Apache OpenWhisk on Kubernetes

building a production-ready serverless stack on and for Kubernetes

David Grove
Apache OpenWhisk Committer
IBM Research

May 4, 2018

APACHE
OpenWhisk

Agenda

- Apache OpenWhisk
- Implementation and Deployment Architecture
  - Critical path of invoking a function
  - Container management alternatives
  - Empirical results
- Beyond simple functions: Serverless composition of Serverless functions
- Future Directions

Apache OpenWhisk

- Production-ready open source Functions-as-a-Service (FaaS)
  - Core FaaS runtime, CLI, language runtimes, provider packages and SDKs
  - Polyglot: JavaScript, Python, Swift, Java, PHP, Go, ... + “blackbox” containers
  - Extensible & pluggable
    - Service Provider Interfaces (SPIs) between many components
    - Additional language runtimes, provider packages, SDKs, etc.
    - Deployment options: docker-compose, VMs, Kubernetes, Mesos, OpenShift, …
- Hosted commercial deployments by IBM, Adobe, …
- Apache incubator project
  - Active open source developer + user community
  - Working towards official releases & graduation from incubation

OpenWhisk on Kubernetes

- Major stages of work
  1. Deploy OpenWhisk runtime containers as StatefulSets, DaemonSets, etc.
  2. Adjust OpenWhisk code/configuration/deployment to better fit Kubernetes
  3. Exploit Kubernetes capabilities to simplify OpenWhisk (early stages)
- https://github.com/apache/incubator-openwhisk-deploy-kube
  - Supports multiple versions of Minikube + Kubernetes
  - Configuration files + deployment scripts for Minikube, single, and multi-node clusters
  - Helm chart NEW!
- Demo later…

Underlying assumptions

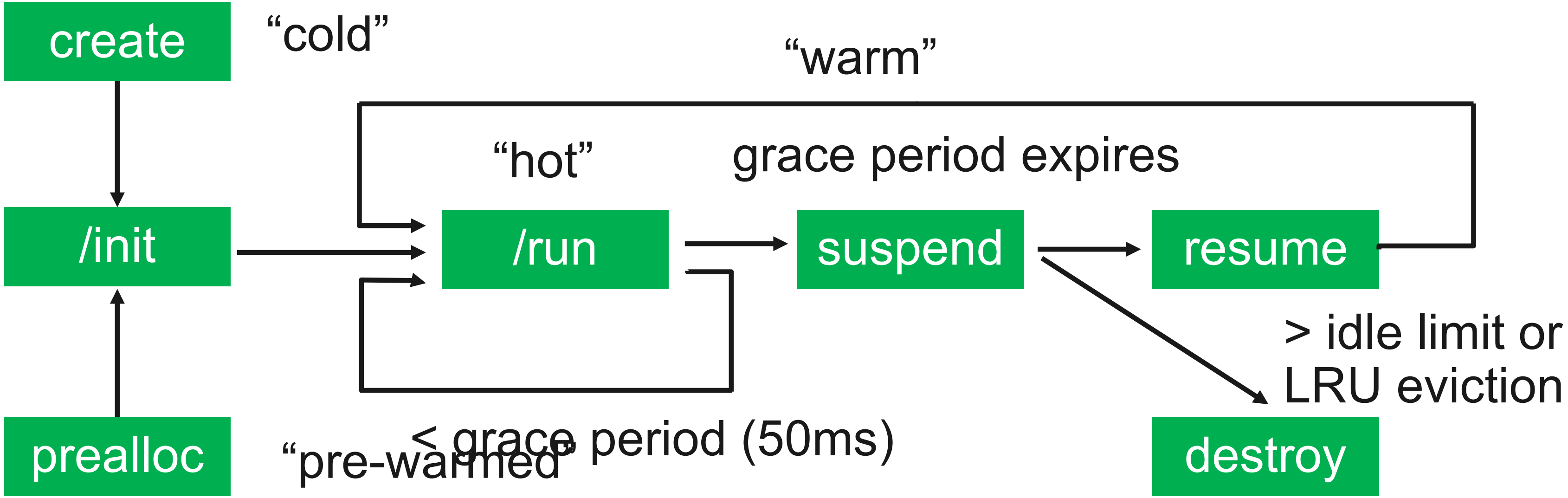
- What does “production ready” imply?
  - Large scale – many, many, many millions of function invocations a day
  - Replicated components – fault-tolerance & scalability
  - Multi-tenant – isolation & security when executing arbitrary code from many users
  - Integration with Provider’s Cloud Platform – IAM, logging, metrics, service bindings, …
- Emphasis on low latency and minimizing system overhead
  - Interactive applications: mobile backends, chatbots, …
  - Pipelines & compositions – one user request may spawn many function invocations
  - Short running functions – harder to amortize system overheads

Invoking a Function

- OpenWhisk executes each <user, function> in distinct containers
- A container may be reused to execute multiple invocations of its unique <user, function>

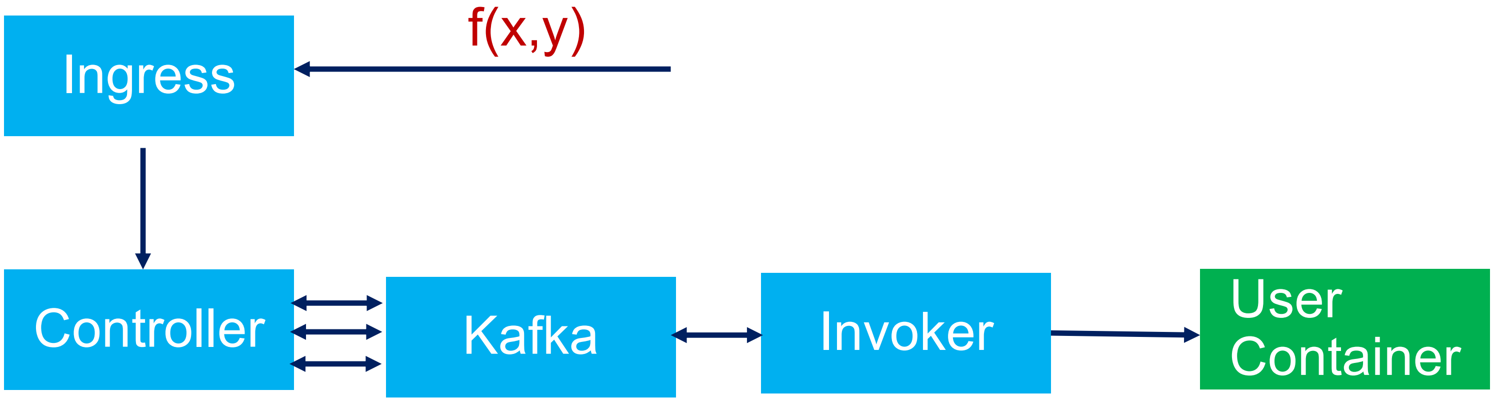
- Container scheduling, caching, and re-use are essential for scalable performance

User Container Life Cycle



Maintain pool of “stem cell” containers
for heavily used language runtimes

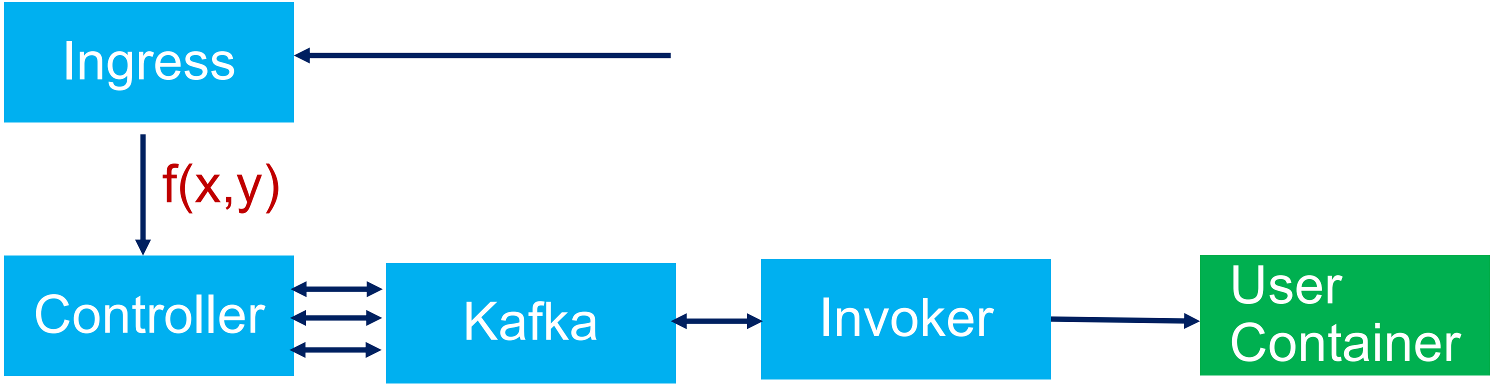
Critical Path of Function Invocation



Ingress

- SSL termination
- Forward to appropriate controller

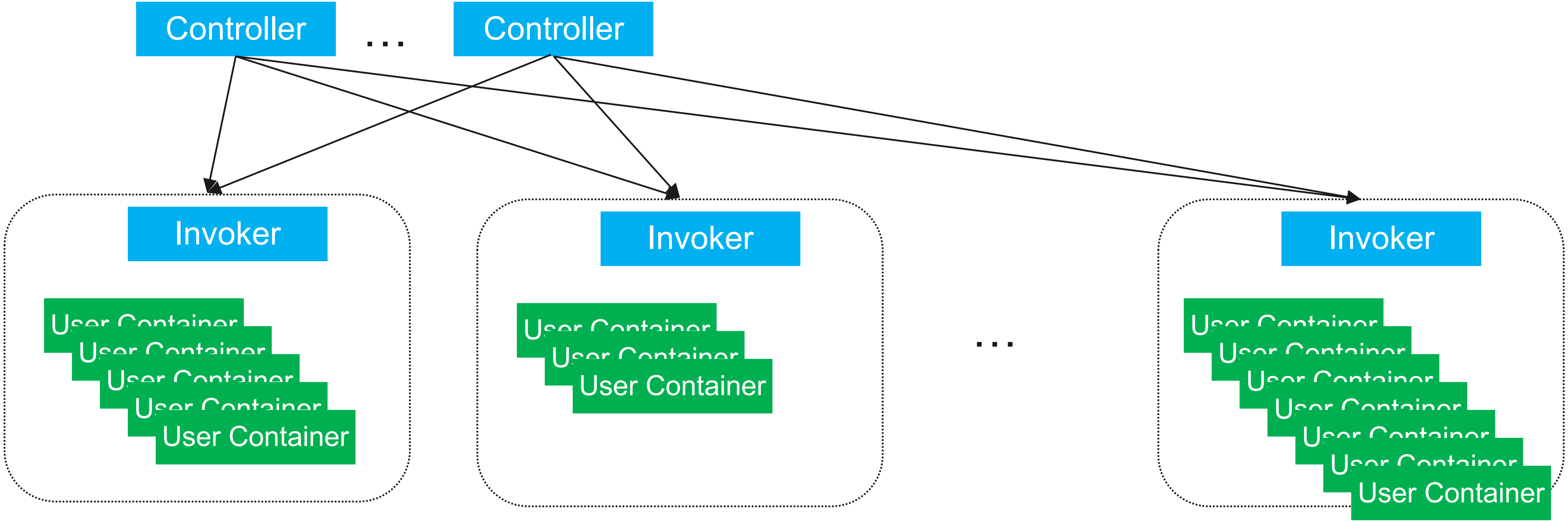
Critical Path of Function Invocation



Controller

- Maintains open http connection for blocking invocation
- Authenticates actor & authorizes resource access
- Admission control (per-actor limits)
- Load Balancer: select invoker to handle this invocation

Load Balancer Algorithm



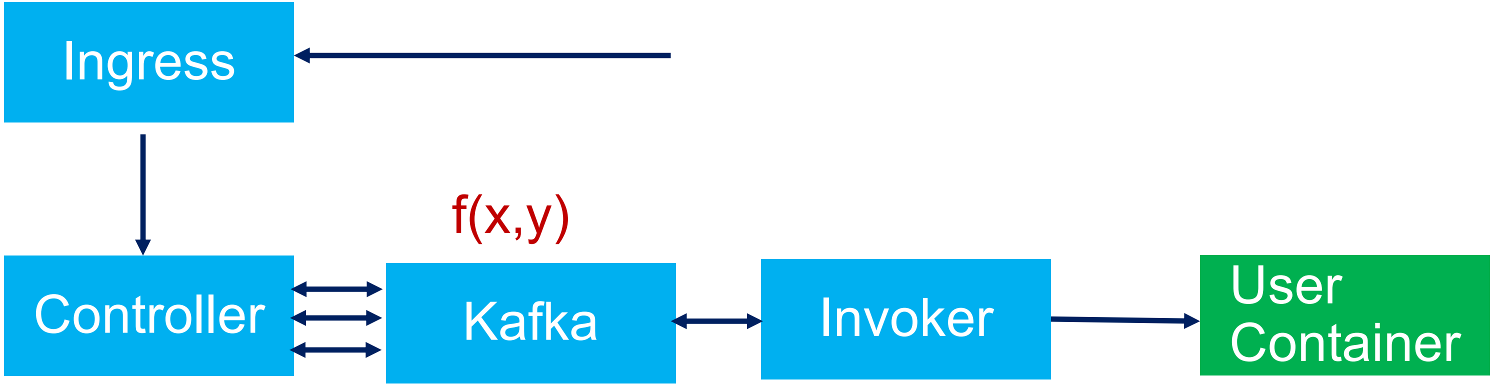
Goal: minimize invocation latency and maximize system throughput

Challenge: scale, replication, and rapid state change

Currently: heuristically use hashing for locality and queue length to approx. load

SPI: enable algorithmic exploration

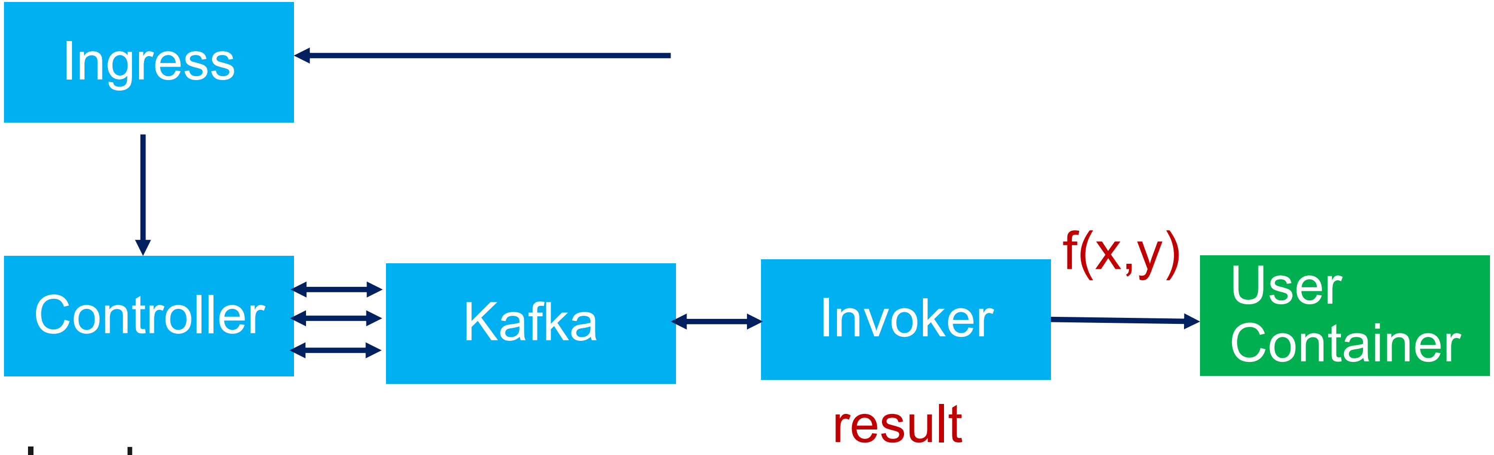
Critical Path of Function Invocation



Kafka

- One topic (queue) per invoker to hold backlog

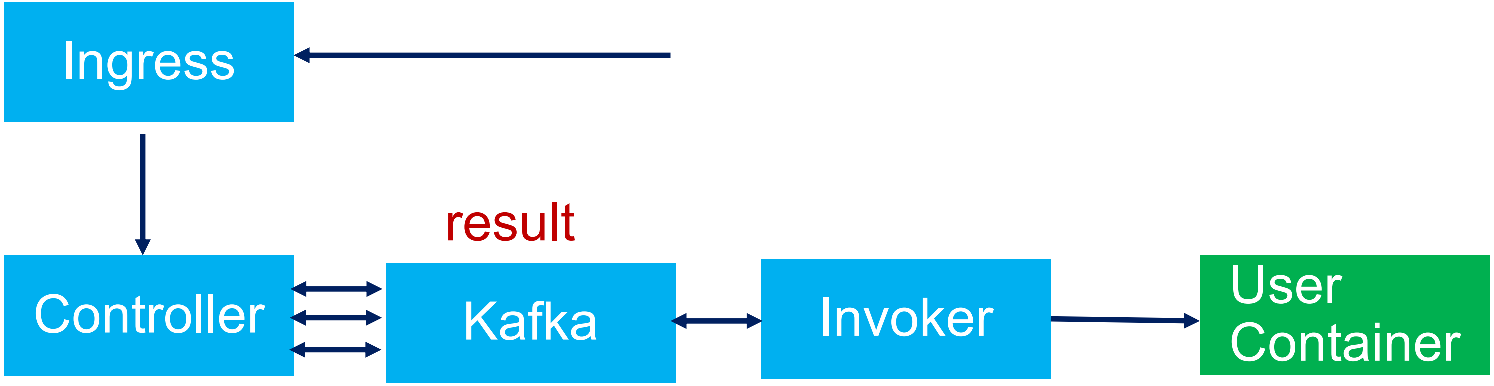
Critical Path of Function Invocation



Invoker

- Select (create) container to execute request
- Invoke action and await results
- Enqueue result in initiating controller’s “completed” topic
- Extract user container logs & forward to logging service

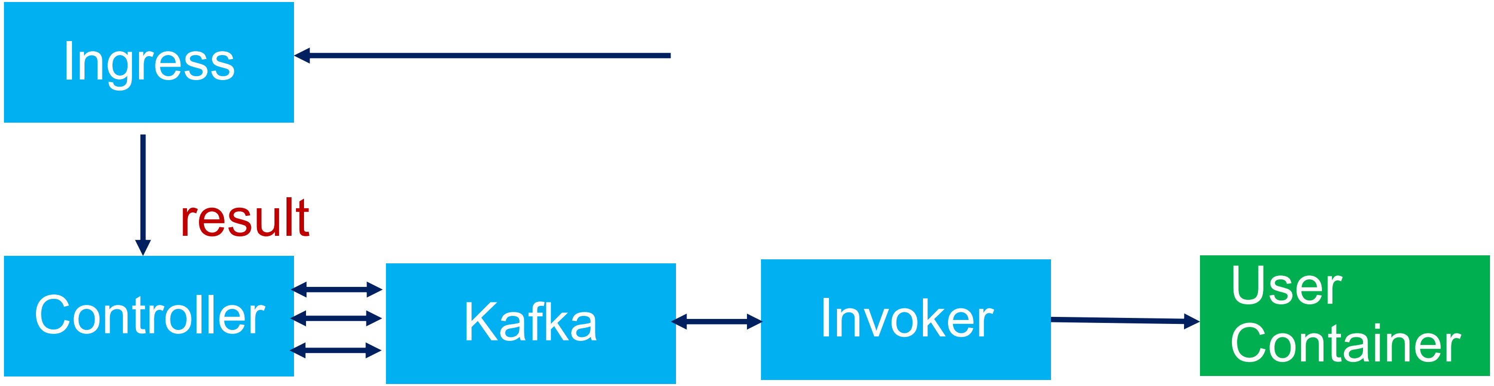
Critical Path of Function Invocation



Kafka

- One completed topic per controller to hold backlog

Critical Path of Function Invocation



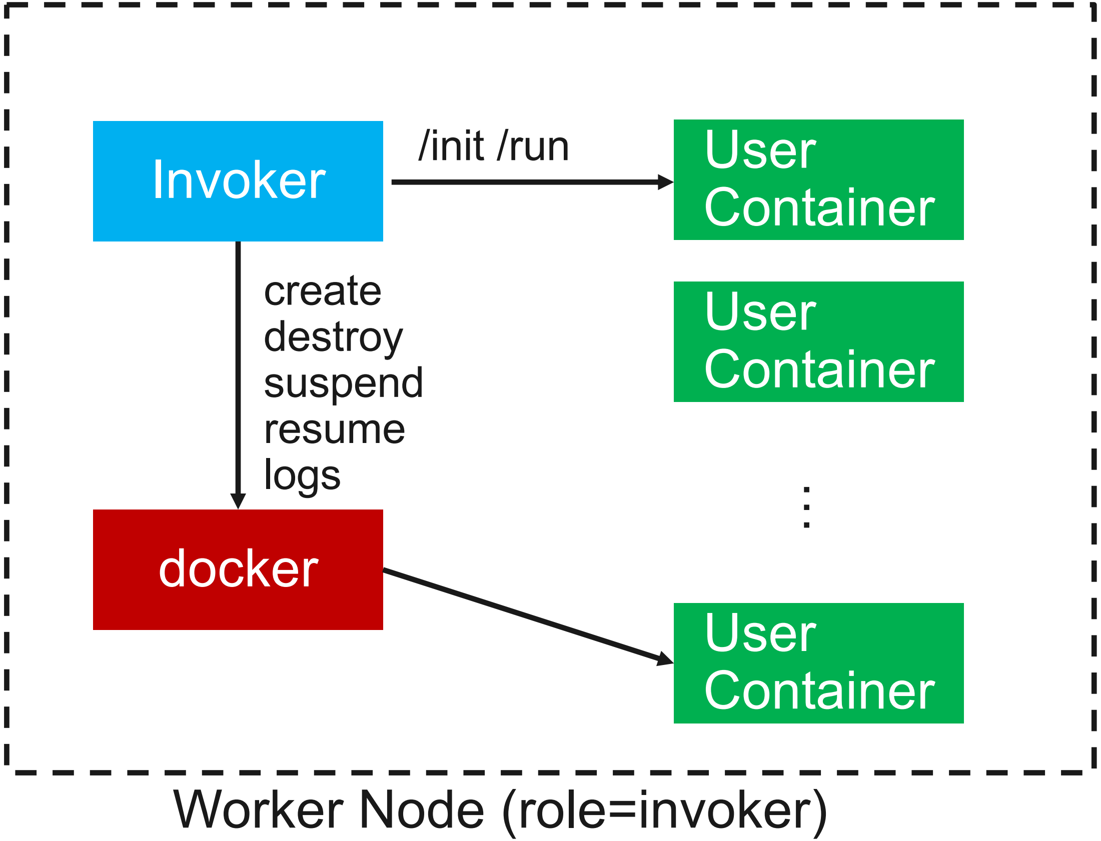
Controller

- If blocking request, return result on open http connection

Invoker – ContainerFactorySPI

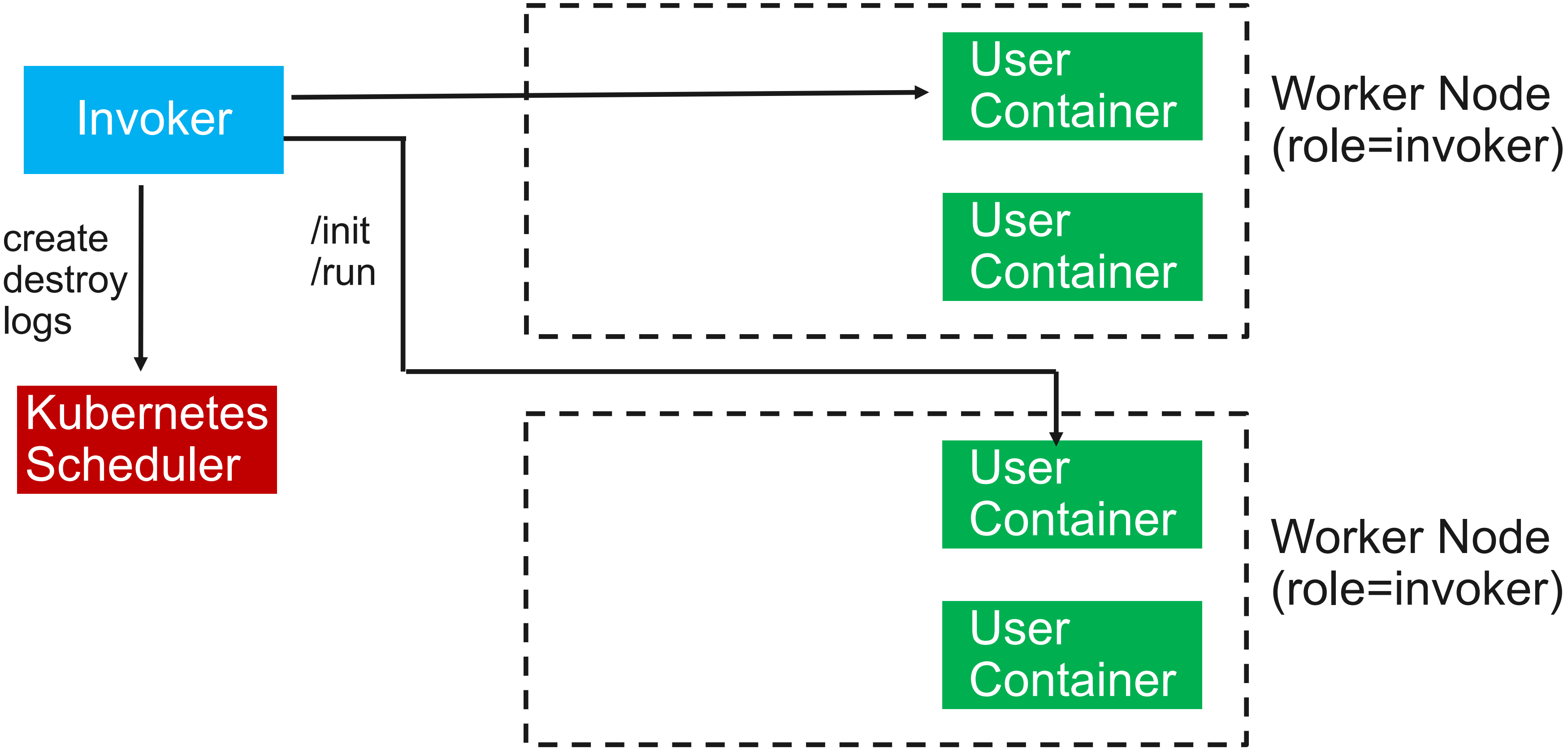
- Pluggable abstraction for container engines
  - DockerContainerFactory
  - KubernetesContainerFactory
  - MesosContainerFactory
  - …
- OpenWhisk on Kubernetes uses:
  - DockerContainerFactory
  - KubernetesContainerFactory

Invoker - DockerContainerFactory

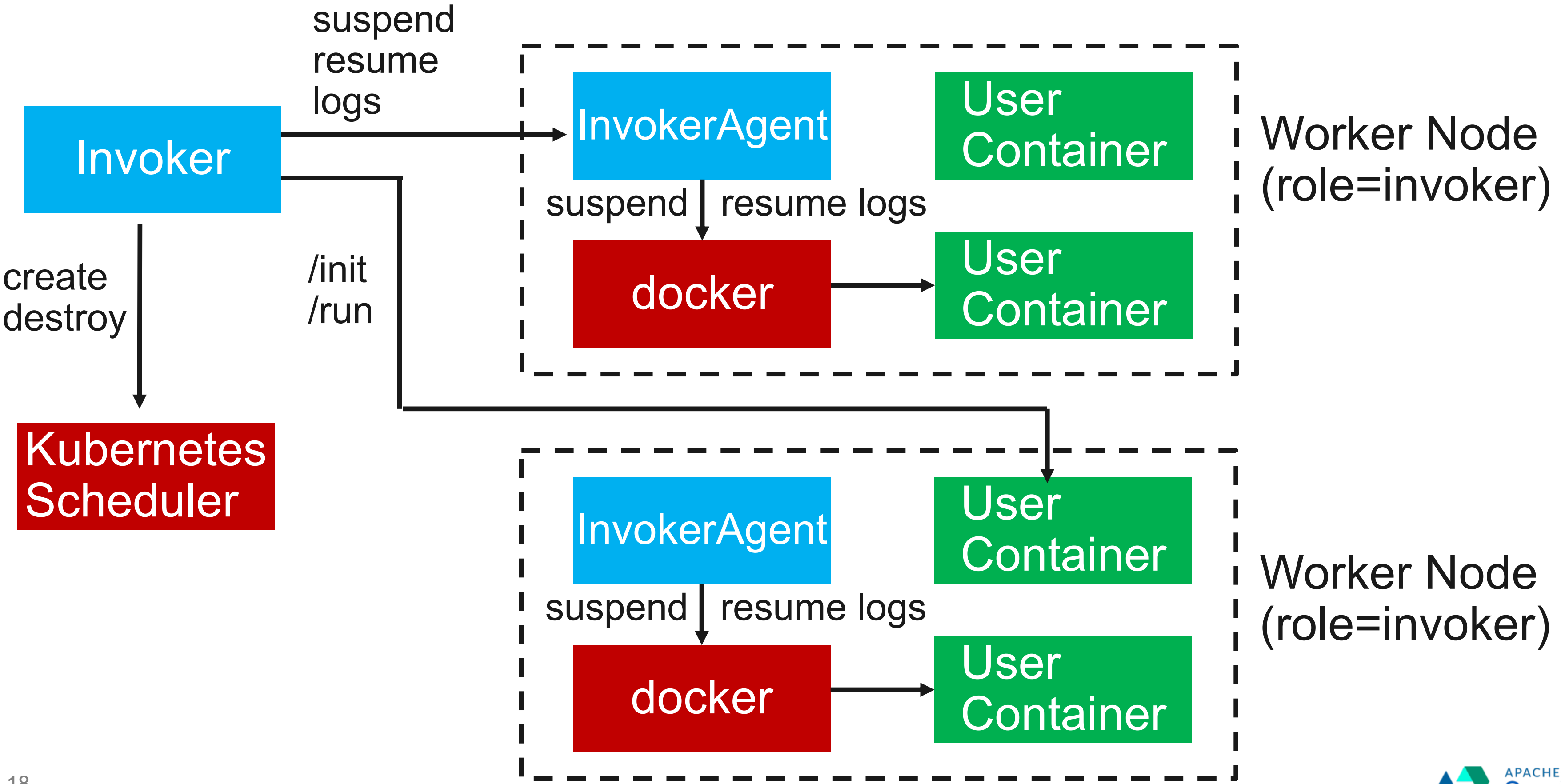


Kubernetes manages OpenWhisk control plane, but not user containers

Invoker - KubernetesContainerFactory



Invoker – KubernetesContainerFactory + InvokerAgent



Experimental Setup

- Goal: understand current performance of ContainerFactory implementations
  - What is the cold start latency?
  - What is the latency when able to re-use a container?
  - Are there throughput differences? If so, why?
- Kubernetes 1.8.8 cluster with 13 worker nodes
  - 2 control plane nodes (16 core x 64GB) + 1 load driver node (16 core x 64 GB)
  - 10 invoker nodes (4 core x 16 GB)
- All experiments
  - Measure full path, starting with Ingress
  - Test driver runs on load driver node (eliminate variable network delays)
  - Use trivial “no-op” actions to emphasize system overheads

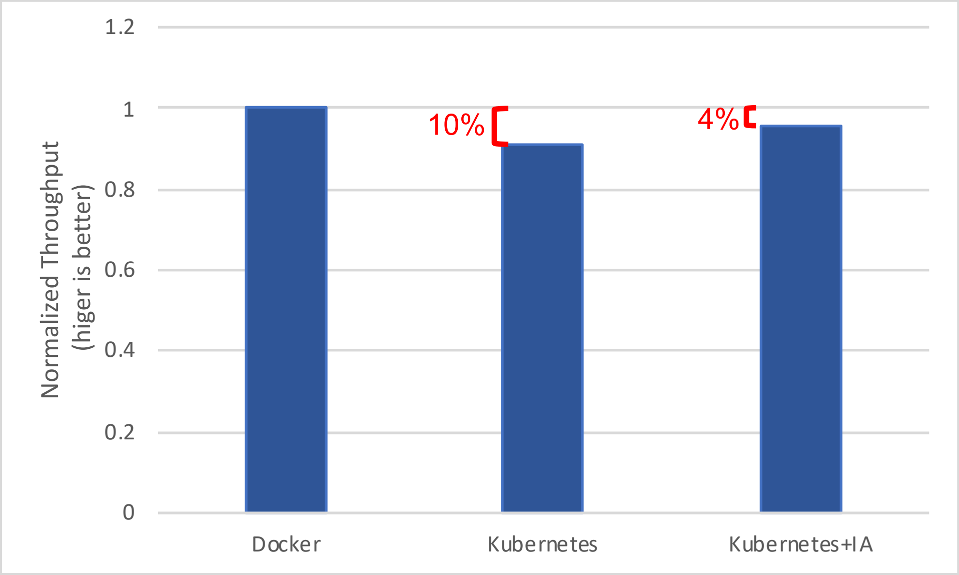
Latency Results

- Measure response time for serially invoking a blocking function (10,000 iterations)

Scenario	Container Factory	P50 (ms)	P90 (ms)	P95 (ms)	P99 (ms)
Cold Start	Docker	720	764	787	1,017
	Kubernetes	2,069	2,612	2,949	3,423
	Kubernetes + IA				
Warm Container	Docker	7	8	11	36
	Kubernetes	19	44	214	602
	Kubernetes + IA	8	11	13	24

Throughput Results

- Workload: load test of many concurrent non-blocking invocations



Analysis:

- Higher log extraction latency delays container reuse, reducing overall maximum achievable throughput

Takeaways / Future Work

- Higher cold-start costs (scheduling, pod creation)
  - Significant latency impact for short-running functions
  - Emphasizes importance of container re-use and caching

- Even though log extraction is “off the critical path” its performance still matters
  - Compute load on “idle” invokers (latency of Kubernetes vs. Kubernetes+IA)
  - Reduction in overall system throughput

Agenda

- Apache OpenWhisk
- Implementation and Deployment Architecture
  - Critical path of invoking a function
  - Container management alternatives
  - Empirical results
- Beyond simple functions: Serverless composition of Serverless functions
- Future Directions

Rethink the cloud programming experience

*We are programming a **cloud computer**.*

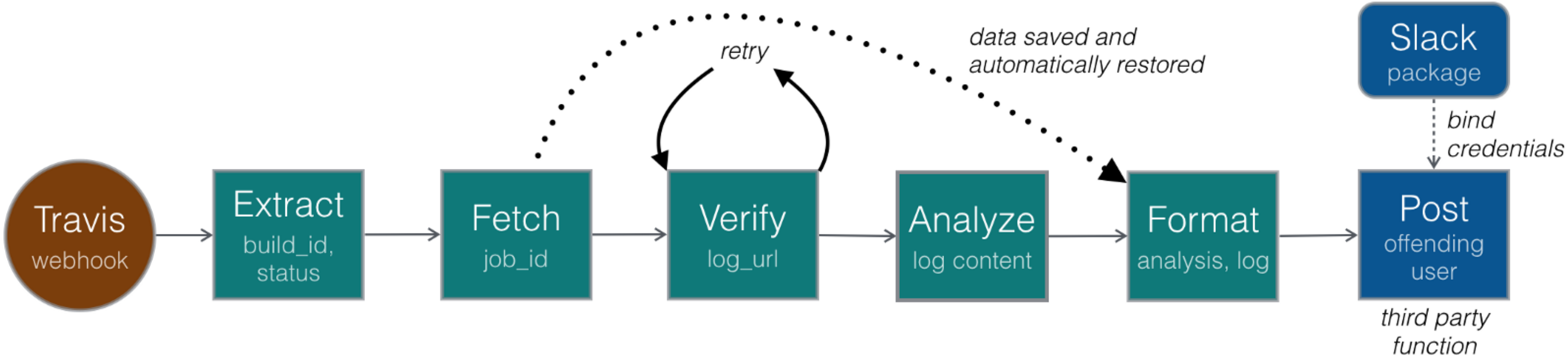
Can we provide a useful computer-like facade over a distributed system?

*Can we do this by **writing actual programs**,*

*and have **productivity enhancing tools** available to help us?*

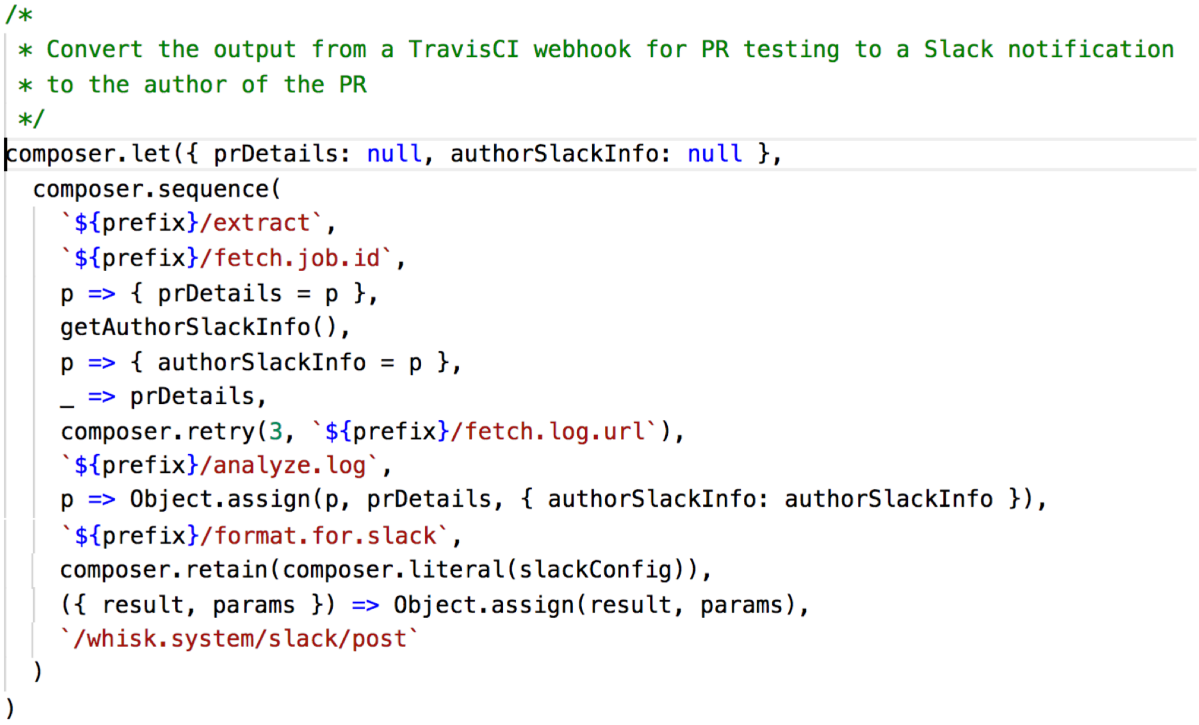
Demo – Composer + Fsh

- Notify me via Slack when Travis finishes testing my pull request

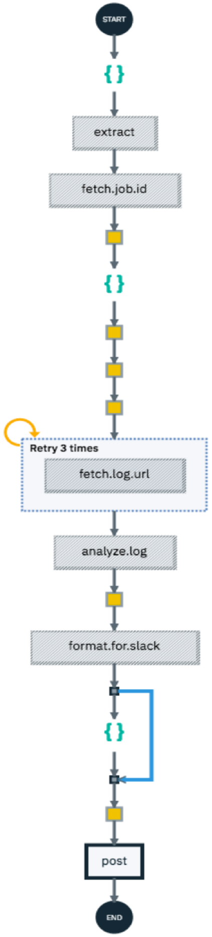


https://github.com/rabbah/travis-to-slack

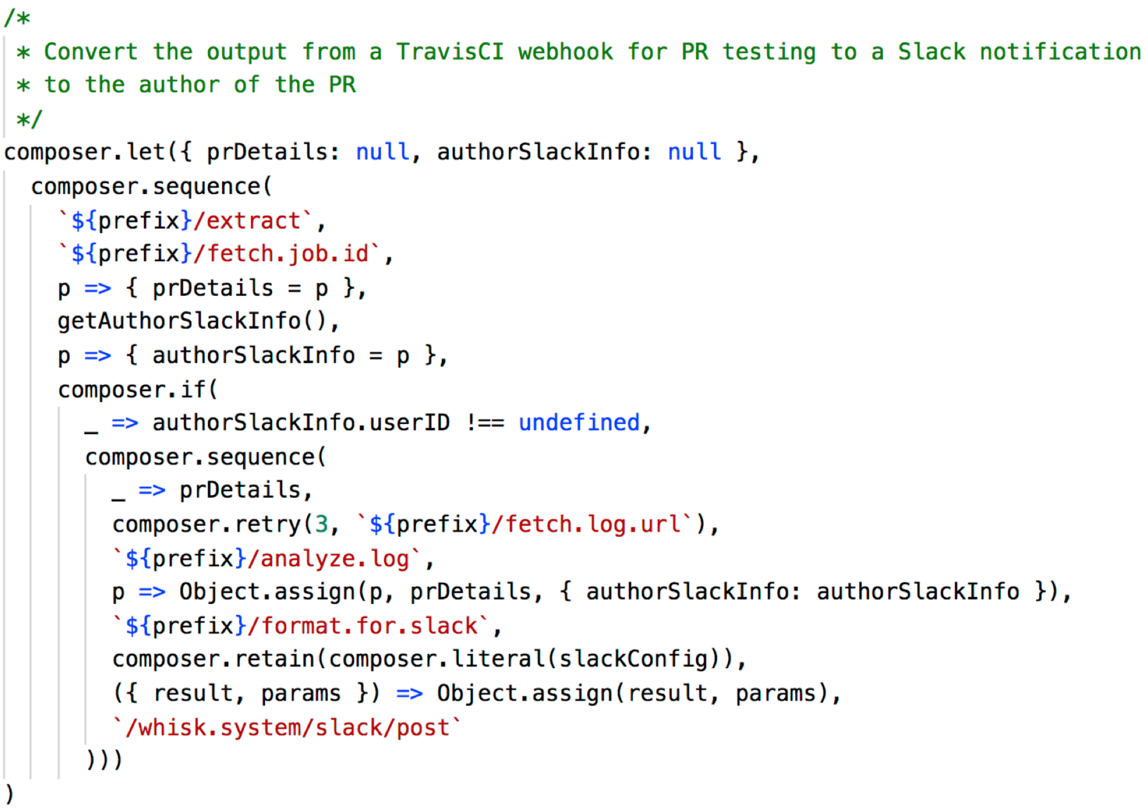
Demo – Composer + Fsh



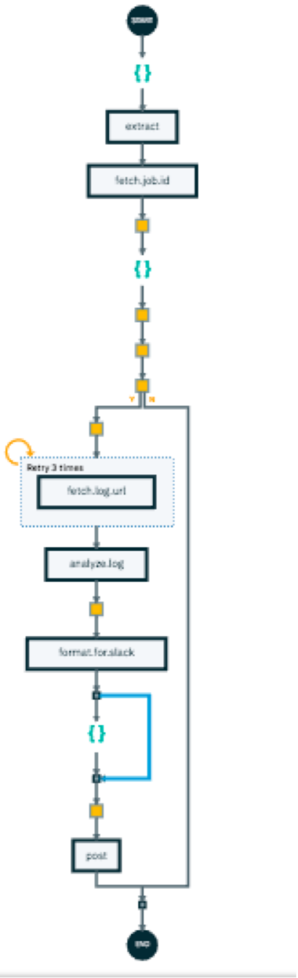
```
/*
 * Convert the output from a TravisCI webhook for PR testing to a Slack notification
 * to the author of the PR
 */
composer.let({ prDetails: null, authorSlackInfo: null },
  composer.sequence(
    `${prefix}/extract`,
    `${prefix}/fetch.job.id`,
    p => { prDetails = p },
    getAuthorSlackInfo(),
    p => { authorSlackInfo = p },
    _ => prDetails,
    composer.retry(3, `${prefix}/fetch.log.url`),
    `${prefix}/analyze.log`,
    p => Object.assign(p, prDetails, { authorSlackInfo: authorSlackInfo }),
    `${prefix}/format.for.slack`,
    composer.retain(composer.literal(slackConfig)),
    ({ result, params }) => Object.assign(result, params),
    `/whisk.system/slack/post`
  )
)
```



Demo – Composer + Fsh



```
/*
 * Convert the output from a TravisCI webhook for PR testing to a Slack notification
 * to the author of the PR
 */
composer.let({ prDetails: null, authorSlackInfo: null },
  composer.sequence(
    `${prefix}/extract`,
    `${prefix}/fetch.job.id`,
    p => { prDetails = p },
    getAuthorSlackInfo(),
    p => { authorSlackInfo = p },
    composer.if(
      _ => authorSlackInfo.userID !== undefined,
      composer.sequence(
        _ => prDetails,
        composer.retry(3, `${prefix}/fetch.log.url`),
        `${prefix}/analyze.log`,
        p => Object.assign(p, prDetails, { authorSlackInfo: authorSlackInfo }),
        `${prefix}/format.for.slack`,
        composer.retain(composer.literal(slackConfig)),
        ({ result, params }) => Object.assign(result, params),
        `/whisk.system/slack/post`
      )))
)
```



What did we just see?

- We can build a more computer-like programming environment for the Cloud!

- Composer
  - Familiar programming constructs: variables, control structures, …
  - Embedded (JavaScript) Domain Specific Language to compose polyglot applications

- Shell
  - Electron-based REPL that runs locally on developer’s machine
  - Rich visualizations to support programming tasks
  - Smooth integration with CLI, code editors, etc.

Ongoing Work

- OpenWhisk on Kubernetes
  - Scalability and performance
  - Enhancement of KubernetesContainerFactory-based Invoker
    - Larger container pools ➔ more options for scheduling/caching policies
    - Fuller integration with Kubernetes scheduler (but latency/scale may be challenging)
  - Better exploitation of Kubernetes deployment and management capabilities
  - Smoother developer transition between FaaS and Kubernetes-deployed microservices
- Composer/Shell
  - Next major step in evolution of cloud developer experience?’
  - Active area of research and development

Get Involved

- OpenWhisk
  - Web: https://openwhisk.apache.org/
  - GitHub:
    - https://github.com/apache/incubator-openwhisk
    - https://github.com/apache/incubator-openwhisk-deploy-kube
  - Slack: http://slack.openwhisk.org/
- Composer/Shell
  - https://github.com/ibm-functions/composer
  - https://github.com/ibm-functions/shell