Serverless Computing

Author: Richard Clauß
Docent: Prof. Dr. Martin Leischner
Date: 06.12.2022

Hochschule
Bonn-Rhein-Sieg

Agenda

1. Introduction
2. Triggers
3. Interfaces to functions
4. Runtimes
5. Cold-Starts
6. Keeping state
7. Serverless Trilemma
8. Workflows
9. Programming for serverless computing

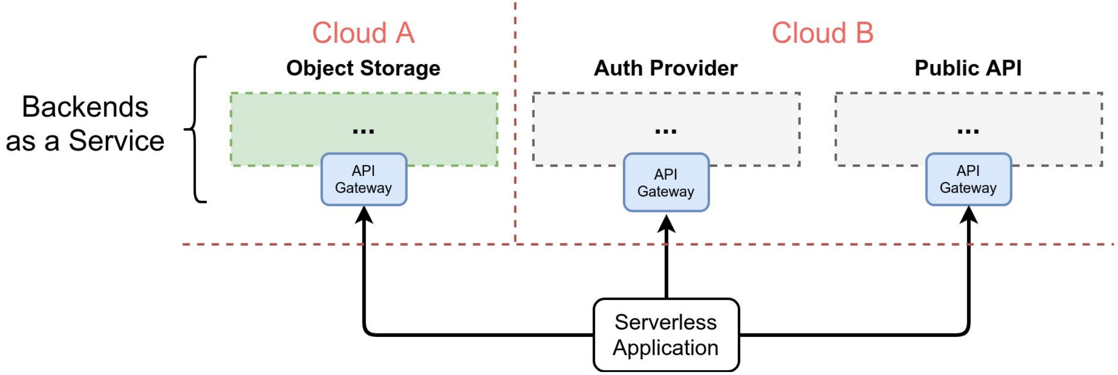
Agenda

10. Suitable workloads and use cases
11. Advantages, disadvantages, learnings

1. Introduction

1.1 What does “serverless” mean?

a) Serverless Architecture



Serverless Architecture

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

Sources: [1]

1.1 What does “serverless” mean?

b) Serverless as cloud-computing execution model

called **“serverless computing”** and
commodified as **“Function as a Service”**

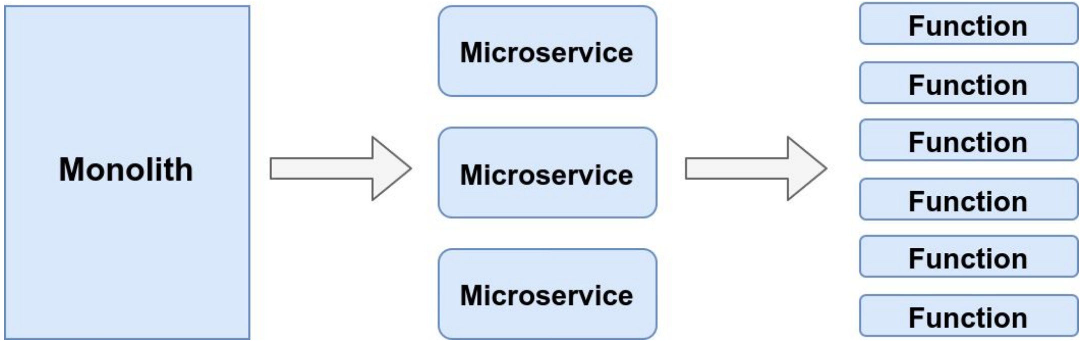
Sources: [1]

1.1 What does “serverless” mean?

b) Serverless as cloud-computing execution model

called **“serverless computing”** and
commodified as **“Function as a Service”**

Are serverless functions just smaller pieces of Microservices?



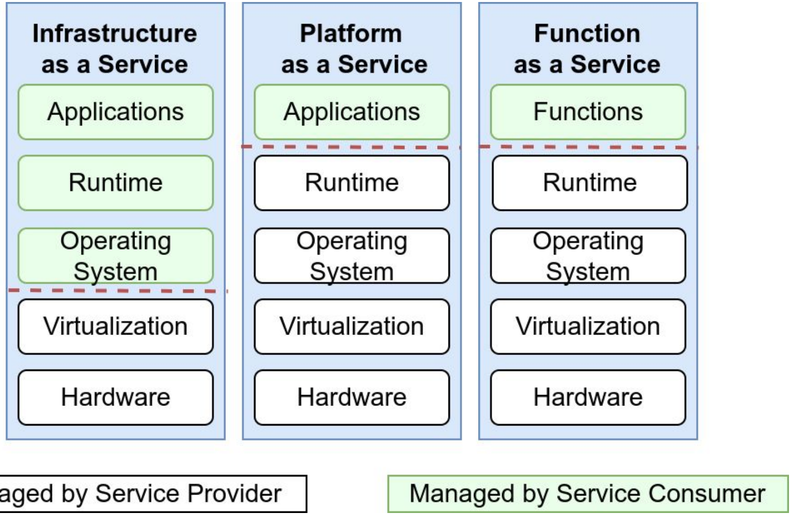
Sources: [1]

1.1 What does “serverless” mean?

b) Serverless as cloud-computing execution model

called **“serverless computing”** and
commodified as **“Function as a Service”**

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



Sources: [1]

1.1 What does “serverless” mean?

Are serverless functions just smaller pieces of Microservices?

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

Yes, serverless functions are usually small fast starting CRI containers,
but the technology in general does not demand that

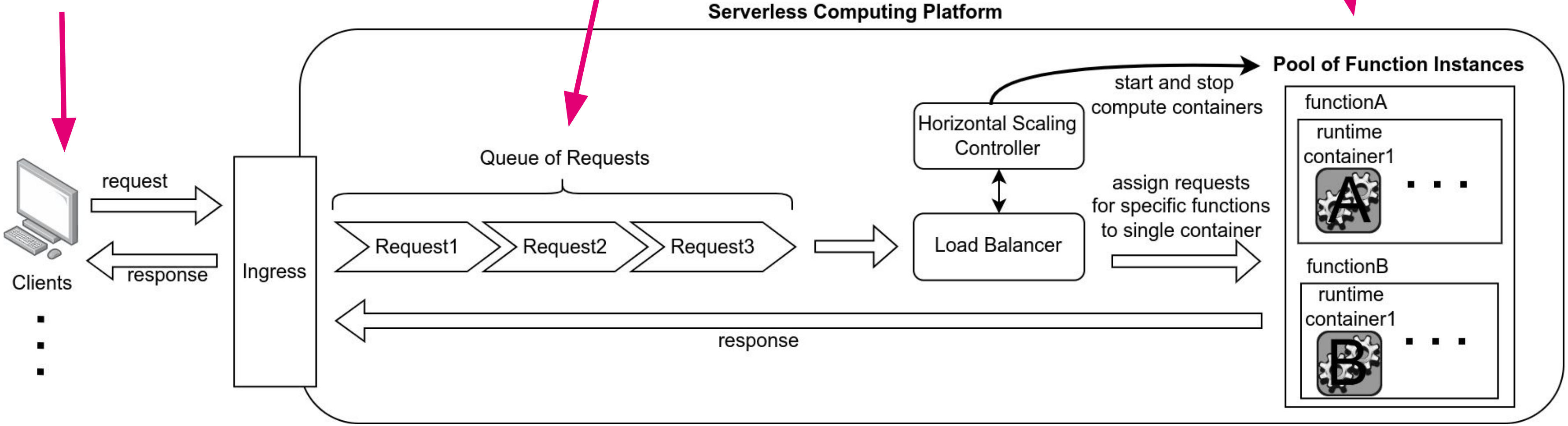
Sources: [1]

1.2 Attempt to find the main properties of serverless computing

Scales transparently, horizontally, automatically and often down to zero

May be kernel packet queue

Different event sources possible



Broadest thinkable serverless computing definition:
“Horizontally automatically scaling programs are executed in response to events”

1.3 FaaS vs. PaaS

How are PaaS and FaaS different if both are containers?

adrian cockcroft
@adrianco

If your PaaS can efficiently start instances in 20ms that run for half a second, then call it serverless.
twitter.com/doctor_julz/st...

Sources: [1]

1.3 FaaS vs. PaaS

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

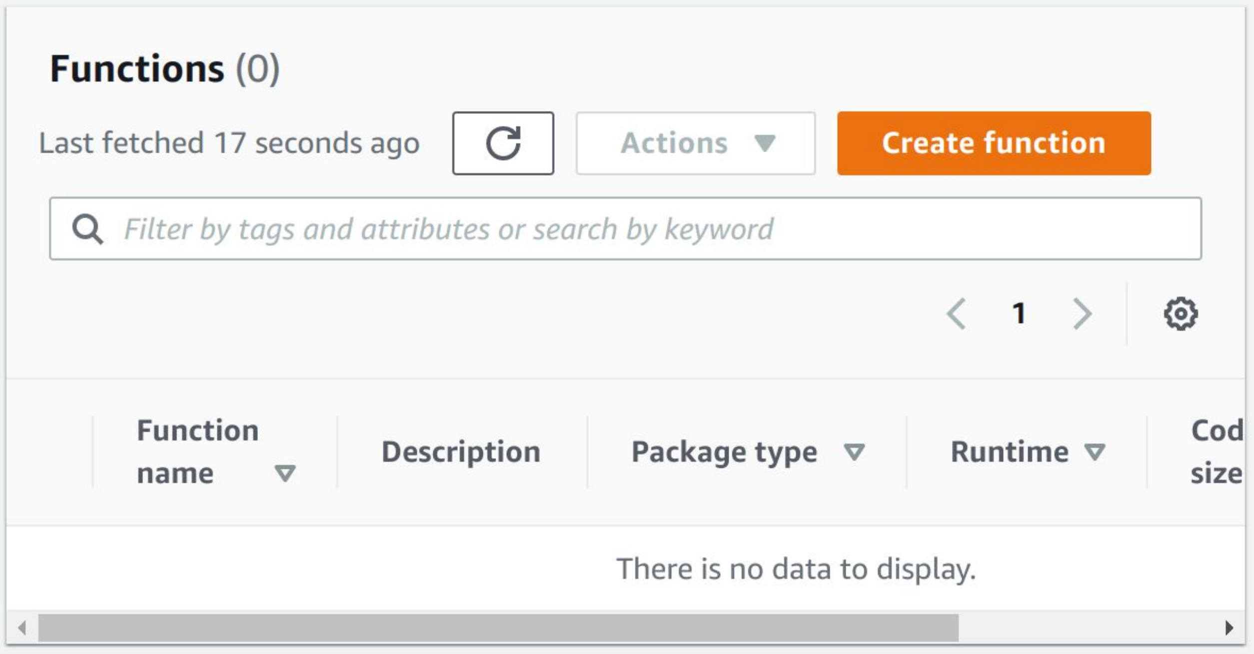
Function as a Service

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

Main benefits of serverless computing:
- More tasks automatically shifted to the cloud compared to PaaS
- Transparent automatic horizontal scaling makes the technology appear “serverless”
- In public clouds finely granular on-demand billing based on milliseconds execution time
- In public clouds no cost because of scale-to-zero

Sources: [1]

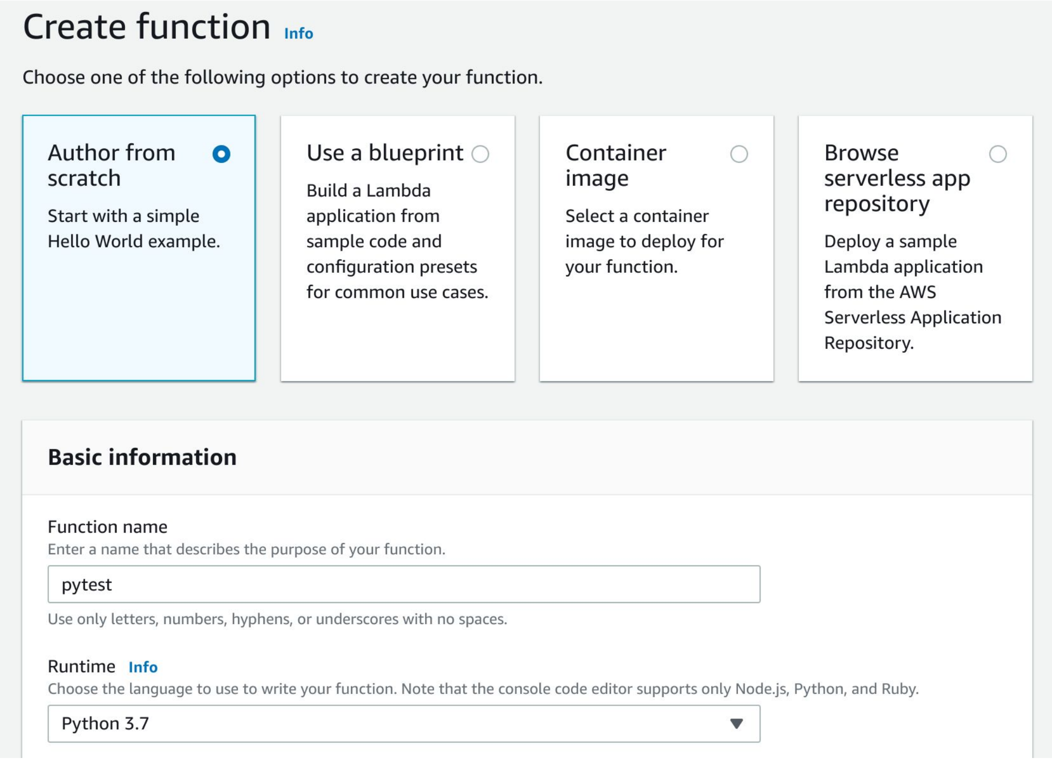
1.4 AWS Lambda Example



Last fetched 17 seconds ago

There is no data to display.

1.4 AWS Lambda Example



Create function

Choose one of the following options to create your function.

- ● **Author from scratch**: Start with a simple Hello World example.
- ○ **Use a blueprint**: Build a Lambda application from sample code and configuration presets for common use cases.
- ○ **Container image**: Select a container image to deploy for your function.
- ○ **Browse serverless app repository**: Deploy a sample Lambda application from the AWS Serverless Application Repository.

Basic information

Function name

Enter a name that describes the purpose of your function.

pytest

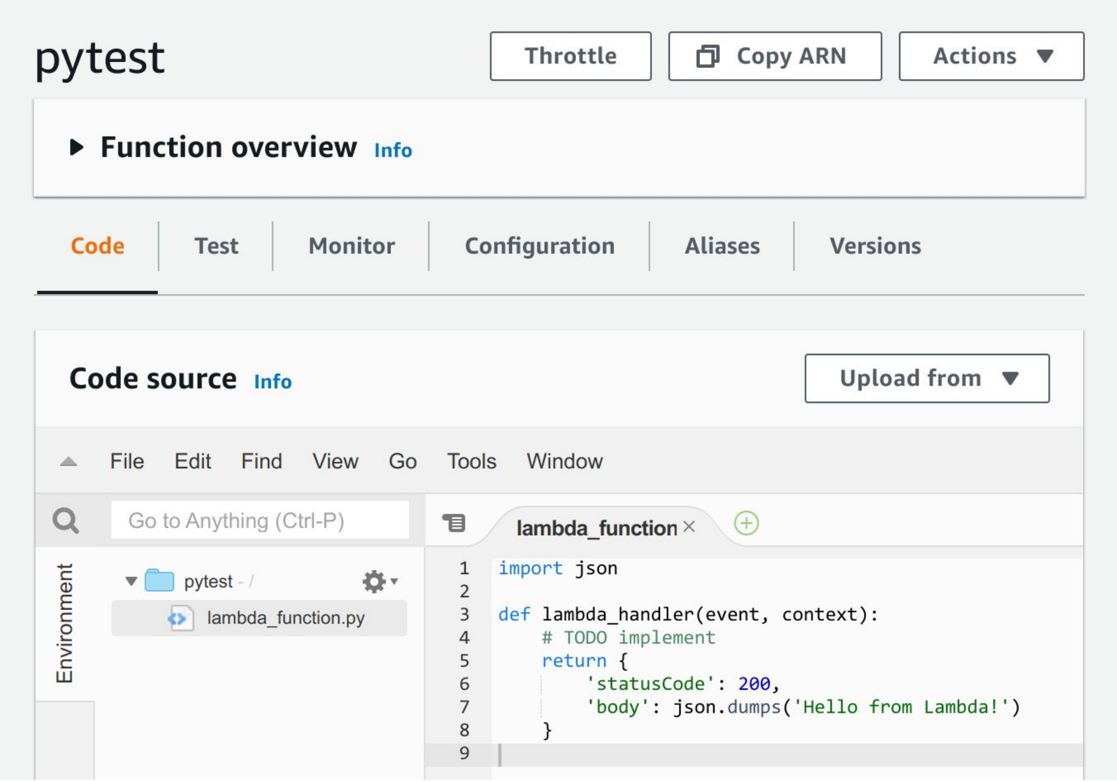
Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime

Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

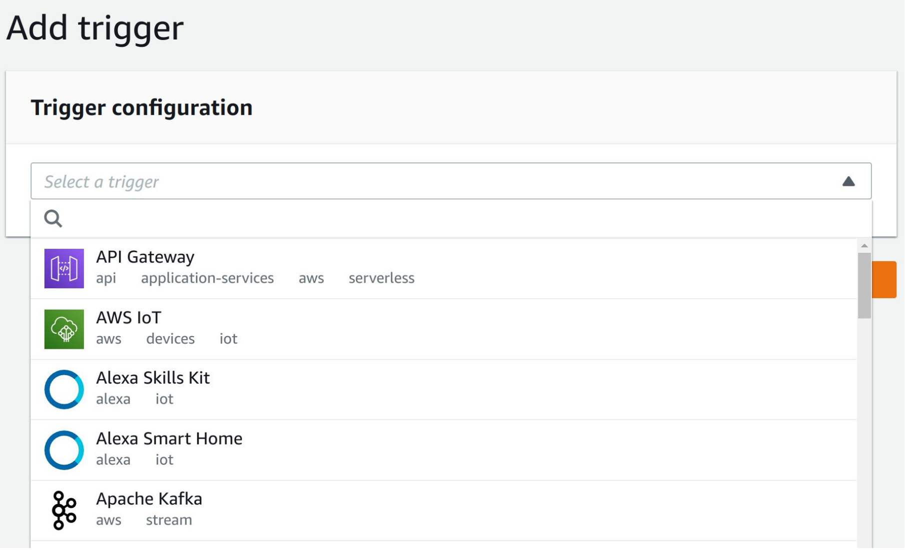
Python 3.7

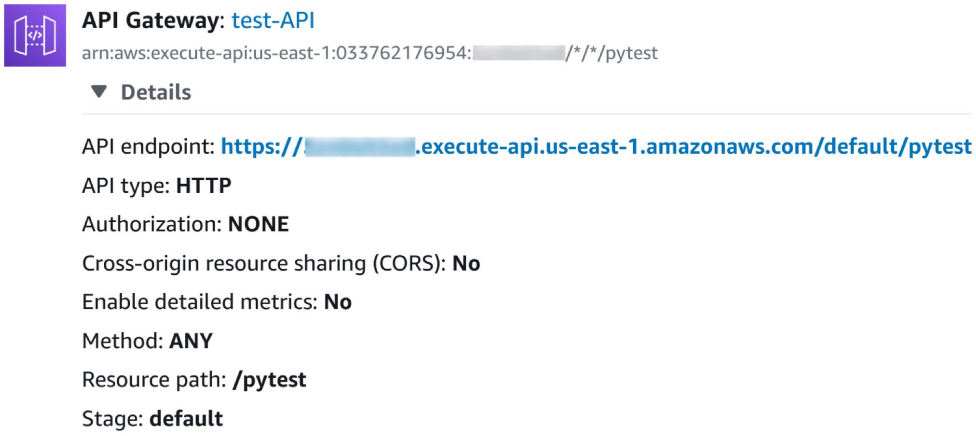
1.4 AWS Lambda Example



```python
import json

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

```

1.4 AWS Lambda Example





API Gateway: test-API

arn:aws:execute-api:us-east-1:033762176954:[redacted]/*/*/pytest

▼ **Details**

API endpoint: **https://[redacted].execute-api.us-east-1.amazonaws.com/default/pytest**

API type: **HTTP**

Authorization: **NONE**

Cross-origin resource sharing (CORS): **No**

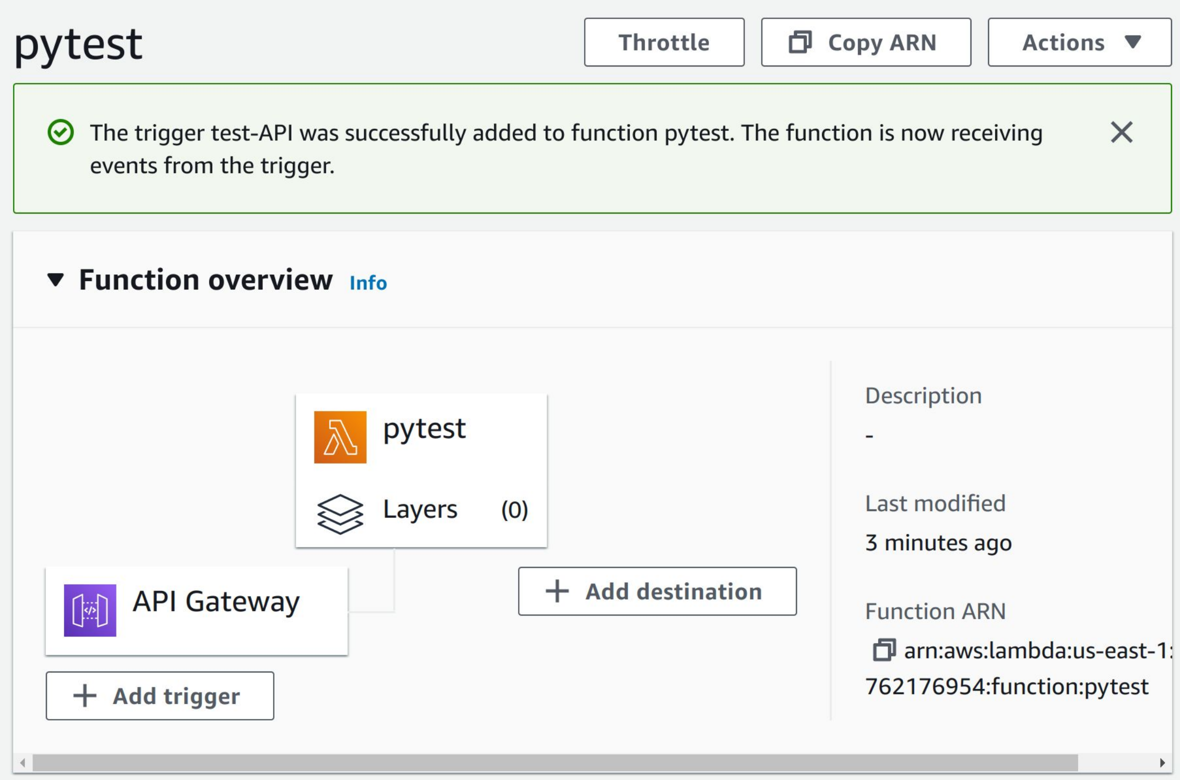
Enable detailed metrics: **No**

Method: **ANY**

Resource path: **/pytest**

Stage: **default**

1.4 AWS Lambda Example



The trigger test-API was successfully added to function pytest. The function is now receiving events from the trigger.

Description

-

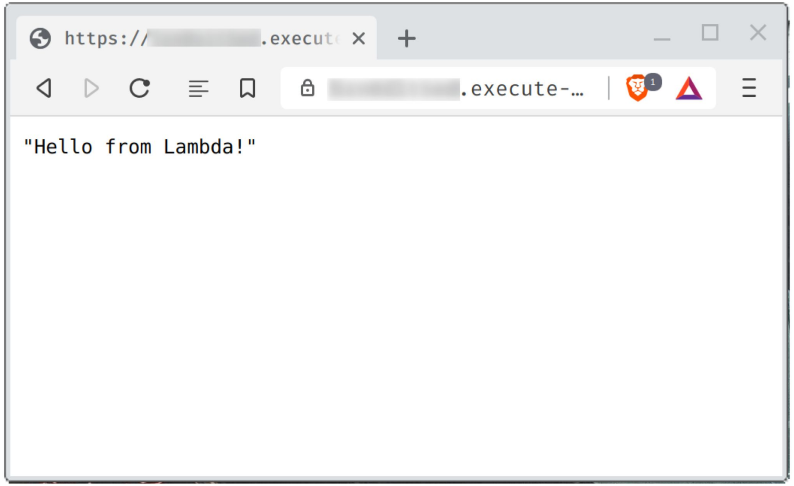
Last modified

3 minutes ago

Function ARN

arn:aws:lambda:us-east-1:
762176954:function:pytest

1.4 AWS Lambda Example



```
"Hello from Lambda!"
```

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**
Memory Size: **128 MB**
Max Memory Used: **50 MB**
Init Duration: **132.34 ms**

billed

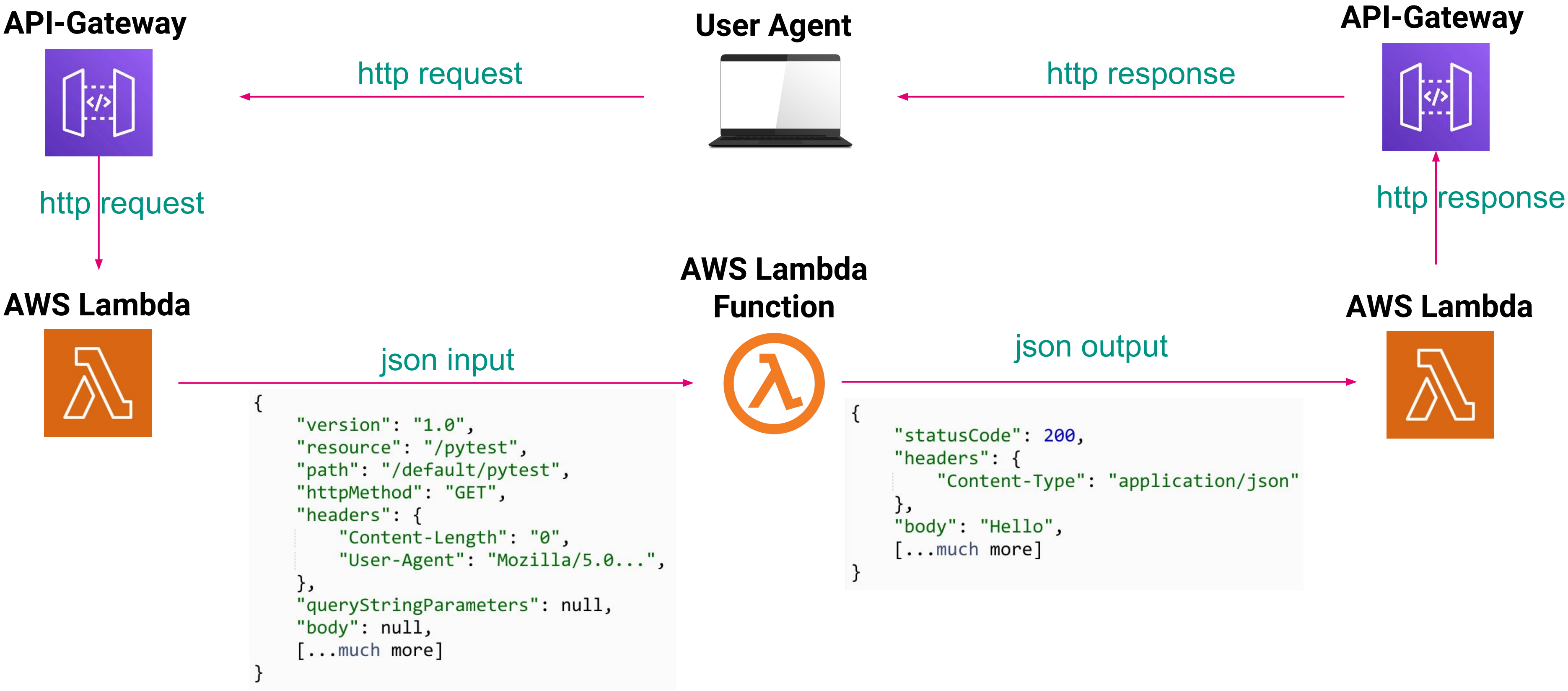
not billed

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

REPORT for **second invocation**:

Duration: 0.89 ms
Billed Duration: 1 ms
("Init Duration" disappeared)

1.4 AWS Lambda Example



API-Gateway

http request

User Agent

http response

API-Gateway

http request

http response

AWS Lambda

AWS Lambda Function

AWS Lambda

json input

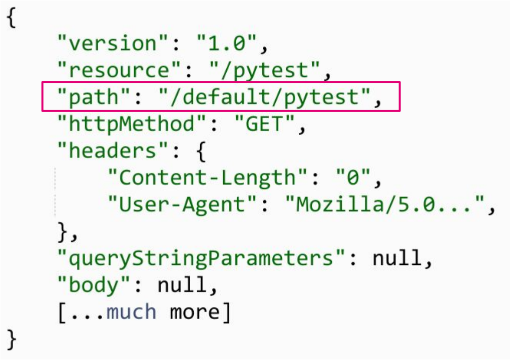
```
{
    "version": "1.0",
    "resource": "/pytest",
    "path": "/default/pytest",
    "httpMethod": "GET",
    "headers": {
        "Content-Length": "0",
        "User-Agent": "Mozilla/5.0...",
    },
    "queryStringParameters": null,
    "body": null,
    [...much more]
}
```

json output

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

1.5 Monoliths as serverless function

Example request to AWS Lambda:



```
{
    "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]
}
```

One function can handle multiple subpaths
→ monolith possible in just one function

1.6 AWS Lambda Function

Event = json Input

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

Context

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

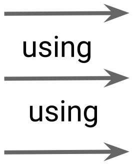
```
import json

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

json output

1.7 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



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)

2. Triggers

2.1 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

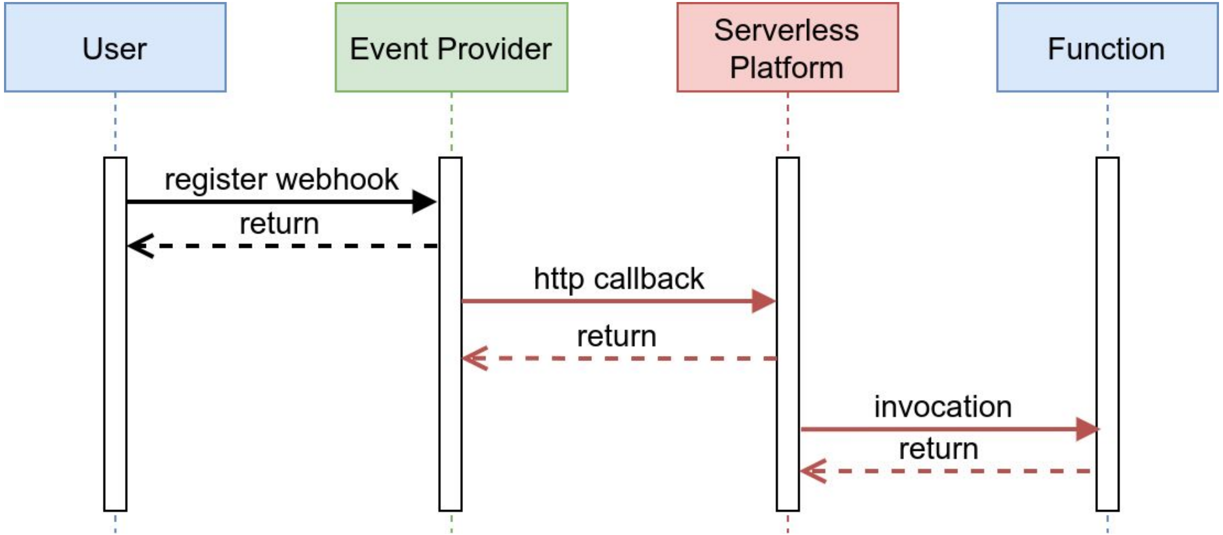
Amazon EventBridge CloudWatch

Sources: [2]

2.2 Webhooks as event source

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

a) Webhooks

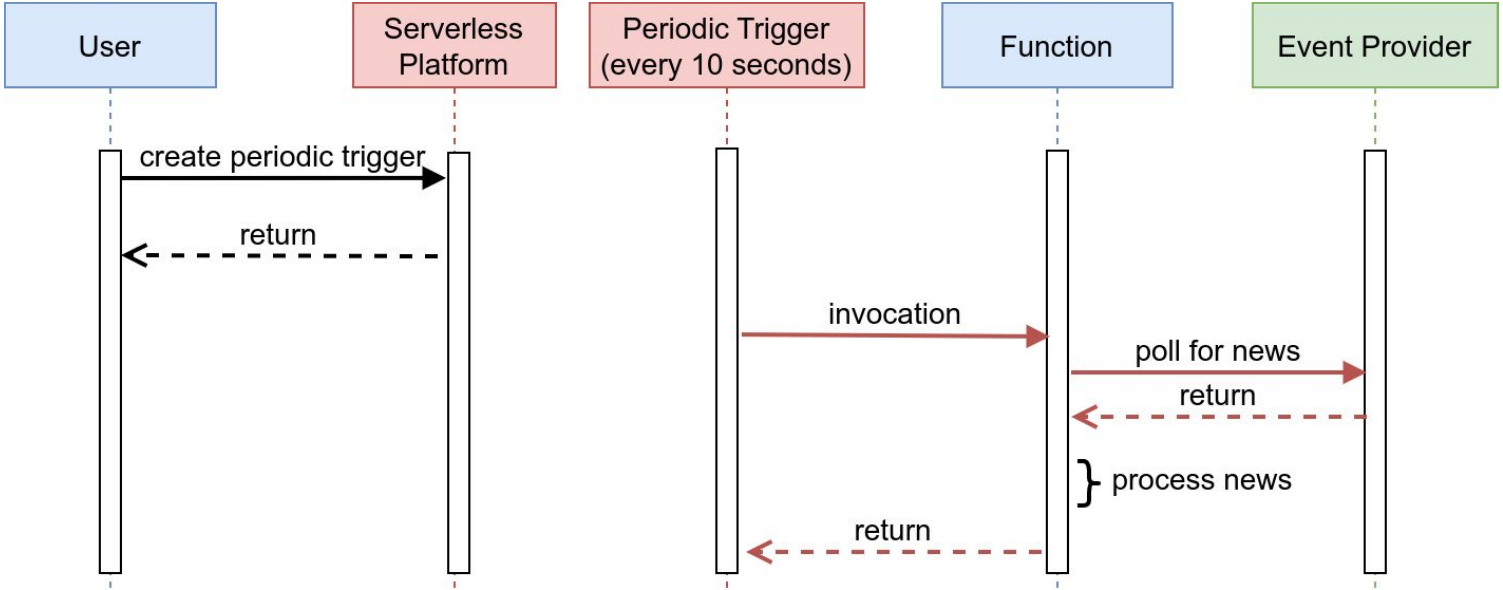


Sources: [3]

2.3 Polling as event source

for example polling an RSS feed every 10 minutes

b) Polling

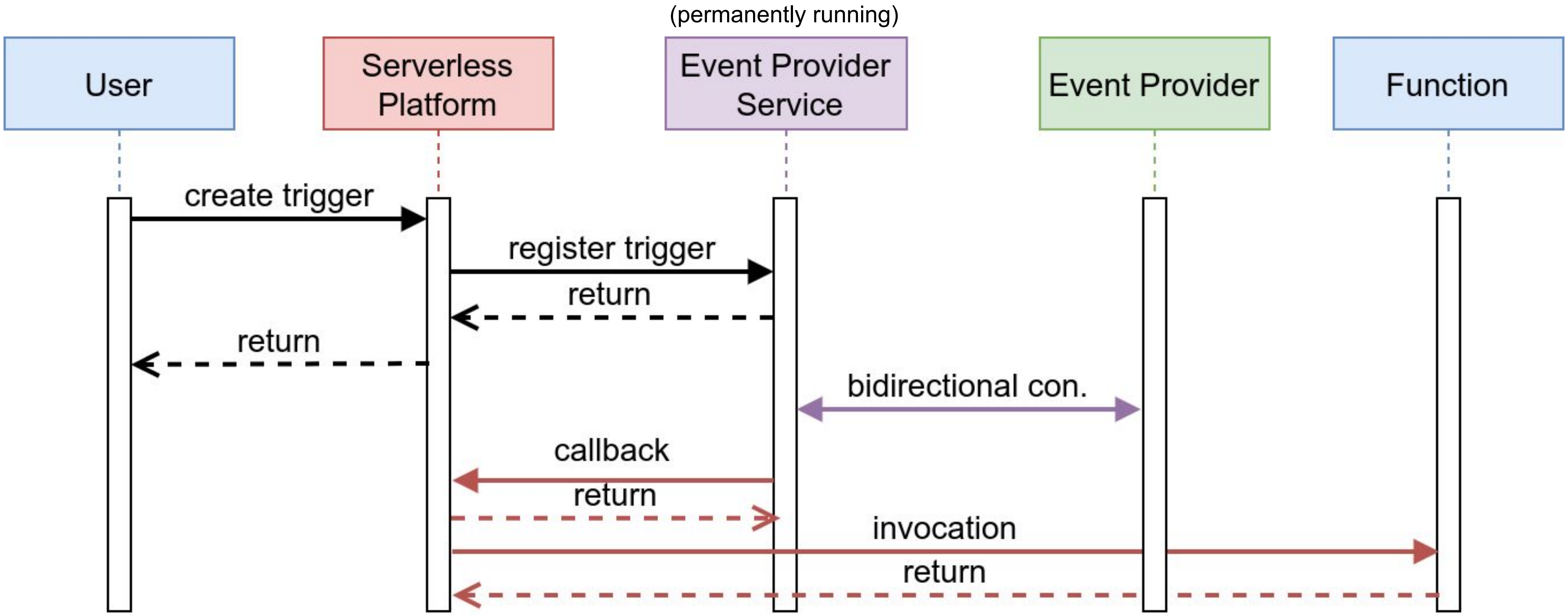


Sources: [3]

2.4 Permanent connections as event source

c) Connections Pattern

for example subscribing to an mqtt topic



Sources: [3]

2.5 Any protocol can be connected to serverless computing

Conclusion

- There are patterns to connect any stateless and stateful protocol to serverless computing
- But this may result in performance penalties
  or the necessity of a permanently running component

Sources: [3]

2.6 Synchronous and asynchronous invocations

Synchronous Invocation

- Blocks until result is available

Sync Example (OpenWhisk)

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

Asynchronous Invocation

- Returns immediately and gives an Activation ID
- Client can retrieve result later

Async Example (OpenWhisk)

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

Sources: [4]

3. Interfaces to functions

(Programmers point of view)

3.1 Interfaces to functions

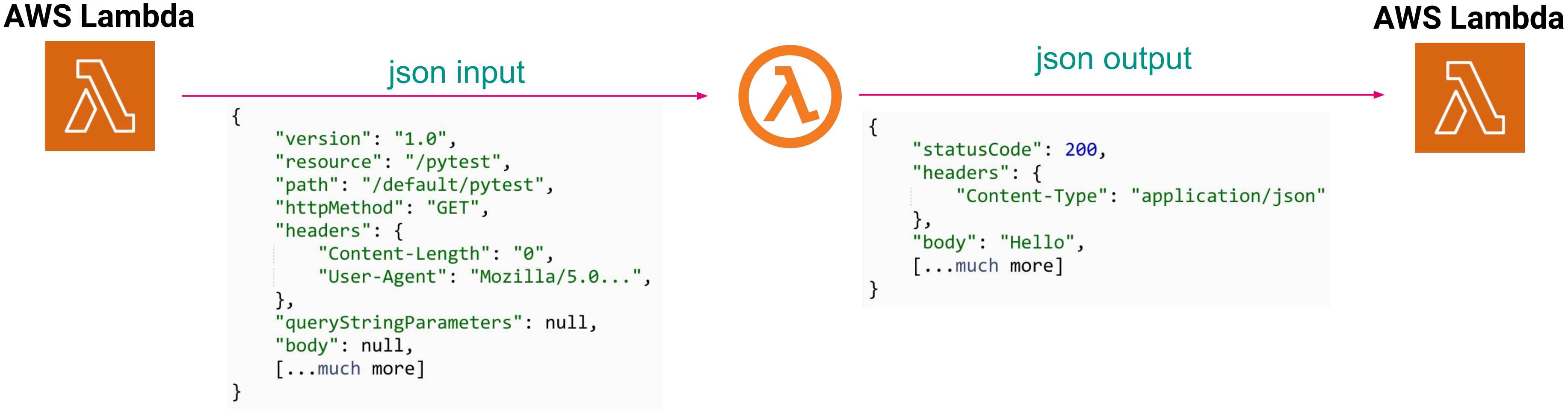
Question: How do programmers receive function invocations?

→ Answer depends on serverless computing implementation

3.2 Language-native map-like data structures

Used in: OpenWhisk, AWS Lambda

a) Platform translates incoming events to json objects



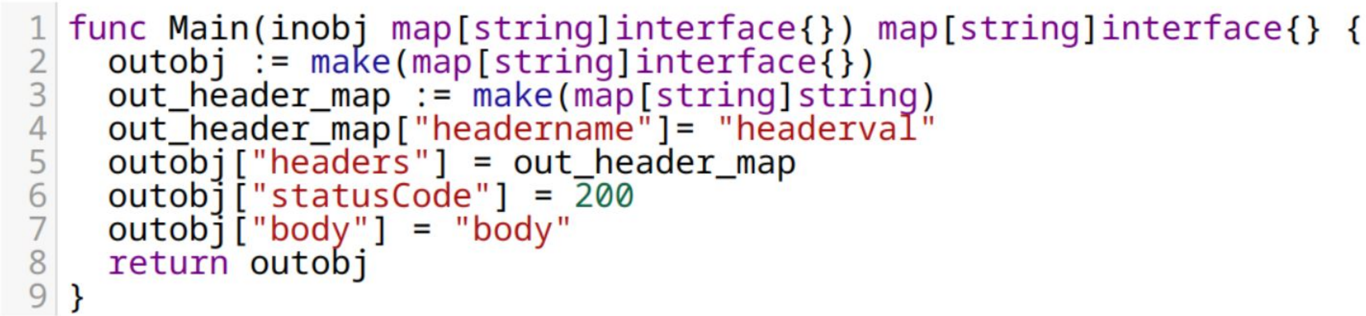
```
{
    "version": "1.0",
    "resource": "/pytest",
    "path": "/default/pytest",
    "httpMethod": "GET",
    "headers": {
        "Content-Length": "0",
        "User-Agent": "Mozilla/5.0...",
    },
    "queryStringParameters": null,
    "body": null,
    [...much more]
}
```

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

3.2 Language-native map-like data structures

b) json objects are then deserialized to language-native map-like data structures

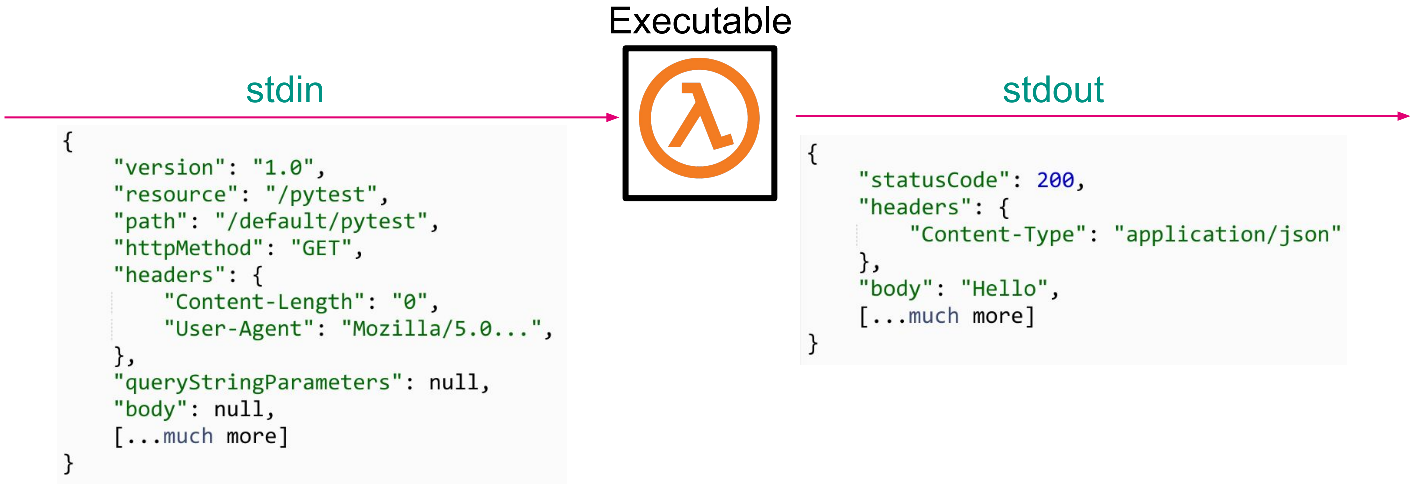
Go example from OpenWhisk:



```go
func Main(inobj map[string]interface{}) map[string]interface{} {
  outobj := make(map[string]interface{})
  out_header_map := make(map[string]string)
  out_header_map["headername"]= "headerval"
  outobj["headers"] = out_header_map
  outobj["statusCode"] = 200
  outobj["body"] = "body"
  return outobj
}
```

3.3 json over file descriptors (for example stdin and stdout)

Used in: OpenWhisk, OpenFaas



```
{
    "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]
}
```

→ Arbitrary executables and bash scripts can be used as serverless functions

3.4 Language-specific runtime SDK

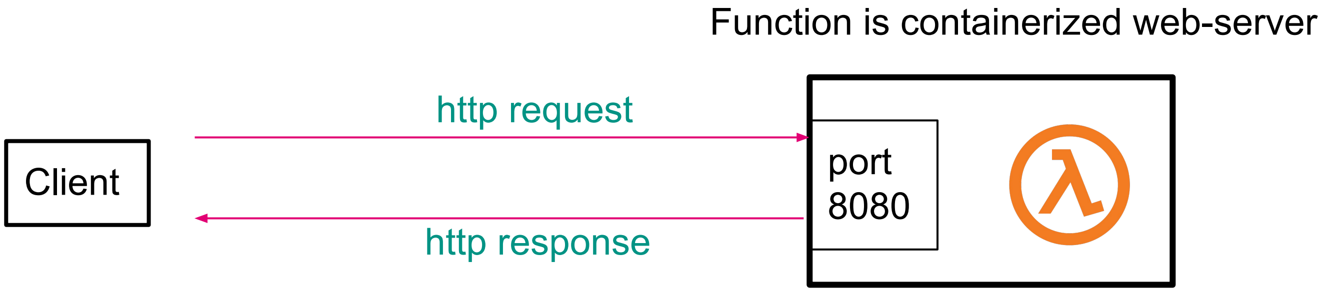
Used in: Fn, OpenFaas

Example from fn project:

```python
def handler(context, data: io.BytesIO=None):
    body = json.loads(data.getvalue())

    return response.Response(
        context,
        response_data=json.dumps({"message": "Hello {0}".format(name)}),
        headers={"Content-Type": "application/json"}
    )
```

3.5 Function receives http-requests on tcp port

Used in: Knative


Function is containerized web-server

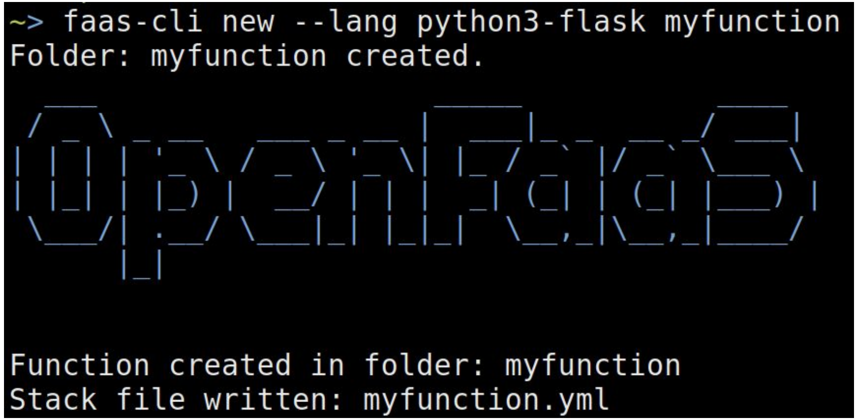
→ Developers can use their favourite web-server libraries
(In my opinion the most modern approach)
→ Function invocation is no additional layer of complexity

3.6 Software abstraction to well-known webserver library

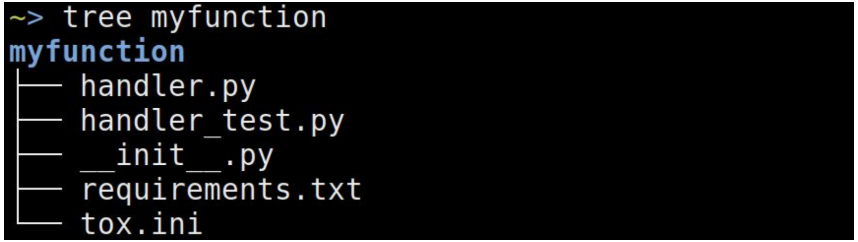
Used in: Fission, OpenFaas, Knative functions

→ Developers can use their favourite web-server libraries
(In my opinion the most modern approach)

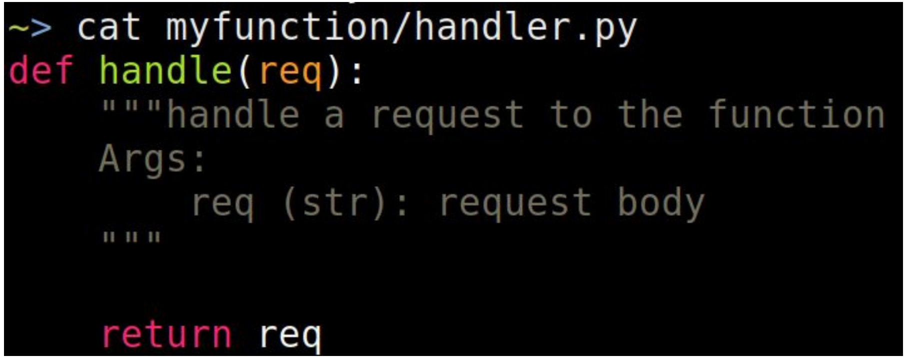
→ abstraction needs to implement runtime specification, which is sometimes not as trivial as with knative



```
~> faas-cli new --lang python3-flask myfunction
Folder: myfunction created.
  ___                   _____           ____
 / _ \ _ __   ___ _ __ |  ___|_ _  __ _/ ___|
| | | | '_ \ / _ \ '_ \| |_ / _` |/ _` \___ \
| |_| | |_) |  __/ | | |  _| (_| | (_| |___) |
 \___/| .__/ \___|_| |_|_|  \__,_|\__,_|____/
      |_|


Function created in folder: myfunction
Stack file written: myfunction.yml
```



```
~> tree myfunction
myfunction
├── handler.py
├── handler_test.py
├── __init__.py
├── requirements.txt
└── tox.ini
```



```
~> cat myfunction/handler.py
def handle(req):
    """handle a request to the function
    Args:
        req (str): request body
    """

    return req
```

3.7 Summary for the function invocation interfaces

Summary

- There are many different ways how programmers receive function invocations
- Letting programmers use well-known web-server libraries in their favourite language is the most modern approach used in fission, knative and openfaas

4. Runtimes

4.1 Deployment unit

There are two observed possibilities:
A) Function is deployed as a container image
B) Function is deployed as source-code file or archive containing source-code files

In case B, generic runtime containers are specialized with function code at runtime

→ Initialization effort contributes to cold-start time
→ Building a container image in the cluster after deployment would also be possible

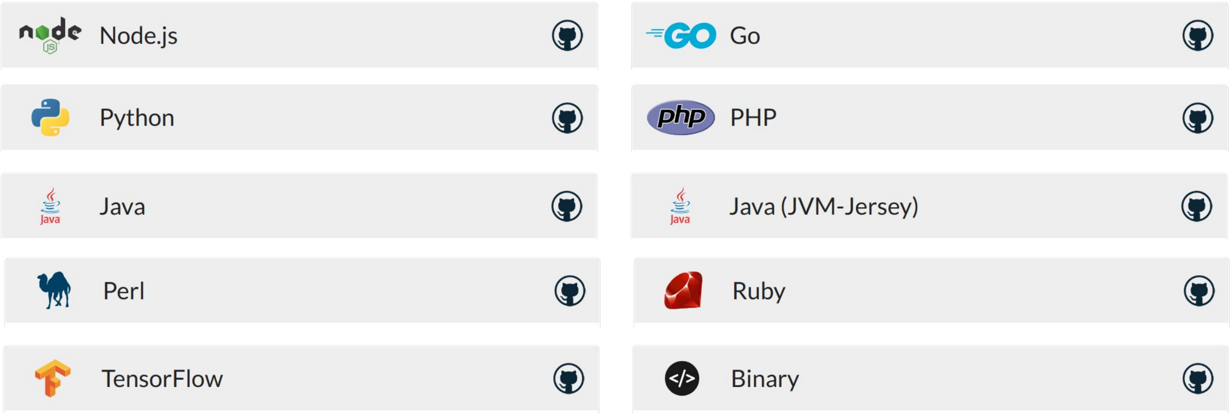
Specialization can not change

specialized runtimes can't be reused for other functions

4.2 Runtimes for programming languages

Runtimes are usually created at the level of programming languages

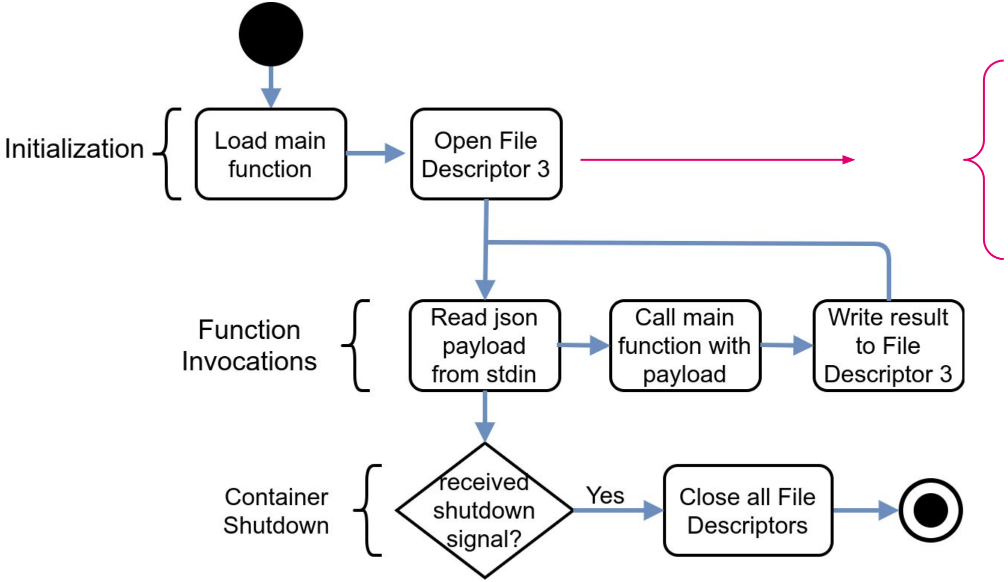
Official Fission “Environments”



Node.js
Go
Python
PHP
Java
Java (JVM-Jersey)
Perl
Ruby
TensorFlow
Binary

→ Runtimes usually provide a few common libraries

4.3 OpenWhisk Python Runtime


Initialization

Load main function

Open File Descriptor 3

Function Invocations

Read json payload from stdin

Call main function with payload

Write result to File Descriptor 3

Container Shutdown

received shutdown signal?

Yes

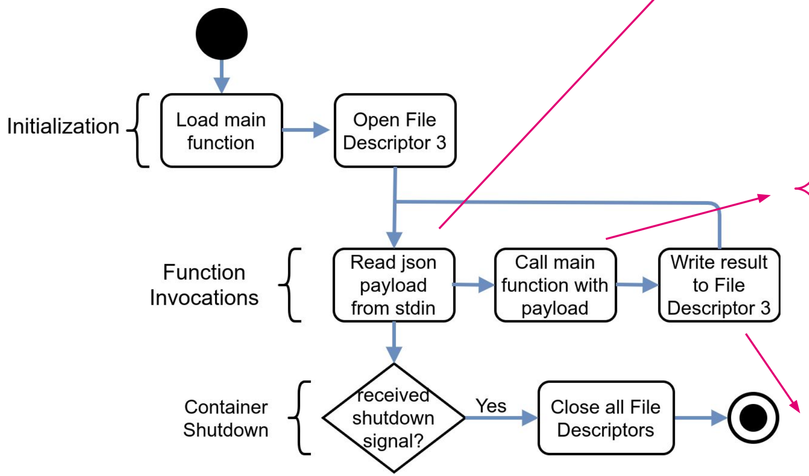
Close all File Descriptors

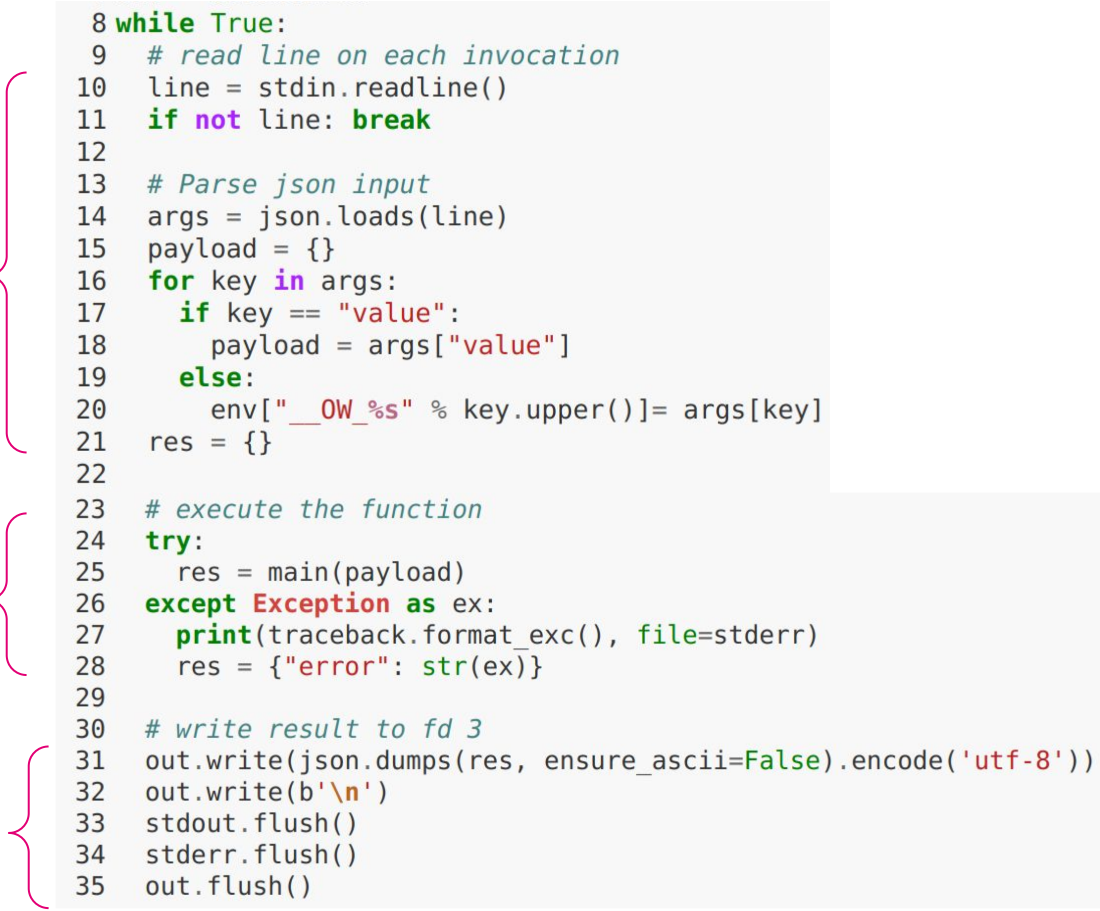
(Code slightly shortened version of the original OpenWhisk Python Runtime Implementation

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

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

```

Sources: [5]

4.3 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: [5]

4.4 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)

4.5 Limitations

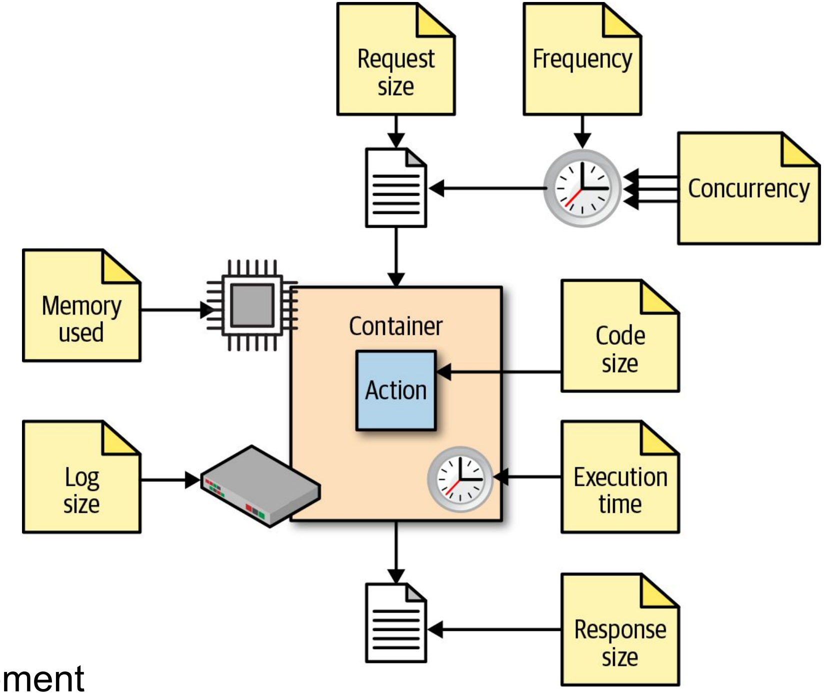
Introduced Problem:
Programmers need to take limitations into account which are specific to the used runtime and software

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

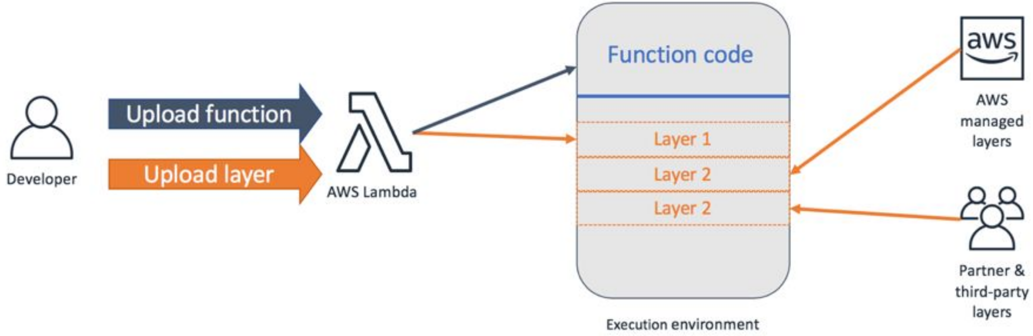
→ payload limits make some use cases hard to implement

Limitations in OpenWhisk:



Sources: [6]

4.6 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: [7,8]

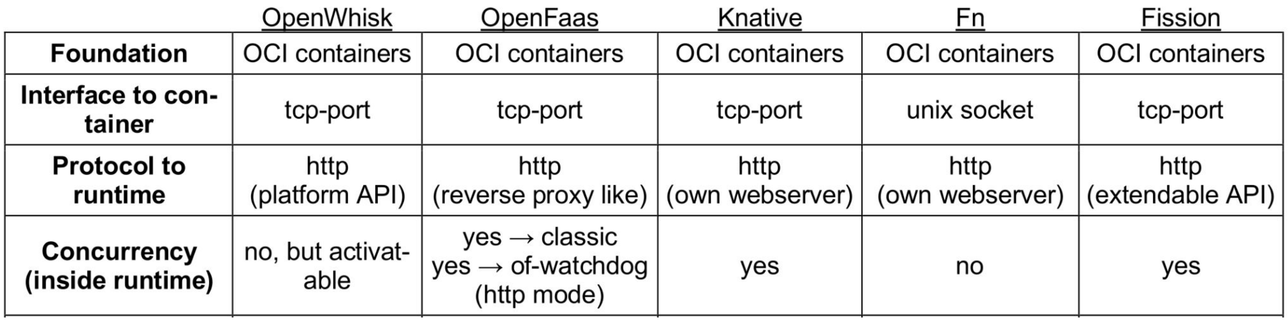
4.7 Relationship between serverless computing and container orchestration

- 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
→ Fission and Knative are integrated into Kubernetes with **Custom Resource Definitions**

4.8 Interface to runtime and whether it supports concurrent invocations



	OpenWhisk	OpenFaas	Knative	Fn	Fission
Foundation	OCI containers	OCI containers	OCI containers	OCI containers	OCI containers
Interface to container	tcp-port	tcp-port	tcp-port	unix socket	tcp-port
Protocol to runtime	http (platform API)	http (reverse proxy like)	http (own webserver)	http (own webserver)	http (extendable API)
Concurrency (inside runtime)	no, but activatable	yes → classic yes → of-watchdog (http mode)	yes	no	yes

The serverless platform can of course handle more than one concurrent request.

So why does it matter if a function can be invoked concurrently multiple times for a single runtime?

→ *Soft limits make it possible to completely avoid cold-starts during scaling out*

→ *Requests can be immediately answered without waiting for a new pod to become available, if runtimes can always tolerate one more request*

5. Cold starts

5.1 Cold- and warm-starts

Cold-start

- Before a function invocation can be processed, a new function instance needs to be started → cold-start delay = initialization time + execution time

Warm-start

- The invocation can be forwarded to an already existing function instance with free capacity → warm-start delay = execution time

Occurrence of cold-starts

- a) When no function instances are running
  - after scale-to-zero, failure or deployment
- b) During scaling out if all present function instances can’t serve more requests.

Sources: [1]

5.2 Avoiding cold-starts

Occurrence of cold-starts

- a) When no function instances are running
  - after scale-to-zero, failure or deployment
- b) During scaling out if all present function instances can’t serve more requests.

Avoid case a)

- disable scale-to zero

  and
- set minimum/initial number of function instances >= 1

Sources: [1]

5.2 Avoiding cold-starts

Occurrence of cold-starts

- a) When no function instances are running
  - after scale-to-zero, failure or deployment
- b) During scaling out if all present function instances can’t serve more requests.

Avoid case b)

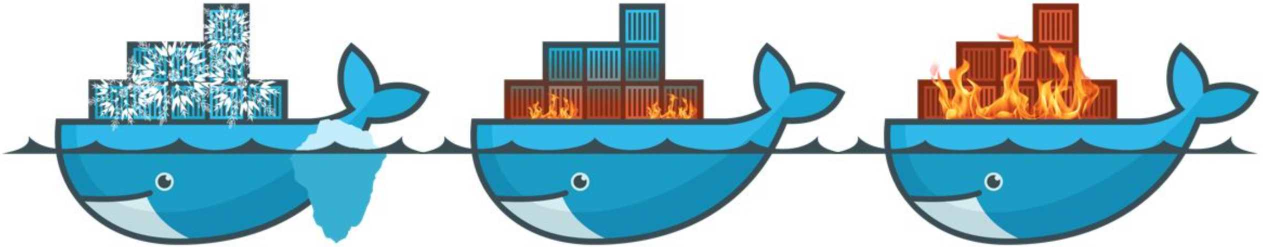
- always have more function instances than needed
  - so there is always an idling function instance available to process an event immediately

or

- have function instances, which support concurrent function invocations and do not set a hard-limit for concurrency
  - by using a soft-limit always one more invocation can be processed immediately
  - scaling is detached from function invocation and happens in the background (“eventual scaling”)

Sources: [1]

5.3 Cold- vs. Prewarm- vs. Warm-start



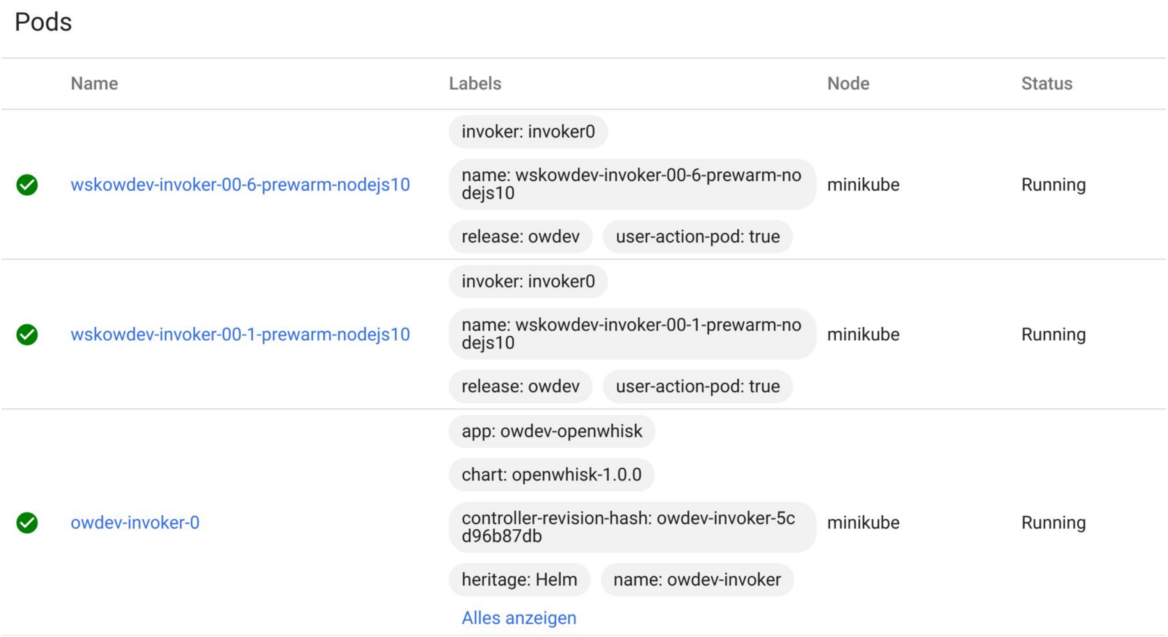
What needs to be done?	cold	prewarm	warm
Starting the container	🟩		
Initializing the action	🟩	🟩	
Running the action	🟩	🟩	🟩

-> prewarming only possible when generic containers are specialized

Sources: [9]

5.4 Prewarm containers in OpenWhisk

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

6. Keeping state

6.1 Why serverless functions need to be stateless

Platform can scale down automatically and transparently

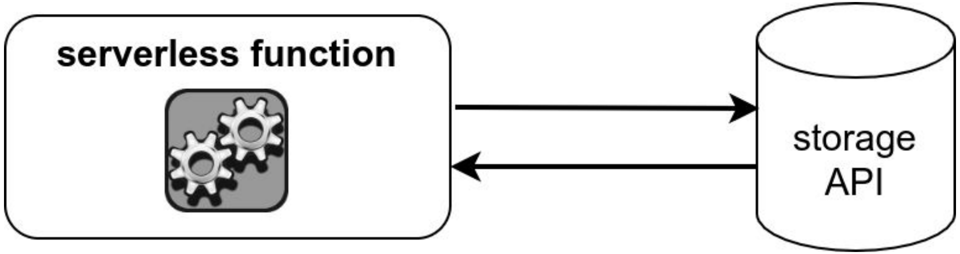
- Content of memory can be removed any time

→ State needs to be externalized
→ Memory is still usable for temporary caching

Sources: [1]

6.2 Externalize state

External storage API



→ **Suitable solution**

Advantages

- Modular development of storage API
- Separated administration
- Separated scalability

Disadvantages

- Might need additional complexity like locking mechanisms or ACID transactions

6.3 Suitable database and storage APIs for FaaS

Properties of database APIs suitable for use with 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 themselves*

6.4 AWS Aurora Provisioned vs. Serverless

Amazon AWS Aurora

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

AWS Aurora - Provisioned

<u>Billing:</u>

- 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

<u>Billing for</u>

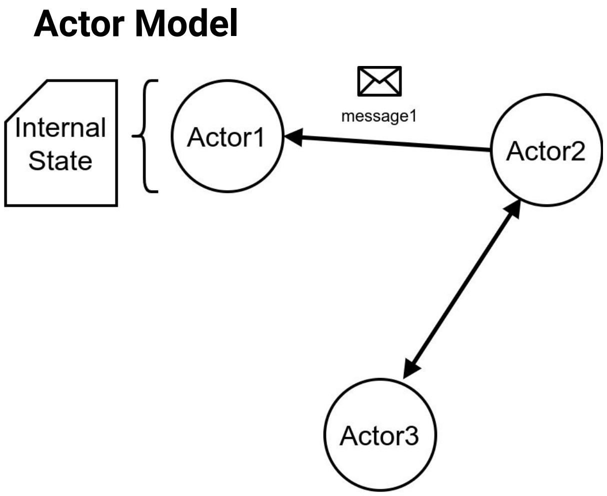
- storage (**per GiB/month**)
- I/O rate (**per 1 Mio req**)
- Aurora Capacity Unit (**per hour**)

(1 ACU ~ 2GiB Memory usage)

Differences in Serverless

Different pricing
+ Autoscales up and down
+ Stateless http “Data API”

6.5 Actors model

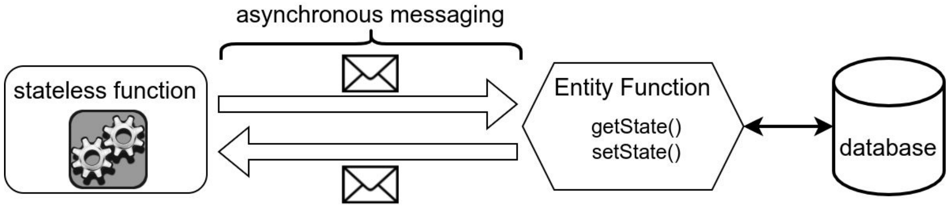


- Actor1 has an internal state
- Access to that internal state only via messages

Sources: [10]

6.6 Entity functions

Entity functions



- Entity functions have a persistent state
- Access to internal state only via event-driven messages
- Backend can be normal storage API

(Practical example: Azure Functions Durable Entities)

Ways to communicate

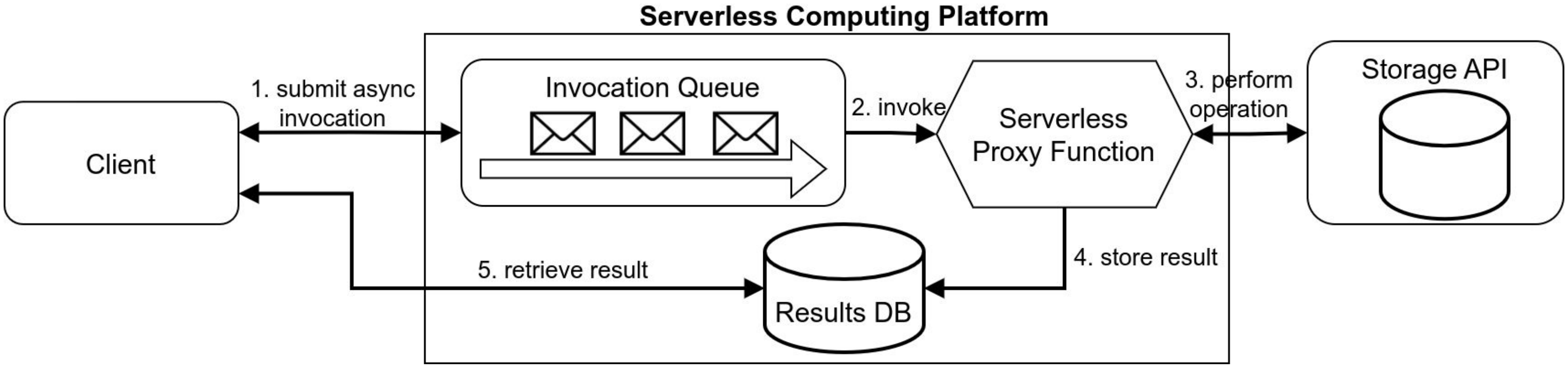
<u>Signaling:</u> fire-and-forget
→ write operations without confirmation

<u>Call:</u> blocking call, waiting for response
→ read and write operations
(identical to normal synchronous APIs)

Sources: [11]

6.7 Serverless computing platforms as entity functions

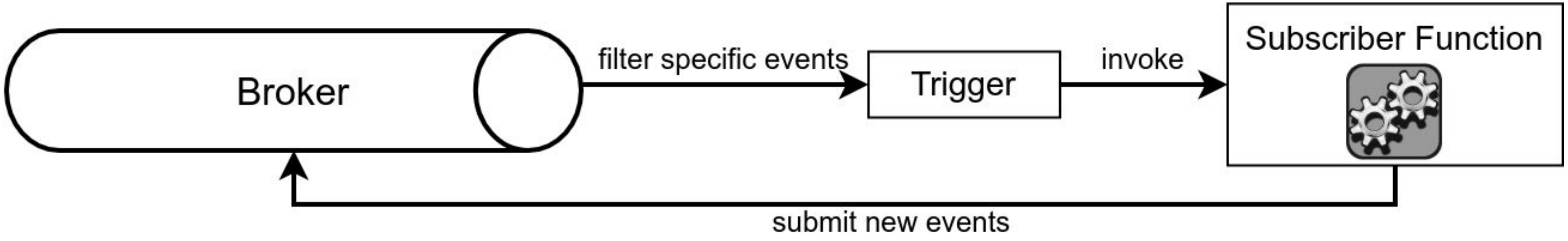
Serverless platforms together with asynchronous invocations can be programmed to be an implementation of entity functions



6.8 Message-broker for asynchronous invocations

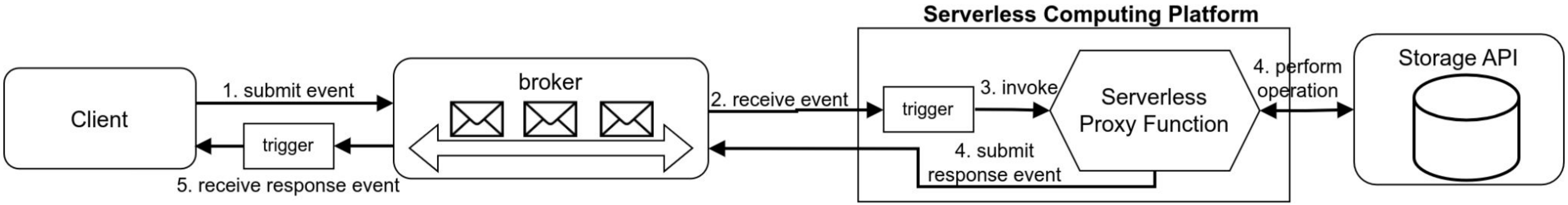
Not all platforms support asynchronous invocations

- But all platforms can be configured to receive events from a message broker



6.8 Message-broker for asynchronous invocations

Connecting a message broker to add the feature of asynchronous invocations to a serverless platform

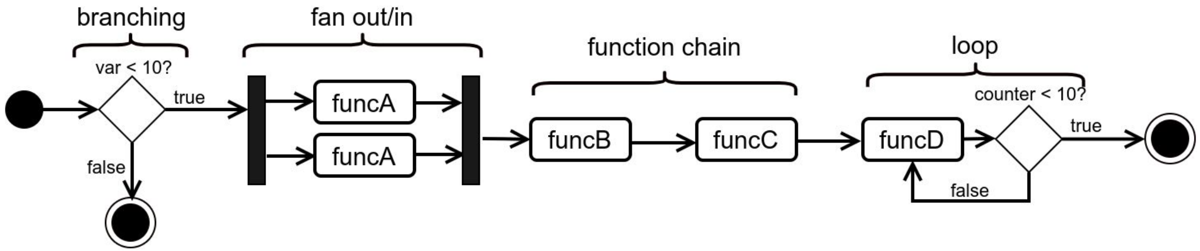


→ **Also a possible implementation of entity functions**

6.9 Store data using in a platform-native workflow system

Storing data in workflows

(some platforms provide a workflow system to connect multiple functions together)



Locations to store data:

- Current workflow state (as in a state machine)
- Parameters forwarded from function to function
- Platform managed workflow context

Advantages

- Function instances use less memory
- Implicit garbage collection
- No double-billing if workflows are ST-safe (explained later)

Disadvantages

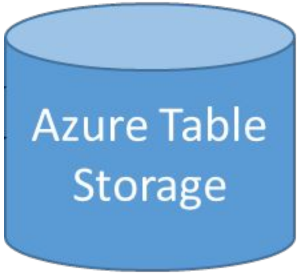
- Code needs to be platform aware
- Impacts modularity (data moved out of API specification to workflows)

6.10 Every method to externalize state uses an external storage

All observed practical implementations of the described methods under the hood externalize state in an external API

Example: “Azure Functions” and “Azure Functions Durable Entities”

Current state is saved in:



Example: Invocations and their parameters inside “OpenWhisk Sequences” are stored in CouchDB.

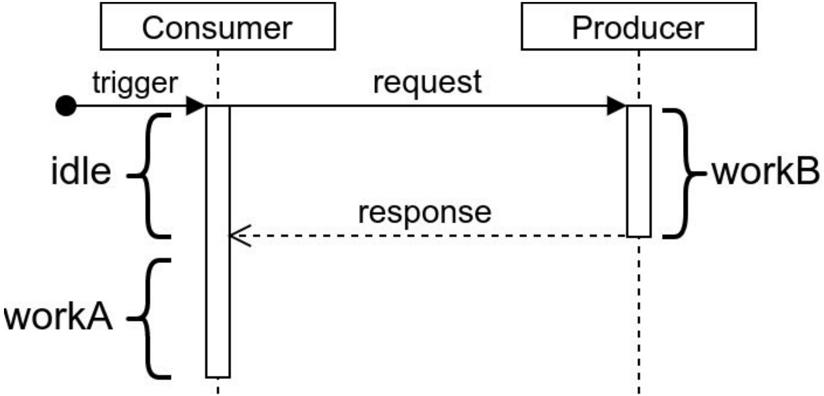
CouchDB
relax

Sources: [12]

7. Serverless Trilemma

7.1 Double billing

If functions invoke other functions synchronously, the time is billed <u>twice</u>

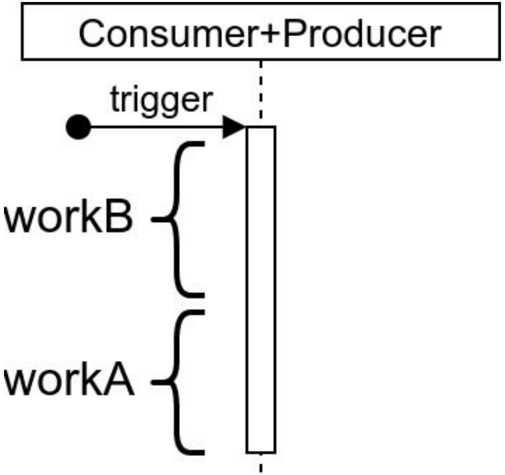


- applies only if billing is based on function execution time
- consumer also blocks a concurrency slot in the function instance

billed_duration = idle + workA + workB

Sources: [13]

7.2 Solution A: Inline producer code



billed_duration = workA + workB

Problems

- Knowledge of internal programming of Producer required
- API of Producer not used anymore

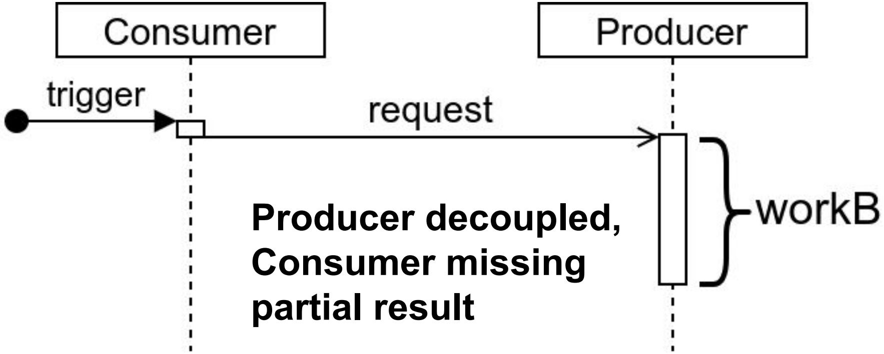
→ **Double billing avoided**

→ **Black-Box principle of producer violated**

Sources: [13]

7.3 Solution B: Asynchronous split off

Invoke other functions asynchronously



Producer decoupled, Consumer missing partial result

→ **Double billing avoided**

→ **Consumer can’t produce same result as before**

→ **Consumer can’t be a substitutable part of a composition anymore, because it doesn’t behave identical to an atomic function**

→ **If intermediate result is not required, solution can be still suitable**
(for example when sending an email notification)

Note:
- Waiting for a result would be synchronous operation with double billing

Sources: [13]

7.4 Serverless Trilemma

Desired properties of serverless computing architectures:

1. **Treat functions as black-box**
2. **No Double-Billing**
3. **Substitutable functions**

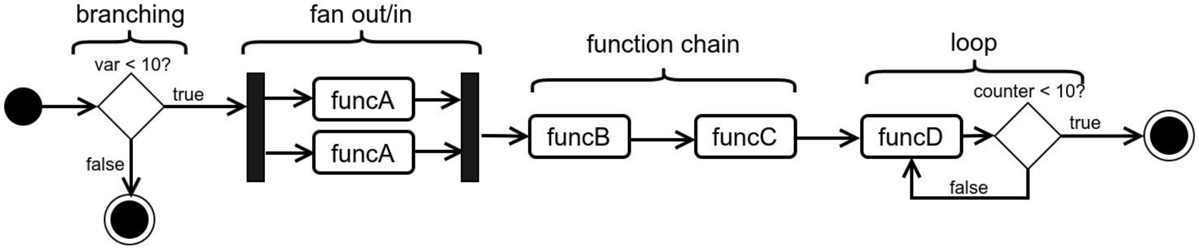
Serverless Trilemma: In an event-driven system one of the properties must be violated

Sources: [13]

8. Workflows

8.1 ST-Safe workflows avoid serverless trilemma

Workflow system provided by the serverless platform



ST-Safe

- A serverless function orchestration system that satisfies all three conditions of the serverless trilemma is called ST-Safe

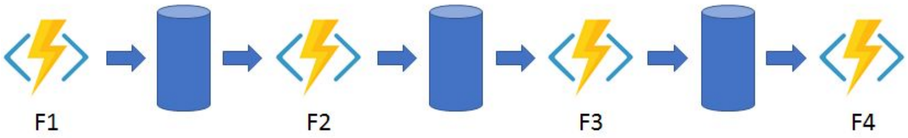
Serverless trilemma can be applied “backwards”

- If a program doesn’t use ST-Safe workflows, it must violate one of the desired properties of the serverless trilemma

Sources: [13]

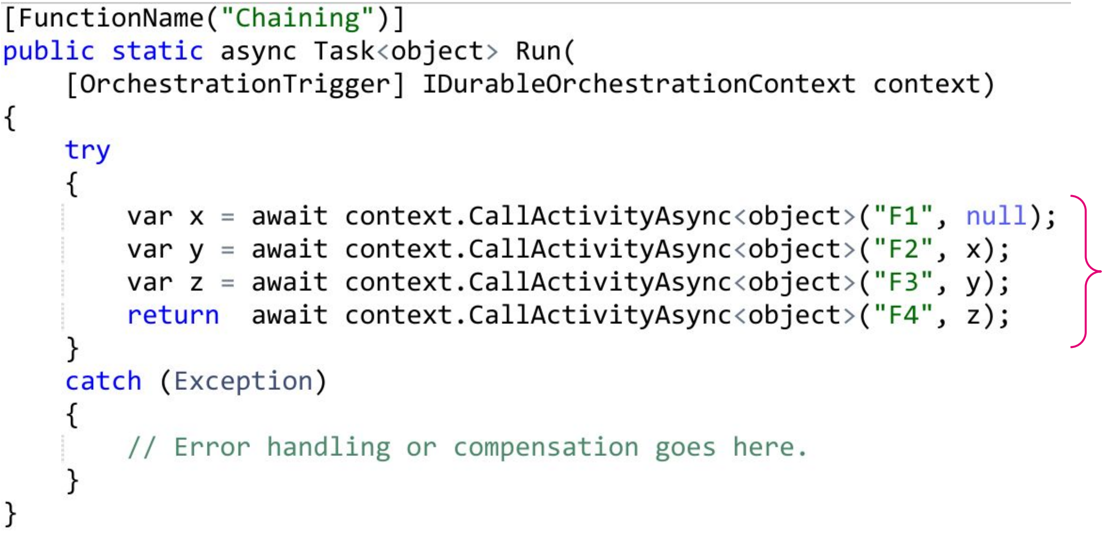
8.2 Workflows in Azure Functions

Function Chaining



→ Azure Durable functions uses syntax known from async programming

orchestrator function



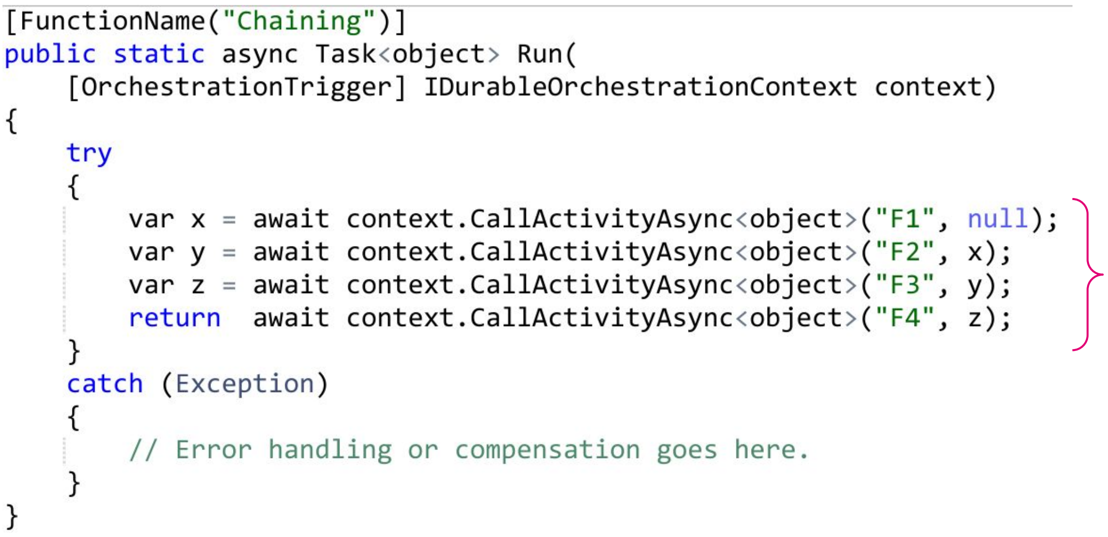
```
[FunctionName("Chaining")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1", null);
        var y = await context.CallActivityAsync<object>("F2", x);
        var z =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

“await” pauses billed execution time

activities

Sources: [14]

8.3 Workflows in Azure Functions - Replaying and Checkpoints



```
[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.
    }
}
```

orchestrator function

activities

How it works:

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

Sources: [14]

9. Programming

for serverless computing

9.1 Consequences of formal foundations paper

Paper "Formal Foundations of Serverless Computing"

- Contains theoretical model of serverless computing based on operational semantics
- does not fit perfectly to recent developments
- but introduces relevant properties programmers need to account for.

Usually given guarantee:
At-Most Once execution

(Retries of failed function invocations are usually a separated feature, which is turned off by default)

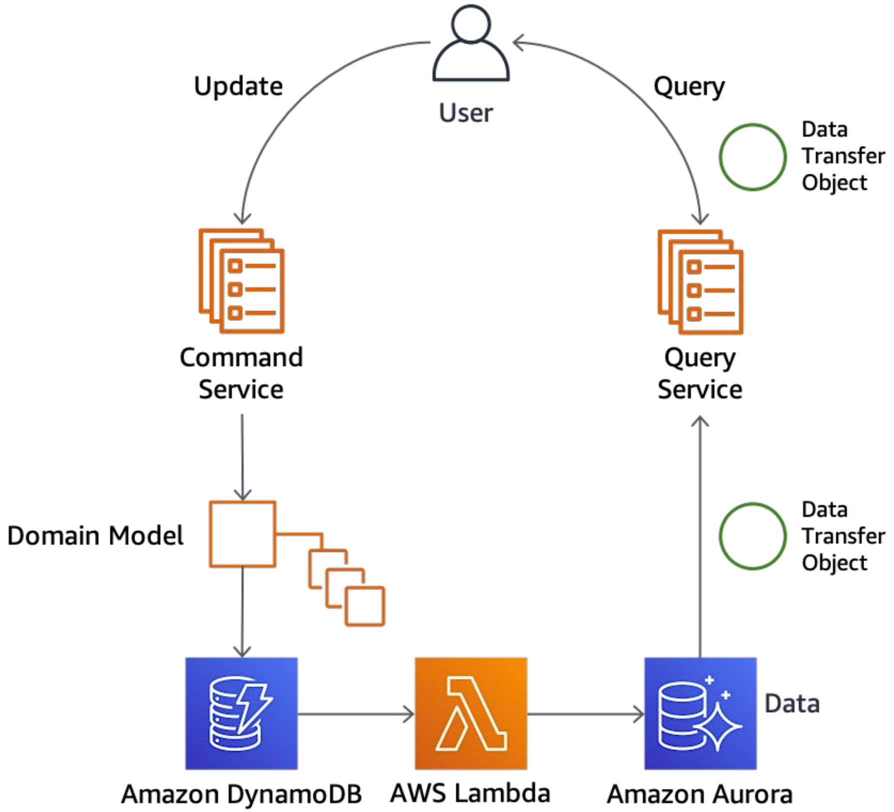
Function as a Service

- Execution of requests
  - not necessarily in order
  - possibly in parallel
  - may happen twice

Solution in short:
ACID transactions in storage backend
+ locking/mutex mechanisms
+ idempotent functions

Sources: [15]

9.2 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 data models!

Sources: [16]

10. Suitable workloads and use cases

10.1 Relationship between Edge Computing and FaaS

Edge Computing

- Process data where it is generated and/or needed

The data a function operates on may be a lot bigger

Functions have small size

(Small source-code files or small docker image layers)

Conclusions

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

→ Small deployment size makes serverless computing especially suitable for edge computing

Sources: [1]

10.2 Bursty traffic patterns

Deployment with permanently provisioned resources

- permanent need for 10 servers

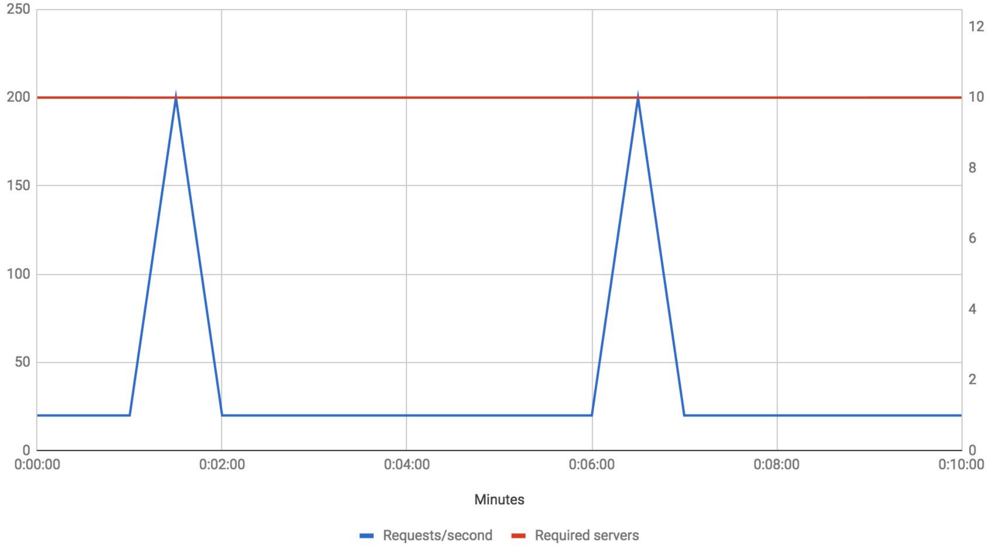
  to account for all possible traffic spikes

Deployment with on-demand resource usage

- scaling down avoids idling servers

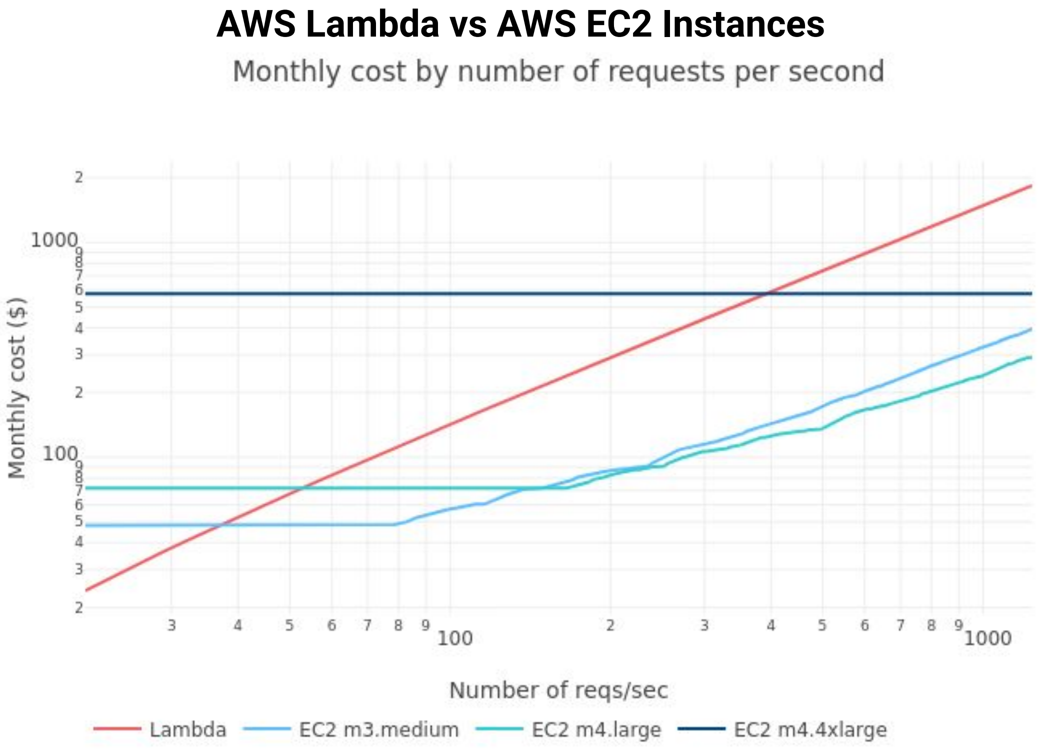
→ Billing based on milliseconds execution time makes serverless computing very suitable for bursty traffic patterns

Bursty traffic pattern (example with provisioned resources)



Sources: [1]

10.3 Exemplary cost calculation - provisioned vs serverless



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

In general

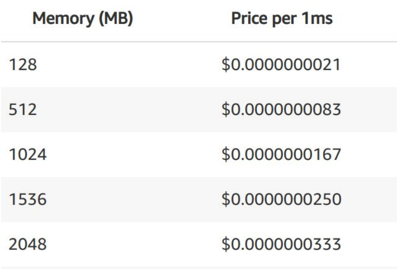
Serverless functions good for

- Low total number of reqs/sec
- High idle times (or bursty traffic)
- Unpredictable scenarios
  → no sizing necessary beforehand

Sources: [17]

10.4 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

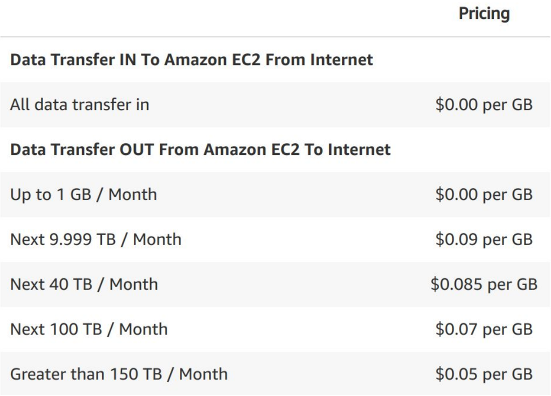
+ price for number of requests

+ traffic price inbound and outbound a region

+ other services used

For example: S3 Object Storage
- GB stored per month
- per 1000 requests
- Replication

	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

AWS Lambda can easily cost more

Sources: [18,19]

11. Advantages, disadvantages, learnings

11.1 Complexity prevented from developers

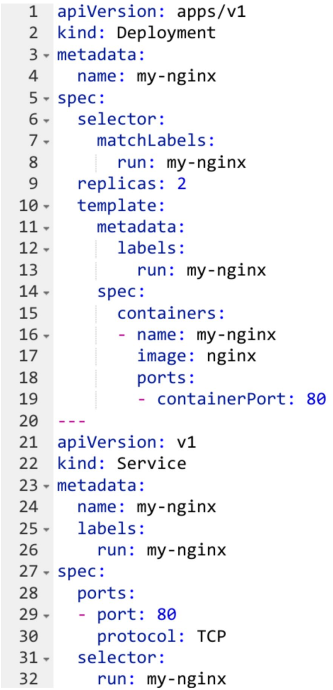
Complexity prevented from developers

- Managing horizontal scaling **(K)**
- Achieving high redundancy and availability **K**
- Evenly distributed load balancing **K**
- Performing rolling updates **(K)**
- No need for system upgrades **K**
- No infrastructure management **K**

K = Advantage already given by kubernetes
(K) = Advantage already given, but limited

Why using serverless computing at all?
→ small config file implicitly gives many features
→ new features

11.2 Using serverless computing enables features implicitly



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

Kubernetes

Knative

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: my-nginx
spec:
  template:
    spec:
      containers:
        - image: nginx
          ports:
            - containerPort: 80
```

Less yaml, but implicitly more features like

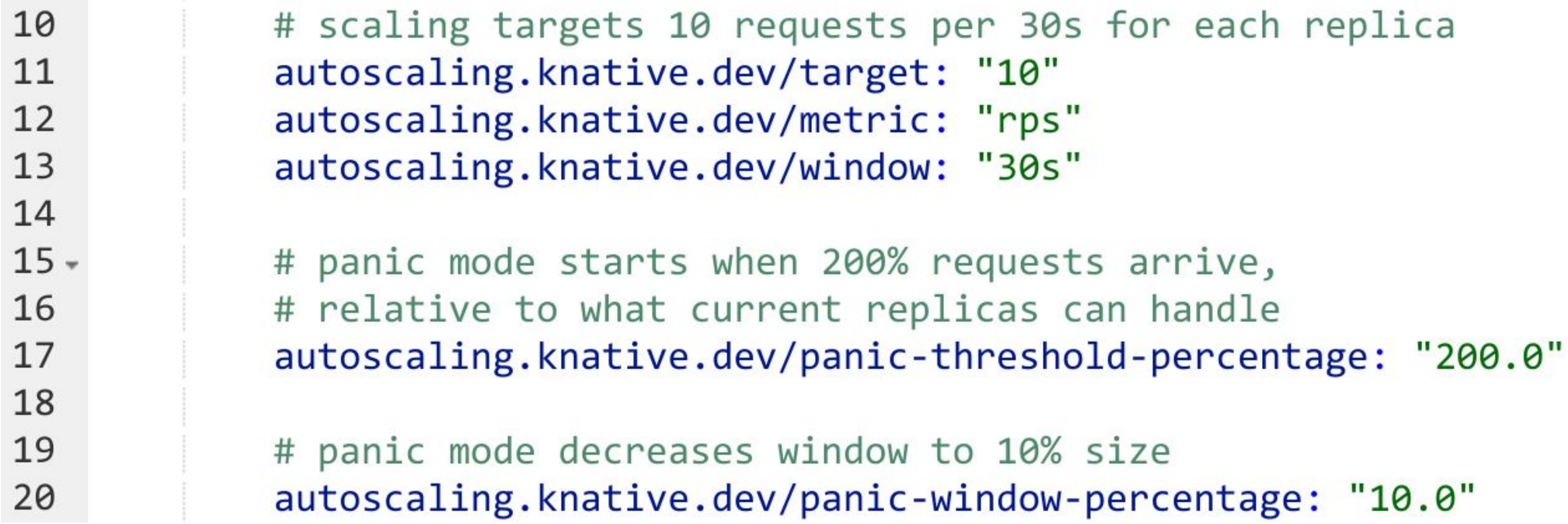
- scale-to-zero
- on-demand resource usage
- revisions
- etc.

→ **new projects can leverage these features without any effort**

11.3 New features of serverless computing platforms

Many new features introduced.

Example: Sophisticated Autoscaling in Knative



```
    # scaling targets 10 requests per 30s for each replica
    autoscaling.knative.dev/target: "10"
    autoscaling.knative.dev/metric: "rps"
    autoscaling.knative.dev/window: "30s"

    # panic mode starts when 200% requests arrive,
    # relative to what current replicas can handle
    autoscaling.knative.dev/panic-threshold-percentage: "200.0"

    # panic mode decreases window to 10% size
    autoscaling.knative.dev/panic-window-percentage: "10.0"
```

11.4 Complexity added for developers

Complexity added for developers

- New software and cli tools to learn
- Need to consider unique properties
  - Stateless
  - Cold-starts
  - Time and resource limitations
  - Serverless-trilemma
- Additional problems
  - Double-billing
  - Serverless trilemma
  - How and why to use workflows
  - New security vulnerabilities <- *there are papers*
  - Need to program event generators
- Less comfort
  - Harder debugging of applications

11.5 Operations when using FaaS

Does using serverless computing in a public cloud mean we don't need an operations team? -> **No**

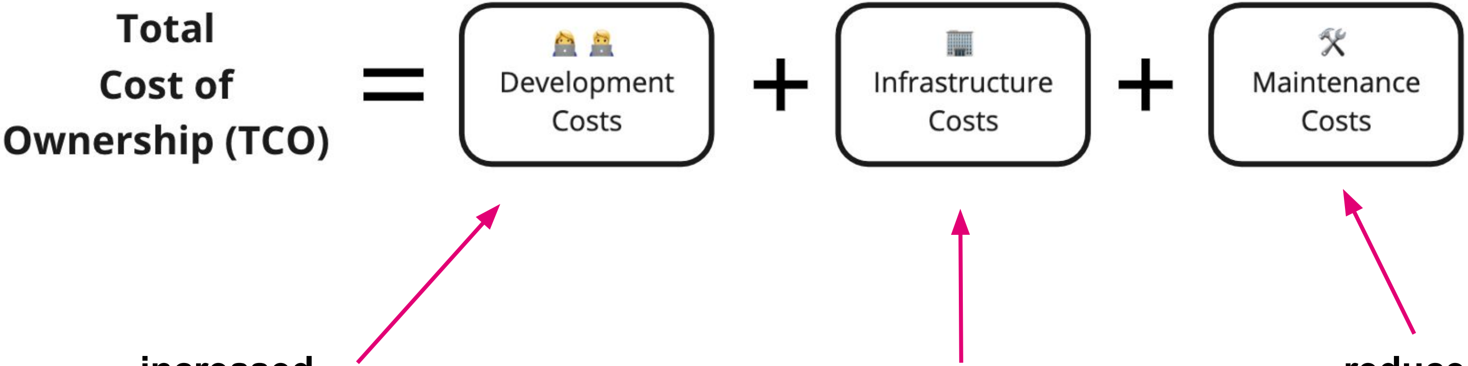
FaaS software still requires operations:

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

But usually cloud provided:

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

11.6 Total cost of ownership



Total Cost of Ownership (TCO) = 👩‍💻 👨‍💻 Development Costs + 🏢 Infrastructure Costs + 🛠 Maintenance Costs

increased

- developers need to consider more concepts

reduced or increased

- depending on req/s, traffic used and storage used

reduced

- more tasks shifted to the cloud
- but operations team still necessary

11.7 New features of serverless computing platforms

My conclusion:

- Making apps stateless is a good idea to automatically achieve horizontal scalability, but this is independent from serverless computing
- The technology serverless computing doesn’t add enough advantages compared to kubernetes to justify another complexity layer

- New and small projects can benefit from effortlessly usable features like automatic horizontal scaling

End of presentation

Thank you for your participation

Feel free to ask questions

Sources (1 / 4)

01. Sources on the state of research (you of course do not need to read everything, this is just for interested people)

H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, “Survey on serverless computing,” J Cloud Comp, vol. 10, no. 1, p. 39, Dec. 2021, doi: 10.1186/s13677-021-00253-7.

(starting on page 16 in the section "Serverless computing challenges and issues (RQ7)" there is a good and concise overview of the open questions of the current state of research on serverless computing. The paper is from 2021 and therefore very current. Starting on page 19, there are also research questions)

A. Bocci, S. Forti, G.-L. Ferrari, and A. Brogi, “Secure FaaS orchestration in the fog: how far are we?,” Computing, vol. 103, no. 5, pp. 1025–1056, May 2021, doi: 10.1007/s00607-021-00924-y.

(Very good overview of selected current papers around the topic of serverless computing. Short paragraphs describe the content of each paper. I recommend reading chapters 4 and 5)

A. Jangda, D. Pinckney, Y. Brun, and A. Guha, “Formal Foundations of Serverless Computing,” Proc. ACM Program. Lang., vol. 3, no. OOPSLA, pp 1–26, Oct. 2019, doi: 10.1145/3360575.

(The paper defines Operational Semantics for Serverless Computing, highlighting a few specifics that derive from this theory for FaaS platforms. First of interest are pages 4 and 5. Page 4 shows a sh and a long source code. The long source code compensates for the problems of FaaS by writing more code. Which problems these are can be read in the following text.)

G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, “Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research,” 2017, doi: 10.13140/RG.2.2.15007.87206.

(The results of a conference on serverless computing. The text provides a few more practical ideas on the direction the technology might continue to take.)

A. Brogi, S. Forti, C. Guerrero, and I. Lera, “How to place your apps in the fog: State of the art and open challenges,” Softw: Pract Exper, vol. 50, n 5, pp. 719–740, May 2020, doi: 10.1002/spe.2766.

(In edge or fog computing, there are a number of scattered servers to which one or more applications are to be deployed. The resulting "Fog Application Placement Problem (FAPP)" is a heavily researched topic. The paper shows on page 5 a review of the various papers that attempt to solve the FAPP problem.)

Sources (2 / 4)

E. Jonas et al., “Cloud Programming Simplified: A Berkeley View on Serverless Computing.” arXiv, Feb. 09, 2019. Accessed: Dec. 05, 2022. [Online]. Available: http://arxiv.org/abs/1902.03383
(General evaluation of the technology and its current problems)

P. Aditya et al., "Will Serverless Computing Revolutionize NFV?," in Proceedings of the IEEE, vol. 107, no. 4, pp. 667-678, April 2019, doi: 10.1109/JPROC.2019.2898101.
(The paper is about running virtualized network functions like those of the 5G core on serverless computing.)

H. Shafiei, A. Khonsari, and P. Mousavi, “Serverless Computing: A Survey of Opportunities, Challenges, and Applications,” ACM Comput. Surv., vol. 54, no. 11s, pp. 1–32, Jan. 2022, doi: 10.1145/3510611.
(Another overview)

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

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

04. GitHub. (17. Mai 2010). Abgerufen am 9. Juni 2021
von https://github.com/apache/openwhisk/blob/master/docs/actions.md

Sources (3 / 4)

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

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

07. 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/

08. Amazon (Hrsg.). (kein Datum). *aws*. Abgerufen am 9. Juni 2021 von
https://github.com/apache/openwhisk/blob/master/docs/actions-python.md

09. 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

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

11. Microsoft (Hrsg.). (kein Datum). *aws*. Abgerufen am 9. Juni 2021 von
https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=csharp

12. Microsoft (Hrsg.). (kein Datum). *aws*. Abgerufen am 9. Juni 2021 von
https://github.com/Azure/durabletask/wiki/Core-Concepts

Sources (4 / 4)

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

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

15. **Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing.** Proc. ACM Program. Lang. 3, OOPSLA, Article 149 (October 2019), 26 pages. https://doi.org/10.1145/3360575

16. **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

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

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

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