Serverless Computing

Author: Richard Clauß

Docent: Prof. Dr. Martin Leischner
Project Owner: Dr.-Ing. Andreas Schieder

Hochschule
Bonn-Rhein-Sieg

Agenda

1. Basics
2. Runtimes
3. Events and Triggers
4. Architecture
5. Networking and Edge
6. Databases, Storage, APIs
7. Stream Processing
8. Performance
9. Software Development

Agenda

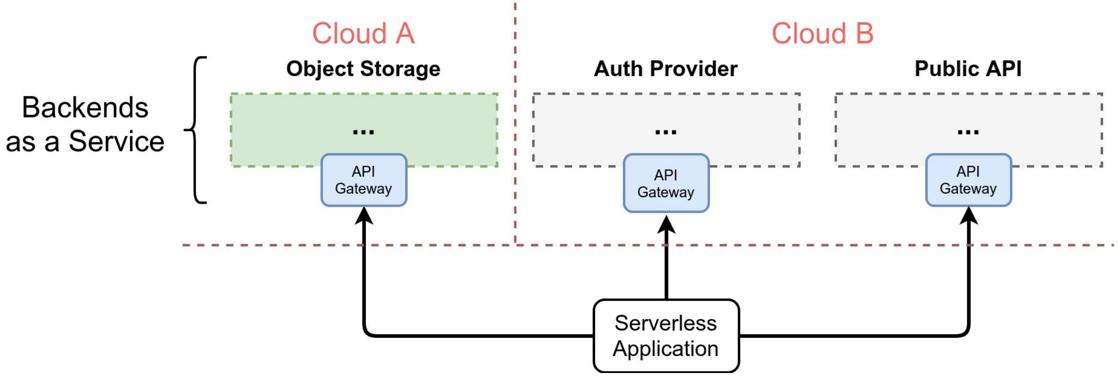
10. Function orchestration and keeping state
11. Economic View
12. Conclusion

1. Basics

Introduction and Definitions

1.1 What does “Serverless” mean?

a) Serverless as an Architecture



Serverless Architecture

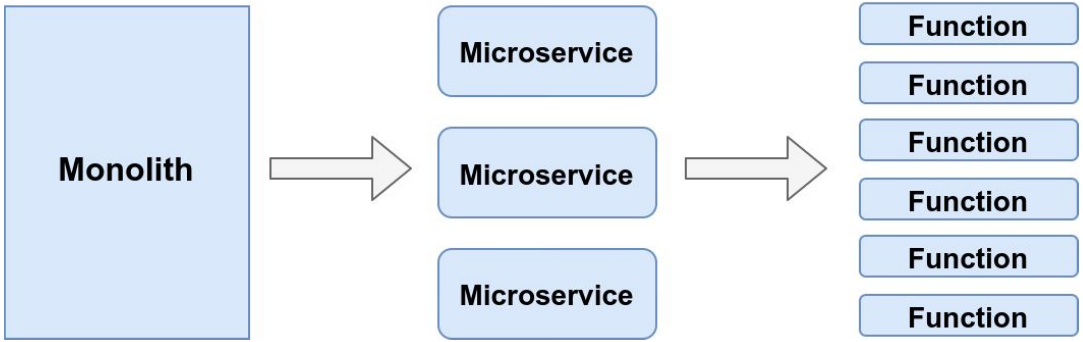
- multiple backends
- might span different cloud providers
- transparent and automatic scaling
- backend servers unknown

1.1 What does “Serverless” mean?

b) Serverless as Cloud-computing Execution Model

called **“Function as a Service”**

Are functions just smaller pieces of Microservices?



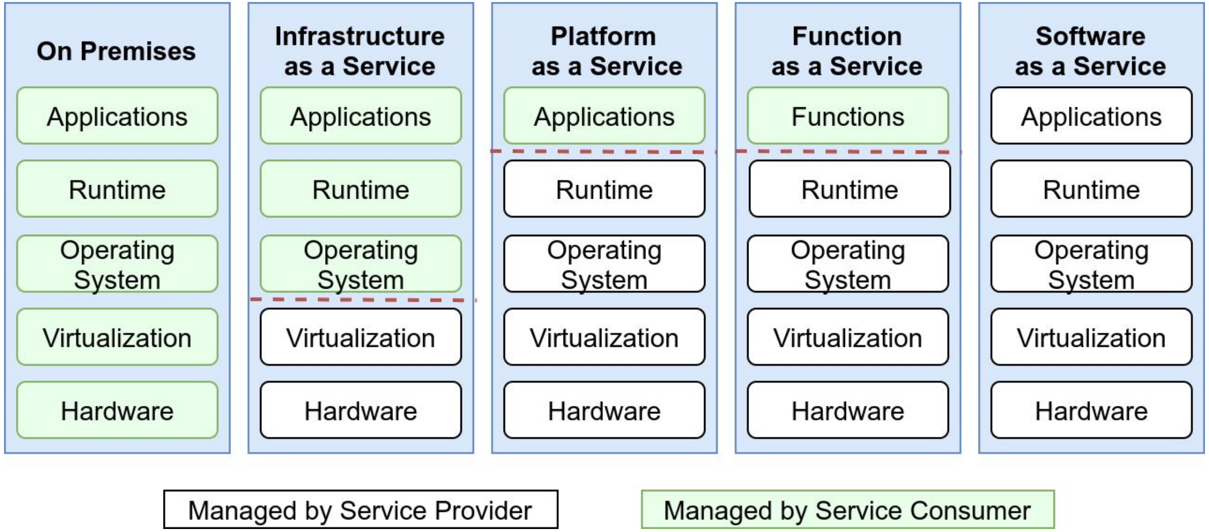
Sources: [6]

1.1 What does “Serverless” mean?

b) Serverless as Backend Execution Model

called **“Function as a Service”**

Which are then run on a runtime like usual PaaS Containers?



Sources: [6]

1.2 FaaS vs. PaaS

Are functions just smaller pieces of Microservices?

Which are then run on a runtime like usual PaaS Containers?

Yes, but not exactly

Platform as a Service

- Runs containers (or on other runtimes)
- Long running (usually)
- Stateless or stateful
- Scales by configuration
- Event-driven or permanently running
- Can have side effects

Function as a Service

- Runs containers (usually)
- Short lived = ephemeral = transient ← implicit need for small size
- Stateless
- Scales automatically
- Event-driven → executed when triggered
- Can have side effects

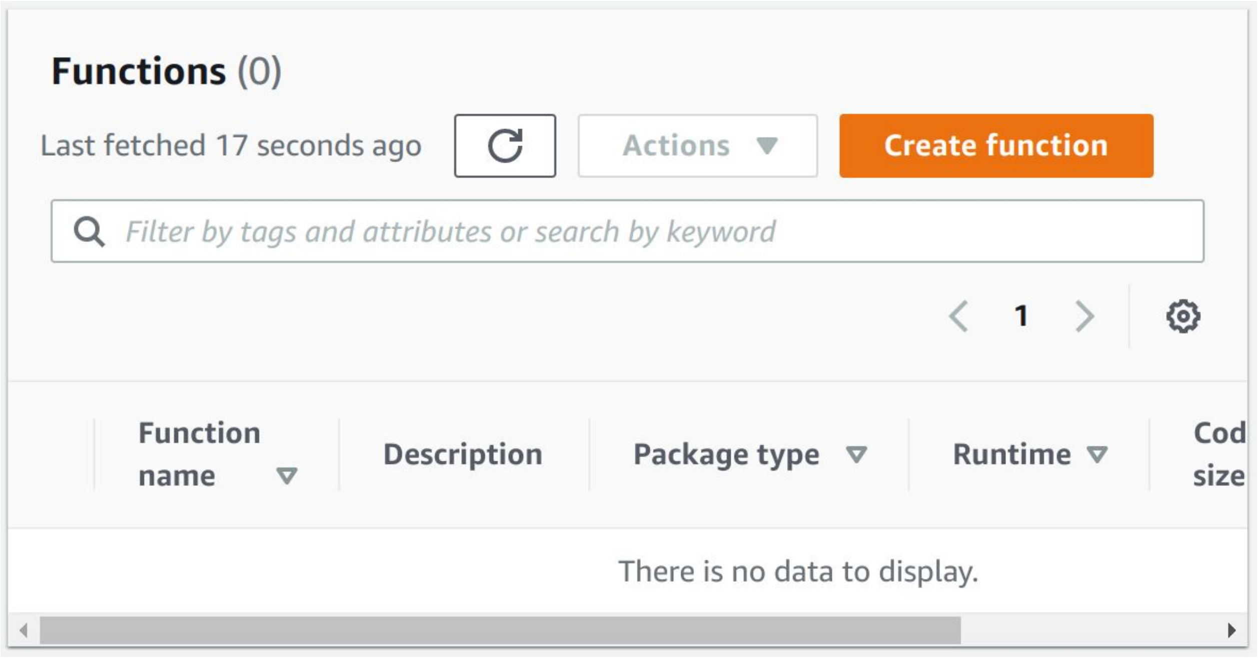
Sources: [6,9]

1.2 FaaS vs. PaaS

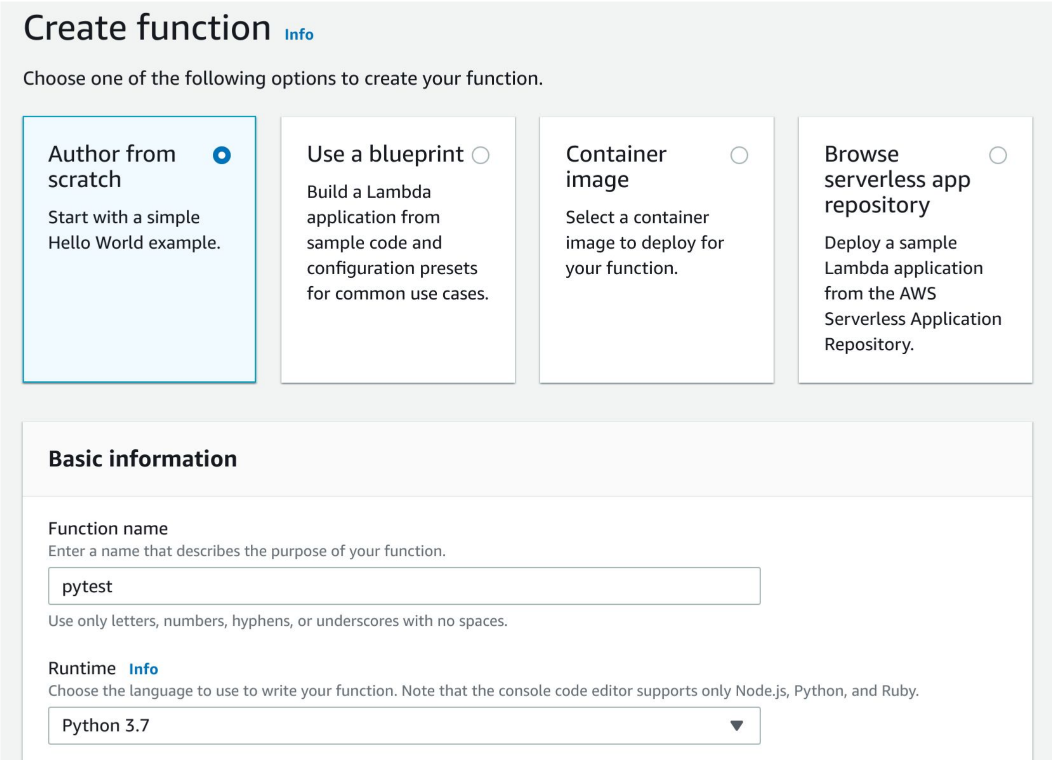
How are PaaS and FaaS different if both are containers?

Sources: [6]

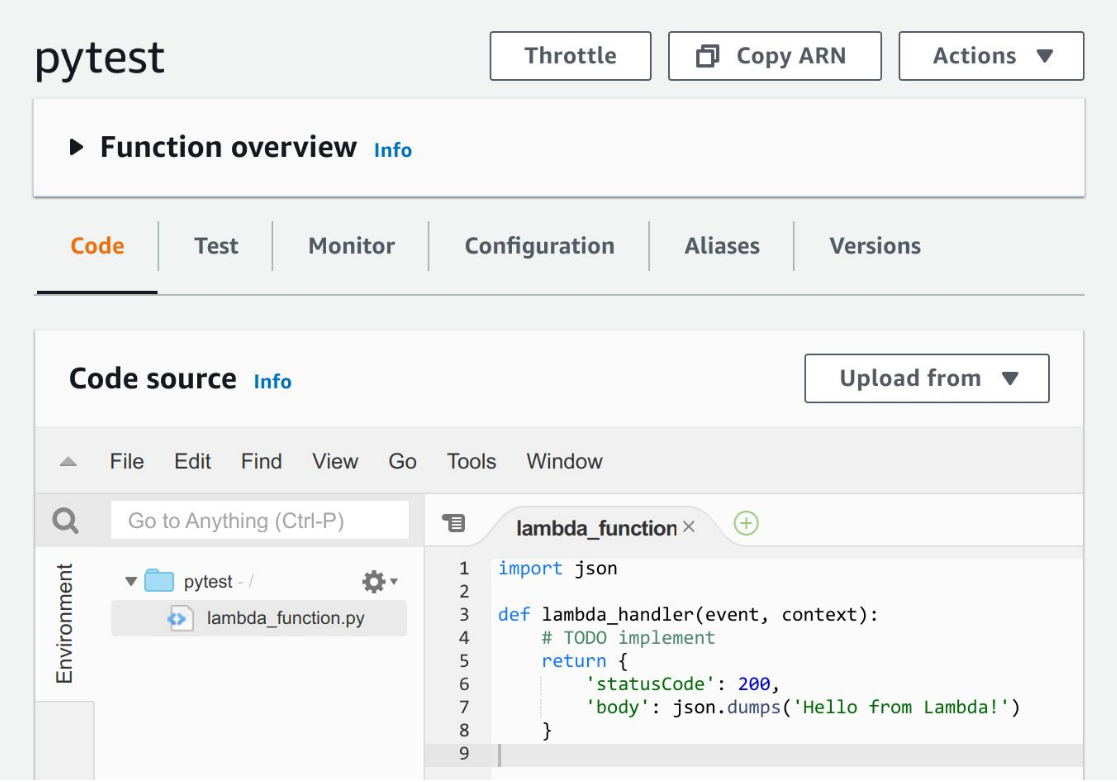
1.3 AWS Lambda Example



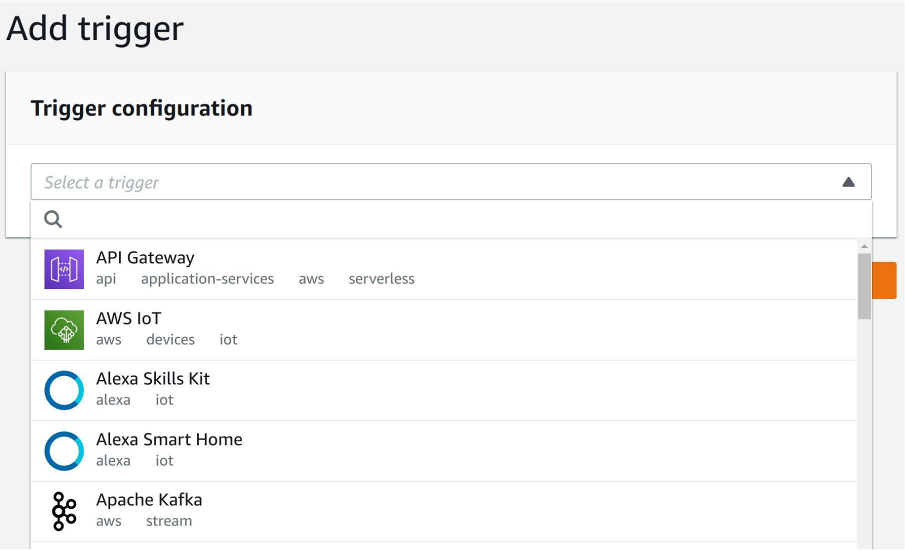
1.3 AWS Lambda Example

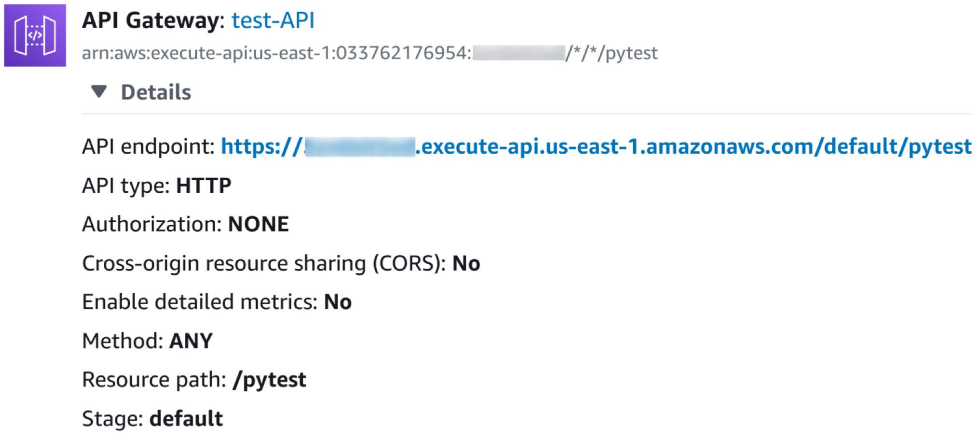


1.3 AWS Lambda Example

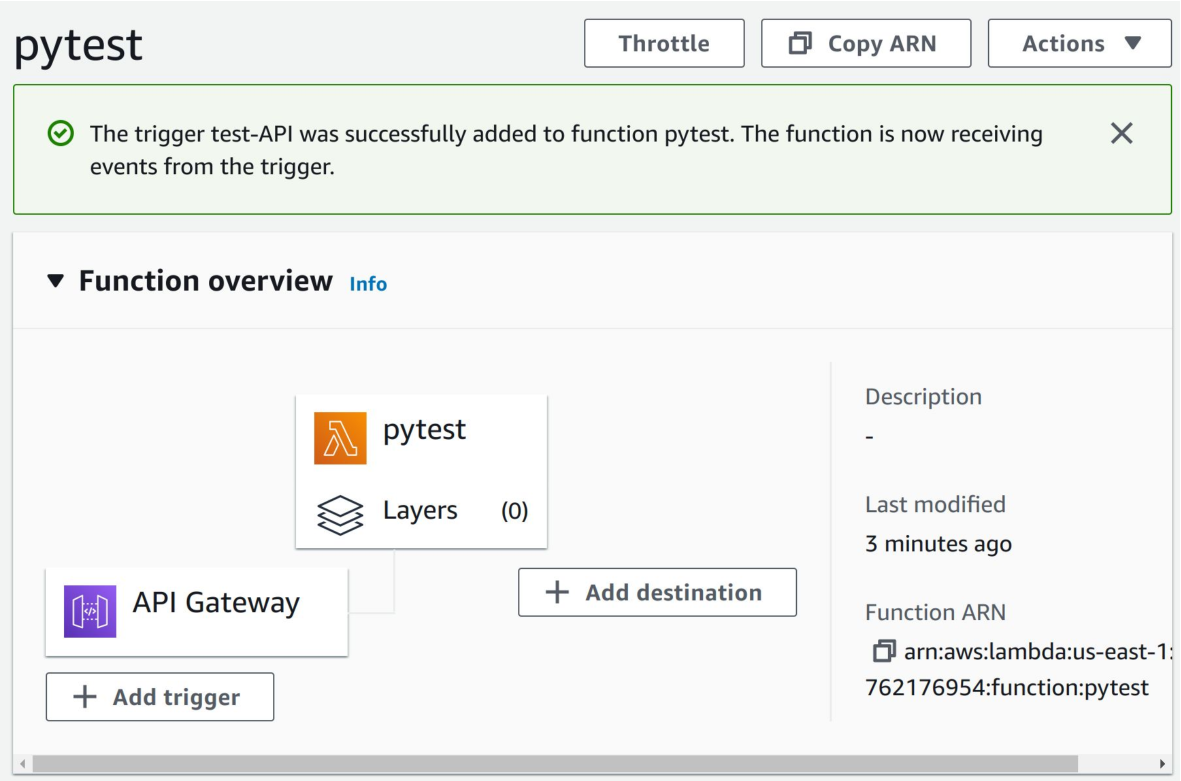


1.3 AWS Lambda Example

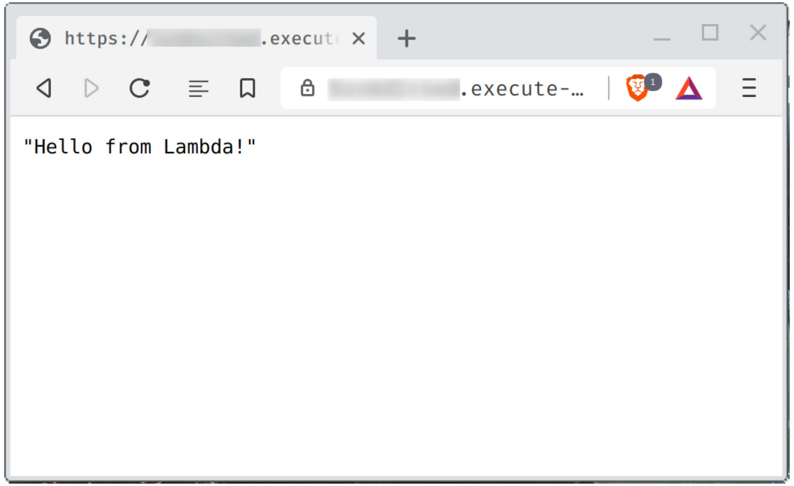




1.3 AWS Lambda Example



1.3 AWS Lambda Example



Function Logs first invocation

START RequestId: df51350a-dd12-46dd-95d0-d23ba7c524cd Version: $LATEST

END RequestId: df51350a-dd12-46dd-95d0-d23ba7c524cd

REPORT

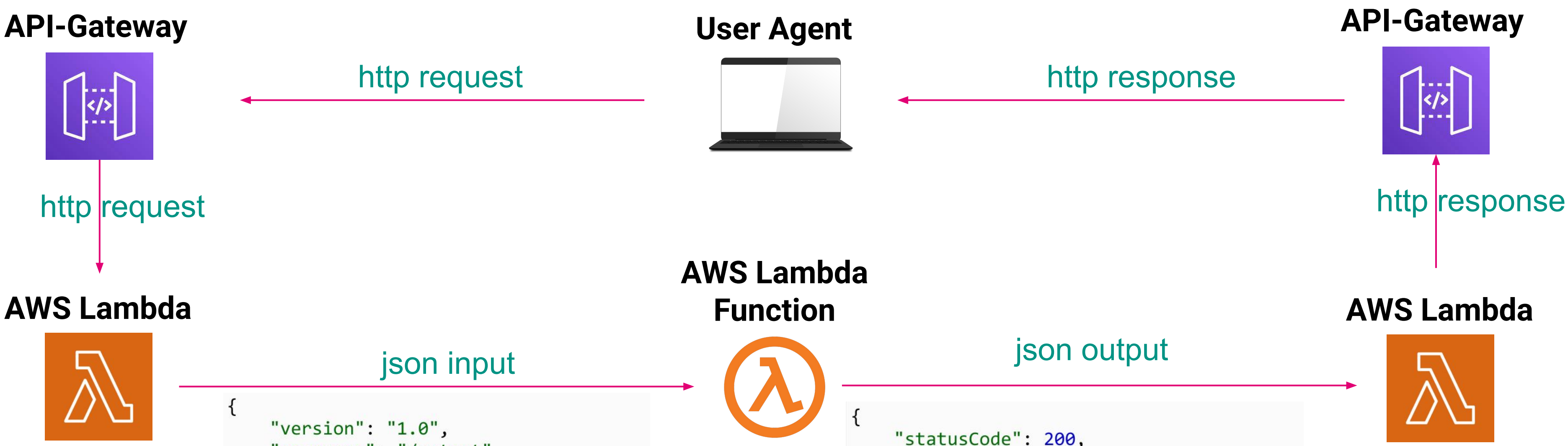
RequestId: df51350a-dd12-46dd-95d0-d23ba7c524cd
Duration: 1.32 ms
Billed Duration: **2 ms** ← billed
Memory Size: **128 MB** ← billed
Max Memory Used: **50 MB** ← not billed
Init Duration: **132.34 ms** ← not billed

Request ID
df51350a-dd12-46dd-95d0-d23ba7c524cd

REPORT for **second invocation**:

Duration: 0.89 ms
Billed Duration: 1 ms
(“Init Duration” disappeared)

1.3 AWS Lambda Example




```
{
    "version": "1.0",
    "resource": "/pytest",
    "path": "/default/pytest",
    "httpMethod": "GET",
    "headers": {
        "Content-Length": "0",
        "User-Agent": "Mozilla/5.0...",
    },
    "queryStringParameters": null,
    "body": null,
    [...much more]
}
```

```
{
    "statusCode": 200,
    "headers": {
        "Content-Type": "application/json"
    },
    "body": "Hello",
    [...much more]
}
```

One function can handle multiple subpaths.

1.3 AWS Lambda Example

Event = json Input

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

Context

- App Name
- Function Name
- Memory Limit
- Amazon Features (Log Group, etc.)
- Custom Environment variables

```python
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

json output

1.4 Triggers in Amazon AWS Lambda

Amazon AWS Lambda is highly integrated into the AWS Service Portfolio, Event Sources include:

Invoke functions on Database updates

DynamoDB

Trigger on File manipulations in Object Storage

amazon S3

Consume real time data streams

amazon Kinesis

Amazon MSK

Message Queues and Work Queues

amazon SQS

Amazon MQ -> RabbitMQ Apache ACTIVEMQ

External Events via Amazon EventBridge

Amazon EventBridge

Scheduled Events (Cronjob) via CloudWatch Events

Amazon EventBridge CloudWatch

Sources: [14]

1.5 Software and Providers

Commercial

- Amazon AWS Lambda
- IBM Cloud Functions
- Oracle Cloud Functions
- Google Cloud Functions
- Microsoft Azure Functions
- Cloudflare Workers
- Vercel Cloud Functions
- Tencent Cloud Functions

using

using

Open Source

- OpenWhisk (Apache-2.0 License)
- Fn (Apache-2.0 License)
- Knative (Apache-2.0 License)
- OpenFaaS (MIT License)
- Kubeless (Apache-2.0 License)
- Fission (Apache-2.0 License)

High amount of FaaS-providers enables use case for serverless framework
→ provides common cli and project structure

Sources: [5]

2. Runtimes

and their implementation in OpenWhisk

2.1 Serverless Runtimes

Serverless Runtimes (in OpenWhisk)

- are containers made for starting and shutting down very fast
- each action=function has its own container

Additional Possibilities

- Run own containers
  → which implement http API
- Run skeleton container
  - own executables
  - shell scripts

  → implements API for us

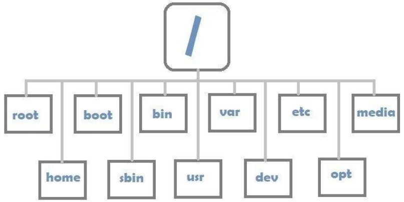
Official OpenWhisk Runtimes

- .net
- Go
- Java
- JavaScript
- PHP
- Python
- Ruby
- Swift
- Ballerina
- Rust

Sources: [4]

2.2 Environment from the perspective of a function

Function sees usual filesystem



(usual container behaviour)

Executed as Non-Root user

(implementation may differ)

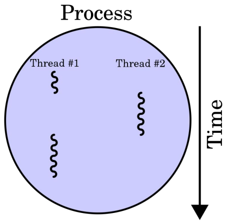
Networking like any other Kubernetes Pod or Container

(e.g. AWS VPC with Internet access)

Execution Time Limit

(function killed if it exceeds limits)

Threading as usual



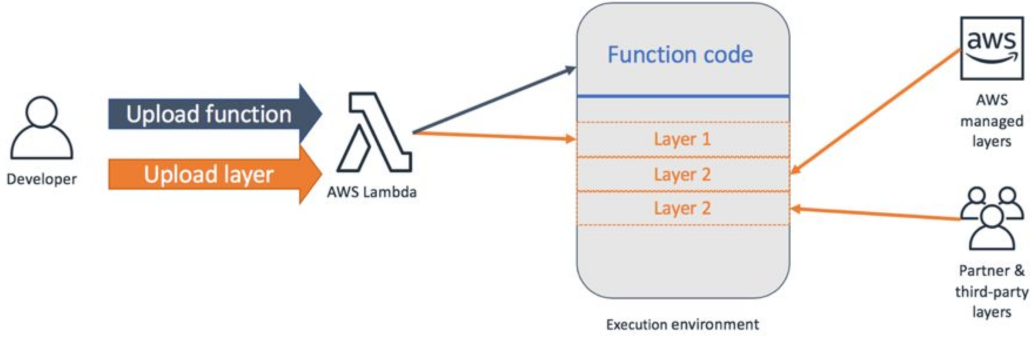
Json input already parsed to native data structure

(statically typed languages
user experience may vary, e.g. java)

Some non-standard libraries are present

(for doing http requests, parsing json, etc.)

2.3 Extending Runtimes



Runtimes can be extended

- Add new container layers
- Provide libraries in deployments

Extend Python Runtimes in OpenWhisk

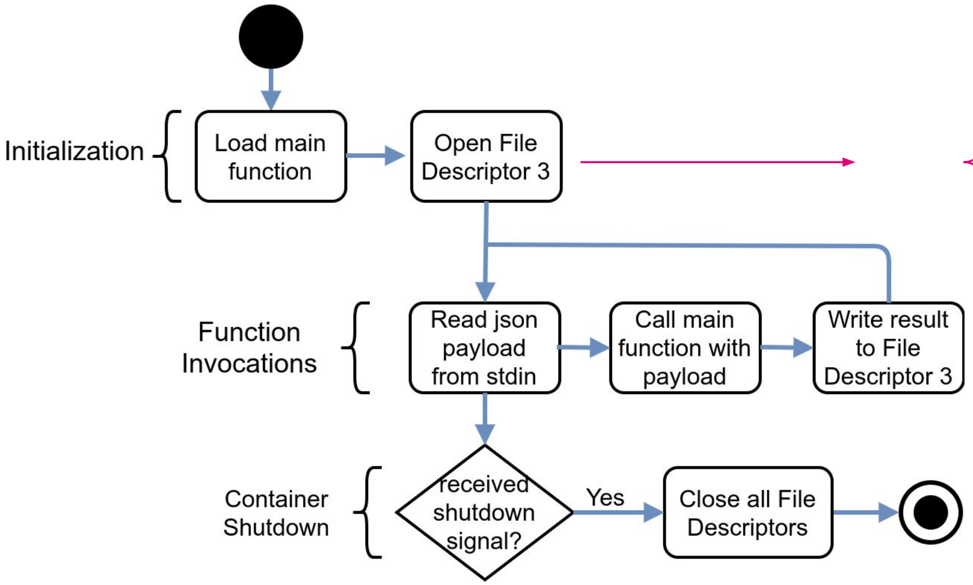
1) virtualenv virtualenv
2) source virtualenv/bin/activate
3) pip install <dependency>
4) zip -r helloPython.zip virtualenv __main__.py
5) wsk action create helloPython --kind python:3 helloPython.zip

```
faas_project
├── hello.py
└── virtualenv
```

Sources: [4,17]

2.4 OpenWhisk Python Runtime

(Code slightly shortened version of the original OpenWhisk Python Runtime Implementation

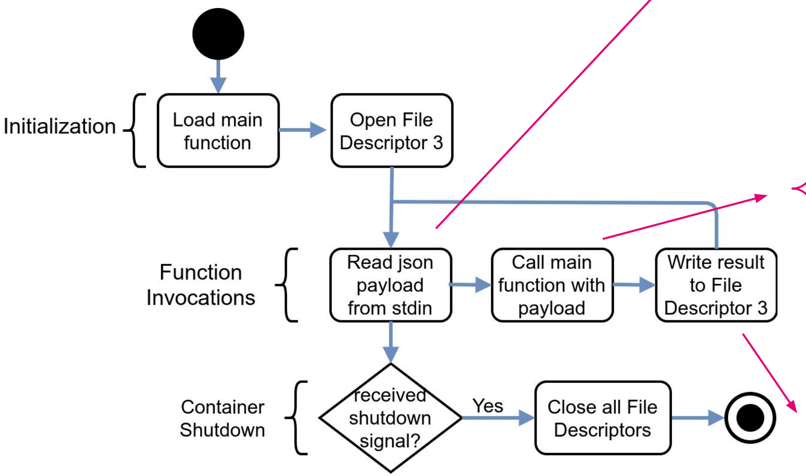



```python
# import the action == function
from main__ import main as main

# Open File Descriptor 3 for output
out = fdopen(3, "wb")

```

2.5 OpenWhisk Python Runtime




```python
while True:
  # read line on each invocation
  line = stdin.readline()
  if not line: break

  # Parse json input
  args = json.loads(line)
  payload = {}
  for key in args:
    if key == "value":
      payload = args["value"]
    else:
      env["__OW_%s" % key.upper()]= args[key]
  res = {}

  # execute the function
  try:
    res = main(payload)
  except Exception as ex:
    print(traceback.format_exc(), file=stderr)
    res = {"error": str(ex)}

  # write result to fd 3
  out.write(json.dumps(res, ensure_ascii=False).encode('utf-8'))
  out.write(b'\n')
  stdout.flush()
  stderr.flush()
  out.flush()
```

Sources: [3]

2.6 Consequences of OpenWhisks Implementation

Consequences of OpenWhisks Implementation

- Json is already parsed to native data structures
- Deserialization and Serialization means overhead

- One runtime serves exactly one action
- Runtimes can’t be reused to serve other actions

- Reexecuting same function is fast (warm start)
- First execution is delayed (cold start)

2.7 Limitations

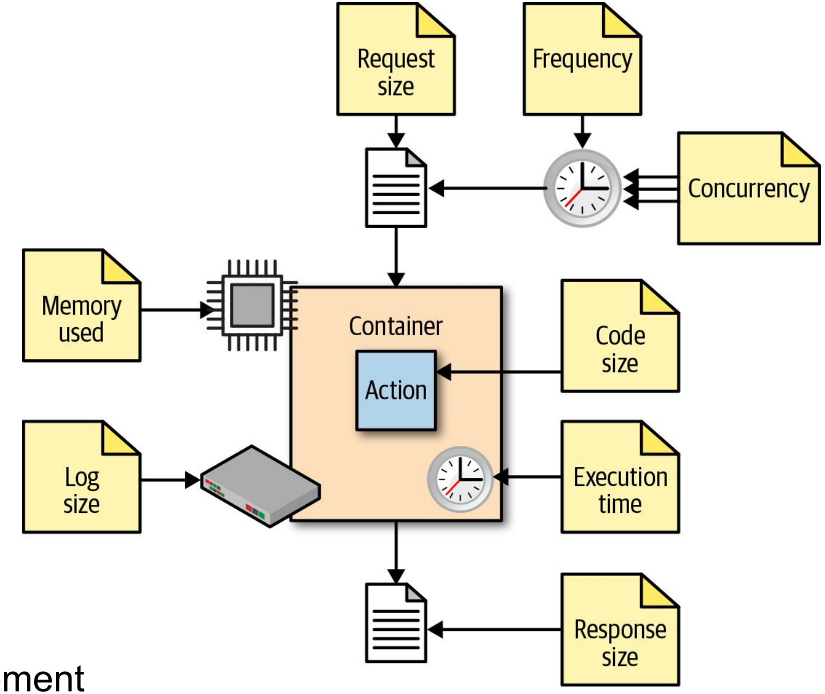
Introduced Problem:
Programmers need to take limitations into account which are specific to the used runtime and software

Limitations in AWS Lambda:

- **unlimited** concurrency (function scale first to region dependent **500-3000** concurrent instances, then **500** more each minute)
- **10240 MiB** max memory usage (def: **128 MiB**)
- **900** sec max execution time (def: **3 sec**)
- **6 MiB** synchronous invocation payload
- **256 KiB** asynchronous invocation payload

→ payload limits make some use cases hard to implement

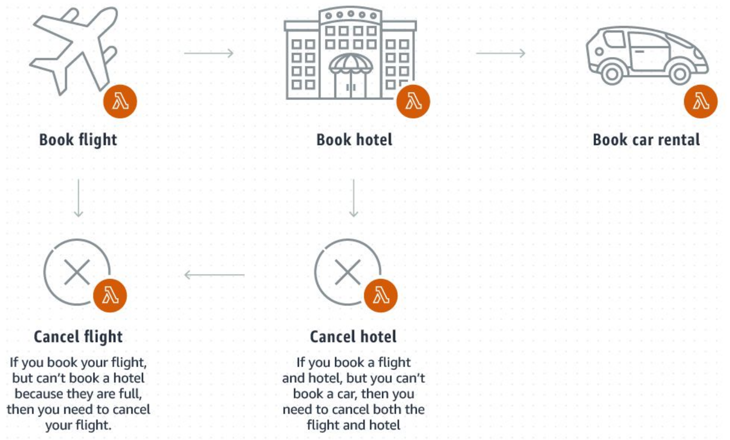
Limitations in OpenWhisk:



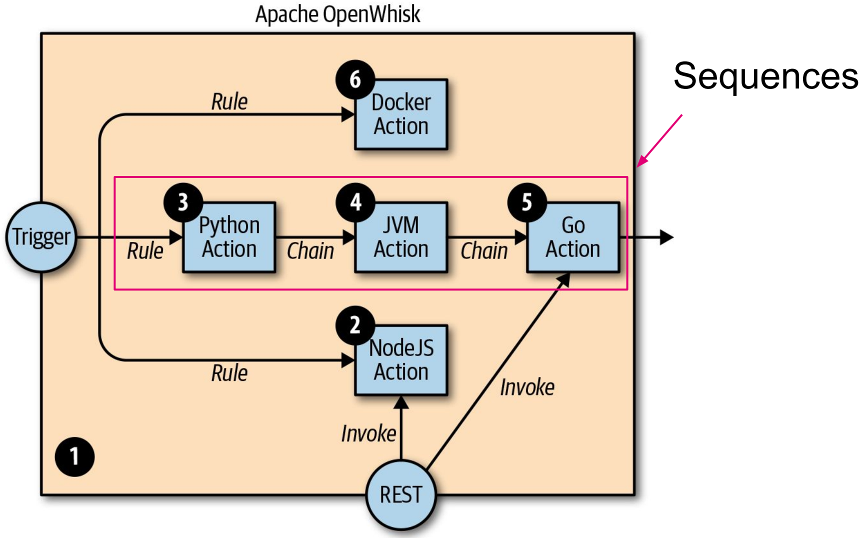
2.8 Orchestrations, Sequences and Step Functions

Applications might need to be split into smaller parts and chained together to avoid limitations.

Amazon AWS Step Functions



OpenWhisk Sequences

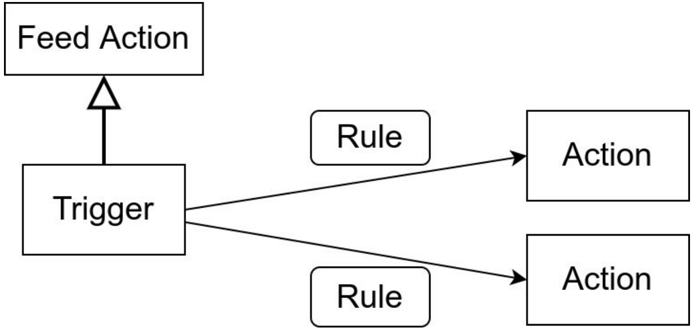


3. Events and Triggers

Closer look to their implementation in OpenWhisk

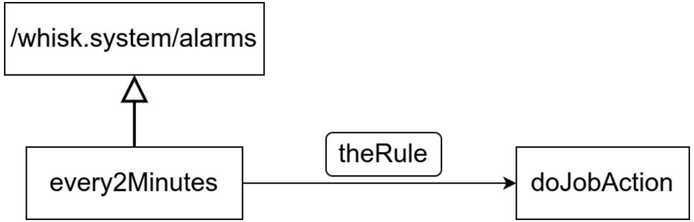
3.1 OpenWhisk Event Model

Feeds, Triggers and Rules



Feed Action = Controls lifecycle for a stream of events
Trigger = By parameters specified class of events
Feed = Stream of events belonging to a specific trigger
Rule = Connects Triggers and Actions
Action = The function to execute on each event

Cronjob Example



Creating a trigger from a feed:

```
wsk trigger create every2Minutes \
   --feed /whisk.system/alarms/alarm \
   --param cron "*/2 * * * *" \
   --param trigger_payload "{\"key\":\"val\"}"
```

Creating a rule:

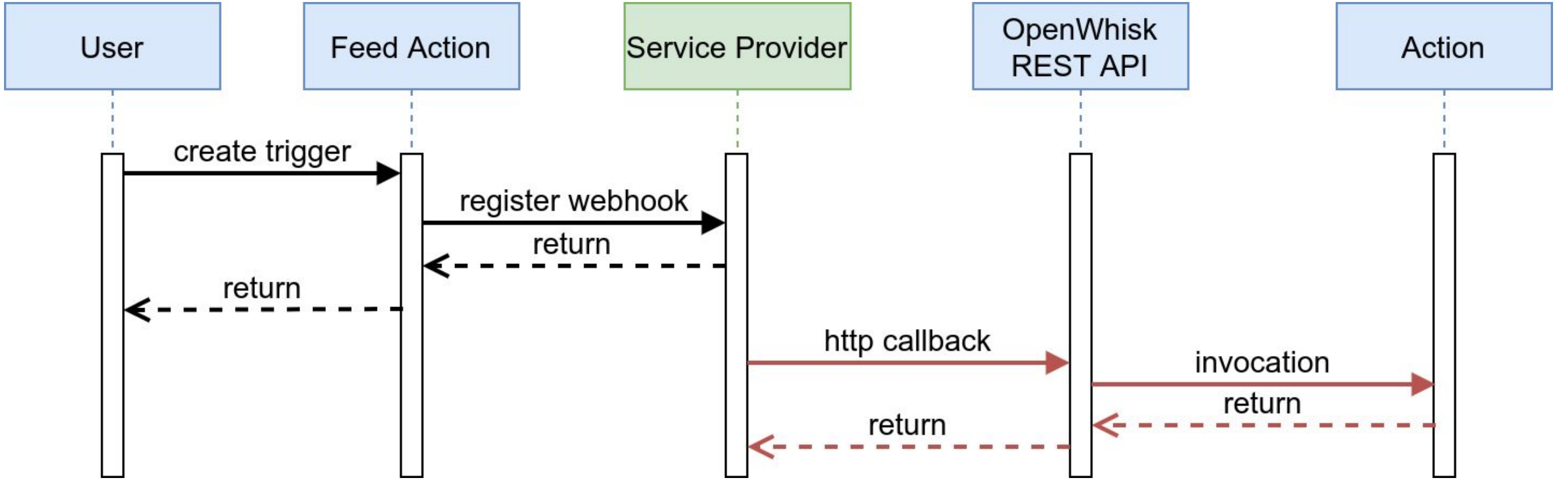
```
wsk rule create theRule every2Minutes doJobAction
```

Sources: [19]

3.2 OpenWhisk Event Model

“Telegram, send me a http request, if my bot received a new message”

a) Webhooks



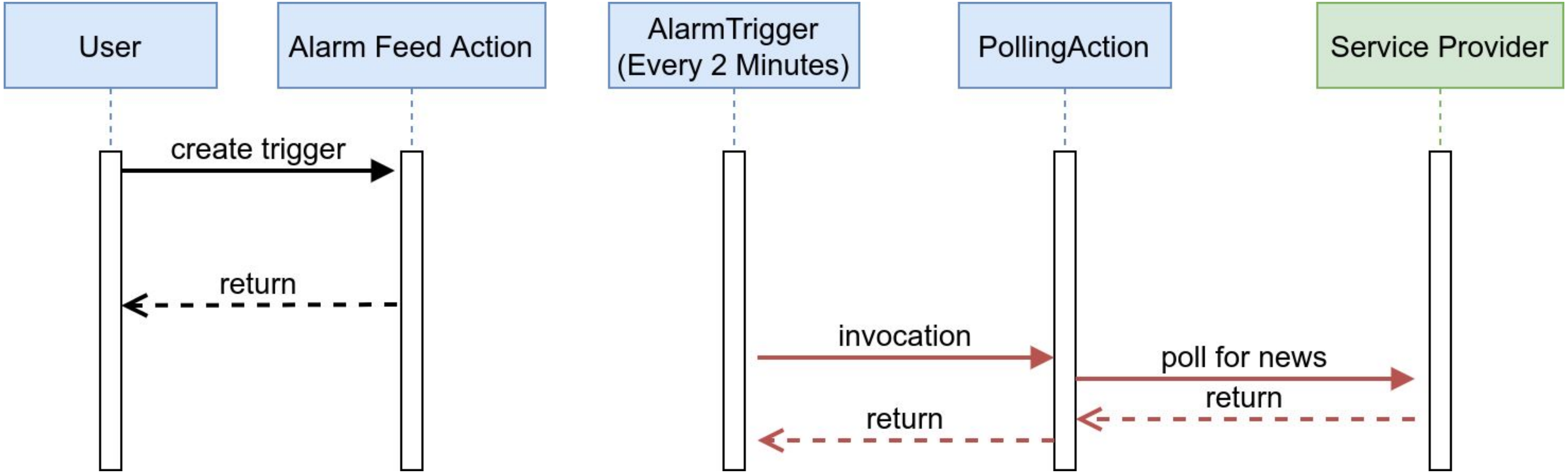
(Creating a trigger is also a usual function invocation of the feed action)
(Triggers are fired through the REST API)

Sources: [19]

3.3 OpenWhisk Event Model

e.g. polling an RSS feed every 2 minutes

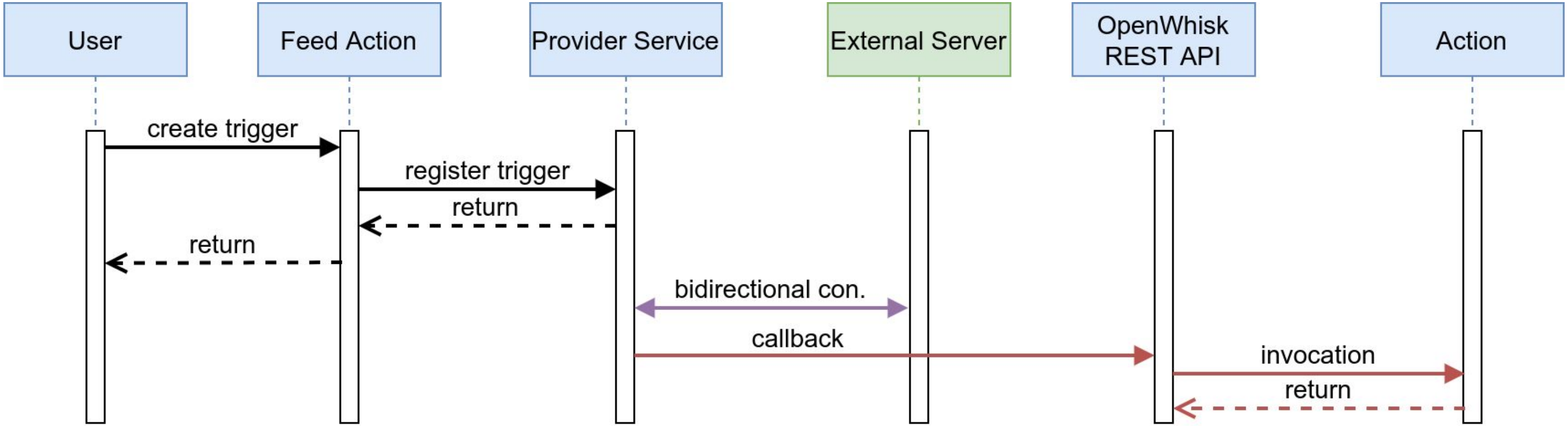
b) Polling



3.4 OpenWhisk Event Model

c) Connections Pattern

when a persistent connection is necessary
or performance matters
(e.g. subscribing to a mqtt topic)



Conclusion: Bridging arbitrary protocols to OpenWhisks REST-Only API is possible!

(Alarm Feed == Provider Service without External Server)

Sources: [19]

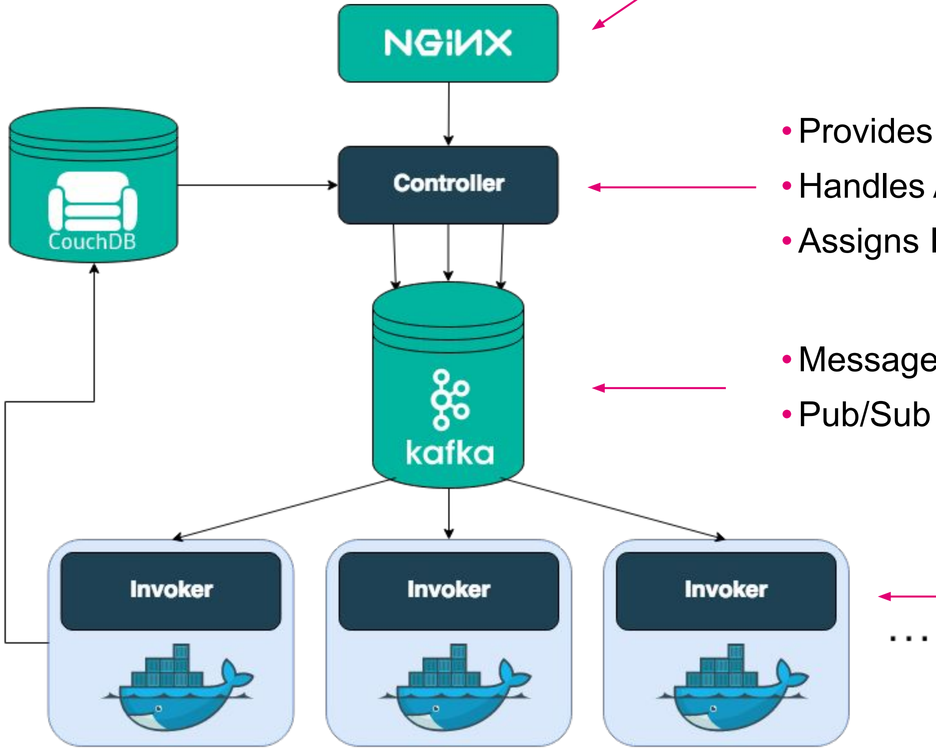
4. Architecture

for OpenWhisk and FaaS in general

4.1 OpenWhisk Architecture

- Reverse Proxy
- Terminates TLS connection

- Stores action code
- Stores results
- Stores configuration
- Stores limitations
- Stores logs
- Stores credentials for authentication



- Provides REST API (web + administration)
- Handles Authentication and Authorization
- Assigns Invoker per Request (Load Balancer)

- Message Queue buffering invocations
- Pub/Sub protocol

- Starts runtime containers
- Initializes runtimes with action code

Sources: [4,20]

4.2 Relationship between FaaS and container orchestration

- Scaling the number of runtime containers requires interfacing with container orchestration
- Most Self-Hosted serverless software supports kubernetes natively

FaaS Software	Kubernetes Support	Helm Chart available
OpenWhisk	✓	✓
Fn	✓	✓
Kubeless	✓	✓
OpenFaas	✓	✓

→ Other supported platforms are highly implementation dependent

5. Networking and Edge

How serverless runtimes can be reached

5.1 Task Sharing

Controlled by developer

- API Endpoints
- Code

Controlled by kubernetes

- Communication between pods
- Ingress (TLS termination may be done by OpenWhisk)
- Pod scheduling

Controlled by OpenWhisk
(default setup)

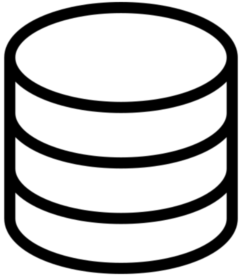
- TLS Termination
- Load Balancing
- Automatic Scaling

→ the lower layer networking happens an abstraction layer below OpenWhisk

5.2 Relationship between Edge Computing and FaaS

Functions have small size

The data a function operates on may be a lot bigger



Conclusions

- Distributing FaaS-functions world-wide is comparatively easy
- Moving data world-wide may be hard

The idea: Edge Computing

- Process data where it is generated and/or needed

Sources: [11]

5.3 Different Kinds of Lambda Functions in AWS

Amazon AWS Lambda

- General Purpose Functions

Use-Cases

- Mobile Backends
- Single Page Applications
- Cloud Based Cron Jobs

Location

- Single AWS Region

Amazon AWS Lambda@Edge

- General Purpose Functions

Location

- World Wide = All Regions

Amazon CloudFront Functions

- Small Functions (< 10KiB code size)
- execution on a per request basis

Use Cases

- Deliver different files based on User-Agent header
- Optimize Caching

Location

- World Wide = All Regions

6. Databases, Storage, APIs

Properties of external APIs

6.1 AWS Aurora vs. Aurora Serverless

Amazon AWS Aurora

- Relational Database
- Drop-In Replacement for MySQL and PostgreSQL

AWS Aurora - Provisioned

Billing:

- storage (**per GiB/month**)
- I/O rate (**per 1 Mio req**)
- instance size **(per hour)**

(e.g. db.t3.medium = 0,082 USD/h)

AWS Aurora - Serverless

Billing for

- storage (**per GiB/month**)
- I/O rate (**per 1 Mio req**)
- Aurora Capacity Unit (**per hour**)

(1 ACU ~ 2GiB Memory usage)

Differences in Serverless

Different billing
+ Autoscales up and down
+ Stateless http "Data API"

Sources: []

6.2 FaaS suitable database and storage APIs

Properties of database APIs suitable for use in Serverless Computing

- Pay-per-use
- Stateless
- Fast (AWS bills idling)
  - No expensive handshakes
  - Sacrifice database normalization for speed
- Automatic Scaling

→ *Same properties like those of FaaS functions!*

Examples of suitable APIs (if pay-per-use)

- NoSQL databases with RESTful API
  CouchDB , MongoDB
- Object Storage with RESTful API
  Amazon S3 , MinIO , Ceph
- E-Mail (SMTP)
  Amazon Simple Email Service
- Publish/Subscribe pattern based Protocols

Examples of not suitable APIs

- Databases with stateful connection
  MySQL , MariaDB

Sources: [21]

6.3 Mounted Filesystems via Volumes

Volumes in Serverless Computing

- Mounted Storage Volumes are usually not supported
- Need to use external APIs for any storage needs

(Notable Exception: Amazon Elastic File System storage is mountable in AWS Lambda)

Still possible: Using temporary files

- Saving and Reading files is possible as usual
- Files are not guaranteed to exist until next execution

<u>Use Cases</u>

- Caching of downloaded files or previous results → warmer warm start

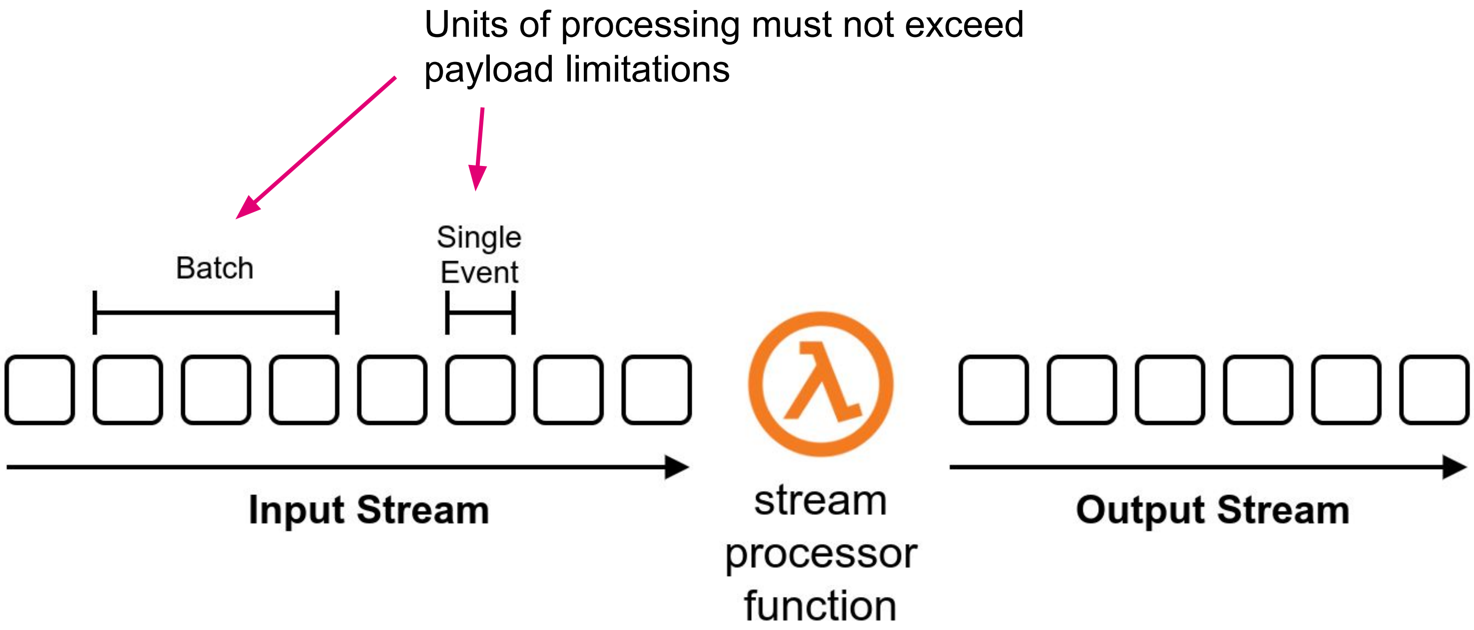
Sources: [7]

7. Stream processing

Properties and integration with FaaS

7.1 Messaging and event streaming

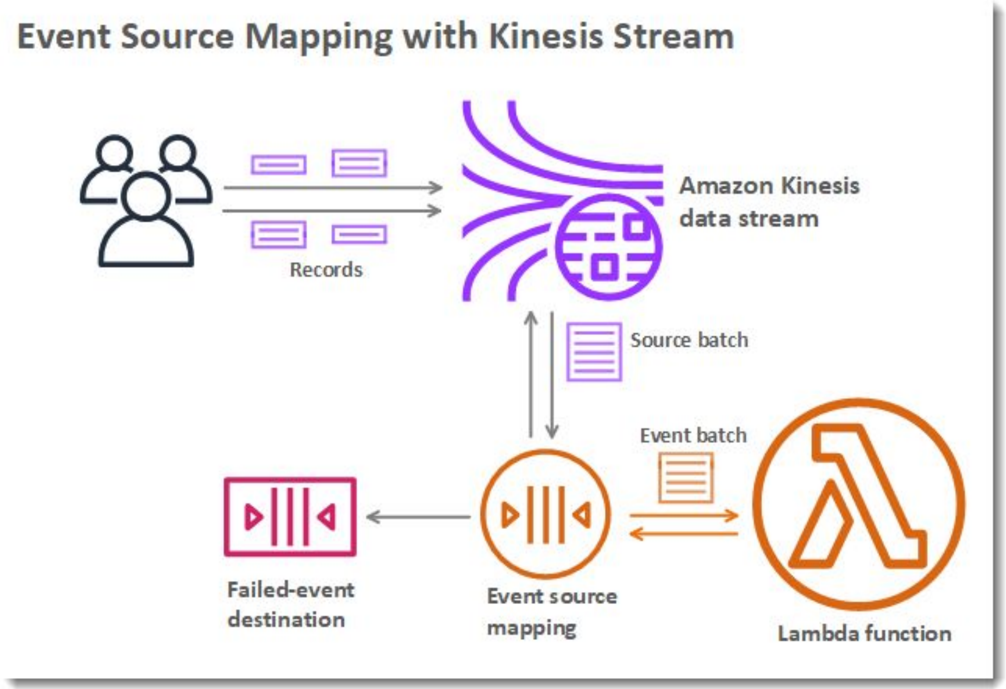
Messaging and event stream processing with FaaS-functions are possible



Sources: [23]

7.2 Messaging and event streaming - AWS Lambda implementation

Event source mapping in AWS Lambda



1. AWS polls new items periodically
2. Lambda function gets batch as input

On Error, the batch will be sent to separate failed-event queue

OpenWhisk feed implementation for Kafka also uses batching!

7.3 Consequences of event source mapping

Features distinguishing messaging and event streaming software:

Support in event source mapping + Serverless

Event history

- Is it possible to consume events from the past?

⇒ No, polling is not under control of the function

Transactional behaviour

- If using transactions following ACID paradigm is possible

```
Example:
Start Transaction
Consume msg1 from Topic1
Publish msg1 to Topic2
Commit transaction
```

⇒ As consumer: No
As producer: Yes

Sources: [21,23]

7.4 Consequences of event source mapping

Features distinguishing messaging and event streaming software:

Support in event source mapping + Serverless

Fine-grained subscriptions

- can topic names be hierarchical?
  topic-only: `dataStreamX`
  hierarchical: `iot/sensors/temperature`

→ Yes, handled by the feed implementation

Scalable number of consumers and messages

- If 10 consumers can be scaled to 100000 consumers easily

→ Scaling consumers:
  Yes, by starting more function containers

Scaling number of messages:
  Only if batch size does not exceed payload limits

Sources: [21,23]

8. Performance

Cold Start Delay and traffic rates

8.1 Overhead

Overhead is inevitable

- Queueing in Kafka
- CouchDB communication
- Cold and Warm start delays
- Function calls to ext. APIs

User latency perception

- **Up to 0.1 seconds:** : The user does not recognize any perceptible delay. Its behavior has an immediate effect.
- **Up to 1 second:** The delay is slightly perceptible, but the flow of work is not interrupted.
- **Up to 10 seconds:** The user must wait and must therefore actively maintain his or her concentration.

```
~> wsk --insecure activation list
Datetime            Activation ID                    Kind      Start Duration   Status            Entity
2021-05-31 09:26:22 1bf8d5ccf2874772b8d5ccf287277211 python:3  cold  2.01s      developer error   guest/flask:0.0.1
2021-05-31 09:26:21 e83ab9c972fd453abab9c972fd753adf python:3  cold  2.18s      developer error   guest/flask:0.0.1
2021-05-31 09:26:17 0997bc70a7dc403797bc70a7dcf0371d nodejs:10 cold  83ms       success           guest/hello:0.0.1
2021-05-29 18:11:35 4a0760d47cf0433d8760d47cf0e33d5b nodejs:10 cold  945ms      application error guest/forecast:0.0.1
2021-05-06 14:51:26 0c7be97bc3f94b0abbe97bc3f9eb0a32 nodejs:10 cold  34ms       success           guest/hello:0.0.1
2021-05-06 02:09:19 b6157a905de14a76957a905de17a7633 nodejs:10 warm  4ms        success           guest/hello:0.0.1
2021-05-06 02:09:09 00c4295252db476484295252dbf764b5 nodejs:10 warm  7ms        success           guest/hello:0.0.1
2021-05-06 02:08:16 77d69680b96f45bc969680b96f45bc4f nodejs:10 cold  26ms       success           guest/hello:0.0.1
2021-05-06 02:02:17 e9d27162f69c40c8927162f69ce0c89c nodejs:10 warm  3ms        success           guest/hello:0.0.1
2021-05-06 02:02:12 d0714748514b462cb14748514b462c74 nodejs:10 warm  4ms        success           guest/hello:0.0.1
2021-05-06 02:02:12 b4e73a8b816045d8a73a8b816005d8a7 nodejs:10 cold  23ms       success           guest/hello:0.0.1
2021-05-06 02:02:10 c9e548bae35f4f61a548bae35f5f61fb nodejs:10 cold  32ms       success           guest/echo:0.0.1
```

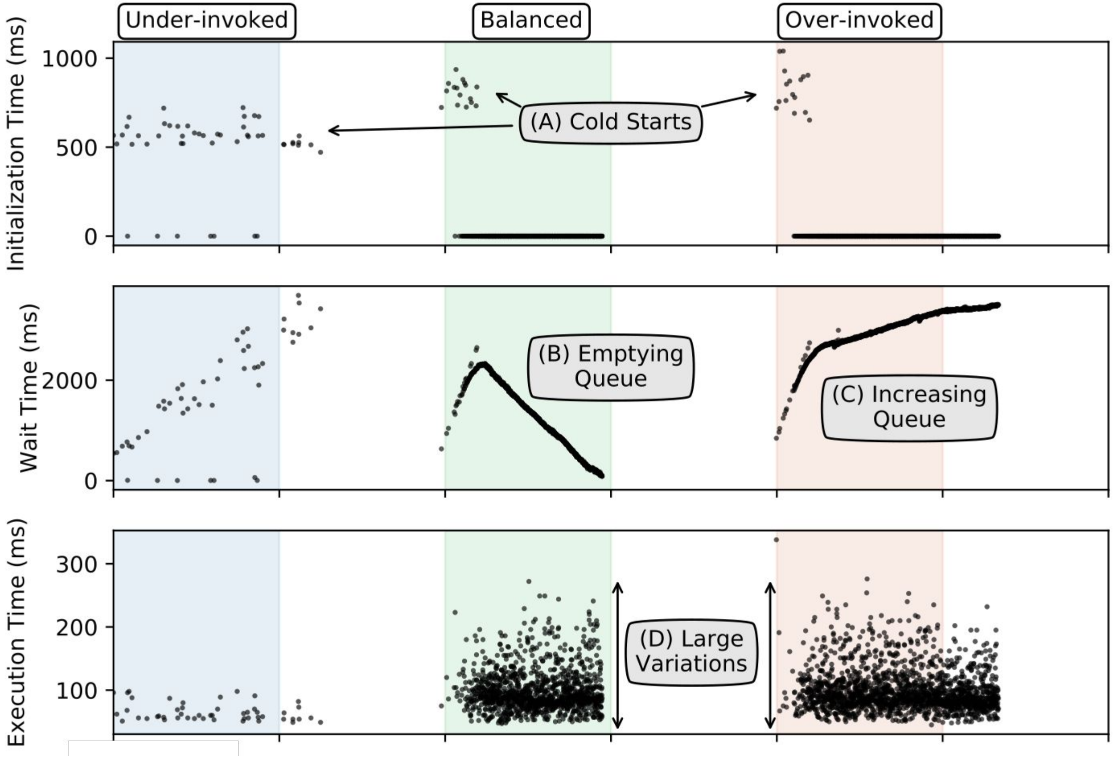
Conclusion: FaaS is <u>in general</u> usable for websites, even on cold start

Sources: [18]

8.2 Cold Start Delay vs. amount of requests

- **Initialization Time** = Runtime container start time
- **Wait Time** = Wait for execution in queue inside OpenWhisk
- **Execution Time** = Actual function execution

- Runtime will be shut down again after idling for some time (for example 45-60 min in AWS Lambda)
- OpenWhisk pauses containers to save memory



8.3 Prewarming

OpenWhisks invoker launches prewarm containers to accelerate cold starts

Pods

	Name	Labels	Node	Status
✅	wskowdev-invoker-00-6-prewarm-nodejs10	invoker: invoker0 name: wskowdev-invoker-00-6-prewarm-nodejs10 release: owdev user-action-pod: true	minikube	Running
✅	wskowdev-invoker-00-1-prewarm-nodejs10	invoker: invoker0 name: wskowdev-invoker-00-1-prewarm-nodejs10 release: owdev user-action-pod: true	minikube	Running
✅	owdev-invoker-0	app: owdev-openwhisk chart: openwhisk-1.0.0 controller-revision-hash: owdev-invoker-5cd96b87db heritage: Helm name: owdev-invoker Alles anzeigen	minikube	Running

Sources: [4]

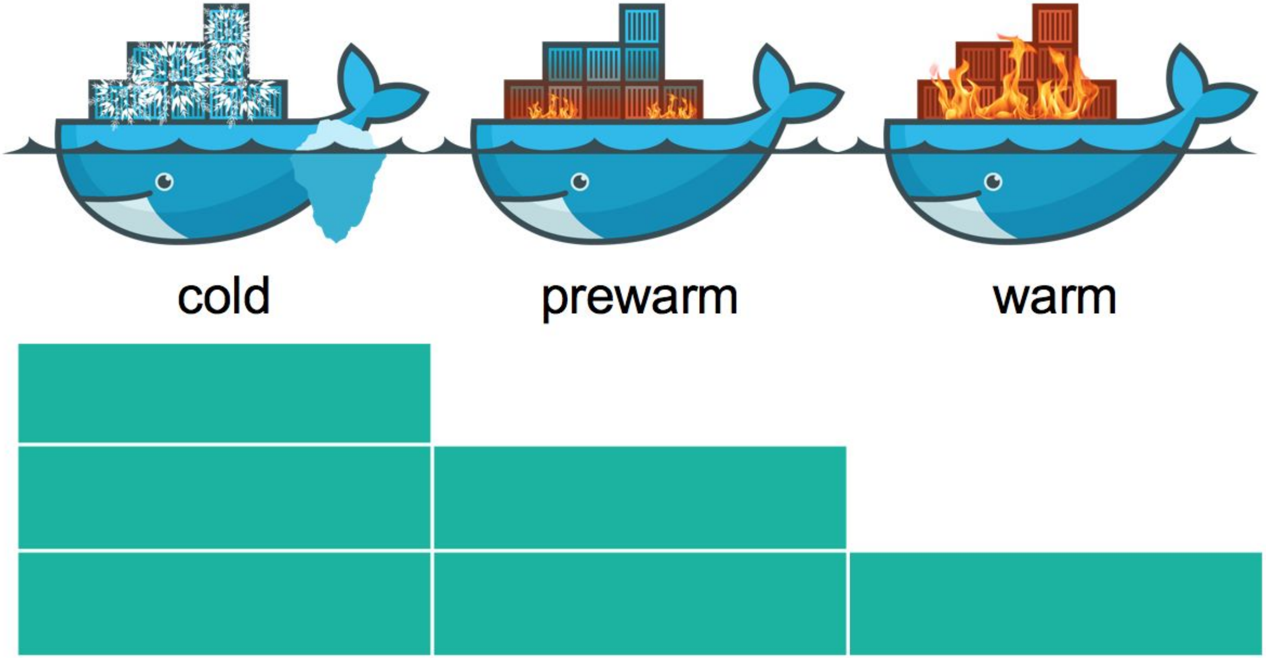
8.4 Cold vs. Prewarm vs. Warm

What needs to be done?

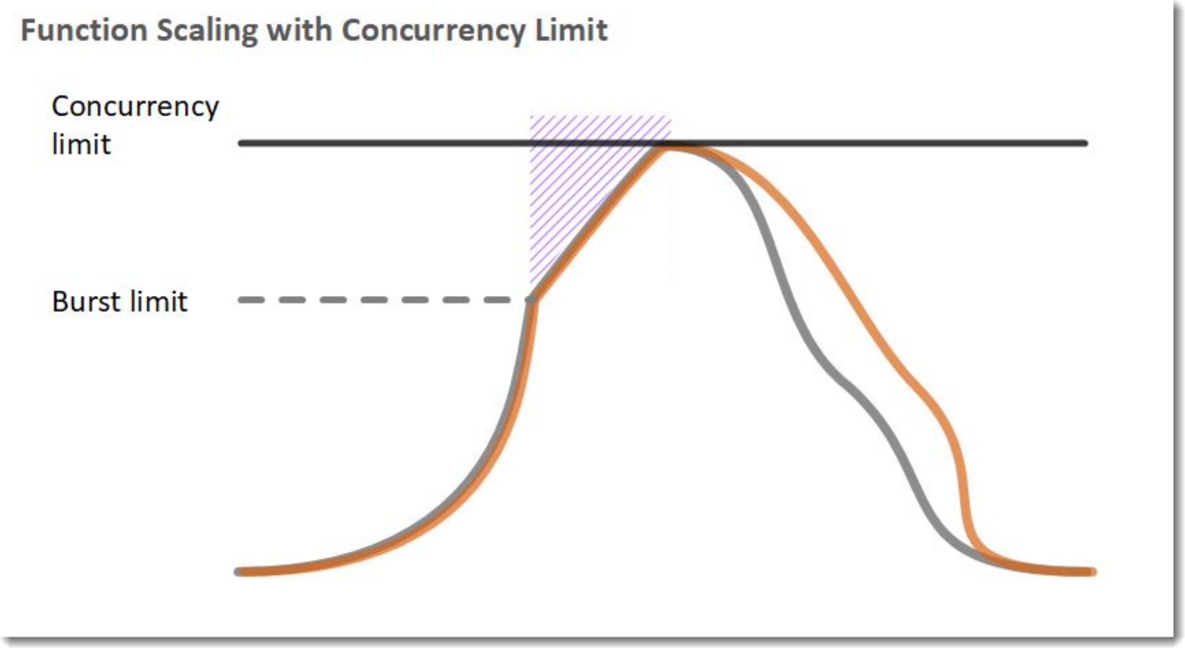
Starting the container

Initializing the action

Running the action



8.5 Scaling in AWS Lambda



Initially

- Initial burst limited to region dependent burst concurrency quota
  (US and EU: 3000)

Then

- 500 new instances each minute

Until

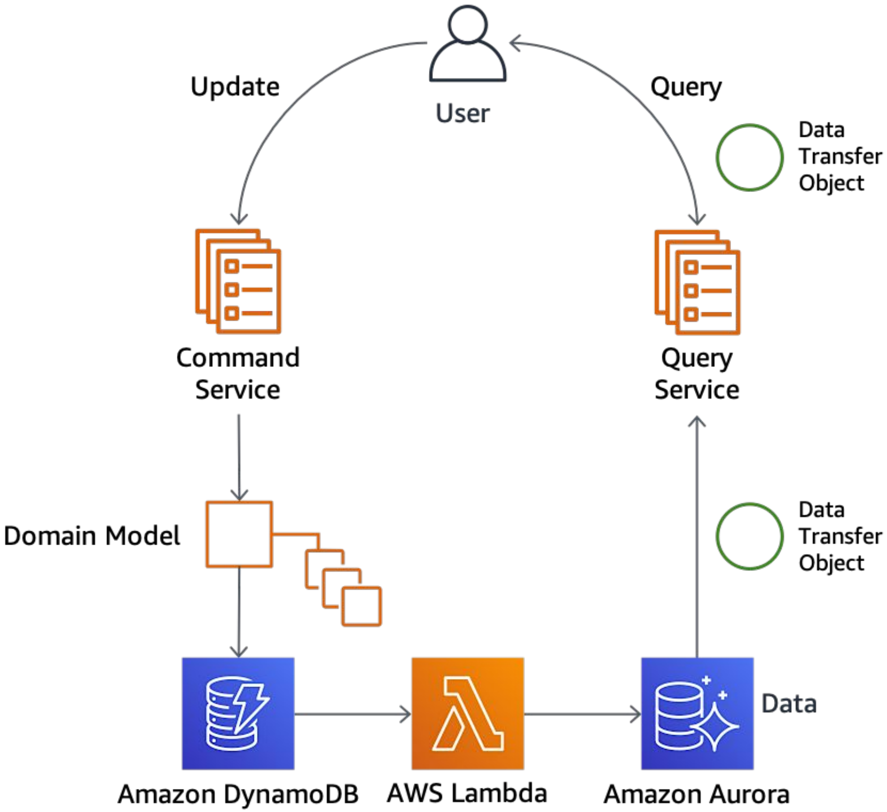
- configurable concurrency limit is reached

Sources: [24]

9. Software Development

Using FaaS in programming

9.1 Command-Query-Responsibility-Segregation



Idea

- Different models for reading and writing data

<u>Command Model</u>

- Scalable and schema-less database for creating and updating data fast

<u>Query Model</u>

- Relational database, normalized with schema for complex queries

Lambda function is triggered on each update and translates between the two models!

10. Function orchestration and keeping state

Workflows and Actors for keeping state and splitting work

10.1 Synchronous and asynchronous invocations

Synchronous Invocation

- Blocks until result is available

<u>Sync Example (OpenWhisk)</u>

```
wsk action invoke \
   /whisk.system/samples/greeting --result --blocking
{
    "payload": "Hello, World!"
}
```

Asynchronous Invocation

- Returns immediately giving an Activation ID
- Client retrieves result later

<u>Async Example (OpenWhisk)</u>

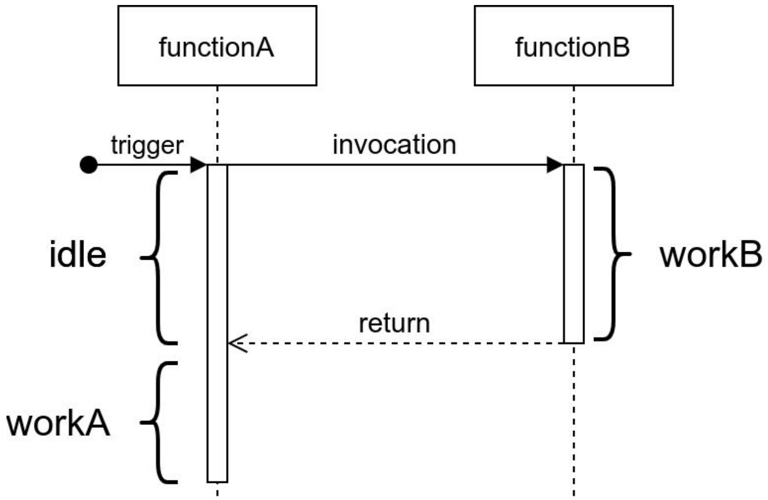
```
wsk action invoke \
   /whisk.system/samples/greeting
ok: invoked /_/pythonfunction with id
733404104295414ab404104295c14ae2
```

-> Both available in OpenWhisk and Amazon AWS Lambda
-> OpenWhisks internals are unaffected
(Queueing, storing results, existence of Activation ID, etc.)

Sources: [12]

10.2 Double billing

If functions invoke other functions synchronously, the work is billed twice



Each call to external APIs also results in billing of idle time!

billed_duration = idle + workA + workB

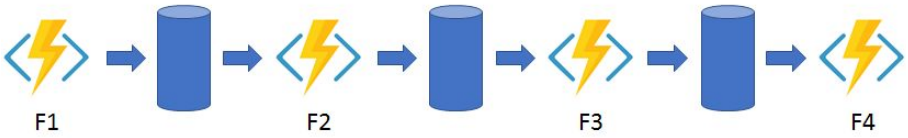
Sources: [26]

10.3 Function Orchestration

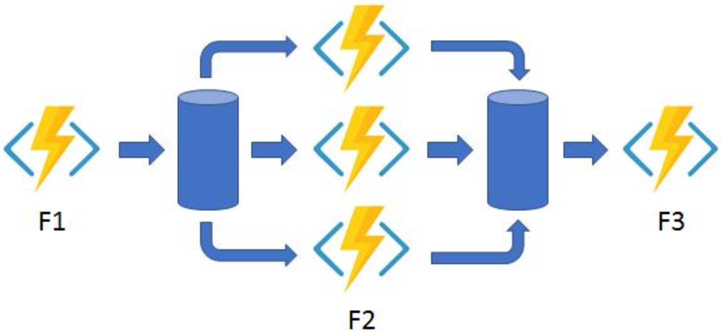
Orchestrate functions based on application patterns

Software and Services:
AWS Step functions
Azure Durable Functions
OpenWhisk Composer

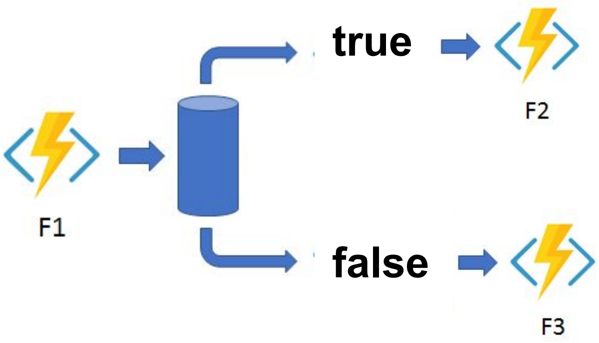
Function Chaining



Fan-Out



Branching

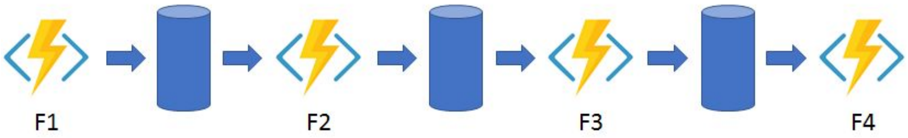


Sources: [27,28]

10.4 Replaying and Checkpoints

Azure Durable functions uses syntax like in async programming!

Function Chaining



orchestrator function

```
[FunctionName("Chaining")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1", null);
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return  await context.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // Error handling or compensation goes here.
    }
}
```

activities

Sources: [28]

10.5 Azure Durable Functions - Replaying and Checkpoints

orchestrator function

```
[FunctionName("Chaining")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1", null);
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return  await context.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // Error handling or compensation goes here.
    }
}
```

activities

How it works:

- orchestrator function is invoked multiple times → replay
- if an activity already happened result is returned immediately → checkpoint from Table Storage

Sources: [28]

10.6 OpenWhisk - Conductor Actions

```
function main(params) {
    let step = params.$step || 0

    switch (step) {
        case 0: return { action: 'actionName', params, state: { $step: 1 } }
        case 1: return { params }
    }
}
```

How it works:

- conductor action is invoked multiple times
- the conductor returns values having special meanings:
    - “action” → action to execute next
    - “state” → state, which will be parameter of next conductor invocation

Sources: [29]

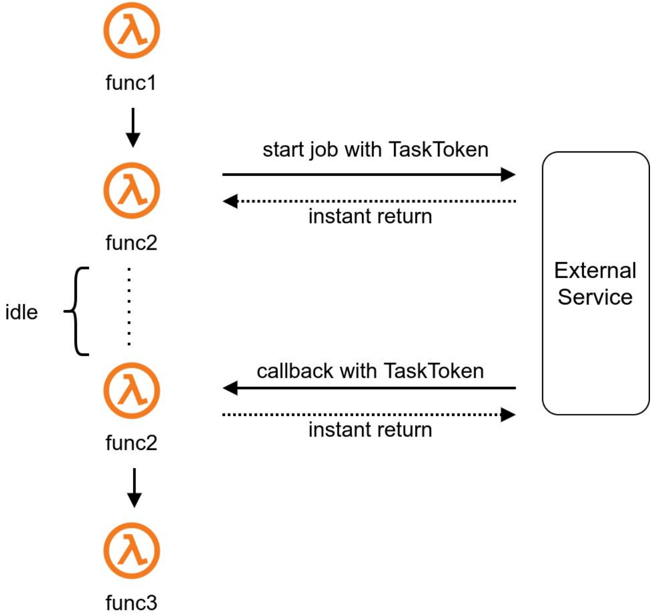
10.7 External Callback

Waiting for an external Event

Examples for events:

- User interaction
- End of asynchronous Job
- Any trigger source

External Callbacks in AWS Step functions:



- idling step functions may be removed from memory
- AWS supports idling for up to **1 year**
  → **long-running workflows**

Sources: [12]

10.8 Serverless Trilemma

Desired properties serverless computing architectures:

1. **Black-Box:** Function should be considered as black-boxex
2. **No Double-Billing:** A function invoking a function should not be billed for idling
3. **Substitution:** A composition of functions should work like a single function

Serverless Trilemma

- In an event driven system (reactive core) one of the rules must be violated

ST-Safe

- Any serverless orchestration system that satisfies all three conditions is called ST-safe

Solution for the Serverless Trilemma

- Sequential Composition

Sources: [37]

10.9 Formal Foundations of Serverless Computing

The formal definition of **operational semantics** for serverless computing can be found in the paper
→ “Formal Foundations of Serverless Computing”

Sources: [38]

10.10 Reduced performance due to missing state and placement control

Missing state

- GPU accelerated FaaS functions need to copy data to and from GPU's memory on each invocation
- Cold start delay is direct result of having no state
  - additional overhead through loading of additional libraries on cold start

Missing placement-control

- More communication needed
  - Transfer required data from external source

Sources: [33]

10.11 Solutions for keeping state

Software and Services:

AWS Step functions

Azure Durable Functions

OpenWhisk Composer

Solutions (i found so far):

- Function Orchestration via Workflows
  - Workflows keeping a state like State Machines
  - Workflows forwarding a state on each invocation
- Externalize state
  - Fast NoSQL databases
  - “State as a Service”
  - (Is often implicitly the solution applied!)
- Persistent Entities (or Actors)
- Parameterizing proxy

redis

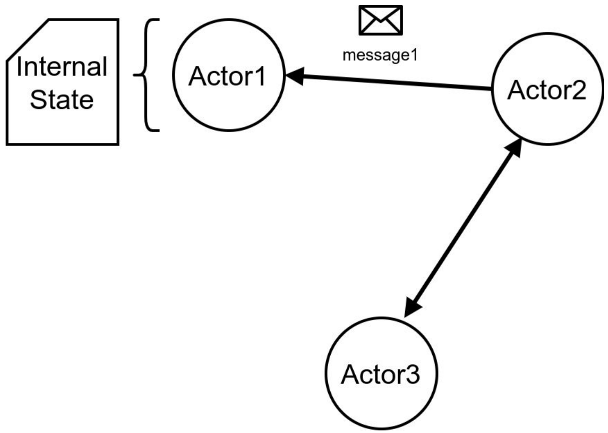
CouchDB

Memcached

Azure durable entity functions

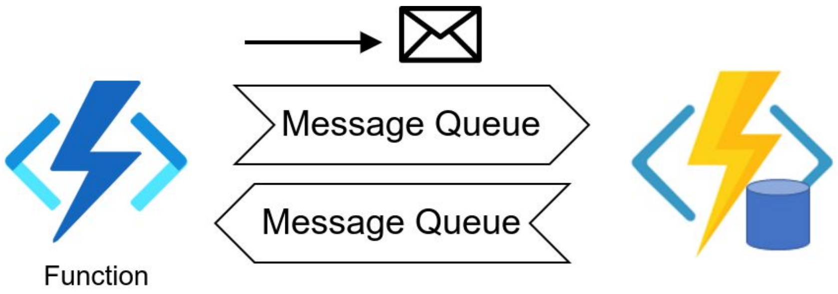
10.12 Entities and Actors

Actor Model



- Actor1 has an internal state
- Access to that internal state only via messages

Entity functions in Azure Durable Functions



- Entity functions have persistent state
- Access to internal state only via messages
- Messages trigger entity functions in an event-driven way

Ways to communicate

Signaling: fire-and-forget

Call: blocking call, waiting for response

10.13 Externalizing state

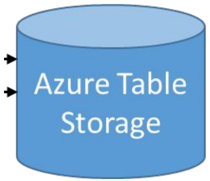
Externalizing state

- Saving and restoring state in an external database, file-system or storage

Storing the state outside of the function is always necessary

- State always needs to be somewhere if the functions and entity functions are supposed to shut down

Azure Durable Functions is Open Source and defines a table storage for persisting the data in its specification

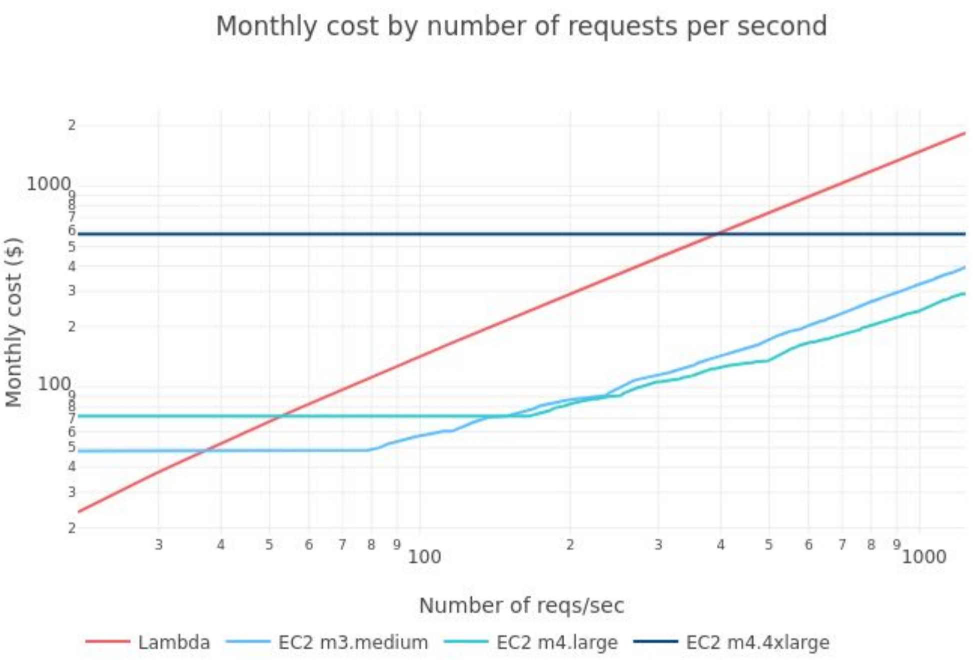


11. Economic View

Cost, need for operations and suitable traffic shapes

11.1 Exemplary cost calculation - provisioned vs serverless

AWS Lambda vs AWS EC2 Instances



In this specific example
→ A high number of reqs/sec makes FaaS uneconomic

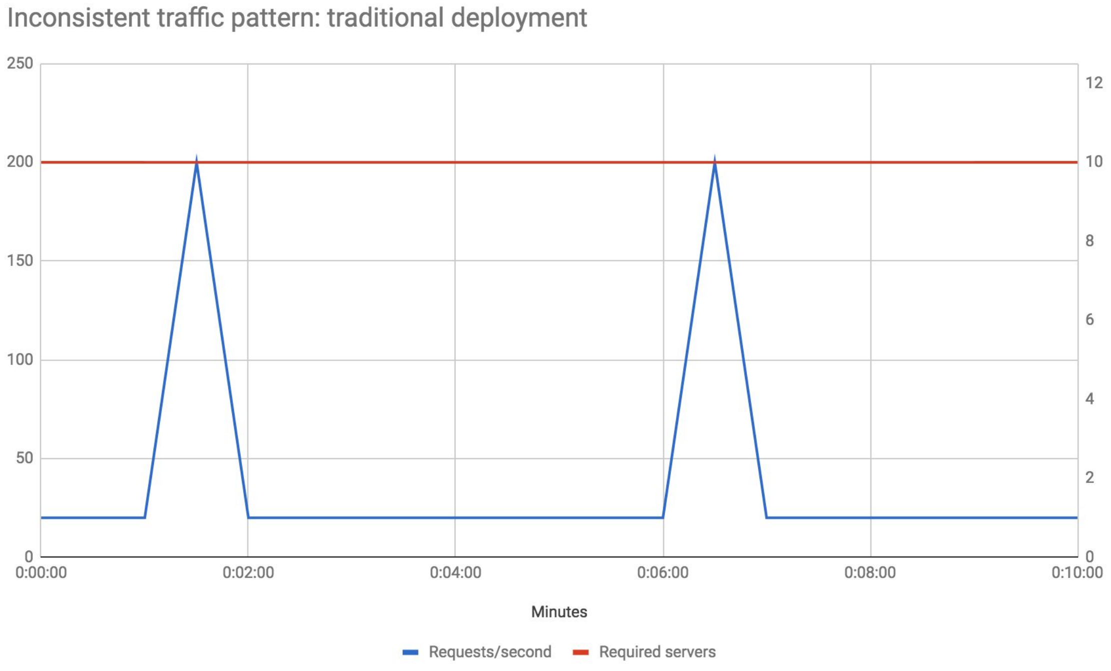
In general, advantageous cases for:

Provisioned instances
- Predictable load
  → tight sizing possible

AWS Lambda functions
- Low amount of reqs/sec
- High idle time
- Unpredictable load
  → no sizing necessary beforehand

11.2 Cost effective traffic shapes

Serverless computing is cost effective and scales well on bursty traffic



Sources: [6]

11.3 Predicting cost is hard in AWS Lambda

(Prices for EU Frankfurt)

Memory (MB)	Price per 1ms
128	$0.0000000021
512	$0.0000000083
1024	$0.0000000167
1536	$0.0000000250
2048	$0.0000000333

execution_time_price

+ price for number of requests

+ traffic price inbound and outbound a region

+ other services used

	Price
Requests	$0.20 per 1M requests

	Pricing
Data Transfer IN To Amazon EC2 From Internet	
All data transfer in	$0.00 per GB
Data Transfer OUT From Amazon EC2 To Internet	
Up to 1 GB / Month	$0.00 per GB
Next 9.999 TB / Month	$0.09 per GB
Next 40 TB / Month	$0.085 per GB
Next 100 TB / Month	$0.07 per GB
Greater than 150 TB / Month	$0.05 per GB

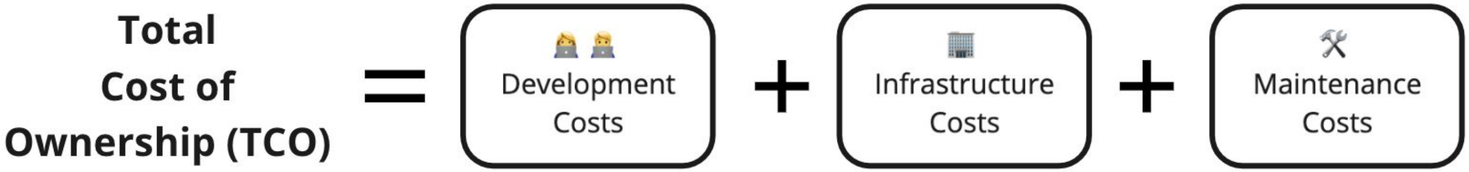
For example: S3 Object Storage

- GB stored per month
- per 1000 requests
- Replication

Price prediction utilities exist, but need precise data

Sources: [30,31]

11.4 Total cost of ownership



reduced

- developers need to consider less low level details

reduced or increased

- depending on use case and traffic shape

reduced

- operations still necessary
- but a lot of tasks shifted to the cloud

Sources: [34]

11.5 Operations when using FaaS

Does Serverless computing mean we don’t need an operations team? -> **No**

FaaS software still requires operations:

- Monitoring
- Logging
- Backup and Recovery strategy
- IT-Security (Intrusion detection, etc.)
- ...

But usually cloud provided:

- Firewall
- Reverse Proxy
- Load Balancers
- Cluster and Nodes
- ...

→ The cloud provider itself also needs sysadmins

Sources: [3]

12. Conclusion

Advantages and Disadvantages

12.1 Pro and Contra of using FaaS

+

- Operations Expenditure reduced
- Faster development time
- Theoretically infinite scalability
- Bursty traffic shapes can be handled
- High efficiency and hardware utilization

−

- Traffic spikes may cause cost explosions
- Cost prediction difficult
- Developers needs to handle tight execution time and resource limits
- Existing architectures need to be redesigned

Sources: [15]

12.2 Pro and Contra of using FaaS in the cloud

+

- No permanently running servers
- High scalability
- Pay per Use
- High efficiency
- Lower CapEx

−

- Traffic spikes may cause cost explosions
- Vendor Lock-In
- OpEx not necessarily cheaper

Sources: [15]

12.3 Opinions on technology readiness

Opinion on technology readiness (limited to AWS Lambda and OpenWhisk)

General

- Performance seems to be sufficiently optimized
  - Prewarming, etc.
- Technology is old enough for being sufficiently stable
  - AWS Lambda released 2014
- Ecosystem underdeveloped
  - No interoperability (like OCI with containers)
  - Availability of Ready-to-Use event sources and extensions limited
- Need for stateful computation not sufficiently solved

OpenWhisk

- No easy sharing of packages

AWS Lambda

- Easy sharing of packages only inside AWS Lambda

12.4 Problems of this presentation

- Some aspects would need more good measurements from practical applications, going by a few examples is not sufficient
  - comparing FaaS cost vs. provisioned
  - practical view on cold-start- and warm-start-delay

13. Sources (1 / 5)

01. Jacobs, S. (14. Februar 2018). Blog.oio.de. Abgerufen am 9. Juni 2021 von https://blog.oio.de/2018/02/14/function-as-a-service/

02. Batschinski, G. (kein Datum). Back4App. Abgerufen am 09. 06 2021 von https://blog.back4app.com/baas-vs-faas/

03. GitHub. (19. Februar 2020). Abgerufen am 9. Juni 2021 von https://github.com/apache/openwhisk-runtime-python/blob/master/core/python3ActionLoop/lib/launcher.py

04. OpenWhisk. (2016). Abgerufen am 9. Juni 2021 von https://openwhisk.apache.org/documentation.html

05. Rooms, B. D. (6. Februar 2020). fauna. Abgerufen am 9. 6 2021 von https://fauna.com/blog/comparison-faas-providers

06. Roberts, M. (22. Mai 2018). martinFowler.com. Abgerufen am 9. Juni 2021 von https://martinfowler.com/articles/serverless.html

07. Sciabarrà, M. (2021). Oreilly. Abgerufen am 9. Juni 2021 von https://learning.oreilly.com/library/view/learning-apache-openwhisk/9781492046158/ch01.html

08. Amazon (Hrsg.). (kein Datum). aws. Abgerufen am 9. Juni 2021 von https://aws.amazon.com/de/step-functions/?step-functions.sort-by=item.additionalFields.postDateTime&step-functions.sort-order=desc

09. Amazon (Hrsg.). (kein Datum). *aws*. Abgerufen am 9. Juni 2021 von https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html

10. Winder, P. (9. Juni 2021). *W-Winder*. Von https://winderresearch.com/a-comparison-of-serverless-frameworks-for-kubernetes-openfaas-openwhisk- abgerufen

13. Sources (2 / 5)

11. *Digital Guide IONOS*. (27. September 2020). Abgerufen am 9. Juni 2021 von https://www.ionos.de/digitalguide/server/knowhow/edge-computing-erklaerung-und-definition/

12. Shahrad, M., Balkind, J., & Wentzlaff, D. (2019). *Architectural Implications of Function-as-a-Service Computing.* Princeton: Columbo.

13. Rodríguez, Á. A. (9. Dezember 2020). *BBVA*. Abgerufen am 9. Juni 2021 von https://www.bbva.com/en/economics-of-serverless/

14. Amazon (Hrsg.). (.). *aws*. Abgerufen am 9. Juni 2021 von https://docs.aws.amazon.com/lambda/latest/dg/invocation-eventsourcemapping.html

15. Jonas, E., Schleier-Smith, J., Sreekanti, V., & Tsai, C.-C. (2019). *Cloud Programming Simplified: A Berkeley View on.* Berkeley.

16. Thömmes, M. (20. April 2017). *Apache OpenWhisk*. Abgerufen am 9. Juni 2021 von https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0

17. Amazon (Hrsg.). (kein Datum). *aws*. Abgerufen am 9. Juni 2021 von https://aws.amazon.com/de/blogs/compute/working-with-aws-lambda-and-lambda-layers-in-aws-sam/

18. Bohl, R. K. (2017). Perceived vs. Actual Loading Time: Create the Impression of a Fast Website. *Ryte Magazine*. Abgerufen am 9. Juni 2021 von https://en.ryte.com/magazine/perceived-vs-actual-loading-time-create-the-impression-of-a-fast-website#:~:text=For%20a%20positive%20user%20experience,in%20less%20than%20one%20second!

T · ·

13. Sources (3 / 5)

19. *GitHub*. (17. Mai 2010). Von https://github.com/apache/openwhisk/blob/master/docs/feeds.md.Abgerufen am 9. Juni 2021

20. GitHub. (17. Mai 2010). Von https://github.com/apache/openwhisk/blob/master/docs/about.md.Abgerufen am 9. Juni 2021

21. Andrew Schofield. (09. November 2018). Abgerufen am 23. Juni 2021 von https://medium.com/@andrew_schofield/does-apache-kafka-do-acid-transactions-647b207f3d0e22. Amazon (Hrsg.). (.). *aws*. Abgerufen am 23. Juni 2021 von https://docs.aws.amazon.com/lambda/latest/dg/invocation-eventsourcemapping.html

23. Callum , J., Stanley, K., & Lane, D. (21. April 2021). *IBM*. Abgerufen am 23. Juni 2021 von https://developer.ibm.com/technologies/messaging/articles/difference-between-events-and-messages/

24. Amazon (Hrsg.). (kein Datum). *aws*. Abgerufen am 23. Juni 2021 von https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html

25. Amazon (Hrsg.). (kein Datum). *aws*. Abgerufen am 23. Juni 2021 von https://docs.aws.amazon.com/whitepapers/latest/modern-application-development-on-aws/command-query-responsibility-segregation.html

26. Lahiri, M. (23. Juni 2021). *serverless.com*. Von https://www.serverless.com/blog/understanding-and-controlling-aws-lambda-costs abgerufen

13. Sources (4 / 5)

27. Gillum, C. (23. Dezember 2021). *Microsoft*. Abgerufen am 23. Juni 2021 von https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp

28. https://github.com/Azure/durabletask/wiki/Writing-Task-Orchestrations. (02. Juli 2020). *GitHub*. Abgerufen am 23. Juni 2021 von https://github.com/Azure/durabletask/wiki/Writing-Task-Orchestrations

29. Stewart, H. (02. Februar 2021). *Github*. Abgerufen am 23. Juni 2021 von https://github.com/Azure/durabletask/wiki/Core-Concepts

30. Amazon (Hrsg.). (kein Datum). *aws*. Abgerufen am 23. Juni 2021 von https://aws.amazon.com/lambda/pricing/?nc1=h_ls

31. Amazon (Hrsg.). (kein Datum). *aws*. Abgerufen am 23. Juni 2021 von https://aws.amazon.com/de/s3/pricing/

32. Lahiri, M. (kein Datum). *serverless_blog*. Abgerufen am 23. Juni 2021 von https://www.serverless.com/blog/understanding-and-controlling-aws-lambda-costs

33. Petersohn, D. (08. Mai 2019). *riselob*. Abgerufen am 23. Juni 2021 von https://medium.com/riselab/two-missing-links-in-serverless-computing-stateful-computation-and-placement-control-964c3236d18

13. Sources (5 / 5)

34. Lefèvre, X. (30. Juli 2020). Abgerufen am 23. Juni 2021 von https://medium.com/serverless-transformation/is-serverless-cheaper-for-your-use-case-find-out-with-this-calculator-2f8a52fc6a68

36. Baldini, I. (25. Oktober 2017). *ACM Digital Libary*. Abgerufen am 23. Juni 2021 von https://dl.acm.org/doi/10.1145/3133850.3133855

37. Agarwal, A., Choudhary, C., & Bhagat, S. (2018). *The Serverless Trilemma - Function Composition for Serverless Computing.* Washington: University of Washington .

38. Cornell University. (04. Oktober 2020). Abgerufen am 23. Juni 2021 von https://arxiv.org/abs/1902.05870

39. Storti, B. (09. Juli 2015). *brianstorti.com*. Abgerufen am 23. Juni 2021 von https://www.brianstorti.com/the-actor-model/