A File System for Serverless Computing

Johann Schleier-Smith and Joseph M. Hellerstein

Presented at HPTS
November 5, 2019

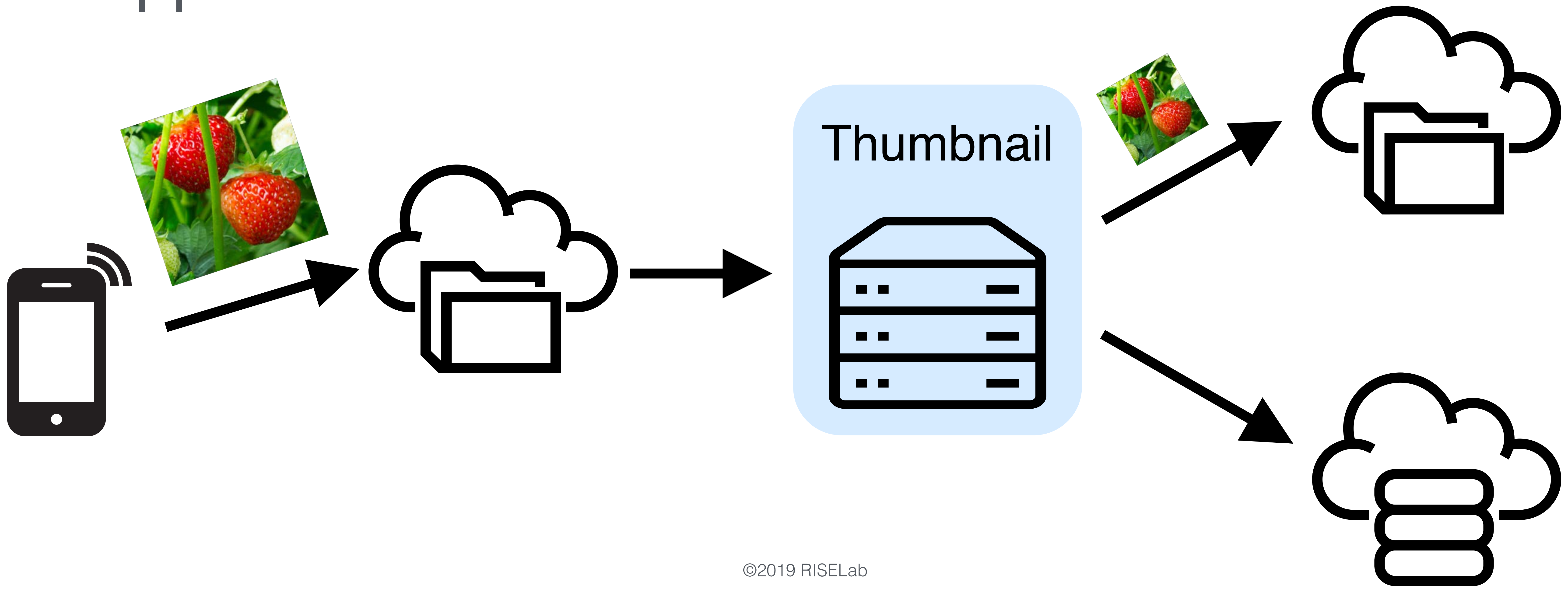
riselab
UC Berkeley

Outline

- Serverless computing background
- Limitations
- The Cloud Function File System (CFFS)
- Evaluation and learnings

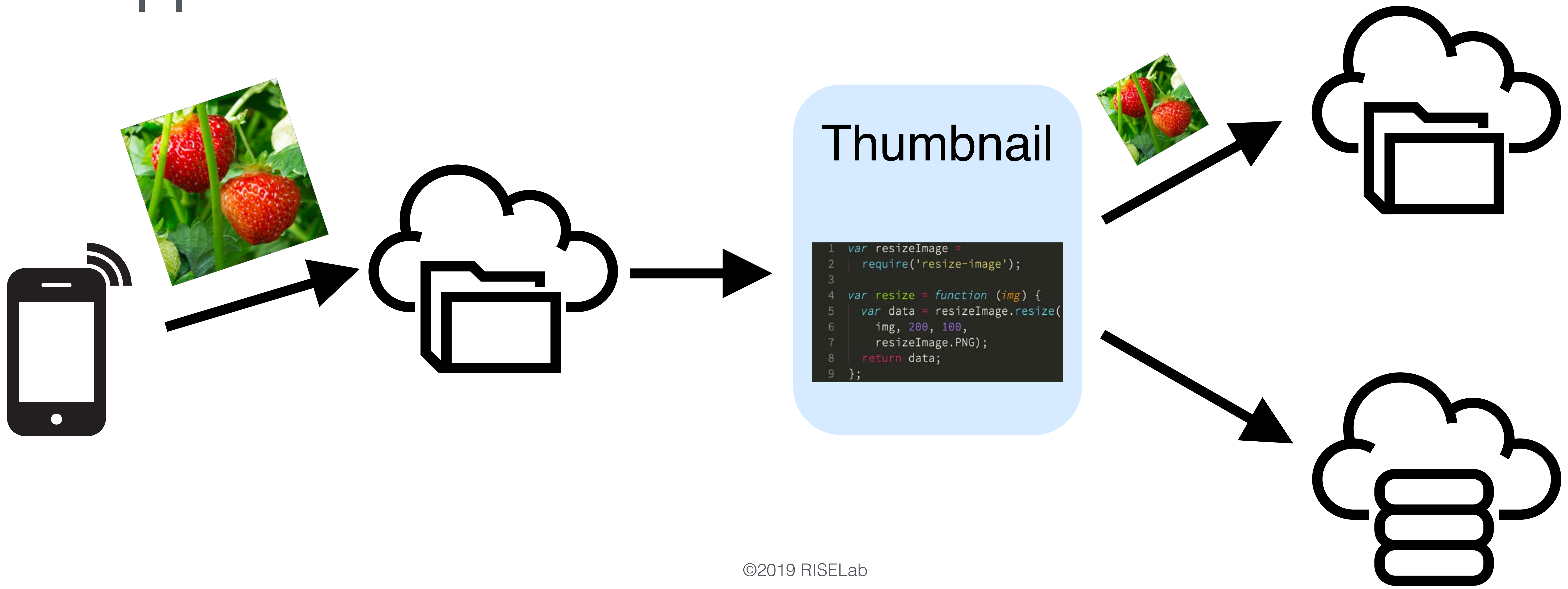
Serverless computing in 2014

- AWS announced Lambda Function as a Service (FaaS) in 2014, others clouds followed quickly
- Write code, upload it to the cloud, run at any scale
- Successful for web APIs and event processing, limited in stateful applications



Serverless computing in 2014

- AWS announced Lambda Function as a Service (FaaS) in 2014, others clouds followed quickly
- Write code, upload it to the cloud, run at any scale
- Successful for web APIs and event processing, limited in stateful applications

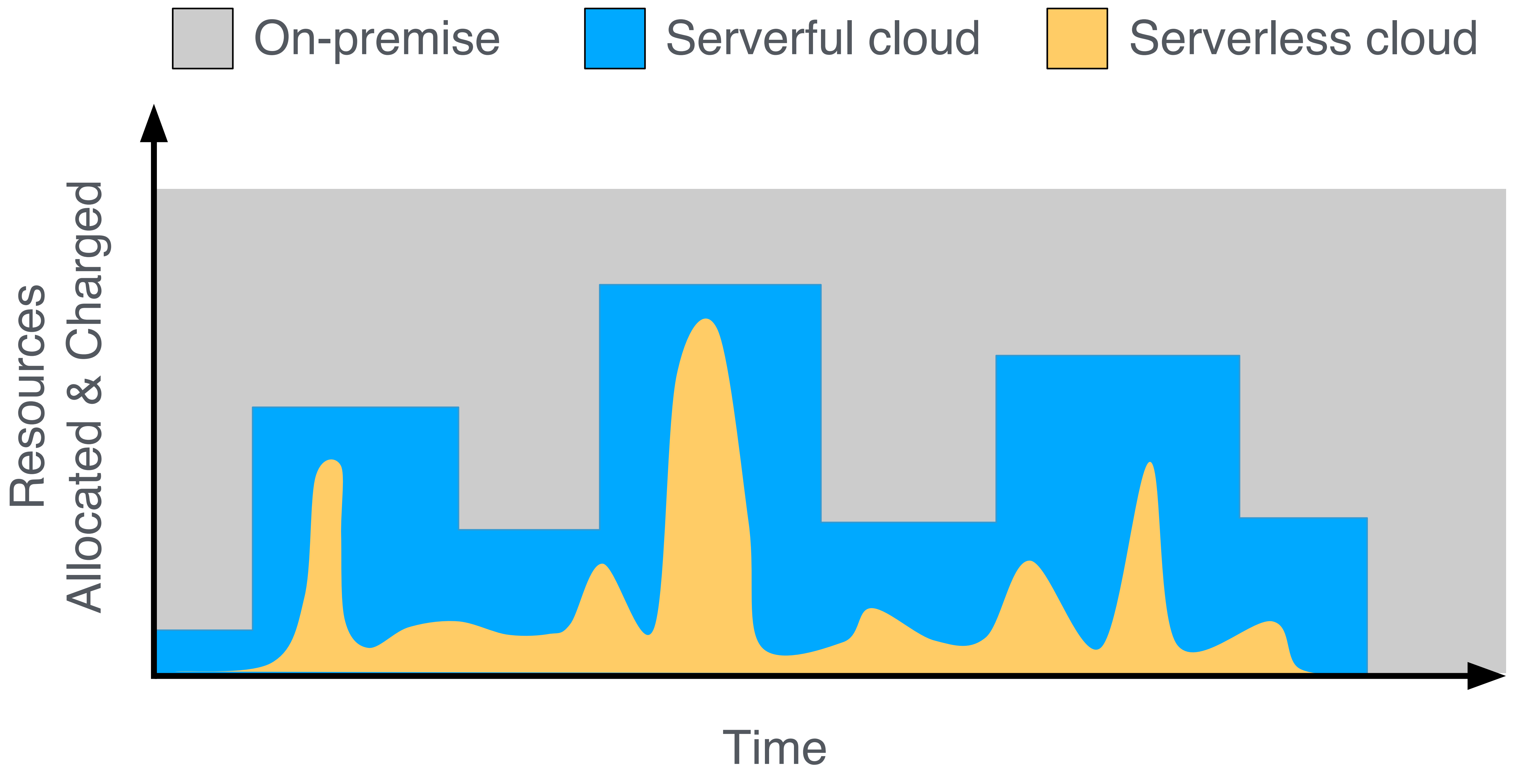


Defining characteristics of serverless abstractions

- **Hiding the servers** and the **complexity of programming** them
- **Consumption-based** costs and **no charge for idle** resources
- **Excellent autoscaling** so resources match demand closely

Understanding serverless computing's impact

Resources



People

Job	Serverful Cloud	Serverless Cloud
Infrastructure Administration	Outsourced job	Outsourced job
System Administration	Simplified job	Outsourced job
Software Development	Little change	Simplified job

Serverless as next phase of cloud computing

- Serverless abstractions offer
  - Simplified programming
  - Outsourced operations
  - Improved resource utilization

Serverless is much more than FaaS

Object Storage
AWS S3
Azure Blobs
Google Cloud Storage

Function as a Service
AWS Lambda
Google Cloud Functions
Google Cloud Run
Azure Functions

Big Data Processing
Google Cloud Dataflow
AWS Glue
AWS Athena
AWS Redshift

Queue Service
AWS SQS
Google Cloud Pub/Sub

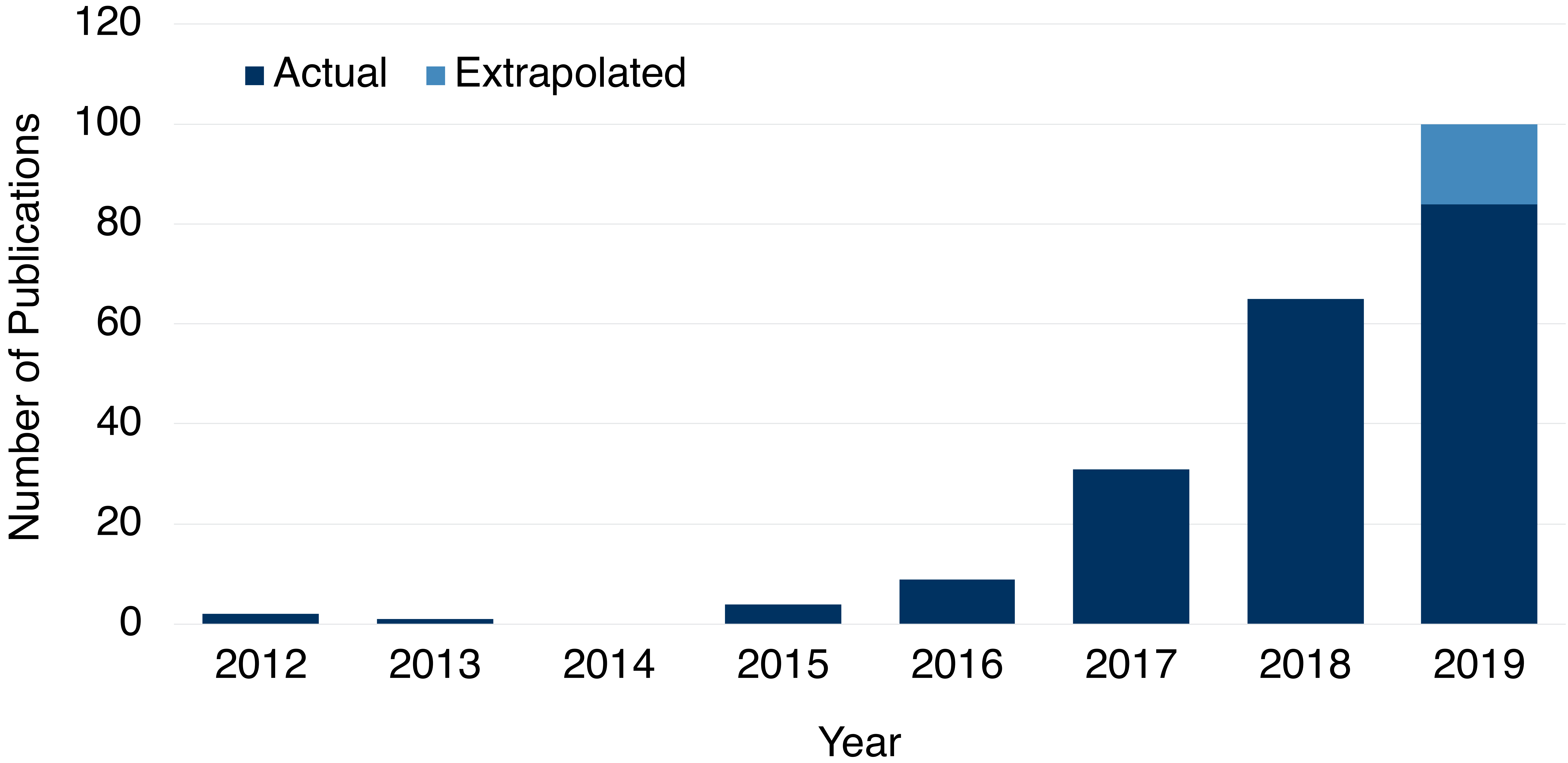
Key-Value Store
AWS DynamoDB
Azure CosmosDB
Google Cloud Datastore

Mobile back end
Google Firebase
AWS AppSync

AWS Serverless Aurora

Google App Engine

Fertile ground for research



Source: dblp computer science bibliography.

Outline

- Serverless computing background
- Limitations
- The Cloud Function File System (CFFS)
- Evaluation and learnings

Limitations of FaaS

Limited runtime

No inbound network connections

Ephemeral state

No specialized hardware, e.g., GPU

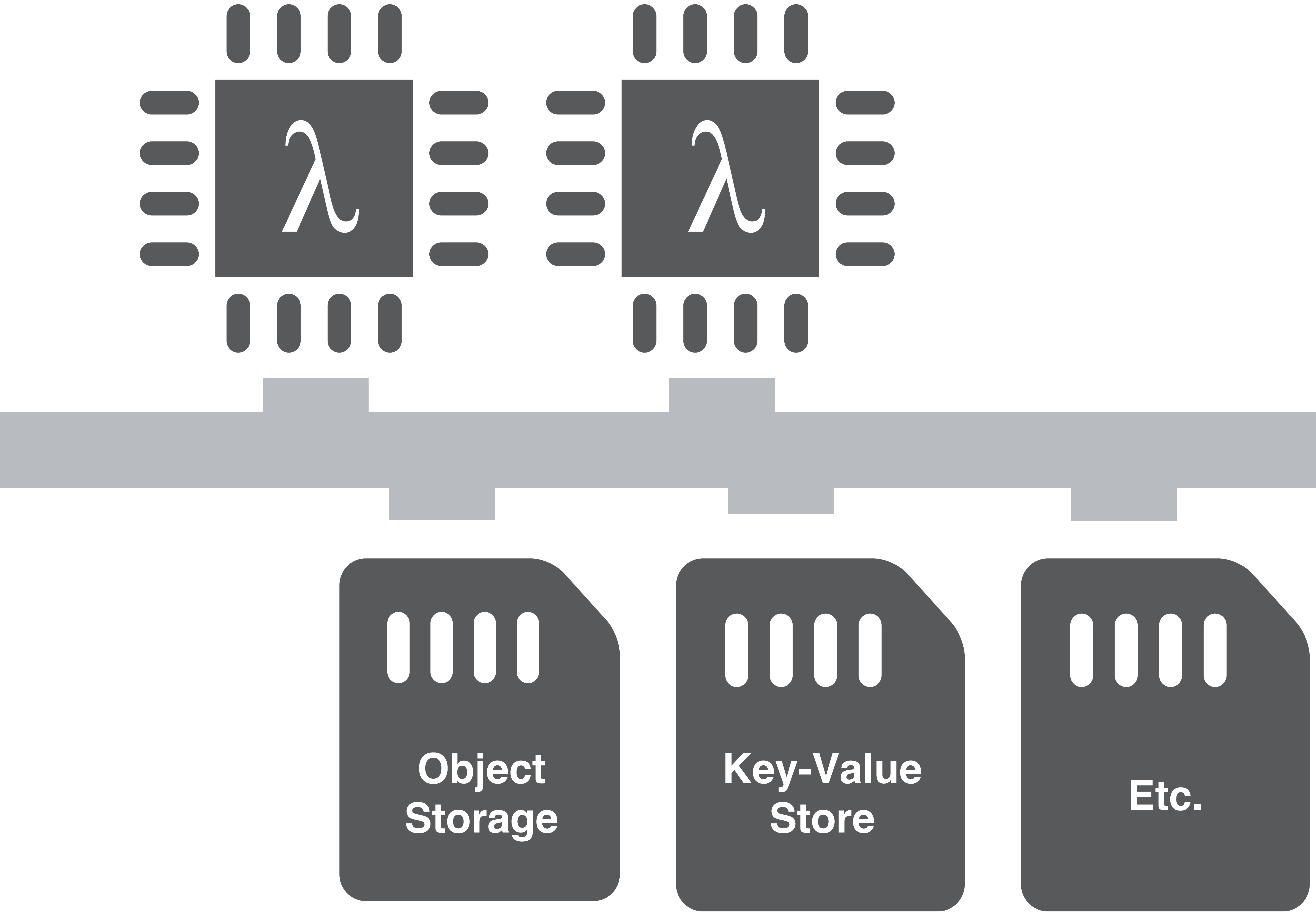
Plenty of complementary stateful serverless services

Object Storage
- AWS S3
- Azure Blobs
- Google Cloud Storage

Key-Value Store
- AWS DynamoDB
- Azure CosmosDB
- Google Cloud Datastore
- Anna KVS (Berkeley)

Others
- AWS Aurora Serverless
- Google Firebase

Combine with FaaS to build applications



Compute:
isolated & ephemeral state

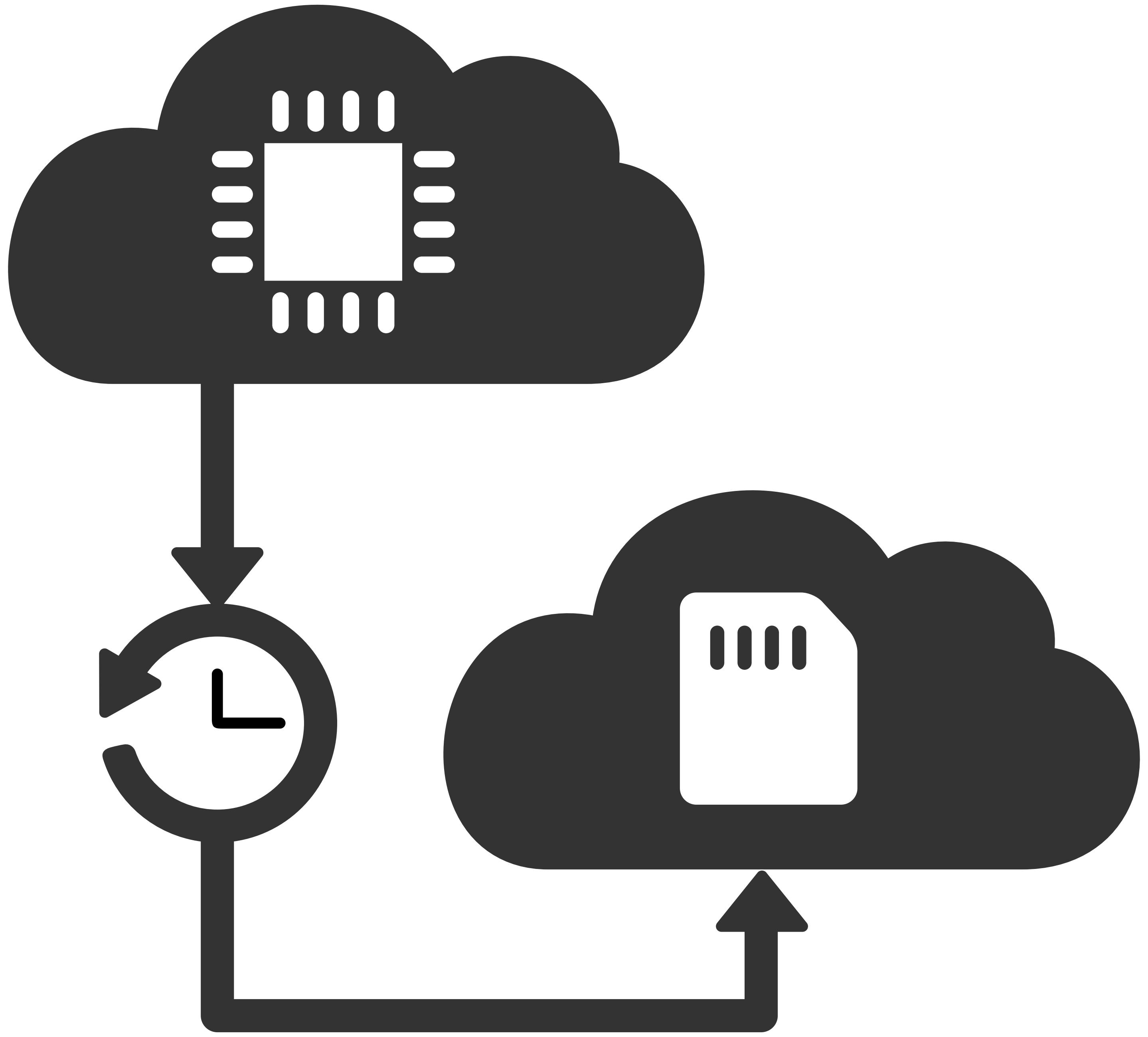
Storage:
shared & durable state

Allows independent scaling

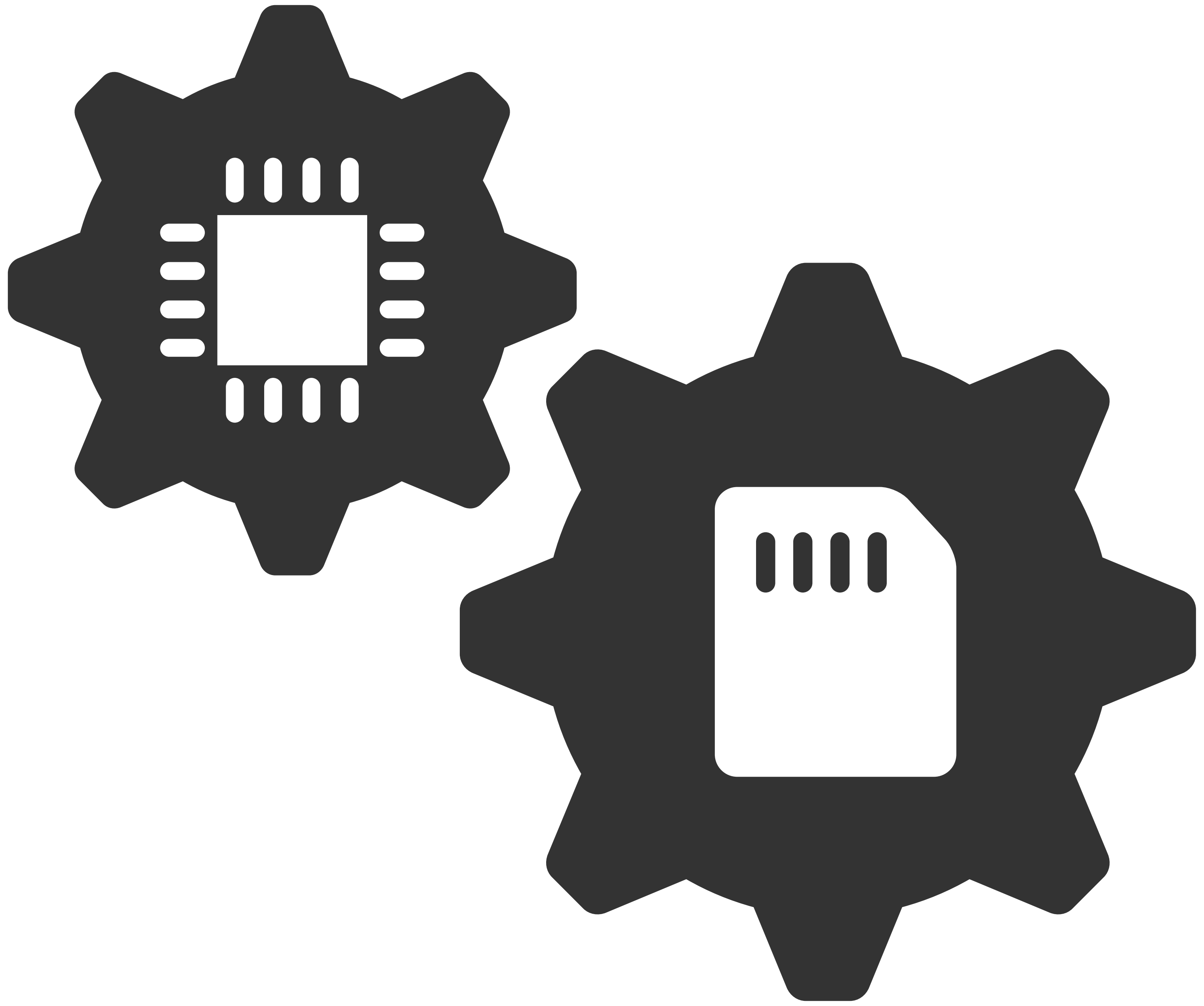
How happy are we?

How happy are we?

Two main problems



Latency

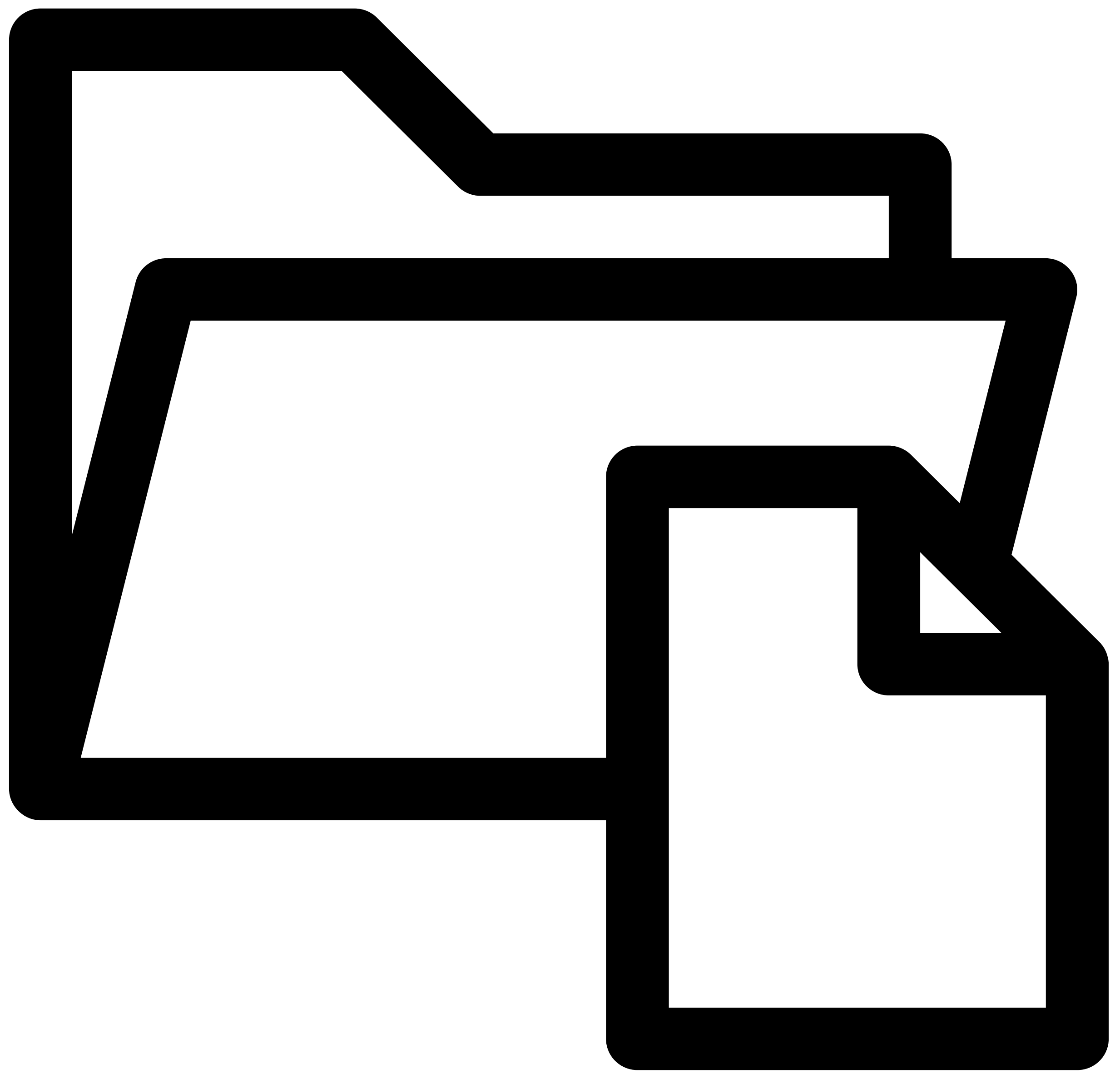


API

Can I please have something like local disk, but for the cloud? 🙏

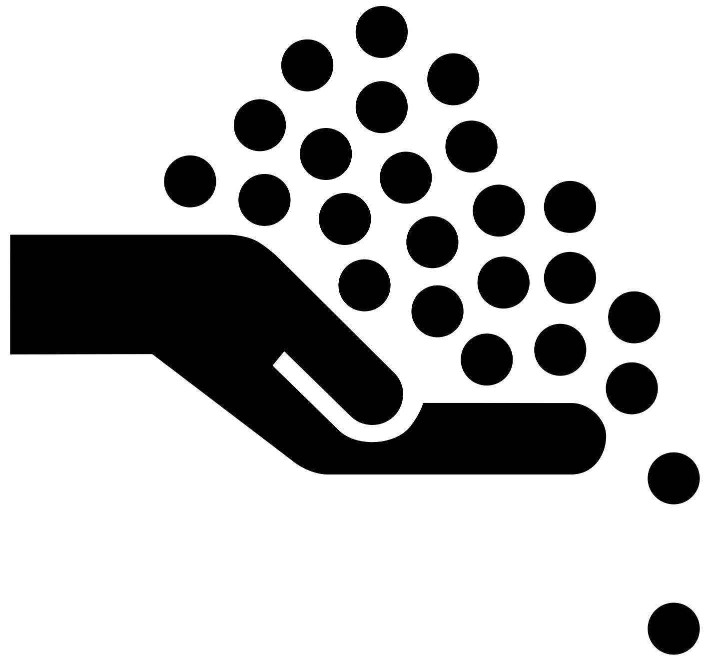
File systems let us run *so much* software

- Data analysis with Pandas
- Machine learning with TensorFlow
- Software builds with Make
- Search indexes with Sphinx
- Image rendering with Radiance
- Databases with SQLite
- Web serving with Nginx
- Email with Postfix



Objections

- It won’t scale
  - Need a simpler data model (key-value store, immutable objects)
  - Need a weaker consistency model (eventual consistency)
  - Performance and reliability will suffer otherwise



- You don’t need it
  - You should be rewriting your software for the cloud anyhow
  - Why not just use use a key-value store?

How does this make me feel?

Outline

- Serverless computing background
- Limitations
- The Cloud Function File System (CFFS)
- Evaluation and learnings

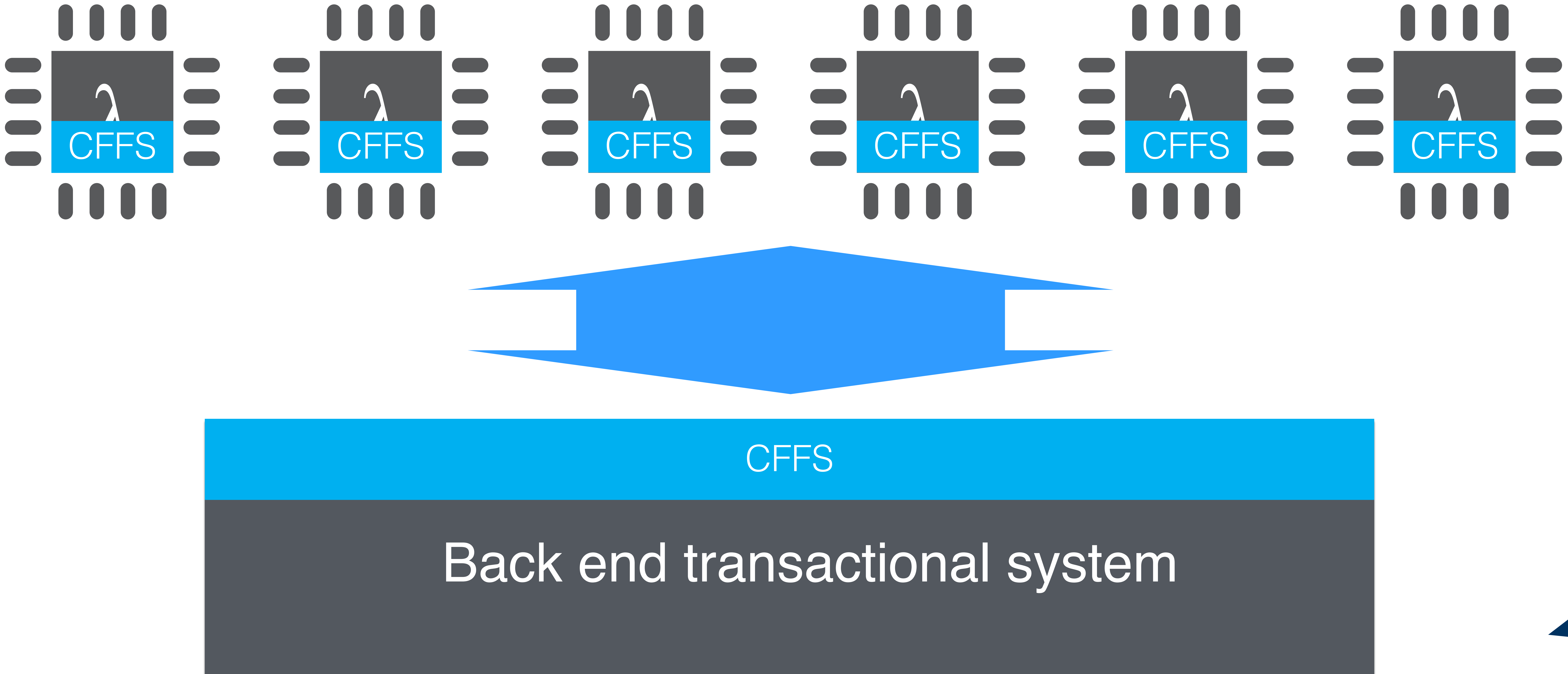
Introducing the Cloud Function File System (CFFS)

- POSIX semantics, including strong consistency
- Local caches for local disk performance
- Works under autoscaling, extreme elasticity, and FaaS limitations
- Implemented as a transaction system

What is special about the FaaS environment?

- Function invocations have well defined beginning and end
- At-least-once execution—expects idempotent code
- Constrained execution model
  - Clients frozen in between invocations
  - No inbound network connections
  - No root access

CFFS Architecture



FaaS instance caching

- "Function as a Service" suggests statelessness, but most implementations reuse instances and preserve their state
- Setup of sandboxed environment takes time
  - Loads selected runtime (e.g., JavaScript, Python, C#, etc.)
  - Configures network endpoint, IAM privileges
  - Loads user code
  - Runs user initialization
- Caching is key to amortizing instance setup

Cloud Function File System (CFFS)

Core API implemented in CFFS

`open`	New descriptor (handle) for file or directory
`close`	Close descriptor
`write / pwrite`	Write / positioned write
`read / pread`	Read / positioned read
`stat`	Get size, ownership, access, permissions, last modified
`seek`	Set descriptor position
`dup / dup2`	Copy descriptor
`truncate`	Set file size
`flock`	Byte range lock and unlock

`mkdir`	Create directory
`rename`	Rename file / directory
`unlink`	Delete file / directory
`chmod`	Set access permissions
`chown`	Set ownership
`utimes`	Update modified / accessed timestamps
`clock_gettime`	Get current time
`chdir`	Set working directory
`getcwd`	Get working directory

`begin`	Start transaction
`commit / abort`	End transaction

POSIX guarantees - language from the spec

“

- Writes **can be serialized** with respect to other reads and writes.
- If a *read()* of file data **can be proven (by any means)** to occur after a *write*() of the data, it must reflect that *write*()…
- A similar requirement applies to multiple write operations to the same file position…
- This requirement is **particularly significant for networked file systems, where some caching schemes violate these semantics**.

”

https://pubs.opengroup.org/onlinepubs/9699919799/

POSIX guarantees in database terms

- Atomic operations
  - Each operation (at the API level) is observed entirely or not at all
  - Some violations in practice
- Consistency model
  - Spec references time, technically requires *strict consistency* (shared global clock)
  - Actually implemented as *sequential consistency* (global order exists, consistent with order at each processor)
  - We use *serializability* to provide isolation and atomicity at function granularity

Open question: what guarantees do applications actually rely on?

Implementation highlights

- Choice of transaction mechanism not fundamental
  - Implemented timestamp-order serilizable
  - Optimistic protocols can be a good fit—FaaS side effects must be idempotent
  - State-of-the-art protocols promise lower abort rates, more effective local caches (e.g., Yu, *et al.*, VLDB 2018)
- Cache updates through on-demand filtererd log shipping
  - Check for updates when function starts execution
  - Eviction messages help back-end track client cache content

CFFS in context: Transactional file systems

- QuickSilver distributed system – IBM, 1991
  - Very close in spirit
  - No FaaS
  - No caching
- Inversion File System – Berkeley, 1993
  - Built on top of PostgreSQL
  - Access through custom library
- Transactional NTFS (TxF) – Microsoft, 2006
  - Shipping in Windows
  - Deprecated on account of complexity

There are many non-transactional shared & distributed file systems

CFFS in context: Shared file systems

- Must choose between consistency and latency
  - Eventual consistency
  - Delegation/lock-based caching
  - No caching

HPC

- Lustre
- GPFS (IBM)
- GlusterFS (RedHat / IBM)
- GFS (Google File System)
- MooseFS
- LizardFS
- BeeGFS

Big data

- HDFS
- GFS (Google)
- Ceph (IBM)
- MapR-FS
- Alluxio

Client-server

- NFS
- SMB

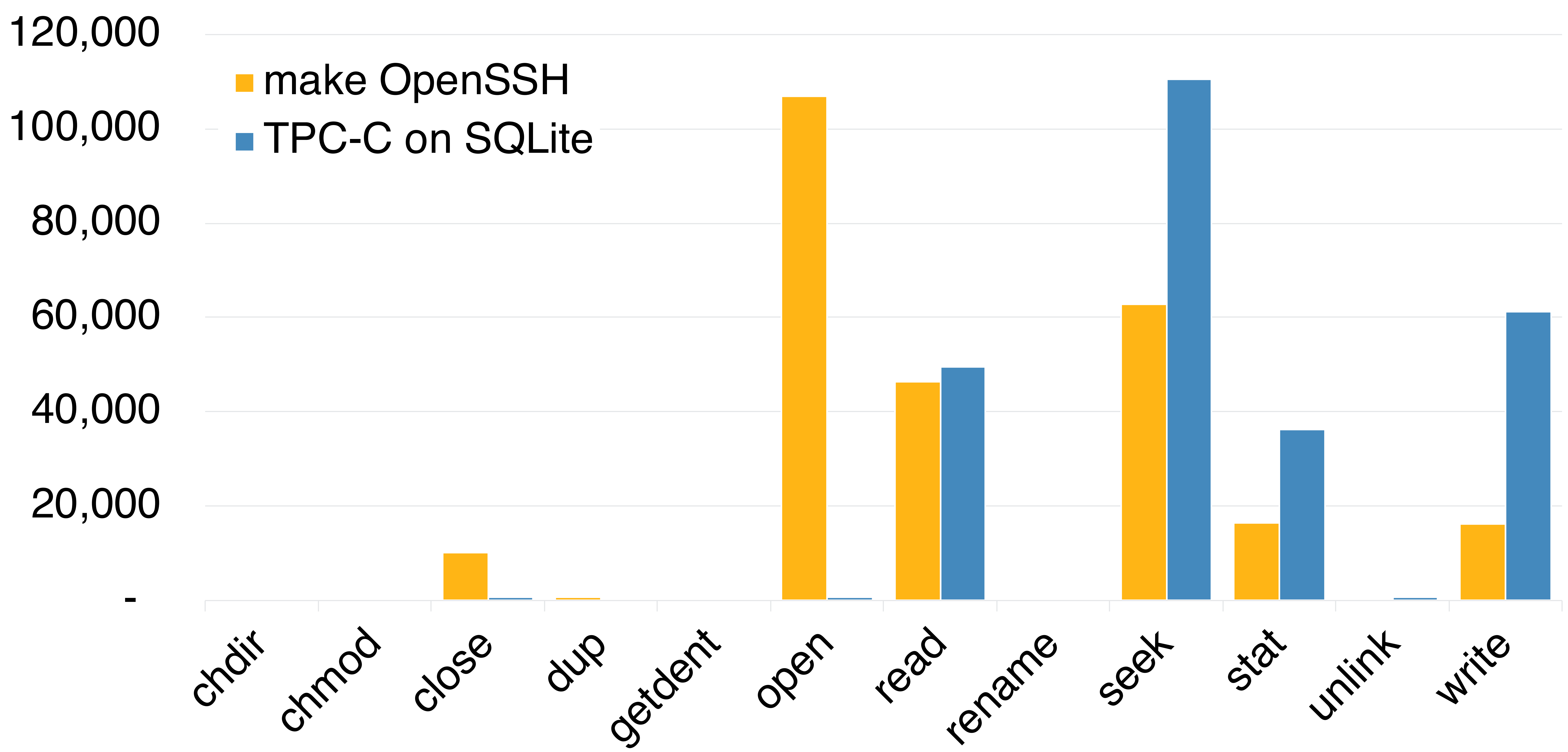
Mainframe

- zFS

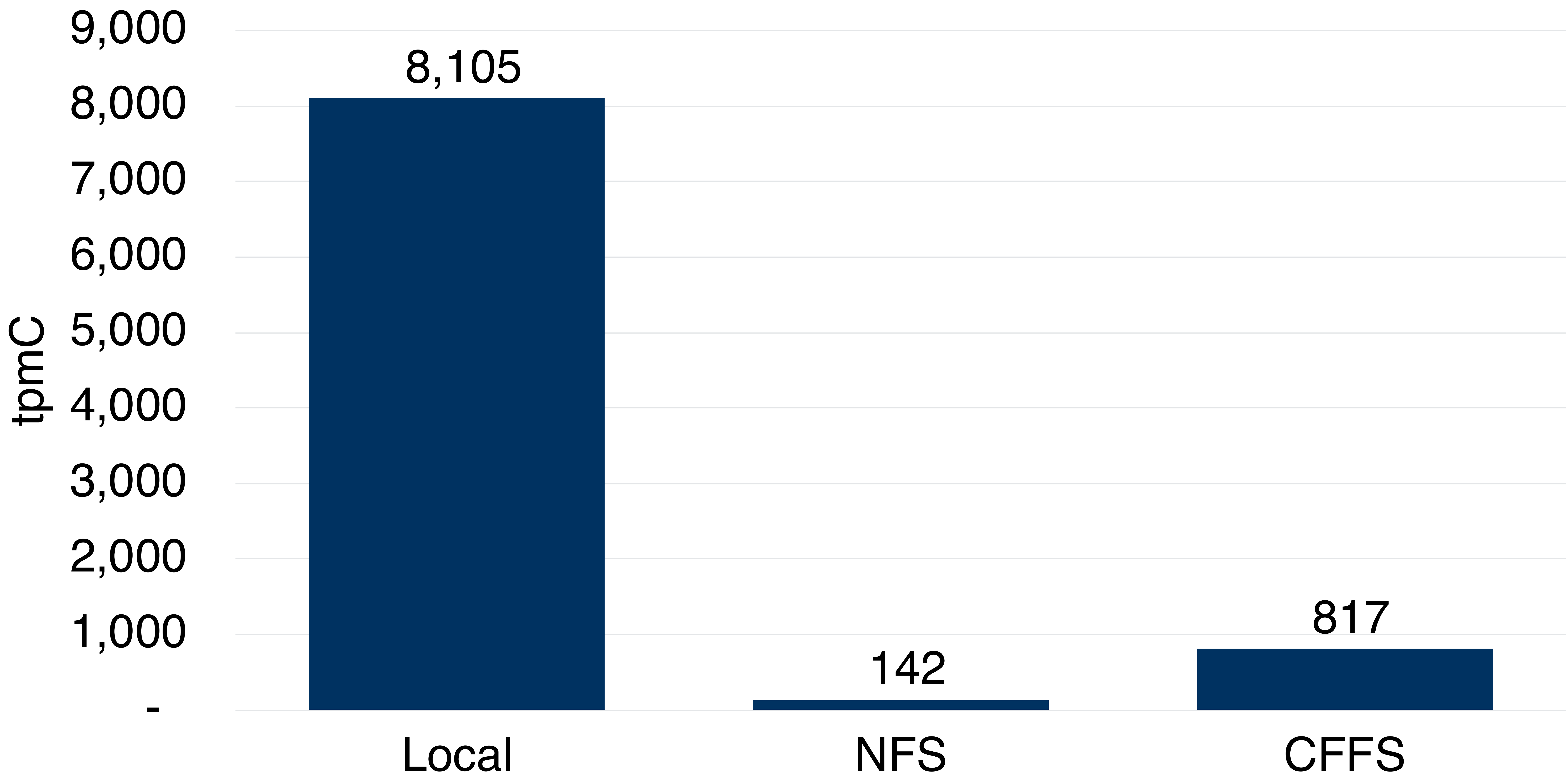
Outline

- Serverless computing background
- Limitations
- The Cloud Function File System (CFFS)
- Evaluation and learnings

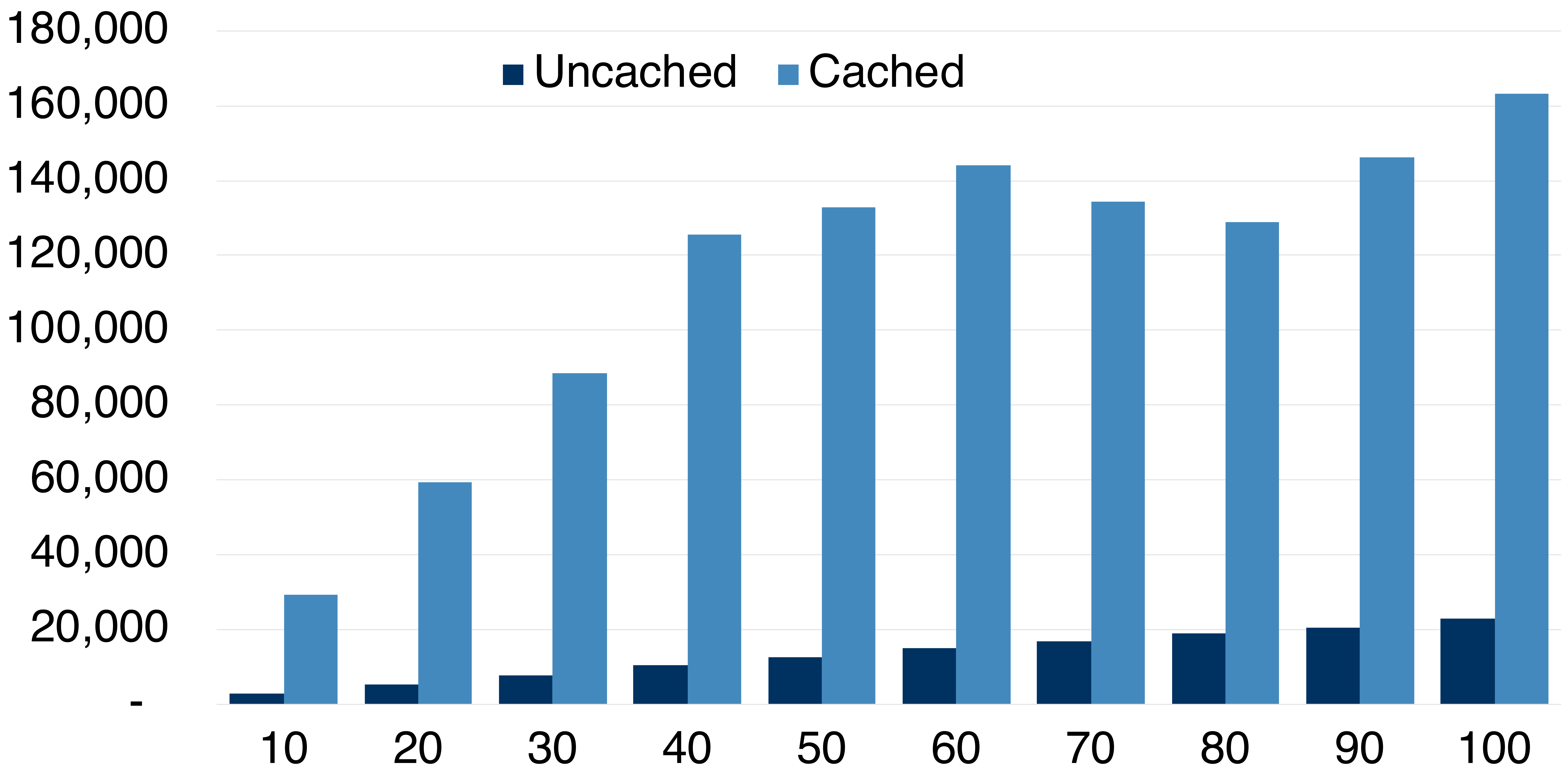
Sample workload call frequencies



Caching benefits (TPC-C / SQLite)



Scaling in AWS Lambda

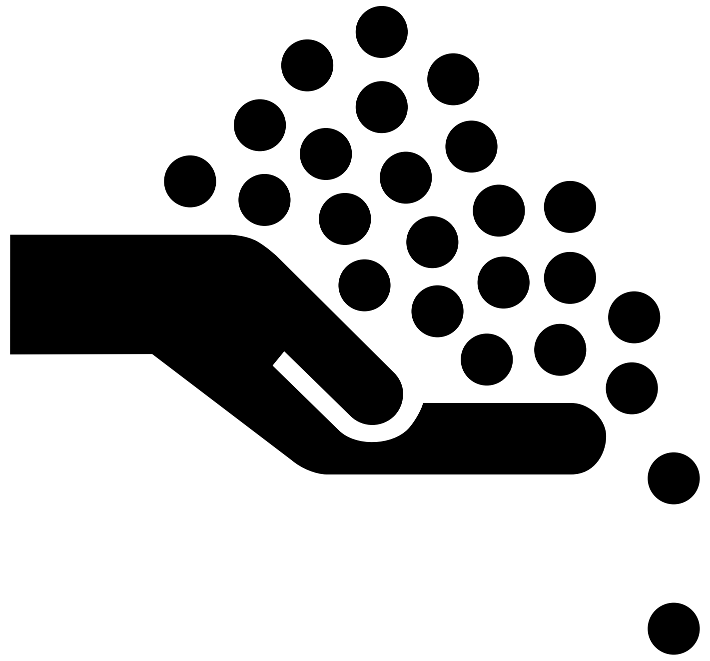


4k random reads

Returning to objections

- It won’t scale—contention risks
  - File length: must be checked on every read
  - Stat command: mainly used use it to check the file length or permissions, but also returns modification time and access time

Challenges here, optimistic they will be overcome



- You don’t need it

I think it will be pretty useful

How happy are we?

How happy are we?

CFFS Summary

- Transactions are a natural fit for FaaS
  - BEGIN and END from function context
  - At-least-once execution goes well with optimistic transactions
  - On-demand filtered log shipping allows asynchronous cache updates
- Overcomes limitations of FaaS & traditional shared file systems
  - Allows caching for lower latency, preserving consistency
  - Highly scalable, especially with snapshot reads
  - POSIX API enables vast range of tools and libraries