Natural Resources Canada | Ressources naturelles Canada

Microservices: APIs, Lambdas and FAAS

λ Glen Newton λ

Section Head, Scientific Computing
Geological Survey of Canada
Natural Resources Canada

NRCan Sectors Cloud Working Group
2021 10 28

Canada

Introduction

- FAAS (Function-as-a-Service) in the X-as-a-service universe
- What is FAAS and how does it work?
- How useful is it / what are the use cases?
- Deep dive using AWS Lambda
- What are the advantages/disadvantages?

What is FAAS? Wikipedia: FAAS

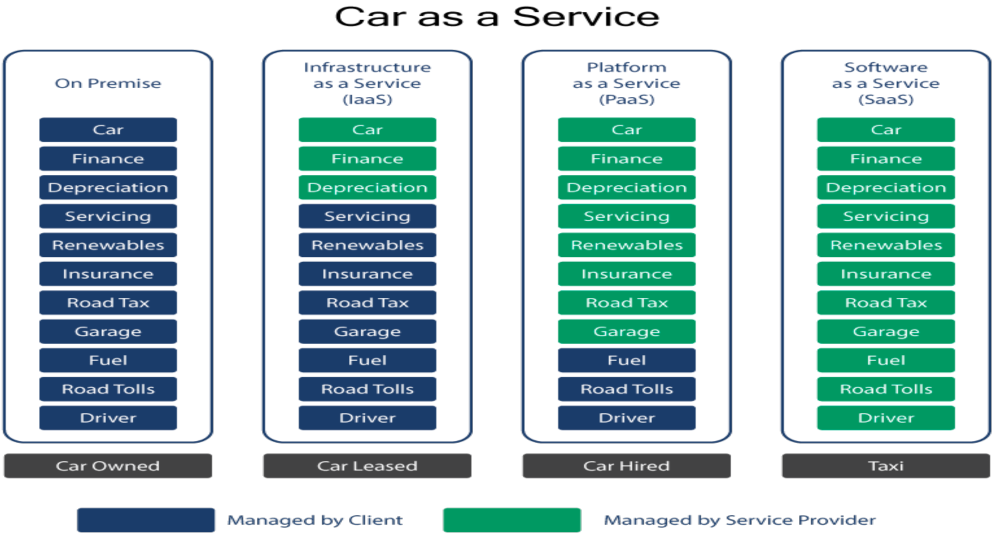
"Function as a service (FaaS) is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app. Building an application following this model is one way of achieving a "serverless" architecture, and is typically used when building microservices applications.

FaaS was initially offered by various start-ups circa 2010, such as PiCloud.

AWS Lambda was the first FaaS offering by a large public cloud vendor, followed by Google Cloud Functions, Microsoft Azure Functions, IBM/Apache's OpenWhisk (open source) in 2016 and Oracle Cloud Fn (open source) in 2017."

Source: `https://en.wikipedia.org/wiki/Function_as_a_service`

Car as a Service



Source: `https://rajeevkuruganti.com/2019/cloud-iaas-paas-saas-and-more/`

Hosting Models: Traditional, Cloud, Cloud Native

Hosting Model

Traditional IT		Cloud	Cloud Native			
On-prem	Co-located	IaaS	CaaS	PaaS	FaaS	SaaS
Applications	Applications	Applications	Applications	Applications	Applications	Applications
Functons	Functons	Functons	Functons	Functons	Functions	Functions
Runtimes and Containers	Runtimes and Containers	Runtimes and Containers	Runtimes and Containers	Runtimes and Container	Runtimes and Container	Runtimes and Container
Management Tools and Operating System	Management Tools and Operating System	Management Tools and Operating System	Management Tools and Operating System	Management Tools and Operating System	Management Tools and Operating System	Management Tools and Operating System
Servers, Storage and Networking	Servers, Storage and Networking	Servers, Storage and Networking	Servers, Storage and Networking	Servers, Storage and Networking	Servers, Storage and Networking	Servers, Storage and Networking
Datacentre	Datacentre	Datacentre	Datacentre	Datacentre	Datacentre	Datacentre

More Control ⟵ ⟶ Less Control

Key

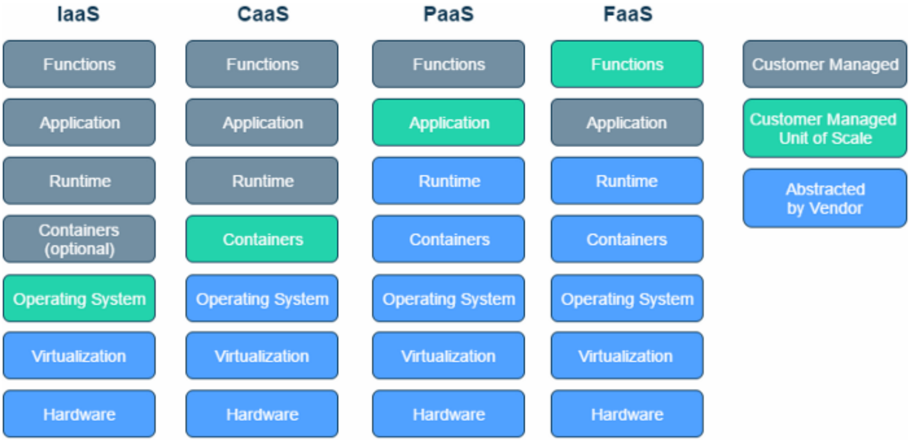
- You Manage
- Provider Manages

Examples

On-prem	Co-located	IaaS	CaaS	PaaS	FaaS	SaaS
• Owning your own datacentre	• Co-location Hosting • Physical Server Leasing	• AWS EC2 • Azure Virtual Machines • Google Compute Engine	• Azure Kubernetes Service • Google Kubernetes Engine • Amazon Elastic Kubernetes Services • OpenShift Dedicated	• AWS Elastic Beanstalk • Azure WebApps • Google App Engine • Heroku • Pivotal Cloud Foundry	• Azure Functions • AWS Lambda • Google Cloud Functions	• Salesforce • Office 365 • Google Business • Oracle • SAP

Source: `https://devops.stackexchange.com/questions/5688/how-can-caas-paas-and-faas-users-know-if-the-operating-system-of-their-server`

Customer-managed unit of scale



Source: `https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362`

FAAS Characteristics 1/2

- Serverless: server completely abstracted / hidden from customer: No hardware, server, VM, OS, etc to manage
- Focus on code rather than infrastructure
- Basically short-lived functions that do a specific task
- Pay only for use / run only when needed
- Event driven

FAAS Characteristics 2/2

- Scalable by design
- Concurrent by design
- Reliable / redundant
- Provisioning/deployment (usually) simple
- Secure: Functions (should) have IAM roles that allow them to do exactly alert(and only) what they need to do.

FAAS Characteristics: Software developer best practices

- Functions (should) perform only one action
- Small, well-defined and specific scope
- Isolation of concerns
- Stateless (with exceptions)
- Limit dependencies / reduce deployment package size. For some languages, the loading of libraries will add to latency
- Within functions, separate cloud vendor-specific handler code from function business logic. Allows for unit testing and (somewhat) limits vendor lock-in.

FAAS/Serverless: Best Practices: Some references

- AWS *Best practices for working with AWS Lambda functions*
- AWS *AWS Lambda Serverless Coding Best Practices*
- Azure *Best practices for reliable Azure Functions*
- Azure *How, when, and why to use Microsoft Azure Cloud Services Azure Functions*
- *Serverless Architectures*
- *Applying Microservice Patterns & Best Practices To FaaS (Part 1 - FaaS Overview)*

AWS Lambda: Case Study

- AWS Lambda = AWS FAAS
- Supports multiple languages: Java, Go, PowerShell, Node.js, C#, Python,Ruby + custom runtimes (any programming language)
- Packaging: zip file or container
- Event driven
- Pay per use (execution time * memory): 1ms billing granularity

AWS Lambda: Default limits 1/2

- Local *ephemeral* storage `/tmp`: 500 MB
- Max concurrency: 1000 instances / region
- Memory: 128 MB to 10 GB
- CPU: Proportional to memory (see below)
- Mx running time (timeout): 15min

AWS Lambda: Default limits 2/2

- Invocation payload: 6 MB (synchronous); 256 KB (asynchronous)
- Deployment package (zip) size: 50 MB (zipped), 250 MB (unzipped)
- Container image code package size limit: 10 GB
- (As of June 2020): Lambdas can mount EFS (elastic file system)
  - This allows lambdas to be stateful across invocations and lambdas, with state stored in EFS volume
  - Mounting adds to cold start latency (see below): "hundreds of milliseconds" of latency
  - If using EFS, lambda needs to be in the same VPC (virtual private cloud) as the EFS volume

AWS Lambda: Proportion of vCPU to memory

Memory	**# vCPU**
128 MB	1 vCPU
832 MB	2 vCPU
3 GB	3 vCPU
5.3 GB	4 vCPU
7 GB	5 vCPU
8.8+ GB	6 vCPU

Go language lambda boilerplate

```
package main

import (
        "fmt"
        "context"
        "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
        Name string `json:"name"`
}

func HandleRequest(ctx context.Context, name MyEvent) (string, error) {
        return fmt.Sprintf("Hello %s!", name.Name ), nil
}

func main() {
        lambda.Start(HandleRequest)
}
```

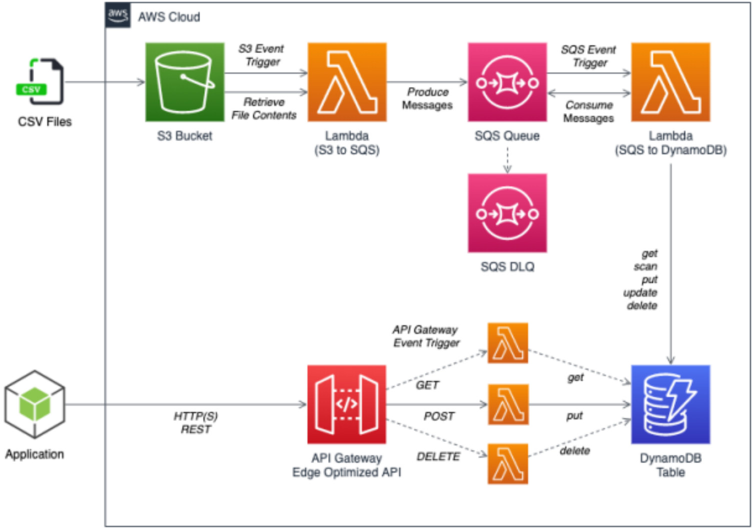
Source: https://docs.aws.amazon.com/lambda/latest/dg/golang-handler.html

Python language lambda boilerplate

```
def lambda_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'], event['last_name'])
    return {
        'message' : message
    }
```

Source: https://docs.aws.amazon.com/lambda/latest/dg/python-handler.html

Example architecture showing both HTTP and S3 events



Source: `https://programmaticponderings.com/2019/10/04/event-driven-serverless-architectures-with-aws-lambda-sqs-dynamodb-and-api-gateway/`

Lambda containers

- Supported containers: Docker or Open Container Initiative (OCI)
- Deploy from Amazon Elastic Container Registry (ECR)
- Many AWS-supplied base image options with pre-installed runtimes
- AWS - Creating Lambda container images
- Azure Functions *do* support Docker (but only for certain hosting plans)
- Google Cloud has Cloud Run, basically like AWS Lambda containers

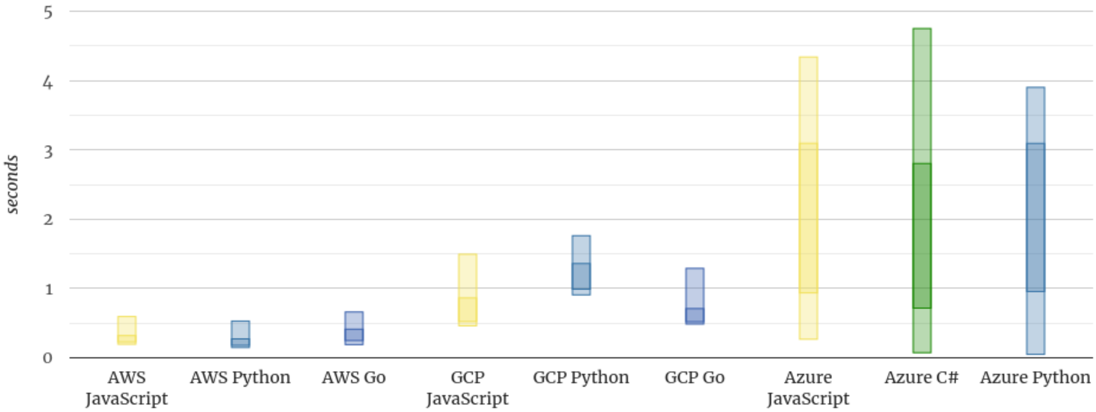
Cold start/latency

When the Lambda is invoked, the following steps take place, each taking time and adding to the latency of the function call:

1. Code download (zip from S3 or container from ECR)
2. Start execution environment
3. Run initialization code
4. Run handler code

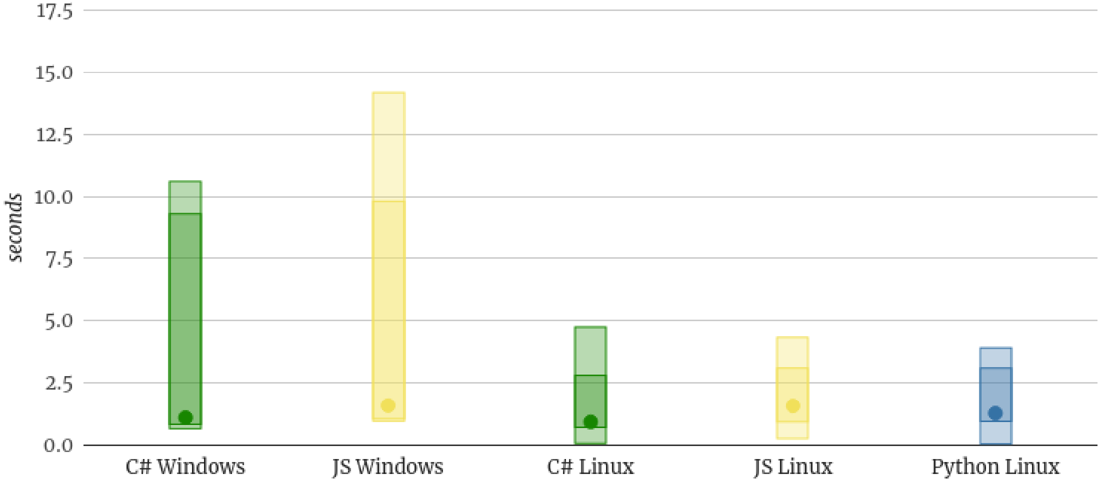
#1 + #2 are the cold start.
Cold start can be avoided using provisioned concurrency.
Full explanation see Operating Lambda: Performance optimization – Part 1

FAAS Cold Start Comparison: AWS, Azure, GCP



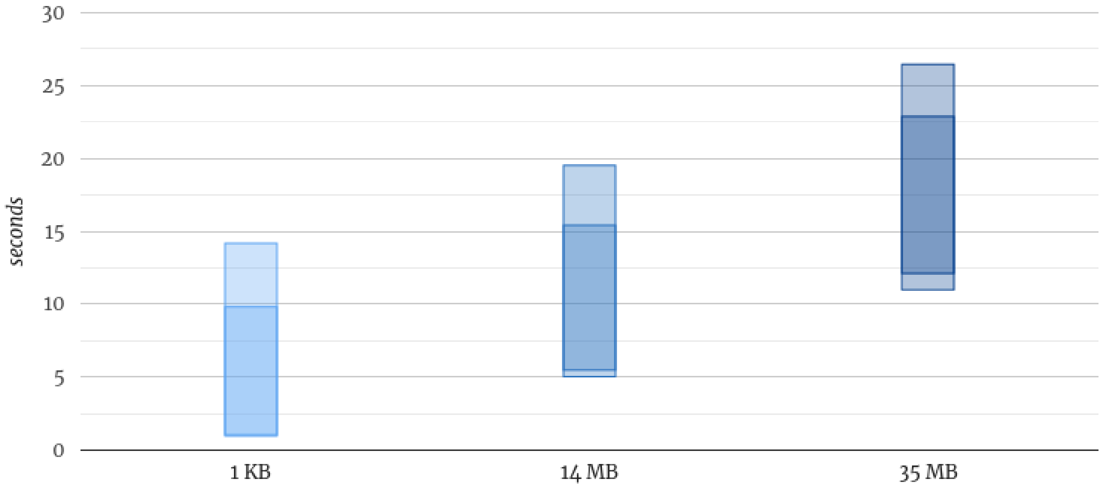
Source: `https://mikhail.io/serverless/coldstarts/big3/`

Cold Starts in Azure Functions: Windows vs. Linux



Source: `https://mikhail.io/serverless/coldstarts/azure/`

Cold Starts in Azure Functions: Impact of Package Size



Source: `https://mikhail.io/serverless/coldstarts/azure/`

Cold start/latency references

1. GCP *Google Introduces Minimum Instances to Reduce Cold Starts*
2. Comparison *Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP*
3. Azure Cold Starts in Azure Functions

Lambda Lifecycle 1/2

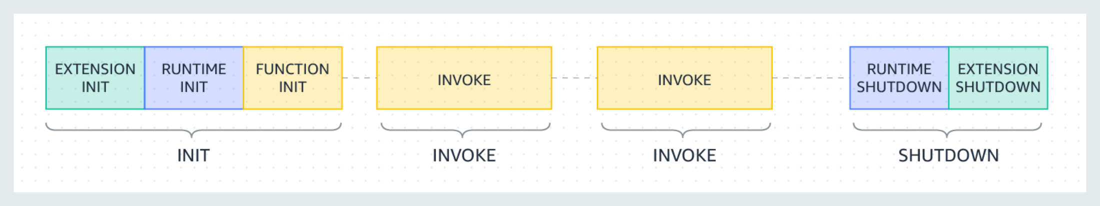
The AWS lifecycle phases are:

1. Init: Happens at first invocation, or, with provisioned concurrency, in advance.
   1.1 Extension init
   1.2 Runtime init
   1.3 Function init
2. Invoke
3. Shutdown
   3.1 Runtime shutdown
   3.2 Extension shutdown

Lambda Lifecycle 2/3

- When a lambda *instance* has responded to an event and completed, it is not immediately removed/deleted: its environment is kept for an undefined period of time.
- If the lambda is reinvoked during this time, this *instance* may be used to respond to the request. If this happens, there is no cold start.
- This experiment from Sept 2020 suggests that lambdas are *"... terminated if a function isn't invoked for 15 minutes ... didn't find the 'guaranteed to be cut-off' time, but it's somewhere between 10 and 15 minutes of inactivity."* – Caveat emptor
- AWS suggests that lambda instances can *cache* data in their `/tmp` directories, just in case they are reinvoked. Additional function logic is needed to take advantage of this situation. If provisioned concurrency is used, then this strategy is a more viable option.

Lambda Lifecycle 3/3



Source: `https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html`

Cold start for different languages (ms), different memory

Memory	Java	Graalvm	.Net	Go	Rust	Python	NodeJS	Ruby
128mb	OOM	1480	11810	1050	844	641	1190	773
256mb	6570	774	5820	661	480	527	769	612
512mb	5180	684	2940	404	304	502	771	677
1024mb	4450	531	1500	299	234	482	656	652
10240mb	2790	501	904	327	219	449	518	649

Source: https://filia-aleks.medium.com/benchmarking-all-aws-lambda-runtimes-in-2021-cold-start-part-1-e4146fe89385

128 MB Average Warm Start (ms)

2021-09-16 21:12:00 UTC

1.	Python	**44.8319076849**
2.	NodeJs	**41.761487337**
3.	GraalVM	**33.3430563774**
4.	Ruby	**27.5446111106**
5.	.Net	**6.99679279171**
6.	Rust	**6.67973684026**
7.	Golang	**6.13062841407**
8.	Java	**-**

2021-09-16 21:13:00 UTC

1.	Python	**47.8771339427**
2.	NodeJs	**39.3268922945**
3.	GraalVM	**35.3736936889**
4.	Ruby	**26.276720429**
5.	.Net	**7.318939927**
6.	Rust	**6.3589843724**
7.	Golang	**5.42087533061**
8.	Java	**-**

2021-09-16 21:18:00 UTC

1.	Python	**47.9682297966**
2.	GraalVM	**39.5942724384**
3.	NodeJs	**38.4373475381**
4.	Ruby	**27.9844864849**
5.	.Net	**7.08470692504**
6.	Rust	**6.64018420763**
7.	Golang	**5.91466843302**
8.	Java	**-**

Source: `https://filia-aleks.medium.com/aws-lambda-battle-2021-performance-comparison-for-all-languages-c1b441005fd1`

Lambda cost exercise 1/4

Example *modest* REST lambda microservice:

- 60 requests / minute (~86k req/day; ~2.6m req/month)
- 512 MB lambda memory (1vCPU)
- 2 second (2000ms) average run time
- AWS Region: Canada (Central)

Lambda cost exercise 2/4: Lambda *only* costs, per month

Using AWS Pricing Calculator:

- `2,592,000 requests x 2,000 ms x 0.001 ms to sec conversion factor = 5,184,000.00 total compute (seconds)`
- `0.50 GB x 5,184,000.00 seconds = 2,592,000.00 total compute (GB-s)`
- `2,592,000.00 GB-s x 0.0000166667 USD = 43.20 USD (monthly compute charges)`
- `2,592,000 requests x 0.0000002 USD = 0.52 USD (monthly request) charges)`

`$43.20 USD + $0.52 USD = $43.72 USD ($54.14 CAD)`

Lambda cost exercise 3/4: API Gateway costs

Using AWS Pricing Calculator:

- `2.592 requests x 1,000,000 unit multiplier = 2,592,000 total REST API requests`
- `Tiered price for:  2592000 requests`
- `2592000 requests x 0.0000035000 USD = 9.07 USD`
- `Total tier cost = 9.0720 USD (REST API requests)`
- `Tiered price total for REST API requests:  9.072 USD`
- `0 USD per hour x 730 hours in a month = 0.00 USD for cache memory`
- `Dedicated cache memory total price:  0.00 USD`

`API Gateway cost (monthly):  $9.07 USD ($11.23 CAD)`

Lambda cost exercise 4/4: Total costs

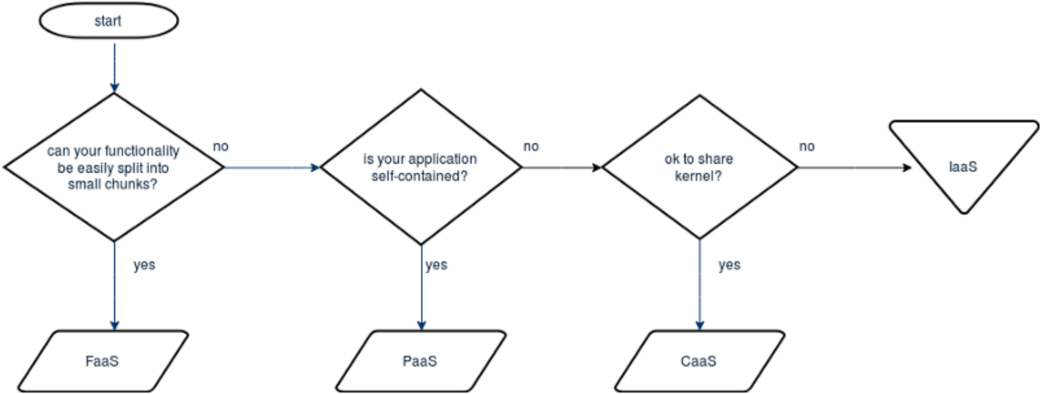
`Total per month (Lambda + API Gateway): $65.37 CAD`

NB: Does not cost-out backend AWS DB costs: most applications would likely have this additional cost. But this would be the same for all implementations, i.e. Lambda, EC2, Fargate, Elastic Kubernetes Service (EKS) etc...

Azure, GCP costing differences

- Azure: billing precision: 1s; Monthly subscriptions
- GCP: billing precision: 100 ms

FAAS Decision Tree (?)



Source: `https://developer.ibm.com/articles/when-to-use-iaas-faas-paas-and-caas/`

FAAS Disadvantages (1/2)

- Needs tuning (to optimize cost vs. performance): Memory and # of vCPUs. Tools exist to automate this: AWS: AWS Lambda Power Tuning. Minor.
- Limits on memory and # vCPUs: max 10 GB RAM and/or 6 vCPUs (AWS Lambda): Does not support many scientific use cases (i.e. large memory work loads, etc).
- Limits on running time (15min for AWS): Too short for some enterprise and scientific workloads. For some use cases, this can be overcome with checkpointing (often used in traditional high performance computing HPC) and lambda pipelining.
- Cold start issues

FAAS Disadvantages (2/2)

- Non-traditional architecture and technologies can challenge some developers / architects
- Stateless (with exceptions), which can be a challenge to designing and architecting
- Vendor lock-in: Not too high a risk: lock-in is more likely with the use of vendor BAAS (backend as a service) services like backend databases, that the lambda uses
- Basic monitoring and debugging : not as mature as other stacks, can be problematic. Improving. See: AWS: Monitoring and observability; Azure: Monitor Azure Functions

FAAS Orchestration

*"Frequently, orchestration is what we actually mean when we are talking about automating. Orchestration is automating many tasks together. It's automation not of a single task but an entire IT-driven process. Orchestrating a process, then, is automating a series of individual tasks to work together. -*BMC Blogs, Stephen Watts, Sept 2020

FAAS Orchestration: AWS

- AWS: Step Functions - *"Step Functions is a serverless orchestration service that lets you combine AWS Lambda functions and other AWS services to build business-critical applications*...Workflows manage failures, retries, parallelization, service integrations, and observability so developers can focus on higher-value business logic."
  - State machine, encoded into JSON
  - API to run and build
  - Visual editor Workflow Studio

FAAS Orchestration: AWS

- Amazon Managed Workflows for Apache Airflow (MWAA)
- (See GCP below)

FAAS Orchestration: Azure

- Durable Functions -"*Durable Functions is an extension of Azure Functions that lets you write stateful functions in a serverless compute environment. The extension lets you define stateful workflows by writing orchestrator functions and stateful entities by writing entity functions using the Azure Functions programming model.*"
- State machine, encoded into programming language, C#, JS, Python, F#, PowerShell
- API to run

FAAS Orchestration: GCP

- Cloud Composer -"*A fully managed workflow orchestration service built on Apache Airflow.*"
  - State machine, Python-centric
  - API to run and update?

NB: AWS also has Apache Airflo-based orchestration (Dec 2020): Amazon Managed Workflows for Apache Airflow (MWAA)