FaaS-Utility

FaaS-Utility: Tackling FaaS Cold Starts with User-preference and QoS-driven Pricing

Henrique Santos, José Simão, Luís Veiga

GECON 2024 - 20th International Conference on the Economics of Grids, Clouds, Systems, and Services
Rome, Italy, 2024/09/26

Cloud Services and FaaS

- Today's Cloud -> Multiple Cloud Services
- FaaS:
  - Stateless-functions
  - (mostly) Short computation

Cloud Services	Business Logic	App	Data	Runtime/ OS	Virtualization/ Storage	Examples
On-premise	User	User	User	User	User	Home Computer
IaaS	User	User	User	User	Provider	Apache Cloudstack
PaaS	User	User	User	Provider	Provider	AWS Elastic Beanstalk
BaaS	User	User	Provider	Provider	Provider	Google Firebase
FaaS	User	Provider	Provider	Provider	Provider	Apache OpenWhisk
SaaS	Provider	Provider	Provider	Provider	Provider	Google Workspace

FaaS Benefits and Use cases

- Benefits
  - Suitable implementation for micro-distributed APIs
  - Naturally highly available
  - Automatically scalable
  - Application scalability and availability not user’s concerns
  - Serverless, backend servers hidden from FaaS function

- Use cases
  - Edge computing
  - Image and video processing
  - Machine learning
  - Scientific computing
  - Event streaming

Current Shortcomings Challenges

- Current FaaS related works
  - Focus on optimization of system resources and performance
  - Little attention to the individual desires of each customer
  - Little focus on flexible pricing mechanisms
    - mostly best-effort or vs. permanent dedicated instances

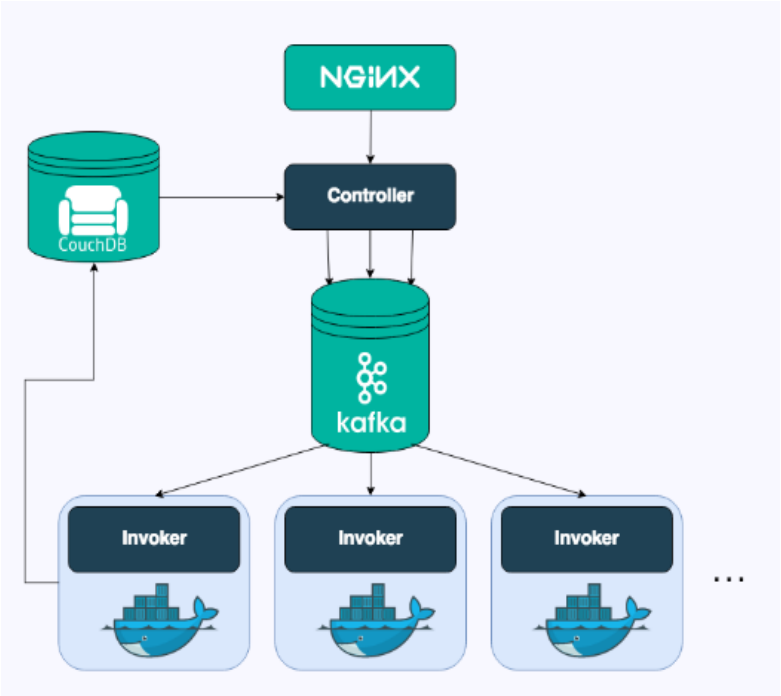
  - Cold start delays in function invocation
  - Focus on optimization of system resources and performance

Proposal

- Extension to FaaS scheduling mechanism in OpenWhisk
- Incorporate Utility-awareness in Faas scheduling/resource allocation
- Client-side:
  - Takes into account customer differences in priority/urgency/QoS
  - Utility expressed in priority/urgency parameter (α) like a slider
  - Implemented via two core approaches
    - more aggressive (extra) container pre-warming/allocation
    - multiple functions invocations returning the fastest result
    - *(assumes function idempotence)*
- Provider-Side:
  - Enables QoS-differentiated service offerings (competitiveness)
  - Allows higher profits adjusting price depending on priority desired by client

Apache OpenWhisk Architecture Overview

- Action
- Trigger
- Rule
- NGINX (REST interface)
- Controller
- Kafka-based distributed message broker
- CouchDB-based Database



Apache OpenWhisk Built-in Scheduler

$Action \leftarrow A$
$ActionContainer \leftarrow Action$
for all $Invokers$ **do**
  if $BusyPoolSize = MaxPoolSize$ **then**
    continue
  else if $ActionContainer \in FreePool$ **then**
    $FreePool \leftarrow FreePool \setminus ActionContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
    return
  else if $PreWarmPoolSize > 0$ **then**
    $PreWarmPool \leftarrow PreWarmPool \setminus PreeWarmContainer$
    $ActionContainer \leftarrow PreWarmContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  else if $FreePoolSize + BusyPoolSize = MaxPoolSize$ **then**
    $FreePool \leftarrow FreePool \setminus LeastRecentContainer$
  else
    $ActionContainer \leftarrow ColdContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  end if
  return
end for
$Queue \leftarrow Queue \cup Action$

- An Action is given to the controller
- Controller oversees multiple Invokers.
- Home Invoker
- Each Invoker has a max capacity of containers
- Each Invoker has 3 types of pools:
  - **Busy Pool** (Action deployment)
  - **Free Pool** (Action Specific Warm Containers)
  - **PreWarm Pool** (PreWarm Containers)

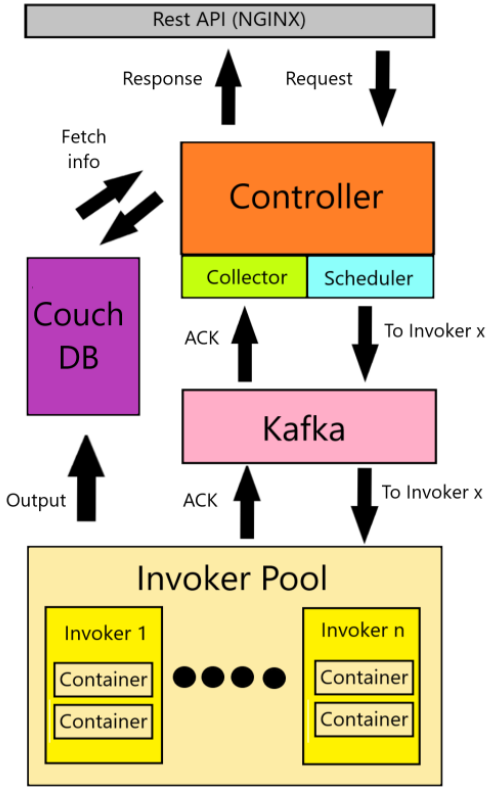
Apache OpenWhisk Built-in Scheduler

$Action \leftarrow A$
$ActionContainer \leftarrow Action$
for all $Invokers$ **do**
  if $BusyPoolSize = MaxPoolSize$ **then**
    continue
  else if $ActionContainer \in FreePool$ **then**
    $FreePool \leftarrow FreePool \setminus ActionContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
    return
  else if $PreWarmPoolSize > 0$ **then**
    $PreWarmPool \leftarrow PreWarmPool \setminus PreeWarmContainer$
    $ActionContainer \leftarrow PreWarmContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  else if $FreePoolSize + BusyPoolSize = MaxPoolSize$ **then**
    $FreePool \leftarrow FreePool \setminus LeastRecentContainer$
  else
    $ActionContainer \leftarrow ColdContainer$
    $BusyPool \leftarrow BusyPool \cup ActionContainer$
  end if
  return
end for
$Queue \leftarrow Queue \cup Action$

- An Invoker is at maximum capacity
- Warm Action Specific Container available to receive the requested action
- Attempting to use a PreWarm Container to receive the requested action
- Remove the Least Recent inactive container
- Create a new Cold Action Specific Container

Apache OpenWhisk Architecture Extension

- Controller Extension
  - Direct modification to existing controller
  - Incorporate Utility-aware policy
  - Scheduler: *“Action-Spreading”* algorithm
  - Message Collector: handle multiple replies

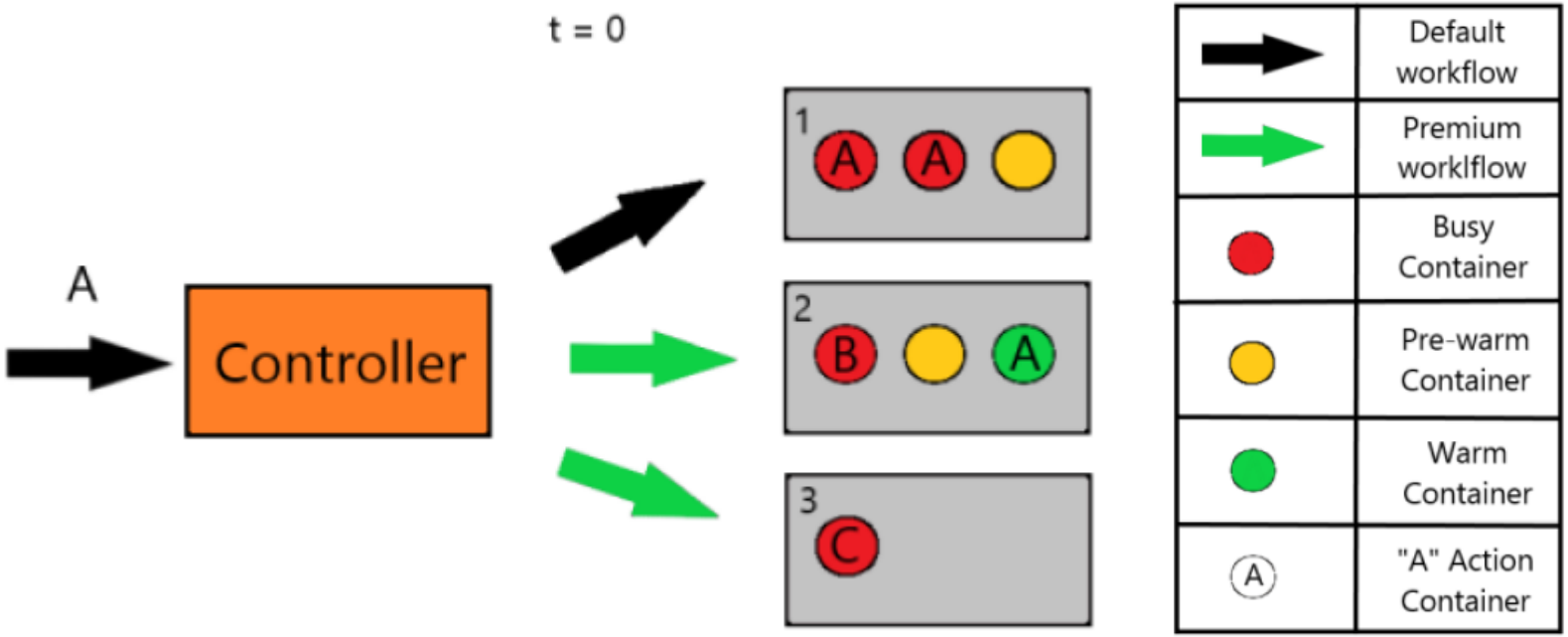


Scheduling Extension *(Action-Spreading)*

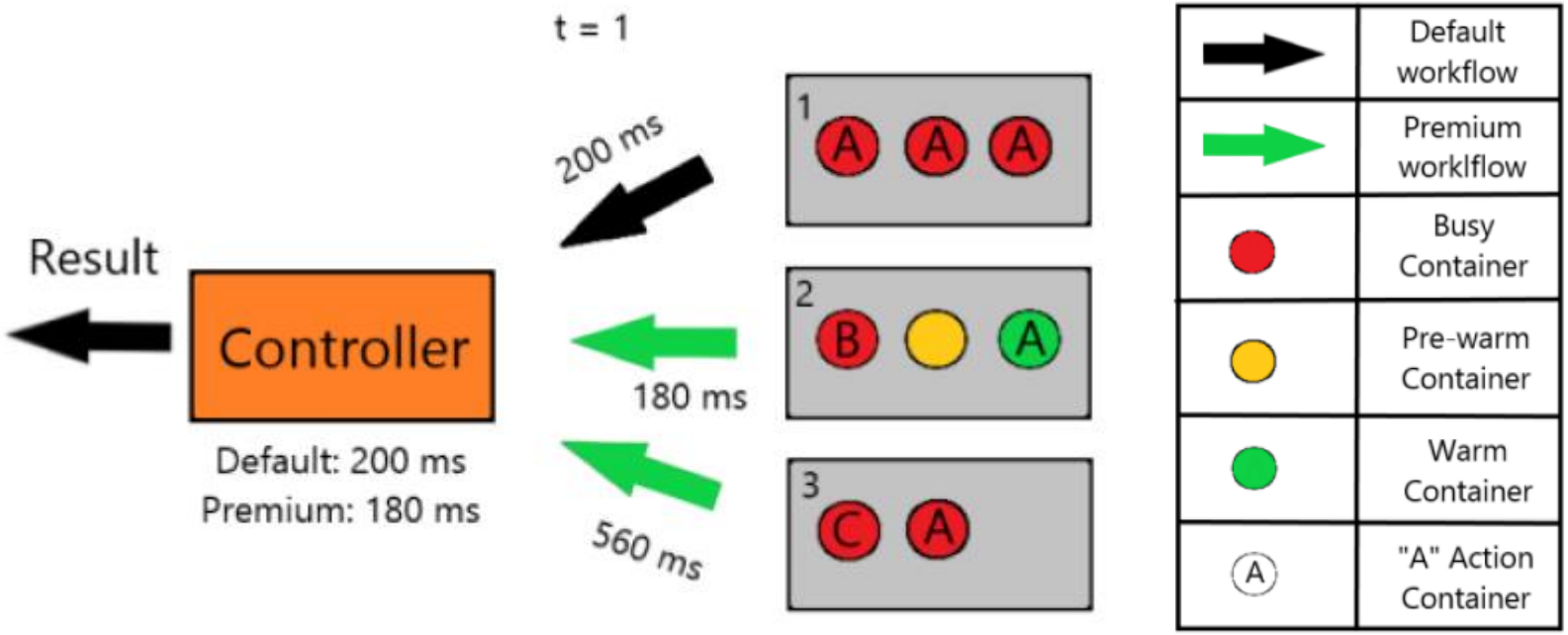
```
Action ← A
ActionContainer ← Action
for all Invokers do
  if BusyPoolSize = MaxPoolSize then
    continue
  else if ActionContainer ∈ FreePool then
    FreePool ← FreePool \ ActionContainer
    BusyPool ← BusyPool ∪ ActionContainer
    return
  else if PreWarmPoolSize > 0  and  Invoker = HomeInvoker then
    PreWarmPool ← PreWarmPool \ PreeWarmContainer
    ActionContainer ← PreWarmContainer
    BusyPool ← BusyPool ∪ ActionContainer
  else if FreePoolSize + BusyPoolSize = MaxPoolSize then
    FreePool ← FreePool \ LeastRecentContainer
  else
    ActionContainer ← ColdContainer
    BusyPool ← BusyPool ∪ ActionContainer
  end if
end for
Queue ← Queue ∪ Action
```

- Over-provisioned system
- An action will *“spread”* throughout the system
- Ignore Pre-warm container when outside of Home Invoker
- Continue searching for more invokers
- Do not stop after finding a viable container

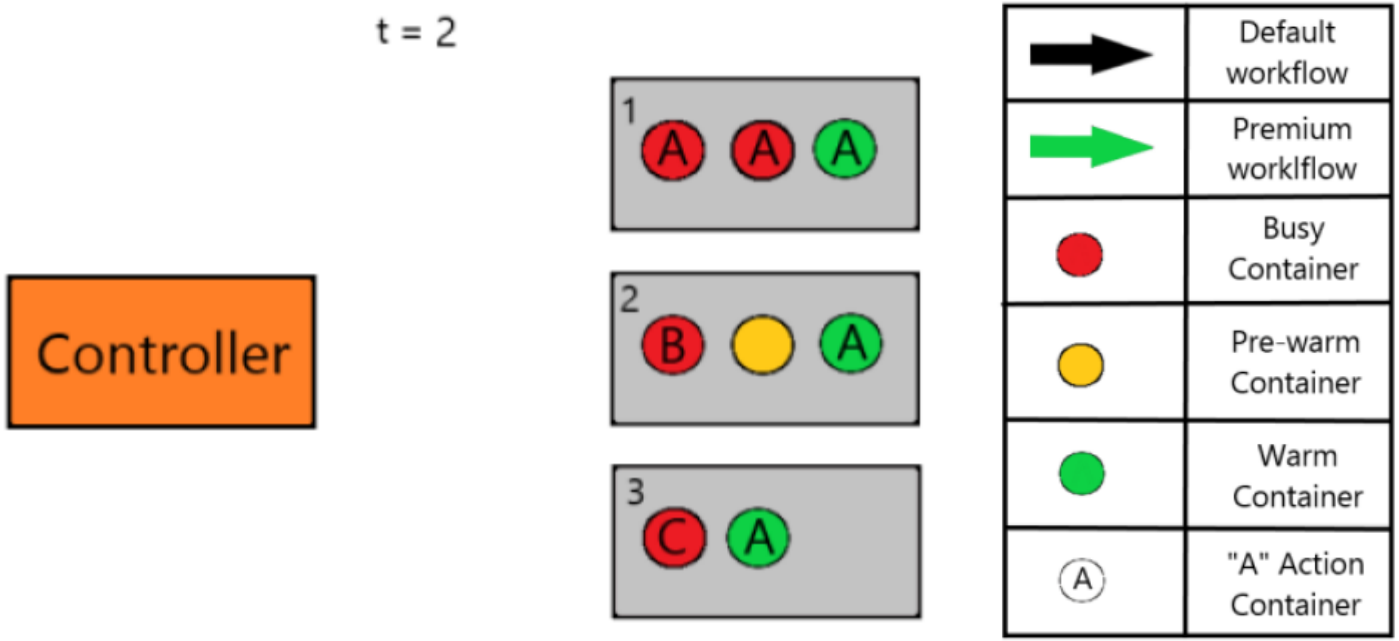
Scheduling Extension *(Action-Spreading)*



Scheduling Extension *(Action-Spreading)*



Scheduling Extension *(Action-Spreading)*



Cost Function *(Action-Spreading)*

- α expresses the ratio of the cost remaining static
- c cost of deployment under default conditions
- C total cost of resources used

$$final\,cost = \alpha \times c + (1 - \alpha) \times C$$

- Implications:
  - Lower α implies higher eagerness, priority, potential cost
  - If all allocated resources used, cost same as default
  - If not all used, partial premium paid on unused resources
  - Clients get better performance at marginal cost
  - Providers able to charge for additional pre-warmed resources

Implementation Details (*Action-Spreading*)

- Action invocation requests must start with “SPREAD_”
- Maximize Scala’s string management functionalities
- Reduces more overhead than adding a new HTTP parameter
- More localized modification to the code

- Updated Message collector
- Allow multiple requests with the same *action_id*
- Collection of requests asynchronously

Evaluation (*Action-Spreading*) Workloads & Metrics

- Workloads
  - Sleep functions (F1)
  - File hashing (F2)
  - Video Transformation (F3)
  - Image classification (F4)
- Metrics
  - Latency
  - Scheduling delay
  - Resource Usage
  - Compared with the Apache OpenWhisk base scheduler

Evaluation (*Action-Spreading*) Sample Testbed Environment

- 3 Invokers
- 1 Controller (enhanced version)
- 1 of each other component

- Two Sub-Environments
  - W (Warm) and C (Cold)
- Two sets of Hardware
  - A (weaker): i7 4-core, 8 threads
    B (stronger): i7 8-core, 18 threads

- Jmeter-based measurements
- 100 user function invocations

- Test 1 (W-A)
  - Scheduler overhead
- Test 2 (C-A) and Test 6 (C-B)
  - Best use case
- Test 3 (W-A) and Test 5 (W-B)
  - Worst use case
- Test 4 (W-A)
  - Parallelism evaluation

- **Take-Away: Approach works better in cold environments and with better hardware available**

Test 1 (Warm-Hardware A)

- Assess worst case scheduler extension internal overhead
  - Base: original OpenWhisk scheduling
  - Default: scheduler extension without Action-Spreading
- Backwards compatibility desired
- Expected reduced overhead when not using functionality
  - actual small increase
  - highly latency-sensitive workload (only sleep F1 function used)

	average latency	median	99% line	variance	extra invoker calls	total time
Default	234	155	2808	479	0	65097
Base	206	134	2538	427	0	64353

Test 2
(Cold-Hardware A)

- F1, F2, and F4 see improved total execution time
  - at the expense of 8 to 9 extra invoker calls
- F1 improves has improvements in all metrics
  - Simpler workload, highly latency sensitive
  - Action-spreading avoids most cold starts
- F2 and F4 mask higher overall average latency
  - collecting the fastest response with extra invoker calls

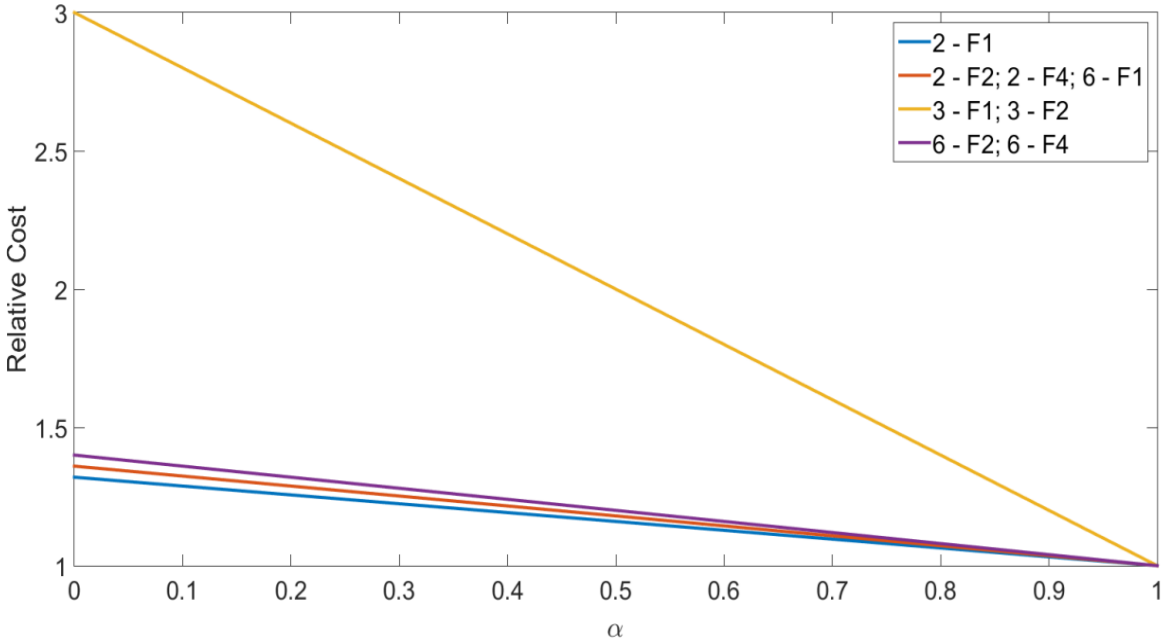
	average latency	median	99% line	variance	extra invoker calls	total time
Base F1	5243	4186	9156	2742	0	24507
Spread F1	2216	655	10379	3377	8	17002
Base F2	14410	12413	24158	6153	0	39009
Spread F2	19821	19524	29214	3802	9	37952
Base F4	29720	29720	41610	10077	0	61223
Spread F4	38330	36376	49497	7653	9	66829

Test 6
(Cold- Hardware B)

- F1, F2, and F4 see improved total execution time
  - at the expense of 9 to 10 extra invoker calls
- F1, F4 improves has improvements in all metrics
  - F1: Simpler workload, highly latency sensitive
  - F4: More resources available in hardware B benefit heavier workload
  - Action-spreading avoids most cold starts
- F2 mask higher overall average latency collecting the fastest response

	average latency	median	99% line	variance	invoker calls	total time
Base F1	1011	236	2366	906	0	17016
Spread F1	607	169	2459	838	9	15236
Base F2	13429	11422	24163	7956	0	38164
Spread F2	18821	17995	29111	3645	10	36346
Base F4	3922	3409	6197	2269	0	21089
Spread F4	3478	2885	5587	2364	10	20560

Evalution:
Utility Function Management



- Seller's modification of α provides an opportunity to mitigate a customer's misuse of the functionality
- The worse the misuse the more control the seller has
- Helps customers to estimate additional cost

Related Work: Scheduling and Pricing

- Scheduling in distributed systems, balancing requests and available resources
  - Cloud, Clusters, Cloud-edge (Fog) [Madej *et al*, 2020]
  - Load balancing, maximizing resource use, energy efficiency, minimizing execution costs
  - Scheduling system for FaaS that is QoS-Aware and implemented in Apache OpenWhisk [Russo *et al*, 2022]
- Pricing strategy is crucial [Al-Roomi *et al*, 2013]
- Difficulties with pricing models in cloud computing [Sharma *et al*, 2021]:
  - Jargon and Architectural Complexity
  - Discrepancy between resource utilization and billing time
  - Accuracy of information lost for quicker response times

Conclusions

- Utility-driven scheduling extension to OpenWhisk
  - “Action-Spreading” approach
- Perfect backwards compatibility
- When used under a cold well provisioned system
  - Latency decrease of up to 2.37 times
  - Maximum of 36% additional cost
- Reduced benefits on already very warm environments
- Positive customer-seller interaction through a utility-inspired function
  - Transparent
  - Single parameter to understand/discuss
  - Provides extra choice to customers and revenue for providers

Future Work

- Further evaluation in grander Kubernetes environments
- Further incorporate priority awareness in invocation queues (Kafka)
- Further testing in other FaaS architectures
- Further study of function estimation cost functions
- Further exploration of parameter ranges competitiveness among clients and providers

Thank you!