microservices

THE HUNTING OF THE SNARK
Chapter 0

The “London school of software engineering”
ThoughtWorks®
CONSULTANTS!

HELL YEAH!
ADVERTISE

1. Decoupling deployment from release
2. Products over projects
3. Threat Modeling

TRIAL

4. BFF - Backend for frontends
5. Bug bounties
6. Data Lake
7. Event Storming
8. Flux
9. Idempotency filter
10. iFrames for sandboxing
11. NPM for all the things
12. Phoenix Environments
13. QA in production
14. Reactive architectures

ASSESS

15. Content Security Policies new
16. Hosted IDE's
17. Hosting PII data in the EU new
18. Monitoring of invariants
19. OWASP ASVS new
20. Serverless architecture new
21. Unikernels new
22. VR beyond gaming new

Unable to find something you expected to see? Your item may have been on a previous radar.
Microservices

The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

25 March 2014

James Lewis
James Lewis is a Principal Consultant at ThoughtWorks and member of the Technology Advisory Board. James' interest in building applications out of small collaborating services stems from a background in integrating enterprise systems at scale. He's built a number of systems using microservices and has been an active participant in the growing community for a couple of years.

Martin Fowler
Martin Fowler is an author, speaker, and general loud-mouth on software development. He's long been puzzled by the problem of how to componentize

Contents
- Characteristics of a Microservice Architecture
  - Componentization via Services
  - Organized around Business Capabilities
  - Products not Projects
  - Smart endpoints and dumb pipes
  - Decentralized Governance
  - Decentralized Data Management
  - Infrastructure Automation
  - Design for failure
  - Evolutionary Design
- Are Microservices the Future?

Sidebars
- How big is a microservice?
- Microservices and SOA
- Many languages, many options
- Battle-tested standards and enforced standards
- Make it easy to do the right thing
- The circuit breaker and production ready code
- Synchronous calls considered harmful
“So our core microservice article got 45,144 unique page views last month, and is currently running at 1837 per day” @martin
25. High performance envy/web scale envy

We see many teams run into trouble because they have chosen complex tools, frameworks or architectures because they 'might need to scale'. Companies such as Twitter and Netflix need to be able to support extreme loads and so need these architectures, but they also have extremely skilled development teams able to handle the cost. Most situations do not require these kinds of feats; teams should keep their web scale envy in favor of simpler solutions that still get the job done.

https://www.thoughtworks.com/radar/
Chapter I

The architects dream
Airline problems with: monolithic databases ~ 2010
Retail Site

48 Cores
256 GB RAM (NUMA)
\[ \sim 1 \times 10^6 \text{ per machine} \]

Departure Control
Airline

- Tightly coupled
- Single point of scaling
- Single point of failure
- Expensive to change
- High operational cost
- High cost of failure
Retail Site

Department Control
F5

Departure Control
BACK IN 2004 (ISH)
BACK IN 2004 (ISH)

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

The mandate closed with:

"Anyone who doesn't do this will be fired. Thank you; have a nice day!"

Everyone got to work and over the next couple of years, Amazon transformed itself, internally into a service-oriented architecture (SOA), learning a tremendous amount along the way.
**Middleware DB**

- **UI**
- **Finance**
- **HR**
- **SSO**

**Direct db access**

- **Views of external data**
- **Read only external data**

**HR**

- **UI**

**Finance**

- **UI**

**Data Warehouse**

- **Canned reports**
- **Cubes / ad-hoc**

- **Direct db access**

**SSO**

- **UI**
The stovepipe enterprise

Stovepipes are “systems procured and developed to solve a specific problem, characterized by a limited focus and functionality, and containing data that cannot be easily shared with other systems.” (DOE 1999)

Logic scattered all over the place

Data scattered all over the place

Difficult to predict the effect of changes

Where are the sources of truth?

BI / MI almost impossible to get at
Insurance - 2011
+$\Delta$ features  

-$\Delta$ features
extremely high cost of delay
Summary of CTM

• basically
  – High cost of delay:
    • long lead times
      • Project teams
      • Long lived branches and awful merges

• (very coupled so stamping over each others code)
problems thereof

• High cost of change
  – sufficiently complex systems become more coupled over time
    • Martin’s tech debt quadrant

• High operational cost

• Lead time to business impact
programmer
anarchy

Java
the UNIX way

fine-grained
SOA
MORE TIME PASSES...
MORE TIME PASSES...
EVEN FINANCE IS NOT IMMUNE

An easier way to pay.

We make money easy.

After months of work (and a wake-off or two) we’re officially a bank.
We’re not open for business just yet, but it’s a huge step forward.
And we couldn’t be happier.

More than skin deep
Chapter 2

What are Microservices?
“loosely coupled service oriented architecture with bounded contexts”

Adrian Cockcroft, GOTO Aarhus 2014
“the first post-devops architectural style”

Neal Ford
replaceable component architectures

Dan North
<table>
<thead>
<tr>
<th>characteristics of microservices</th>
</tr>
</thead>
<tbody>
<tr>
<td>componentisation via services</td>
</tr>
<tr>
<td>organised around business capabilities</td>
</tr>
<tr>
<td>decentralised data management</td>
</tr>
<tr>
<td>products not projects</td>
</tr>
<tr>
<td>decentralised governance</td>
</tr>
<tr>
<td>smart endpoints and dumb pipes</td>
</tr>
<tr>
<td>evolutionary design</td>
</tr>
<tr>
<td>infrastructure automation</td>
</tr>
<tr>
<td>designed for failure</td>
</tr>
</tbody>
</table>
## Characteristics of microservices

<table>
<thead>
<tr>
<th>Componentisation via services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organised around business capabilities</td>
</tr>
<tr>
<td>Decentralised data management</td>
</tr>
<tr>
<td>Products not projects</td>
</tr>
<tr>
<td>Decentralised governance</td>
</tr>
<tr>
<td>Smart endpoints and dumb pipes</td>
</tr>
<tr>
<td>Evolutionary design</td>
</tr>
<tr>
<td>Infrastructure automation</td>
</tr>
<tr>
<td>Designed for failure</td>
</tr>
</tbody>
</table>
“It is perfectly true, as philosophers say, that life must be understood backwards. But they forget the other proposition, that it must be lived forwards.”

Søren Kierkegaard
History

The lawful good product owners of the publishing house had long lived in awe and fear of their publishing systems.

In awe, for they had made a tremendous amount of Gold, but in fear of the time taken to change them, their slowness and their fragility.

A messenger was sent to fetch help from a distant land famed for its mighty wizards. You have taken up the challenge...
You must save the product owners by rebuilding their content delivery system. You start off the project. In the course of discussions you discover that your goals are three fold:

1. improve availability
2. improve performance
3. reduce the cost of delay

An Enterprise Architect approaches and addresses you.

You may use:

- Summon Walking Skeleton  turn to 4
- Analysis Paralysis  turn to 3

If you have none of these you will have to draw your sword and fight (turn to 178)
3.

You cast Analysis Paralysis at the Enterprise Architect.

“Foolish young adventurer” says the architect, “we follow the evolutionary school of architecture and we shall have none of the lawful-evil ways of waterfall”.

The last thing you see before everything goes dark is the architect incanting in a strange voice.

You have died. Turn to page 1.
1.

You must save the product owners by rebuilding their website. You start off the project. In the course of discussions you discover that your goals are three fold:

1. improve availability
2. improve performance
3. reduce the cost of delay

An Enterprise Architect approaches and addresses you.

You may use:

- Summon Walking Skeleton  turn to 4
- Analysis Paralysis  turn to 3

If you have none of these you will have to draw your sword and fight (turn to 178)
4.

Your walking skeleton coalesces in a cloud of noxious gasses and solidifies as a java dropwizard application.

You reach into your backpack and deploy the content store. Your walking skeleton reaches out its skeletal arms and grabs armfuls of raw xml.

Would you like to:

- Transform the xml inside the skeleton  turn to 6
- Use a magic box  turn to 5
5.

You throw the magic box in between the walking skeleton and the content store.

A villager approaches and exclaims: “this beautiful content I see in front of me seems to take an awful long time to get here”

You must somehow make the content arrive faster.

If you have a http cache in your inventory, you may use it now.

- Cache in between S3 and content  
  turn to 10
- Cache in between skeleton and content  
  turn to 33
6.

The skeleton gurgles, grunts and then doubles in size.

A villager approaches and exclaims: “this beautiful content I see in front of me seems to take an awful long time to get here”

You try to add a cache into the skeleton’s bony skull. First you cast sticky sessions. With a splash it rebounds, soaking you in the stench of the unscalable.

Desperately, you try terracotta and then the oracle of coherence. Nothing seems to work. The murky substances overwhelm you.

You have died. turn to page 1.
The cache causes the content load times to drop from 300ms to 150ms.

The villager says “this wonderful content is now arriving more swiftly than even the knight-messengers of the Empress”.

The villagers are happy but all too soon, all is not well for the content has a long tail. You must work out how to refresh the content when it changes.

You can either:

- Refresh the content when it appears from the ether [turn to 150]
- Trust that it will be fast enough on first view [turn to 22]
The tail is just too long. When villagers or merchants try to use the content it is just too slow to arrive.

The amount of Gold diminishes and over the years the village fades into a forgotten hamlet, then to a legend and a myth.

You have died, turn to page 1.
Content trickles into the store. You keep up by listening for the new content and casting “wget” on the cache to keep it refreshed.

New types of content appears - content the villagers have never seen before. Content the walking skeleton is unable to combat.

Fortunately, through Continuous Delivery you are able to keep up with the changed content but the cache doesn’t. The cache becomes stale.

How will you keep your delivery continuous?

- cast cache shards turn to 255

If you are unable to shard the cache turn to page 48
33.
The HTTP cache has an instant effect. Latency drops from 300ms to 10ms.

Changes to the content mount up. Every time one of the lawful-good researches publishes something, the cache must be refreshed. Every time the skeleton changes it’s appearance, the cache must be refreshed.

The villagers need you to do something. Will you:

- Suffer the long tail turn to 22
- Refresh the cache on API and content changes turn to 150
Your walking skeleton coalesces in a cloud of noxious gasses and solidi as a java dropwizard application. You reach into your backpack and deploy the S3 content store. Your walking skeleton reaches out it's skeletal arms and grabs armfuls of raw xml.

Would you like to:

- Transform the xml inside the skeleton
  - turn to 6
- Use a magic box
  - turn to 5
characteristics of microservices

- componentisation via services
- organised around business capabilities
- decentralised data management
- products not projects
- decentralised governance
- smart endpoints and dumb pipes
- evolutionary design
- infrastructure automation
- designed for failure
## Characteristics of Microservices

<table>
<thead>
<tr>
<th>Organised Around Business Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Componentisation via Services</td>
</tr>
<tr>
<td>Decentralised Data Management</td>
</tr>
<tr>
<td>Products Not Projects</td>
</tr>
<tr>
<td>Decentralised Governance</td>
</tr>
<tr>
<td>Smart Endpoints and Dumb Pipes</td>
</tr>
<tr>
<td>Evolutionary Design</td>
</tr>
<tr>
<td>Infrastructure Automation</td>
</tr>
<tr>
<td>Designed for Failure</td>
</tr>
</tbody>
</table>
A **capability** is a combination of people, processes, systems that provides value to customers (internal or external)

The **what** of the business, not the **how**
property company

Commercial property

Residential property

The Money
The Money

Billing

Forecasting
insurance company

home

motor

life
insurance company

home

motor

life

and cross-cutting capabilities

my account
Each capability decomposed into smaller sub-domains based on your **functional** and **cross-functional** needs.
How big are they?
“objects should be no bigger than my head”
AND WHILE I HAVE A GIANT HEAD, ITS NOT FULL OF MUCH STUFF SO THATS OK...
AS WE CHUNK UP DOMAINS, EACH DOMAIN SHOULD BE SMALL ENOUGH TO FIT IN MY HEAD.
<table>
<thead>
<tr>
<th>characteristics of microservices</th>
</tr>
</thead>
<tbody>
<tr>
<td>componentisation via services</td>
</tr>
<tr>
<td><strong>organised around business capabilities</strong></td>
</tr>
<tr>
<td>decentralised data management</td>
</tr>
<tr>
<td>products not projects</td>
</tr>
<tr>
<td>decentralised governance</td>
</tr>
<tr>
<td>smart endpoints and dumb pipes</td>
</tr>
<tr>
<td>evolutionary design</td>
</tr>
<tr>
<td>infrastructure automation</td>
</tr>
<tr>
<td>designed for failure</td>
</tr>
</tbody>
</table>
characteristics of microservices

<table>
<thead>
<tr>
<th>Componentisation via services</th>
</tr>
</thead>
<tbody>
<tr>
<td>organised around business capabilities</td>
</tr>
<tr>
<td>decentralised data management</td>
</tr>
<tr>
<td>products not projects</td>
</tr>
<tr>
<td>decentralised governance</td>
</tr>
<tr>
<td>smart endpoints and dumb pipes</td>
</tr>
<tr>
<td>evolutionary design</td>
</tr>
<tr>
<td>infrastructure automation</td>
</tr>
<tr>
<td>designed for failure</td>
</tr>
</tbody>
</table>
The Web

http://www.flickr.com/photos/photophile/4527076709/
“be of the web, not behind the web”

Ian Robinson, author, REST in Practice
characteristics of microservices

- componentisation via services
- organised around business capabilities
- decentralised data management
- products not projects
- decentralised governance
- smart endpoints and dumb pipes
- evolutionary design
- infrastructure automation
- designed for failure
<table>
<thead>
<tr>
<th>characteristics of microservices</th>
</tr>
</thead>
<tbody>
<tr>
<td>componentisation via services</td>
</tr>
<tr>
<td>organised around business capabilities</td>
</tr>
<tr>
<td>decentralised data management</td>
</tr>
<tr>
<td>products not projects</td>
</tr>
<tr>
<td>decentralised governance</td>
</tr>
<tr>
<td>smart endpoints and dumb pipes</td>
</tr>
<tr>
<td>evolutionary design</td>
</tr>
<tr>
<td>infrastructure automation</td>
</tr>
<tr>
<td>designed for failure</td>
</tr>
</tbody>
</table>
insurance company
insurance company
separate lines of business

- home
- motor
- life
separate lines of business

home

motor

life

and cross-cutting capabilities

my account
cross-functional teams delivering lines of business
characteristics of microservices

- componentisation via services
- organised around business capabilities
- decentralised data management
- products not projects
- decentralised governance
- smart endpoints and dumb pipes
- evolutionary design
- infrastructure automation
- designed for failure
<table>
<thead>
<tr>
<th>characteristics of microservices</th>
</tr>
</thead>
<tbody>
<tr>
<td>componentisation via services</td>
</tr>
<tr>
<td>organised around business capabilities</td>
</tr>
<tr>
<td>decentralised data management</td>
</tr>
<tr>
<td>products not projects</td>
</tr>
<tr>
<td>decentralised governance</td>
</tr>
<tr>
<td>smart endpoints and dumb pipes</td>
</tr>
<tr>
<td>evolutionary design</td>
</tr>
<tr>
<td>infrastructure automation</td>
</tr>
<tr>
<td>designed for failure</td>
</tr>
</tbody>
</table>
military
Share tools, don’t enforce standards

MAKE IT EASY TO DO THE RIGHT THING
<table>
<thead>
<tr>
<th>Category</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aminator</td>
<td><img src="https://example.com/aminator.jpg" alt="Aminator" /></td>
</tr>
<tr>
<td>Archaius</td>
<td><img src="https://example.com/archaius.jpg" alt="Archaius" /></td>
</tr>
<tr>
<td>Asgard</td>
<td><img src="https://example.com/sgard.jpg" alt="Asgard" /></td>
</tr>
<tr>
<td>Astyanax</td>
<td><img src="https://example.com/astyanax.jpg" alt="Astyanax" /></td>
</tr>
<tr>
<td>Blitz4j</td>
<td><img src="https://example.com/blitz4j.jpg" alt="Blitz4j" /></td>
</tr>
<tr>
<td>Brutal</td>
<td><img src="https://example.com/brutal.jpg" alt="Brutal" /></td>
</tr>
<tr>
<td>COMEDY</td>
<td><img src="https://example.com/comedy.jpg" alt="COMEDY" /></td>
</tr>
<tr>
<td>Cloud-Prize</td>
<td><img src="https://example.com/cloud-prize.jpg" alt="Cloud-Prize" /></td>
</tr>
<tr>
<td>Curator</td>
<td><img src="https://example.com/curator.jpg" alt="Curator" /></td>
</tr>
<tr>
<td>Denominator</td>
<td><img src="https://example.com/denominator.jpg" alt="Denominator" /></td>
</tr>
<tr>
<td>Edda</td>
<td><img src="https://example.com/edda.jpg" alt="Edda" /></td>
</tr>
<tr>
<td>Eureka</td>
<td><img src="https://example.com/eureka.jpg" alt="Eureka" /></td>
</tr>
<tr>
<td>EVCache</td>
<td><img src="https://example.com/evcache.jpg" alt="EVCache" /></td>
</tr>
<tr>
<td>Exhibitor</td>
<td><img src="https://example.com/exhibitor.jpg" alt="Exhibitor" /></td>
</tr>
<tr>
<td>Frigga</td>
<td><img src="https://example.com/frigga.jpg" alt="Frigga" /></td>
</tr>
<tr>
<td>Gcviz</td>
<td><img src="https://example.com/gcviz.jpg" alt="Gcviz" /></td>
</tr>
<tr>
<td>Governator</td>
<td><img src="https://example.com/governator.jpg" alt="Governator" /></td>
</tr>
<tr>
<td>Hystrix</td>
<td><img src="https://example.com/hystrix.jpg" alt="Hystrix" /></td>
</tr>
<tr>
<td>Karyon</td>
<td><img src="https://example.com/karyon.jpg" alt="Karyon" /></td>
</tr>
<tr>
<td>Netflix-Graph</td>
<td><img src="https://example.com/netflix-graph.jpg" alt="Netflix-Graph" /></td>
</tr>
<tr>
<td>NfWebCrypto</td>
<td><img src="https://example.com/nfwebcrypto.jpg" alt="NfWebCrypto" /></td>
</tr>
</tbody>
</table>
characteristics of microservices

-货物化服务
-围绕业务能力组织
-去中心化数据管理
-产品而非项目
-智能端点和数据管道
-去中心化治理
-演进设计
-基础设施自动化
-设计为了失败
-微服务的特性
The characteristics of microservices include:

- Componentisation via services
- Organised around business capabilities
- **Decentralised data management**
- Products not projects
- Decentralised governance
- Smart endpoints and dumb pipes
- Evolutionary design
- Infrastructure automation
- Designed for failure
monolith - single database

microservices - application databases
Capabilities own their own data
(can be cached elsewhere with appropriate policies)
what about transactions in this model?
How to beat the CAP theorem

The CAP theorem states a database cannot guarantee consistency, availability, and partition-tolerance at the same time. But you can’t sacrifice partition-tolerance (see here and here), so you must make a tradeoff between availability and consistency. Managing this tradeoff is a central focus of the NoSQL movement.

Consistency means that after you do a successful write, future reads will always take that write into account. Availability means that you can always read and write to the system. During a partition, you can only have one of these properties.

Systems that choose consistency over availability have to deal with some awkward issues. What do you do when the database isn’t available? You can try buffering writes for later, but you risk losing those writes if you lose the machine with the writes.
"As bad as anything else"

Beating the CAP Theorem Checklist

Your ( ) tweet ( ) blog post ( ) marketing material ( ) online comment advocates a way to beat the CAP theorem. Your idea will not work. Here is why it won't work:

( ) you are assuming that software/network/hardware failures will not happen
( ) you pushed the actual problem to another layer of the system
( ) your solution is equivalent to an existing one that doesn't beat CAP
( ) you're actually building an AP system
( ) you're actually building a CP system
( ) you are not, in fact, designing a distributed system

Specifically, your plan fails to account for:

( ) latency is a thing that exists
( ) high latency is indistinguishable from splits or unavailability
( ) network topology changes over time
( ) there might be more than 1 partition at the same time
( ) split nodes can vanish forever
( ) a split node cannot be differentiated from a crashed one by its peers
( ) clients are also part of the distributed system

http://ferd.ca/beating-the-cap-theorem-checklist.html
characteristics of microservices

- componentisation via services
- organised around business capabilities
- decentralised data management
- products not projects
- decentralised governance
- smart endpoints and dumb pipes
- evolutionary design
- infrastructure automation
- designed for failure
characteristics of microservices

- componentisation via services
- organised around business capabilities
- decentralised data management
- products not projects
- decentralised governance
- smart endpoints and dumb pipes
- evolutionary design
- infrastructure automation
- designed for failure
AUTOMATE

ALL THE THINGS!
compile, unit and functional test  
acceptance test  
inTEGRATION test  
user acceptance test  
performance test  
run on build machine  
deployed on build machine  
deployed to integration environment  
deployed to UAT environment  
deployed to performance environment  
deploy to production  
Fast Feedback  
More Confidence
phoenix infrastructure
characteristics of microservices

- Componentisation via services
- Organised around business capabilities
- Decentralised data management
- Products not projects
- Decentralised governance
- Smart endpoints and dumb pipes
- Evolutionary design
- Infrastructure automation
- Designed for failure
“it pushes the accidental complexity into the infrastructure”

Martin Fowler
“Every socket, process, pipe, or remote procedure call can and will hang. Even database calls [...]”

M. Nygard, “Release It”
Chapter 3

Some (un)expected consequences
microservices **should** allow us to go as **“fast as possible”**

*be cheap to replace*

*be deployable on demand*

*be resilient on imperfect networks*

*but it’s not as simple as that*
Monitoring    Organisational Structure

Deployment    Integration

Testing    Architectural Safety
How **big** are they?

*How many can you support?*
How big are they?

How many can you support?
Consider a single application - it's a website, let's call it A.
we want to get A into production and since we are hipsters we are going to practice continuous delivery - we will have a full automated build pipeline
we want to get A into production and since we are hipsters we are going to practice continuous delivery - we will have a full automated build pipeline

Tappety tap

<table>
<thead>
<tr>
<th>Compile, unit and functional test</th>
<th>Acceptance test</th>
<th>Integration test</th>
<th>Deploy to production</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run on build machine</td>
<td>Deployed on build machine</td>
<td>Deployed to integration environment</td>
<td></td>
</tr>
</tbody>
</table>
we want to get A into production and since we are hipsters we are going to practice continuous delivery - we will have a full automated build pipeline

Tappety tap
How many environments do we need?
How many environments do we need?

- Compile, unit and functional test
- Acceptance test
- Integration test
- Deploy to production
- Run on build machine
- Deployed on build machine
- Deployed to integration environment
OK, so we are going to be cool and use microservices

A

B
and we might as well call them something interesting

webapp

customers
and they have a dependency on one another...

webapp  customers
How do we traditionally make sure that new versions of the services work with each other?

Let me illustrate this
git push origin master
git push origin master
git push origin master
git push origin master
git push origin master
git push origin master
What should V2 of the *blue* app be tested against here
This is in production, so presumably we should test against this?
git push origin master
git push origin master

V2

V1

V2
git push origin master

git push origin master
git push origin master

V2

V1

V2

V1
git push origin master
git push origin master

git push origin master
I’m sorry Dave, I can’t let you do that
Locks == Delay
2 services

\[ \Rightarrow \]

4 environments
2 services
2 services

4
2 services

4

>600
the death of the integration environment
1972 - Dennis Ritchie invents a powerful gun that shoots both forward and backward simultaneously. Not satisfied with the number of deaths and permanent maimings from that invention he invents C and Unix.

A **capability** is a combination of people, processes, systems that provides value to customers (internal or external)

The **what** of the business, not the **how**
The problem with projects

delivering stuff

selling stuff
The problem with projects

delivering stuff

selling stuff
The problem with projects

delivering stuff

selling stuff

146
The problem with projects
The problem with projects

delivering stuff

selling stuff
If you aren’t *really* careful with your API design

your beautiful microservices end up in a tangled, coupled mess
6 week periods of “hardening”

Feature starts dev

iteration 1
iteration 2
iteration 3

regression testing
performance testing
deployment tests

Feature deployed

cycle time
Without deploying into production, inventory is built up - inventory costs money and the more we have the more risky our deployments.
beware the distributed monolith!
WHAT DO YOU DO?
I MAKE TOOLS THAT MAKE TOOLS

...THAT MONITOR CODE THAT DEPLOYS TOOLS THAT BUILD TOOLS FOR DEPLOYING MONITORS...

20 MINUTES LATER...
...FOR MONITORING DEPLOYMENT OF TOOLS FOR...
BUT WHAT'S IT ALL FOR?

Honestly, no idea. Porn, probably.

https://xkcd.com/1629/
TERRY PRATCHETT
A SLIP OF THE KEYBOARD
COLLECTED NONFICTION
TERRY PRATCHETT
A SLIP OF THE KEYBOARD
COLLECTED NONFICTION
The Boojam!

Large organisations tend to be “functionally split” for efficiency reasons

Many use scrummerfall or project based teams

Throw away integration testing? Are you made?

We suddenly need a whole new set of skills

Design for failure

Architectural safety

Zookeeper?!!
Chapter 4

Facing these consequences
It is not possible to get the benefits of microservices without serious organisational change.
“...organizations which design systems ... are constrained to produce designs which are copies of the communication structure of those organizations”

Melvyn Conway, 1968
The mirroring phenomenon is consistent with two rival causal mechanisms. First, designs may evolve to reflect their development environments. In **tightly-coupled organizations**, dedicated teams employed by a single firm and located at a single site develop the design. Problems are solved by face-to-face interaction, and performance “tweaked” by taking advantage of the access that module developers have to information and solutions developed in other modules. **Even if not an explicit managerial choice, the design naturally becomes more tightly-coupled.**

By contrast, in **loosely-coupled organizations**, a large, distributed team of volunteers develops the design. Face-to-face communications are rare given most developers never meet. Hence fewer connections between modules are established. **The architecture that evolves is more modular** as a result of the limitations on communication between developers.

"Exploring the Duality between Product and Organizational Architectures : A Test of the "Mirroring" Hypothesis"
http://www.hbs.edu/faculty/Publication%20Files/08-039_1861e507-1dc1-4602-85b8-90d71559d85b.pdf
loosely-coupled organizations ⇒ the design becomes more tightly-coupled.

tightly-coupled organizations ⇒ the architecture is more modular
Lines of business
Lines of business
Value streams
Value streams
teams
each team owns one or more services
~10-20
~10-20
~160-200
~10-20 multiples thereof
~160-200
The effect of distance on communication
co-locate as much as possible

take advantage of serendipitous conversations
Good Monitoring
Good Monitoring

Fast Remediation
Test in production

Good Monitoring

Fast Remediation

QA
SEMANTIC MONITORING
Web Shop

Customer Service

Expectations
Consumer Driven Contracts
production != live
blue / green deploys

canary releases

infrastructure as code
My hypothesis is that you can use organisational boundaries to reason about which testing patterns to apply and which integration patterns to use.
The “chunking up from microservices to teams to value streams to lines of business to organisations” practice onion*

*I might need a better name for this
between organisational boundaries

typically requires:

Low change rate
High stability

Semantic Versioning
Tolerant Reader
between business capabilities

Higher change rate
Lower stability

Semantic Versioning
Contract Testing
Tolerant Reader
between teams

Higher rate of change
Lower stability

Semantic Versioning
Contract Testing
Tolerant Reader
within teams

Highest rate of change
Lower stability

Conversational change
Tolerant Reader
FINAL THOUGHTS
1. Rule of **Modularity**: Write simple parts connected by clean interfaces.
2. Rule of **Clarity**: Clarity is better than cleverness.
3. Rule of **Composition**: Design programs to be connected to other programs.
4. Rule of **Separation**: Separate policy from mechanism; separate interfaces from engines.
5. Rule of **Simplicity**: Design for simplicity; add complexity only where you must.
6. Rule of **Parsimony**: Write a big program only when it is clear by demonstration that nothing else will do.
7. Rule of **Transparency**: Design for visibility to make inspection and debugging easier.
8. Rule of **Robustness**: Robustness is the child of transparency and simplicity.
9. Rule of **Representation**: Fold knowledge into data so program logic can be stupid and robust.
10. Rule of **Least Surprise**: In interface design, always do the least surprising thing.
11. Rule of **Silence**: When a program has nothing surprising to say, it should say nothing.
12. Rule of **Repair**: When you must fail, fail noisily and as soon as possible.
13. Rule of **Economy**: Programmer time is expensive; conserve it in preference to machine time.
14. Rule of **Generation**: Avoid hand-hacking; write programs to write programs when you can.
15. Rule of **Optimization**: Prototype before polishing. Get it working before you optimize it.
16. Rule of **Diversity**: Distrust all claims for “one true way”.
17. Rule of **Extensibility**: Design for the future, because it will be here sooner than you think.
the 17 rules of UNIX programming

1. Rule of **Modularity**: Write simple parts connected by clean interfaces.
2. Rule of **Clarity**: Clarity is better than cleverness.
3. Rule of **Composition**: Design programs to be connected to other programs.
4. Rule of **Separation**: Separate policy from mechanism; separate interfaces from engines.
5. Rule of **Simplicity**: Design for simplicity; add complexity only where you must.
6. Rule of **Parsimony**: Write a big program only when it is clear by demonstration that nothing else will do.
7. Rule of **Transparency**: Design for visibility to make inspection and debugging easier.
8. Rule of **Robustness**: Robustness is the child of transparency and simplicity.
9. Rule of **Representation**: Fold knowledge into data so program logic can be stupid and robust.
10. Rule of **Least Surprise**: In interface design, always do the least surprising thing.
11. Rule of **Silence**: When a program has nothing surprising to say, it should say nothing.
12. Rule of **Repair**: When you must fail, fail noisily and as soon as possible.
13. Rule of **Economy**: Programmer time is expensive; conserve it in preference to machine time.
14. Rule of **Generation**: Avoid hand-hacking; write programs to write programs when you can.
15. Rule of **Optimization**: Prototype before polishing. Get it working before you optimize it.
16. Rule of **Diversity**: Distrust all claims for “one true way”.
17. Rule of **Extensibility**: Design for the future, because it will be here sooner than you think.
never done
“This, milord, is my family's axe. We have owned it for almost nine hundred years, see. Of course, sometimes it needed a new blade. And sometimes it has required a new handle, new designs on the metalwork, a little refreshing of the ornamentation . . . but is this not the nine hundred-year-old axe of my family? And because it has changed gently over time, it is still a pretty good axe, y’know. Pretty good.”
The 16th rule of Unix programming

The Rule of Diversity

DISTRUST ALL CLAIMS FOR “ONE TRUE WAY”
Thanks!