Make a successful start with Microservices

Create fast, resilient and scalable system using Spring boot, Netflix APIs (Eureka, Hystrix) with Microservices architecture

Hemant Gaur
@hgaur_com

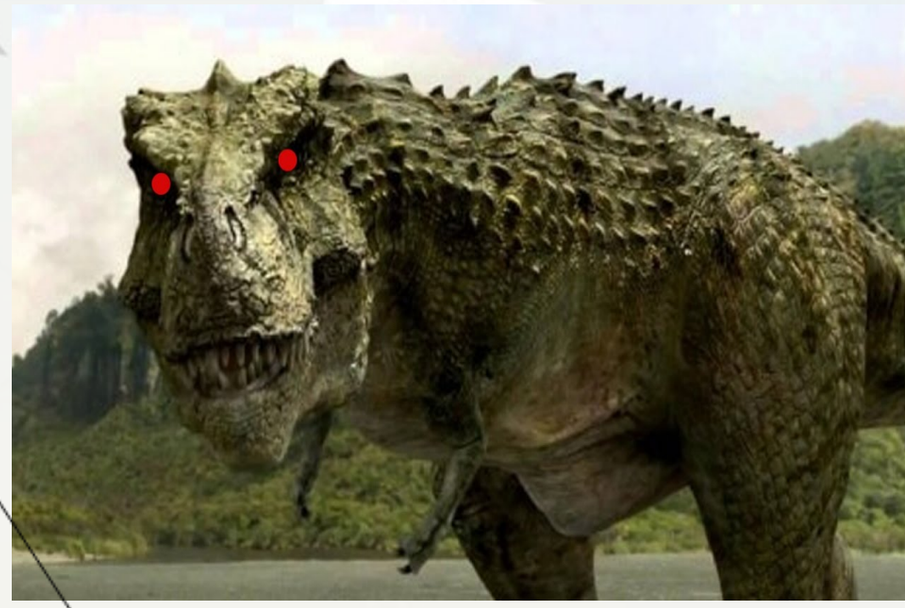
Topics

- A look back to Software Systems
- Microservices: What, Why and How?
- Requirements for Microservices
- Quick overview of the technologies involved
- Improve Team performance (Technical and Social)
- Live demo of breaking a monolithic to Microservices using spring boot and Netflix APIs
- Good, ugly of Microservices

A look back to Software Systems

Journey we travelled so far

Power-full, Big, Adamant



Load balancer

Web Layer

Service 1

Service 2

Service 3

Service 4

DAO layer

DB main

DB replication

Web Layer

Service 1

Service 2

Service 3

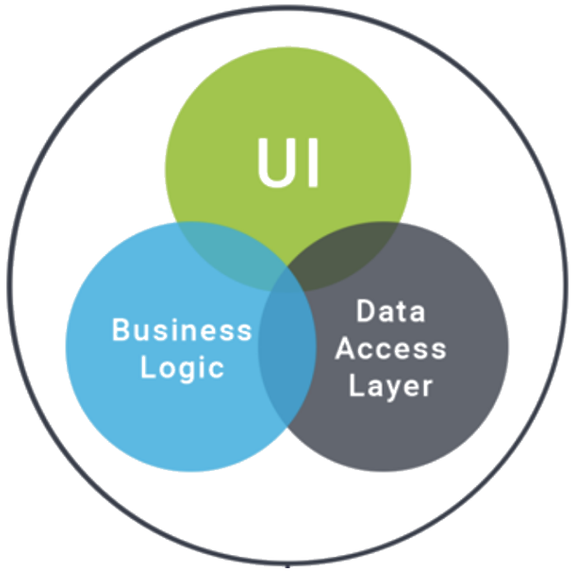
Service 4

DAO layer

The System, *a look back*

Good things

- Everything was at one place, Logs, business code, issues



- Easy to manage from an operational perspective

- Usually, communication to different modules is fast because they are part of the same physical system, no network overhead.

- Easy to refactor system modules scope, because they are part of the same system

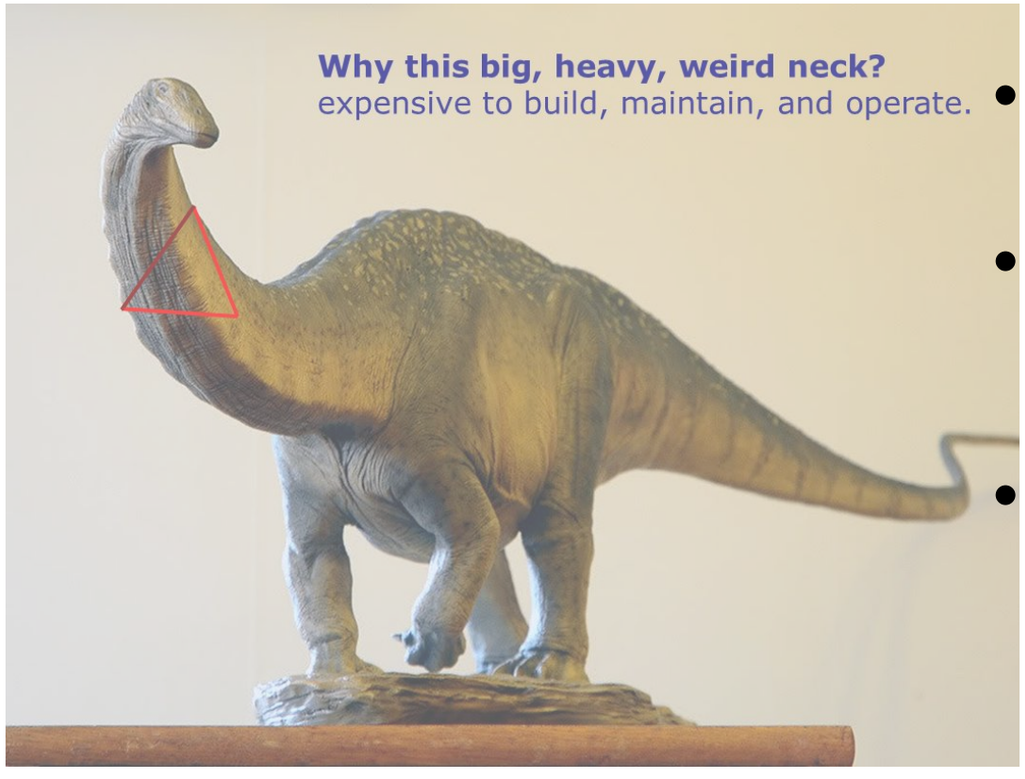
- People know the full system, and they can quickly identify issues and changes in the system

We used to enhance the same system, to provide a full range of functionalities making a complete package that was a selling point.

Not so good things

- Release manager was there :)

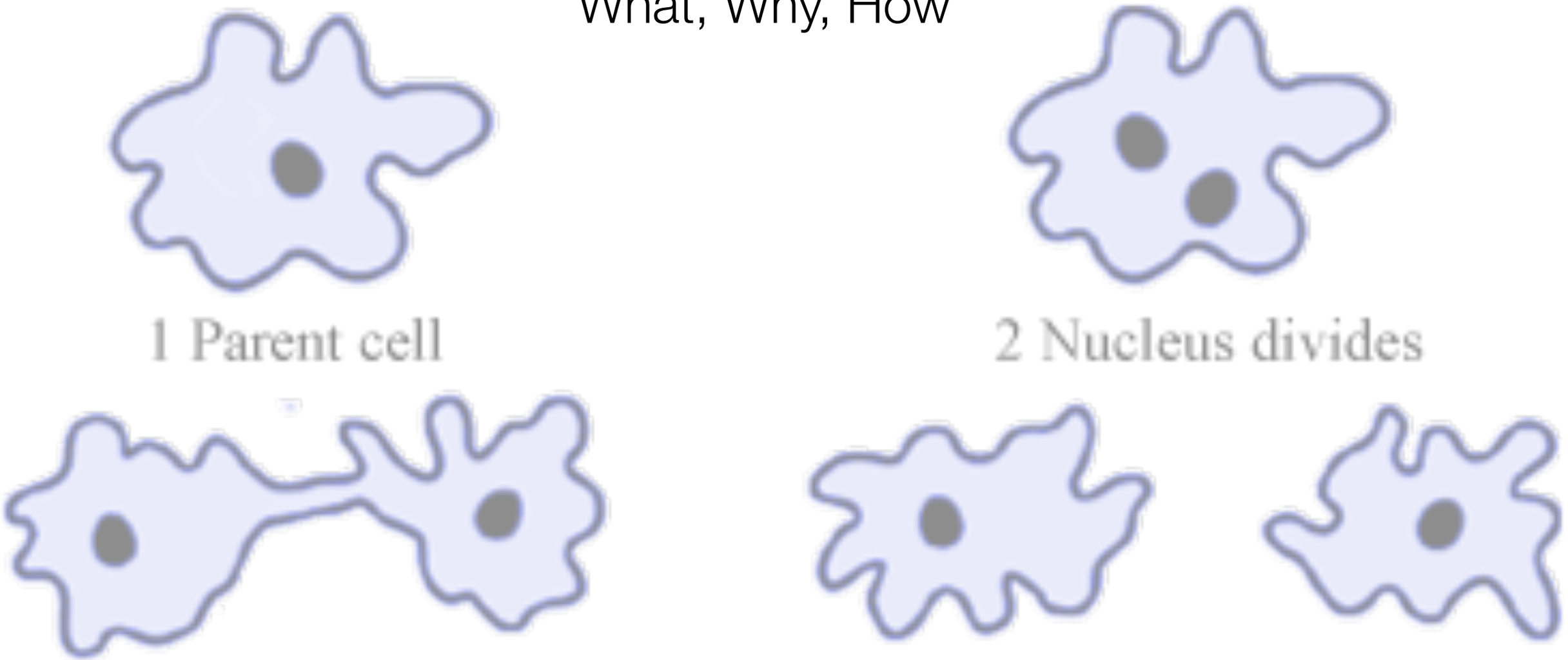
- Easy to manage but difficult to change and refactor
- If things work they work well but one goes down, it takes all down



- Ramp up is very difficult may take 1 to 3 months
- Possible to mess the module scope and intermix the responsibilities
- The growth was like a dinosaur, massive and difficult to move. Usually, some components are significant and we don’t know why.

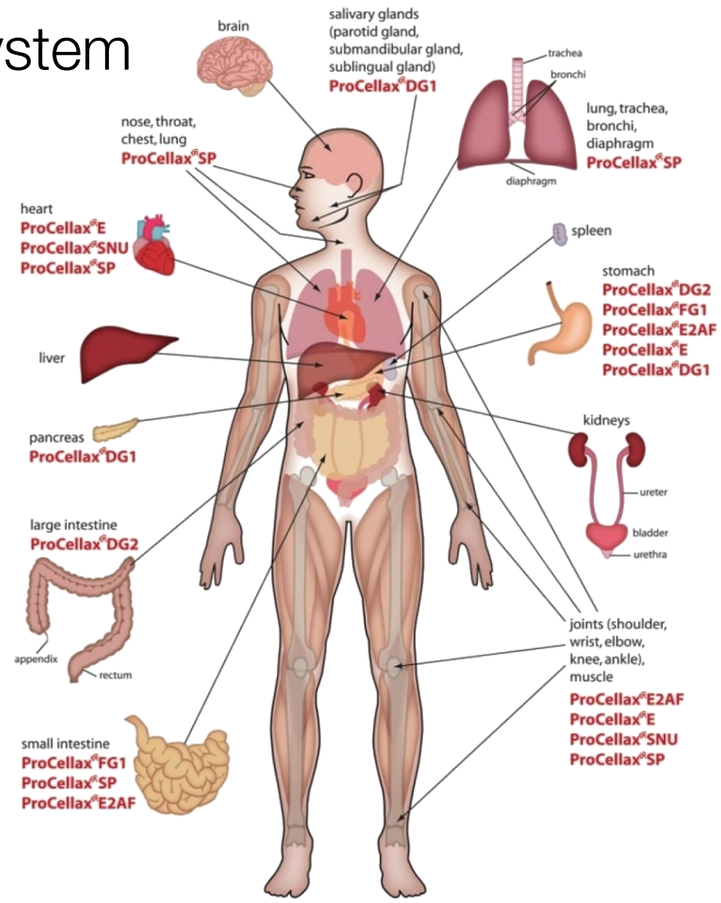
Microservices

What, Why, How



What

- Typical service should be as big as one developer can understand and work independently
    - Easy smaller components make a big system



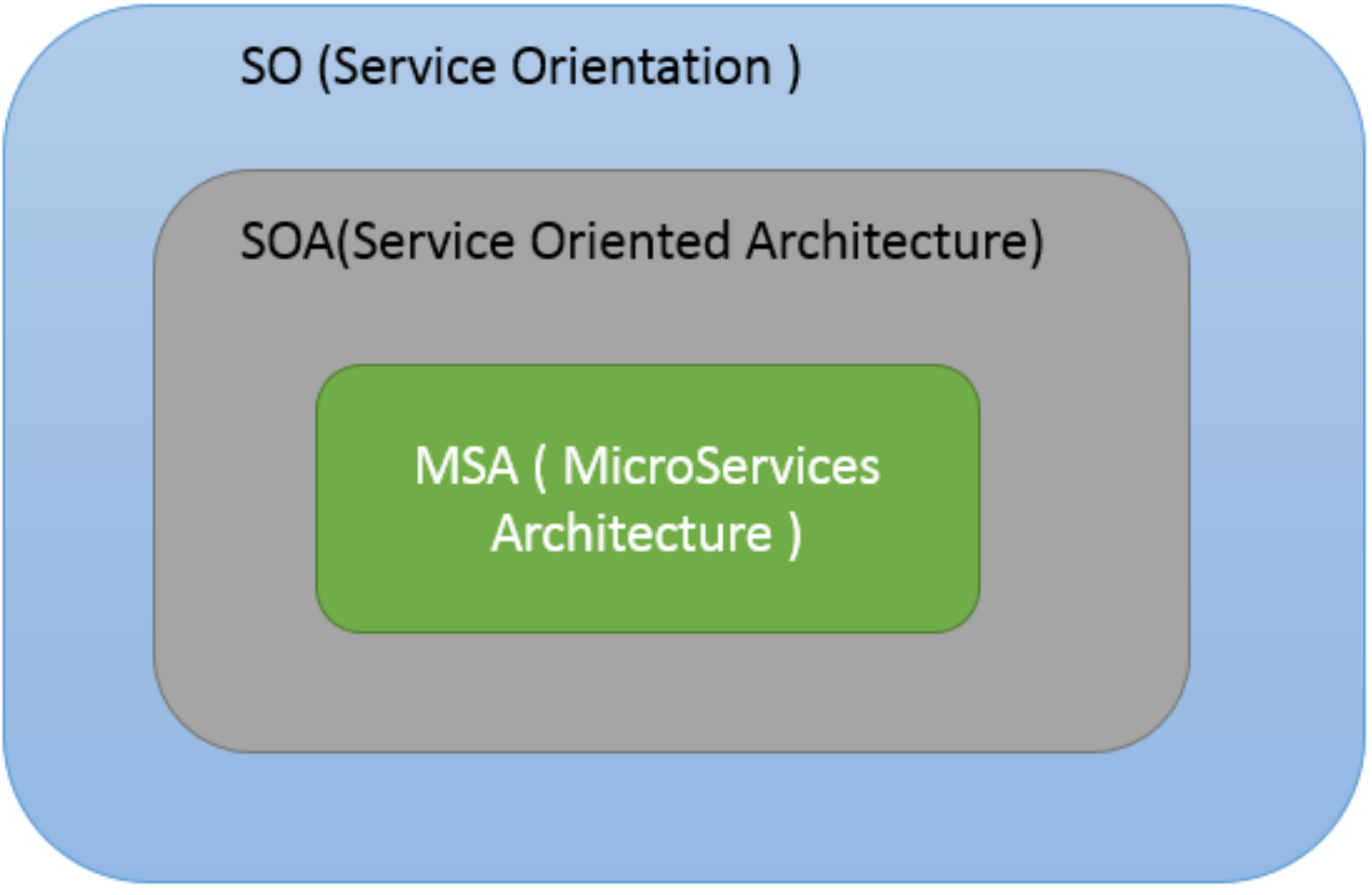
- Independent identity and business domain of Each service

- Each service should be the golden source for data of business domain; only that service must provide that data, No direct DB contact for other domain data

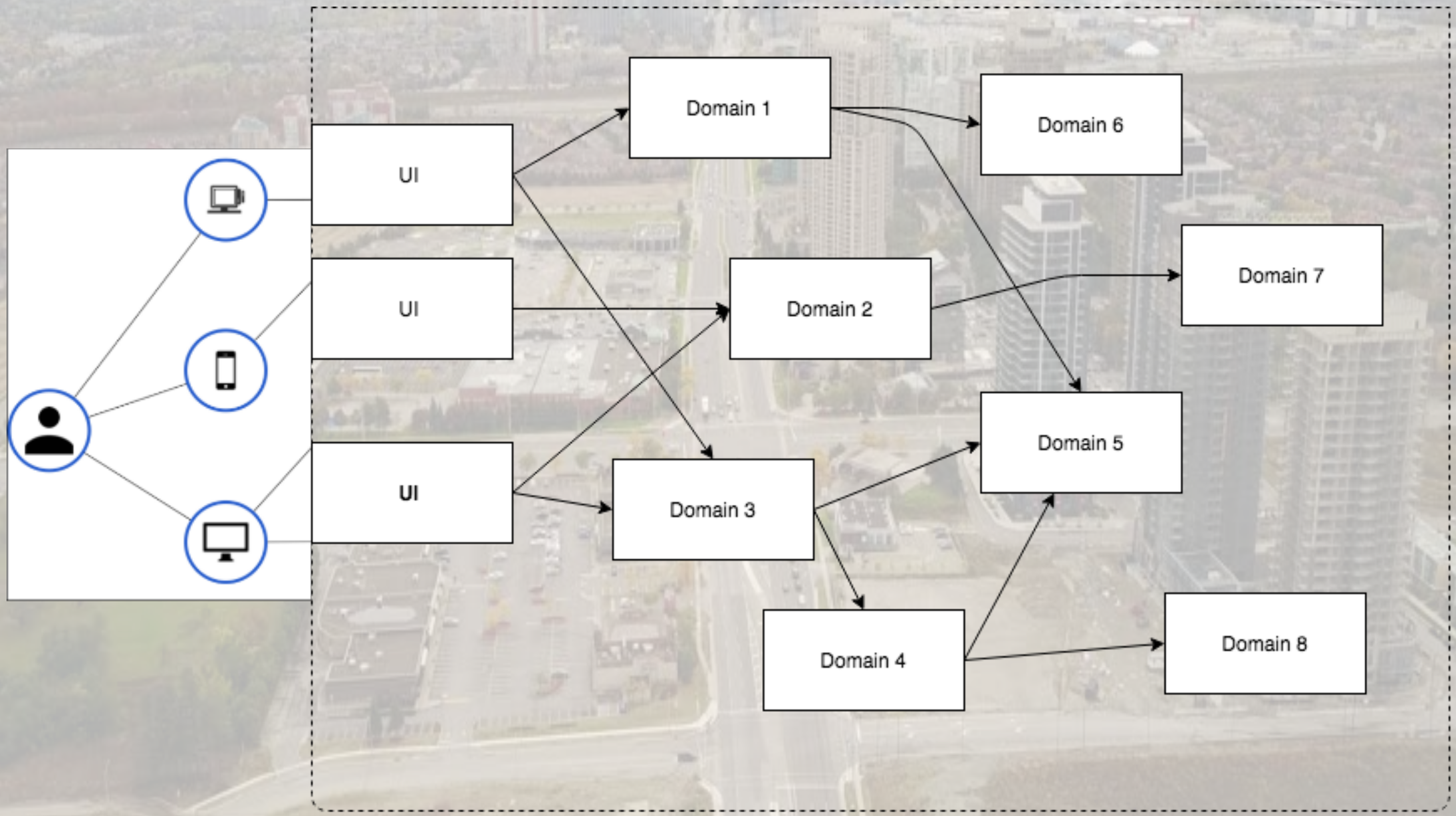
- Independent release cycle and deployment

Scope of one service is fixed

- Team size varies from 1 service - 1 developer to 1 developer - 10 services and 5 developers - 1 service


SO (Service Orientation)
SOA(Service Oriented Architecture)
MSA (MicroServices Architecture)

Build like a City



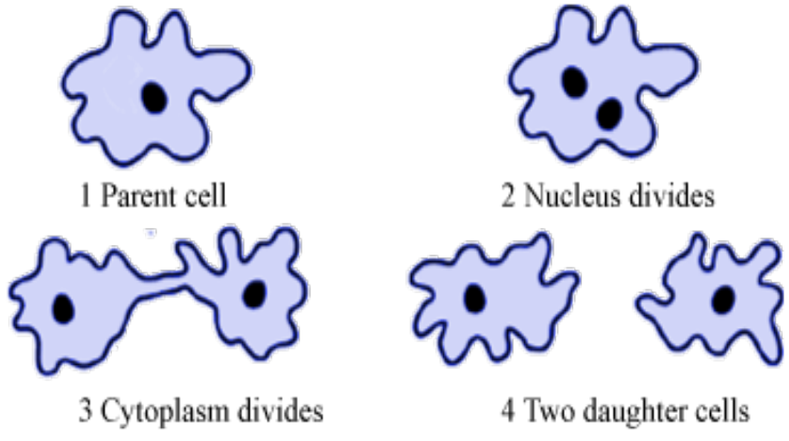
Why

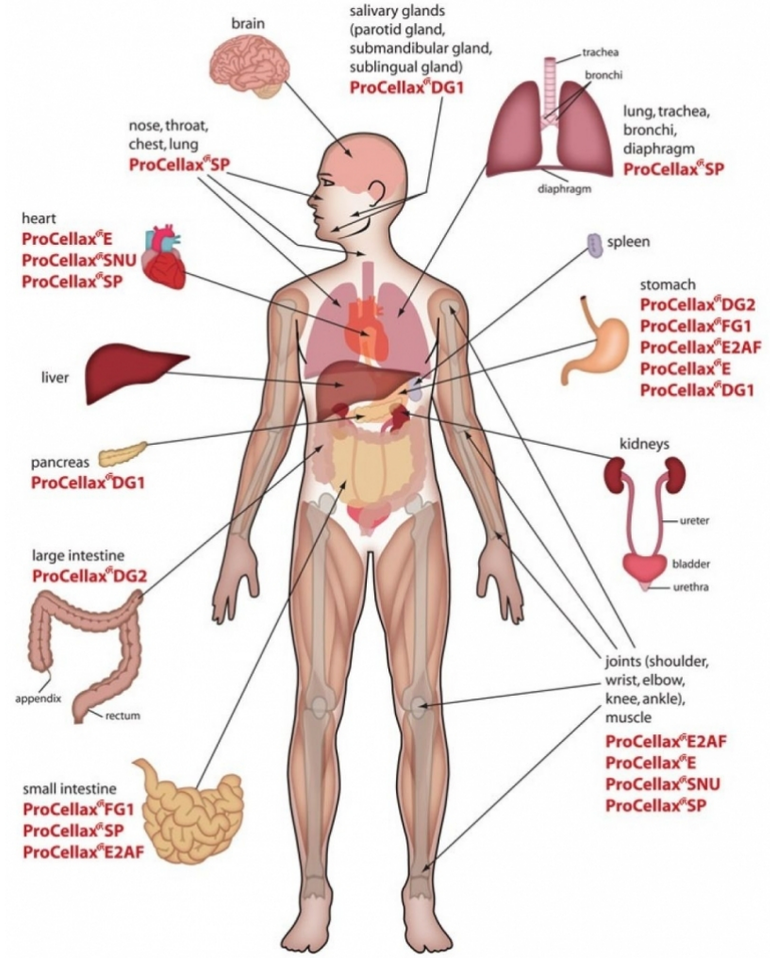
- Growth like an Amoeba

- Smaller the better but one service should consist entirely independent module to work

- Agreeing on common things and honoring the commitment is required.

- Database access is a single point of failure





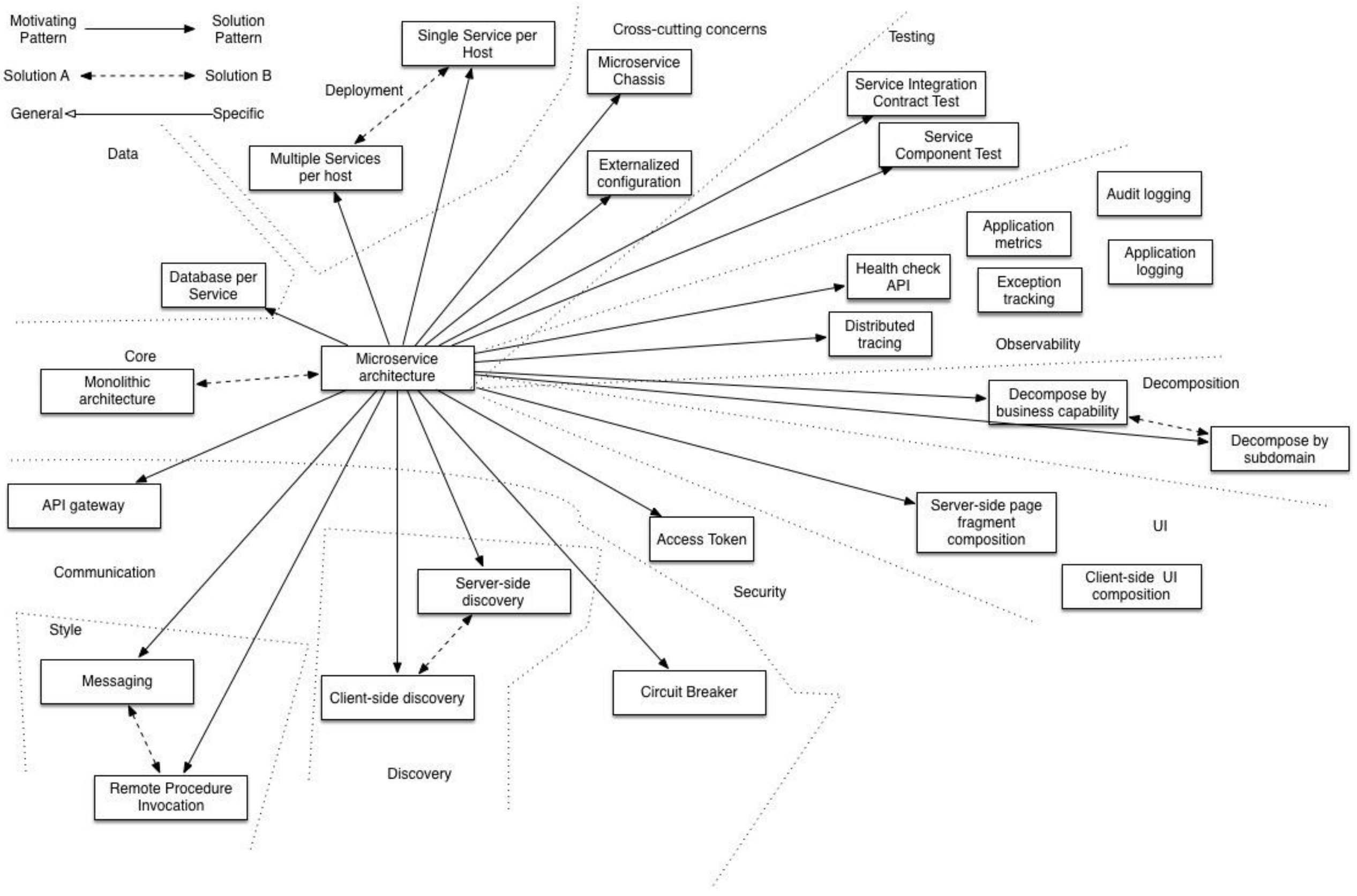
How ?

1 Parent cell

2 Nucleus divides

3 Cytoplasm divides

4 Two daughter cells



Source- http://microservices.io

Requirements for Microservices-1

- Framework to easily create Microservices - **Spring boot, Dropwizard, Vertx, Java EE**
- Universal services for authentication to call different services: **OAuth, google token, JWT, SSO, Spring security**
- Way to access logging of different systems and a single traceId or request Id to map in all systems - **Splunk**, or **ELK stack**: Elasticsearch for search, Logstash for data collection, and Kibana for data visualization
- Metrics system to track each service health- **Graphite, Atlas, Prometheus,** Spring Boot **Actuator, Geneos**
- Monitoring systems or Dashboard to monitor and raise alert in case of issues: **Icinga, pager duty, Geneos, Graphana**
- Automated build - **Jenkins, Bamboo, Team city, Cruise control**
- Deployment - **Taxis, UDeploy, Cloud foundry, Docker container, Fuse fabric**

Requirements for Microservices- 2

- Ability to provision on-demand physical computing power.
  - Cloud environment - **AWS, Azure, Google Cloud, Openshift, in-house cloud, Cloud-foundry**
- Service Discovery and Service registration - **Eureka, Camel**
- Call to different service - **Hystrix command, Camel routes and processor**
- Automated test cases - **Selenium, Gherkins, Cucumber**
- Unit testing and integration testing - **J-units, gherkins, Cucumber, Mockito, RMock, Spring test suite, Github rest test.**

Code

https://github.com/irudra/session

Improve team social Behavior

- Less blame game
- More sense of responsibility in their tech stack

- Team focus on solving the problem
- Trust each other more
- More satisfaction in team as they have more chances to learn and experiment

Too Good, Good and Bad

1 Parent cell

2 Nucleus divides

3 Cytoplasm divides

4 Two daughter cells

Too Good

- Independent Deployment
- Easy Introduction of new Technology
- Abilities to Innovate
- Can deploy experimental applications with partial traffic
- Need not to forecast early, but we can enhance, reduce, shift the capacity of the system as you find new information in real time

Good

- Different Module with clear responsibilities
- Separation of business domain data and ability to move database without taking permission from another system
- Ability to refactor with very little or no impact on another component
- Low performance and bottleneck application are easy to find, change or remove.

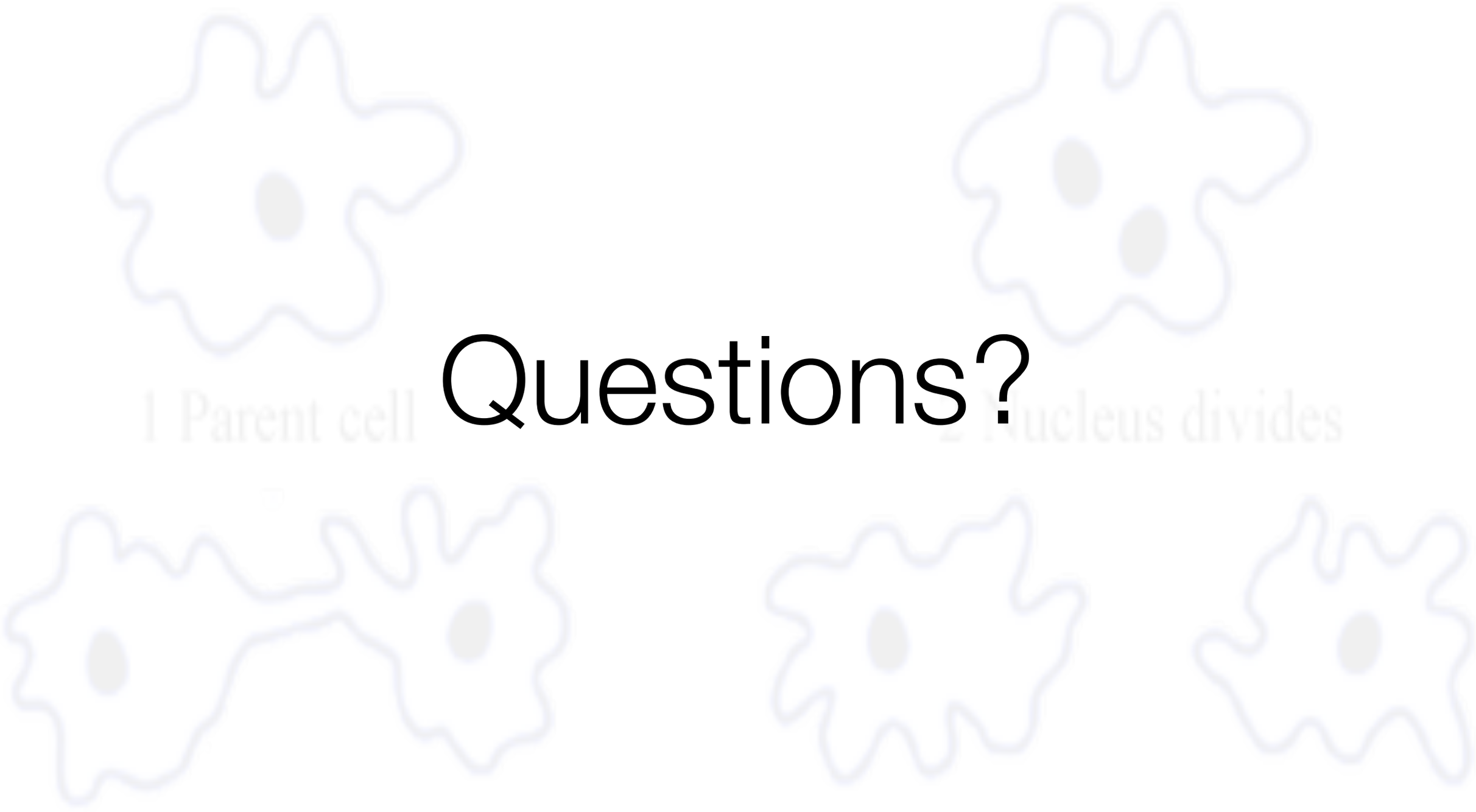
Ugly

- Difficult to manage hundreds of services
- Different independent team might prefer their agenda
- Nanoservices- Too fine grain service might reduce the resiliency
- Too many technologies might make it difficult to move or find resources
- System can partially stop working and corrupt system if not taken care properly

References

- microservices.io
- Images are taken from google search
- Inspiration from various goto Conference
- Own experience of working as a Consultant in various industry leader organizations

Questions?


1 Parent cell
2 Nucleus divides
3 Cytoplasm divides
4 Two daughter cells