CSCI 5409 Adv. Topics in Cloud Computing – Fall, 2023
Week 13 – Lecture 2 (Dec 1, 2023)

Microservices (2)

Dr. Lu Yang
Faculty of Computer Science
Dalhousie University
luyang@dal.ca

Housekeeping and Feedback

- Start recording
- Please finish SLEQ

Anti-Patterns (What not to do!)

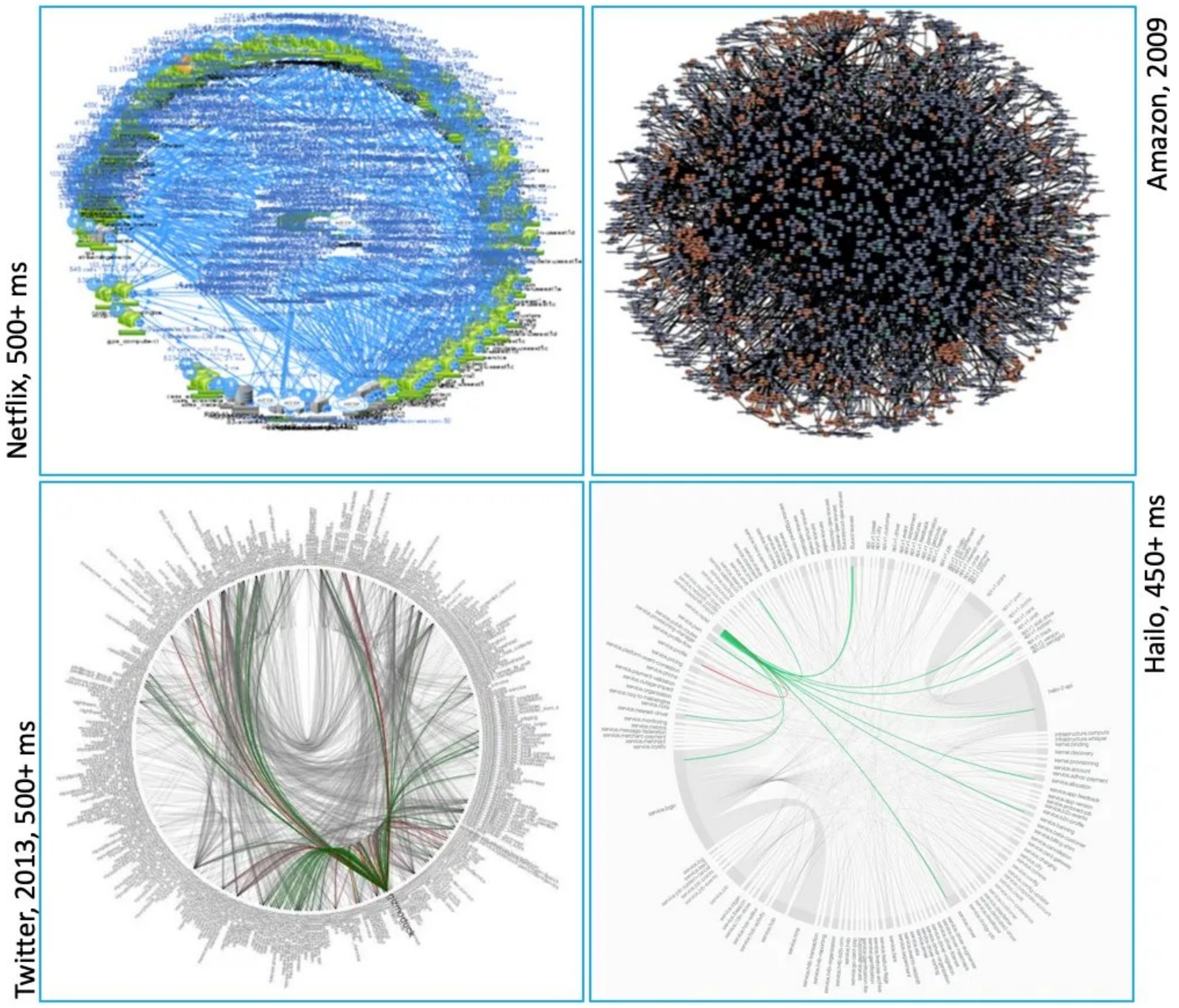
1. **The first rule of microservices is, don't build microservices.** Stated more accurately, don't start with microservices. Microservices are a way to manage complexity once applications have gotten too large and unwieldly to be updated and maintained easily. Until you feel that pain, you don't even really have a monolith that needs refactoring.
2. **Don't do microservices without DevOps or cloud services.** Building out microservices means building out distributed systems, and distributed systems are hard (and they are especially hard if you make choices that make it even harder). Attempting to do microservices without either a) proper deployment and monitoring automation or b) managed cloud services to support your distributed, heterogenous infrastructure, is asking for a lot of unnecessary trouble.

Anti-Patterns (What not to do!)

3. **Don't make too many microservices by making them too small.** If you go too far with the "micro" in microservices, you could easily find yourself with overhead and complexity that outweighs the overall gains of a microservice architecture. It's better to lean toward larger services and then only break them apart when they start to develop characteristics that microservices solve for—namely that it's becoming hard and slow to deploy changes, a common data model is becoming overly complex, or that different parts of the service have different load/scale requirements.

4. **Don't try to be Netflix.** Netflix was one of the early pioneers of microservices architecture when building and managing an application that accounted for one-third of all Internet traffic—a kind of perfect storm that required them to build lots of custom code and services that are unnecessary for the average application. You're much better off starting with a pace you can handle, avoiding complexity, and using as many off-the-shelf tools as possible.

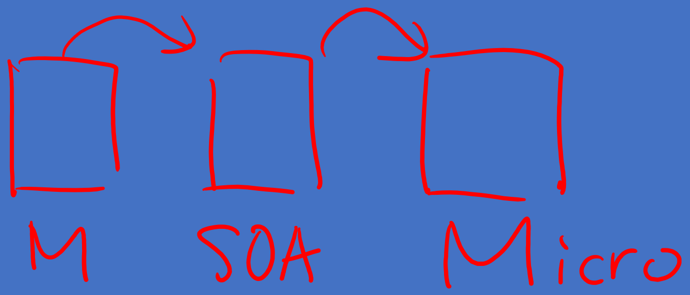
The “Deathstar” Anti-Pattern



Deathstars (a.k.a. The Knot)

Monolith

- Evolves after organizations transition to microservices.
- Takes time; it's not an initial problem but will eventually become one.
- Characterized by any service being allowed to call any other service.
- Revealed in testing when a service passes initial testing but fails during integration as there are too many different components with which it interacts – integration testing becomes cumbersome and time consuming.
- To avoid it you need:
  - Monitoring and observability, not just of services, but of their interactions.
  - Mediation between services including:
    - Traffic management and routing.
    - An "orchestration" or "choreography" strategy – a plan for how services will interact.
    - An overall security policy.



Microservices and SOA

Don't turn microservices into SOA. Microservices and service-oriented architecture (SOA) are often confused, given that at their most basic level, they both build reusable components that can be consumed by other applications.

The difference between microservices and SOA is that microservices projects typically involve refactoring an application so it's easier to manage, whereas SOA is concerned with changing the way IT services work enterprise-wide.

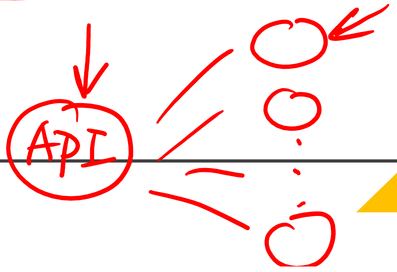
- **Microservices are used to create an application.**
- **SOA combines applications to form a complex system.**

Authentication

Monolithic vs. Microservices Authentication

- A monolithic app consists of a single indivisible unit. It typically consists of a client-side user interface, a server-side app, and a database, all tightly integrated to deliver all functions in one unit. It has all the resources it needs, so there is no need for authentication within a monolithic application. Authentication only needs to be handled when users need to access the app.
- By contrast, a microservices application has multiple independent components integrated via APIs. Whenever a microservice communicates with other microservices, you must make sure it is authenticated. Authentication ensures that only legitimate services and users have access to each microservice. In addition, like in a monolithic app, there is a need to authenticate end-users.
- When implemented correctly, authentication and authorization are essential parts of a microservices app. They serve as an additional security checks for all accessed resources, preventing security gaps and blind spots.

SSO



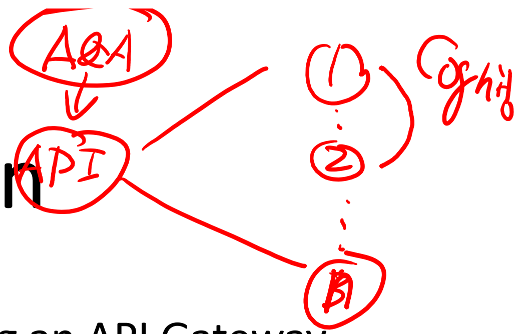
Microservices Authentication Challenges

In a microservices architecture, each microservice implements a specific function or part of the business logic. Each microservice access request must be authenticated and approved, which creates several challenges:

- **Central dependency**—authentication and authorization logic must be handled separately by each microservice. You could use the same code in all microservices, but this requires that all microservices support a specific language or framework.
- **Violating the single responsibility principle**—microservices are supposed to fulfill only one function. If you add global authentication and authorization logic to microservices, they now perform an additional function, making them less reliable and more difficult to manage.
- **Complexity**—authentication and authorization in microservices can lead to very complex scenarios. Consider that there might be users, microservices, and third-party systems accessing every microservice. This complexity can make implementation and maintenance difficult.

AWS
Cognito
IAM STS

Solution 1: Edge-Level Authorisation

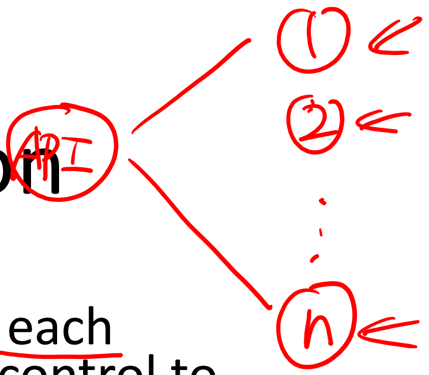


In a simple scenario, authorization only occurs at the edge, typically using an API Gateway. You can use an API Gateway to centralize authentication and authorization for all downstream microservices. The gateway enforces authentication and access control for each microservice. Use mitigation controls such as mutual authentication between microservices to prevent external connections to internal services.

This strategy has the following disadvantages:

- Less secure—if an attacker gets past the gateway, they can freely access any microservice. An API Gateway as a single access point violates the "defense in depth" principle.
- More difficult to manage—if the system is complex with many roles and access control rules, pushing all authorization decisions to an API gateway can become unmanageable.
- Limited access to development teams—typically, operations and maintenance teams set up API gateways, so the development team cannot change permissions directly. This disconnect can lead to communication and process overhead.

Solution 2: Service-Level Authorization



This strategy enables direct authentication and authorization for each microservice. The advantage is that each microservice has more control to enforce its access control policies. A service-level authorization architecture includes the following:

- Policy Administration Point—lets administrators create, manage, and test access rules.
- Policy Decision Point—checks which access control policy applies to the current request and evaluates whether to grant or deny the request.
- Policy Enforcement Point—provides access decisions, enforcing the access policy for specific requests.
- Policy Information Point—allows elements in the system to retrieve data about policies or receive account attributes to make policy decisions.

Solution 3: External Entity Identity Propagation

AWS STS

This strategy can make authorization decisions while taking into account user context. For example, it can change the authorization decision based on user ID, user roles or groups, user location, time, or other parameters.

To perform authentication based on entity context, you must receive information about the end-user and propagate it to downstream microservices. A simple way to achieve this is to take an Access Token received at the edge and transfer it to individual microservices. This strategy provides the most granular control over microservice authentication. However, it has two main drawbacks:

- Not secure—the content of the token is shared with all microservices, and as a result, attackers can compromise it. A possible solution is to sign tokens via a trusted issuer.
- It may require internal microservices to support multiple authentication techniques, such as JWT, OIDC, or cookies.

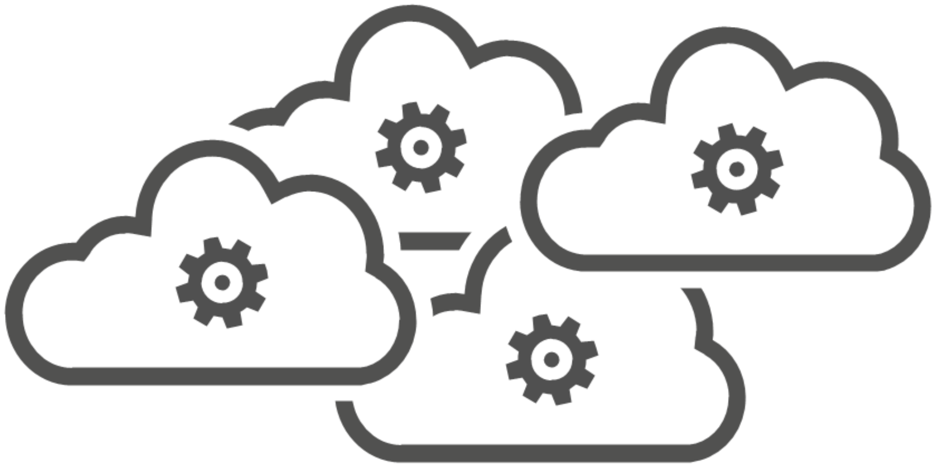
Cross-cutting Concerns

- Cross-cutting concerns are aspects of the architecture that affect more than one service.
- Micro-service approaches can make cross-cutting concerns (such as authentication) difficult to implement.
- Some examples of cross-cutting concerns are:
  - Caching and memory management, data validation, logging, monitoring, real-time constraints, synchronisation, etc.
- For example, each service may have its own log, but merging those logs to enable a full-system view can be challenging and difficult.

Prepare for Change

- Microservice architecture and cloud computing are (relatively) new.
- Best practice is constantly evolving and improving.
- Clouds are continuously developing new mechanisms to support the use and implementation of microservices.
- Be prepared to exploit new and improved mechanisms.
- Remember the “Improvement Kata” – strive for continuous improvement. Don’t just implement services and expect the system to be “finished.”

When your cloud app starts to get big, you need:



Microservices