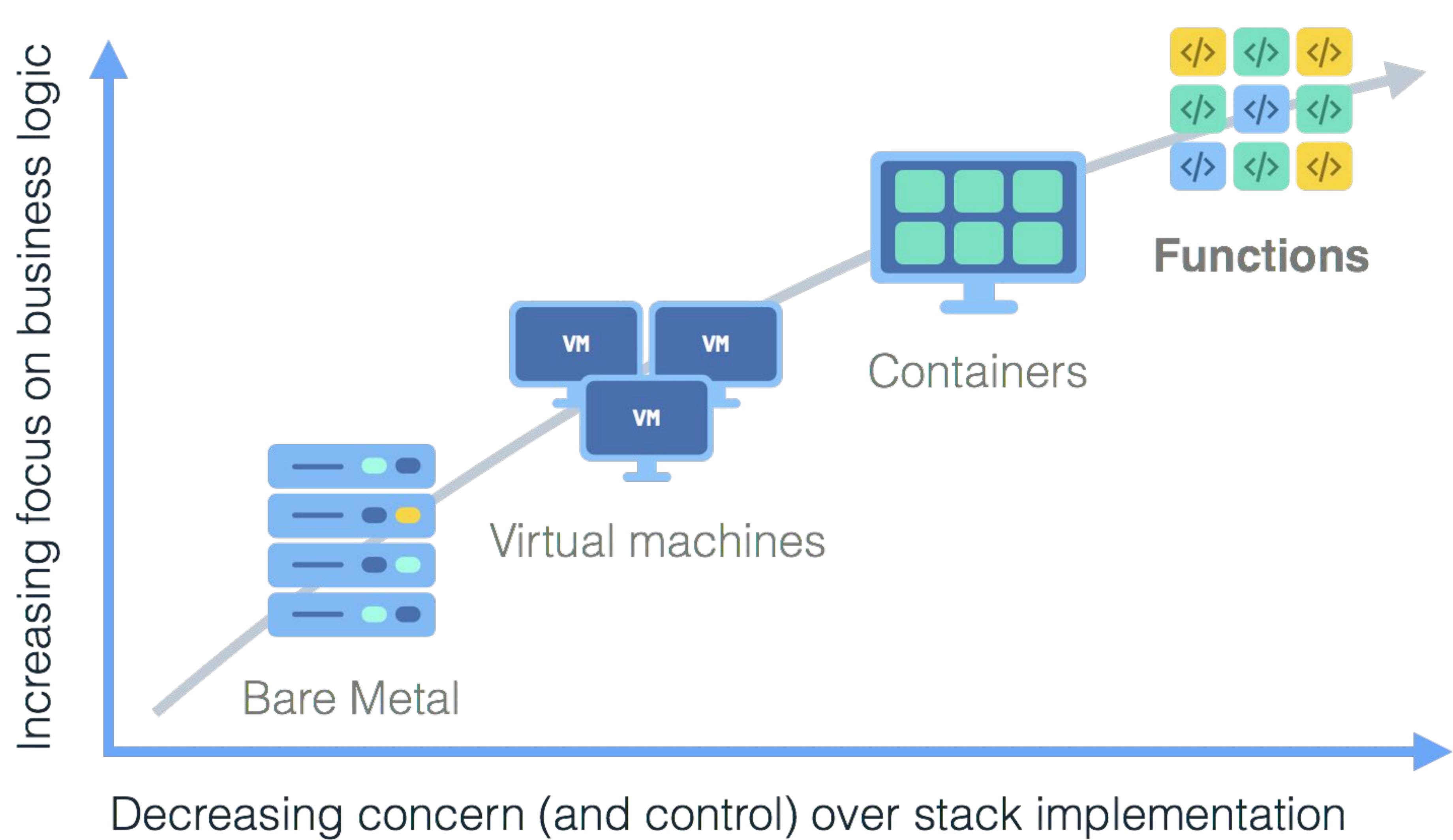
Serverless Computing

Function as a Service

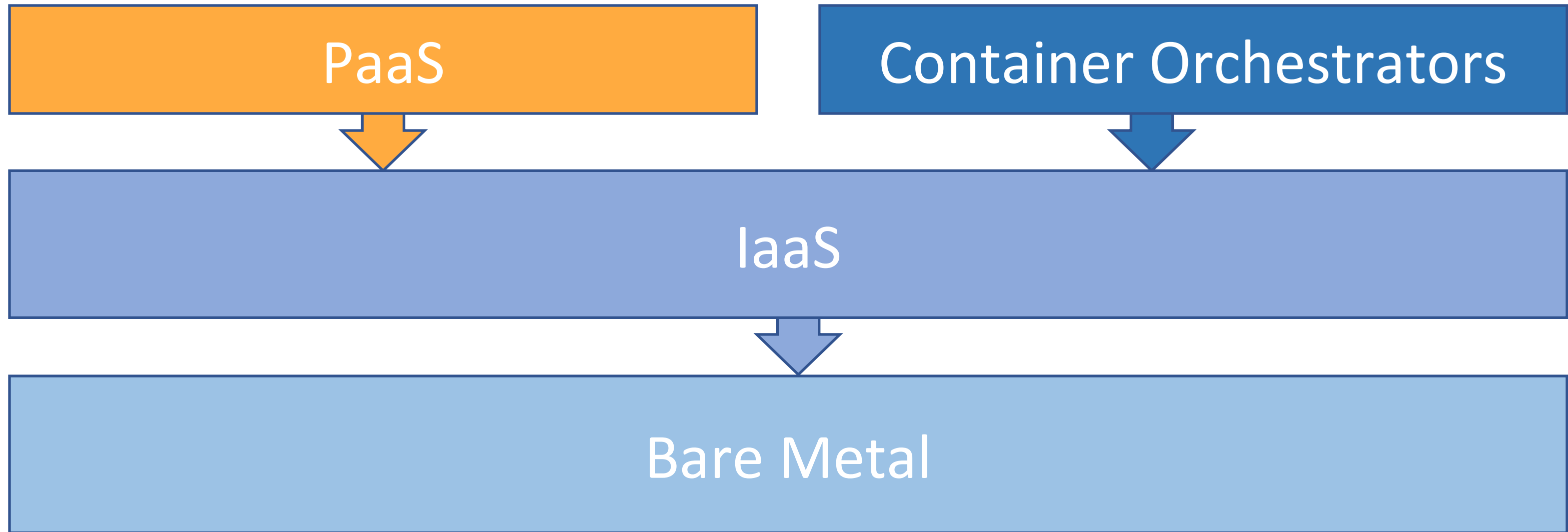
Paul Castro, Vatche Ishakian, Vinod Muthusamy and Aleksander Slominski

Outline

- Cloud Computing Evolution
- What is Serverless
- What makes Serverless attractive
  - Scalability
  - Management
  - Cost
- Type of applications for Serverless
- Current Platforms for Serverless
  - Lambda, Google Functions, OpenWhisk, OpenLambda, Functionless from Kubernetes
- Serverless Architecture (OpenWhisk)
  - From what is publically available
- Programming Model
  - Triggers, actions, rules, chains
- Research Challenges and Questions
- Hands-on exercises (second part)


Increasing focus on business logic
Functions
Containers
VM
VM
VM
Virtual machines
Bare Metal
Decreasing concern (and control) over stack implementation

Evolution Of Serverless

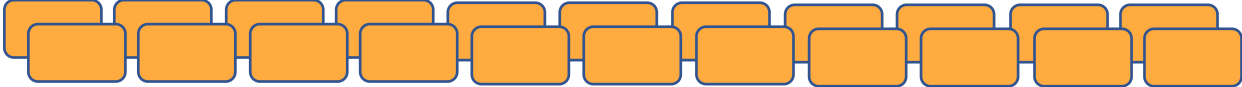


Monolithic Application

Break-down into microservices

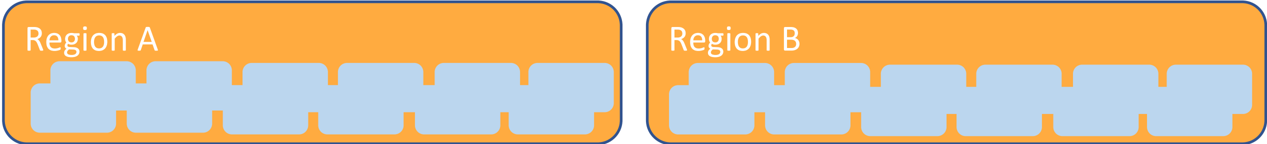
Explosion in number of containers / processes:

Make each micro service HA



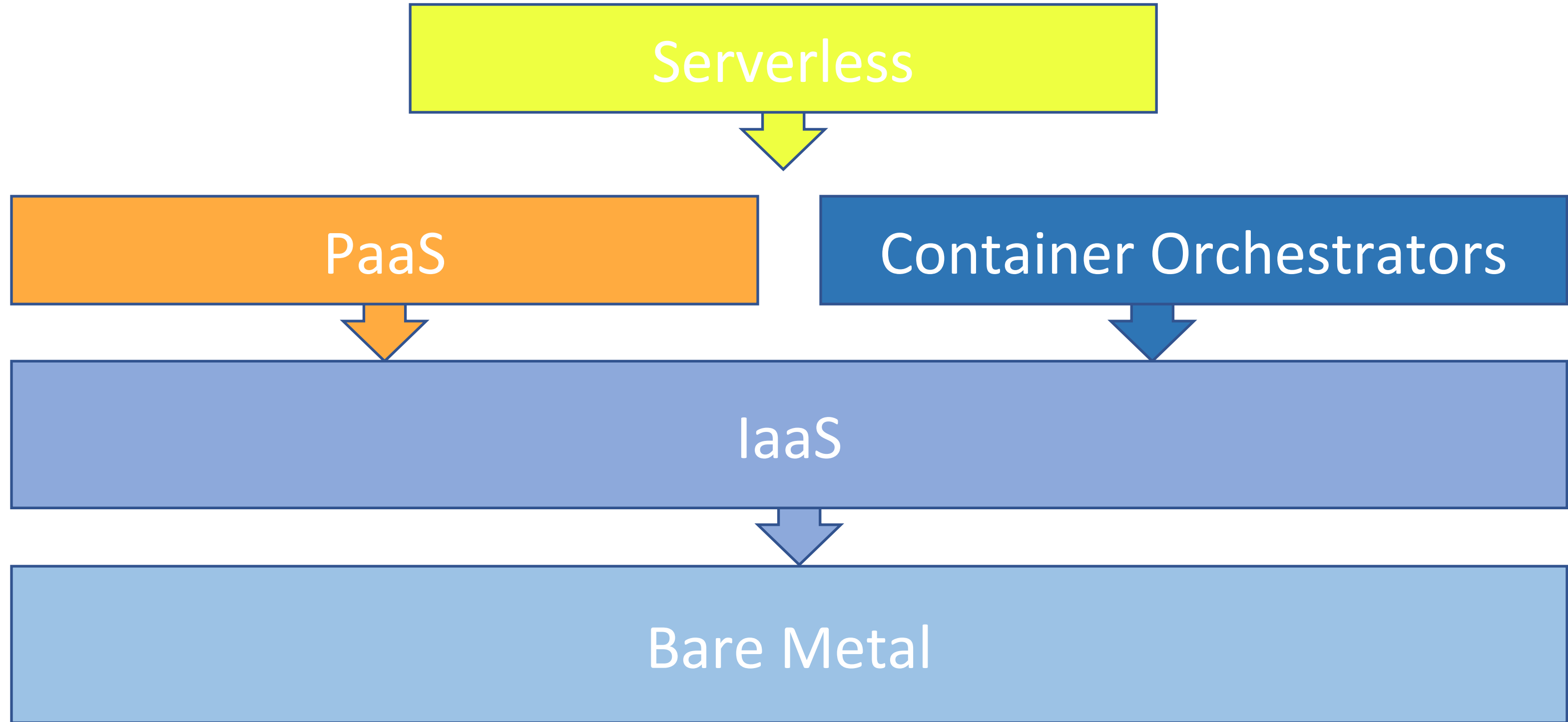
Increase of infrastructure cost footprint

Protect against regional outages



Increase of operational management cost and complexity

Enter Serverless



What is Serverless?

a cloud-native platform

for

short-running, stateless computation

and

event-driven applications

which

scales up and down instantly and automatically

and

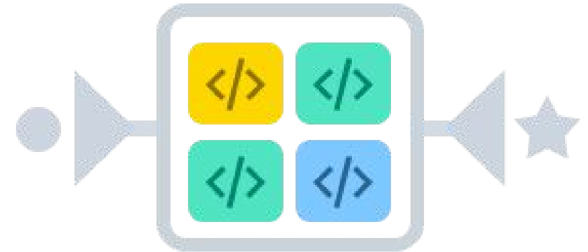
charges for actual usage at a millisecond granularity

Server-less means no servers?
Or worry-less about servers?

Runs code **only** on-demand on a per-request basis

Serverless deployment & operations model

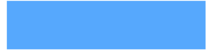
No servers



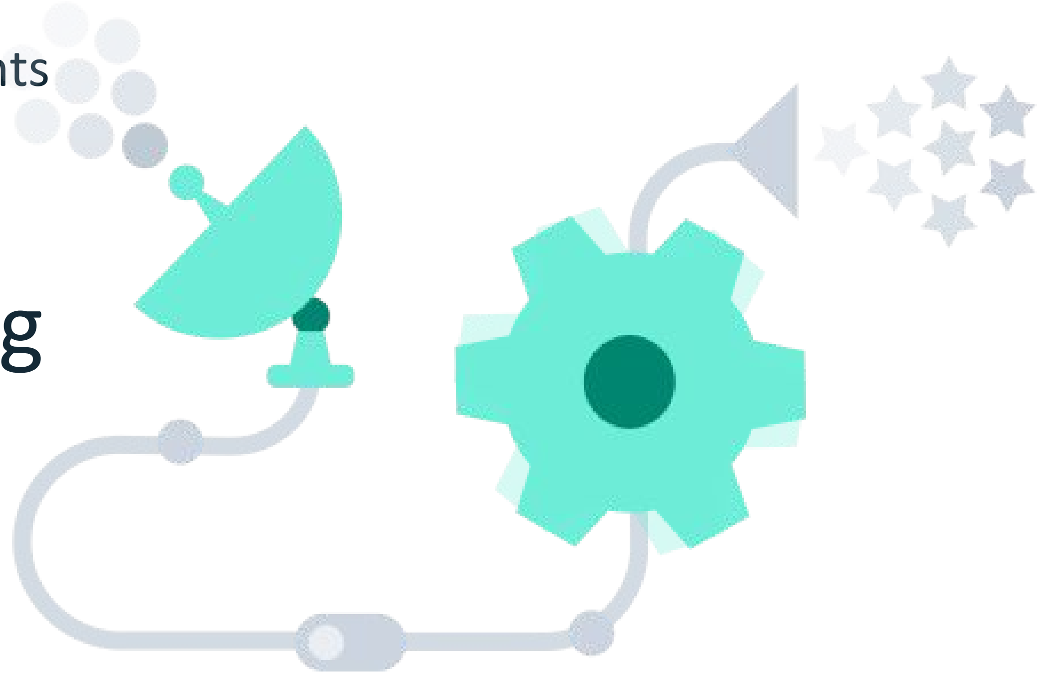
Just code

What triggers code execution?

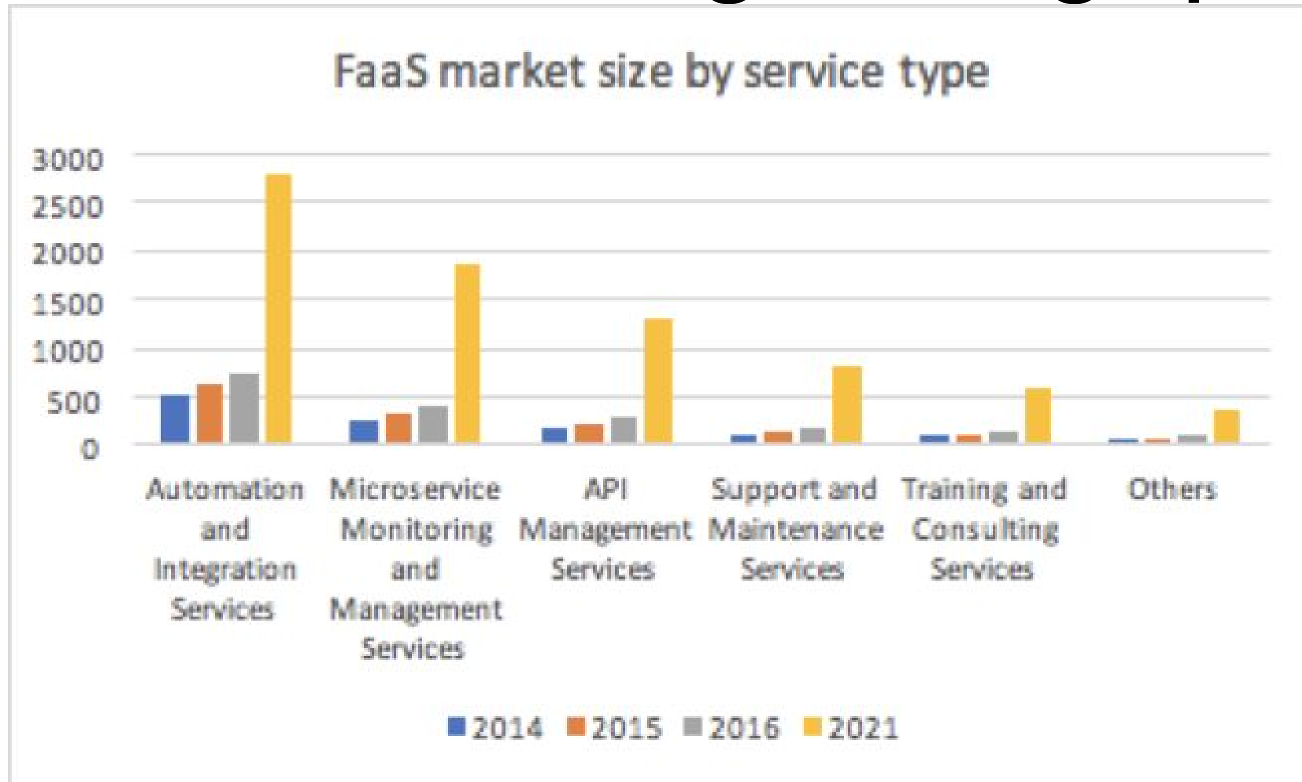
Runs code **in response** to events



Event-programming model

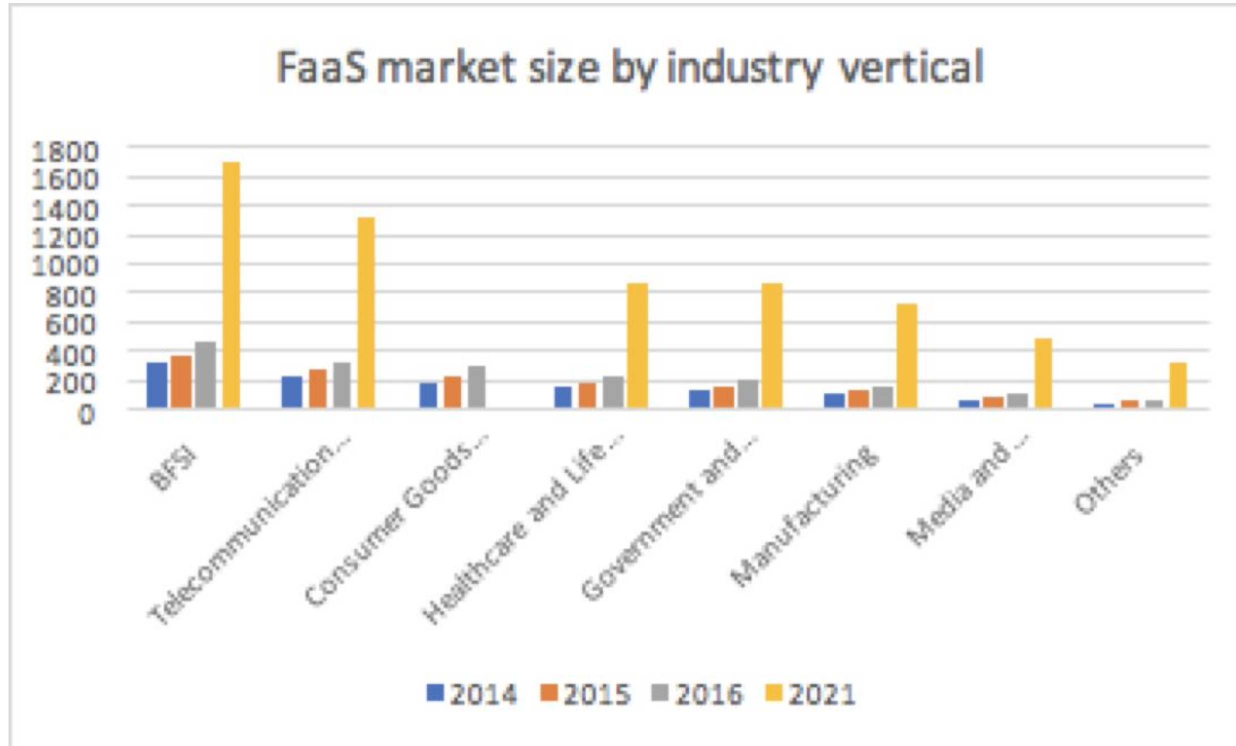


FaaS market is growing quickly



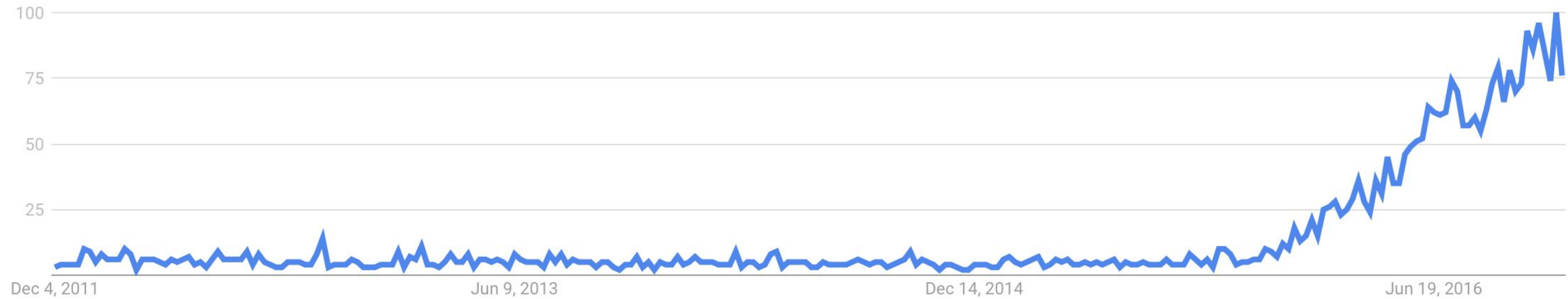
Source: FaaS Market - Global Forecast to 2021 - study by MarketsAndMarkets (http://www.marketsandmarkets.com/)

FaaS market is growing quickly



Source: FaaS Market - Global Forecast to 2021 - study by MarketsAndMarkets (http://www.marketsandmarkets.com/)

Google Search Trend over time



Why is Serverless attractive?

- Making app development & ops dramatically faster, cheaper, easier
- Drives infrastructure cost savings

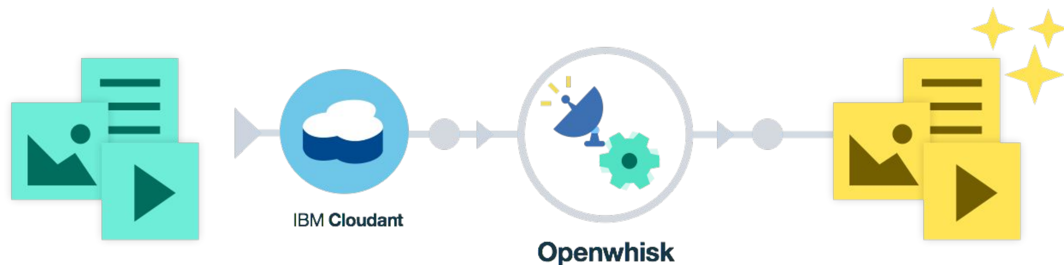
	On-prem	VMs	Containers	Serverless
Time to provision	Weeks-months	Minutes	Seconds-Minutes	Milliseconds
Utilization	Low	High	Higher	Highest
Charging granularity	CapEx	Hours	Minutes	Blocks of milliseconds

Source: Jason McGee, IBM; Serverless Conference 2017.

Key factors for infrastructure cost savings

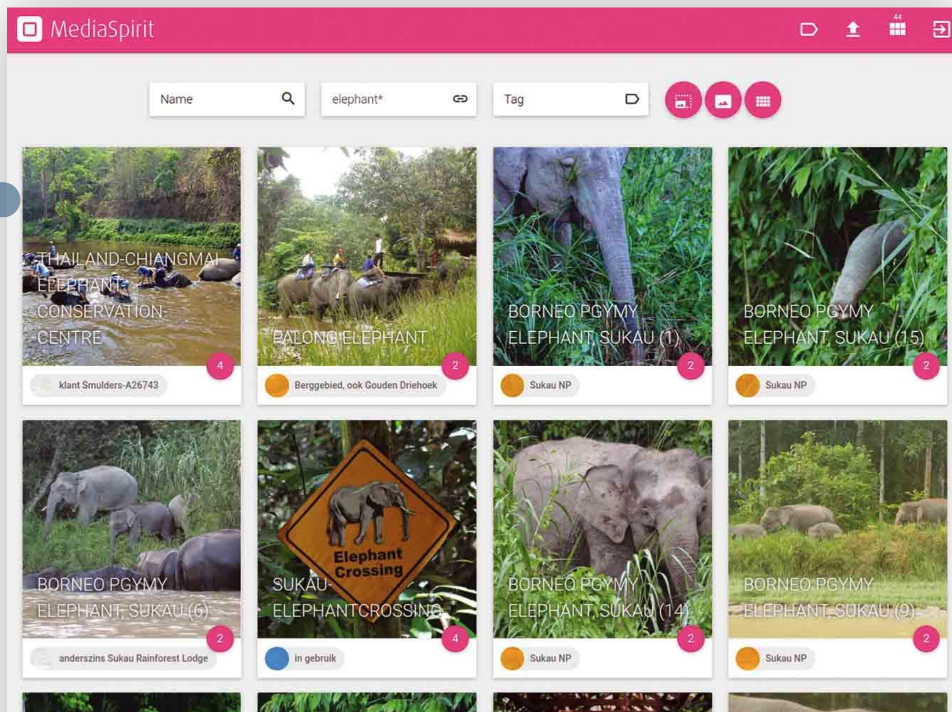
	Traditional models (CF, containers, VMs)	Serverless
High Availability	At least 2-3 instances of everything	No incremental infrastructure
Multi-region deployment	One deployment per region	No incremental infrastructure
Cover delta between short (<10s) load spikes and valleys (vs average)	~2x of average load	No incremental infrastructure
Example incremental costs	2 instances x 2 regions x 2 = 8x	1x

Data processing



http://ecc.ibm.com/case-study/us-en/ECCF-CDC12387USEN

10x faster
90% less cost

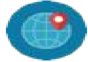


What is Serverless good for?

Serverless is **good** for

short-running
stateless
event-driven

Microservices

 Mobile Backends

Bots, ML Inferencing

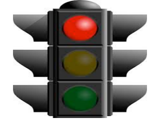
IoT

Modest Stream Processing

Service integration

Serverless is **not good** for

long-running
stateful
number crunching



Databases

Deep Learning Training

Heavy-Duty Stream Analytics

Spark/Hadoop Analytics

Numerical Simulation

Video Streaming

Current Platforms for Serverless

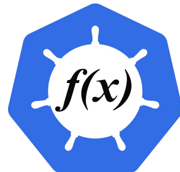
AWS Lambda

OpenLambda

Kubernetes

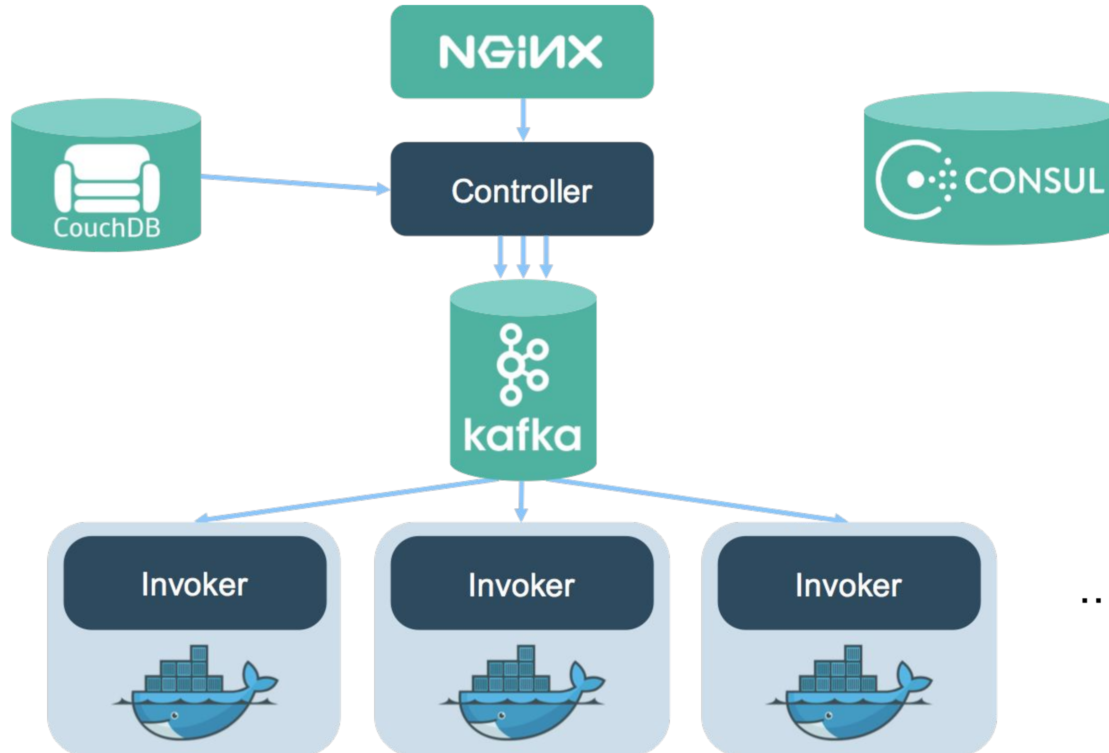
Azure Functions

Google Functions

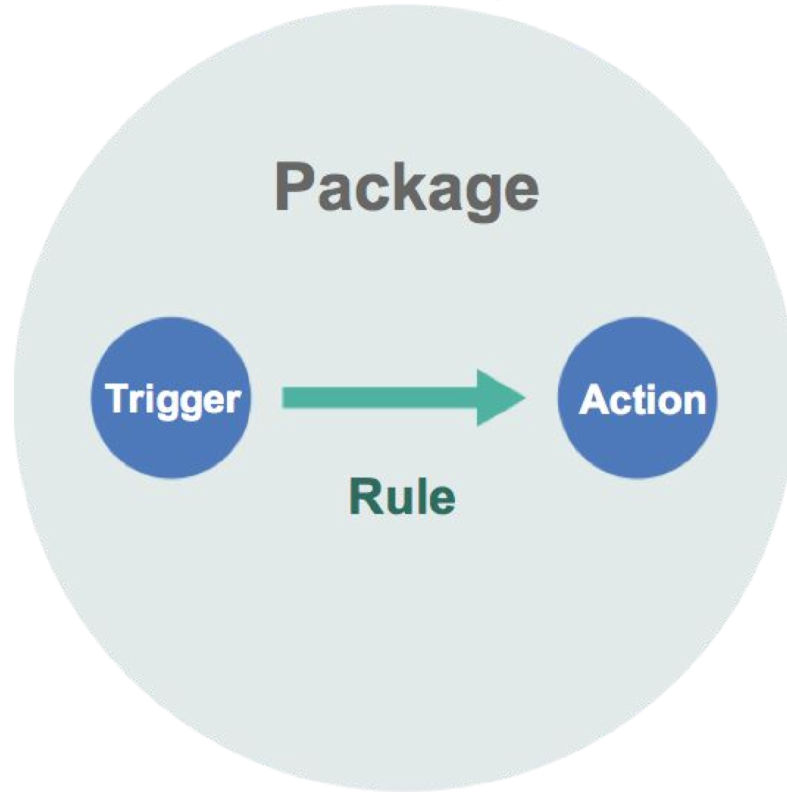


Red-Hat

Apache OpenWhisk Serverless Architecture

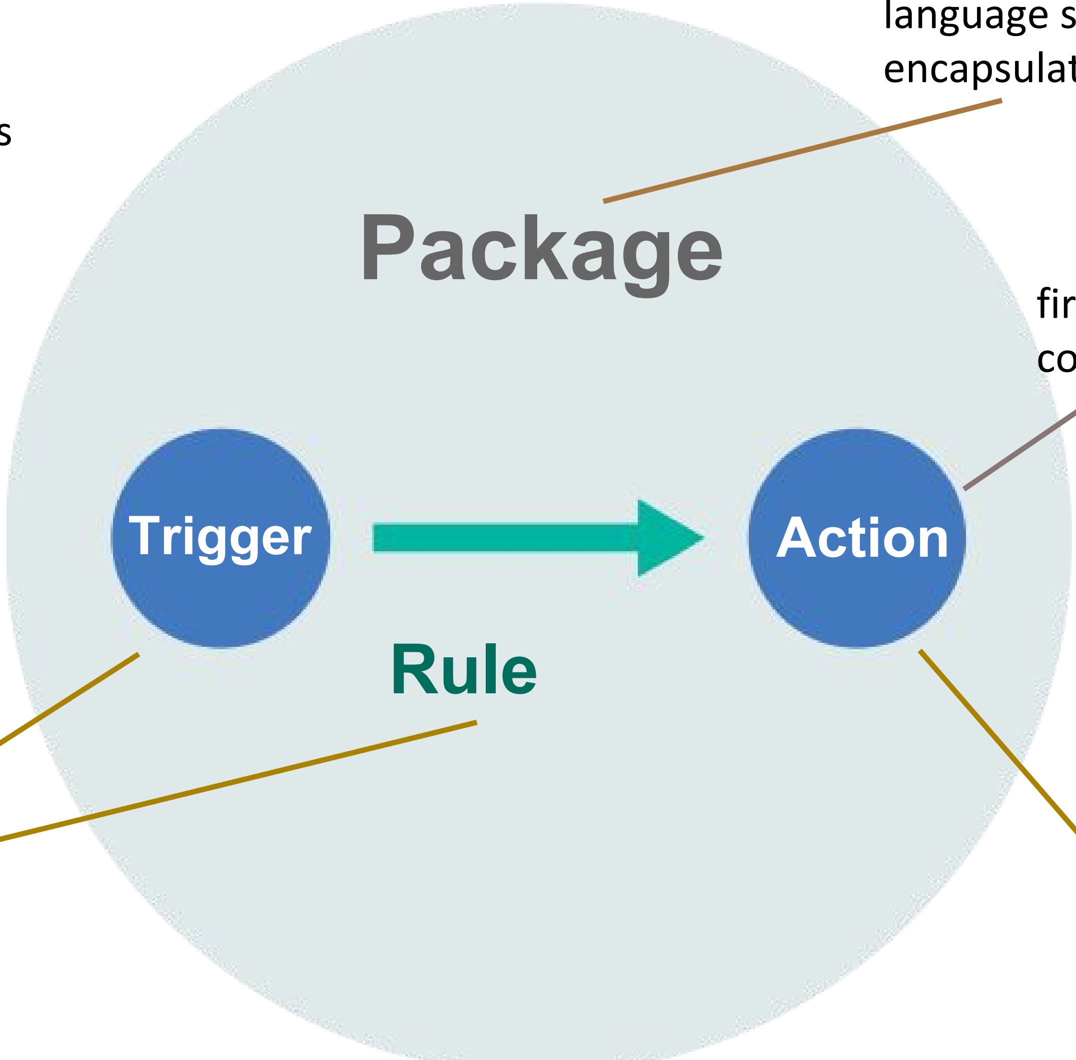


Apache OpenWhisk:
High-level serverless programming model



all constructs first-class — powerful extensible language

language support to encapsulate, share, extend code



first-class functions compose via sequences

first-class event-driven programming constructs

docker containers as actions

A

Action: a stateless function (event handler)

A

Action: javascript

```
function main(params) {
   console.log("Hello " + params.name);
   return { msg: "Goodbye " + params.name) };
}
```

python™

A

Action: Python

```
def lambda_handler(event, context):
    print("hello world")
```

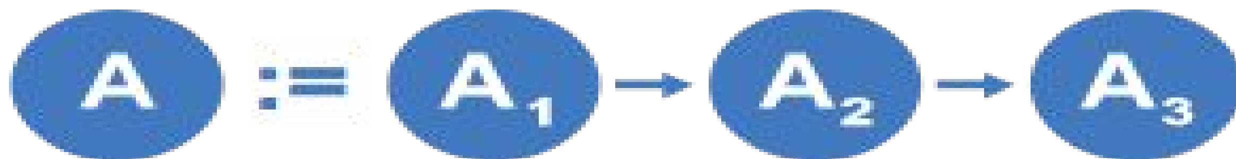
A

Action: Swift

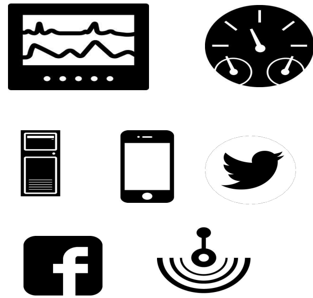
```
func main(params:[String:Any]) -> [String:Any] {
    var reply = [String:Any] ()
    if let name = params[“name”] as? String {
        print(“Hello \(name)”)
        reply[“msg”] = “Goodbye \(name)”
    }
    return reply
}
```

A

Action: sequence

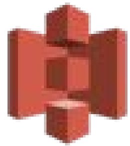


Trigger: a class of events (feed)

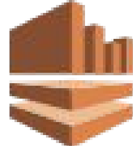


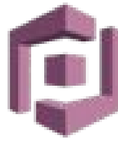
AWS Lambda Trigger Sources

DATA STORES

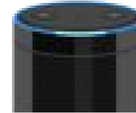

Amazon S3

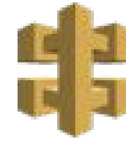
Amazon DynamoDB


Amazon Kinesis


Amazon Cognito

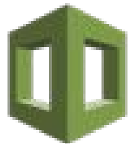
ENDPOINTS

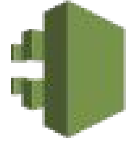

Amazon Alexa

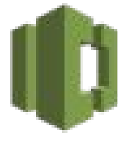

Amazon API Gateway

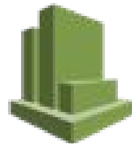
AWS IoT

CONFIGURATION REPOSITORIES

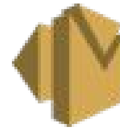

AWS CloudFormation

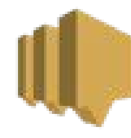

AWS CloudTrail

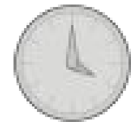

AWS CodeCommit


Amazon CloudWatch

EVENT/MESSAGE SERVICES


Amazon SES


Amazon SNS


Cron events

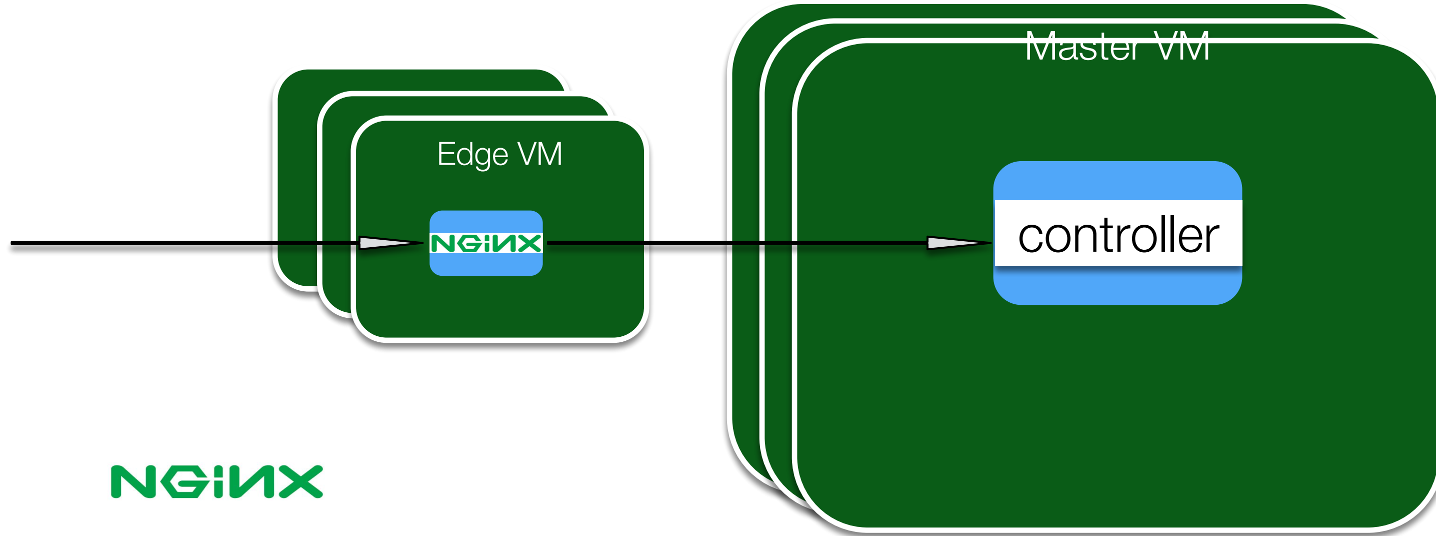
Source: Deep Dive into AWS Lambda, Vyom Nagrani , Manager Product Management, AWS Lambda, AWS Online Tech Talks, January 2017.

R

Rule: a mapping from a Trigger to an Action

Apache OpenWhisk: Step 1. Entering the system

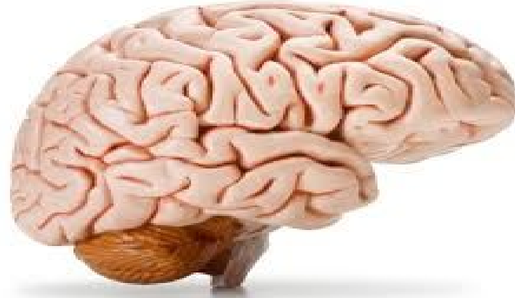
POST /api/v1/namespaces/myNamespace/actions/myAction



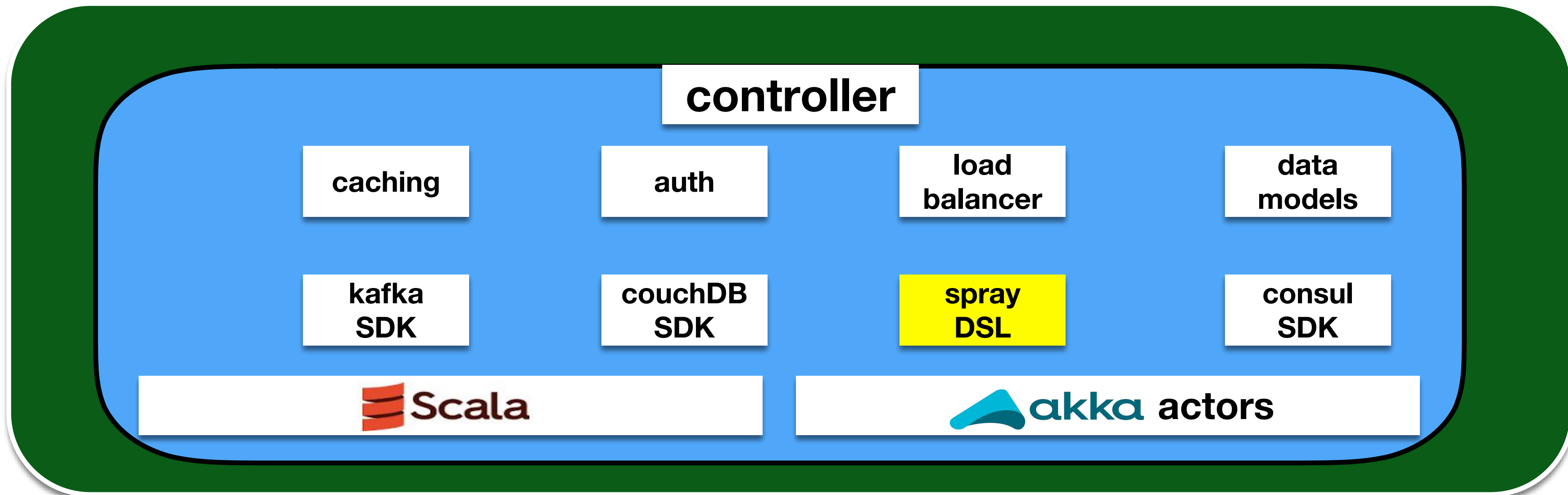
Apache OpenWhisk: Step 2. Handle the request

Master VM

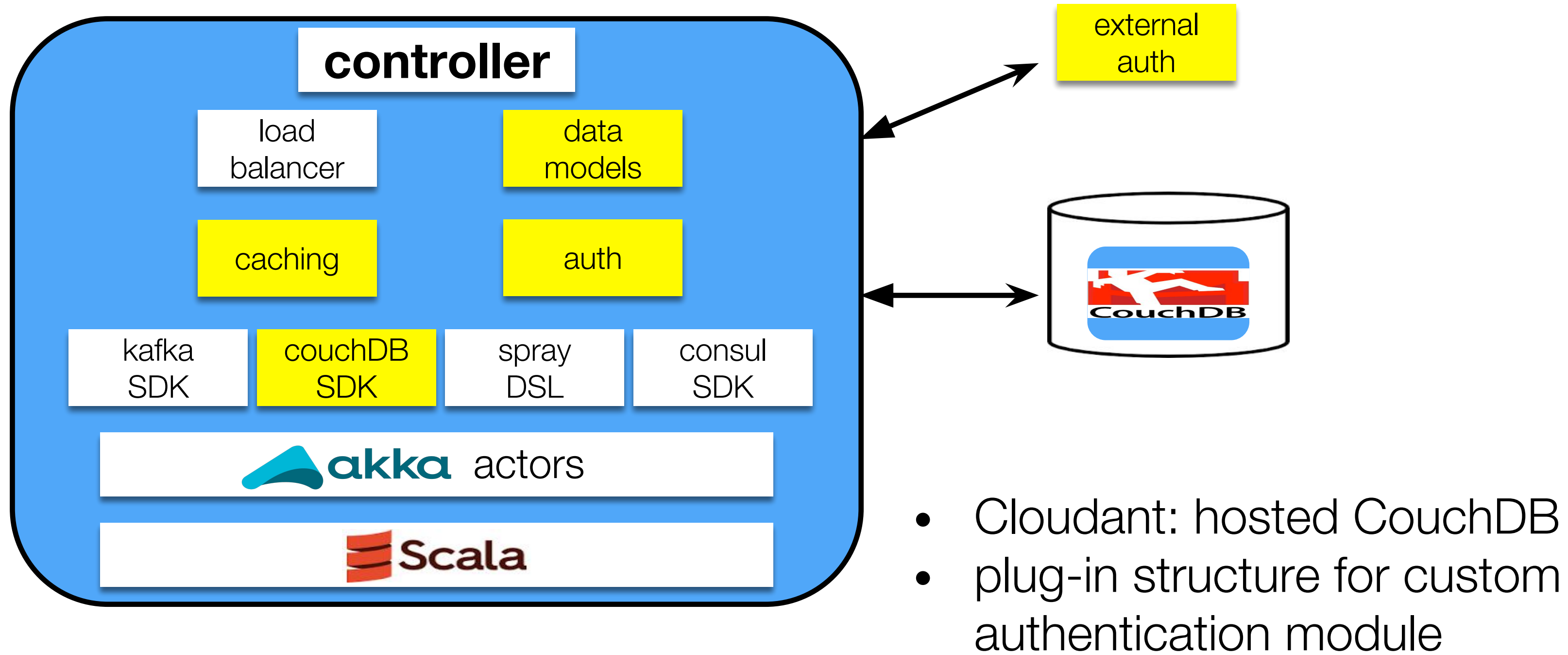
controller



Apache OpenWhisk: Step 2. Handle the request



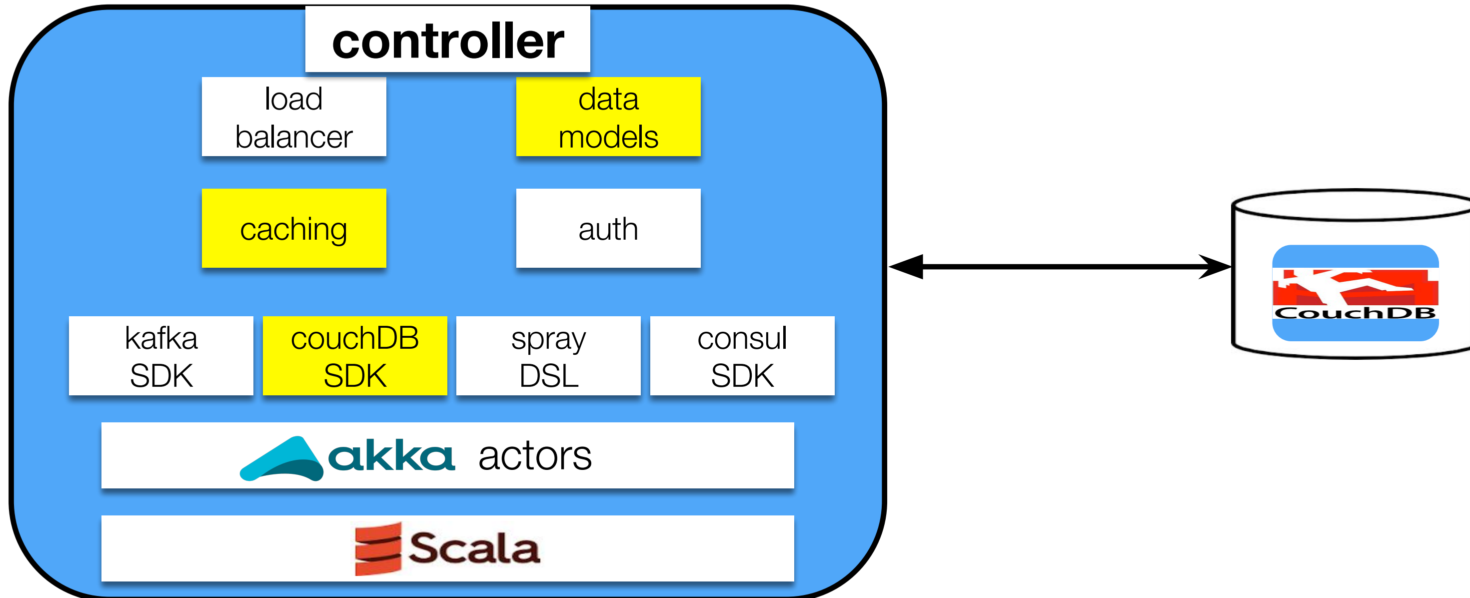
Apache OpenWhisk: Step 3. Authentication + Authorization



- Cloudant: hosted CouchDB
- plug-in structure for custom authentication module

Apache OpenWhisk: Step 4. Get the action

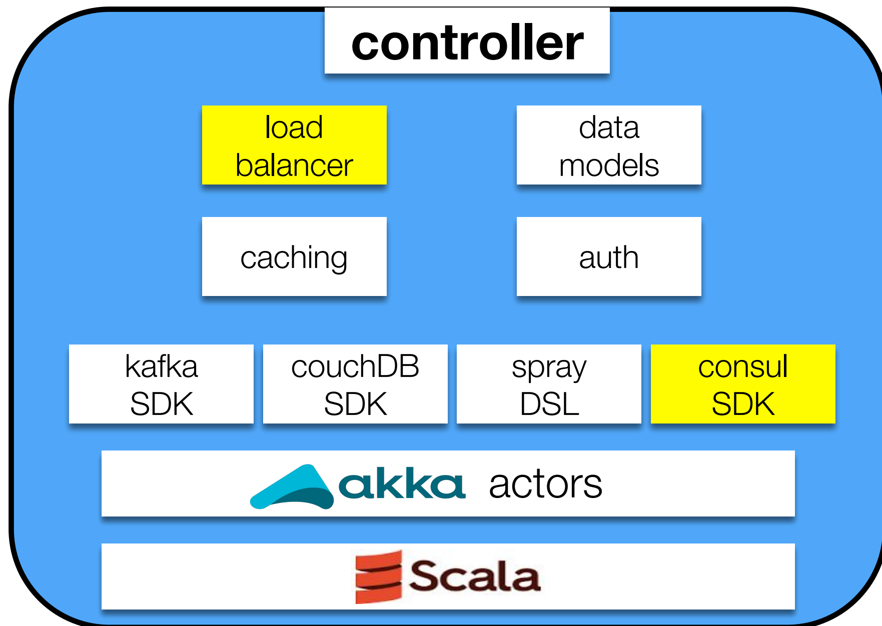
- check resource limits
- actions stored as documents in CouchDB
  - binaries as objects (attachments)

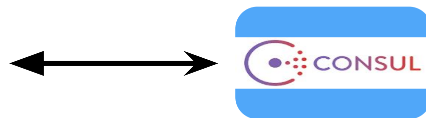


Apache OpenWhisk: Step 5. Looking for a home

Load balancer: find a slave to execute

Slave health, load stored in





Why

?

- Sequentially consistent KV store
- Replication, Fault Tolerance
- Health Check / Monitoring utilities

Apache OpenWhisk: Step 6. Get in line!

Master VM

controller

load balancer

data models

caching

auth

kafka SDK

couchDB SDK

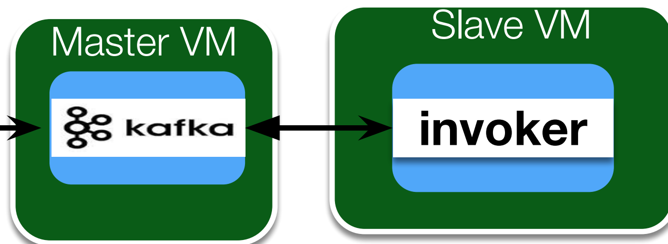
spray DSL

consul SDK

akka actors

Scala

- High throughput fault-tolerant queues
- *Point-to-point* messages via topics
  - explicit load balancing

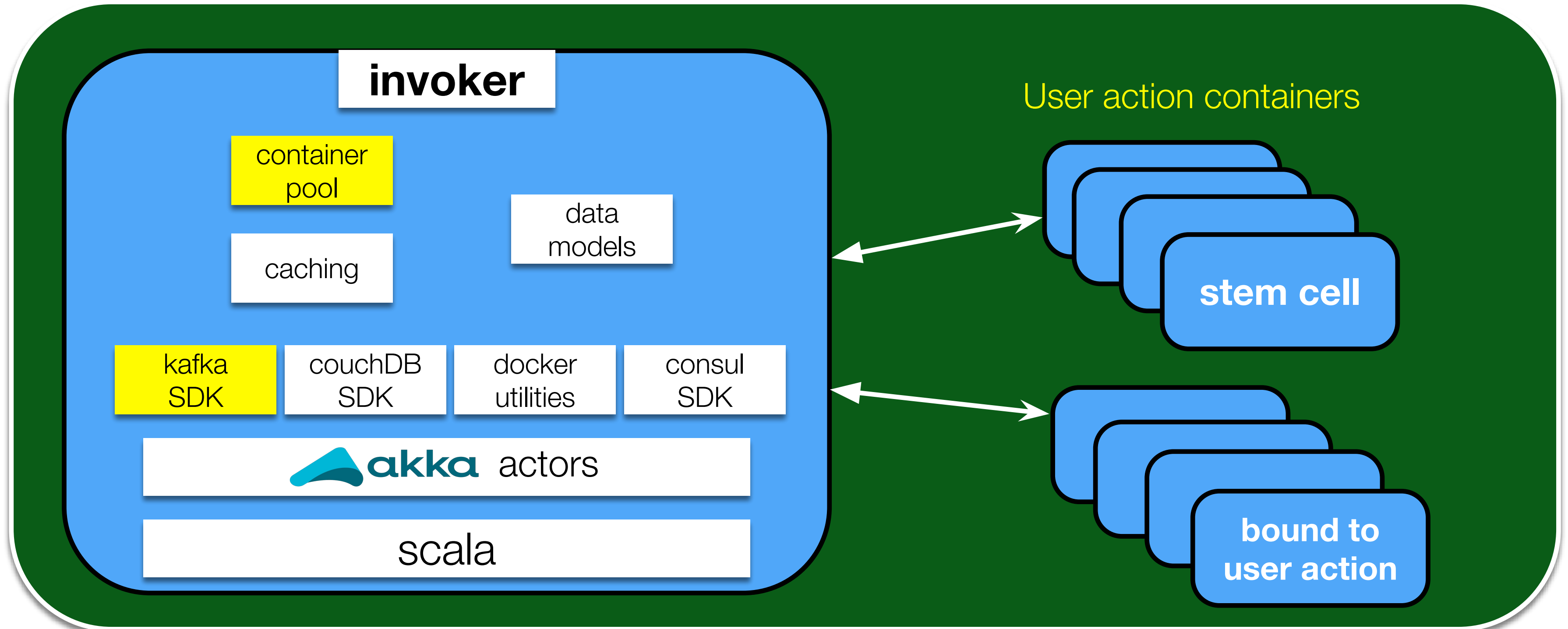


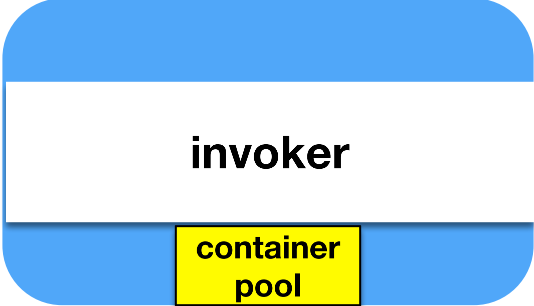
Apache OpenWhisk: Step7. Get to Work!

Slave VM

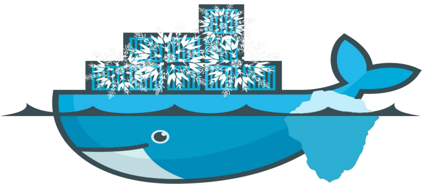
Apache OpenWhisk: Step 7. Get to work!

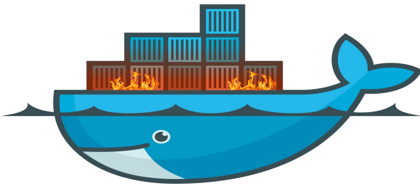
- each user action gets it own container (isolation)
- containers may be reused
- container pool allocates and garbage collects containers

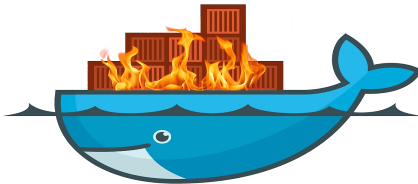



invoker
container pool

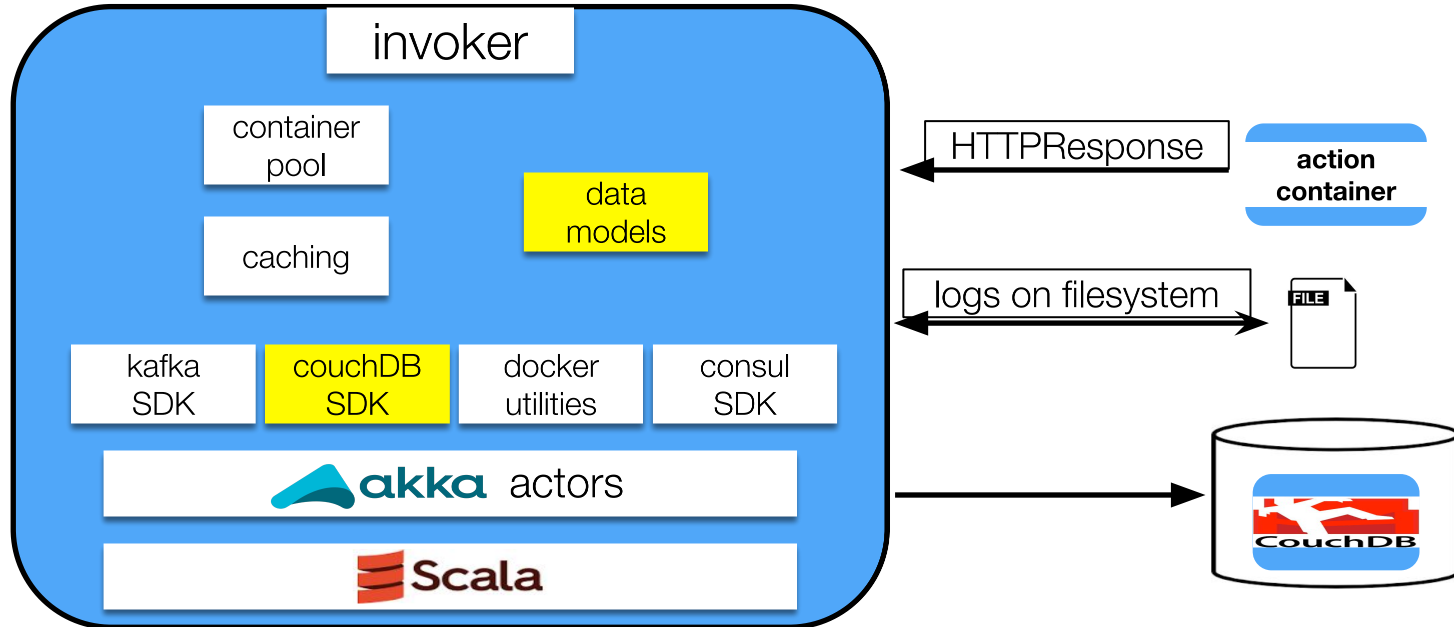
Docker run
HTTP POST /init
HTTP POST /run
cold start
stem cell container
warm container







Apache OpenWhisk: Step 8. Store the results.



Additional architectural concerns for Serverless for service providers

- Cold start problem
  - Keep invokers ready ("stem cell") or running ("warm") after invocation
  - Tradeoff with latency and resource reservation
- Auto scale
  - Add to and remove from the invoker pool
  - Hibernate when idle
- Fine-grained billing
  - Overhead of metering
  - Choice of which resources to bill (CPU, memory, network, …)
  - Understandable billing policy (simple vs detailed)?

Related work

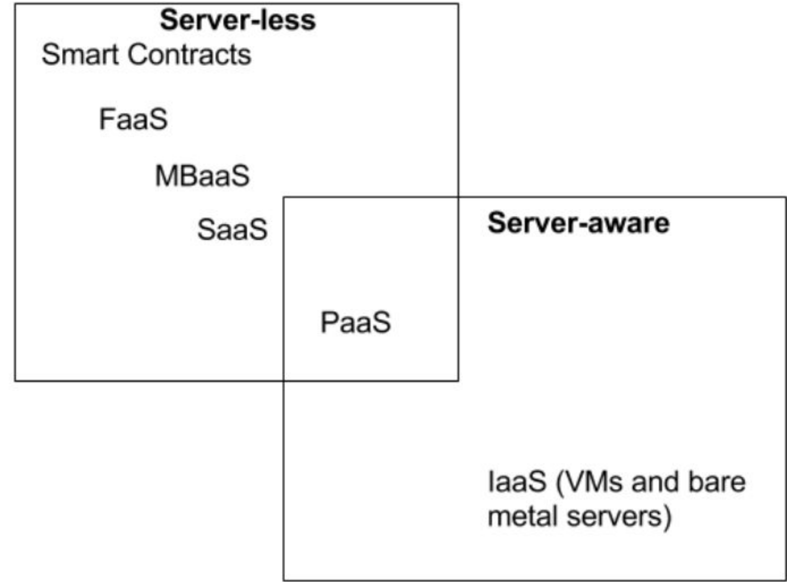
- Reactive programming
- Event-based applications
- Stream processing systems
- Dataflow programming
- Workflows and business processes
- Service composition
- Service oriented architectures
- many more ...

Future of Serverless:
Research Challenges and Questions

Serverless as next step in Cloud Computing?

- Cost - pay-as-you-go is enough?
- Server-less - can servers be really hidden?
- Problem of state: stateless, state in other place, or state-ful supported in FaaS?
- Security - no servers!
- Legacy systems and serverless?
    - Hybrid model?

Cloud computing: server-less vs server-aware?



Programming model(s) for Serverless?

- Tools
- Deployment
- Monitoring and debugging
  - Short-lived functions, scaling to large invocations,
  - Looking for problems is like finding needles in ever growing haystack?
- Serverless IDEs?
- Decompose micro-service into FaaS?
  - Code granularity is function?
- Managing state inside and outside FaaS
- Concurrency, recovery semantics, transactions?

Open Problems - how FaaS fits into cloud?

- Just another *aaS?
- Can different cloud computing service models be mixed?
- Can there be more choices for how much memory and CPU can be used by serverless functions?
- Does serverless need to have IaaS-like based pricing?
- What about spot and dynamic pricing with dynamically changing granularity?

Open Problems: new tooling needed?

- Granularity of serverless is much smaller than traditional server based tool
- Debugging is much different if instead of having one artifact (a micro-service or traditional monolithic app) developers need to deal with a myriad of smaller pieces of code …
  - That haystack can grow really big really fast ...

Open Problems: can “legacy” code be made to run serverless?

- Today the amount of existing (“legacy”) code that must continue running is much larger than the new code created specifically to run in serverless environments
- The economical value of existing code represents a huge investment of countless hours of developers coding and fixing software
- Therefore, one of the most important problems may be to what degree existing legacy code can be automatically or semi-automatically decomposed into smaller-granularity pieces to take advantage of these new economics?

Open Problems: is serverless fundamentally stateless?

- Is serverless fundamentally stateless?
- Current serverless platforms are stateless will there be stateful serverless services in future?
- Will there be simple ways to deal with state?
- Can there be serverless services that have stateful support built-in
  - And with different degrees of quality-of-service?

Open Problems: patterns for building serverless solutions?

- Combine low granularity basic building blocks of serverless (functions, actions, triggers, packages, ...) into bigger solutions?
- How to decompose apps into functions so that they user resources optimally?
- Are there lessons learned that can be applied from OOP design patterns, Enterprise Integration Patterns, etc.?

Open Problems: serverless beyond traditional cloud of servers?

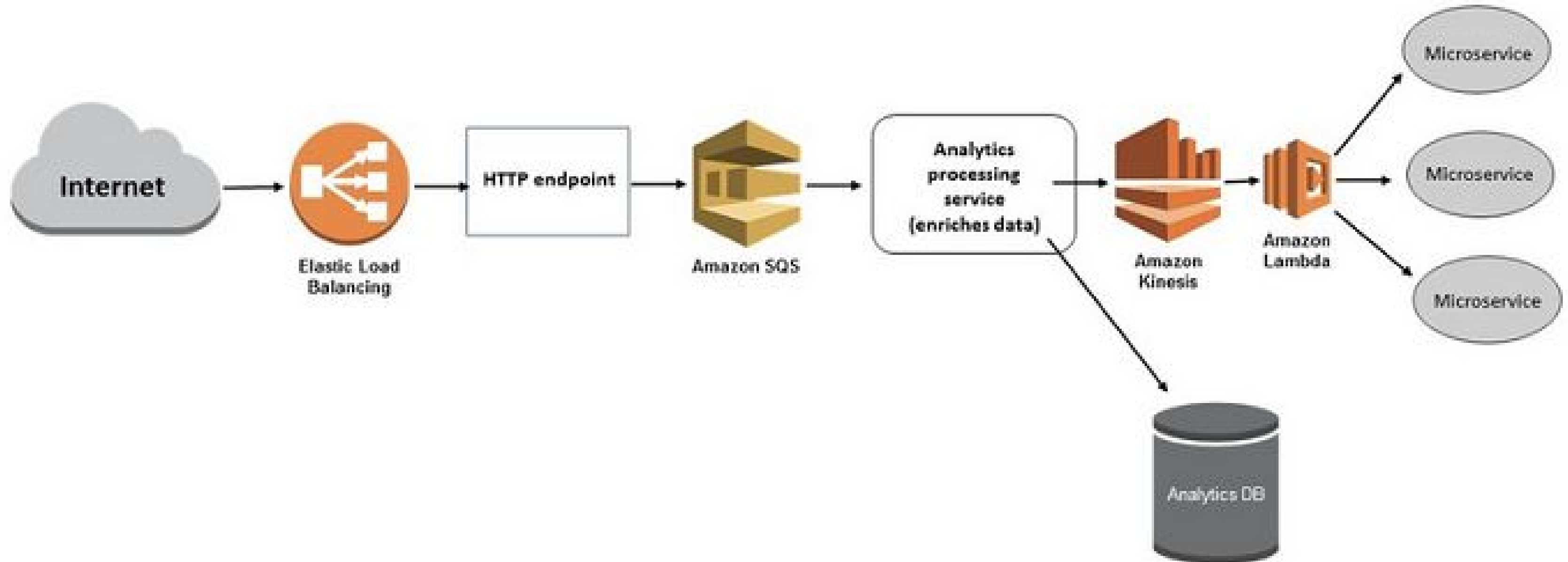
- IF functions is running outside of data-center is it serverless?
  - Cost, scalability, ...
- Internet of Things (IoT) will have many small devices each capable of running small amount of code - like functions in serverless?
- New domains, new concerns?
  - For example for IoT energy usage may be more important than speed?
- Are Blockchain smart contracts server-less?
  - For example when Ethereum users are running smart contracts they get paid for the "gas" consumed by the code, similar to fuel cost for an automobile but applied to computing (no need for data-center!)

Beyond tutorial

- Workshop afternoon with papers and panel discussion
- Slack channel for research discussions?
- And more in our chapter in upcoming book "Research Advances in Cloud Computing”
  - https://www.springer.com/us/book/9789811050251#aboutBook

Backup

AWS Lambda Use Case



Serverless Architecture (Apache OpenWhisk)