Container Bootcamp

# Microservices:

# Advantages & Challenges

## Advantages

## Scale Agile Architecture

- Let architecture drive the organization

- Limit communication between components
- …and teams

- “Meeting Avoidance Strategy”

## Technology Freedom

- Choose technology per service
- Best-of-breed easier
- Easier experiments
- Less coordination
- Higher motivation

## Robustness

- One failing Microservice doesn’t crash the system
- Separate process, VM …
- Resilience: Microservices are resilient against failing Microservices
- Deployment Monolith: A single error can crash everything

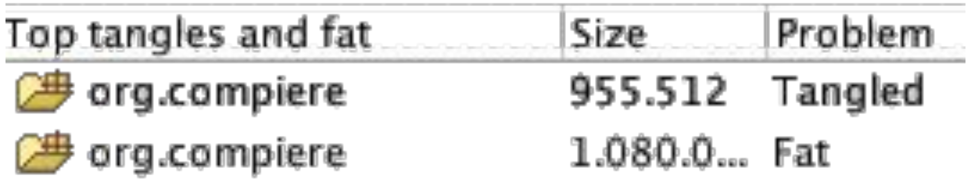
## Independent Scaling

- Microservices are distributed
- Add a load balancer
- Add more VMs
- Microservice scales
- Much easier than for a Deployment Monolith

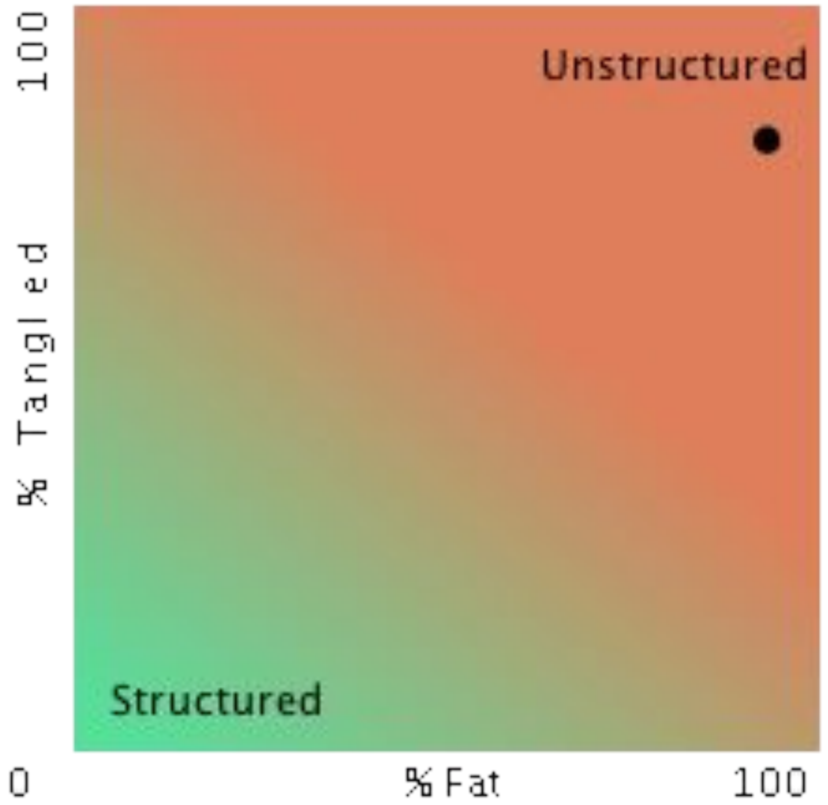
## Legacy Apps

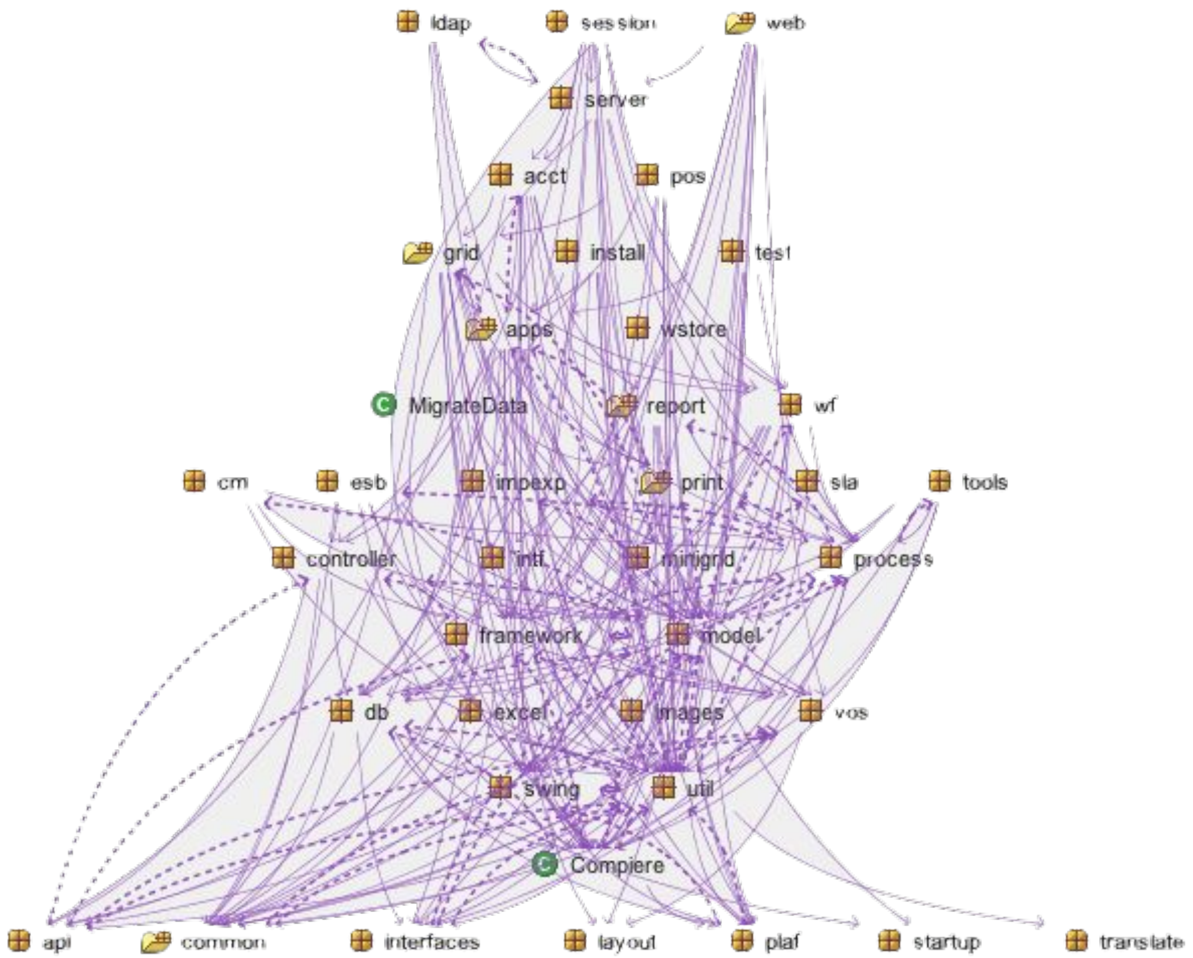
**Tangles:** package-level cycles f.e. cyclic dependencies

**Fat:** number of edges in the dependency graph

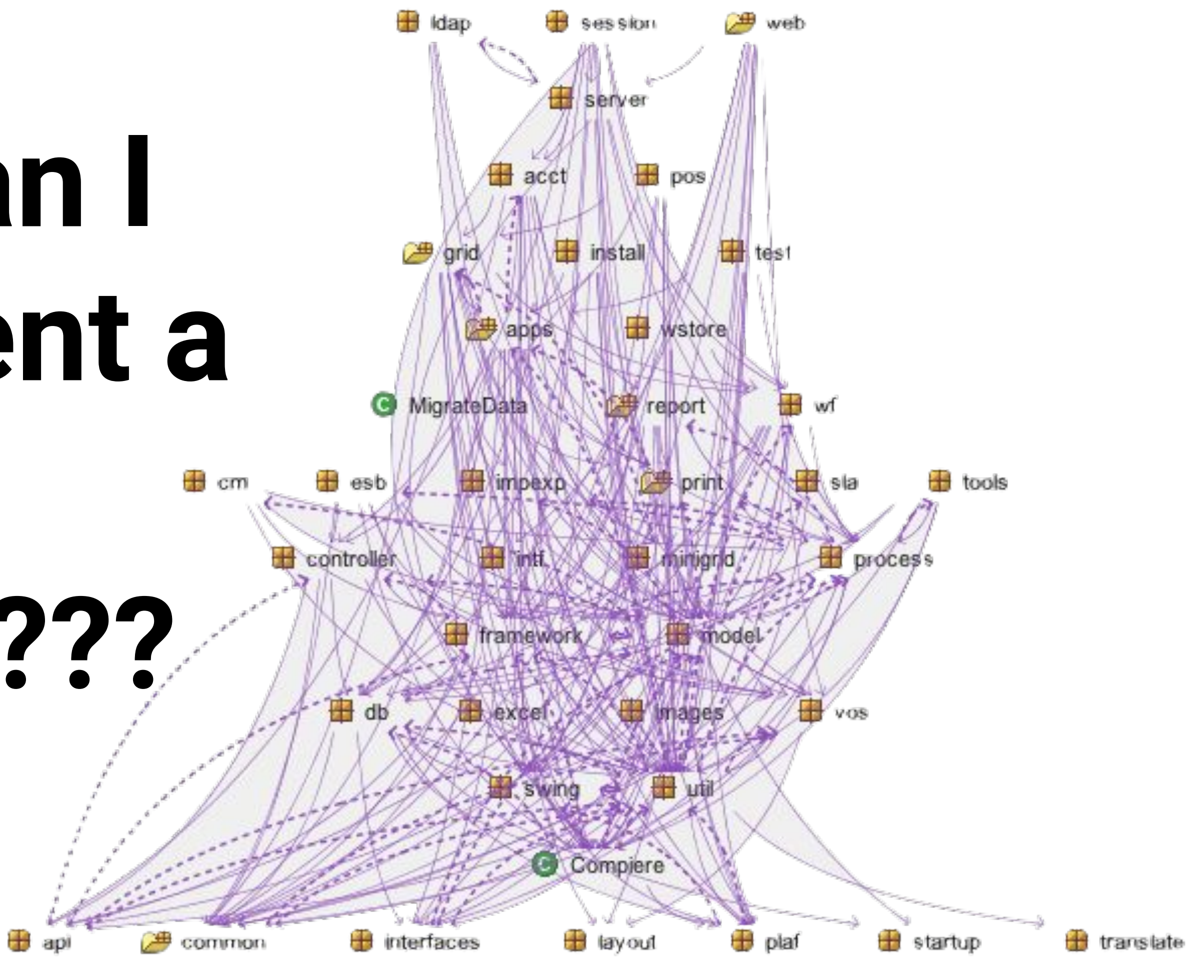


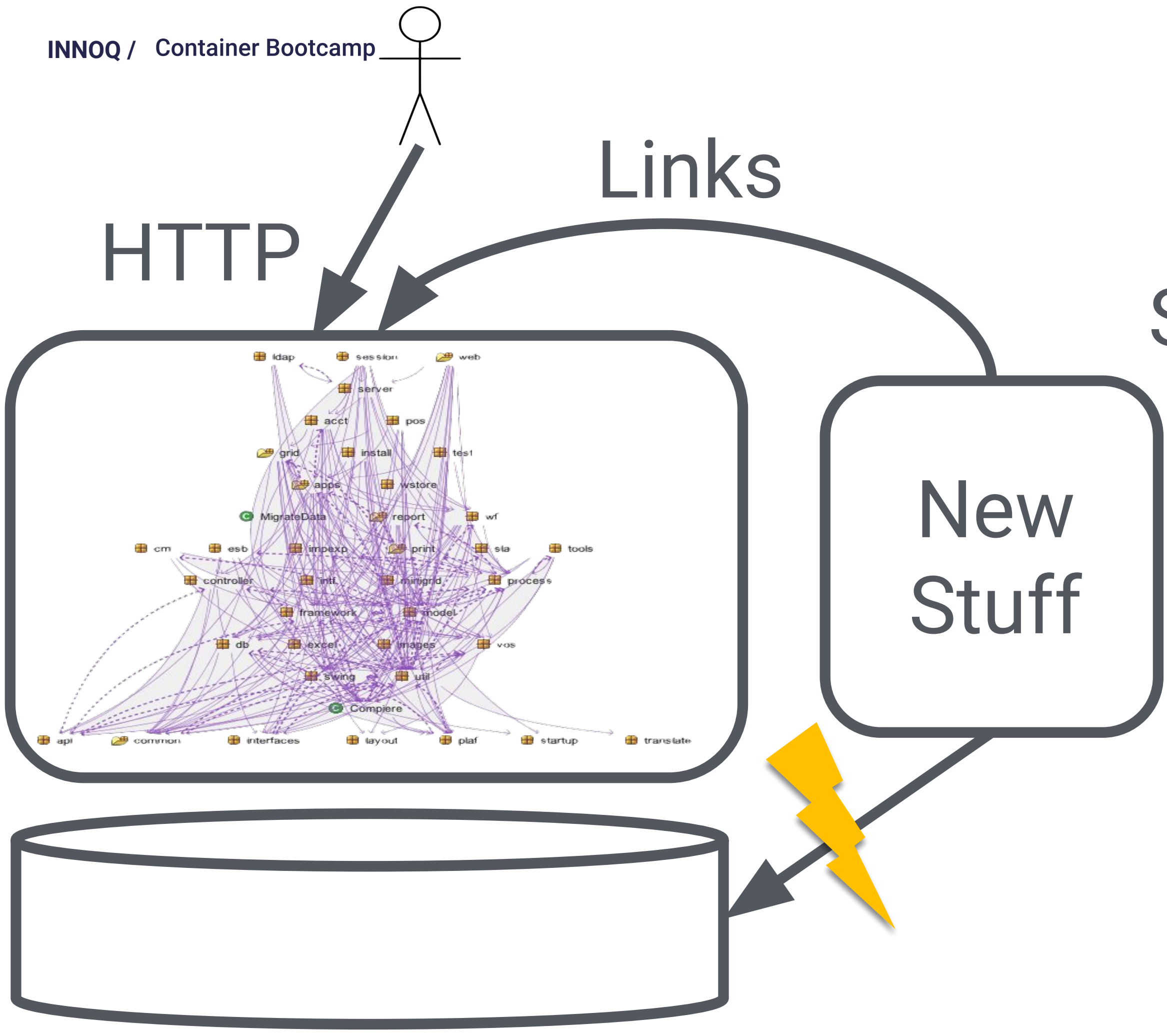
| Top tangles and fat | Size | Problem |
|---|---|---|
| org.compiere | 955.512 | Tangled |
| org.compiere | 1.080.0... | Fat |




ldap
session
web
server
acct
pos
grid
install
test
apps
wstore
MigrateData
report
wf
cm
esb
impexp
print
sla
tools
controller
intf
minigrid
process
framework
model
db
excel
images
vos
swing
util
Compiere
api
common
interfaces
layout
plaf
startup
translate

# How can I implement a new feature???





No legacy code

Any technology

Small code base

## Sustainable Development

## Monoliths

- Architecture rot
- …not maintainable any more
- …and can’t be rewritten / replaced

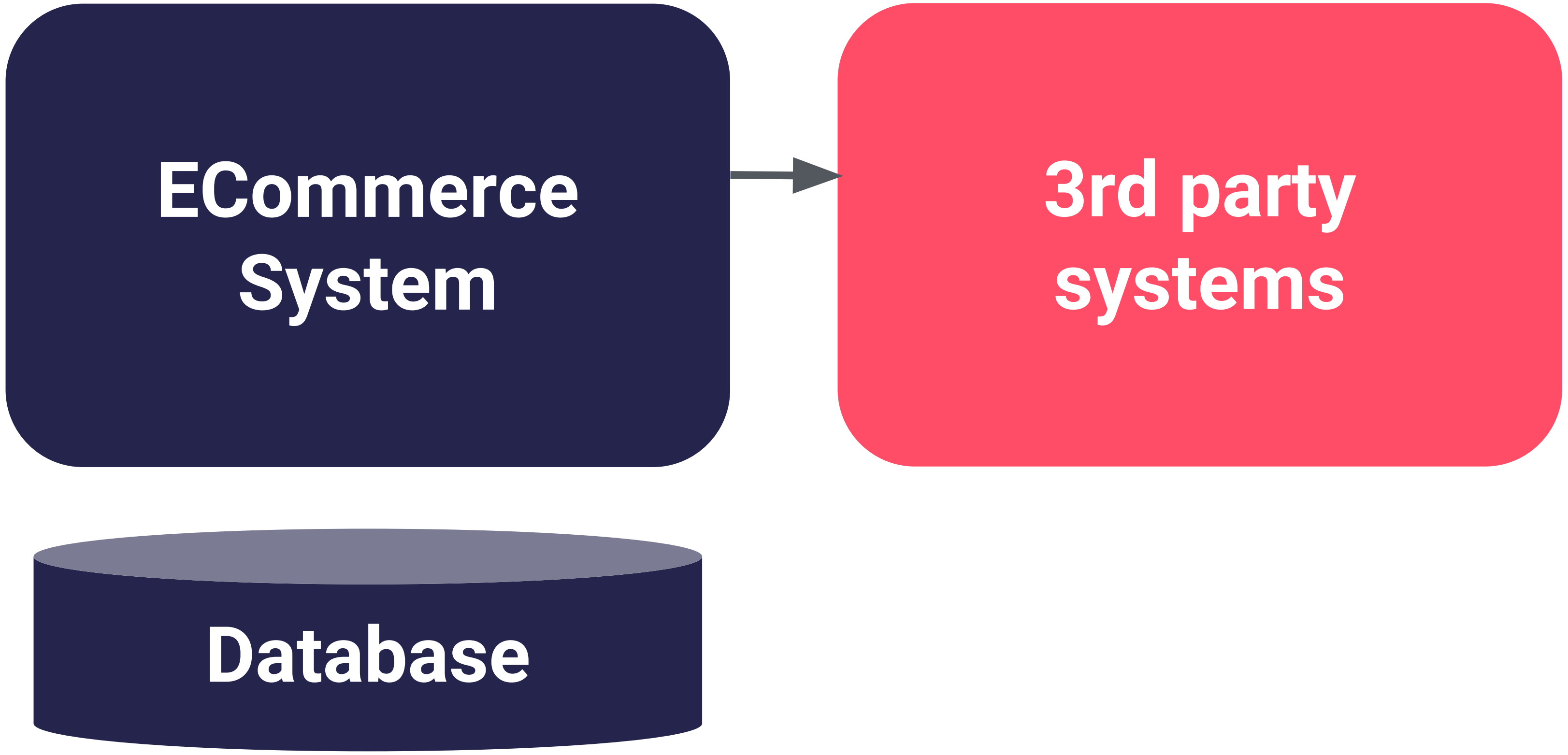
## Microservices

- Distributed system of small units
- Architecture violations harder

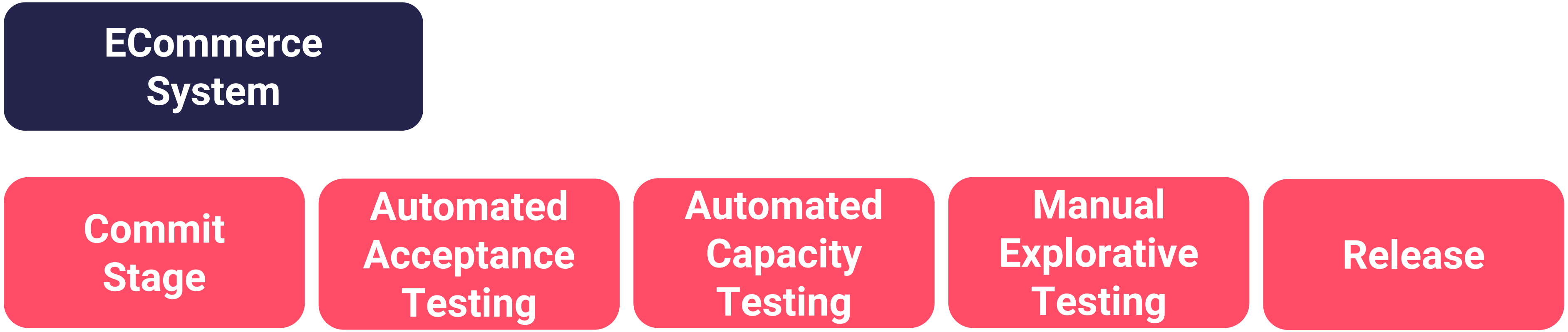
- Small units
- Easy to replace

## Continuous Delivery

## Monolith



## Continuous Delivery: Build Pipeline

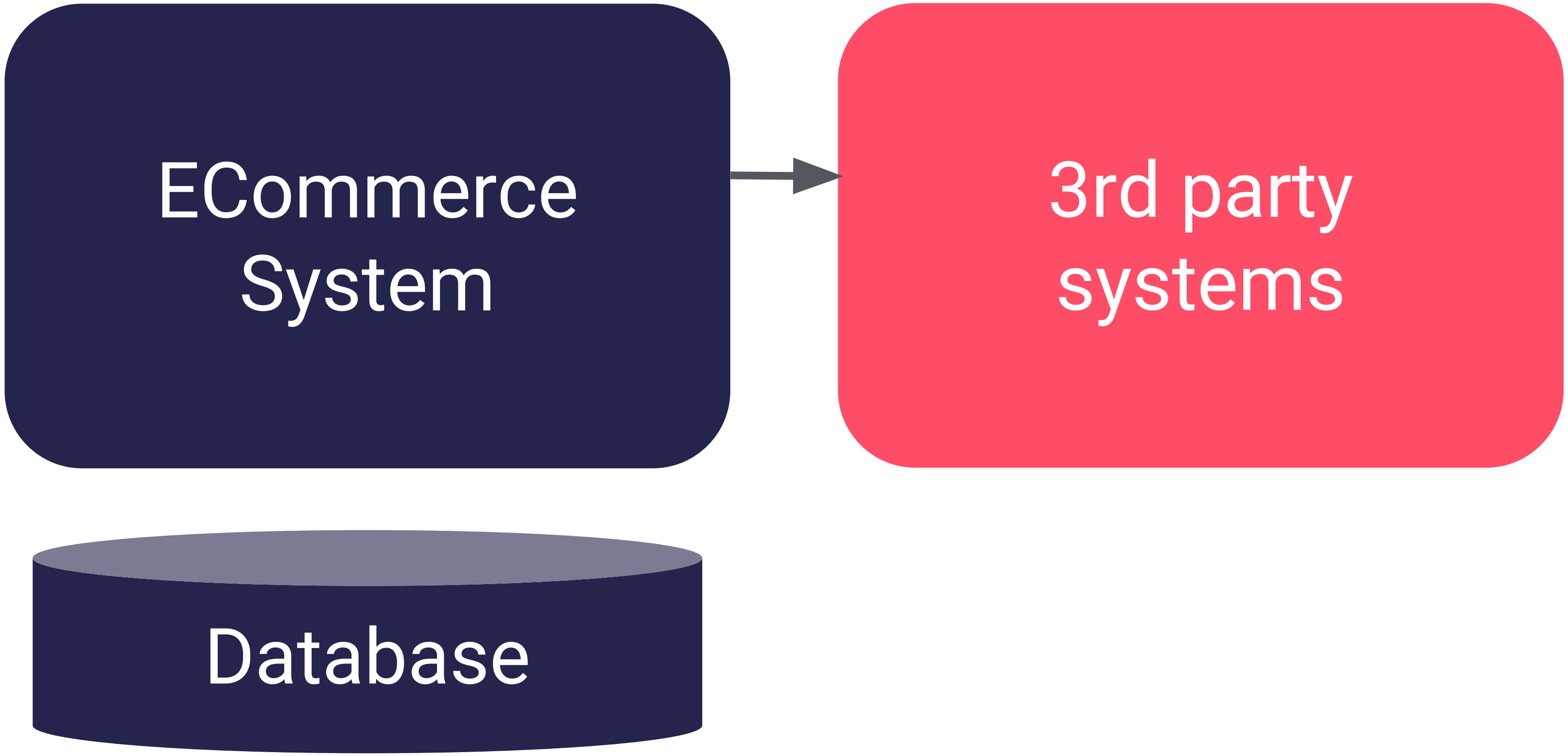

ECommerce System
Commit Stage
Automated Acceptance Testing
Automated Capacity Testing
Manual Explorative Testing
Release

## Build Pipeline: Problems

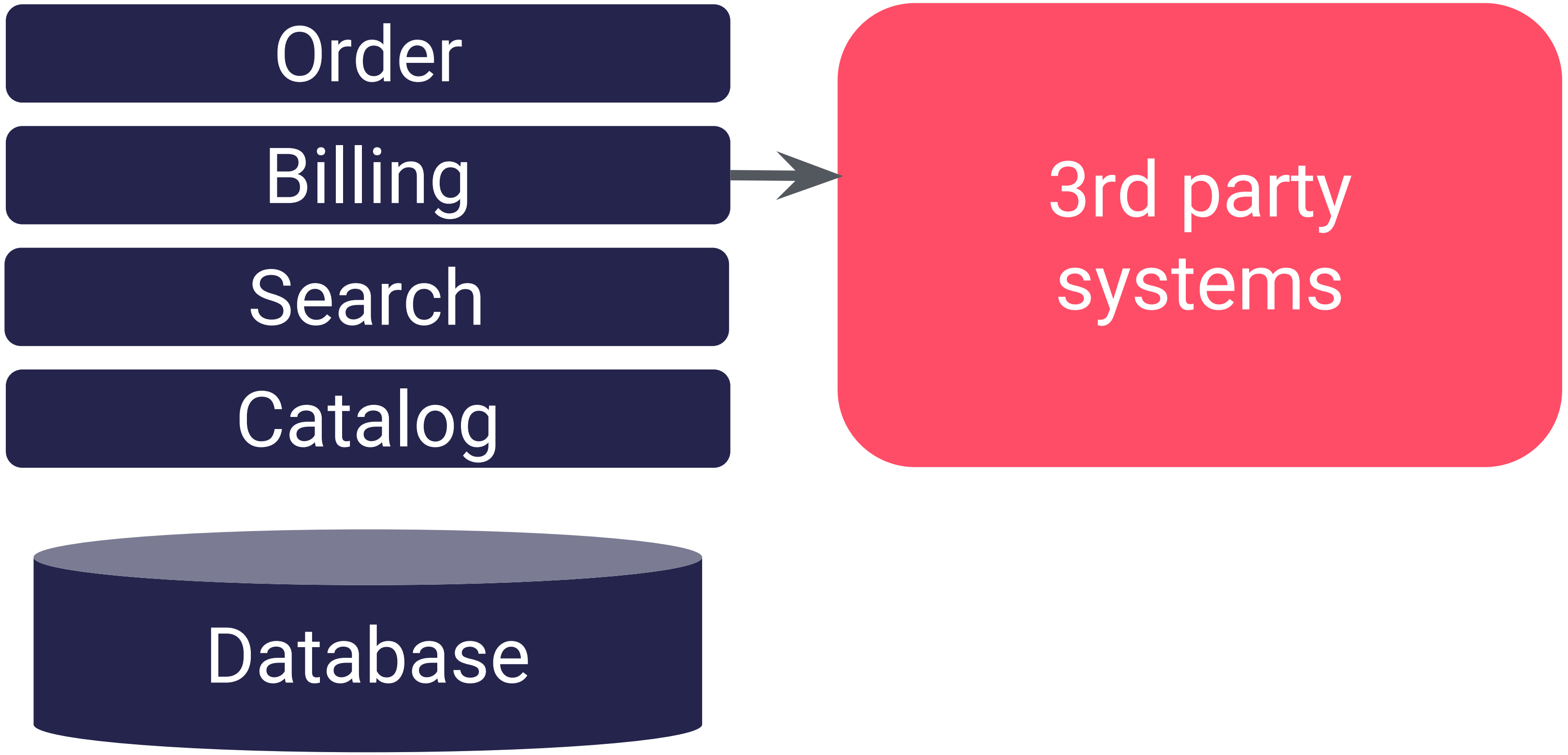
- Complex infrastructure
- Huge database
- 3rd party integration

- Slow feedback
- Test everything for each commit
- Huge deployment unit
- Deployment slow

## Microservices



## Microservices



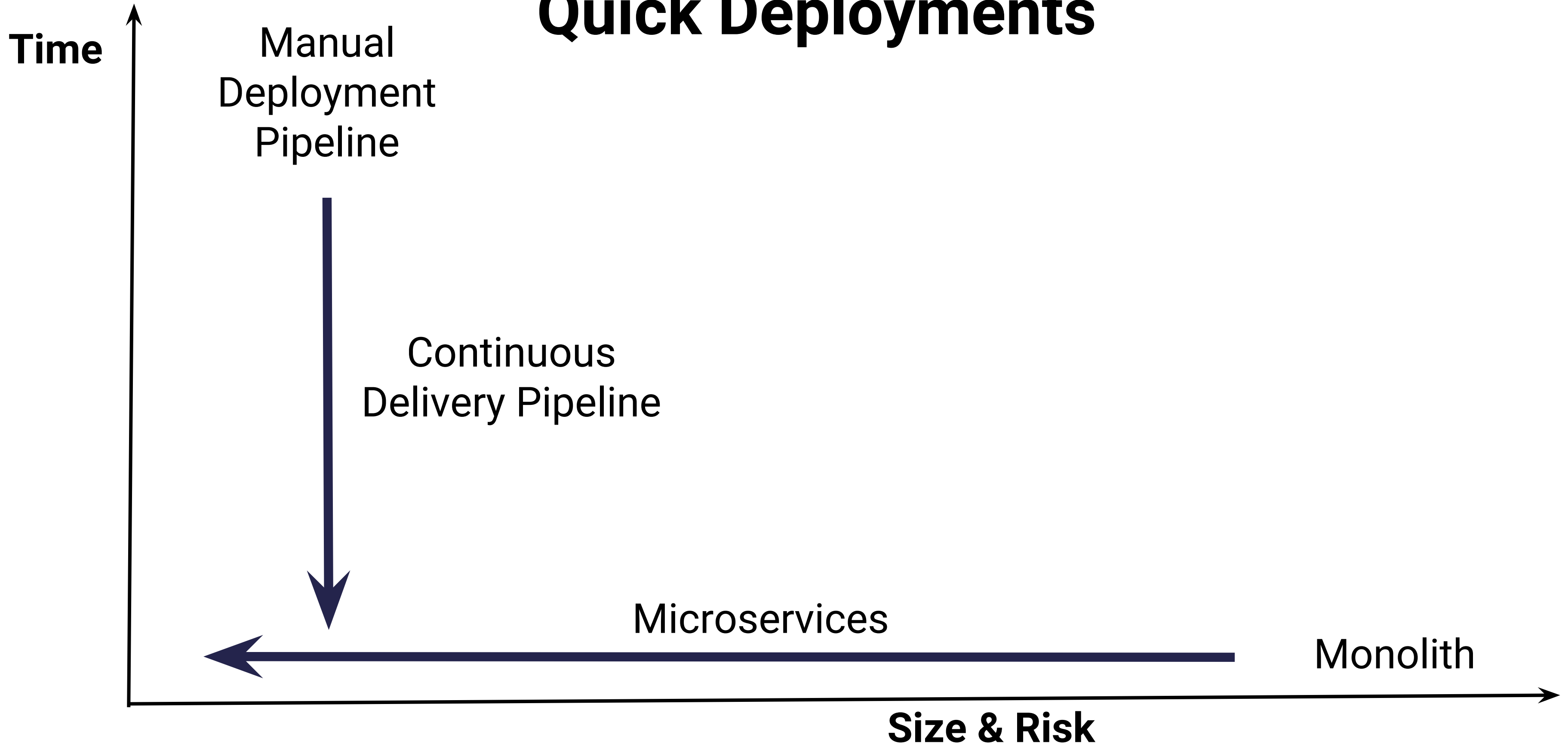
| | | | | | |
|---|---|---|---|---|---|
| Order | Commit Stage | Automated Acceptance Testing | Automated Capacity Testing | Manual Explorative Testing | Release |
| Billing | Commit Stage | Automated Acceptance Testing | Automated Capacity Testing | Manual Explorative Testing | Release |
| Customer | Commit Stage | Automated Acceptance Testing | Automated Capacity Testing | Manual Explorative Testing | Release |

## Build Pipeline for Microservices

- Independent deployment
- Build pipeline per Microservice

- Smaller
- Easier to set up
- Fewer features (3rd party systems)
- Faster Feedback: Less tests

## Quick Deployments



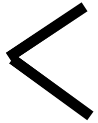
## Advantages

Vote

Scaled Agile Architecture

Handle Legacy more efficient

Sustainable development speed

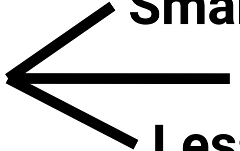


Strong Modularization

Replaceable Services

Robustness

Continuous Delivery



Smaller Deployment Units

Easier to set up CD

Less risk in deployments

Independent Scaling

Choose best technology for job!

## Challenges

## Deploy & Operate?

## Component Model

- No restriction on language etc
- Individual processes
- + infrastructure (database etc)

- Monitoring
- Logging

## Possible Component Models

- Virtual machine

- Docker container

- Installable software (RPM, deb)
- + deployment / config scripts

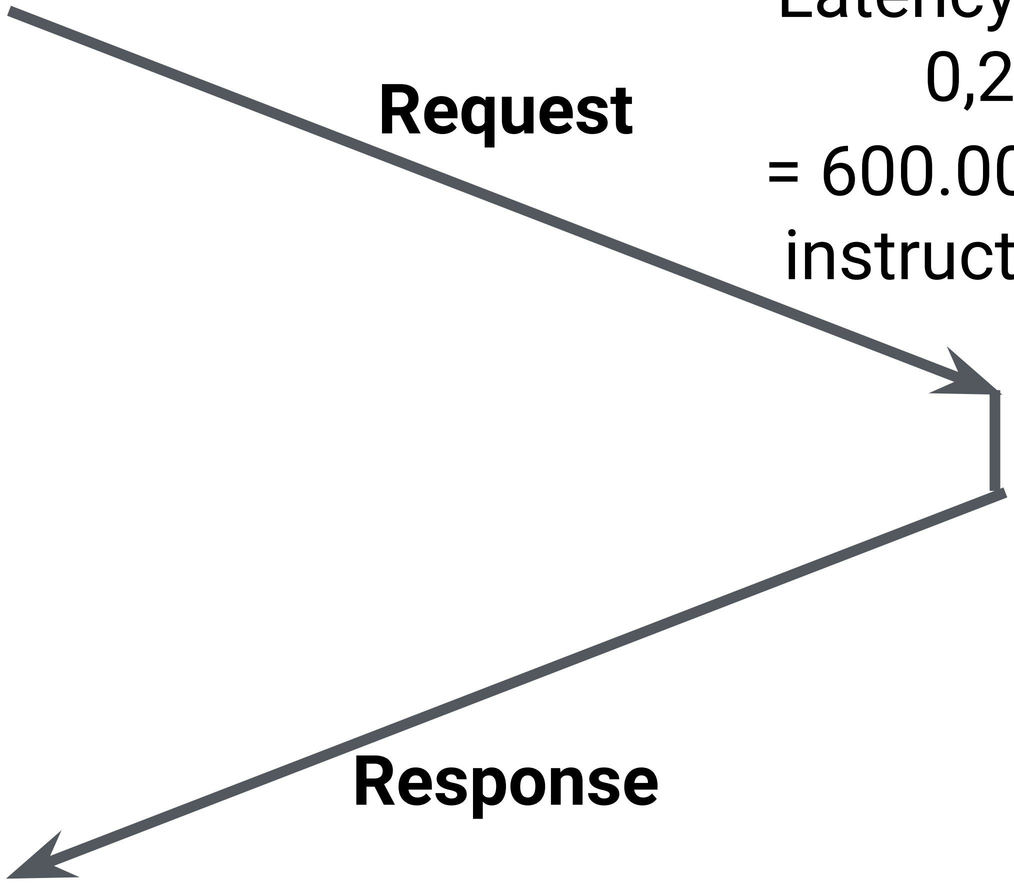
## Distributed System

## 1st Law of Distributed Objects

- Don’t Distribute Your Objects!
- Too much remote communication & overhead
- Lesson learned from CORBA etc.
- Microservice can include a GUI

- http://martinfowler.com/bliki/FirstLaw.html

Latency Round Trip:
0,2-0,5 ms
= 600.000-1.500.000
instructions@3GHz



## 1st Law & Microservices

- Small Microservices mean a lot of communication
- Violate the 1st Law
- Seems to work, though

- http://martinfowler.com/articles/distributed-objects-microservices.html

## Too small =

## too much communication

## Code Reuse

- Reuse across technology stacks hard
- Code dependencies are evil!
- Deployment dependency
- No more independent deployment
- Update hell

## Code Reuse

- Avoid code reuse!
- Or make it Open Source projects (Netflix)
- Service reuse is fine

## Global Refactorings?

- Move code from service to service
- Might be a port to a different language
- Separate in a new service?
- More services = more complex

- Very hard

## Functional Architecture

- **Teams should be independent**
- **i.e. one team = one functionality**
- **Otherwise: Coordination hard**

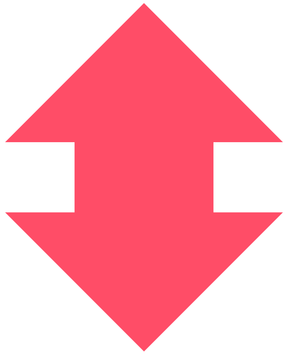
- **Functional architecture much more important**
  (FA: identifies system function and their interactions)

## Refactoring & Code Reuse

⇕

## Individual Technology Stacks

## Refactoring hard



## Functional architecture much more important

## Need to get architecture right first time

## Architecture evolves

## Start BIG

- Won’t have too much code at the start anyway
- Refactoring easier
- Can build architecture as you go

- Let the functional architecture grow!

## Managing Dependencies Between (>100) Services?

- Monoliths can be easily analyzed
- Microservices?
- With heterogeneous technology stacks?
- Need to come up with your own solution

## Global Architecture?

- Defines common communication infrastructure
- Optional: Common Ops
- Very different from usual architecture

## Global Architecture

- Manages dependencies
- Which service/team does what?

- Detailed Understanding not needed
- Enough to understand Microservices per Use Case

## Network?

- Microservices = Distributed Systems
- Services & network might fail
- Need to deal with failure

- Resilience: System must survive failure of parts
- Makes system highly available

## Deployment?

- One Deployment Pipeline per services
- Should be fully automated
- Too many services for manual steps

- Infrastructure investment needed
- Common deployment?

## Infrastructure?

- >50 server per system
- Resource consumption probably high

- Virtualization must be fully automated
- Docker might save ressources

## Conclusion

## Use If…

- Time to market is important
- Legacy systems must be modernized
- Sustained development speed
- Continuous Delivery should be implemented
- Independent scaling or robustness are relevant.