



Microservices AntiPatterns and Pitfalls

Nikhileshkumar Ikhar

<https://www.linkedin.com/in/nikhar/>

What is

Antipattern

something that seems like a good idea when you begin, but leads you into trouble

Pitfall

something that was never a good idea, even from the start

Data-Driven Migration AntiPattern

- This occurs mostly when you are migrating from a monolithic application to a microservices architecture.
- **Too Many Data Migrations**
 - The main problem with this type of migration path is that you will rarely get the granularity of each service right the first time.
- **Functionality First, Data Last**
 - to avoid this antipattern is to migrate the functionality of the service first and worry about the bounded context between the service and the data later.

The Timeout AntiPattern

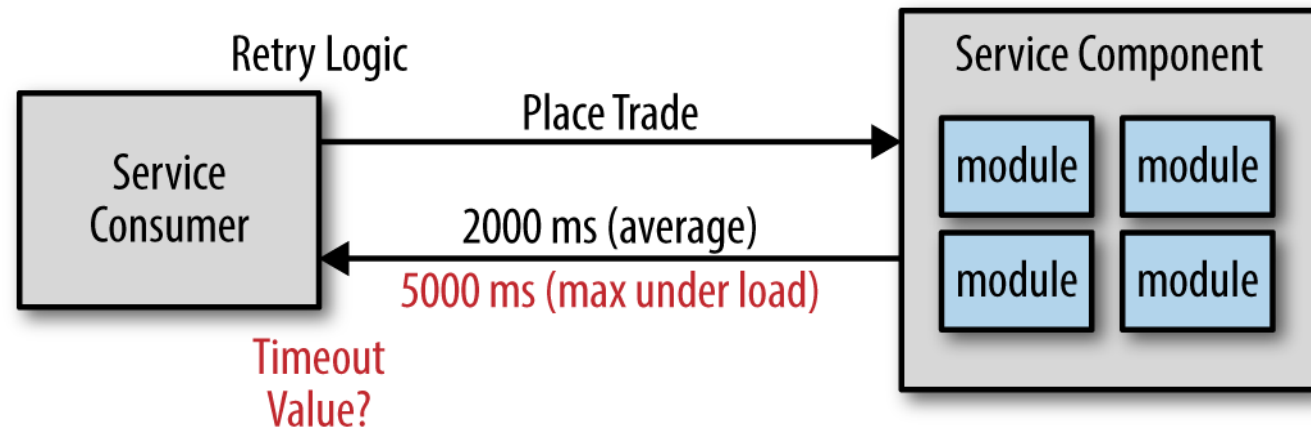
- Challenges of any distributed architecture is managing remote process availability and responsiveness.
- If service is available but not responsive, user won't wait for infinite. This will need timeout value.
- **How to calculate timeout value?**
 1. calculate the database timeout within the service
 - determine the service timeout
 2. calculate the maximum time under load
 - double it, thereby giving you that extra buffer in the event it some- times takes longer.

Trading Example.

The Timeout AntiPattern

- It causes *every request* from service consumers to have to wait 10 seconds just to find out the service is not responsive
- We need response in 2 sec

Understress



Timeout Value = 5000ms x 2 = 10000ms

The Timeout AntiPattern

Good Solution

- **Using the Circuit Breaker Pattern**

- monitors the remote service, ensuring that it is alive and responsive.
- if service is live, allow requests.
- If service is unresponsive, the circuit breaker opens, thus preventing requests from going through until the service once again becomes responsive

- **How?**

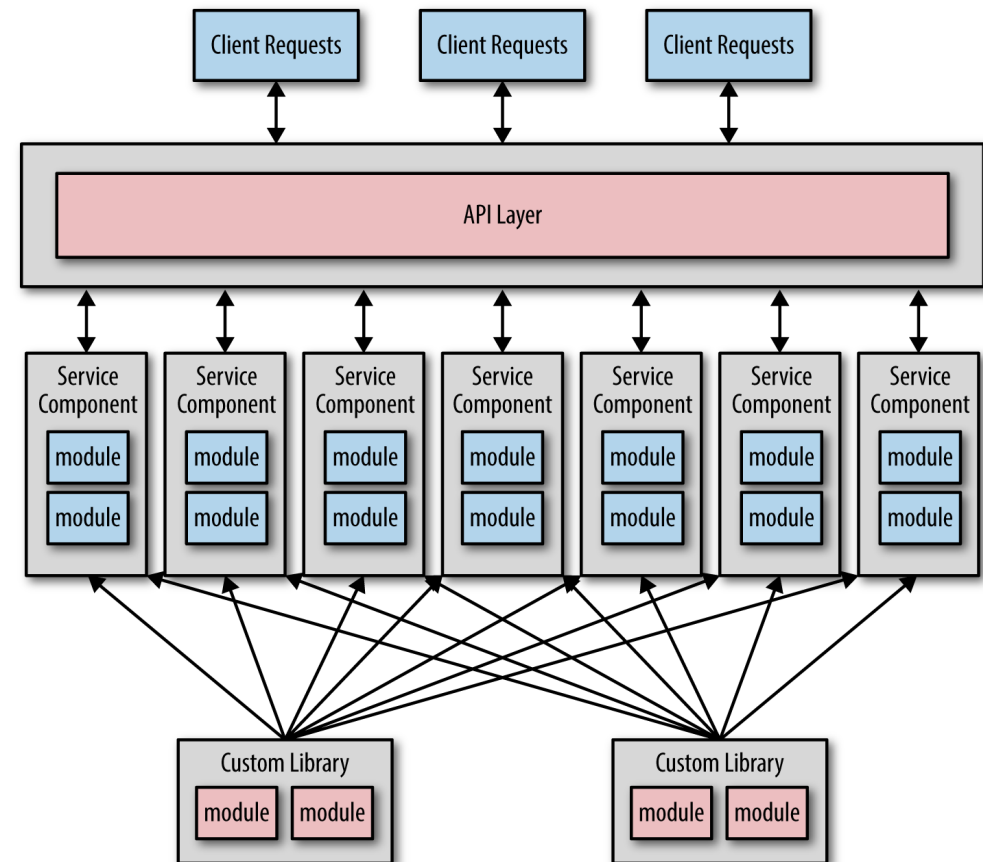
- Use Ping, fake transaction
- Monitor all request, once threshold is reached.
 - Break the path. Allow 1 of 10 request.
 - Once service is normal. Allow all requests.

- **E.g.**

- Loadbalancer
- K8s

The “I Was Taught to Share” AntiPattern

- Services use common libraries.
- We keep common libraries in .jar common.
- **Too Many Dependencies**
 - several issues, including overall reliability, change control, testability, and deployment.



Aws Cognito Okta JWT

1. Directly pull from master

The “I Was Taught to Share” AntiPattern

- **Techniques for Sharing Code**

- best way to avoid this antipattern is simply not to share code between services.
- Use libraries versioning
- create context-based libraries like *security.jar*, *persistence.jar*, *dateutils.jar*

Reach-in Reporting AntiPattern

- **Issues with Microservices Reporting**
- Database pull
 - Pull the data directly from the service databases
 - Couple of applications together use a shared database.
 - This means that the services no longer own their data.
 - database refactoring affects all applications together.

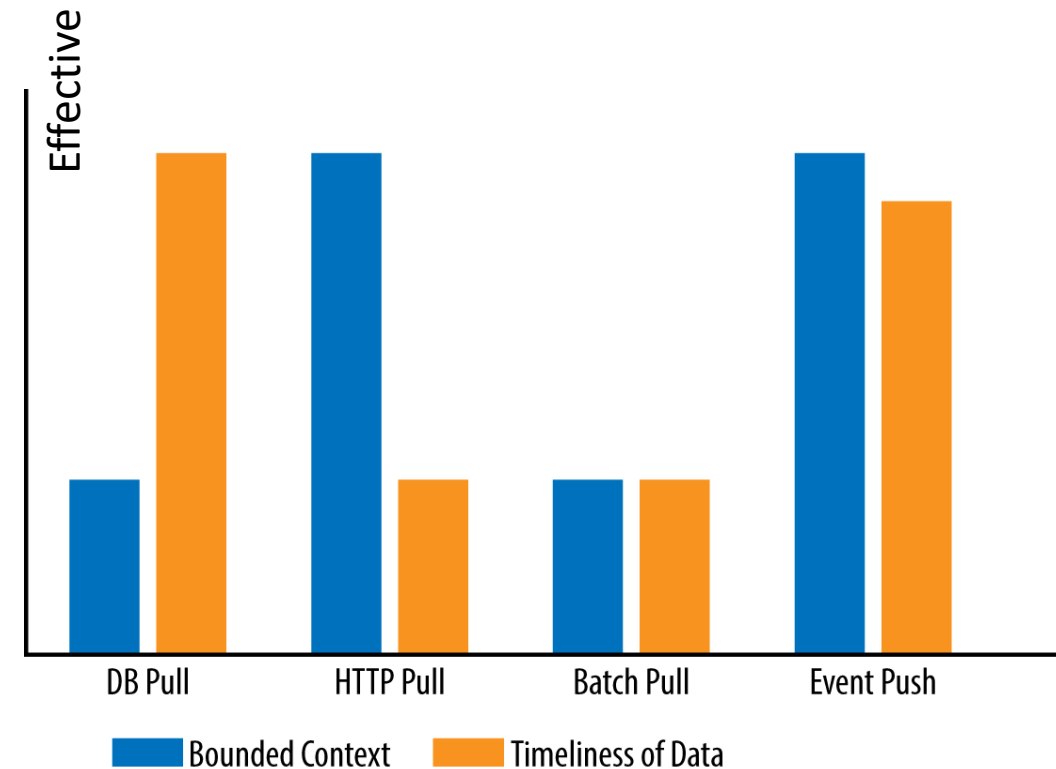
Imagine Analytics Dashboard

Reach-in Reporting AntiPattern

- Pull data through http api
 - it is unfortunately too slow, particularly for complex reporting requests.
 - the data volume might be too large of a payload for a simple HTTP call.
- Batch pull model
 - separate reporting database or data warehouse that contains the aggregated and reduced reporting data.
Runs every night
- Event-based push model
 - asynchronous event processing to make sure the reporting database has the right information as soon as possible.
Good solution. Difficult to implement
Use events to publish new data in reporting db

Reach-in Reporting AntiPattern

- The HTTP pull model preserves the bounded context, but has issues associated with timeouts and data volume.
- The batch pull model turns out to be the least-desirable model out of the four options because optimizes neither the bounded context nor the timeliness of data.
- Only the event-based push model maximizes both the bounded context of each service and the timeliness of reporting data.



Timeliness can be measured as the time between when information is expected and when it is readily available for use
Bounded context within microservices includes the service and its corresponding data

Grains of Sand Pitfall

- The *grains of sand pitfall* occurs when architects and developers create services that are too fine-grained. Wait— isn't that why it's called *microservices* in the first place? The word "micro" implies that a service should be very small, but how small is "small"?
- small as a "class"?
- A service performs a specific function in the system.
- The service should have a clear and concise roles and responsibility statement and have a well-defined set of operations.
- The number of implementation classes should not be a defining characteristic for determining the granularity of a service.

Grains of Sand Pitfall

- **Analyse Service Scope and Function**

- Documenting or verbally stating the service scope and function is a great way to determine if the service is doing too much.
- Using words like “and” and “in addition” is usually a good indicator that the service is probably doing too much.

- **Analyse Database Transactions**

- In microservice, it is extremely difficult to maintain an ACID transaction between two or more remote services.
- microservices architectures generally rely on a technique known as BASE transactions (basic availability, soft state, and eventual consistency)
- if you can't live with eventual consistency you will generally move from fine-grained services to more coarse-grained ones, thereby keeping multiple updates coordinated within a single service context,

Grains of Sand Pitfall

- **Analyze Service Choreography**

- commonly referred to as interservice communication.
- decreases the overall performance of your application since each call to another service is a remote call.
 - $5 * 100 \text{ msec} = \frac{1}{2} \text{ sec}$ in interservice communication.
- The more remote calls you make for a single business request, the better the chances are that one of those remote calls will fail or time out.
- If you find you are having to communicate with too many services to complete single business requests, then you've probably made your services too fine-grained.

Developer Without a Cause Pitfall

- Programmers know the benefits of everything and the trade-offs of nothing. 😞
- Architects must understand both.

Developer Without a Cause Pitfall

- **Making the Wrong Decisions**

- fine-grained service -> impacts performance and reliability -> impacts interservice communication time between them
- What if multiple service are consolidated to avoid above?
 - It increases reliability
- What about trade-offs?
 - Trade-offs are Deployment, change control, and testing
- Reverse is true for Coarse service to finer services.

Jump on the Bandwagon Pitfall

- While the microservices architecture is a very powerful and popular architecture style, it's not suited for every application or environment.

Advantage	Disadvantage
<i>Deployment</i>	<i>Need Organizational change</i>
<i>Testability</i>	<i>Interservice communication Performance</i>
<i>Change control</i>	<i>Reliability is low</i>
<i>Modularity</i>	<i>Invest in DevOps</i>
<i>Scalability</i>	

Jump on the Bandwagon Pitfall

- What are my business and technical goals?
- What am I trying to accomplish with microservices?
- What are my current and foreseeable pain points?
- What are the primary driving architecture characteristics for this application (e.g., performance, scalability, maintainability, etc.)?

The Static Contract Pitfall

- This occurs when you fail to version your service contracts from the very start, or even not at all.
- How to support changes in contract? (request schema)
 - use versioning
- Version can be in header
 - difficult to implement.
 - Headers are not just in HTTP.

AMQP, JMS, gRPC?

The Static Contract Pitfall

- **Schema Versioning**

- version is known after parsing the data
- complex schema interfere with automatic conversion.
 - like JSON to Java obj

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "properties": {  
    "version": {"type": "integer"},  
    "acct": {"type": "number"},  
    "cusip": {"type": "string"},  
    "sedol": {"type": "string"},  
    "shares": {"type": "number", "minimum": 100}  
  },  
  "required": ["version", "acct", "shares"]  
}
```

Are We There Yet Pitfall

- This pitfall occurs when you don't know how long the remote access call takes
- You might assume the latency is around 50 milliseconds, but have you ever measured it?
- Do you know what the average latency is for your environment?
- Do you know what the “long tail” latency is (e.g., 95, 99, 99.5 percentiles) for your environment? Measuring both metrics is important, because even with good average latency, bad long-tail latency can destroy you.

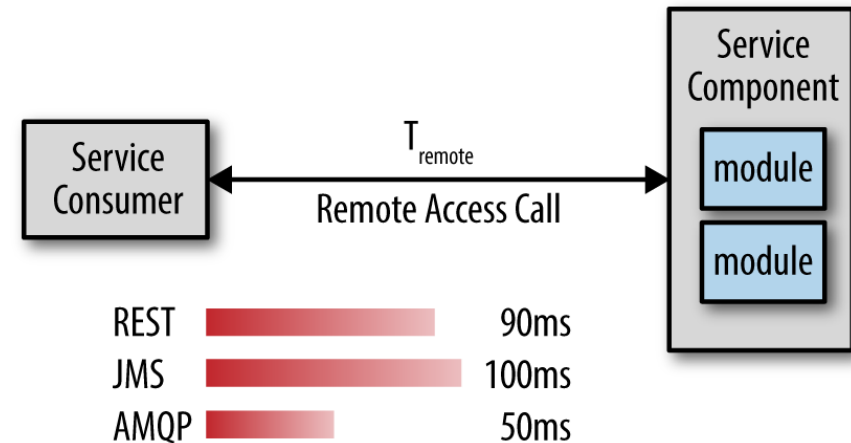
Tail latency is the small percentage of response times from a system, out of all of responses to the input/output (I/O) requests it serves, that take the longest in comparison to the bulk of its response times.

Amazon, every 100 milliseconds of latency causes a 1% decrease in sale OR

At Shopzilla, reducing latency from seven seconds to two seconds increased page views by 25%, and revenue by 7%

Are We There Yet Pitfall

- **Measuring Latency**
 - important piece we miss while measuring latency is protocol latency



Give It a Rest Pitfall

- REST is popular
- The give it a rest pitfall is about using REST as the only communication protocol and ignoring the power of messaging to enhance your microservices architecture.
- For example, in a RESTful microservices architecture,
 - how would you handle asynchronous communications?
 - What about the need for broadcast capabilities?
 - What do you do if you need to manage multiple remote RESTful calls within a transactional unit of work?

Bank

Give It a Rest Pitfall

- **Asynchronous Requests**

- Use asynchronous communication for interservice communication (Kafka)
- increases overall performance
- increases reliability
 - no need to use circuit breaker, timeout

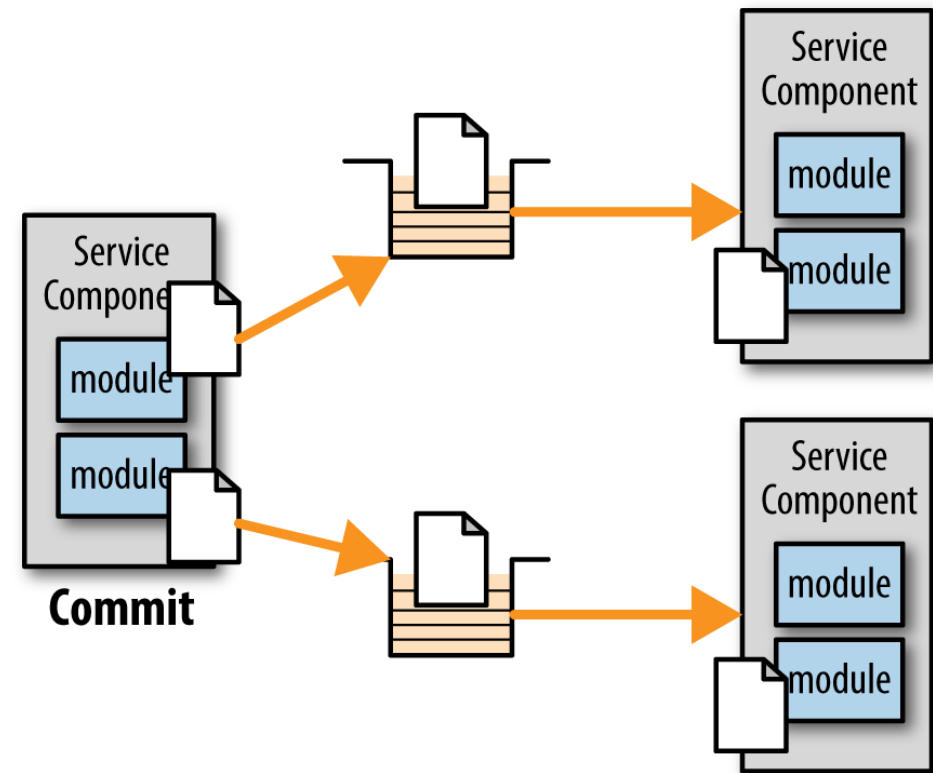
- **Broadcast Capabilities**

- REST can't broadcast (RabbitMQ can)
- where do we use broadcast? Share market data is broadcasted to various brokers

Give It a Rest Pitfall

- **Transacted Requests**

- Message is sent chunks. Commit is used to mark end of message.
- Until the service consumer performs a commit, those messages are held in the queues.
- Once the service consumer performs a commit, both messages are then released.



Ref

- Microservices AntiPatterns and Pitfalls
 - By Mark Richards

