When should you use a Serverless Approach?

Thoughts from a *pragmatic* CTO about
when and how to use Serverless

Paul Johnston
CTO of Movivo

@PaulDJohnston on twitter and medium

(There will be time for questions later…)

Movivo

AWS Lambda

18 months

AWS Lambda

Not the only way of doing serverless

When should you use Serverless approach?

tl;dr Most of the time (99% probably)

What *is* the Serverless Approach?

FaaS* instead of "Servers"
Lots of Services
Event Driven
Distributed
Scalable
More Ops than Dev

* FaaS = Function as a Service

What *is* the Serverless Approach?

It’s the bleeding edge of
Cloud Native

(but I would say that)

Serverless

“Super Advanced Cloud”:
Functions
Events
Services

Quick mention: Serverless Framework

Not the only way of doing Serverless
but it might work for you.

When doing this talk, mainly talking about Events + Functions + Services, and AWS and not the framework

When should you use Serverless?

Well, let's get into the details of the approach…

…then answer that question

How we used to do it

Physical Servers
Virtual Servers
Instances
Managed instances
Containers

The “Server” Proposition

Servers are essentially still there
Frameworks
Considered a solved problem
15+ years of “progress”

The Server Problem

They are *your* servers
You need to maintain them
Almost invariably monolithic
Entrenched Thinking
Long term cost

FaaS, Infrastructure and Services

No longer your servers
More complex to manage
More reliant on third parties
Reduced Maintenance

But…

Serverless is still new
Some pioneers
Hesitancy in many
Patterns are still to emerge
Scale is easy and hard

CTOs like challenging this

CTOs are an interesting bunch
But they're usually the pioneers
and they often ask the best questions

Function as a Service (FaaS)

Very new idea (unless you're Simon Wardley or Google)
No simple analogue
"A bit like crack"
Still pioneering

Cost (mostly)
Decouples logic
Encourages distributed thinking
Errors are contained
Scale
No bulky "frameworks"

Events and Queues

Known but forgotten
Microservices
Feels like "backward step"
Difficult concept for developers

Efficient
Asynchronous by default
Hot swappable FaaS
Hardest part - most valuable

Data

RDBMS first (often)
Frameworks often
means ORMs (ugh)
Over complexity

Events force
distributed thinking
Data driven design
Store what you need
Optimise for write/read
Data at Rest/in motion

Infrastructure

“Servers”
Security patches
Maintenance
Testing environments
(containers)
DevOps

Infrastructure as Code
More moving parts
Harder to test
(currently)
Easier to train newbies
OPS(dev) not DevOps

Security

Constant challenge
"Server" approach known
Lots of tools
People cost

Provider patches servers
Smaller code base
Service provider may access data
DoS handled by provider

Deployment and CI/CD

Known tools
DevOps people
Few tools for serverless out there
Infrastructure concern often separated

"Roll your own"
More Ops than Dev
Staged deploys hard
Harder to canary, blue/green etc
Service Mesh? Event Routing?

Testing

Lots of tools on market
Solved problem?
Developers over rely on it?

Unit testing easy
Other testing different and easier?
Test Boundaries
Change
Need infrastructure as code to do it effectively

People

Developers often do frameworks
Several “go to” technologies

Smaller codebase
Easier to understand
Different skills
Easier to onboard
Productive fast

Vendor lock-in

Containers are easy to move
Servers are pretty much the same everywhere

It's service lock-in
But events allow you to switch
Choose providers more carefully
Analytics, logging etc best of breed

Maintenance

Servers
Patches
Security
Constant threats
Upgrades

Your service provider does the servers
You handle your code
Smaller codebase
Maintain Infrastructure as Code

Pros and Cons of Serverless

Pros:	Cons:
Maintenance	Testing
Scale	Infrastructure
Cost	3rd Party Services
Efficiencies	
Infrastructure	
Security	

So we know what Serverless is…

… but when should you use it?

Most of the time

It’s an appropriate solution for most of the client/server based solutions I’ve seen

In fact, I reckon 99% of solutions could move to this approach

But…

#1: Real time systems

Latency introduced through service usage
Cold-start can add some time (not much)

But most “real time” is not actually real time.
You can usually get away with request-response for most things

#2: Compute Intensive tasks

Serverless services limited by compute
Consider how service runs

Better to use an Instance
or a physical server for this

Or split into sequential/parallel tasks (Serverless)

#3: Very mission critical systems

(at least without thinking first)

You’re still working on someone else’s systems

What if they go down? (AWS + S3?)

It’s still your service

It is possible to consider failover but it’s hard

Caching and functionality at edge

#4: Where you need control

You lose control over things like:
Configuration (of systems)
Issue resolution (3rd parties fix it when it’s fixed)
Security (e,g. data for regulatory needs)

When should* you use Serverless approach?

tl;dr Most of the time (99% probably)

*or maybe it should be “could”

London - 7 July 2017 - St John's Hoxton

One Day. One Track. One Community.

A collaboration in conjunction with the Serverless London User Group, JeffConf is an attempt to move past the word Serverless, and focus on the use of these platforms and the value they provide.

jeffconf.com @jeffconf

Serverless community conference: London 7th July 2017

When should you use a Serverless Approach?

Thank you.
Any questions?

Paul Johnston
CTO of Movivo

@PaulDJohnston on twitter and medium