OpenLambda

Tyler Harter, **Edward Oakes**, Stephen Sturdevant, Leon Yang,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Motivation

Serverless

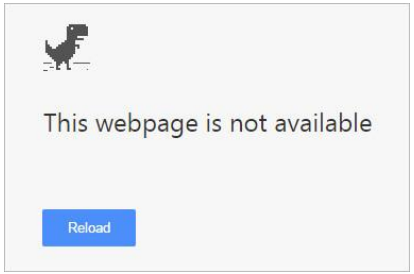
Research Questions

Interpreter Caching

Scaling

Naturally bursty web applications (e.g. Hamilton tickets)

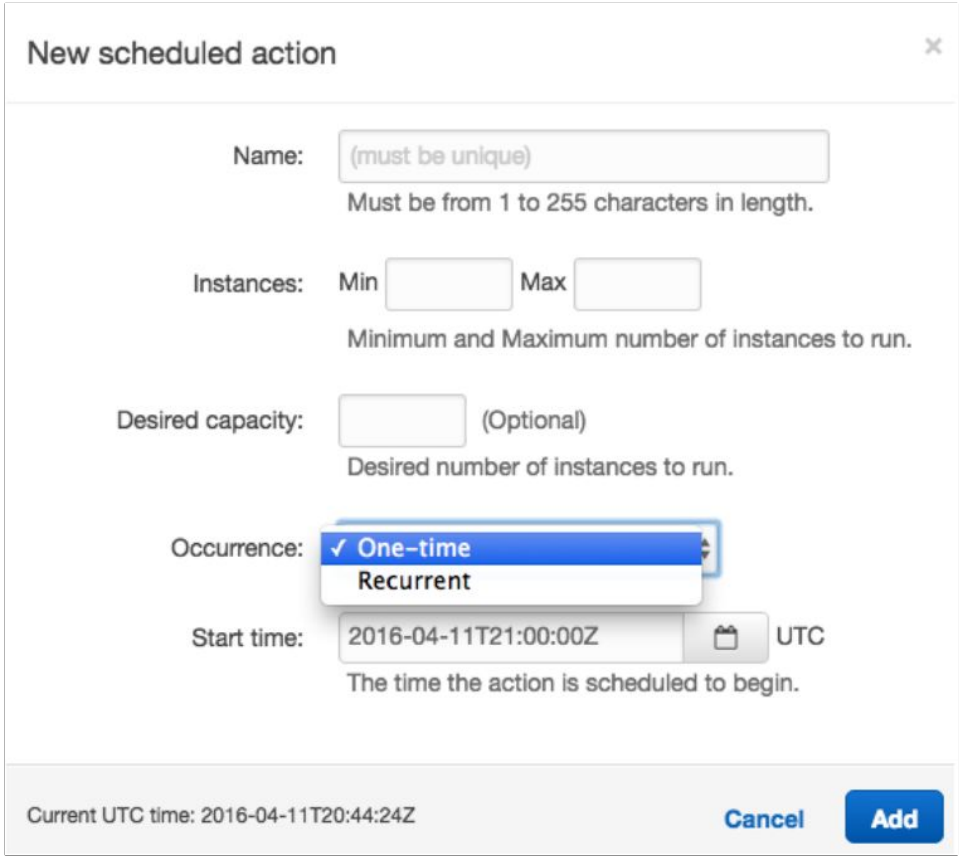
Hard to scale web servers



(ノ °□°)ノ ⁀┻┻

Scaling

- VMs are too slow to scale reactively
  - Have to predict a burst in order to withstand it
- Predictively scaling web servers is wasteful
  - Why pay for resources you don't use?



Configuration

- Developer has to install a web server, configure network settings, API endpoints, etc
  - Even for “Hello, World!”
- Developer cycles are often more valuable than CPU cycles...

and more....

```
13  # You may use the command line option '-S' to verify your virtual host
14  # configuration.
15
16  #
17  # Use name-based virtual hosting.
18  #
19  NameVirtualHost *:80
20
21  #
22  # VirtualHost example:
23  # Almost any Apache directive may go into a VirtualHost container.
24  # The first VirtualHost section is used for all requests that do not
25  # match a ServerName or ServerAlias in any <VirtualHost> block.
26  #
27  <VirtualHost *:80>
28      ServerAdmin webmaster@dummy-host.home
29      DocumentRoot /www/docs/dummy-host.home
30      ServerName dummy-host.home
31      ServerAlias www.dummy-host.home
32      ErrorLog logs/dummy-host.home-error_log
33      CustomLog logs/dummy-host.home-access_log common
34  </VirtualHost>
35
36  <VirtualHost *:80>
37      ServerAdmin webmaster@dummy-host2.home
38      DocumentRoot /www/docs/dummy-host2.home
39      ServerName dummy-host2.home
40      ErrorLog logs/dummy-host2.home-error_log
41      CustomLog logs/dummy-host2.home-access_log common
42  </VirtualHost>
```

Motivation

Serverless

Research Questions

Interpreter Caching

Microservices Architecture

Problem:

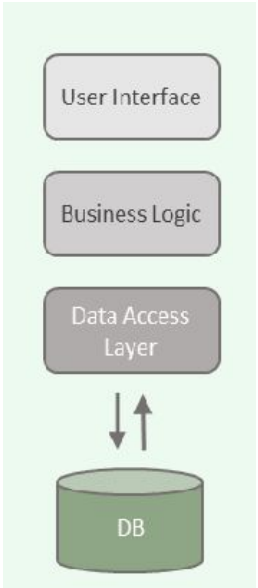
- Complex web applications lead to large, hard-to-debug codebases
- Effective deployment of web applications is difficult and important

Solution:

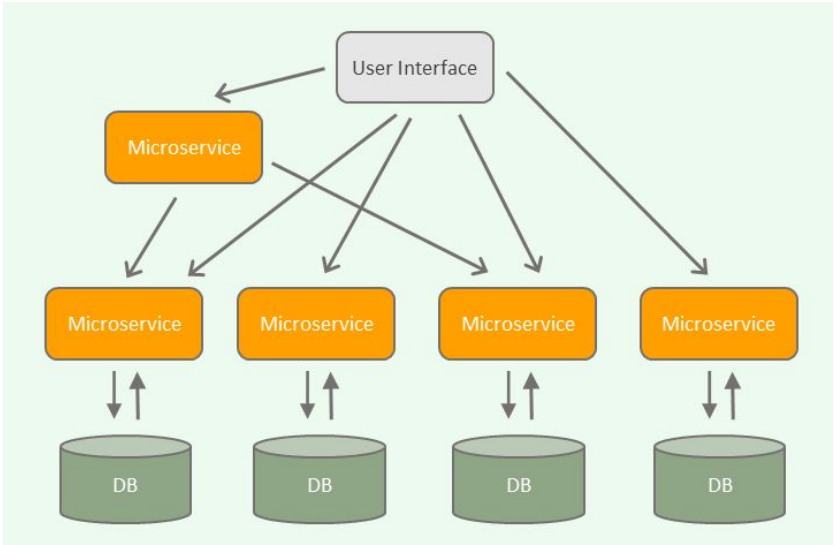
- Divide applications into modular pieces, call them **services**
- Communicate via a lightweight network mechanism
    - e.g. remote HTTP API

Microservices Architecture

MONOLITHIC ARCHITECTURE



MICROSERVICES ARCHITECTURE



Microservices Architecture

Philosophy:

- Services are small - perform a single, fine-grained function
- Each service is developed and deployed independently
  - Allows for better tuned scaling

Benefits:

- Easier to deploy
- Better tuned **scaling** - no longer have to scale the whole monolith
- Naturally **modular**
- Smaller codebases and simpler design = **faster** development

“Serverless”

- Natural extension of the “Microservices” paradigm
- Developers simply upload code and invoke it via a trigger (API, database, etc)
  - Blind of all provisioning, load balancing, etc
- Heavily influenced by the onset of Docker
  - Container startup << VM startup
- Not actually serverless...

Cloud Provider Perspective

- Code store
  - Metadata for decision-making?
- Persistent storage
  - Lambdas provide no state guarantees
- Servers to execute requests
  - Which VM to choose for any given request?
- Isolation mechanism
- Fine-grained billing
- Debugging support
- Triggers
  - R.I.P. RethinkDB?

Motivation

Serverless

Research Questions

Interpreter Caching

Debugging

- Web applications are extremely complex
  - Netflix: over 500 microservices
- Modularity can make debugging hard
  - What “path” led to the invocation?

How can we support developers in debugging their Lambda-based applications?

https://www.slideshare.net/brucewong3/the-case-for-chaos

Databases

- Lambdas are stateless, rely on entirely on distributed storage mechanisms
- Have access to the code before requests come in
  - How can we leverage this to optimize Lambda placement?
- The Lambda storage access pattern is new, how does it performance on distributed DBs, NFS, etc
  - What is the best distributed storage scheme for serverless? Something new?

Load Balancing

- Cloud provider has complete control over where requests are executed
- Want to optimize for:
  - Database accesses (analyze access patterns?)
  - Locality
    - Sending requests to the same machine means things are warmed up
    - Lambdas which call each other should be nearby
  - Packages
    - In memory > on disk > on network
- Trigger-based Lambdas lead to interesting scheduling problems
  - Cron-job Lambdas with leeway

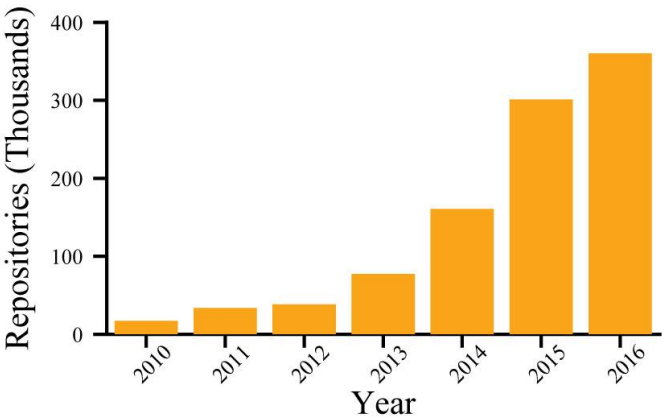
Package Support

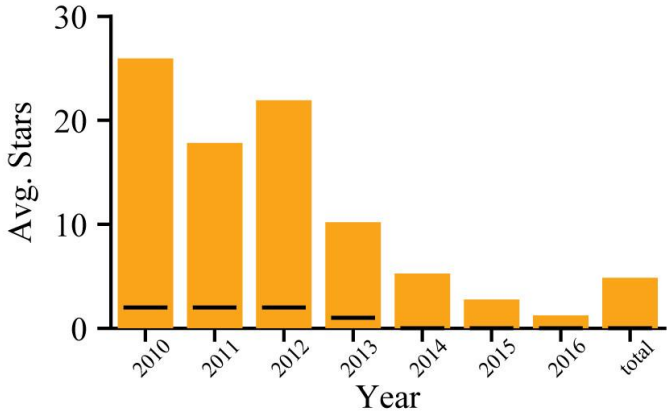
- Web development is highly reliant on external libraries/packages
  - AWS Lambda: only supports standard packages, makes doing anything interesting very difficult
- Can we allow Lambdas to seamlessly access all of the PyPI repository? What are the performance implications of this?
  - Over the network? On disk? In memory?

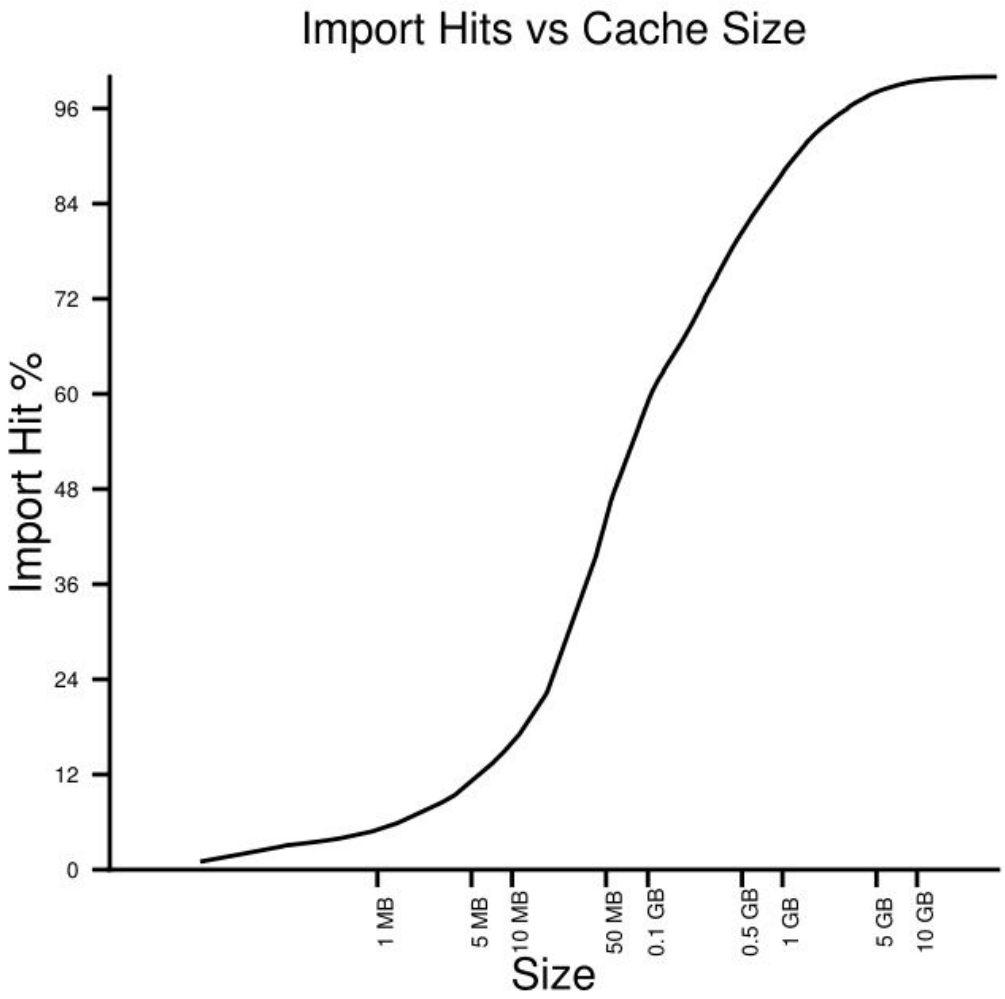
Motivation
Serverless
Research Questions
Interpreter Caching

How big is PyPI?

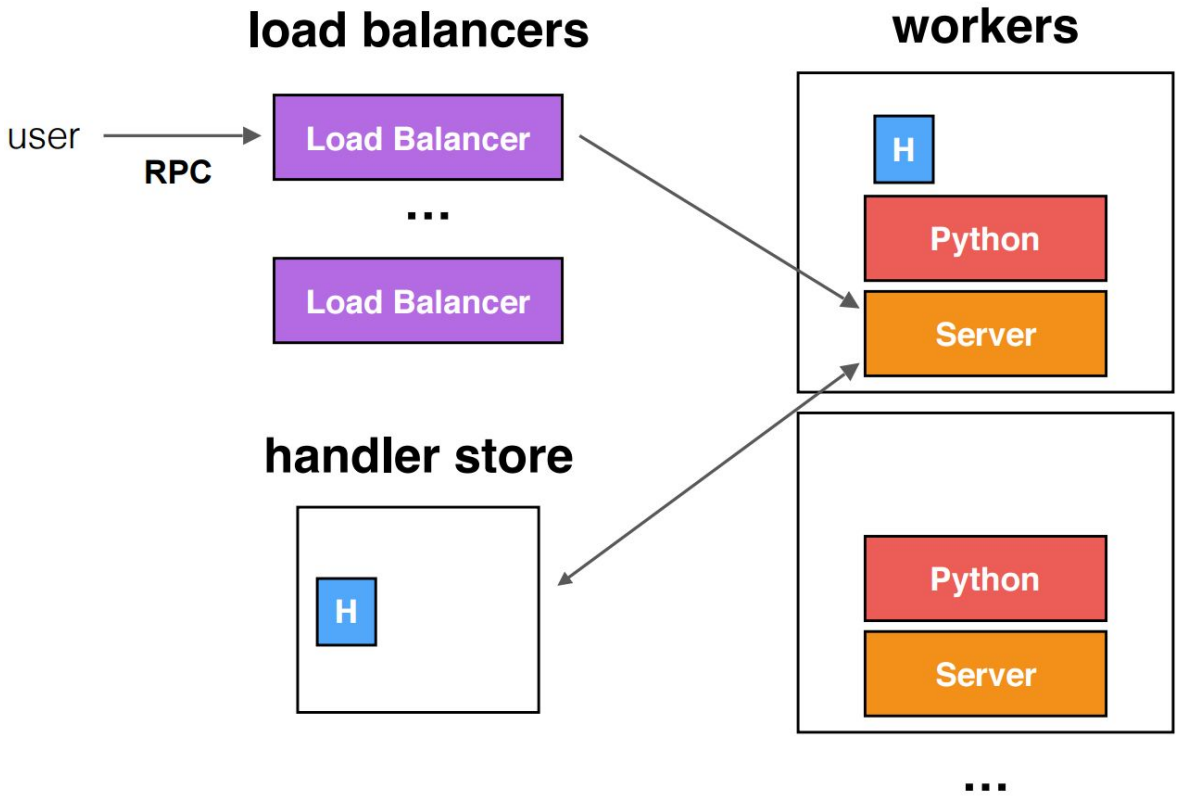
- Scraped ~1,000,000 GitHub repositories labeled as Python project
- Parsed out all import statements and matched them to PyPI packages
  - Not trivial, some assumptions required
- Hit ratio = number of import statements covered by packages in the cache



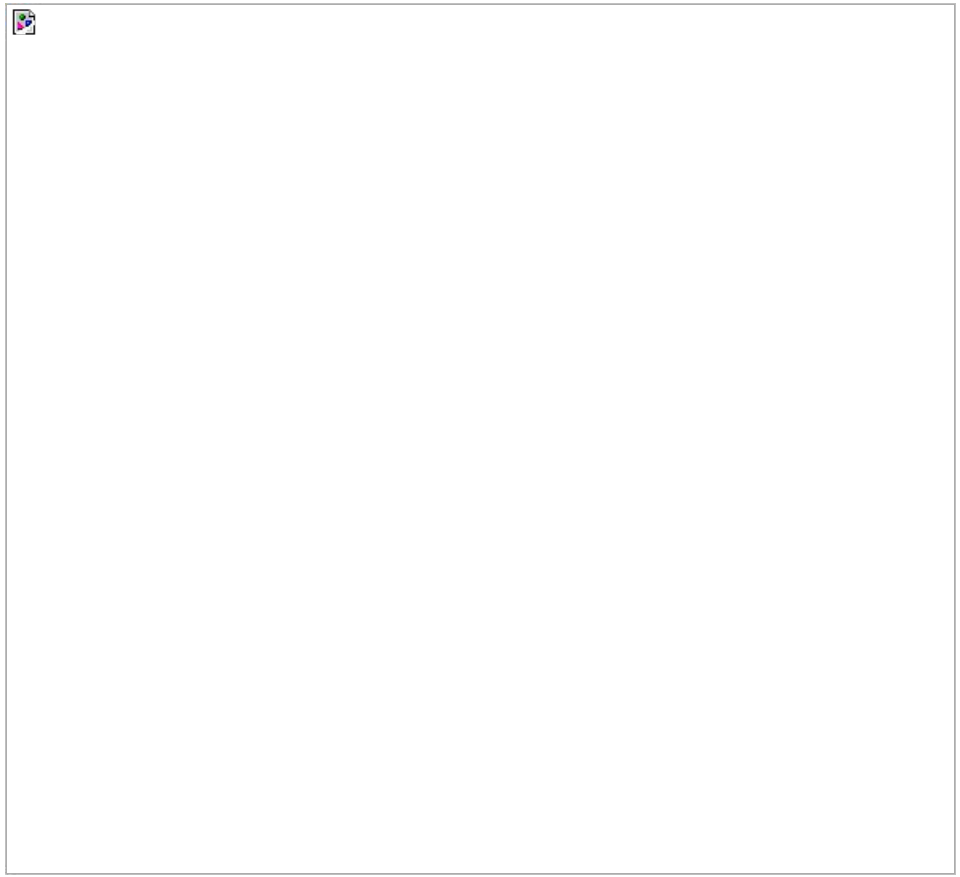



Import Hits vs Cache Size
Import Hit %
96
84
72
60
48
36
24
12
0
1 MB
5 MB
10 MB
50 MB
0.1 GB
0.5 GB
1 GB
5 GB
10 GB
Size

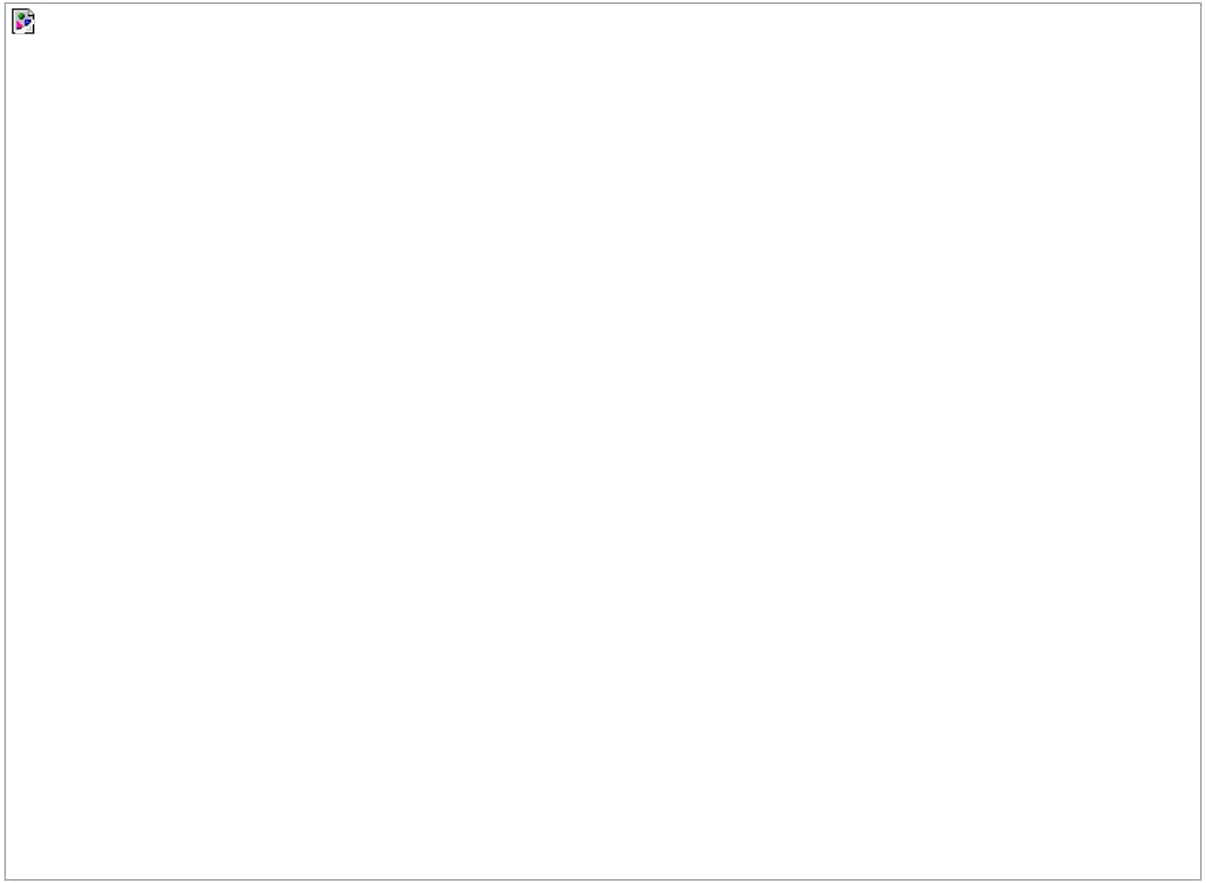
OL Architecture



Worker (no caching)



Worker (caching)



Package Caching

- Pre-initialized interpreters = packages already in memory
- Performance:
  - How do we choose which packages to cache (import before fork)?
  - How do we evict packages? Can’t remove an imported package from a running interpreter
  - How do we match a request to an interpreter in the pool?
  - How can we leverage this caching in the load balancer?
- Security:
  - Importing on host machine is dangerous
    - Don’t want to have to whitelist packages…
  - Packages could overwrite global namespace (shared between all)
- Evaluation:
  - How do we generate a representative workload to evaluate caching mechanisms?

Questions?