Asynchronous Communication in Microservices

My Personal Take

Łukasz Młodzik

Everything you need for complex recurring billing

~1 million out of the box billing permutations make launch and experimentation easy

WE ARE HIRING
RUBY DEVS

Agenda

1. Microservices
   - when and why
2. Examples and implementations
3. Summary

Microservices

an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms.

Martin Fowler

Pros of Microservices

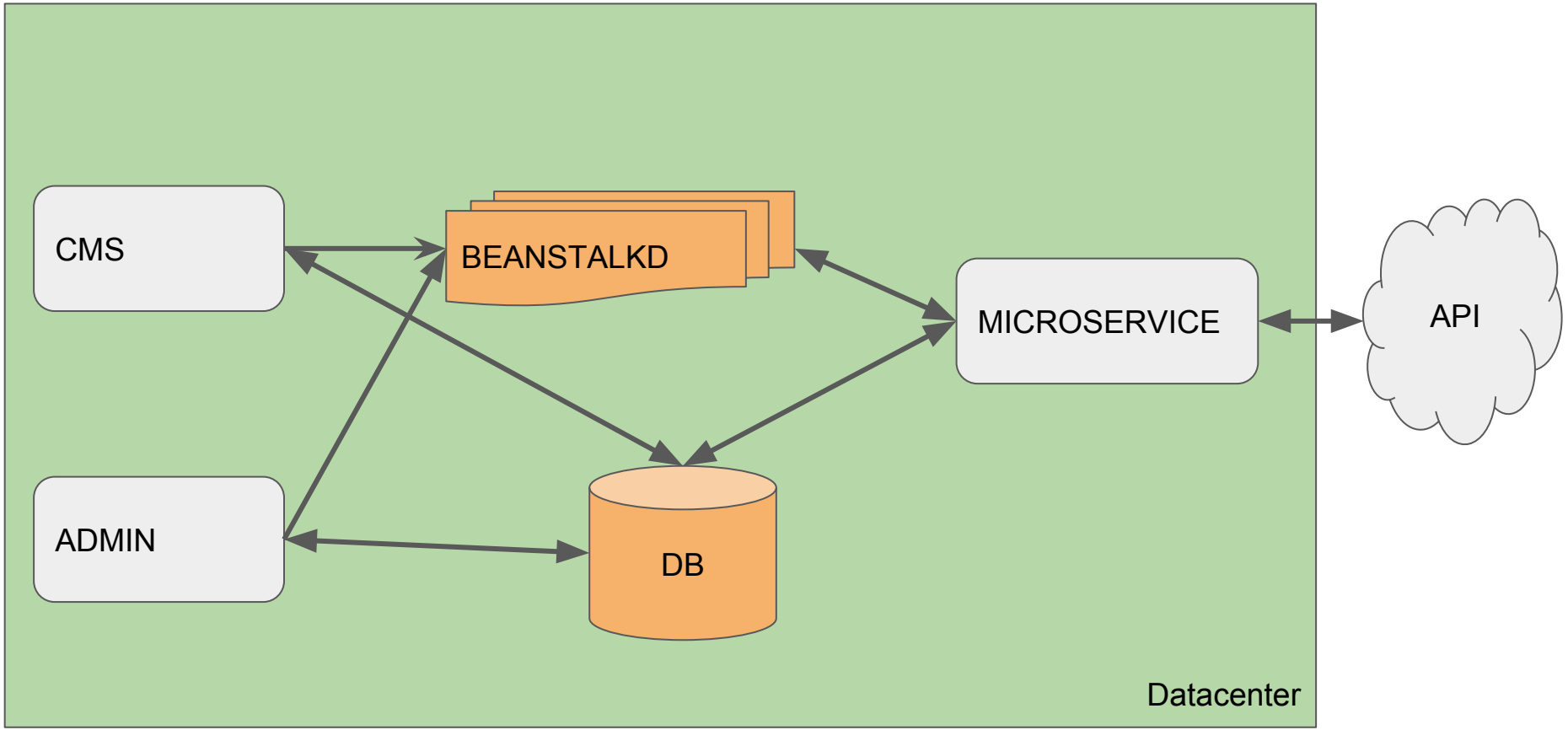
1. Separation of domain logic
2. Ability to scale horizontally
3. Faster development and feature release
4. Better modularity

Cons of Microservices

1. Harder to test
2. Higher cost of infrastructure
3. Learning curve - more complex system

Example 1

Two apps one queue


CMS
BEANSTALKD
MICROSERVICE
API
ADMIN
DB
Datacenter

What is beanstalkd?

Beanstalk is a simple, fast work queue.

```
beanstalk = Beanstalk::Pool.new(['10.0.1.5:11300'])
beanstalk.put('hello')

beanstalk = Beanstalk::Pool.new(['10.0.1.5:11300'])
loop do
  job = beanstalk.reserve
  puts job.body # prints "hello"
  job.delete
end
```

Summary

Solution:

A gem included in both projects.

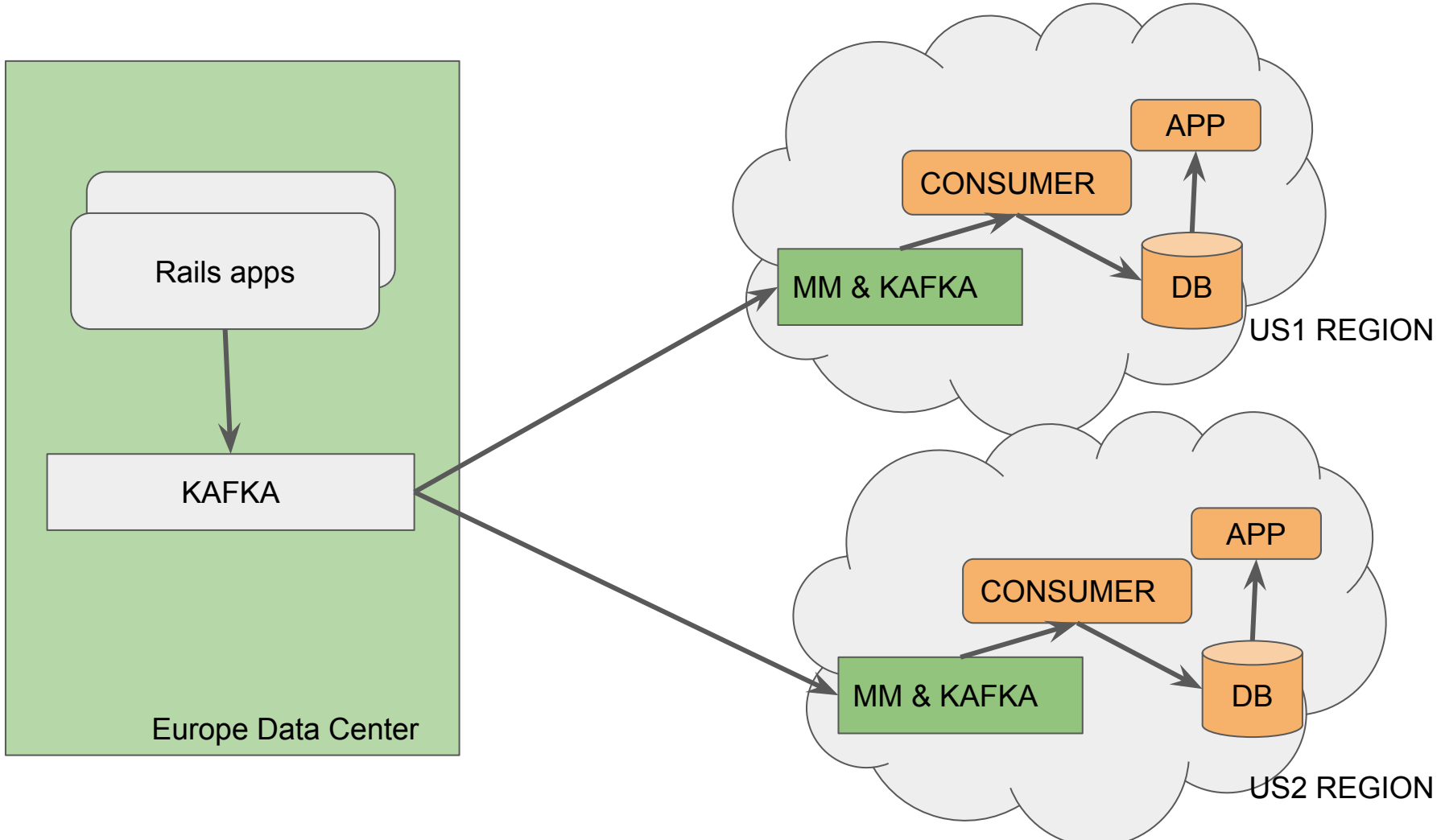
Sending ids of changed objects to microservice which in the end used same db.

Problems:

1. Problems with tracking job statuses
2. Problems with monitoring who triggered sync and where

Example 2

From one datacenter to multi region cloud


Rails apps
KAFKA
Europe Data Center
APP
CONSUMER
MM & KAFKA
DB
US1 REGION
APP
CONSUMER
MM & KAFKA
DB
US2 REGION

Kafka to the rescue

Emitt kafka messages containing prepared data.

Mirror them with MirrorMaker to the destination data center.

For cost saving data is serialized with Apache Avro.

Store them in region db.

Sample from AVRO schema

A compact, fast, binary data format.

```
{
  "name": "event_producer",
  "doc": "Producer Module Name creating this event and writing it to Kafka",
  "default": "unknown",
  "type": {
    "namespace": "com.company.types",
    "name": "EventProducer",
    "type": "enum",
    "symbols": [
      "dashboard",
      "cms",
      "ms1"
    ]}}
```

Pros

1. One source of ‘truth’
2. If main app is not reachable we can still operate
3. Serving from the closest region to user
4. Endpoints can be written in different language

Cons

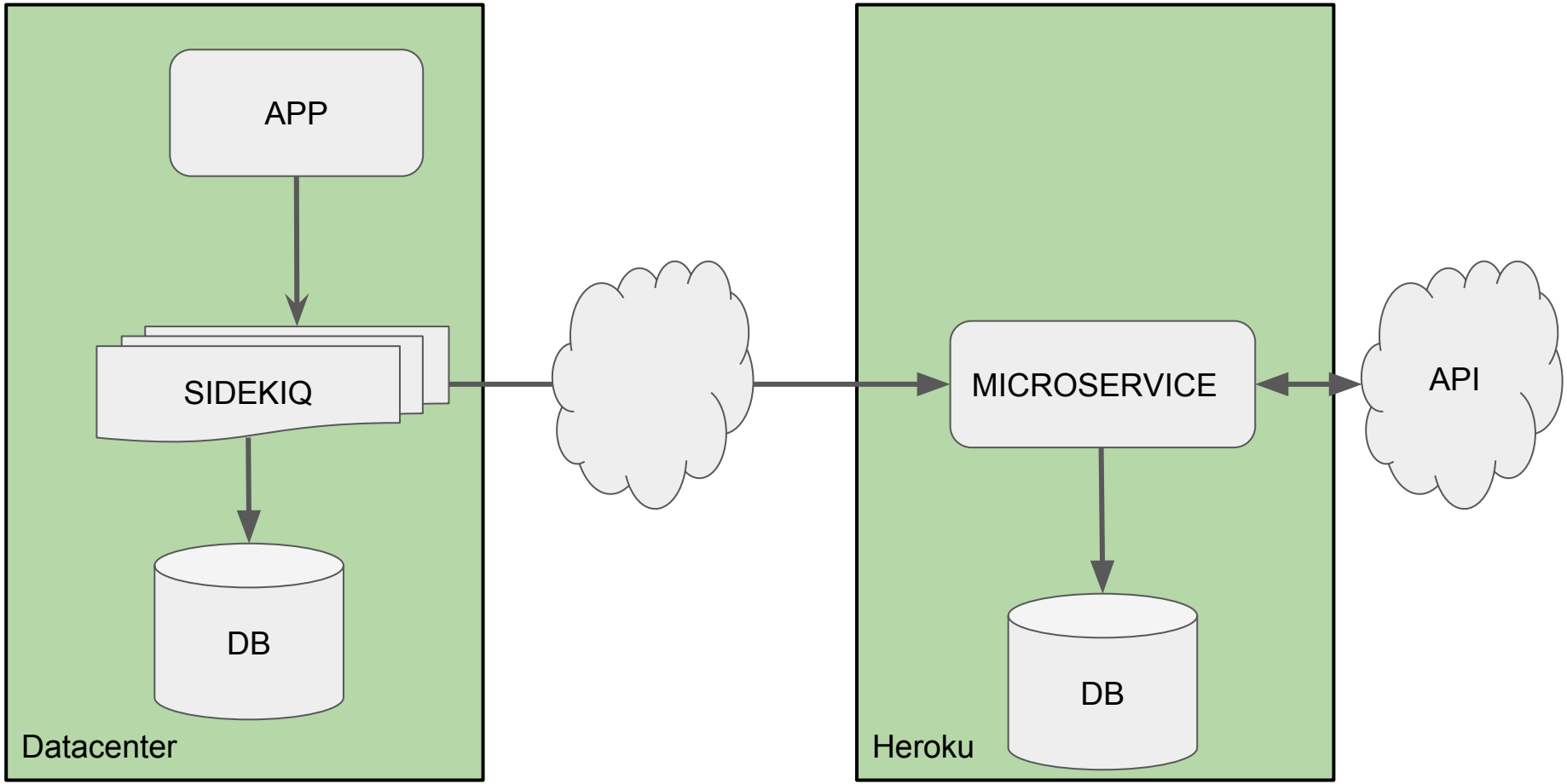
1. Data migration lag between DC’s
2. Cost of transferring data in the cloud
3. Infrastructure overhead (VPNs, security)
4. Testing and debugging is hard

How did we migrate?

1. Setup monitoring, MONITOR EVERYTHING
2. Shadow traffic (duplicate), compare graphs and outputs
3. Make a switch and monitor
4. Remove unused code

Example 3

One way data synchronisation


APP
SIDEKIQ
DB
Datacenter
MICROSERVICE
DB
Heroku
API

Pros

1. Separate team working on synchronization
2. Can be located in different datacenter (ie Heroku)
3. No rocket science (sidekiq, resque as workers)

Cons

1. Error handling and support
2. Probably you will need dashboard just to manage sync state

Sharing state

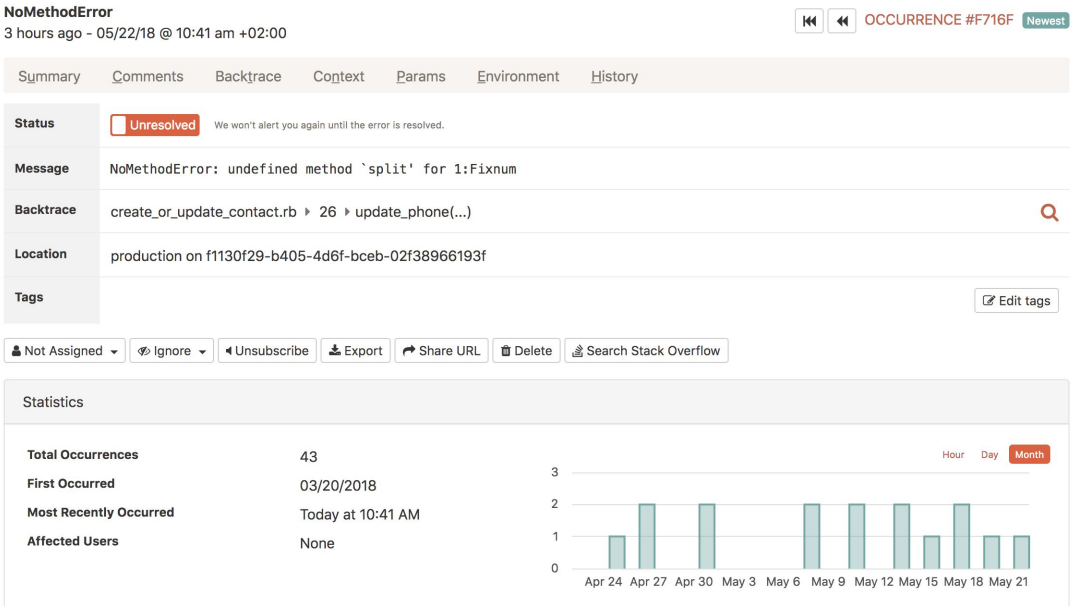
- If you need to share a state, you can have an endpoint which main app consumes regularly and updates data in main DB.
- If you have an API in main app, microservice can use it to share a state

Summary

- Always assume that communication will fail. Implement retry policies.

Summary

- Use error logging systems (honeybadger, rollbar, etc)



Summary

- Log sent and received calls.

Summary

- Use graphs to monitor services. They will show if something is wrong.

Summary

- If you can - stay with the monolith - it will ease communication pattern.
- If you don’t need to share state (ie. reporting app, generating stats) you can go with the microservices.
- Remember about monitoring and strong devops team

Thank you!