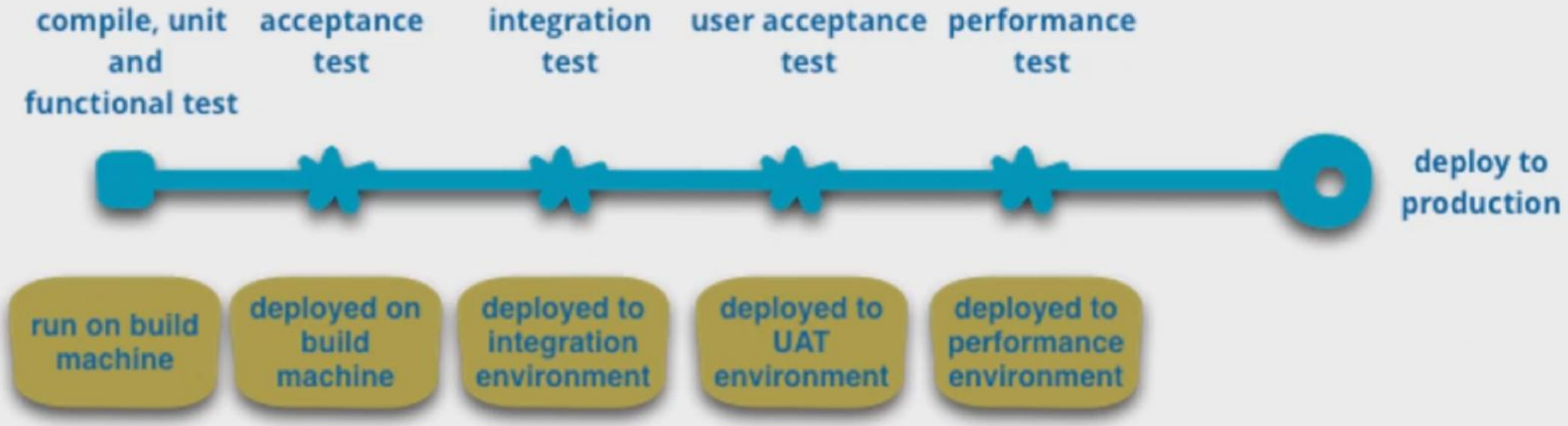
monolith - single database



microservices - application databases

6. Automatizzazione del deployment

- Infrastruttura per il continuous delivery



Caratteristiche dei microservices

1. Strutturazione a componenti con Servizi
2. Organizzati attorno alle competenze di Business
3. Prodotti non progetti
4. Smart endpoints e dumb pipes
5. Gestione decentralizzata del calcolo e dei dati
6. Automatizzazione del deployment
7. Progettati per i malfunzionamenti
8. Progettazione evolutiva

6. Progettati per i malfunzionamenti

- Tolleranza ai malfunzionamenti
- Ogni team ha chiaro in mente che altri microservices possano non funzionare e di come questo influenzi gli utenti
  - Non essendo sotto la loro responsabilità
  - Ma apparentemente può sembrare loro responsabilità
- Real time monitoring dei microservizi
  - Con azioni automatiche di restore del servizio
- Dashboards del sistema sono di aiuto

Gestione dei cambiamenti

- Per suddividere una componente si usa il criterio di "sostituibilità indipendente e aggiornabilità"
- Importante criterio per valutare la suddivisione in servizi:
  - se ci si trova a cambiare ripetutamente due servizi insieme, allora probabilmente vanno uniti
- Release planning effettuato per microservice invece che per applicazione
  - semplificazione e velocizzazione (non si deve attendere tutti gli sviluppi)
- Versioning dei servizi, una possibilità..
  - ma da evitare se possibile: meglio la tolleranza

Microservices: il futuro?

- Molte aziende che sono pioniere, evidenziano vantaggi
  - Amazon, Netflix, The Guardian
- Microservices e SOA
  - *"microservices are service orientation doing right"* ?? ☺
  - non proprio: l'enfasi sul team distribuito, responsabilità distribuite, etc. non sono presenti negli Enterprice Service Bus, che sono orientati a architetture monolitiche
- Il problema della organizzazione dei team:
  - multidisciplinari
  - con responsabilità complete del prodotto
  - con molte competenze (difficili da trovare?)

Serverless Computing

- La conclusione di un lungo viaggio
- Big software: la sua "morte"
- I Microservices
- Serverless Computing
- Conclusioni

Serverless computing

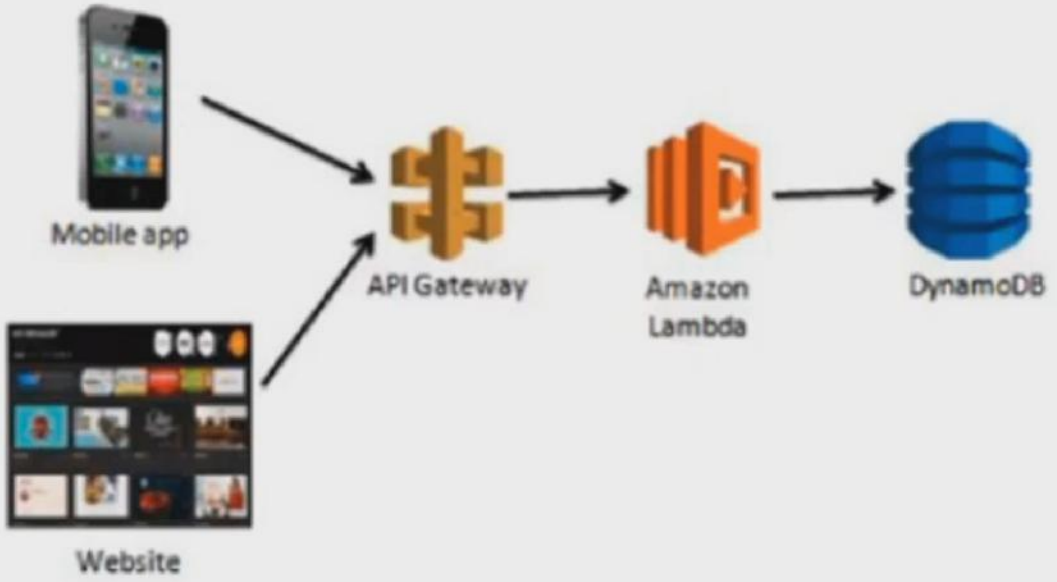
- Tecnologia conosciuta come "*Function as a Service*" (FaaS)

Serverless computing

- Tecnologia conosciuta come "*Function as a Service*" (FaaS)
- Il cloud provider fornisce gestione completa del container su cui la funzione viene eseguita
- Shift verso computazioni event-driven, loosely coupled computations
- Eventi da trattare, ad esempio
  - richieste API
  - oggetti che vengono inseriti / ricercati nello storage a oggetti
  - modifiche ad un database
  - schedule
  - comandi vocali, bots, etc.

Un esempio: Call da mobile/desktop su Amazon

- Esempio
  1. un API gateway, prende API HTTP request
  2. la passa ad una funzione Lambda
  3. che usa un servizio di DB



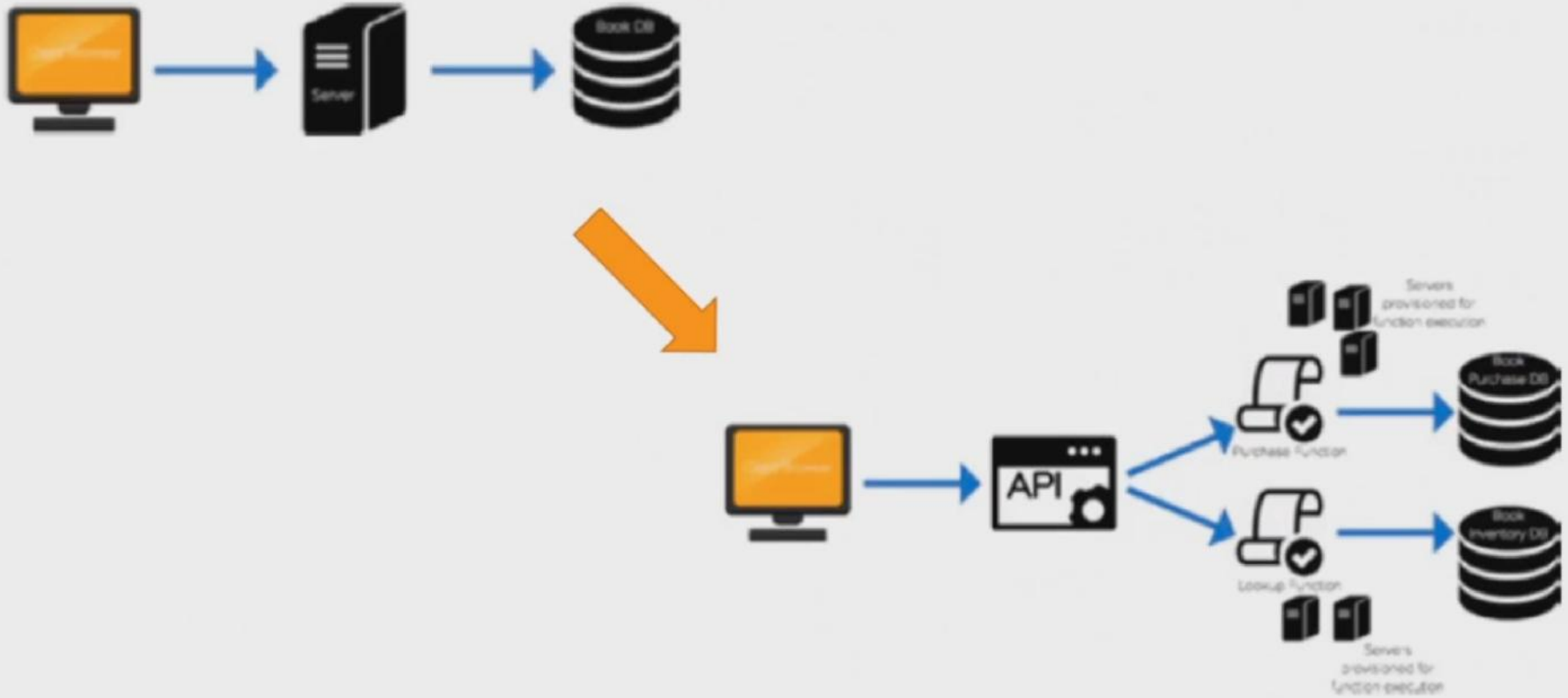
Functions as a Service (FAAS)

- FaaS

FaaS vs PaaS

- In PaaS, c'e' l'esecuzione costante di almeno un processo in background sul server
  - scalabilita raggiunta con l'aggiunta di piu server
  - Rimane visibile allo sviluppatore
- In FaaS, si esegue l'applicazioni in risposta a eventi
  - senza gestire i server
  - utilizzata on demand
    - modello di esecuzione basato su eventi
    - in questo modo è presente quando necessario
    - non richiede un processo server sempre attivo
      - Richieste generalmente piu lente alla prima richiesta, ma in seguito caching
    - in genere si paga per *function execution time*

Dalla architettura tradizionale a FaaS



Differenze sostanziali

- Sviluppo lievemente diverso, che usa gli strumenti forniti dal cloud provider
- Processi indipendenti
  - ogni funzione come un *microservice serverless*: stateless e event based
- Indipendenza, lightweight, scalabili

Benefici e casi di uso

- Sviluppo e deployment rapido
  - Infrastrutture, manutenzione, scalabilità, versioning, backup, etc. gestiti dal provider
- Facilità di uso
  - Varietà di funzionalità disponibili (anche molto evolute)
- Costo minore
- Scalabilità significativa
  - automaticamente fornita dal cloud provider sulla sua infrastruttura
  - non responsabilità dello sviluppatore

Costi di FaaS: un esempio

AWS Lambda	Azure Functions	Google Cloud Functions
First million requests a month free	First million requests a month free	First 2 million requests a month free
$0.20 per million requests afterwards	$0.20 per million requests afterwards	$0.40 per million requests afterwards
$0.00001667 for every GB-second used	$0.000016 for every GB-second used	$0.000025 for every GB-second used

I limiti del serverless computing:

- Richiesta di mantenere il controllo dell'infrastruttura
  - limitato alla parte di infrastruttura hw
  - limitato controllo possibile da utente
- Non utile per procedure batch (esecuzione lunga)
- Vendor (cloud vendor) lock-in
  - possibile una rifattorizzazione del codice per limitare il problema in poche parti del codice
- "Cold start" di una funzione (dopo inattività)
  - "ping" regolare da schedulare
- Infrastruttura condivisa (con concorrenti? Netflix e Disney ☺)

Conclusioni

- La morte del Big software
- Microservices
- Serverless Computing
- Conclusioni