



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

Microservices Interest Group
2021 10 14



- ▶ 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.

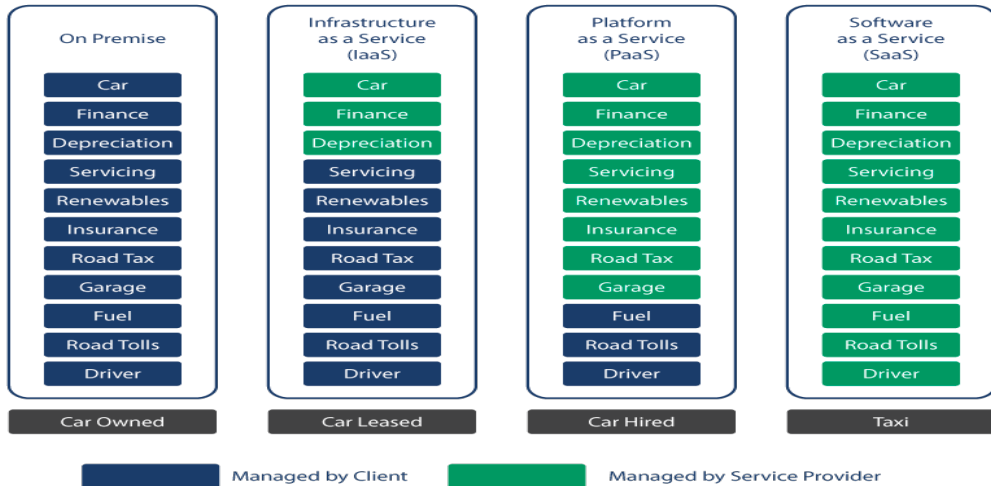
*AWS **Lambda** was the first FaaS offering by a large public cloud vendor, followed by Google **Cloud Functions**, Microsoft **Azure Functions**, IBM/Apache's OpenWhisk (open source) in 2016 and Oracle Cloud Fn (open source) in 2017."*

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

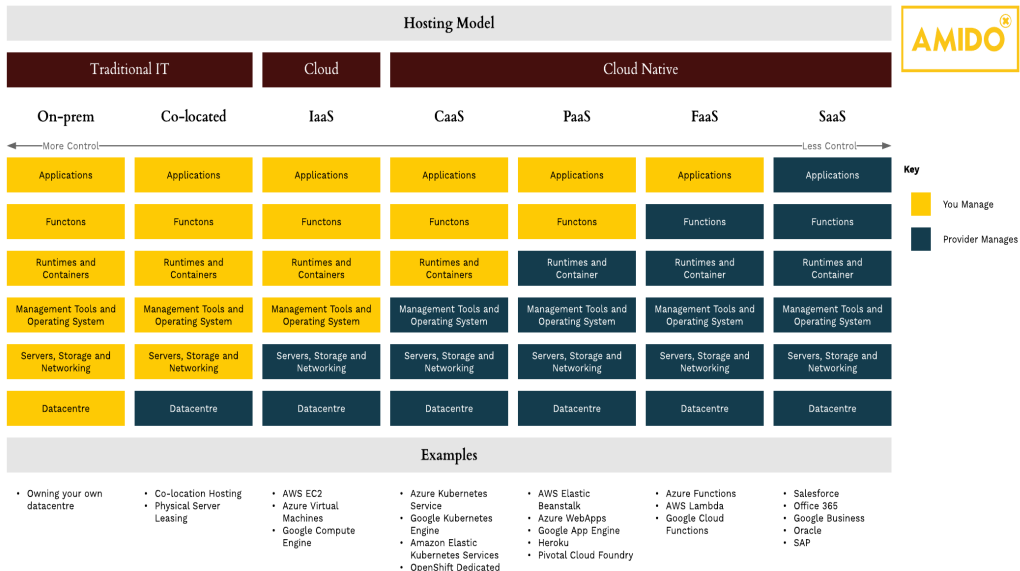




Car as a Service

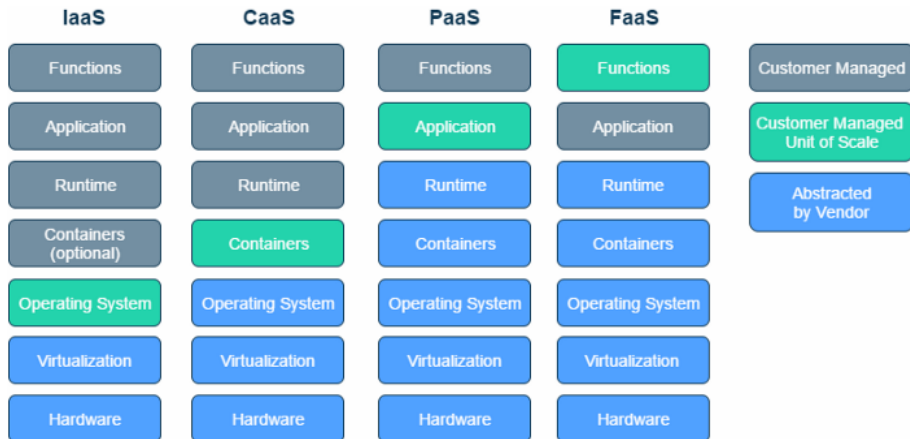


Hosting Models: Traditional, Cloud, Cloud Native



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





- ▶ Serverless: server completely abstracted / hidden from customer: No hardware, server, VM, OS, etc to manage
- ▶ Focus on code rather than infrastructure
- ▶ Short-lived
- ▶ Use only when needed / pay only for use
- ▶ Event driven
- ▶ Scalable by design
- ▶ Concurrent by design





- ▶ Reliable / redundant
- ▶ Provisioning/deployment (usually) simple
- ▶ **Secure**: Functions have as-restrictive-as-possible IAM roles; they only have permissions to do their one activity.





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





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





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

AWS Lambda: Default limits 1/2



- ▶ Local *ephemeral* storage /tmp: 500 MB
- ▶ Max concurrency: 1000 instances
- ▶ Memory: 128 MB to 10 GB
- ▶ CPU: Proportional to memory (see below)
- ▶ Timeout: 15min





- ▶ 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): Mount EFS (elastic file system)
 - ▶ This allows lambdas to be stateful across invocations and lambdas, with state stored in EFS volume
 - ▶ Mount adds to cold start latency (see below): “hundreds of milliseconds” of latency
 - ▶ 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/go-lang-handler.html>



Natural Resources
Canada

Ressources naturelles
Canada

Canada

Python language lambda boilerplate

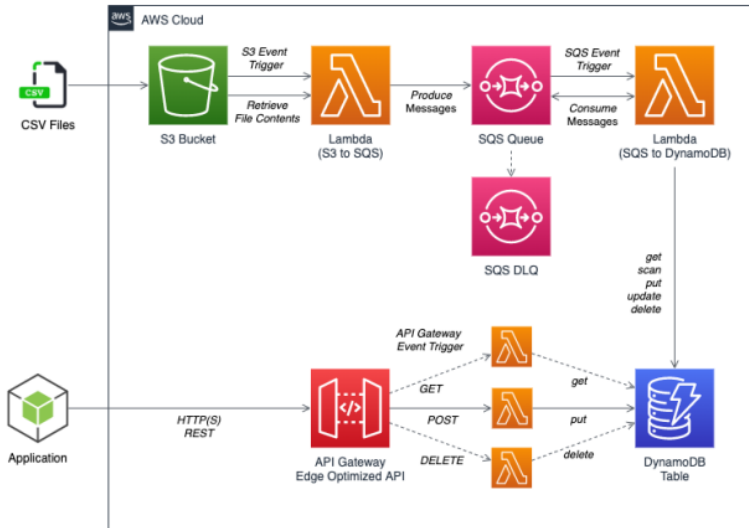


```
def lambda_handler(event, context):  
    message = 'Hello_{ }_{ }!'.format(event[ 'first_name' ], event[ 'last_name' ])  
    return {  
        'message' : message  
    }
```

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



Example architecture showing both HTTP and S3 events





- ▶ 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

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

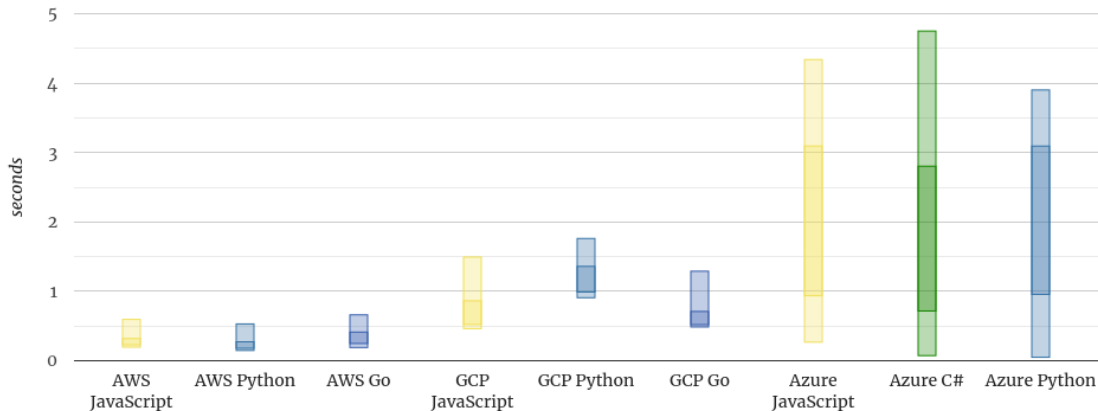
#1 + #2 are the **cold start**.

Cold start can be avoided using provisioned concurrency.

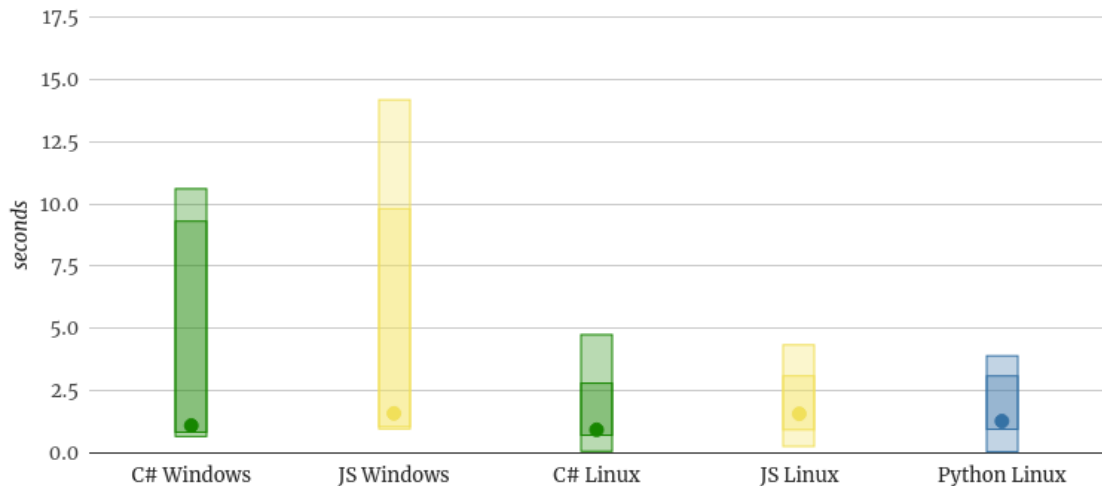
Full explanation see [Operating Lambda: Performance optimization – Part 1](#)



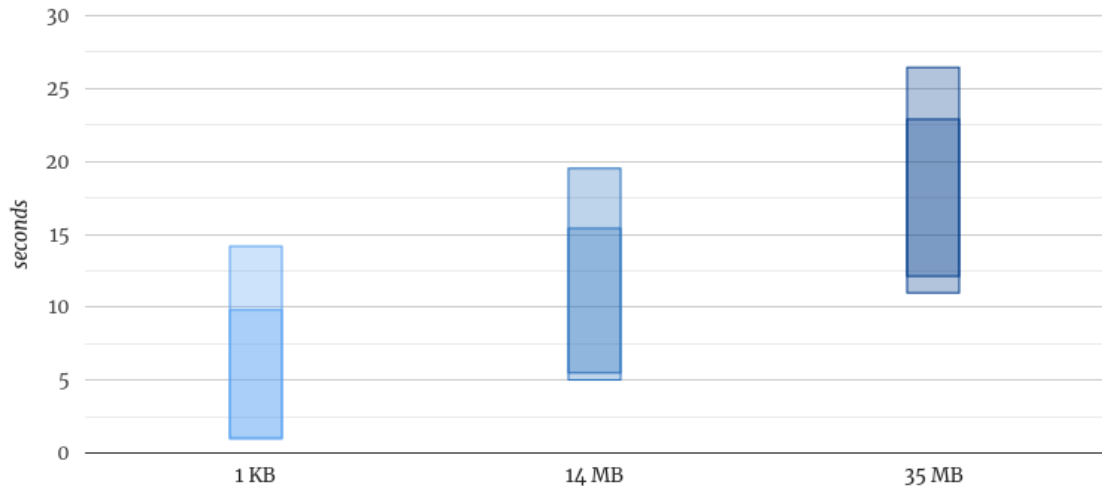
Cold Start Comparison: AWS, Azure, GCP



Cold Starts in Azure Functions: Windows vs. Linux



Cold Starts in Azure Functions: Impact of Package Size





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*





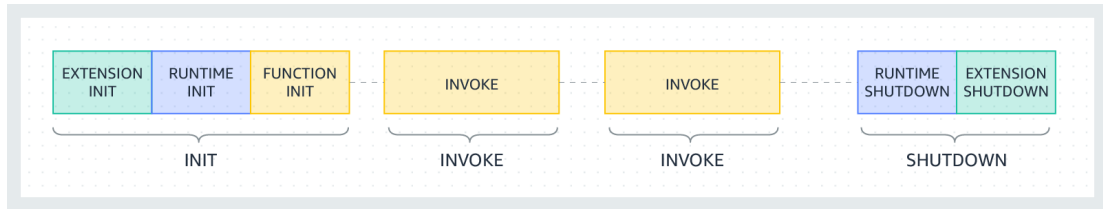
The AWS lifecycle phases are:

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



- ▶ 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



Cold Start for different languages (milliseconds), 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 (milliseconds)











2021-09-16 21:12:00 UTC

1.  Python 44.8319076849
2.  NodeJs 41.761487337
3.  GraaIVM 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.  GraaIVM 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.  GraaIVM 39.5942724384
3.  NodeJs 38.4373475381
4.  Ruby 27.9844864849
5.  .Net 7.08470692504
6.  Rust 6.64018420763
7.  Golang 5.91466843302
8.  Java -



Example modest REST lambda microservice:

- ▶ 60 requests / minute (86,400 request / day)
- ▶ 512 MB lambda memory
- ▶ 2 second (2000ms) average run time
- ▶ AWS Region: Canada (Central)



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



Using AWS Pricing Calculator:

- ▶ $2,592,000 \text{ requests} \times 2,000 \text{ ms} \times 0.001 \text{ ms to sec conversion factor} = 5,184,000.00 \text{ total compute (seconds)}$
- ▶ $0.50 \text{ GB} \times 5,184,000.00 \text{ seconds} = 2,592,000.00 \text{ total compute (GB-s)}$
- ▶ $2,592,000.00 \text{ GB-s} \times 0.0000166667 \text{ USD} = 43.20 \text{ USD (monthly compute charges)}$
- ▶ $2,592,000 \text{ requests} \times 0.0000002 \text{ USD} = 0.52 \text{ USD (monthly request) charges}$

$\$43.20 \text{ USD} + \$0.52 \text{ USD} = \$43.72 \text{ USD} (\$54.14 \text{ CAD})$



Lambda cost exercise 3/4: API Gateway costs



Using AWS Pricing Calculator:

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

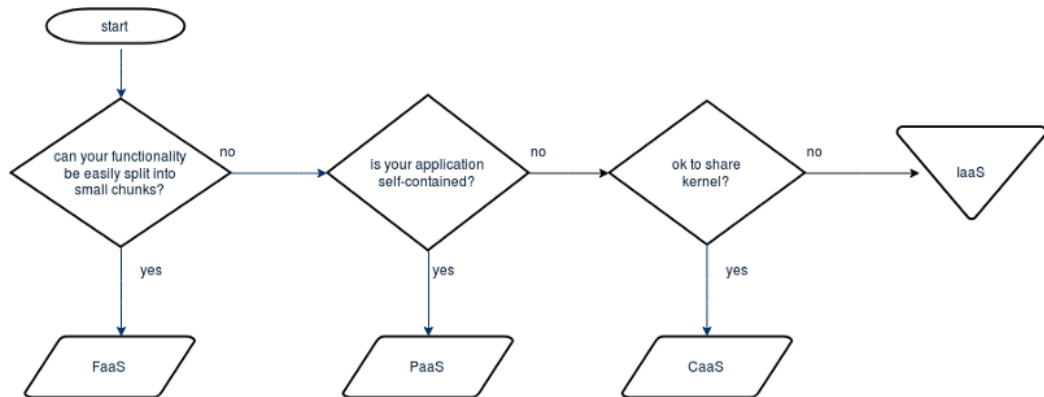


Total per month (Lambda + API Gateway): \$65.37 CAD

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. . .



FAAS Decision Tree (?)





- ▶ Needs tuning (to optimize cost vs. performance): Memory and # of vCPUs. Tools exist to automate this: AWS: [AWS Lambda Power Tuning](#).
- ▶ 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 workloads, 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



- ▶ Non-traditional architecture and technologies can challenge some developers
- ▶ Stateless, 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](#)



- ▶ MS Azure Functions *do* support Docker (but only for certain hosting plans)
- ▶ Google Cloud has Cloud Run, basically like AWS Lambda containers





Canada

Made with: https://github.com/gnewton/nrcan_latex_template



Natural Resources
Canada

Ressources naturelles
Canada

Canada