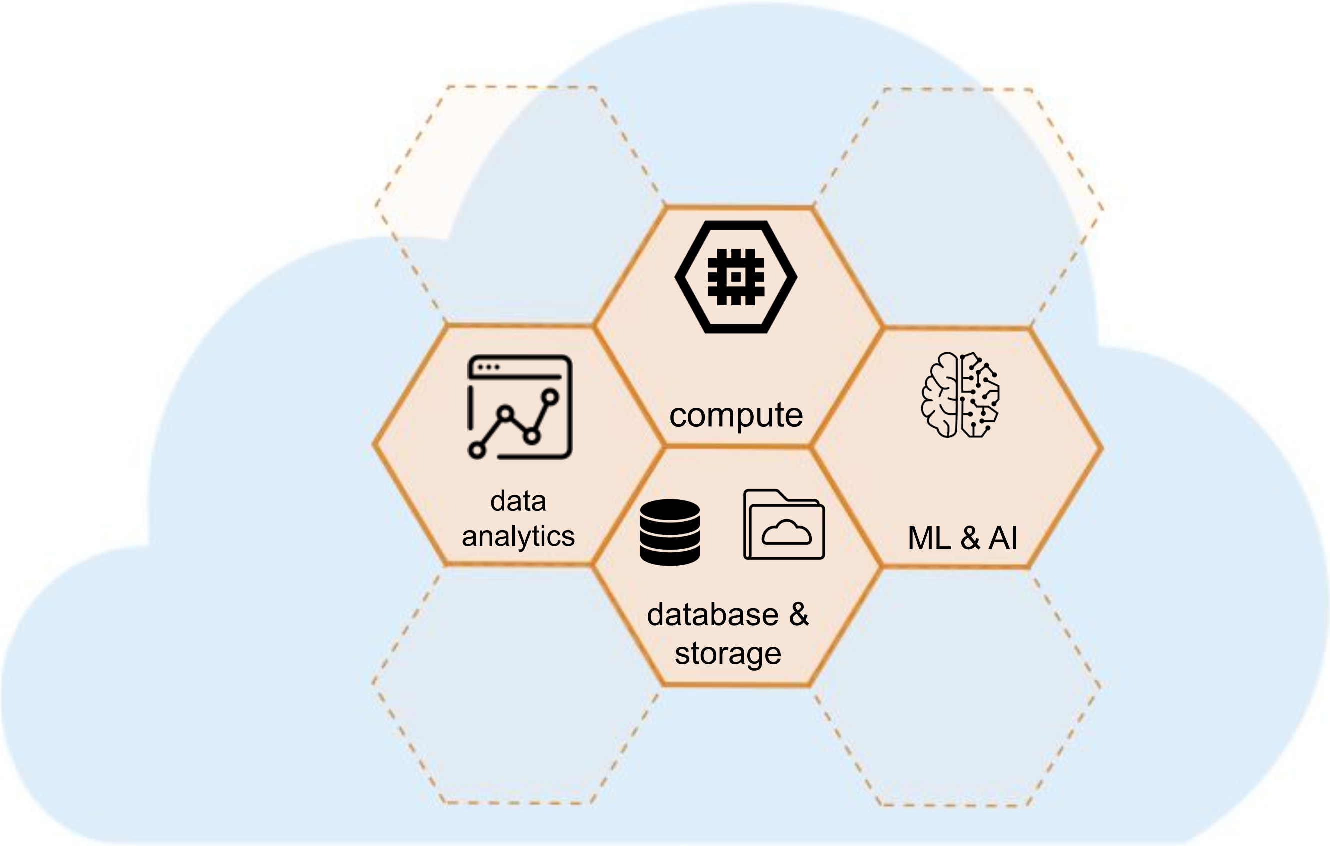
The Serverless Shell

Aurèle MAHEO, Pierre SUTRA

Short Bio

- (2015) PhD, High Performance Computing
    - Thesis topic: “Improving the Hybrid model MPI+Threads through Applications, runtimes and Performance tools”
    - Exascale Computing Research
- (2011) MS, High Performance Computing
- (2008) MS, Computer Science

Cloud ecosystem



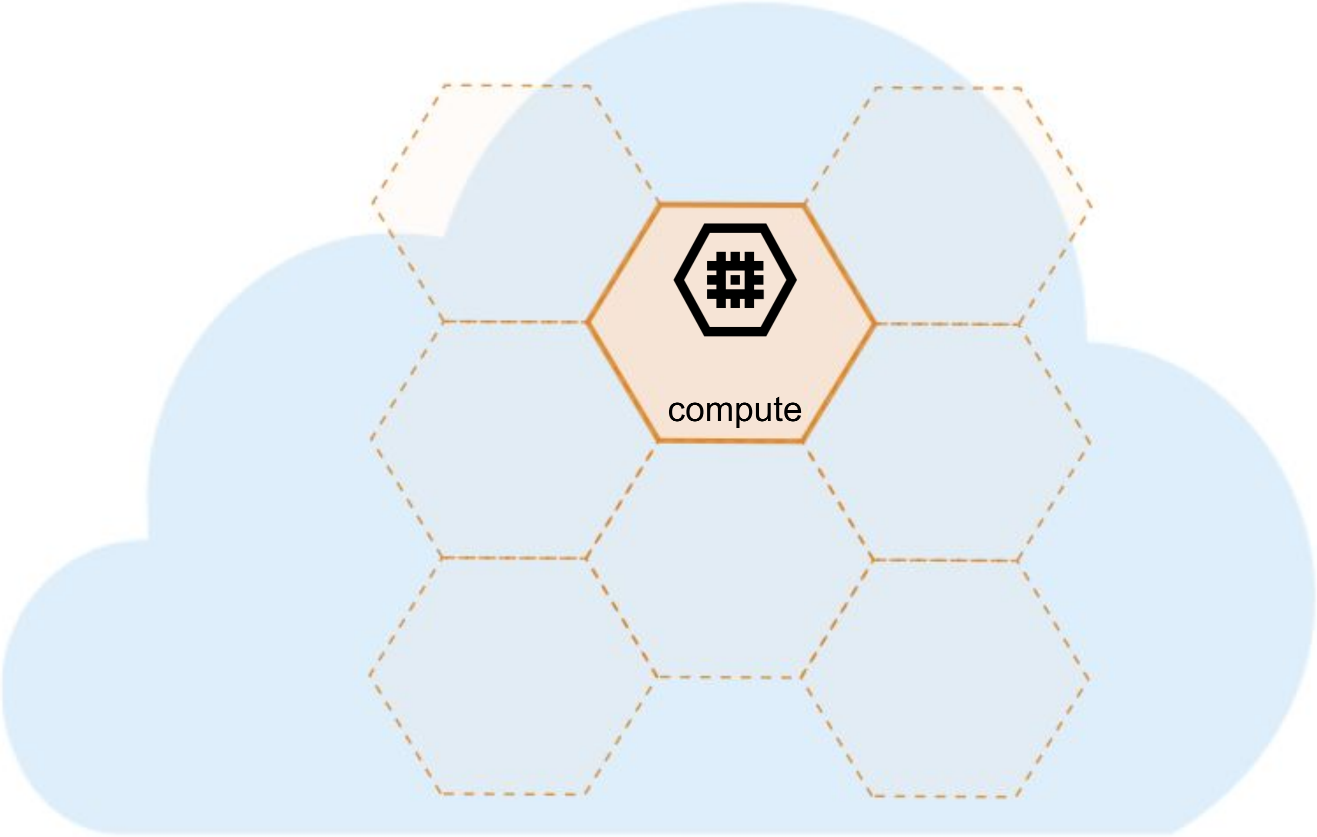
compute

data analytics

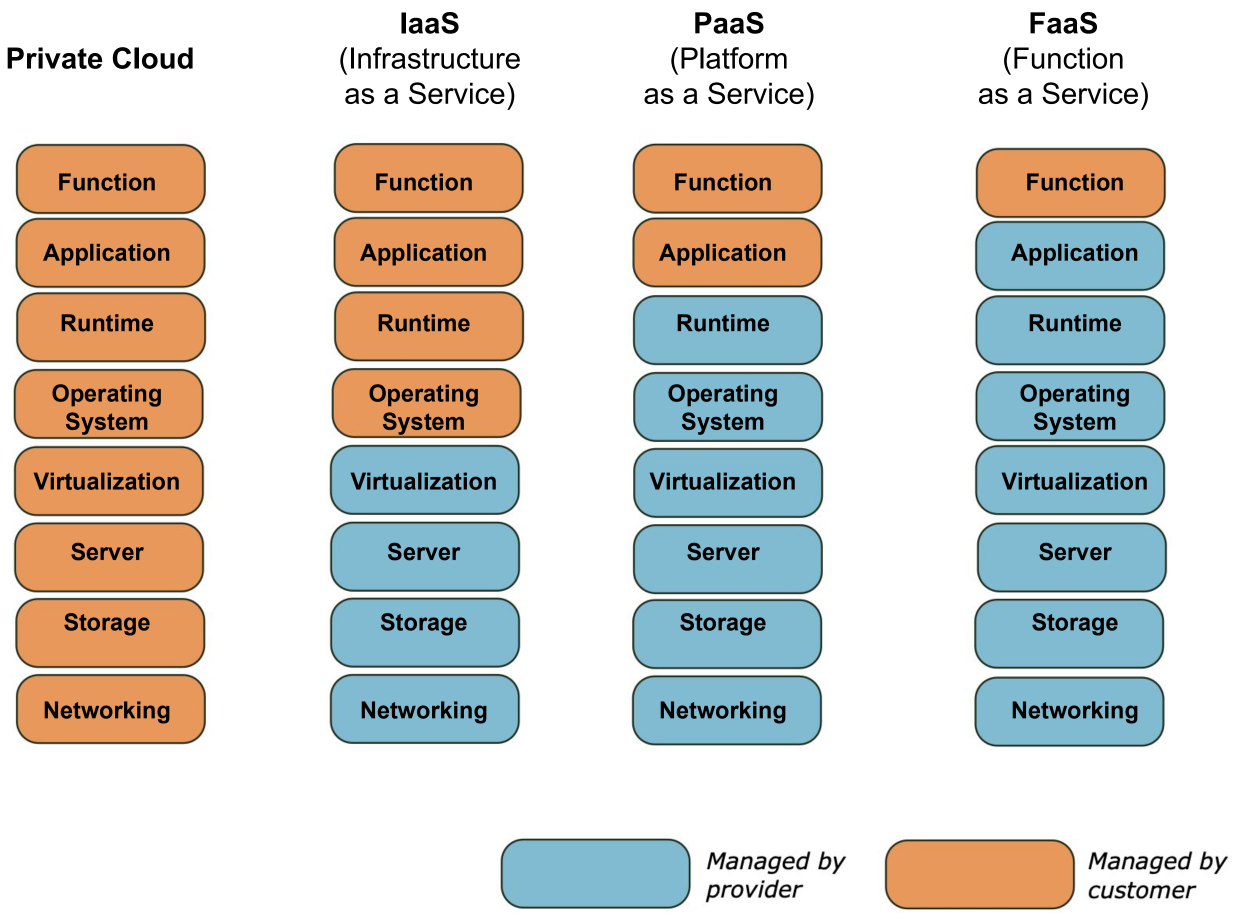
ML & AI

database & storage

Cloud Computing



Cloud Computing models



Private Cloud

Function
Application
Runtime
Operating System
Virtualization
Server
Storage
Networking

IaaS
(Infrastructure as a Service)

Function
Application
Runtime
Operating System
Virtualization
Server
Storage
Networking

PaaS
(Platform as a Service)

Function
Application
Runtime
Operating System
Virtualization
Server
Storage
Networking

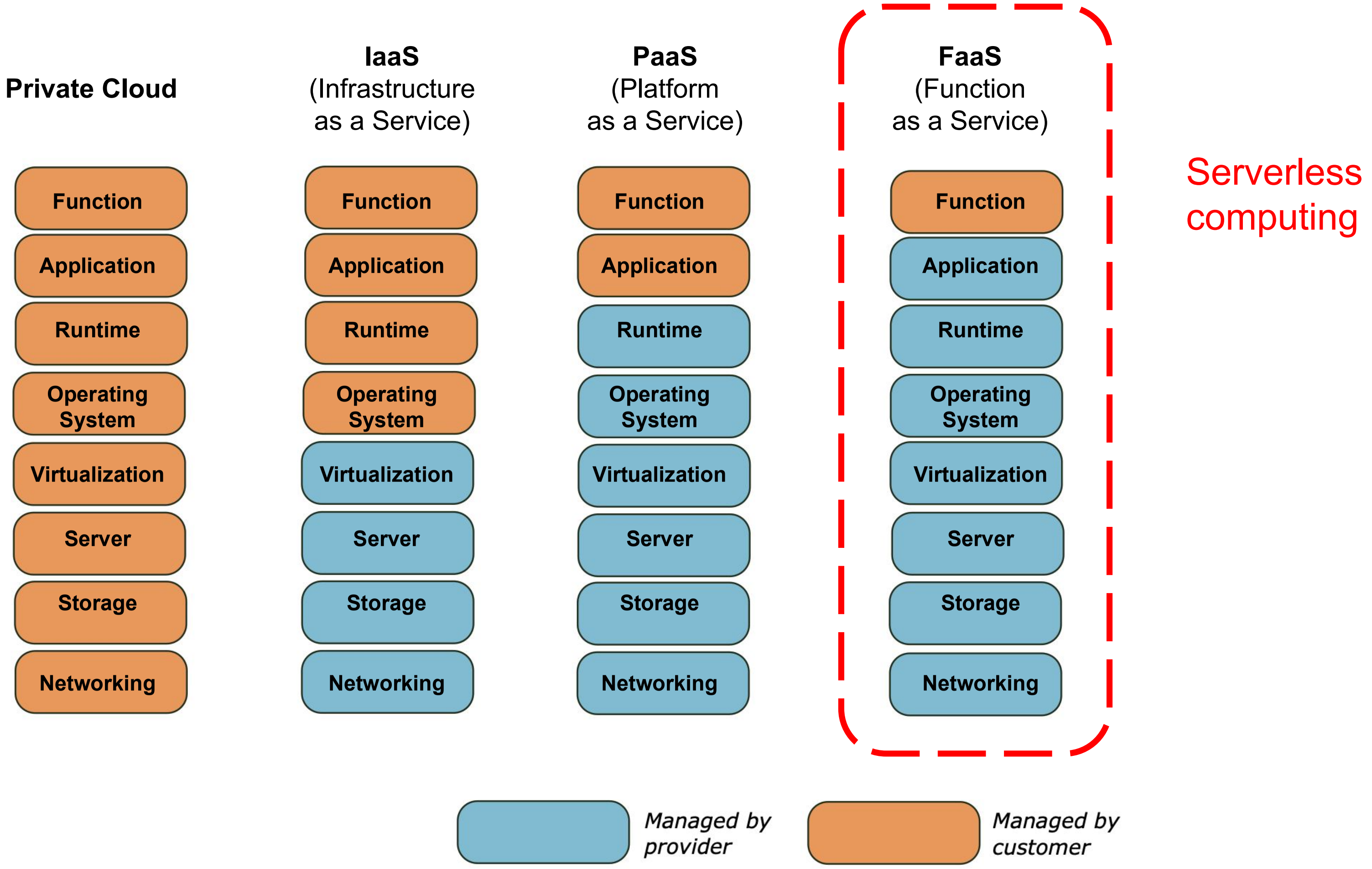
FaaS
(Function as a Service)

Function
Application
Runtime
Operating System
Virtualization
Server
Storage
Networking

Managed by provider

Managed by customer

Cloud Computing models



Private Cloud	**IaaS** (Infrastructure as a Service)	**PaaS** (Platform as a Service)	**FaaS** (Function as a Service)
🟧 Function	🟧 Function	🟧 Function	🟧 Function
🟧 Application	🟧 Application	🟧 Application	🟦 Application
🟧 Runtime	🟧 Runtime	🟦 Runtime	🟦 Runtime
🟧 Operating System	🟧 Operating System	🟦 Operating System	🟦 Operating System
🟧 Virtualization	🟦 Virtualization	🟦 Virtualization	🟦 Virtualization
🟧 Server	🟦 Server	🟦 Server	🟦 Server
🟧 Storage	🟦 Storage	🟦 Storage	🟦 Storage
🟧 Networking	🟦 Networking	🟦 Networking	🟦 Networking

Serverless computing

🟦 *Managed by provider*

🟧 *Managed by customer*

Serverless computing

- Cloud Computing Execution model
- Code execution fully managed by cloud provider
- Seamless server provisioning, administration and maintenance
- Composed of compute runtimes (**F**unction **as a S**ervice)

Serverless computing landscape

AWS Lambda

Google Cloud Functions

Microsoft Azure Functions

Apache OpenWhisk

OpenFaaS

...

Example of AWS Lambda

Processing large datasets with UNIX shell

- Languages for data processing
  - Python
  - UNIX shell
  - R
  - ….

- Why UNIX shell ?

  - Semantically minimal language
  - Very convenient to process data

    ```
    $> curl dataset | cmdA | cmdB | cmdC | … | cmdN
    ```

  - Possible to express parallelism using **GNU Parallel**

    ```
    $> cat input | parallel -j<jobs> cmdA
    ```

Processing large datasets with UNIX shell

- **Limitations**
  - Problems arise when dealing with large datasets
  - Sequential mode: Long processing
  - Parallel mode: Meets bottlenecks ((+100 parallel jobs))
    - Machine is the limit
    - Need of huge local compute resources

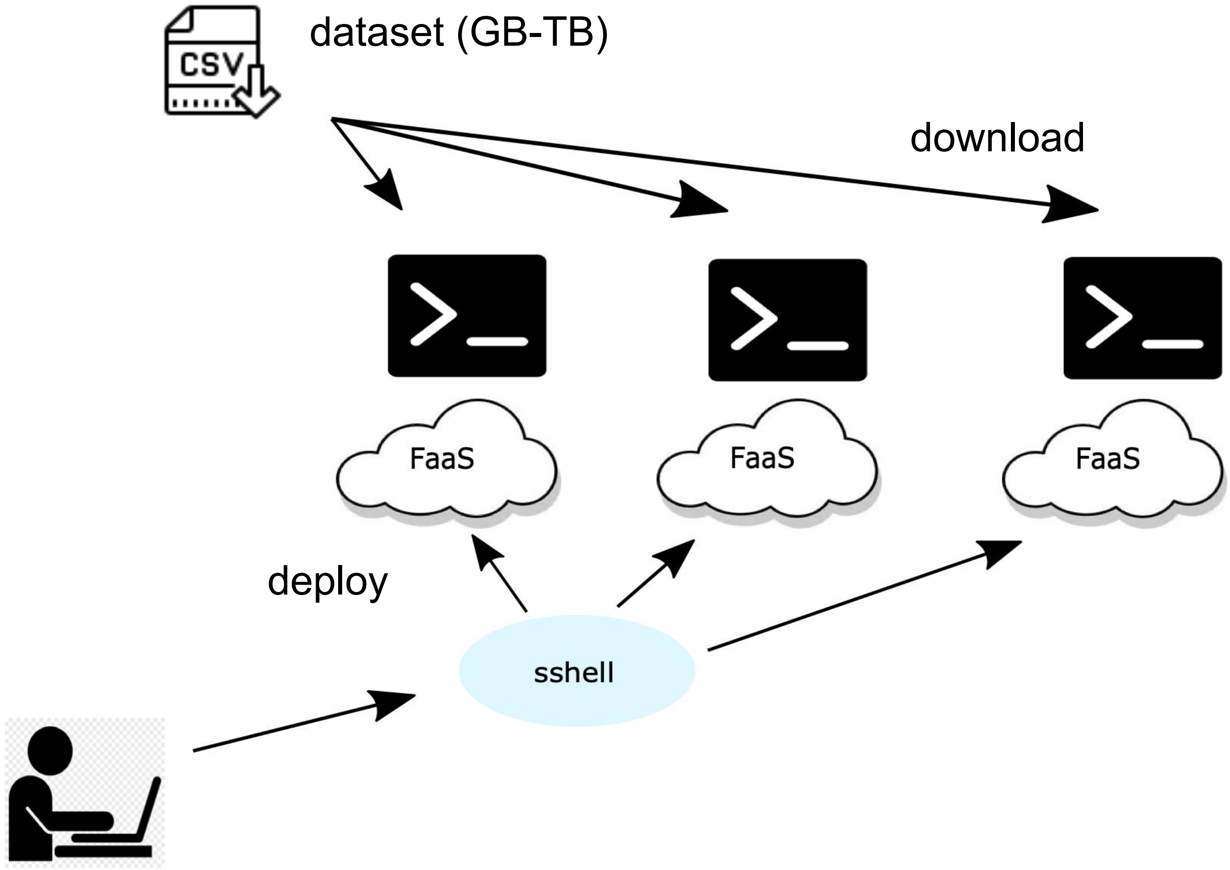
- **Solution & Objectives**
  - Combine the power of the serverless computing and simplicity of UNIX shell
  - Port System mechanisms to the serverless platform
  - Adapt UNIX shell for serverless
  - Augment it to express stateful patterns

The Serverless Shell

- Usage

```
$> sshell ls -C /
  RequestId: d6215a3a-a41c-4384-b779-215cfa06b30c
  Duration: 13 ms Memory Used: 98 MB
  bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
  boot  etc  lib   media  opt  root  sbin  sys  usr
```

The Serverless Shell



Parallel processing

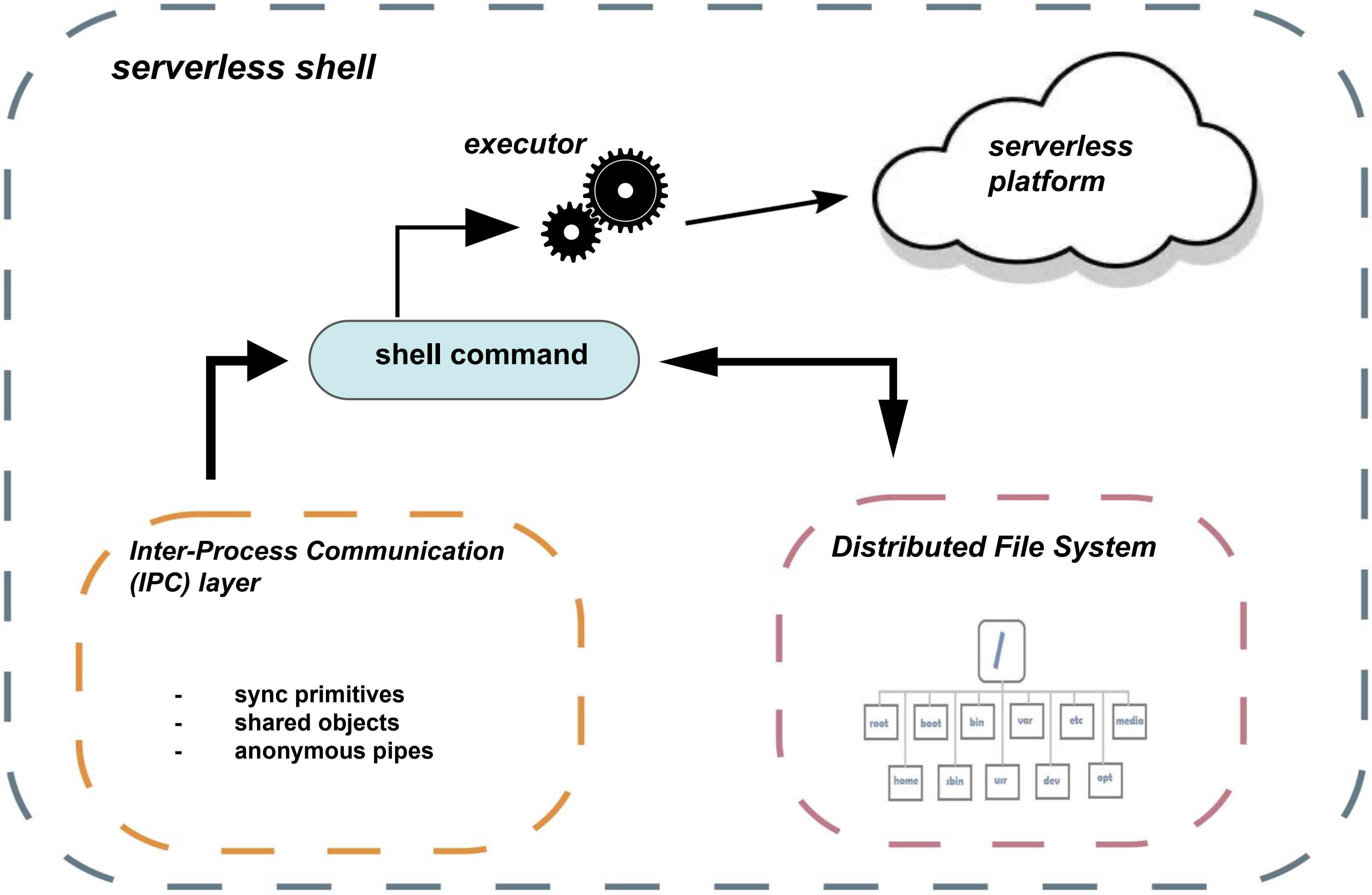
- Calculate average using **sshell**

```
$> TMP_DIR=/tmp/$(whoami)

CCBASE="http://commoncrawl.s3.amazonaws.com"
CCMAIN="CC-MAIN-2019-43" # oct. 2019
RANGE="-r 0-10000000"

curl -s ${CCBASE}/crawl-data/${CCMAIN}/warc.paths.gz \
   | zcat | head -n ${INPUT} > ${TMP_DIR}/index

average(){
   while read l; do
         sshell "curl -s ${RANGE} ${CCBASE}/${l} | 2>/dev/null zcat -q | grep ^Content-Length " &
   done < ${TMP_DIR}/index | awk '{ sum += $2 } END { if (NR > 0) print int(sum / NR) }'
}
```

System design

Serverless shell:

- serverless platform
- executor
- IPC layer
- Distributed file system



serverless shell

executor

serverless platform

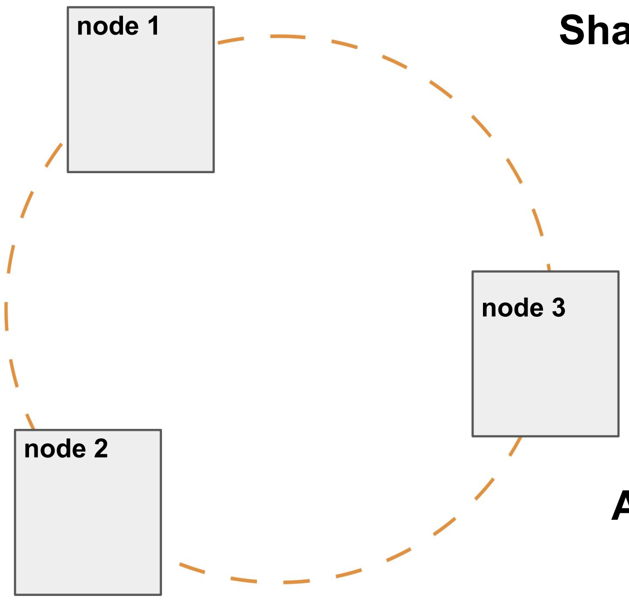
shell command

Inter-Process Communication (IPC) layer

- sync primitives
- shared objects
- anonymous pipes

Distributed File System

System design / IPC layer



Synchronization primitives

$> primitive -n name operation [parameters]

Shared objects

data structures:

- hash tables
- red-black trees
- lists
- arrays
- counters

Anonymous pipes

System design / IPC layer

- def: **Stateful** application Versus **Stateless** application

Stateless	**Stateful**
No use of shared object in application	Use of shared object in application (map, etc)

IPC layer

- sshell example involving synchronization primitive (barrier)

```
$>

JOBS=100
BARRIER=$(uuid)
seq 1 1 $((JOBS-1)) | parallel -n0 sshell --async barrier -n
${BARRIER} await -p ${JOBS}
sshell barrier -n ${BARRIER} await -p ${JOBS}
```

Inter-Process communication

- Anonymous pipes
- Rewrite **“ $> cmdA | cmdB “ :**

```
$>

# direction connection

sshell "nc -N -l 8080 | cmdB& rdv de41a38e -1 $IP" &
sshell "HOST=$(rdv de41a38e); exec 3<>/dev/tcp/${HOST}/8080; cmdA >&3; echo EOF
>&3"
```

```
$>

# file system

sshell "cmdA | awk '{print \$0}END{print \"EOF\"}' > /fs/de41a38e" &
sshell "tail -n +0 --pid=\$\$ -f --retry /fs/de41a38e 2>/dev/null | { sed \"/EOF/ q\" && kill \$\$ ;} |
grep -v ^EOF\$ | cmdB"
```

Implementation

- **Language** : Java
- **SLOC** : ~3K
- **Build system** : Maven
- **Java version** : GraalVM 19.3.0
- **FaaS platform** : AWS Lambda
- **Distributed File system** : AWS Elastic File System (EFS)
- **IPC layer** : Distributed Shared Objects (DSO)

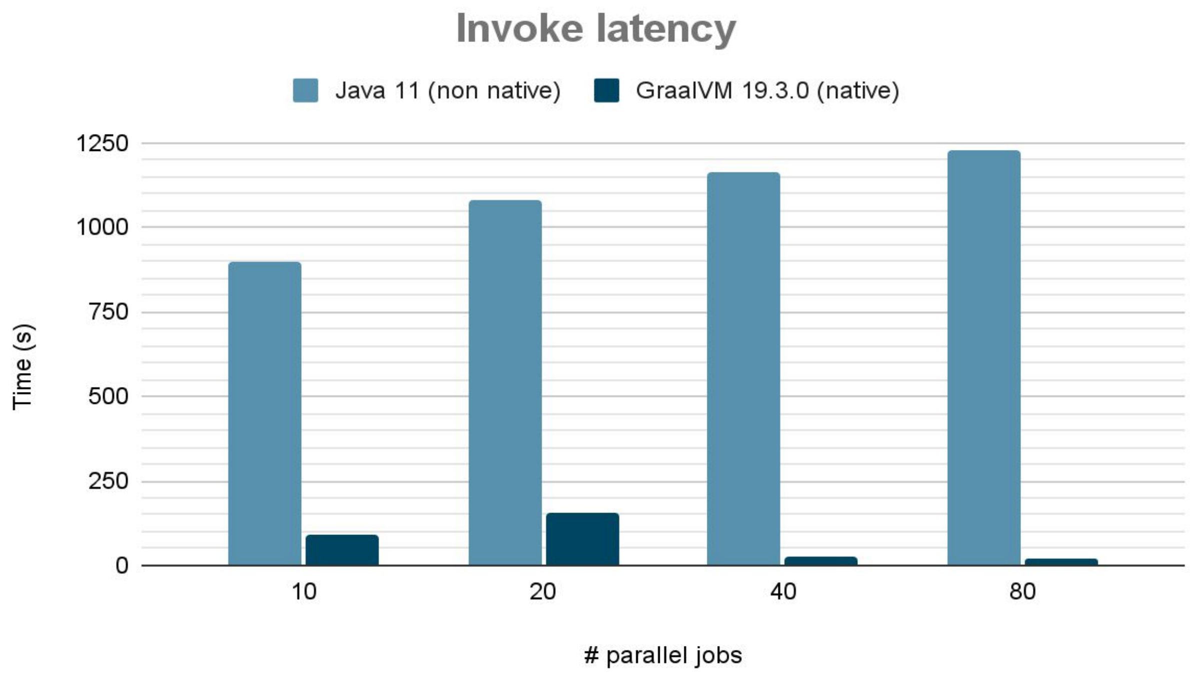
Evaluation

- Set up
  - Experiments conducted from an AWS EC2 machine t2.2xlarge (8 vCPUS - 32 GB RAM)
  - Use default parameters in AWS Lambda
  - Each serverless function: 1 GB of memory

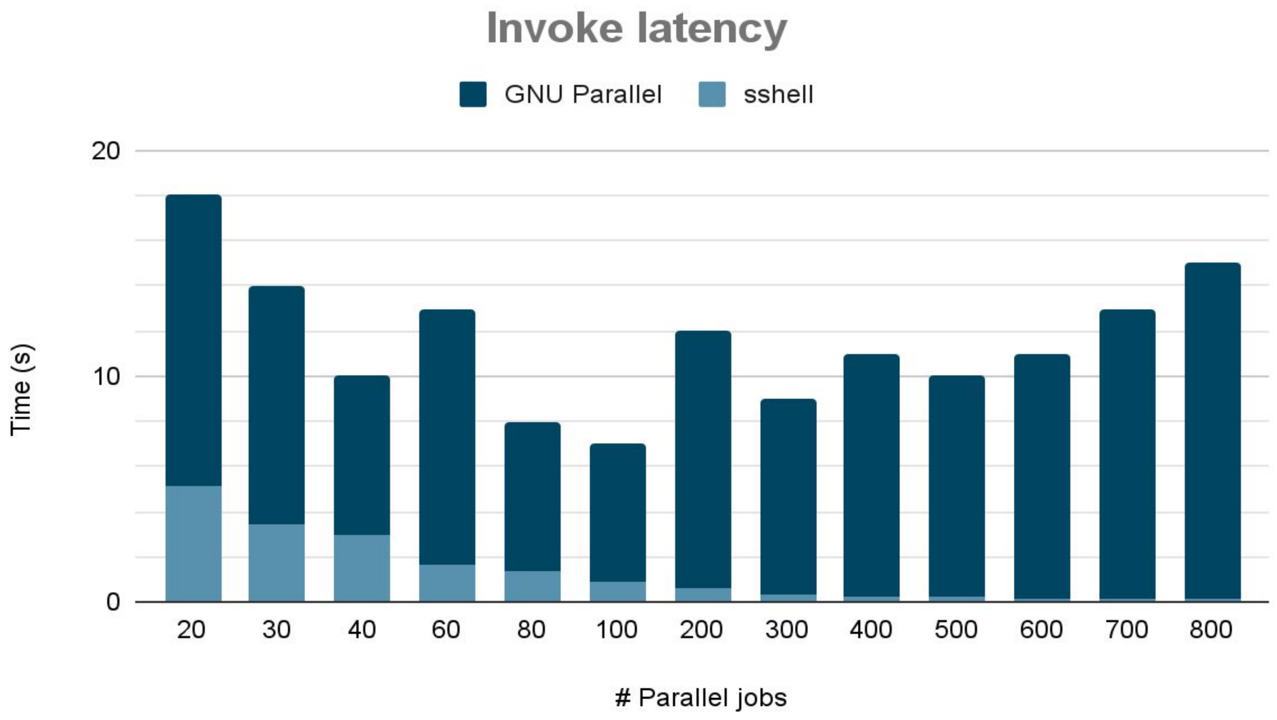
- Performance metrics
  - Invoke latency
  - I/O (AWS S3 and AWS EFS)
  - Compute
  - Sync
  - Sort

Evaluation / Preliminaries

- sshell built with traditional Java SDK VS Native Java SDK (GraalVM 19.3.0)



Evaluation / Preliminaries



Evaluation / Preliminaries

- Peak transfer rate AWS EFS <> AWS Lambda

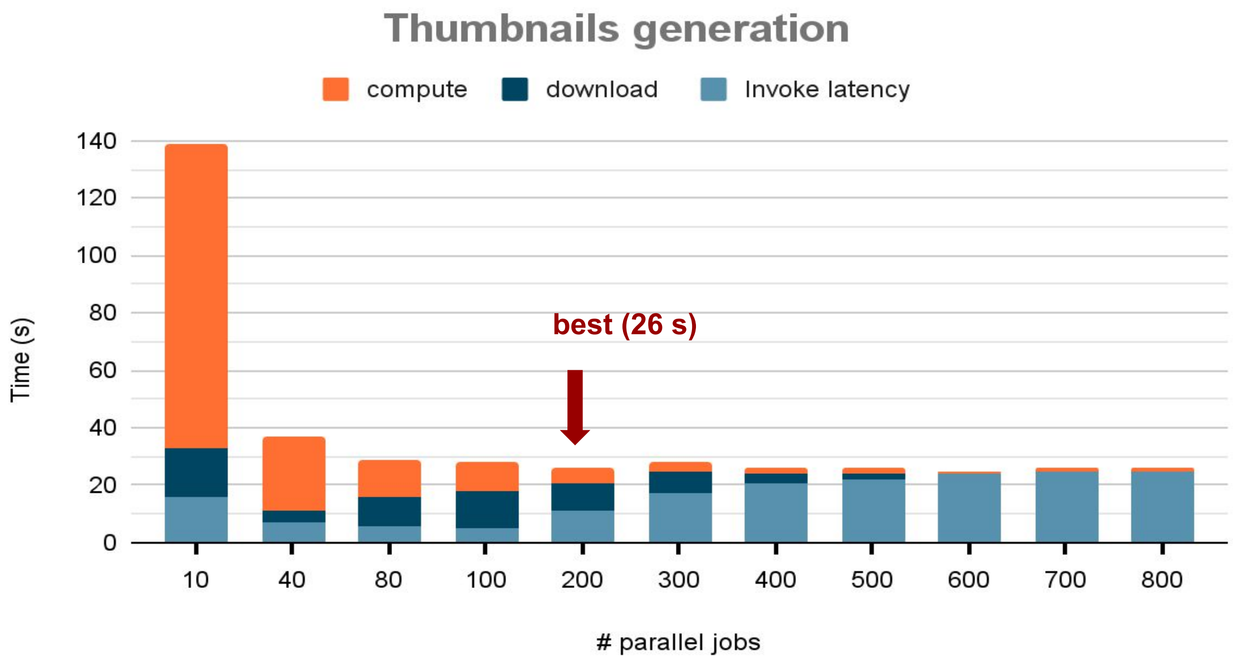
	Download	**Upload**
Sequential	72 MB/s	77 MB/s
Parallel	3418 MB/s	1333 MB/s

Evaluation / Micro Benchmarks

- **Thumbnails generation**
  - Parse a set of 1090 images
  - Generate for each image a 10KB thumbnail
  - Files stored in AWS EFS

- **Port scan analysis**
  - Parse a 40 GB trace containing a full Internet scan of port 80
  - Steps
    - clean raw input data using zannotate
    - isolate Internet Protocol (IP) then Autonomous System (AS)
    - merge the 2 outputs together
    - count the number of IPs and AS

Evaluation / Micro Benchmarks



- **compute component decreases as # parallel jobs increases**
- **Invoke latency increases as # parallel jobs increases**

Evaluation / Micro Benchmarks

- Port Scan analysis - native code version

```
$>
    #  Step 1 - Annotate
    cat $JSONFILEEC2 | zannotate -routing -routing-mrt-file=$MRTFILEEC2 -input-file-type=json > $EFSEC2PORTSCANPATH/annotated

    # Step  2 - Extract IP
    cat $EFSEC2PORTSCANPATH/annotated | jq ".ip" | tr -d '"' > $EFSEC2PORTSCANPATH/extract_ip

    # Step 3 - Extract ASN
    cat $EFSEC2PORTSCANPATH/annotated | jq -c ".zannotate.routing.asn" > $EFSEC2PORTSCANPATH/extract_asn

    # Step 4 - Calculate popularity
    pr -mts, $EFSEC2PORTSCANPATH/extract_ip $EFSEC2PORTSCANPATH/extract_asn | awk -F',' "{ a[\$2]++; } END { for (n in a) print n
   \",\" a[n] } " | sort -k2 -n -t',' -r > $EFSEC2PORTSCANPATH/as_popularity
```

Evaluation / Micro Benchmarks

- Port Scan analysis - sshell code version

```
$>
 echo Run Port scan analysis - version stateless
 JOBS=$1

 # Step 1 - split input JSON file into chunks
 mkdir $EFSEC2PORTSCANPATH/ckdir
 chmod 777 $EFSEC2PORTSCANPATH/ckdir
 cd $EFSEC2PORTSCANPATH/ckdir
 cat $JSONFILEEC2 | parallel -j200 --pipe --block 40M "cat >
${EFSEC2PORTSCANPATH}/ckdir/ckjson_{#}"
 split --verbose -n $CHUNKS $JSONFILEEC2 ckjson
 cd -
 durationportscansplitjson=$(expr $clock2 - $clock1)
 echo "Port scan 1st part - split: $durationportscansplitjson s"
 # Step 2 - annotate each chunk with sshell
 mkdir ${EFSEC2PORTSCANPATH}/annotateddir
 chmod 777 ${EFSEC2PORTSCANPATH}/annotateddir
 ls ${EFSEC2PORTSCANPATH}/ckdir > cklist.out
```

```
$>
 echo size of input elements:
 cat cklist.out | wc -l
 cat cklist.out | parallel -j$JOBS -I,, --env sshell "sshell \" cat
${EFSLAMBDAPORTSCANPATH}/ckdir/,, | zannotate -routing
-routing-mrt-file=$MRTFILELAMBDA -input-file-type=json >
$EFSLAMBDAPORTSCANPATH/annotateddir/annotated_\${PARALLEL_SEQ} \""

 # Step 3 - parse IP
 mkdir $EFSEC2PORTSCANPATH/ipdir
 chmod 777 $EFSEC2PORTSCANPATH/ipdir
 ls $EFSEC2PORTSCANPATH/annotateddir | parallel -j$JOBS -I,, --env sshell
"sshell \" cat ${EFSLAMBDAPORTSCANPATH}/annotateddir/,, | jq \"""".ip\"""" >
$EFSLAMBDAPORTSCANPATH/ipdir/ip_\${PARALLEL_SEQ} \""
```

Evaluation / Micro Benchmarks

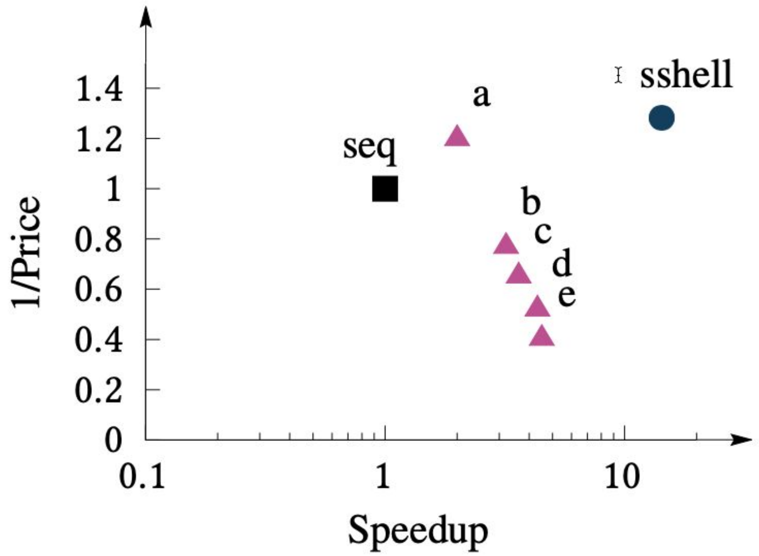
- Port Scan analysis - sshell code version

```
$>
 # Step 4 - parse ASN
 mkdir $EFSEC2PORTSCANPATH/asndir
 chmod 777 $EFSEC2PORTSCANPATH/asndir
 ls $EFSEC2PORTSCANPATH/annotateddir | parallel -j$JOBS -I,, --env
sshell "sshell \" cat ${EFSLAMBDAPORTSCANPATH}/annotateddir/,, | jq
-c \"'"'.zannotate.routing.asn\"'"' >
$EFSLAMBDAPORTSCANPATH/asndir/asn_\${PARALLEL_SEQ} \""
 #echo $(processaspopularity)

 # Step 5 - Output popularity
 cat $EFSEC2PORTSCANPATH/ipdir/ip_* >
$EFSEC2PORTSCANPATH/ipdir/ip_aggr
 cat $EFSEC2PORTSCANPATH/asndir/asn_* >
$EFSEC2PORTSCANPATH/asndir/asn_aggr
 pr -mts, $EFSEC2PORTSCANPATH/ipdir/ip_aggr
$EFSEC2PORTSCANPATH/asndir/asn_aggr | awk -F',' "{ a[\$2]++; } END
{ for (n in a) print n \",\" a[n] } " | sort -k2 -n -t',' -r >
$EFSEC2PORTSCANPATH/as_popularity
```

Evaluation / Micro Benchmarks

- Port Scan analysis
  - **seq**: sequential implementation
  - **sshell** : better alternative
  - **a-e** : native execution

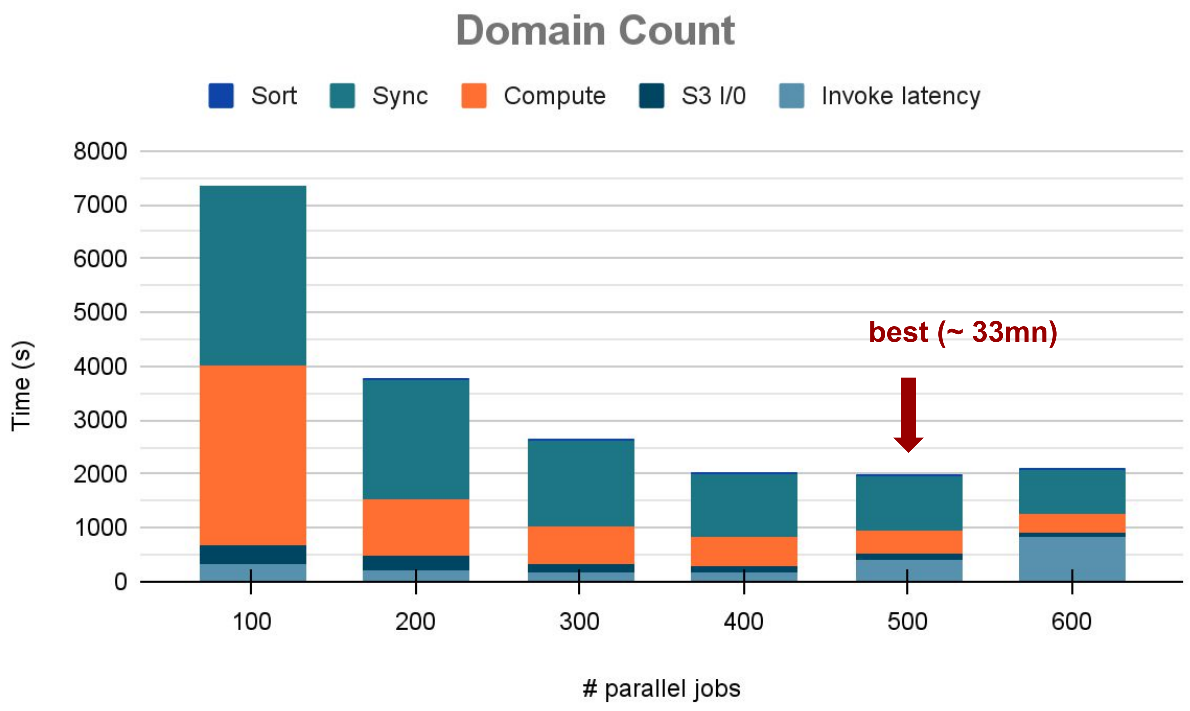
ID	c5 EC2 instance (# VCPUS)
a	16
b	36
c	48
d	72
e	98



Evaluation / Large scale application

- **Domain count: Ranking the popularity of web domains**
  - Download archives containing web pages from Commoncrawl (~ 20 TB compressed)
  - Uncompress archives
  - Extracts the outgoing links
  - Count the number of times the domain is mentioned in each page
  - Aggregates results
  - Sort results to construct output

Evaluation / Large scale application

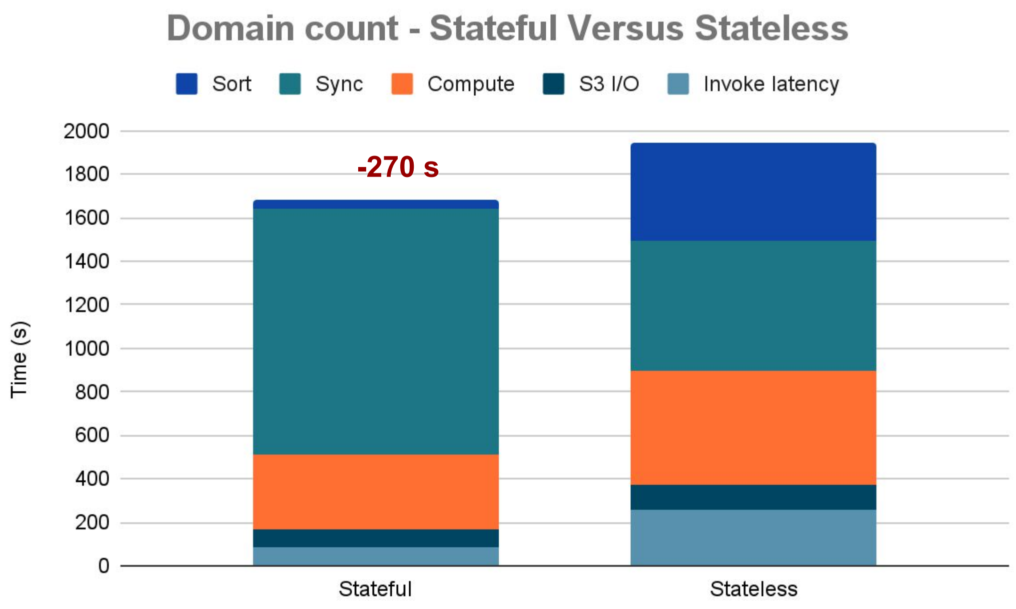


best (~ 33mn)

- **compute component decreases as # parallel jobs increases**

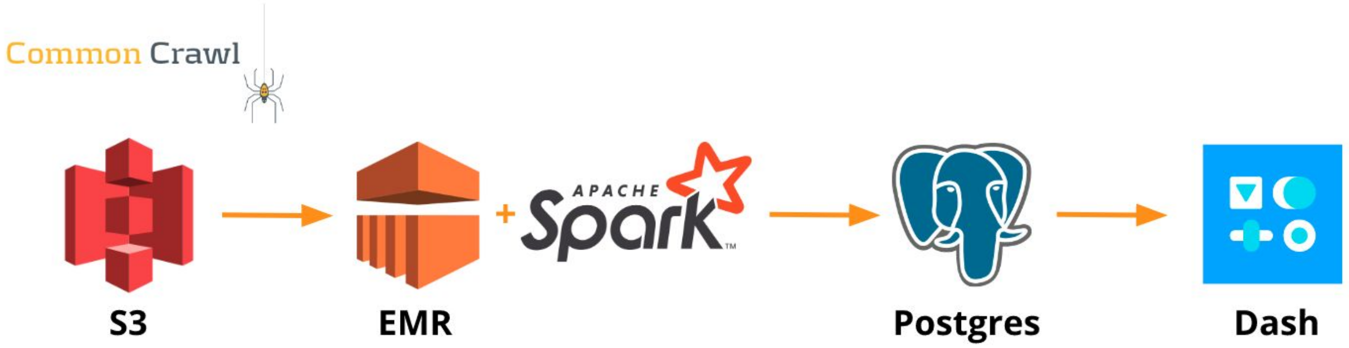
Evaluation / Large scale application

- 2 versions for sync component
  - Stateless: use of **AWK** language
  - Stateful: Use of **treemap** DSO object



Evaluation / Large scale application

- **LinkRun - Pipeline**



Evaluation / Large scale application

- **Sshell version versus LinkRun**

	SLOC	**Pricing**	**Time**	**Dataset size**
Linkrun	716	$200-260	26-48h	17.62 TB
Sshell	51	$19	28mn	20.17 TB

Future Work

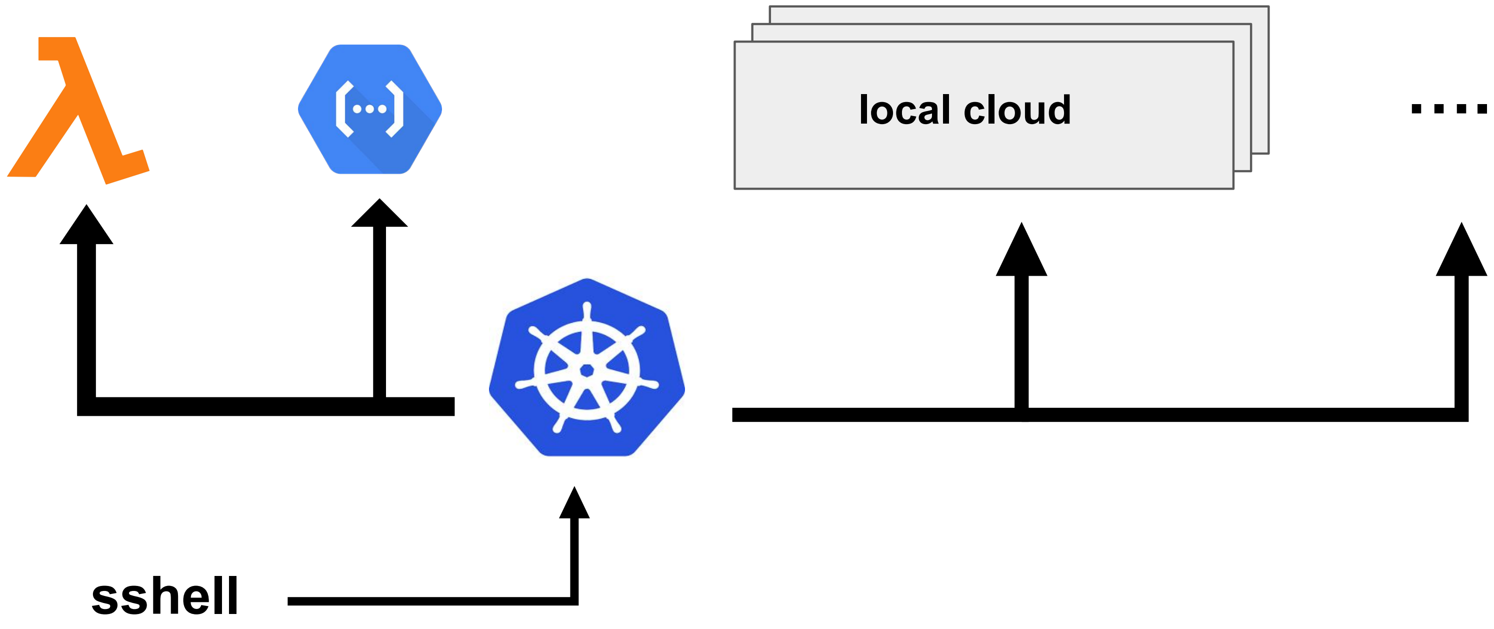
- PaSH : Data-parallel Shell Processing
  - Parallelizes POSIX shell scripts
  - Rewrite them using named pipes (**mkfifo**)
  - Given an input script, enable parallelization using named pipes

- Add back-end to pash:
  - Rewrite PaSH output pipes

  - *$> ./pash input.sh **-sshell** {1,2,3}*

Future Work

- Support multiple FaaS platforms (Google Cloud Platform)
- Connect **sshell** to Kubernetes
    - **Kubernetes** : container orchestrator



References

- "Posh: A Data-Aware Shell", Deepti Raghavan et al. Usenix ATC 2020, Boston, USA. 2020
- “PaSh: light-touch data-parallel shell processing”, Nikos Vasilakis et al. EuroSys '21: Proceedings of the Sixteenth European Conference on Computer SystemsApril 2021

The Serverless shell