Tecnologie Cloud e Mobile

Lez. 13

Micro-Services e ReST

Giuseppe Psaila

Università di Bergamo

giuseppe.psaila@unibg.it

Problemi nello Sviluppo Delle Applicazioni Web

Problematiche di Gestione

- La tecnologia della programmazione e delle basi di dati consente di realizzare funzionalità (servizi) molto sofisticati
- Ormai, tutti i sistemi informativi sono applicazioni web (o quasi tutti, i «sistemi **legacy**» non lo sono)
- Come gestire lo sviluppo e il rilascio di applicazioni web di grandi dimensioni?

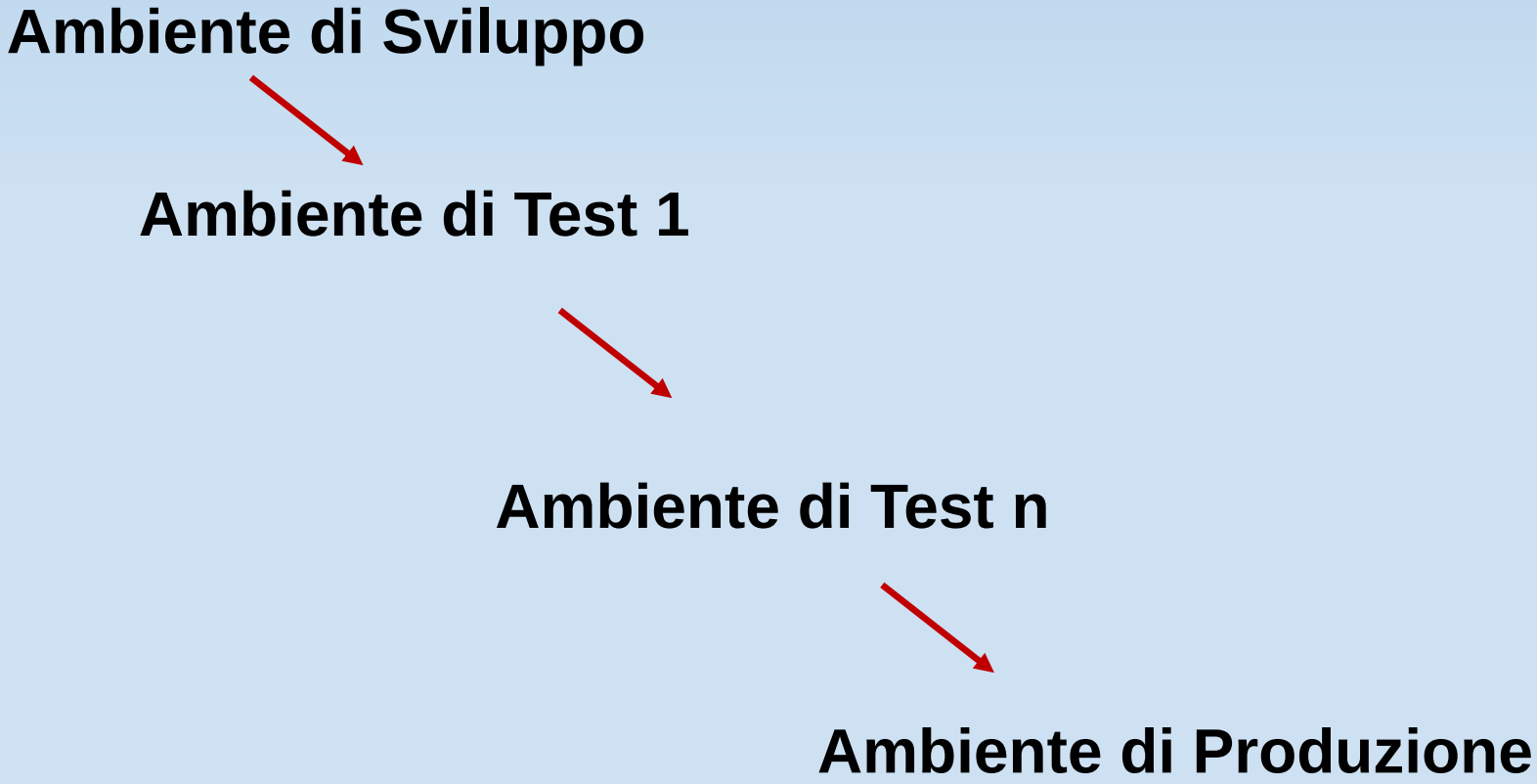
Il Processo di Deploy

- Lo sviluppo del software non può avvenire sulle stesse macchine su cui viene eseguito per fornire il servizio agli utenti
- Perché lo sviluppo delle nuove funzionalità rende non funzionante il software, fino a che queste non siano state completate

Il Processo di Deploy

- **Ambiente di Produzione**
  I sistemi che forniscono il servizio agli utenti
- **Ambiente di Sviluppo**
  I sistemi sui quali si sviluppa il nuovo software
- **Ambienti di Test**
  I sistemi sui quali si effettuano i test

Deploy da un ambiente ad un Altro



Il Processo di Deploy

- Ogni passaggio di ambiente si chiama «**Deploy**»
- Il Deploy in produzione è il più critico perché un errore di installazione causa malfunzionamenti verso gli utenti finali

Il Processo di Deploy

- Ma come testare il software nel modo più realistico possibile?
- Negli ambienti di test si creano delle copie integrali (o molto estese) dei dati nei database dell’ambiente di produzione
- Alcuni sistemi chiamano gli ambienti di test «**sandbox**»

Il Deploy sulle PaaS nel Cloud

- Una buona Platform as a Service DEVE fornire il supporto a sviluppo, testing e deploy
- Con la gestione integrata del Versioning
- In questo modo, la PaaS consente di gestire le dipendenze tra parti del software ed automatizzare il processo di deploy
- **Il processo di deploy NON DEVE essere effettuato a mano!!!**

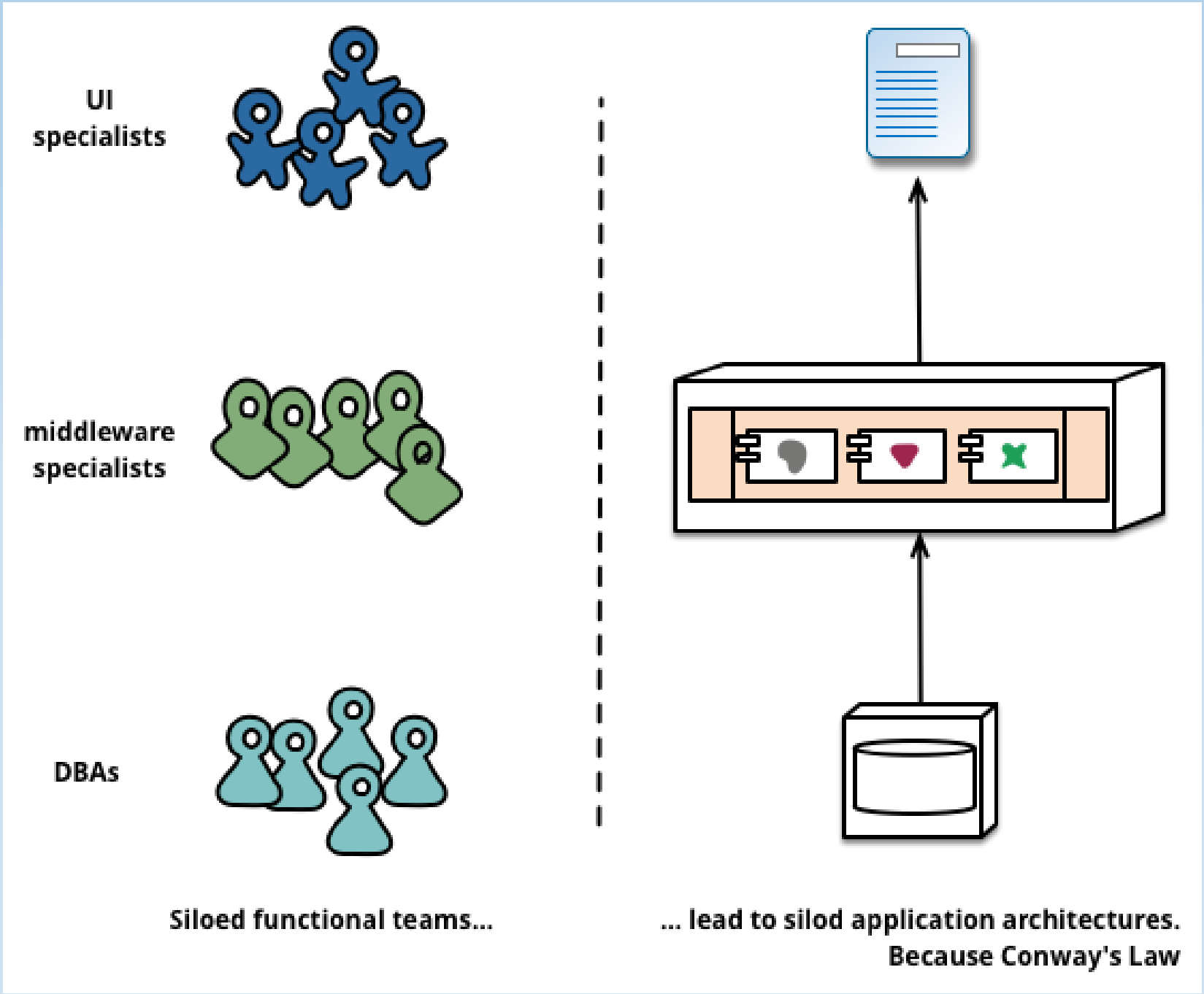
Modularizzazione del Software

- Per controllare la complessità dello sviluppo, è bene «modularizzare il software»
- Cioè, suddividerlo in «moduli» o «componenti», ciascuno dei quali contribuisce a realizzare il servizio finale
- Ma come suddividere?

Layer

- Nelle applicazioni web, una suddivisione maturarle è data dalla 3-tier architecture
  - **View**: HTML + JavaScript
  - **Controller o Server side**: i programmi lato server che controllano la dimanica e realizzano la business logic dell’applicazione
  - **Model o Data Layer**: il database che raccoglie i dati che rappresentano il modello della realtà su cui si opera

Layer


UI specialists
middleware specialists
DBAs
Siloed functional teams...
... lead to silod application architectures.
Because Conway's Law

Layer: Vantaggi e Svantaggi

- **Vantaggi**: suddivisione per **competenze tecniche**, cioè il team dell'interfaccia ha il controllo sull'intera applicazione (**omogeneità**)
- **Svantaggi**: i team sono grossi e devono continuamente parlarsi tra di loro e tutti con gli utenti finali: **elevato tempo di coordinamento necessario**

Sistema Monolitico

Tipicamente, il sistema risultante è «**monolitico**», cioè

- Le varie parti del sistema vengono sviluppate in base alle necessità dei programmatori, cercando di riusare il codice per quanto possibile.
- **Effetto**: al crescere della complessità del sistema diventa sempre più difficile fare modifiche, perché queste possono avere un impatto molto diffuso

Modularizzazione Verticale

- Per controllare la complessità, e spesso anche solo per poter compilare il programma,
- Si creano i «moduli»
- Un modulo è un insieme di funzioni (e di funzionalità) in qualche modo omogenee tra di loro
- Il concetto di namespace aiuta a evitare i conflitti tra moduli diversi integrati nella stessa applicazione

Modularizzazione Verticale

- Ma come si decide quali funzioni mettere in un modulo?
- In base alla **sensibilità** e alle **necessità** dei programmatori
- Privilegiando una visione «**in piccolo**», più che una visione «**in grande**»
- Per esempio, l'approccio object oriented va proprio in questa direzione, «**modularizzazione in piccolo**» (o «**design in piccolo**»)

Le Librerie

- Le «librerie» sono un’evoluzione del concetto di modularizzazione
- Sono insiemi di funzionalità volti a fornire ai programmatori soluzioni sofisticate per problemi non banali
- Per esempio: gestire i formati geografici e operare trasformazioni su questi

Le Librerie

- **Problema**: essendo un’evoluzione del concetto di modulo, sono ancora pensati per una «modularizzazione in piccolo»
- Si pensi a Python, dove le librerie sono moduli

Modularizzazione in Piccolo

- La modularizzazione in piccolo non risolve i problemi dei sistemi monolitici
- Perché, anche se modularizzati al loro interno, rimangono monolitici
- Tipicamente, la ricerca del **riuso estremo del software**, attraverso l'approccio object-oriented, **crea moltissime dipendenze** tra parti fortemente interconnesse

Modularizzazione in Piccolo

- **Problema**: la forte interconnessione rende molto difficile modificare parti anche piccole del software, perché le modifiche hanno impatto, in cascata, sulle molteplici altre parti che dipendono da queste

La Dimensione dei Team

- La dimensione dei team di sviluppo è un aspetto non trascurabile
- **Perché più un team è grosso, più è rigido e di difficile controllo**
- Inoltre, **un team grosso tende a imporsi sugli altri team**, con conseguenti discussioni e lotte tra i team

La Dimensione dei Team

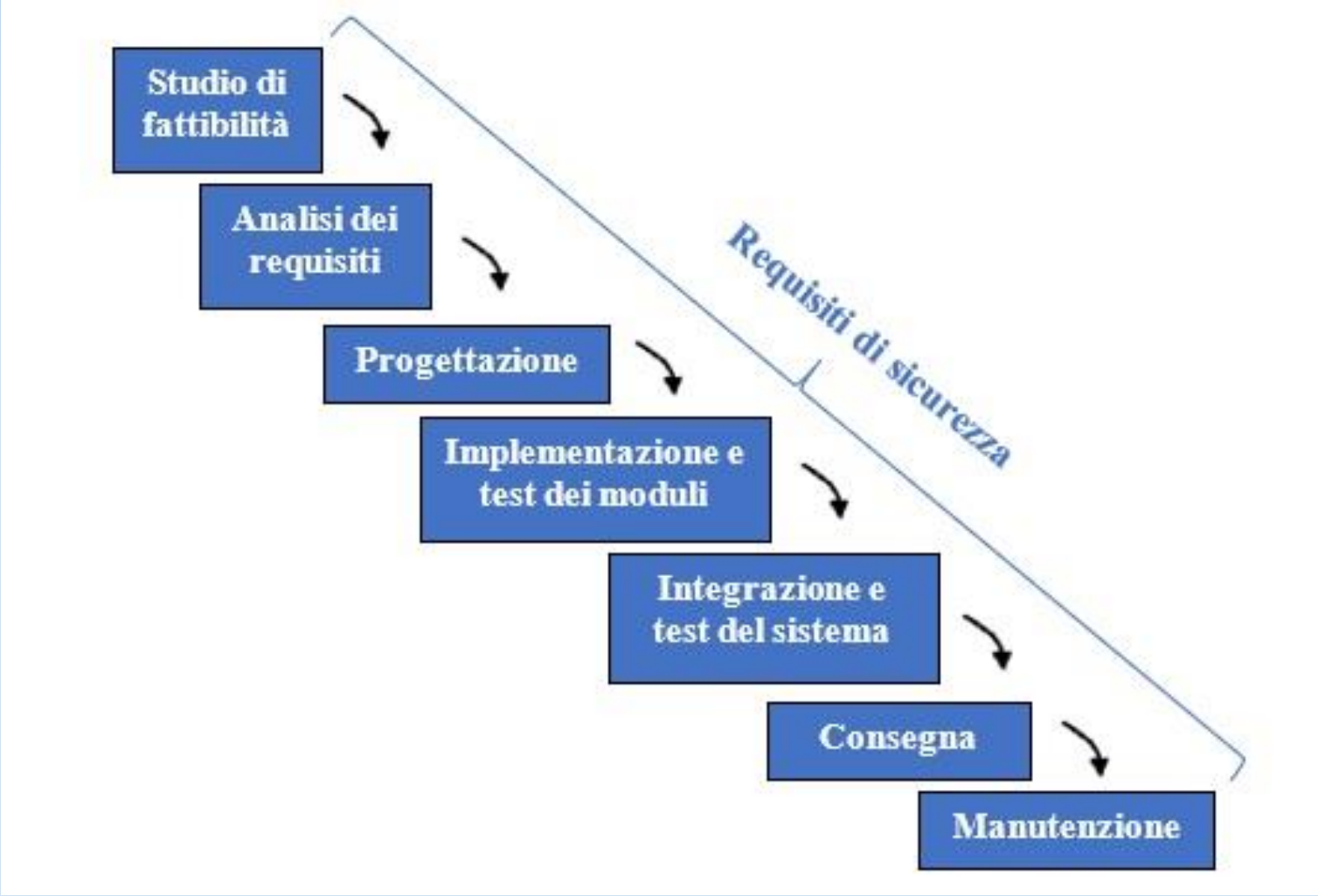
- Un team grosso è più focalizzato su come scrive il codice che sul servizio offerto da questo
- La focalizzazione è sul «progetto»
- Una volta terminato, il progetto è finito, i componenti del team vorrebbero occuparsi di altro
- Ma il software va «mantenuto»: chi lo mantiene?

Cicli di Vita del Software

- **A Cascata**
  Dalla Specifica dei requisiti alla consegna, senza ripensamenti, poi si fa la manutenzione del software
- **A Spirale**
  Lo sviluppo avviene per affinamenti successivi, ridefinendo le specifiche

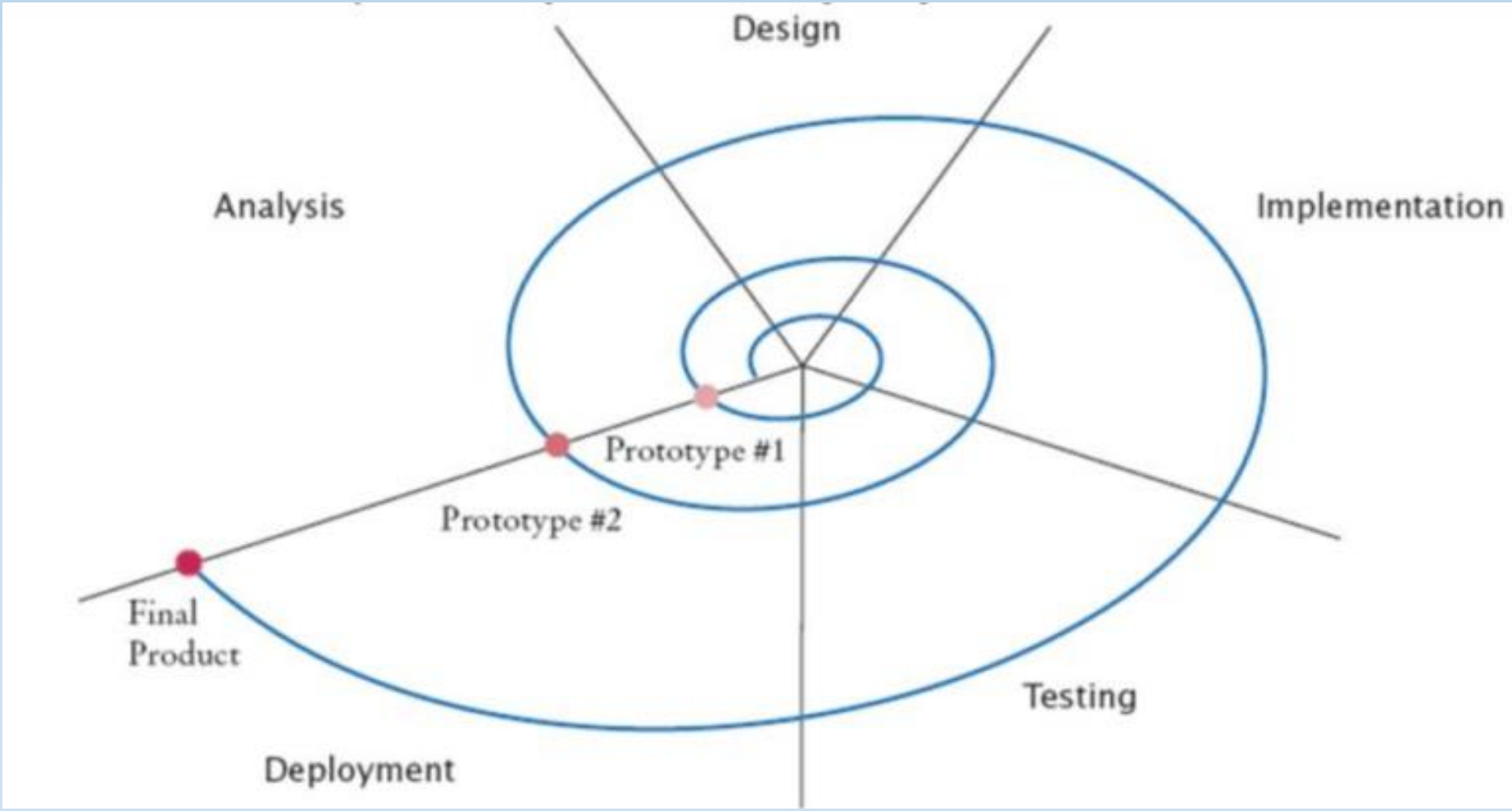
Cicli di Vita del Software

- A Cascata



Cicli di Vita del Software

- A Spirale



Cicli di Vita del Software

- **Approccio Agile**
  Evolve il modello a spirale, organizzando il team in unità molto piccole, ma con responsabilità ben precise
- Esempio: salesforce.com è passata, dopo soli 2 anni di vita, dal modello a cascata all'approccio agile
  In questo modo riescono a rilasciare tre nuove release ogni anno

Aprroccio a Cascata

- Nell’approccio a cascata, dopo il rilascio, il team si disinteressa del sistema
- Che tipicamente viene preso in carico da un altro team
- Ma questo nuovo team non lo ha sviluppato
- Quindi fa fatica a tenerne sotto-controllo la complessità e, di conseguenza, a fare le modifiche

Scalabilità

- Un problema tipico nella gestione dei sistemi informativi è la «**scalabilità**»
- Cioè la possibilità di adattarsi a carichi di lavoro crescenti

Scalabilità

- Il cloud computing ha messo una pezza al problema
- Perché consente di aumentare le risorse di calcolo a disposizione con estrema facilità
- Ma ha solo attenuato il problema
- Quali sono le fonti della «**non scalabilità**»?

Scalabilità

Le fonti della «non scalabilità» possono essere diverse

- Codice scritto male o basato su strutture dati inefficienti
- Algoritmi intrinsecamente non scalabili
- Funzionalità ridondanti, che devono, comunque, essere eseguite per svolgere il lavoro
- In ogni caso, **sono parti del sistema, non tutto il sistema**

Scalabilità e Monoliti

- Ma un sistema monolitico è un tutt'uno
- Può essere eseguito in un unico processo di sistema
- Quindi, si deve scalare l'intero sistema
- **Se si potesse spezzare il sistema monolitico in sotto-sistemi diversi, si potrebbe scalare solo il sotto-sistema critico**

SOA: Service-Oriented Architecture

- Un primo tentativo (dal punto di vista storico, circa 2005) è stato il concetto di Service-Oriented Architecture
- L’idea è di organizzare il sistema per composizione di sotto-sistemi
- Ogni sotto-sistema fornisce servizi ben definiti
- Richiamabili attraverso meccanismi di comunicazione

SOA

Quali meccanismi di comunicazione?

- RPC, Remote Procedure Calls
- Protocollo SOAP, Simple Object Application Protocol
- Web Services, con HTTP
- Message Queuing

SOA

- Purtroppo, le cose non sono andate come sperato
- L’approccio SOA ha portato alla realizzazione di «**sotto-sistemi monolitici**»
- Cioè un sistema è formato da vari sotto-sistemi, ciascuno di questi monolitico
- Effetto: invece di ridurre i tempi e i costi di evoluzione, questi sono ulteriormente aumentati

SOA

- Eppure l’idea è buona
- Dove era il problema?
- Nell’**approccio allo sviluppo** e nella **dimensione dei team**
- Dall’esperienza pratica, nasce il concetto di **Micro-services**

I Micro-Services

Approccio a Micro-servizi

- Che cosa è?
- È uno «stile architetturale»
- Cioè un modo di organizzare l'architettura del sistema informativo
- L'idea è la seguente:
  organizzare il sistema come una miriade di sotto-sistemi, totalmente indipendenti, che comunicano tra di loro attraverso la rete

Caratteristiche

- **Molti stack**
  Ogni servizio contiene più o meno l’intero stack tecnologico necessario per realizzarlo
- **Molti server**
  Ogni servizio viene eseguito su un server (o un gruppo di server) specifico per il servizio
  In questo modo, invece di avere un unico punto di «failure», se ne hanno tanti

Caratteristiche

- **Tolleranza ai guasti**
  Il fatto che un servizio si basi su altri servizi, costringe il programmatore a gestire l'eventuale failure dei servizi su cui si appoggia
- **Logging e Monitoring**
  I servizi devono continuamente registrare le attività svolte e i malfunzionamenti
  Occorre predisporre un servizio di «monitoring», per tenere sotto controllo lo stato dell'intero sistema

Caratteristiche

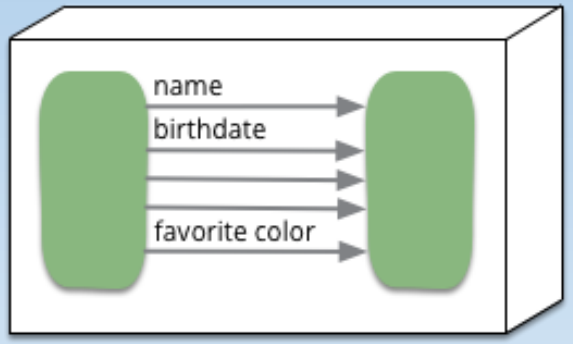
- **Chiara suddivisione dei compiti svolti dai vari micro-servizi**
  Ogni micro-servizio deve avere un compito ben preciso da svolgere: in fase di «design in grande», il «system architect» deve capire chi fa che cosa
- **Chiara definizione dei confini del micro-servizio**
  Il system architect definisce in modo chiaro e preciso i confini del micro-servizio: tutto ciò che non è di sua responsabilità, non è considerato (lo fa un altro micro-servizio)

Caratteristiche

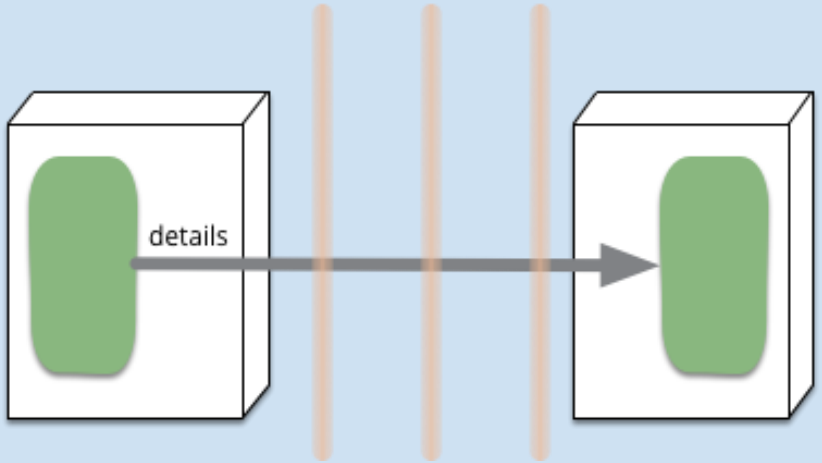
- **Chiara definizione delle interfacce (API) del micro-servizio**
  L'interfaccia fornita da ogni micro-servizio deve essere ben documentata, perché è attraverso questa interfaccia che gli altri micro-servizi lo usano

Caratteristiche

- **Chiara definizione delle interfacce (API) del micro-servizio**



With two modules in the same process, it's best to use many fine-grained calls…



…but when modules are remote, then favor few coarse-grained calls.

Caratteristiche

- **Deploy indipendente**
  Ogni micro-servizio è sviluppato in modo «indipendente» dagli altri, su server indpendenti; anche il processo di deploy deve quindi essere indipendente

Caratteristiche

- **Interoperabilità**
  Lo sviluppo indipendente, il deploy su server indipendenti e le API ben documentate e indipendenti DEVONO consentire di sostituire un servizio con uno equivalente, senza che il resto del sistema se ne accorga

Vantaggi dei Micro-Services

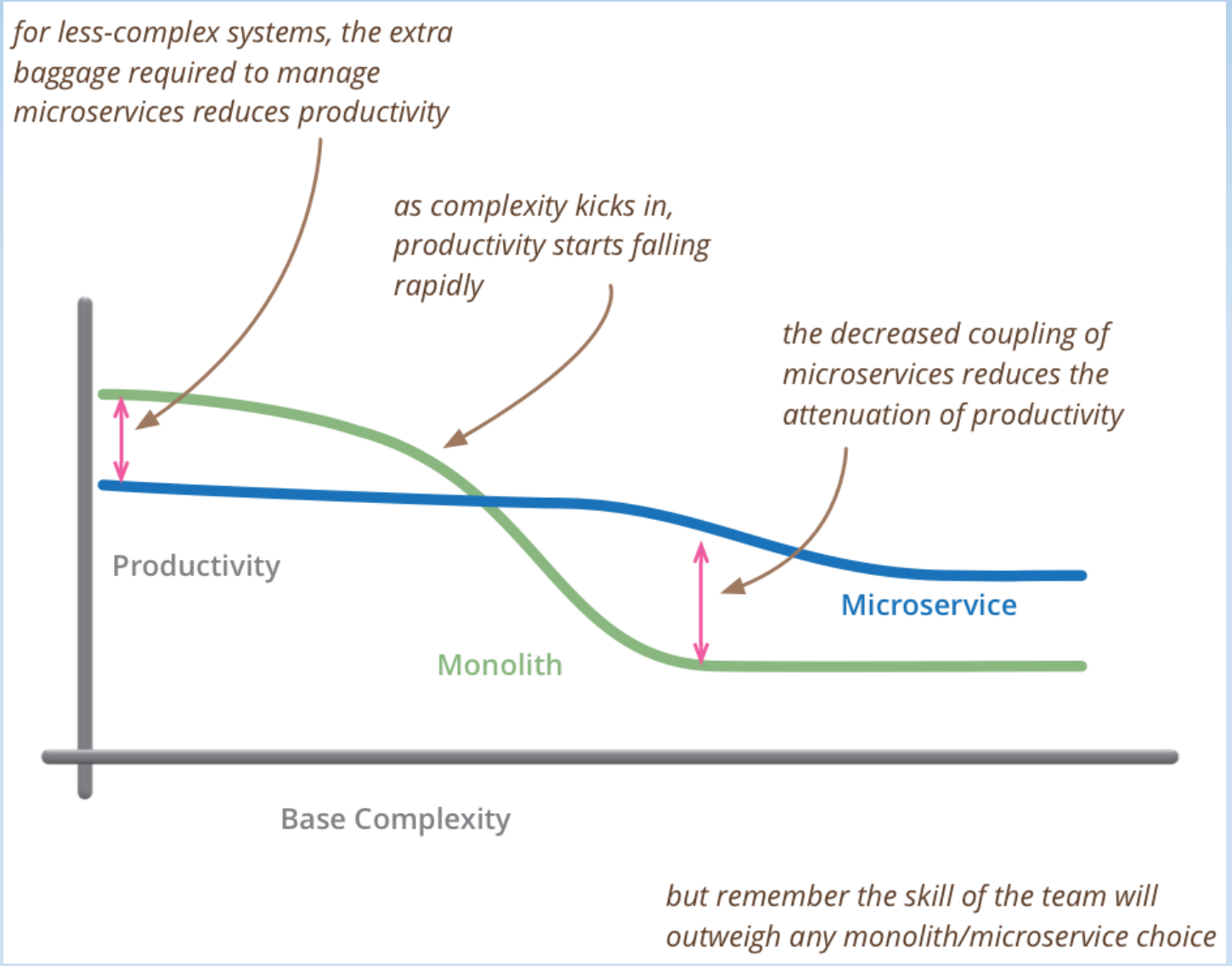
- **Manutenibilità**
  Il codice di un micro-servizio risulta (deve risultare) piccolo, facile da modificare
- **Reimplementazione dei servizi**
  Se un servizio non fornisce prestazioni soddisfacenti, può essere facilmente modificato o sostituito con uno più performante
- **Scalabilità**
  Solo i server che eseguono i servizi più critici dal punto di vista della scalabilità andranno potenziati

Vantaggi dei Micro-Services

- **Riduzione dei costi**
  Nelle prime fasi di vita del sistema, i tempi e i costi sono più alti dell’approccio monolitico, ma poi, al crescere della complessità, sia i tempi che i costi diventano più bassi dell’approccio monolitico

Vantaggi dei Micro-Services

- Riduzione dei costi

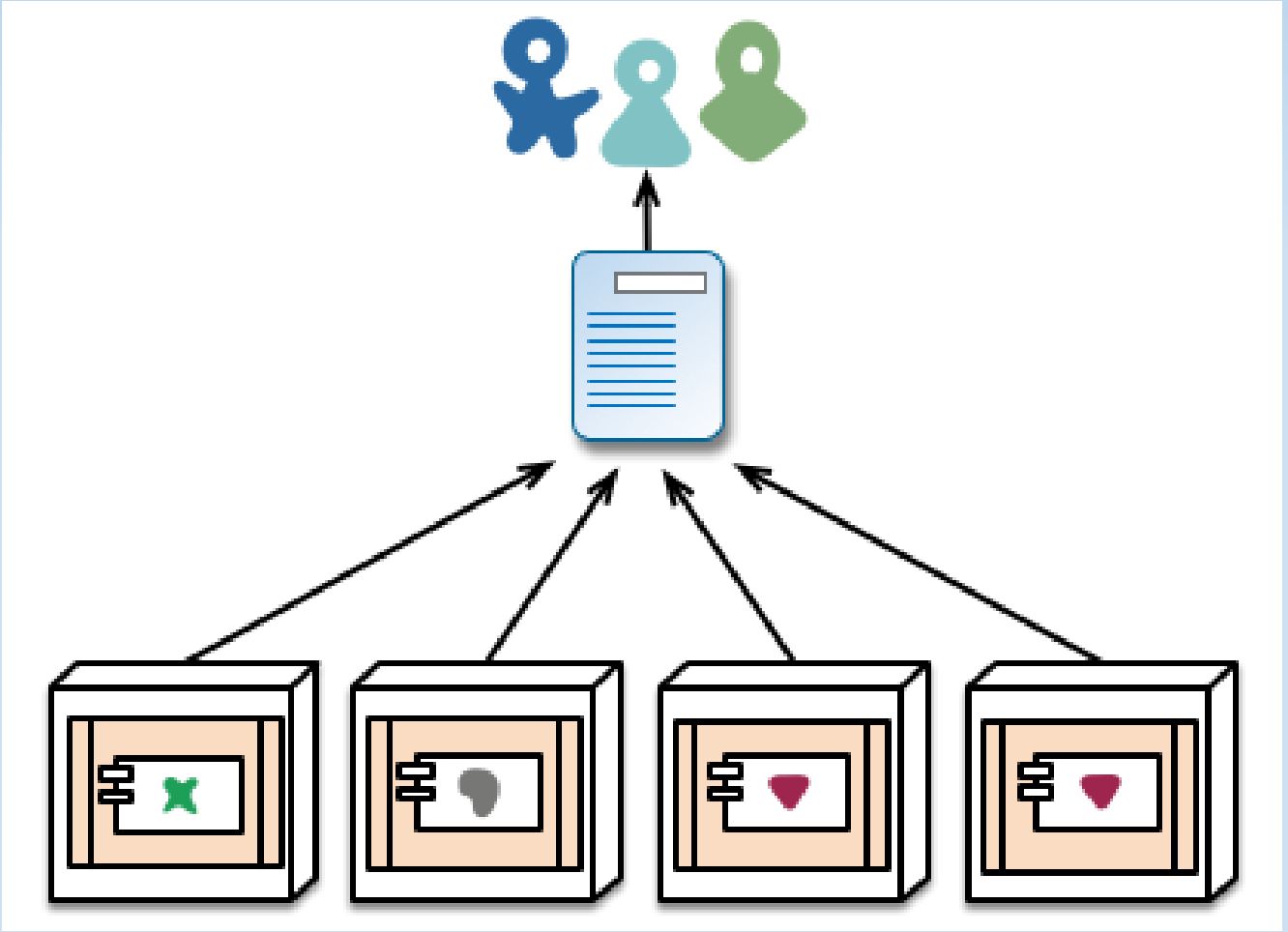


Vantaggi dei Micro-Services

- **Facile aggiunta di funzionalità**
  Aggiungere funzionalità al sistema informativo vuol dire aggiungere alcuni micro-servizi, da integrare modificando alcuni micro-servizi di coordinamento

Vantaggi dei Micro-Services

- Facile aggiunta di funzionalità

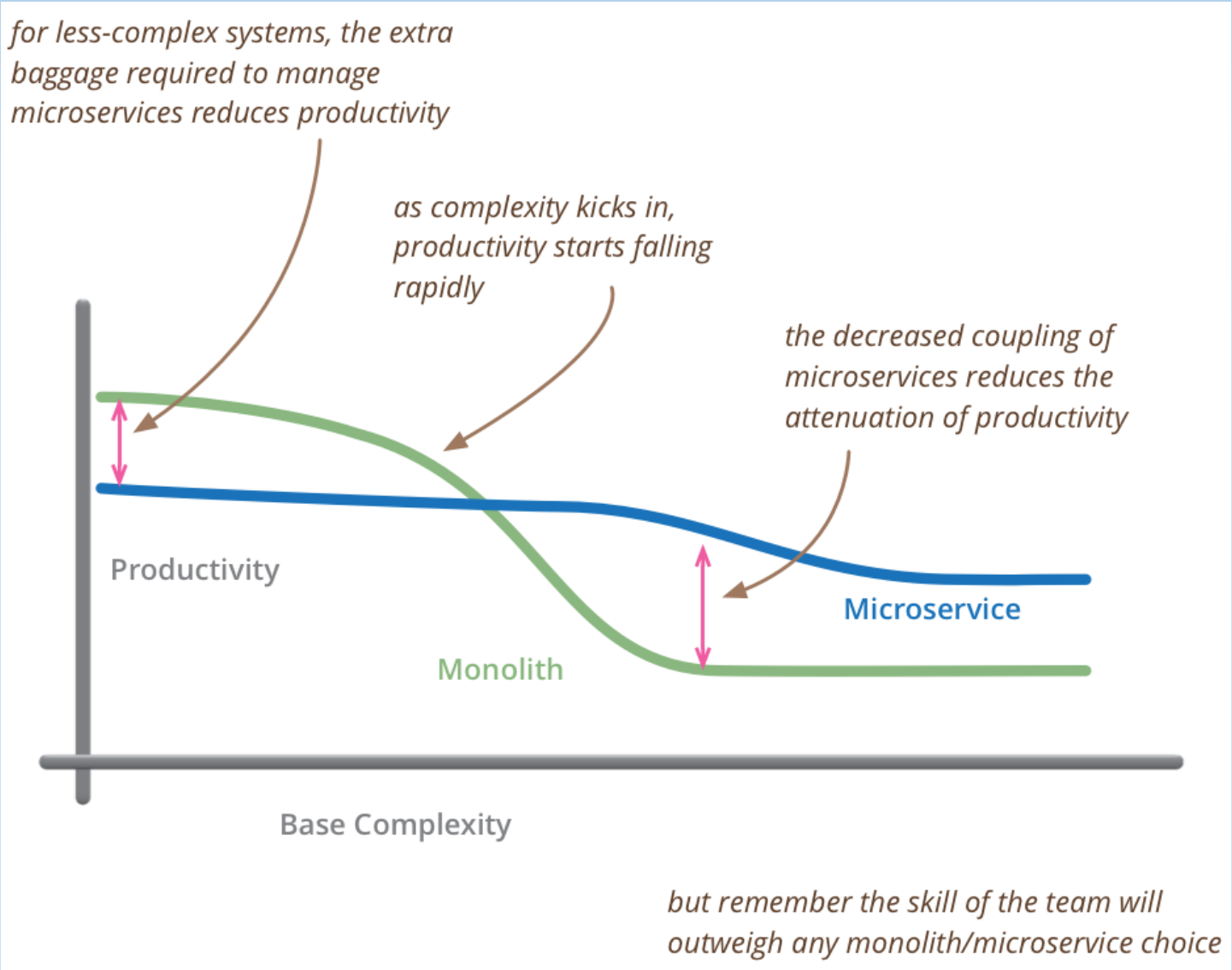


Svantaggi dei Micro-Services

- **Maggiori costi iniziali**
  All’inizio, i tempi e i costi di sviluppo sono pù alti, perché occorre:
  - Cambiare mentalità
  - Scrivere il codice tenendo conto della gestione dei malfunzionamenti degli altri micro-servizi
  - Gestire la possibile ridondanza dei dati

Svantaggi dei Micro-Services

- **Maggiori costi iniziali**



Svantaggi dei Micro-Services

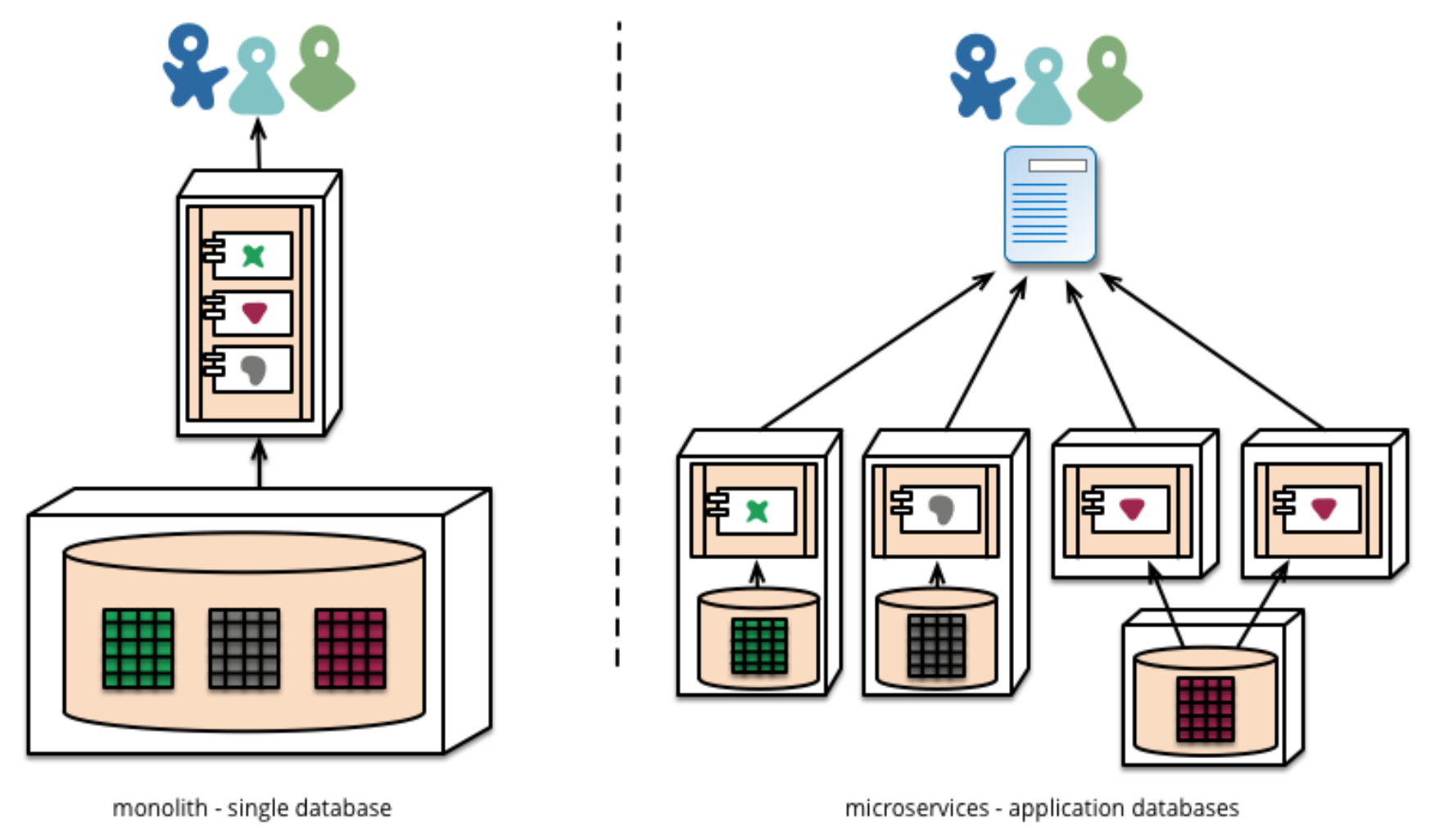
- **Tecnologie eterogenee**
  Servizi diversi possono essere realizzati con tecnologie diverse. Conseguenze:
  - Non si ha più un linguaggio (o una serie di linguaggi) standard a livello aziendale
  - Si potrebbe pensare di finire in una situazione caotica
  - Ma si può scegliere la migliore tecnologia per ogni singolo micro-servizio

Svantaggi dei Micro-Services

- **Molti data store**
  Ogni micro-servizio può avere il suo (o i suoi) data store. Conseguenze:
  - Ogni micro-servizio usa la tecnologia più adatta, senza interferire con il resto del sistema
  - Ma questo porta a non avere più un unico modello aziendale dei dati
  - E, necessariamente, crea ridondanza nei dati
  - Non ci si può appoggiare ad un servizio transazionale centralizzato e unico

Svantaggi dei Micro-Services

- Molti data store



Aspetti Organizzativi

- Si creano piccoli team, di poche persone
- Ogni team ha la piena responsabilità di gestire alcuni micro-servizi, per tutta la loro vita operativa
- Ma quanto «micro» deve essere un servizio?
- Non c’è una misura unica, dipende dal servizio
- Tipicamente, lo si lega ad una funzionalità di business ben precisa (per esempio, effettuare il pagamento con carta di credito)

Aspetti Organizzativi

- Ma allora, quanto grande è un team e quanti micro-servizi può/deve gestire?
- Approccio Amazon:
  «Two-Pizza Team»
  Una pizza americana è per 6 persone, quindi 2 pizze fanno mangiare 12 persone
  In Amazon, i team non superano le 12 persone e coprono tutto lo stack tecnologico

Aspetti Organizzativi

- Altro approccio:
  «Half douzen, Half douzen»
  cioè un team di una mezza dozzina di persone gestisce una mezza dozzina di servizi
- Approccio Agile
  Se poi i team applicano le modalità di lavoro agile, dovrebbero essere in grado di ottenere una elevata efficacia e, di conseguenza, una elevata efficienza nello sviluppo e aggiornamento del micro-servizio

Aspetti Organizzativi

- **Riduzione dei costi di coordinamento**
  Visto che ogni team è focalizzato sulla gestione ed evoluzione di un micro-servizio,
- Le necessità di coordinare i team tra di loro si riducono di molto
- Riducendo, di conseguenza, i costi impliciti dell’attività di coordinamento

Come Invocare un Micro-Service

- In principio, una qualsiasi modalità di invocazione dei micro-servizi potrebbe essere utilizzata
- Ma alla fine si usa HTTP (o HTTPS)
- Perché la soluzione migliore è che i micro-servizi siano realizzati secondo lo stile architetturale ReST

ReST

ReST

- Representational
- State
- Transfer

Che Cosa È?

- È uno stile architetturale
- Ancora una volta, un modo di organizzare l’architettura con cui i servizi sono definiti
- Purtroppo, viene confuso con i Web Service richiamabili con il protocollo HTTP

Chiariamo

- Un Web Service è un servizio orientato ad altre applicazioni
- Che espone delle API (Application Programming Interface) usate da altre applicazioni per ottenere servizi, fornendo e/o ottenendo dati
- Siccome viene richiamato effettuando richieste HTTP, prende il nome di «Web Service»

Servizi Web e ReST

- Un servizio web può essere sviluppato con lo stile architetturale ReST
- Ma può essere sviluppato senza seguire questo stile architetturale

ReST e ReSTful

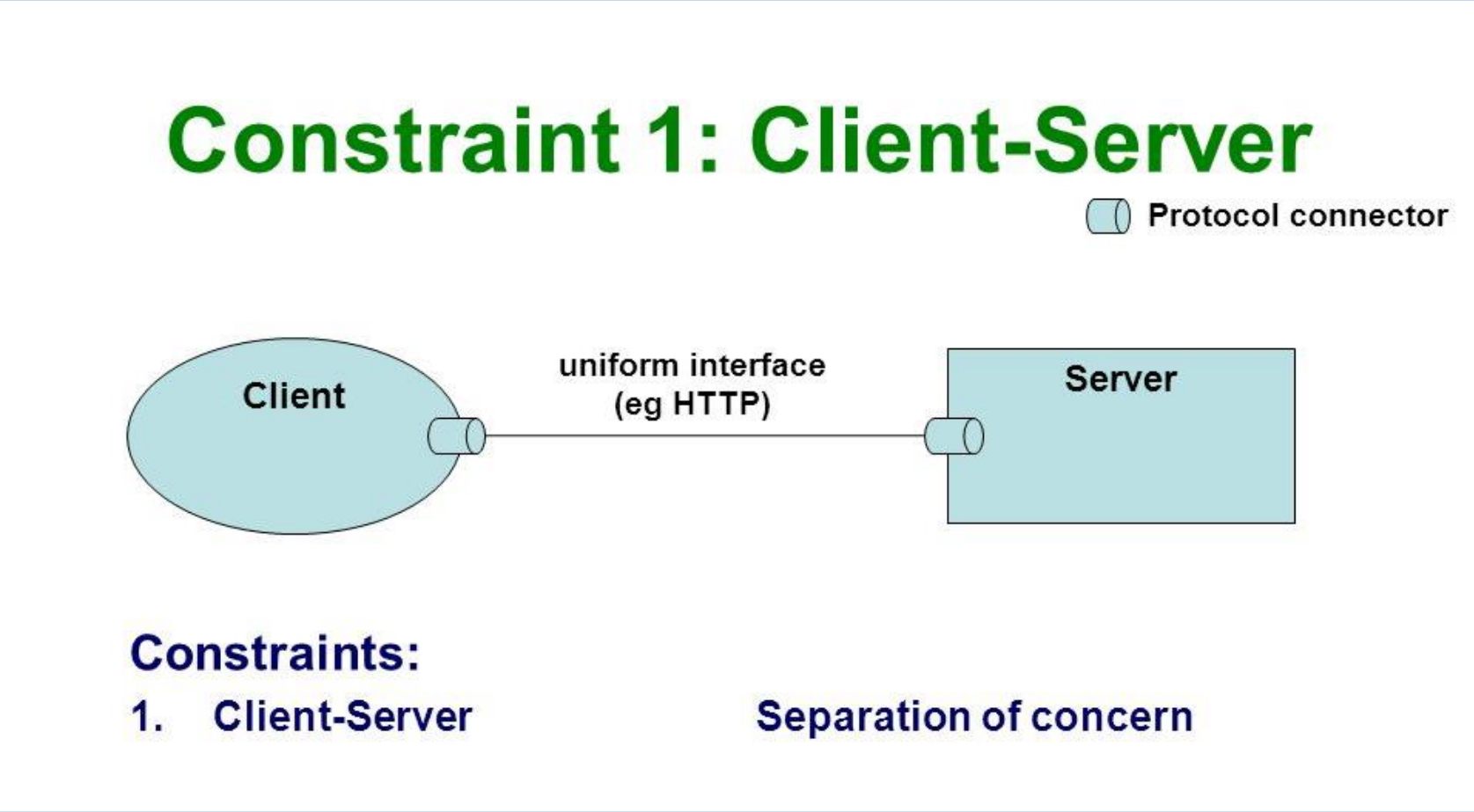
- Lo stile ReST impone alcuni vincoli sul modo in cui realizzare un architettura basata sui servizi
- Un servizio «ReSTful» o un’architettura ReSTful indicano che il servizio o l’architettura rispettano i vincoli/principi definiti dallo stile ReST
- I vincoli/principi di ReST sono focalizzati sul modo in cui i dati vengono trasmessi

Principi ReST

- **Client-Server**
  Un architettura ReST è client-server. Cioè un servizio fa da server ad un altro servizio/applicazione che funge da client
- Questo per separare chiaramente gli ambiti di intervento del client e del server

Principi ReST

- Client-Server



Principi ReST

- **State-less**
  La comunicazione deve essere senza stato, cioè non deve essere basata sullo scambio di messaggi multipli tra client e server
- Lo scambio di messaggi multipli, anche distanti nel tempo, costringe il server a mantenere lo stato della comunicazione con molti client
- State-less: il lavoro richiesto al server si esaurisce con la risposta alla richiesta

Principi ReST

- **Cache**
  Visto che un servizio fornisce dei dati, se le condizioni che portano a fornire quei dati non sono cambiate, due richieste vicine nel tempo che riguardano la stessa risorsa dovrebbero ottenere esattamente la stessa risposta
- Questo consente di sfruttare la memoria cache dei browser o dei proxxy per non rieseguire le richieste (usando la vecchia risposta alla stessa richiesta)

Principi ReST

- **Interfaccia Uniforme**
  I contenuti dei messaggi scambiati tra client e server devono essere uniformi
- Si intende che devono essere basati su un formato standard
- Indipendente, il piùù possibile, dall’applicazione

Principi ReST

- **Interfaccia Uniforme: Risorse**
  Una risorsa è un oggetto o la rappresentazione di qualcosa di significativo nel dominio applicativo. È possibile interagire con le risorse attraverso le API.
- Una richiesta al servizio richiede una risorsa
- Esempio: un prodotto di Amazon

Principi ReST

- **Interfaccia Uniforme: Manipolazione attraverso Rappresentazioni**
  La stessa risorsa può essere rappresentata inmolti modi: XML, JSON, PNG
  Il servizio può fornire rappresentazioni diverse per la stessa risorsa
- Esempio: il prodotto Amazon è rappresentato con un XML o con un JSON
  Il client usa quella rappresentazione per gestire il prodotto

Principi ReST

- **Interfaccia Uniforme: Hypermedia come motore dell'applicazione**
  Le azioni sulle risorse sono guidate da link, presenti nella rappresentazione delle risorse stesse
- Esempio: nella rappresentazione del prodotto,
  - Un link rappresenta l'azione per avere maggiori dettagli
  - un link rappresenta l'azione per acquistarlo

Esempio: XML

```
<album>
 <title>the title</title>
 <code>1234</code>
 <description>A new piece of art</description>
 <link rel="/artist" href="/artist/blackMen"/>
 <link rel="/purchase" href="/purchase/1234"/>
</album>
```

Gli Elementi link

- Il documento XML descrive una risorsa di tipo «album»
- I due elementi «link» descrivono due lik associati con la risorsa, cioè due azioni:
  - Il primo consente di ottenere la descrizione dell'artista
  - Il secondo consente di effettuare l'acquisto dell'album
- Si noti l'attributo «rel», che indica per quale motivo il link è associato al documento, cioè l'azione possibile

Ecco il Significato di ReST

- Ecco perché ReST, cioè Representational State Transfer
- Perché il server trasferisce al client la rappresentazione dello stato della risorsa, con associati i link che descrivono tutte le azioni possibili che si possono effettuare sulla risorsa stessa
- L’uniformità è data dal fatto che si usano gli URI

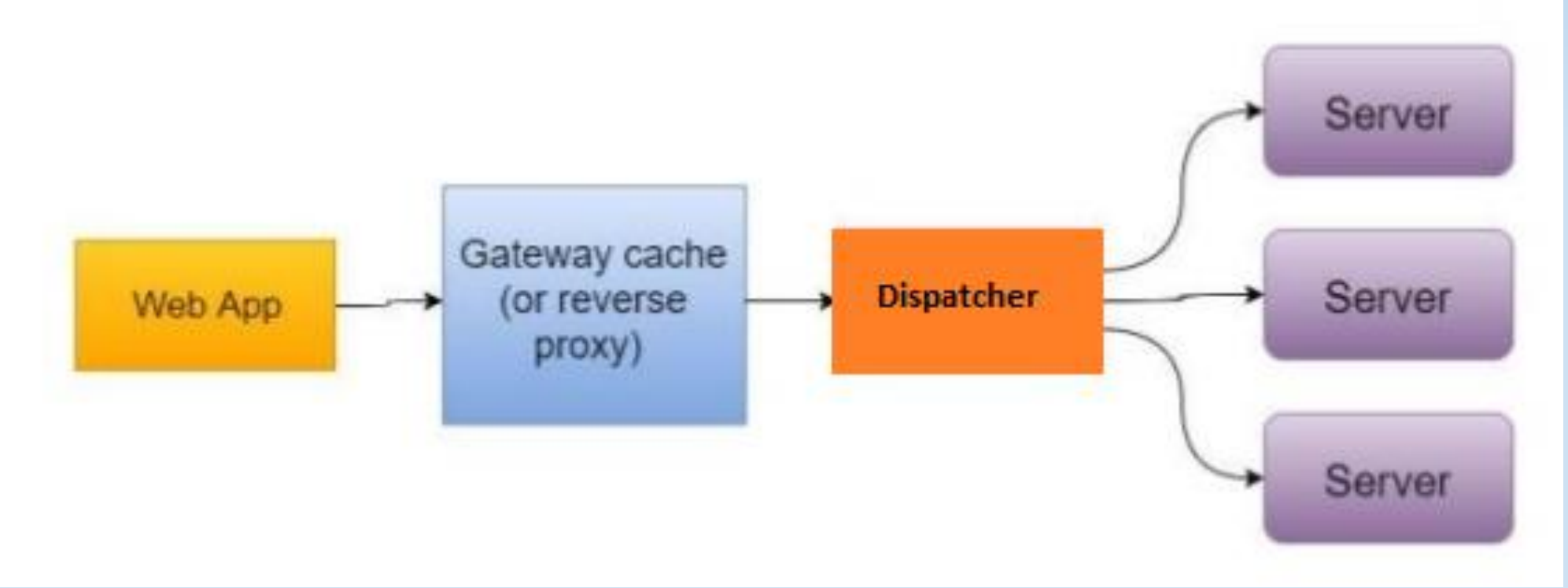
ReST e Micro-Services

- L’approccio ReST è l’ideale nei sistemi basati su micro-services
- Perché un micro-servizio fornisce la rappresentazione di una risorsa con le azioni possibili su di essa e i link ai micro-services che le eseguono

ReST e Micro-Services

- Un micro-servizio è associato ad un tipo di risorsa che gestisce, sarà il **dispatcher** delle richieste web a indirizzare la richiesta verso il servizio e il server appropriati
- In questo modo, si ottiene un elevatissimo grado di componibilità dei servizi

ReST e Micro-Services



XML o JSON?

- XML fornisce di suo il concetto di «link»
- Nel senso che è universalmente accettato un elemento «link» con le caratteristiche viste prima
- E JSON?
- Si può fare riferimento alla proposta *Hypertext Application Language (HAL)*

Esempio: JSON

```
{ "type": "album",
   "title": "the title",
  "code": "1234",
  "description": "A new piece of art",
  "_links": { "artist": "/artist/blackMen"/",
                    "purchase": "/purchase/1234" }
}
```

Esempio: JSON

- Il campo "_links" contiene tutti i link associati all’album, dove il nome del campo descrive il tipo di azione possibile

Conclusioni

Possiamo trarre alcune conslusioni riassuntive

- I micro-servizi non esisterebbero senza ReST
  Infatti, ReST è stato ideato nel 2000, quindi fornisce la base concettuale per evolvere le SOA verso i micro-services

Conclusioni

- I micro-services non sono una tecnologia, ma un approccio allo sviluppo
- Richiedono un salto culturale all'interno delle organizzazioni
- Richiedono all'organizzazione di strutturarsi in modo molto diverso, cioè tanti piccoli gruppi con la responsabilità di pochi servizi
- Con una guida autorevole

Riferimenti

- Newman, Sam. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.
- Microservices **https://martinfowler.com/articles/microservices.html**
- Introduzione a ReST **https://italiancoders.it/introduzione-a-rest/**