OpenWhisk Tech Interchange

2/19/20 host: rodric @rabbah

- New features, Pull request status & help needed
- Release plans
- Containerless functions (preliminary results, next steps)

Notable Updates

- Enhancements to “action loop” proxy implemented in Go
  https://github.com/apache/openwhisk-runtime-go/pull/121
  https://github.com/apache/openwhisk-runtime-python/pull/82
  - Add asynchronous handshake between proxy and runtime
  - Better detection of failed function init
  - Normalizes log extraction to match “old” style
  - Applies improved proxy to Python 2,3,3ai runtimes
- Native support for TypeScript
  https://github.com/apache/openwhisk-runtime-nodejs/pull/160
  - Pending final comments

PRs, ready to merge?

- CosmosDB indexing PR 4807 (ready to merge?)
- Elastic store for activations PR 4724 (small nits, ready to merge?)
- Metrics for web action results PR 4726 (ready to merge?)
- Encrypt parameters PR 4756 (pending final review)
- Java 11 PR 4706 (performance concern on dev list, merge?)

PRs, help needed

- waitTime for sequences PR 4819 (help with review needed)
- Admin interface to change runtimes PR 4790 (pending tests, status?)
- Invoker health check PR 4698 (status?)
- Optional activation results PR 4659 (status?)

Release Plans

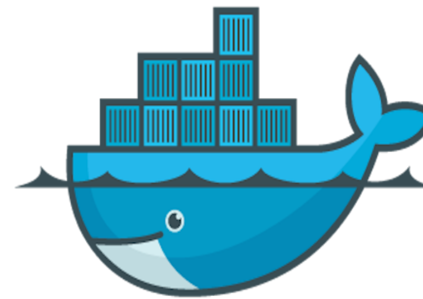
- Dave Grove to review remaining items for core release

Containerless Functions

Serverless Elasticity

resource isolation and provisioning

isolates
5ms



containers
500ms

Cloudflare Workers

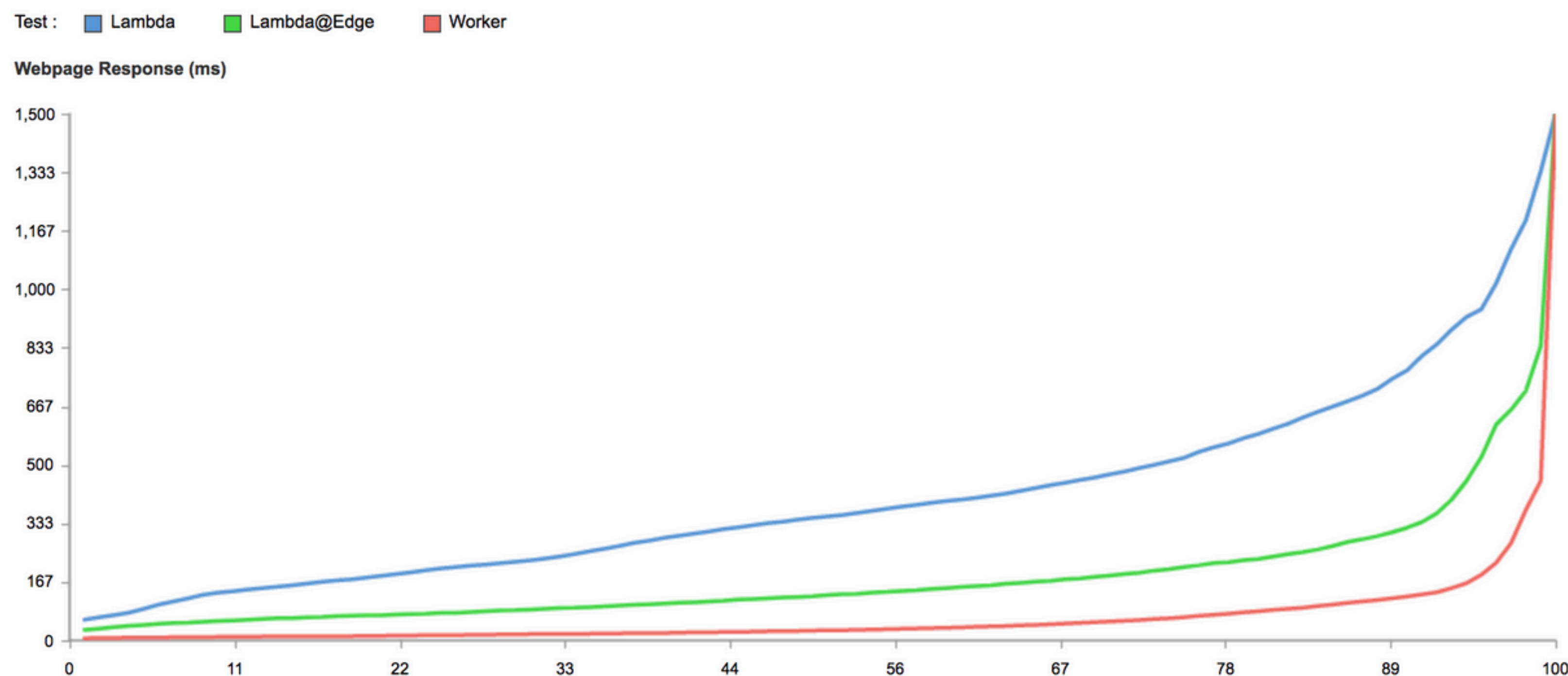
Zack Bloom

7/2/2018, 1:50:42 PM EDT

A few months ago we released a new way for people to run serverless Javascript called Cloudflare Workers. We believe Workers is the fastest way to execute serverless functions.

If it is truly the fastest, and it is comparable in price, it should be how every team deploys all of their serverless infrastructure. So I set out to see just how fast Worker execution is and prove it.

tl;dr Workers is much faster than Lambda and Lambda@Edge:



V8 vs. WASM

waSCC Showcase Docs Posts Projects Tutorials

waSCC

A dynamic, elastically scalable WebAssembly host runtime for securely connecting actors and capability providers

Build your *functions* and *services* in WebAssembly and run them *anywhere*.

Star 39

Get Started

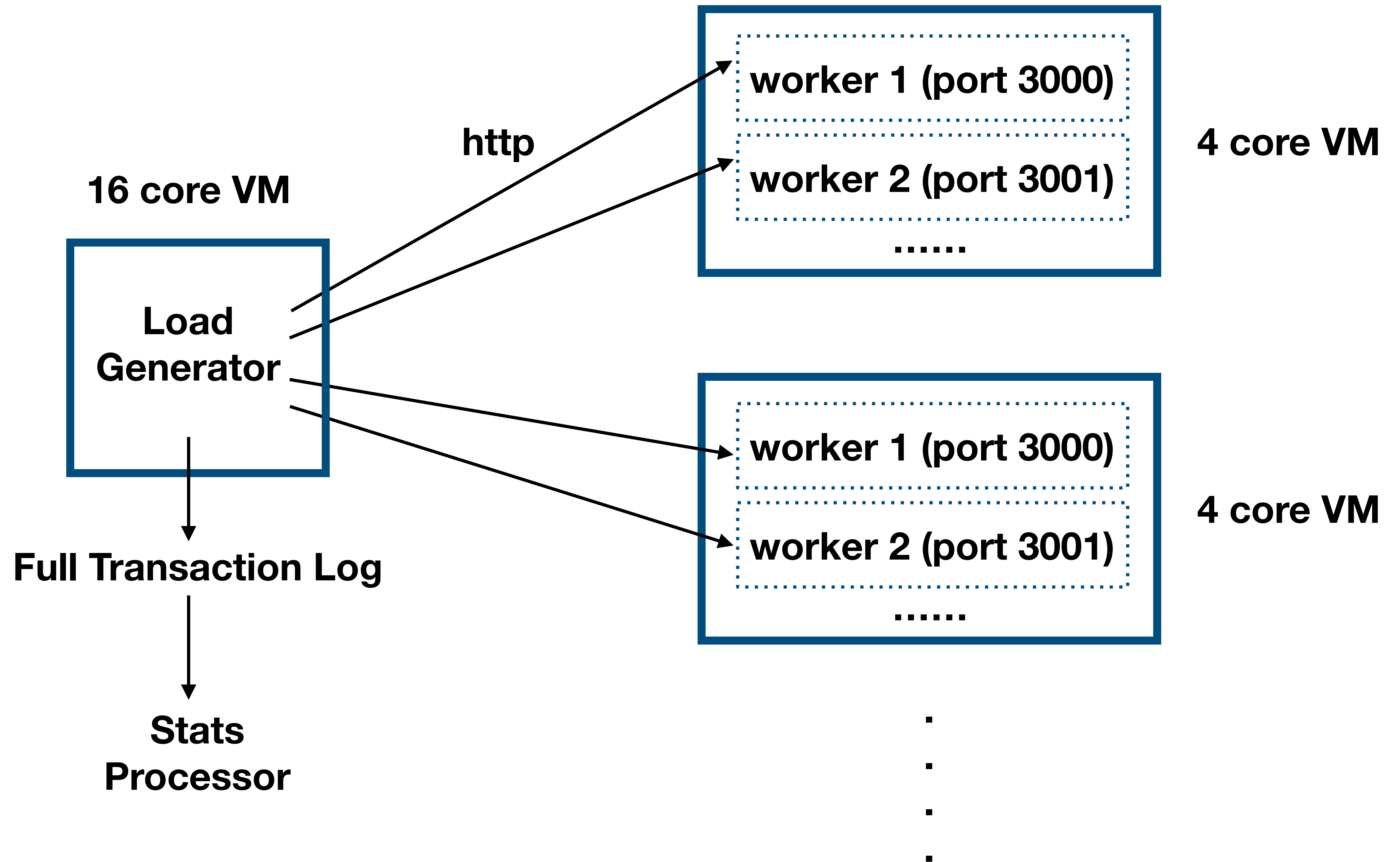
View Documentation ›

Latest release v0.3.0

Summary of **What** we’ve done

- Credits: Perry Cheng & Rodric Rabbah (Nimbella), Dragos Haut & Chetan Mehrotra & Tyson Norris (Adobe)
- What are isolates:
  - Strong memory isolation
  - String in, String out
  - Must widen interface (trampoline) for “fetch”, “promise”, “timeout”
  - Trampolines are attack surfaces
- What we wanted to know:
  - Under load, quantity performance and latency vs. containers
  - Quantify impact of isolate reuse on performance
  - Assume one function per isolate
- What we have done:
  - Load generator
  - Proof of concept integration with OpenWhisk Standalone Controller
- What’s next:
  - POEM-2 (thanks Dominic for POEM-1 http://bit.ly/wsk-poem-1)

Architecture



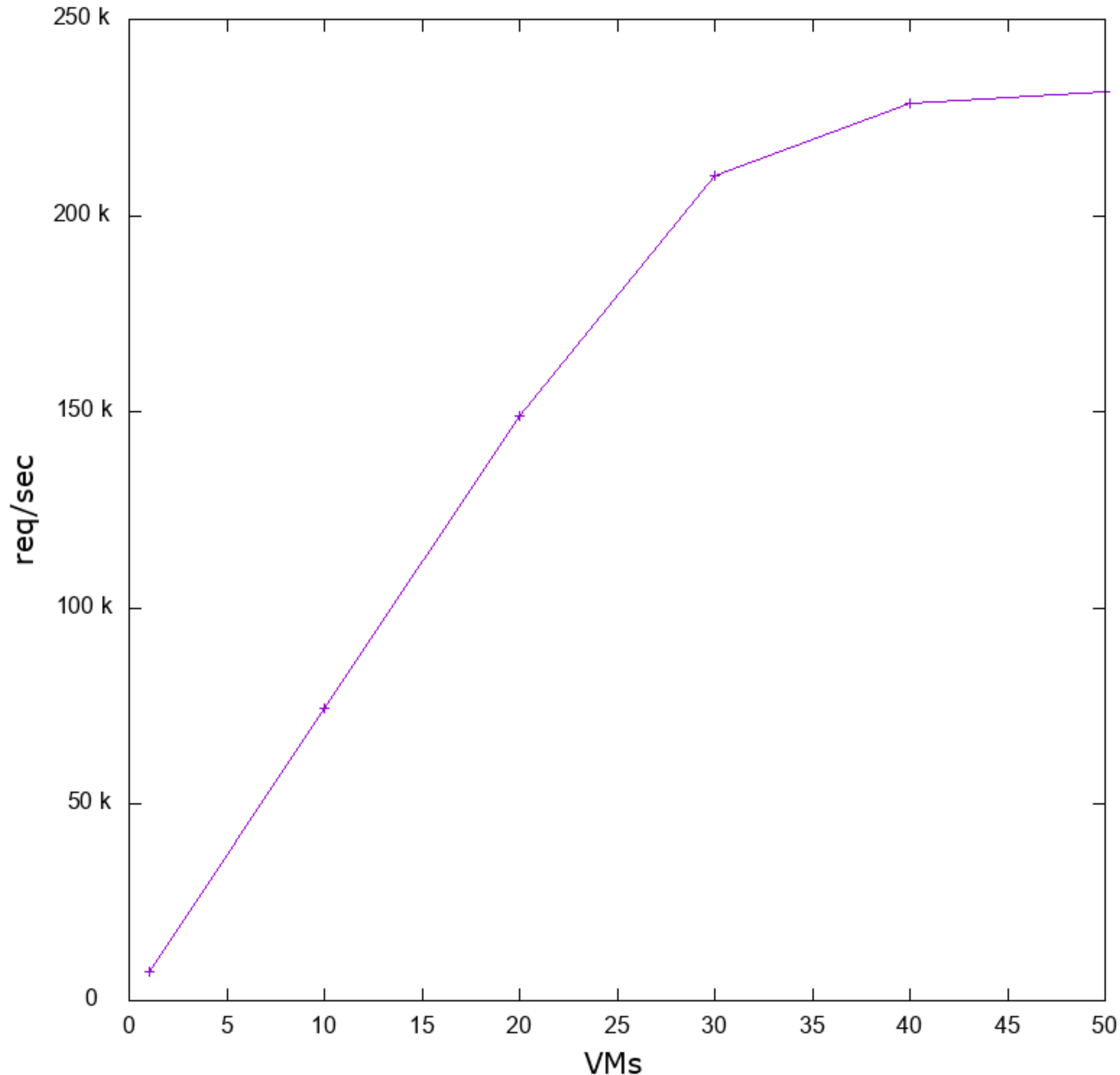
Load Generator

- Custom **Golang** load generator and stats processor
  - First tried wrk, hey, … but each had deficiencies
- **Configurable** with a json file specifying IP, URI+Verb, weight
- **Not a load balancer**
- **16 cores** to support 100K req/sec without perturbation
  - Since 4 core VM can sustain only 55-60K req/sec

Worker

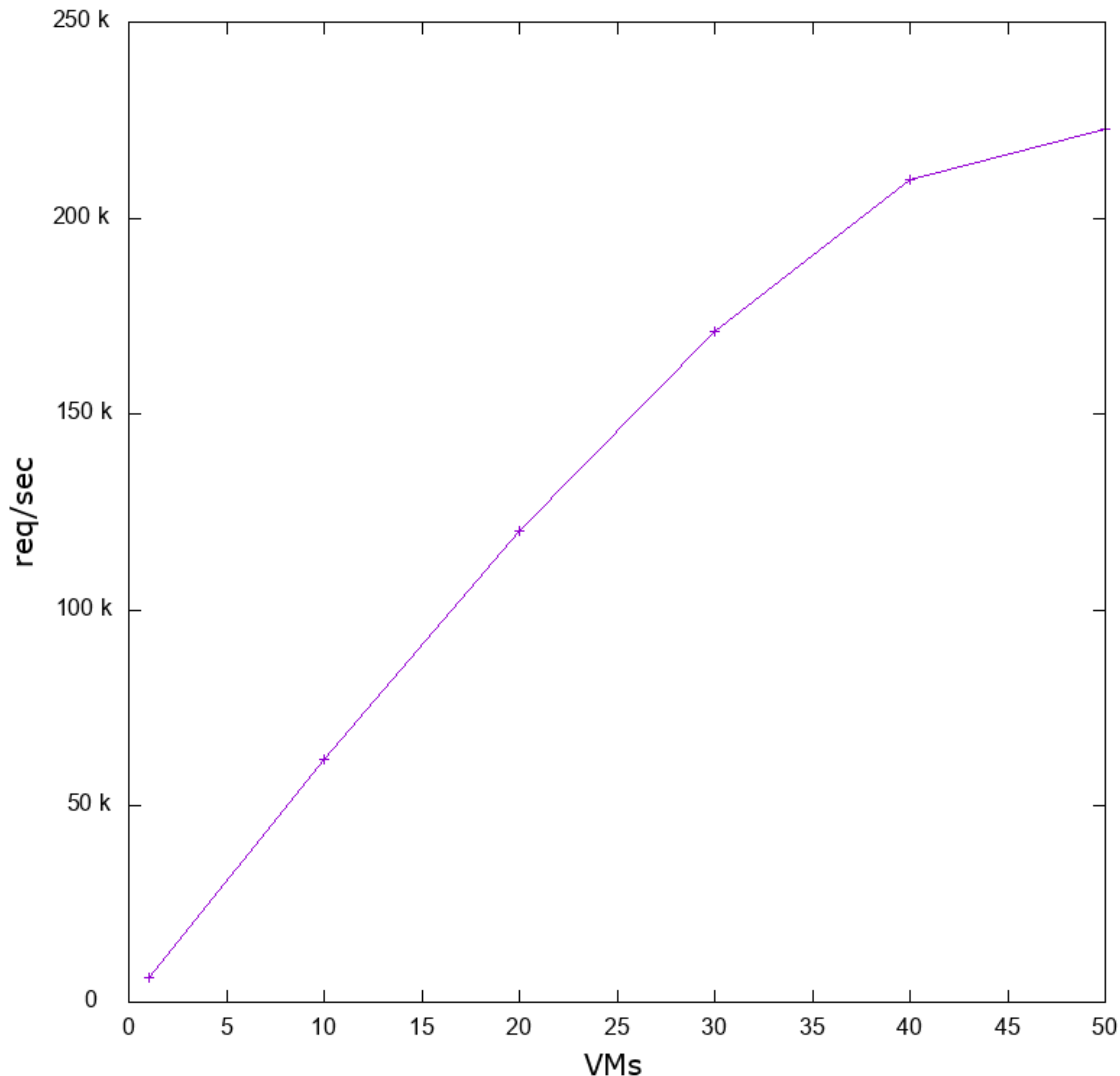
- Written in javascript node.js with express as the server
- **/status** endpoint immediately responds as a health check and establish rate limit
- **/eval** endpoint will accept a function name, function parameters, and a **mode**
  - **unsafe** = javascript's unsafe eval
  - **fresh** = a fresh V8 isolate for every eval
  - **isolate** = reuse V8 isolate if possible
- Run multiple instances per VM since javascript is single-threaded

Sanity Check



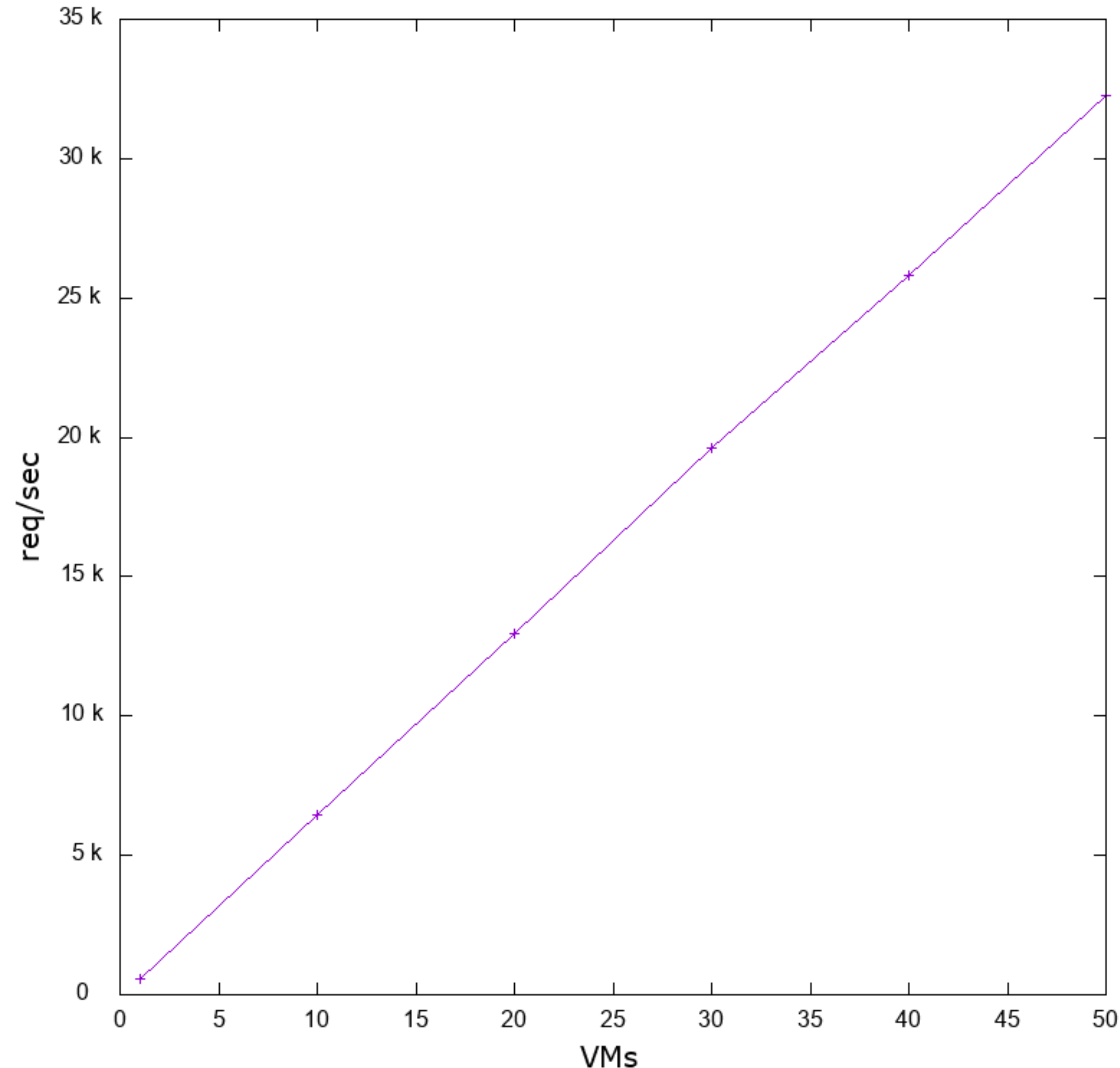
- Initial scaling is linear proving out the load generator's use of concurrency
- Leveling off at 230K is also expected since 16 core VM is rate limited based on core count
- Conclusion: For testing thing up to 100K (or even 150K), this framework is performant.

Rough Upper Bound



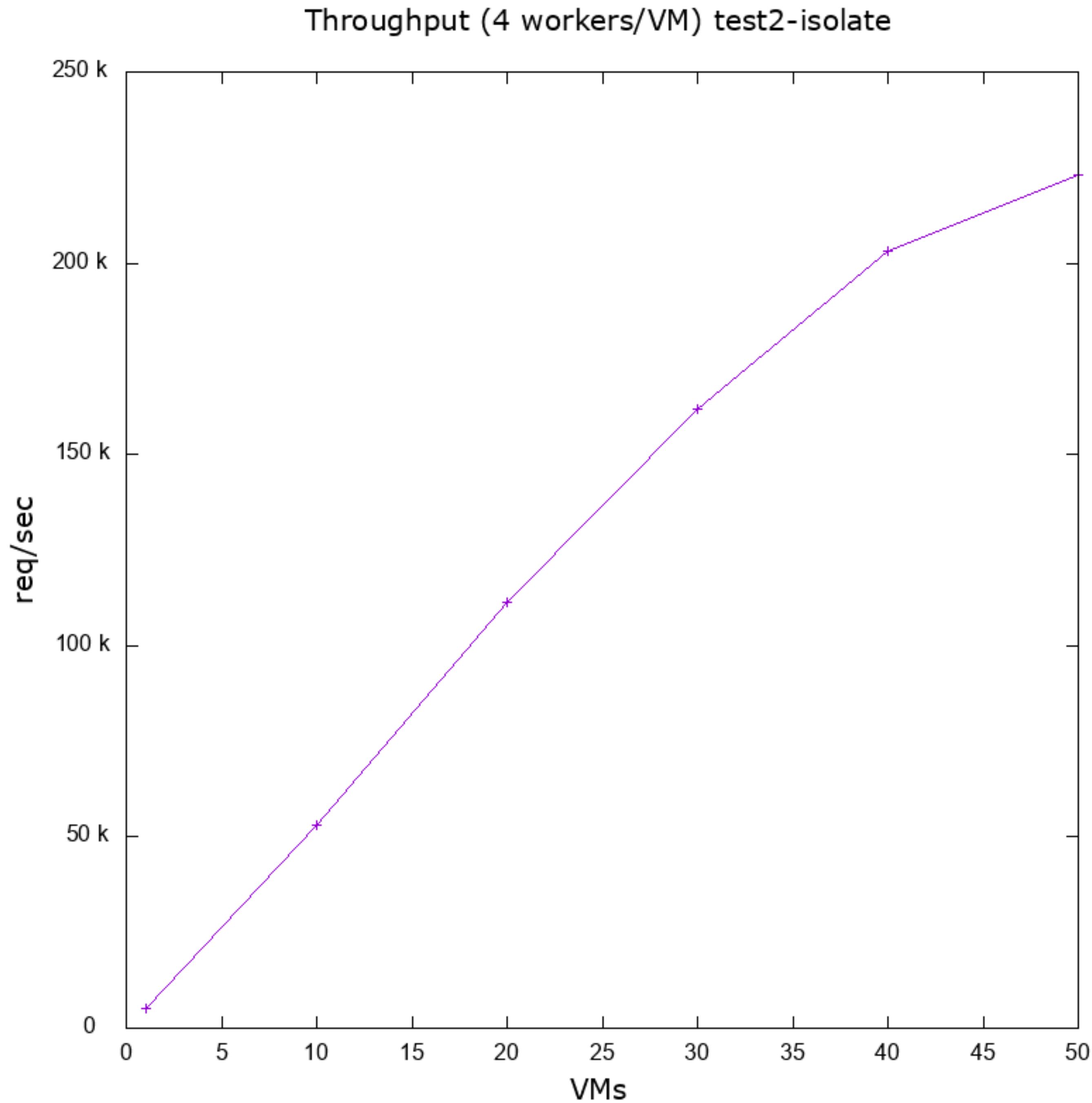
- Unsafe “eval” is almost as fast as expected at 220
- Test Function: Given a number, multiply by another number.
- Execution time still dwarfed by “eval” and not actual function

Lower Bound



- A fresh isolate for each function call is created and destroyed.
- 32K at 50VM means we need about 160VMs to achieve 100K
- Conclusion: Isolate creation is much faster than container creation but will still remain the bottleneck for short-running functions.

Full Reuse



- We maintain a cache map of all isolates indexed by the function.
- Need about 20 (4 core) VMs to get to 100K req/sec
- Conclusion: Reuse is **very important** for running short-short-running functions because isolate CRUD is still the bottleneck.