Electric Cloud

Continuous Delivery of Microservices: Patterns and Processes

Anders Wallgren
CTO, Electric Cloud
@anders_wallgren

Avan Mathur Product
Manager, Electric Cloud
@Avantika_ec

What are Microservices?

- A pattern for building distributed systems:
  - A suite of services, each running in its own process, each exposing an API
  - Independently developed
  - Independently deployable
  - Each service is focused on doing one thing well

"Gather together those things that change for the same reason, and separate those things that change for different reasons."

– Robert Martin

What's cool about Microservices?

- Divide and conquer complex distributed applications
- Loose coupling, so each service can:
  - choose the tooling that's appropriate for the problem it solves
  - can be scaled as appropriate, independent of other services
  - can have its own lifecycle independent of other services
- Makes it easier to adopt new technologies
- Smaller more autonomous teams are more productive
- Better resource utilization

Isn’t this just SOA warmed over?

- SOA was also meant to solve the monolithic problem
- No consensus evolved on how to do SOA well (or even what SOA is)
- Became more about selling middleware than solving the problem
- Tends to mandate technology stacks
- Doesn’t address how to break down monoliths beyond “use my product to do it”

Microservices evolved out of real world problem solving

Can’t I just modularize and use shared libraries?

- Technological coupling
- No longer free to use the tools that are fit for purpose
- Generally not able to deploy a new version without deploying everything else as well
- Makes it much easier to introduce API coupling – process boundaries enforce good API hygiene

Code reuse is a good thing, but it’s not the best basis for a distributed architecture

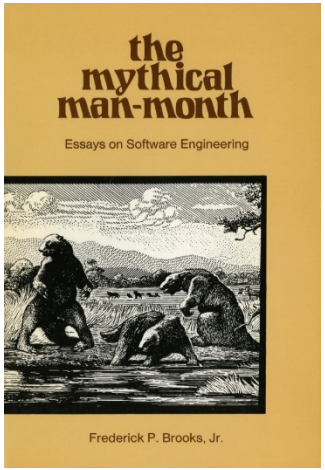
What's good/bad about monolithic apps?

- Can be easier to test
- Can be easier to develop
- Can be easier to deploy

- Can't deploy anything until you deploy everything
- Harder to learn and understand the code
- Easier to produce spaghetti code
- Hard to adopt new technologies
- You have to scale everything to scale anything

So...Was Fred Brooks Wrong?

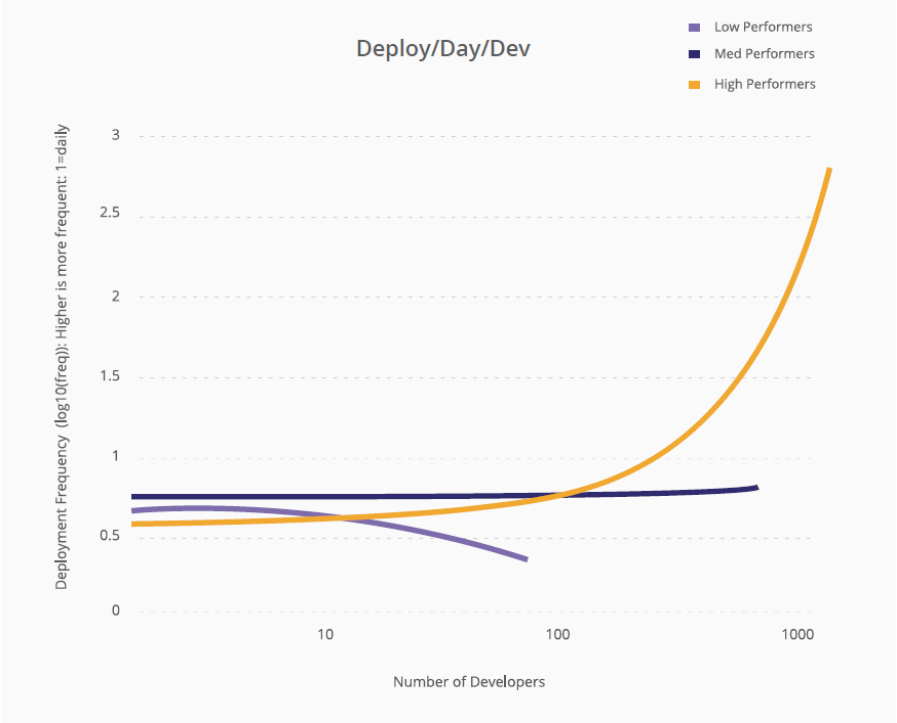
Emphatically, no



But!

A properly constructed microservices architecture makes it vastly easier to scale teams and scale applications

Number of deployments per day per developer



https://puppetlabs.com/2015-devops-report-ppc

Electric Cloud

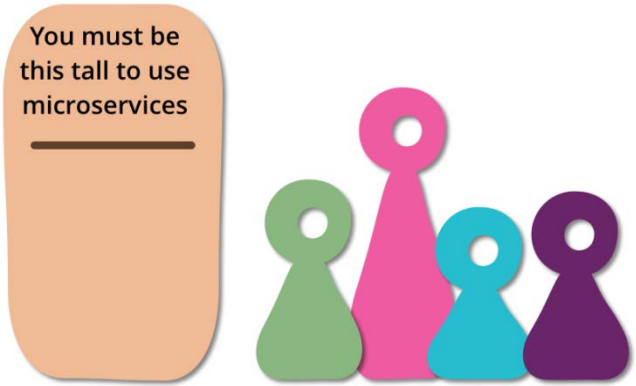
Should I use Microservices?

- If you already have solid CI, automated testing, and automated deployment, and you're looking to scale, then maybe
- If you don't have automated testing, then you should ~~probably~~ definitely worry about that first
- You have to be (or become) very good at automated deployment, testing and monitoring to reap the benefits.

Microservices are not a magic hammer that will make your other problems go away

Am I ready for microservices?

- If you're just starting out, stay monolithic until you understand the problem better
- You need to be good at infrastructure provisioning
- You need to be good at rapid application deployment
- You need to be good at monitoring
- You need to have good domain/system comprehension



http://martinfowler.com/bliki/MicroservicePrerequisites.html

What's difficult about Microservices?

- Distributed Systems Are Hard
  - Service composition is tricky to get right, can be expensive to change
  - Inter-process failure modes have to be accounted for
  - Abstractions look good on paper but beware of bottlenecks
  - Service discovery
- State management – transactions, caching, and other fun things
- *Team-per-service* or *Silo-per-service*? + Conway's Law
- Legacy apps: Rewrite? Ignore? Hybrid?
- Good system comprehension is key
- Your service might be small, but how large is its deployment footprint?

(Some) Microservices Best Practices

What makes a good micro service?

- Loose coupling
  - A change to service A shouldn't require a change in service B
  - Small, tightly focused API
- High cohesion
  - Each service should have a specific responsibility
  - Domain-specific behavior should be in one place
  - If you need to change a behavior, you shouldn't have to change multiple services

What size should my services be?

- The smaller the services, the more benefit you get from decoupling
- You should be able to (re-)rewrite one in a “small” number of weeks
- If you can’t make a change to a service and deploy it without changing other things, then it’s too large
- The smaller the service, the more moving parts you have, so you have to be ready for that, operationally

Electric Cloud

Testing

- If you do lots of manual testing address that first
- Unit testing and service-level testing (with other services stubbed or mocked)
- End-to-end testing is more difficult with microservices (and tells you less about what broke)
- Unit tests >> service tests >> end-to-end tests
- Use mocking to make sure side-effects happen as expected
- Consider using a single pipeline for end-to-end tests
- Performance testing is more important than in a monolith
- As always, flaky tests are the devil

Environments & Deployment

- Keep your environments as close to production as is practical (Docker/Chef/Puppet, etc)
- One service per host
  - Minimize the impact of one service on others
  - Minimize the impact of a host outage
- Use VMs/containers to make your life easier
  - Containers map very well to microservices
  - “Immutable servers”
- PaaS solutions can be helpful, but can also constrain you

Automate all the things!

MTTR or MTBF?

- There is a point of diminishing returns with testing (especially end-to-end testing)
- You may be better off getting really good at remediating production problems
  - Monitoring
  - Very fast rollbacks
  - Blue/green deployments
  - Canary deployments
- Not all services have the same durability requirements

Breaking apart the monolith

- Do it incrementally, not as a big-bang rewrite. You're going to get it wrong the first time.
- Look for *seams* – areas of code that are independent, focused around a single business capability
- Domain-Driven Design and it's notion of Domain Contexts is a useful tool
- Look for areas of code that change a lot (or needs to change)
- Don't ignore organizational structure (Conway's Law)
- Dependency analysis tools can help, but are no panacea

Breaking apart the monolith - Data

- RDBMS may well be your largest source of coupling
- Understand your schema
  - Foreign key constraints
  - Shared mutable data
  - Transactional boundaries.
- Is eventual consistency OK?
  - Avoid distributed transactions if possible
- Split data before you split code
- Do you need an RDBMS at all or can you use NoSQL?

Things to look out for

- It isn't necessarily *easier* to do it this way…
- Your services *will* evolve over time – you'll split services, perhaps merge them, etc. Just accept that.
- You need to be rigorous in handling failures (consider using, e.g. Hystrix from Netflix to bake in better resiliency)
- Reporting will need to change – you probably won't have all the data in a single place (or even a single technology)
- "The network is reliable" (and the rest of the 8 fallacies)
- Be careful about how you expose your data objects over the wire
- "But my service relies on version X of ServiceA and now I'm down"

Things to Think About

- Consistent logging & monitoring output across services
- Avoid premature decomposition
  - If starting from scratch, stay monolithic, keep it modular and split things up as your understanding of the problem evolves
- Consider event-based techniques to decrease coupling further
- Postel's Law: "*Be conservative in what you do, be liberal in what you accept from others*"

Monitoring Best Practices for Microservices

Electric Cloud

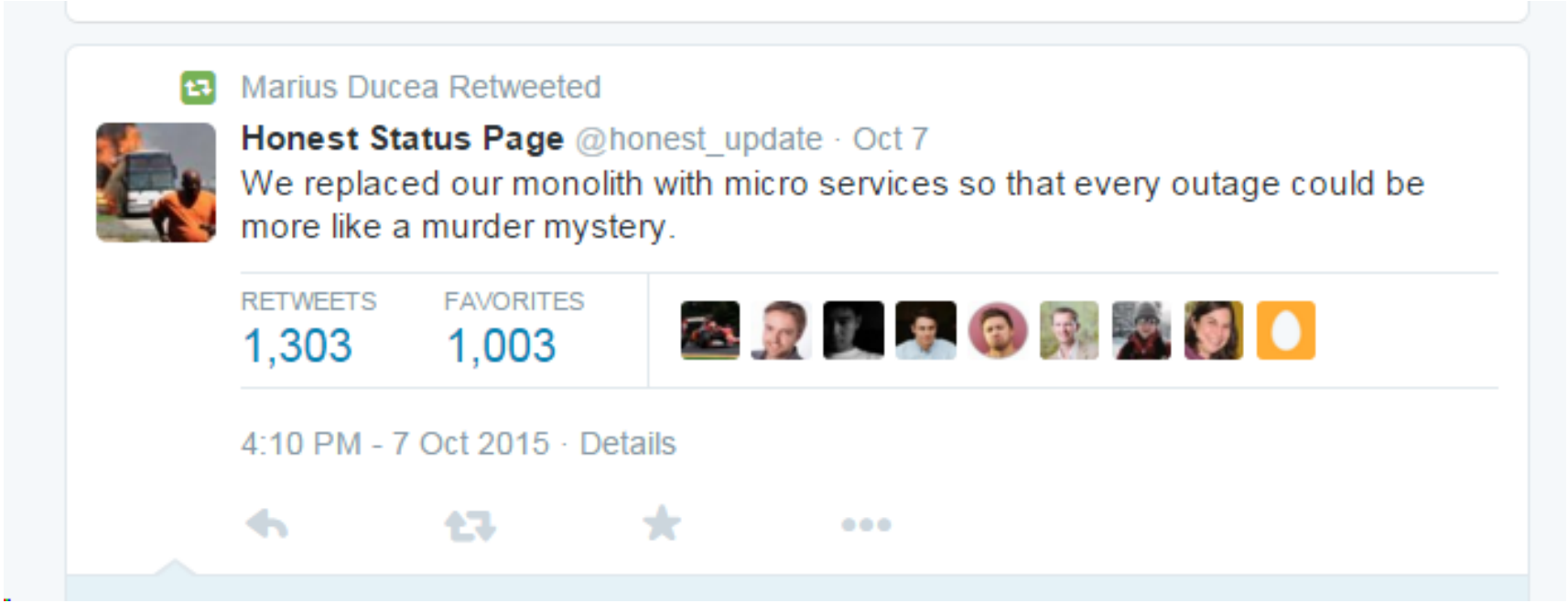
https://neo4j.com/blog/managing-microservices-neo4j/

Electric Cloud

How does monitoring change?

- Monitoring a monolith is easier than microservices since you really only have one thing that can break…
- With a large number of services, tracking the root cause of a failure can be challenging
- Satisfying an end-user request can touch dozens of services

The Importance of Monitoring



Electric Cloud

Monitoring Best Practices

- All services should log and emit monitoring data in a consistent fashion (even if using different stacks)
- Monitor latency and response times between services
- Monitor the host (CPU, memory, etc)
- Aggregate monitoring and log data into a single place
- Log early, log often
- Understand what a well-behaving service looks like, so you can tell when it goes wonky
- Use techniques like correlation ids to track requests through the system
  - “So then requestId 0xf00dfee8 in the log on ms-app-642-prod becomes messageId 1125f34c-e34e-11e2-a70f-5c260a4fa0c9 on ms-route-669-prod?”

Software Pipeline Best Practices for Microservices

Electric Cloud

Best Practices for CD Pipelines of Microservices-based Apps

- Your Automated Software Pipeline Is Your Friend™
  - Ideally, one platform handles all your software delivery
  - How's your test coverage?
  - Are your tests automated? Really automated?
- Self-service automation/ChatOps approaches
  - Reduce onboarding time, waiting, complexity
- Your solution should provide a real-time view of all the pipelines' statuses and any dependencies or exceptions.
- Make sure your deployment pipeline plugs into your monitoring so that alerts can trigger automatic processes such as rolling back a service, switching between blue/green deployments, scaling and so on.

Best Practices for CD Pipelines of Microservices-based Apps

- One repository per service
- Independent CI and Deployment pipelines per service
- “Automate all the things”: plug in all your toolchain to orchestrate the entire pipeline (CI, testing, configuration, infrastructure provisioning, deployments, application release processes, and production feedback loops.)
- Your pipeline must be tools/environment agnostic to support each team’s workflow and tool chain
- Test automation tools and service virtualization are critical

Best Practices for CD Pipelines of Microservices-based Apps

- Track artifacts through the pipeline (who checked-in the code, what tests were run, pass/fail results, on which environment it was deployed, which configuration was used, who approved it and so on)
- Bake in compliance into the pipeline by binding certain security checks and acceptance tests
- Allow for both automatic and manual approval gates
- Create reusable models/processes/automation for your various pipelines

Why Microservices in Containers?

- 2002: One service per metal box
  - “I remember my first dual-core box, too!”
  - “Why is that 32-core server idle all the time? Can I have it?”
- 2007: Hypervisor + 1 VM + Multiple services in that VM
  - “Yeah, can’t run ServiceA and ServiceB side by side, conflicting versions of…”
  - “Yeah, we did that until ServiceC filled up /tmp and took down ServiceD"
  - “Yeah, we tend to run ServiceE by itself once we’re past QA”
- 2012: Hypervisor + Multiple VMs + 1 Service in each VM
  - “Yeah, each VM OS has a copy of that in memory, so…”
- 2013: Containers: run multiple services in isolation without the OS overhead

Resources

- http://www.infoq.com/presentations/Breaking-the-Monolith
- http://martinfowler.com/tags/microservices.html
- http://www.amazon.com/Building-Microservices-Sam-Newman/dp/1491950358
- http://highscalability.com/blog/2014/7/28/the-great-microservices-vs-monolithic-apps-twitter-melee.html


Thank you!
Questions?