UNIVERSITY OF TECHNOLOGY SYDNEY

MICROSERVICES ARCHITECTURE

Dr Tom McBride

UTS: ENGINEERING AND INFORMATION TECHNOLOGY

Microservices architectures

- Monolithic architectures encounter problems of scaling and upgrading as they become large.
- One response has been to shift to a microservices architecture
- A software architecture style in which complex applications are composed of small, independent processes communicating via language-agnostic APIs.

A short history of microservices architectures

- Term “Micro-web-services” used by Dr Peter Rodgers in 2005
- Functionality provided by a collection of independent services
- “Microservice” was discussed at a workshop of software architects in May 2011.
  - A common architecture style many of them had recently been exploring.
- Described as “fine grained SOA” or “SOA” done right”

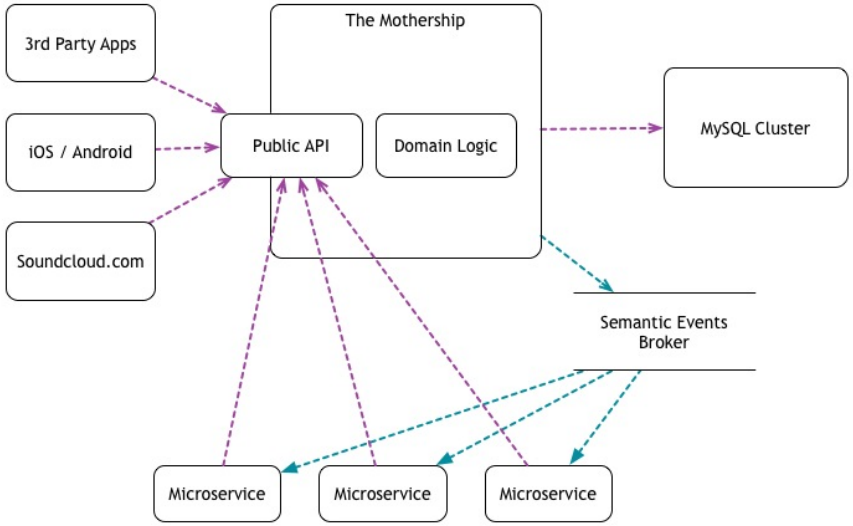
There comes a time

- As applications grow the connections and interdependencies between their parts tends to increase
- Eventually this becomes incomprehensible, unmaintainable and unmanageable
- The situation could be improved by strict encapsulation, loose coupling and strict layering
- Scaling and performance concerns favour an architecture in which separate components can be deployed on independent servers.
  - Allows horizontal scaling
  - Separate components improve maintenance and management

A GENERAL DESCRIPTION OF MICROSERVICES

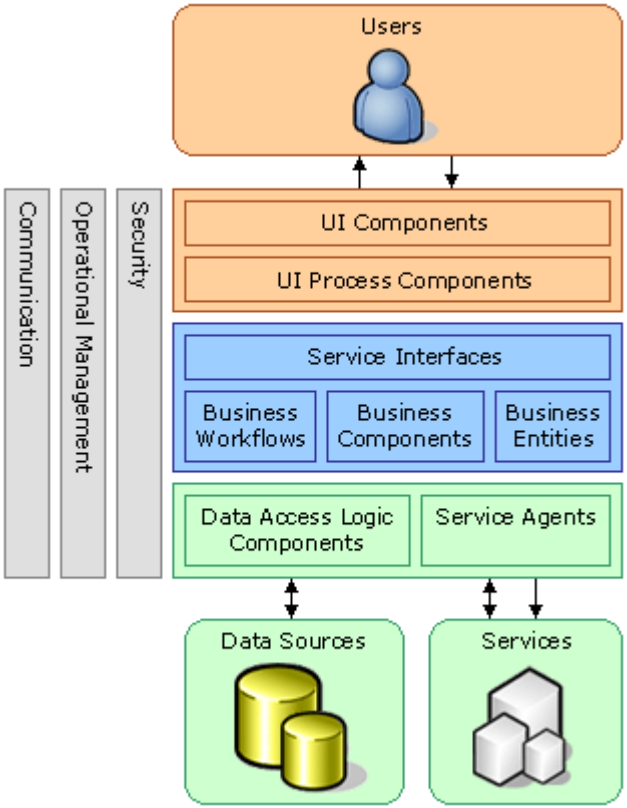
In a simple sentence

- In computing, **microservices** is a software architecture style in which complex applications are composed of small, independent processes communicating via language-agnostic APIs.
- These services are small, highly decoupled and focus on doing a small task, facilitating a modular approach to system-building.



A layered architecture

- Each service is small and independent
- Little, if any, communication between services
- Services may call on a lower layer for lower level services, data access, hardware interface and similar



Martin Fowler's description

- In short, the microservice architectural style 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.

In contrast to monolithic architecture

- Enterprise Applications are often built in three main parts: a client-side user interface (consisting of HTML pages and javascript running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application.
- It is the server side that is built as a single monolith – a single logical executable.
- All changes to the system involve rebuilding and redeploying a new version of the server-side application.

Properties of microservices architectures

- The services are easy to replace
- Services are organized around capabilities, e.g. user interface front-end, recommendation, logistics, billing, etc
- Each service should have a single responsibility.
- Each service is encapsulated.
- Tight cohesion and loose coupling matter, a lot.

UNIVERSITY OF TECHNOLOGY SYDNEY

ADVANTAGES OF MICROSERVICES ARCHITECTURE

Advantages of a microservices architecture

- Each microservice
  - Is relatively small, easier to understand.
  - Can be deployed. independently of other microservices
  - Can be scaled independently.
  - Can be deployed on hardware best suited to its resource requirements.
  - Fault isolation and rollback is easier
- Development teams can be organised around separate microservices.
- Microservices can be developed using different technology stacks.

Small services are easier to understand

- Each microservice is relatively small
- Easier for a developer to understand (and get right)
- Small code base starts faster in the IDE
- Developers tend to be more productive when the code loads, compiles and deploys faster

Independent services

- Changes can be deployed independently
- Continuous deployment becomes possible
- Same team becomes responsible for development, deployment and maintenance

Flexible and responsive scaling

- Since each service is a separate process each can be scaled independent of other services
- A microservice can be deployed on hardware that suits its resource requirements. E.g. a CPU intensive vs memory intensive vs DB intensive

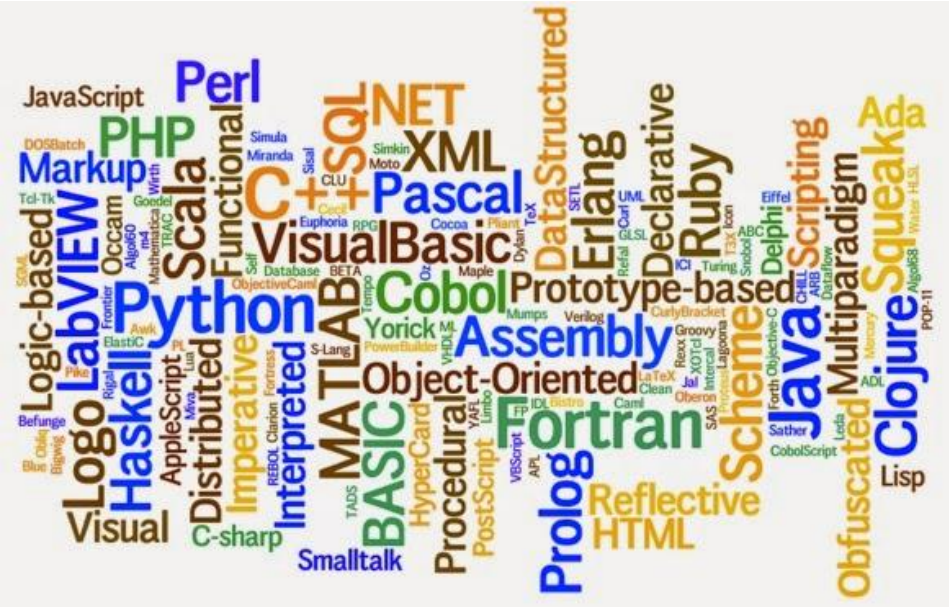
Scalable development

- Development can be organised around small independent development teams.
- Each team is solely responsible for a single microservice or collection of related services.
- Each team can develop, deploy and scale their service independently of other services.
- Responsibility migrates toward “development to disposal” (cradle to grave)

Fault isolation

- A fault in one service will not bring down the entire system
- E.g. a memory leak in a service is contained within that service.
- A faulty service can be taken down or rolled back to a previous version relatively easily.

Polyglot systems



- Microservices are not restricted to the same technology stack as all other services.
- In principle, each service can use languages and frameworks that suit the service.
- Provides an opportunity to trial new technology without “bet the business” commitment.
- Mature organizations restrict their technology choices.

UNIVERSITY OF TECHNOLOGY SYDNEY

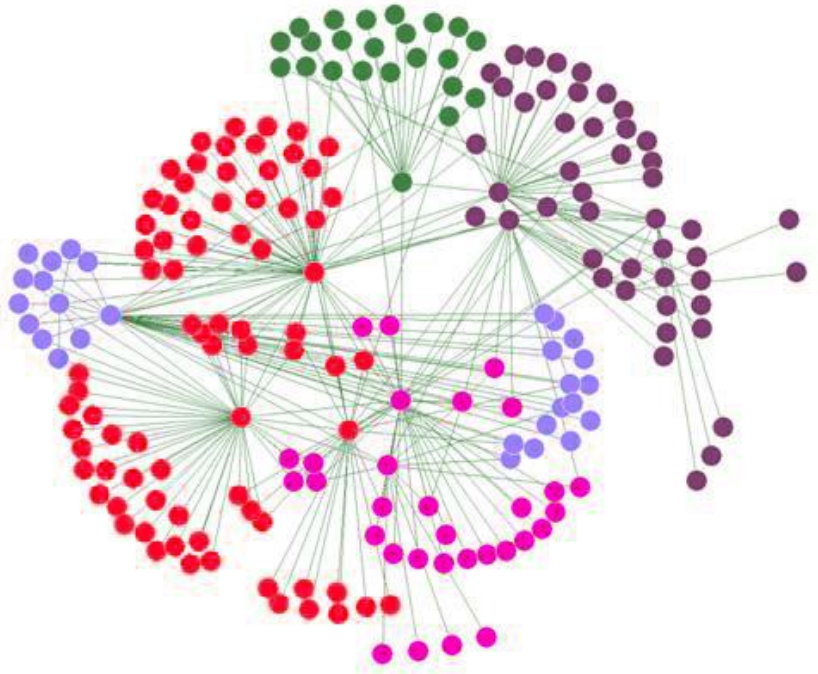
DISADVANTAGES OF MICROSERVICES ARCHITECTURES

Drawbacks and Issues

- Additional complexity of a distributed system
  - Developers must implement an inter-process communication mechanism.
  - Writing automated tests that involve multiple services is challenging.
  - It can be difficult to create consistent test environments.
  - Subtle behaviours can emerge from the interactions between services.
- Significant operational complexity
  - There are multiple instances of different types of services that must be managed in production. This requires a high level of automation.
- Eventual consistency
  - Maintaining strong consistency becomes extremely difficult, so everyone has to manage eventual consistency
- Features that span multiple services requires a rollout plan based on the dependencies between the services.

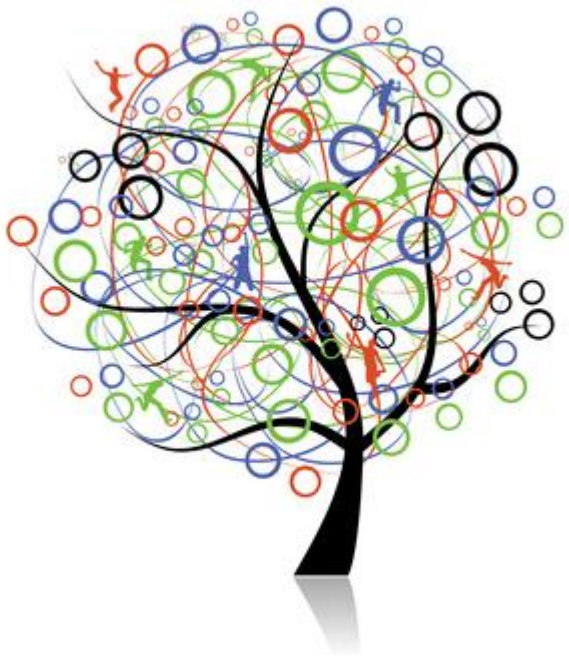
The complexities of distributed systems

- Inter-process calls are slow
  - If a function calls a number of services, and they call a number of services the response times add up to a considerable latency problem
- Microservices must be coordinated at the function level
- Asynchronous programming is hard to get right, much harder to debug
- Testing and debugging becomes equally complex and difficult.



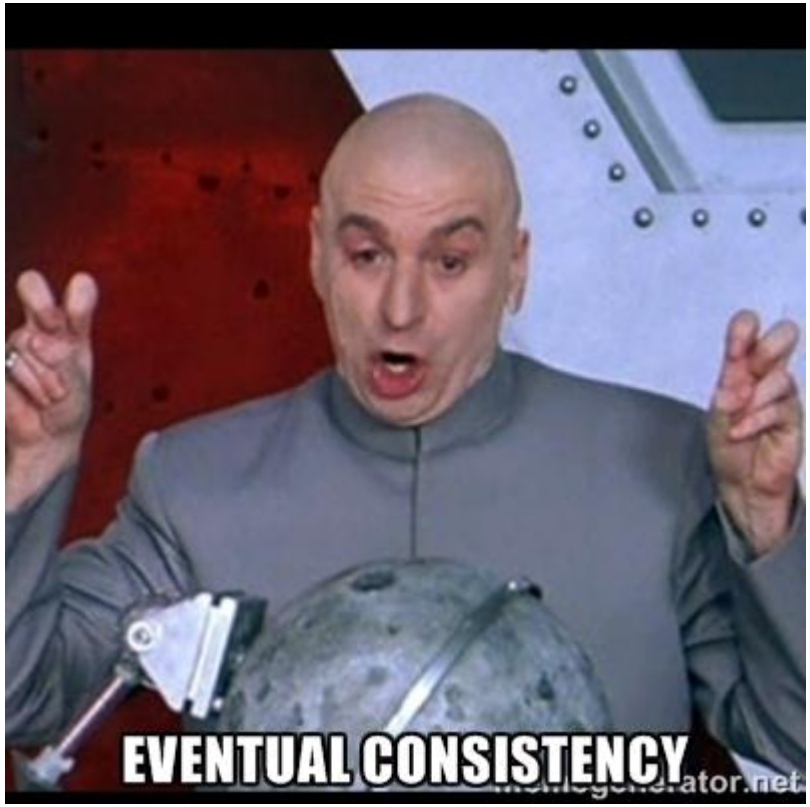
Operational complexity

- Managing and monitoring multiple services is more demanding than the equivalent monolithic system.
- Continuous delivery becomes essential.
- Automation and management tools are still immature
- Organizations tend to adopt “You build it, you support it”
  - Developers don’t like late night support calls
  - Response is to develop resilient services



Eventual consistency

- Maintaining strong consistency is very hard with distributed systems.
- Everyone must manage eventual consistency.
- Usability problem when updates are not immediately reflected.
- Debugging becomes difficult because the investigation will occur after the inconsistency window has closed.



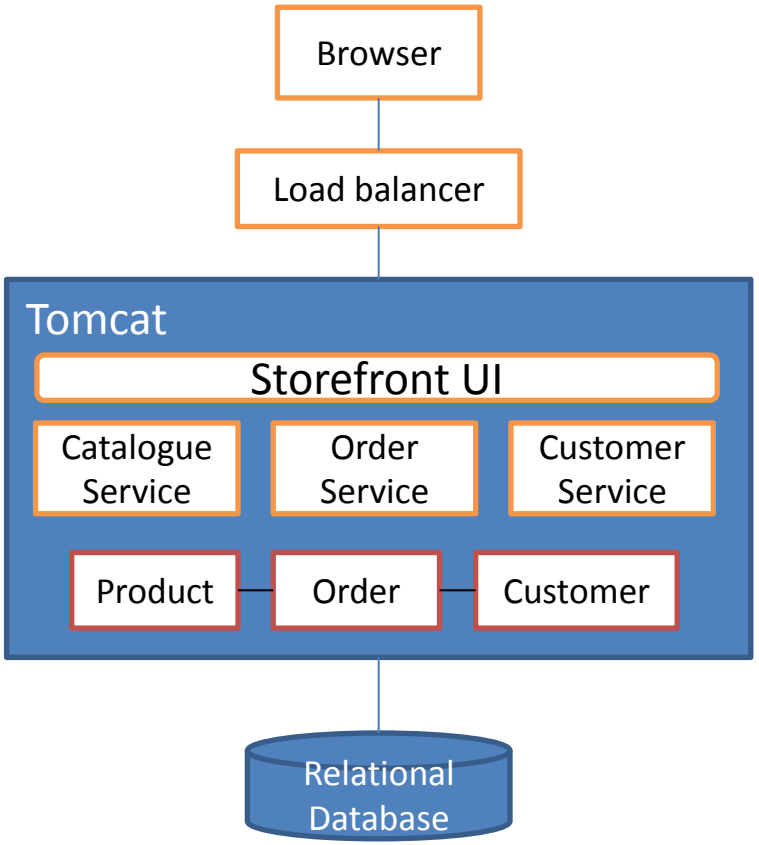
FROM MONOLITH TO MICROSERVICES

When to use this architectural style

- The first version of an application usually doesn't have the problems this microservices style solves.
- Using an elaborate distributed architecture will slow down development.
- Later, when scaling is a challenge, tangled dependencies might make it difficult to decompose the monolithic application into a set of services.
- When the development team becomes too large (~>25)
- So the question become one of timing

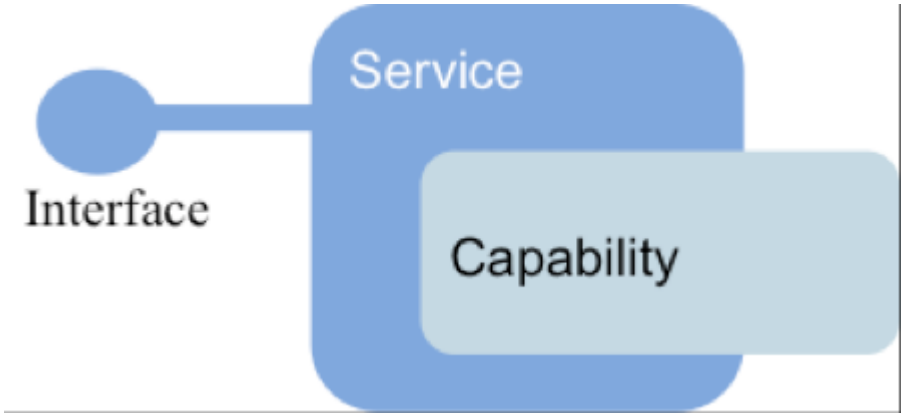
Where most start

- Imagine you are building an online store that takes orders from customers, verifies inventory and available credit, ships merchandise.
- Quite likely you would build a monolithic application like this one.
- The application is deployed as a monolith.
- Simple to develop and deploy.



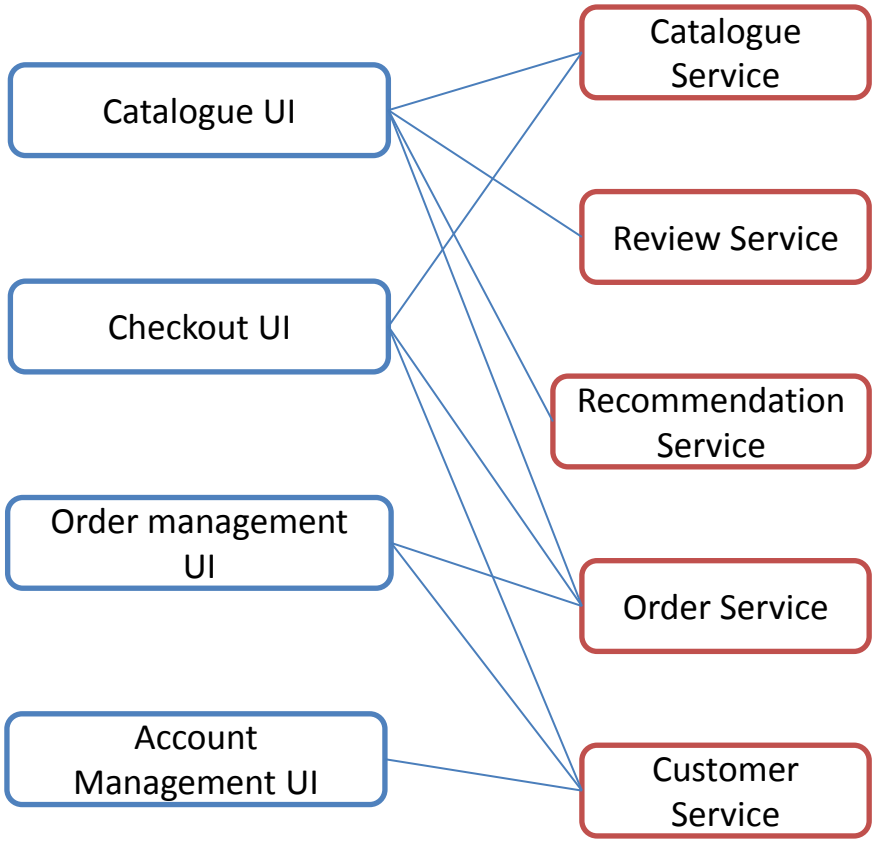
Converting from monolith to microservices

- Separate the monolithic functionality into small services so that any function is provided by a combination of these small services.
- Implement each small service as a separate process
- Develop a well-defined interface that isolates and encapsulates a module within it.
- The architecture necessarily becomes layered.

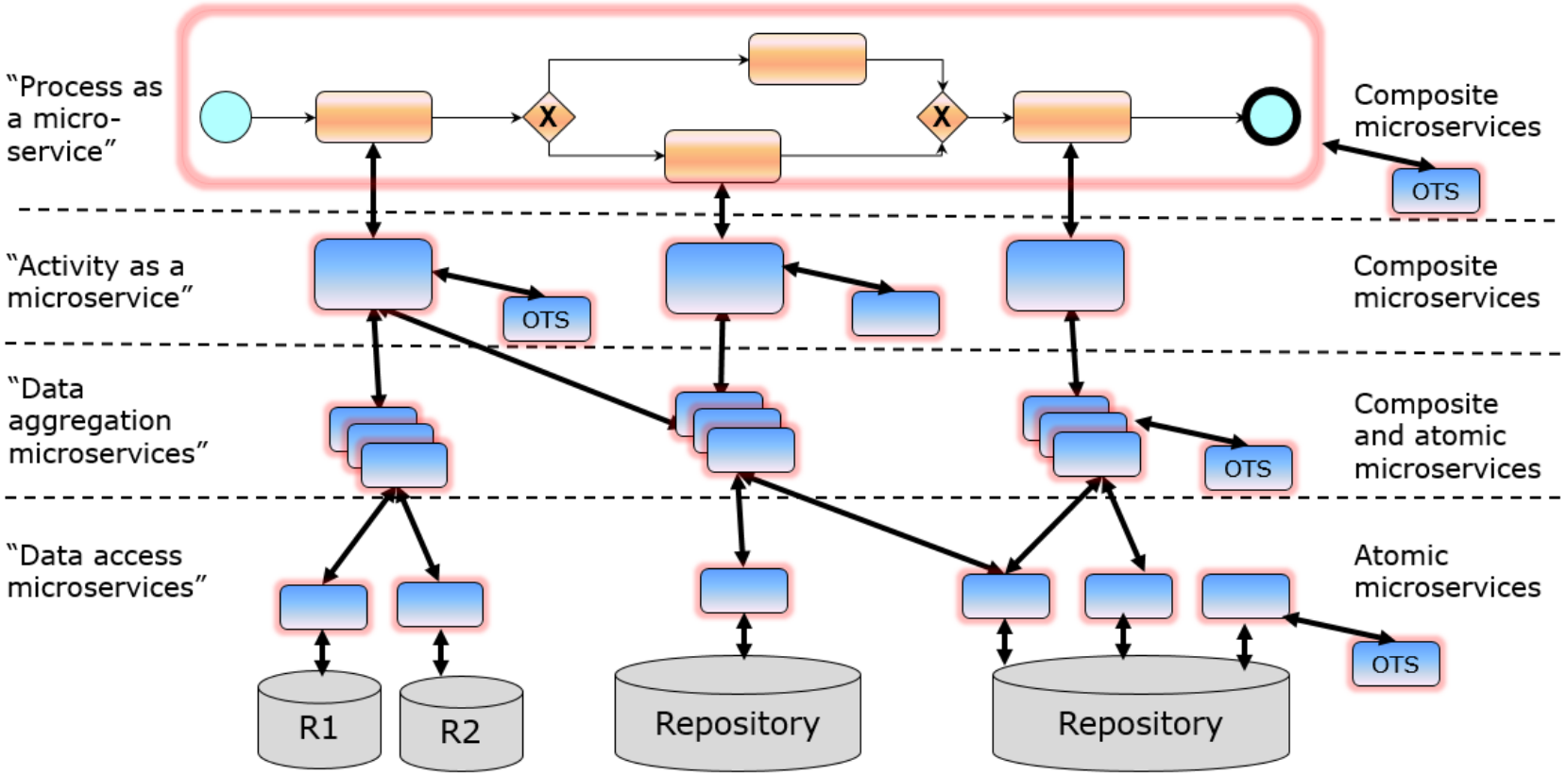


The decomposed application – macroservices to microservices

- The decomposed application consists of various front end macroservices and multiple back-end microservices
- Transaction coordination is done at the macroservice level
- Microservice failure should be handled at the macroservice level

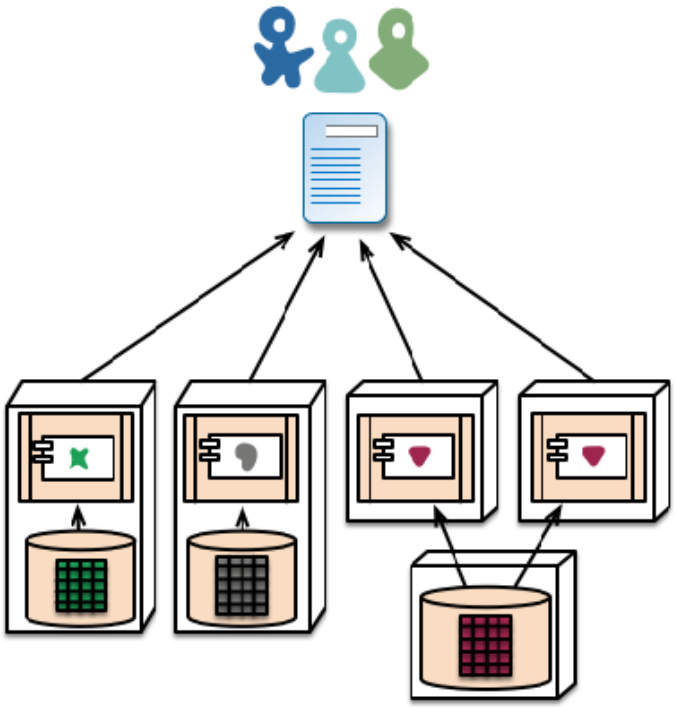


Microservices layering



Distributed data management

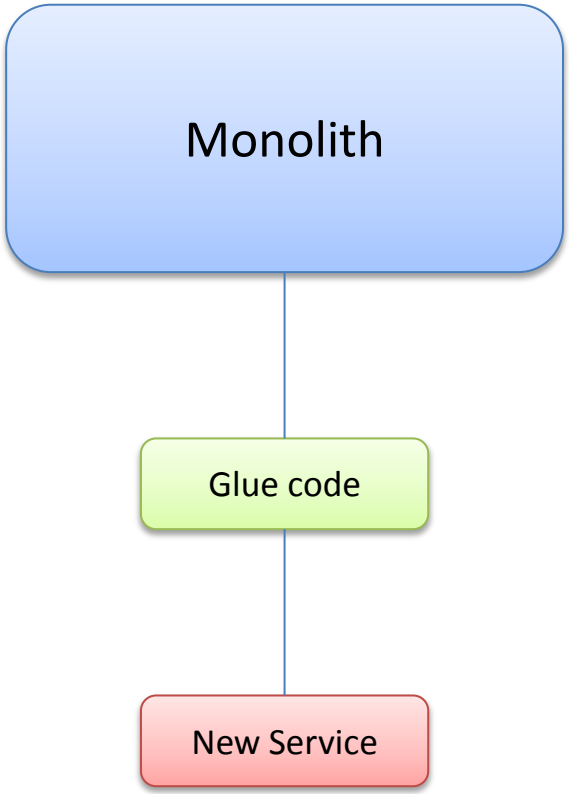
- To ensure loose coupling each microservices has its own database (schema)
- Different services might use different types of database (ACID, graph, NoSQL)
- Incurs all the problems of distributed data management



microservices - application databases

Refactoring a monolith

- Implement new functionality as a standalone service
- Requires glue code to integrate the service with the monolith.
- Often the glue code is messy and complex, but a necessary first step



UNIVERSITY OF TECHNOLOGY SYDNEY

Lessons from Google and eBay

SERVICES ARCHITECTURES AT SCALE

Before they were big

- eBay was a monolithic Perl application
- Twitter started as a Rails application
- Amazon.com started as a monolithic C++ application

Google

amazon

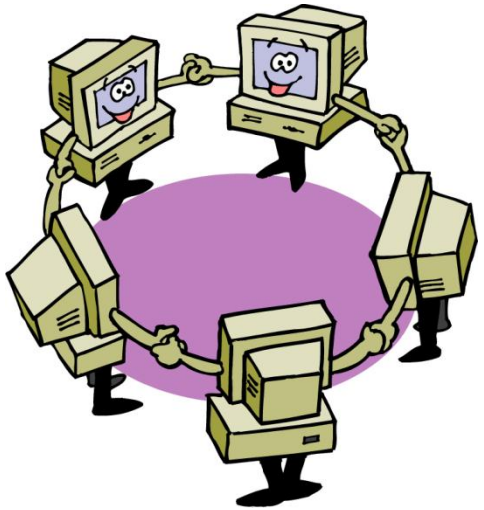
ebay

Architecture without the architect

- A central architecture board becomes a bottleneck
- Better to have that talent work on something really usable by individual teams
  - A library
  - A tool
  - A set of guidelines

Standardization without central control

- Standardized communication between IT services and infrastructure components is very important
- Network protocol
  - Google uses Stubby
  - eBay uses RESTful HTTP
- Serialization format
  - Google uses protocol buffers
  - eBay uses JSON
- Standards become standard by being better than the alternatives



Building a service as a service owner

- A well-performing service in a large-scale ecosystem has a single purpose, a simple and well-defined interface and is very modular and independent
- Service owner goals are to provide client functionality, quality software, stable performance and reliability. Over time, these should improve.
- Using DevOps approach the team owns the service from creation to disposal.
- Having to support a service encourages teams to develop resilient services.

Microservices anti-patterns

- Service that does too much
  - Becomes a miniature monolith
  - Difficult to understand and scary to change
  - Increases upstream and downstream dependencies
- Shared persistence
  - Shared persistence breaks encapsulation
  - Unwittingly re-introduces coupled services