RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing

Haodong Tian

April. 19, 2024

0. TLDR

1. Background: Cold-start Latency.

2. Existing Methods: Mitigating cold-starts introduces mem overheads.

Method	Latency (how fast)	Memory (less overhead)
Container sharing	⭐⭐⭐⭐⭐	⭐
Full-cache	⭐⭐⭐	⭐⭐⭐
Partial-cache	⭐	⭐⭐⭐⭐⭐

3. Motivation: Strike a balance between latency and memory, by combining Partial-cache and Container sharing.

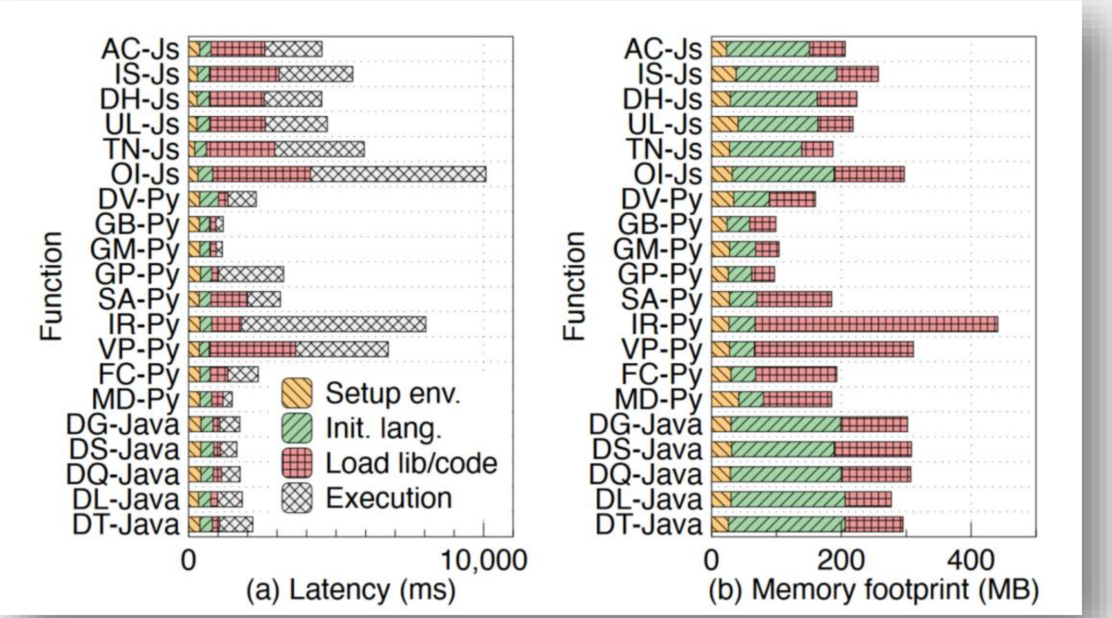
4. Solution (*RainbowCake*): Layer-wise Caching + Carefully designed *Prewarm* and *Keep-alive* policy.

5. Result: SOTA.

1. Background

Cold-starts Breakdown

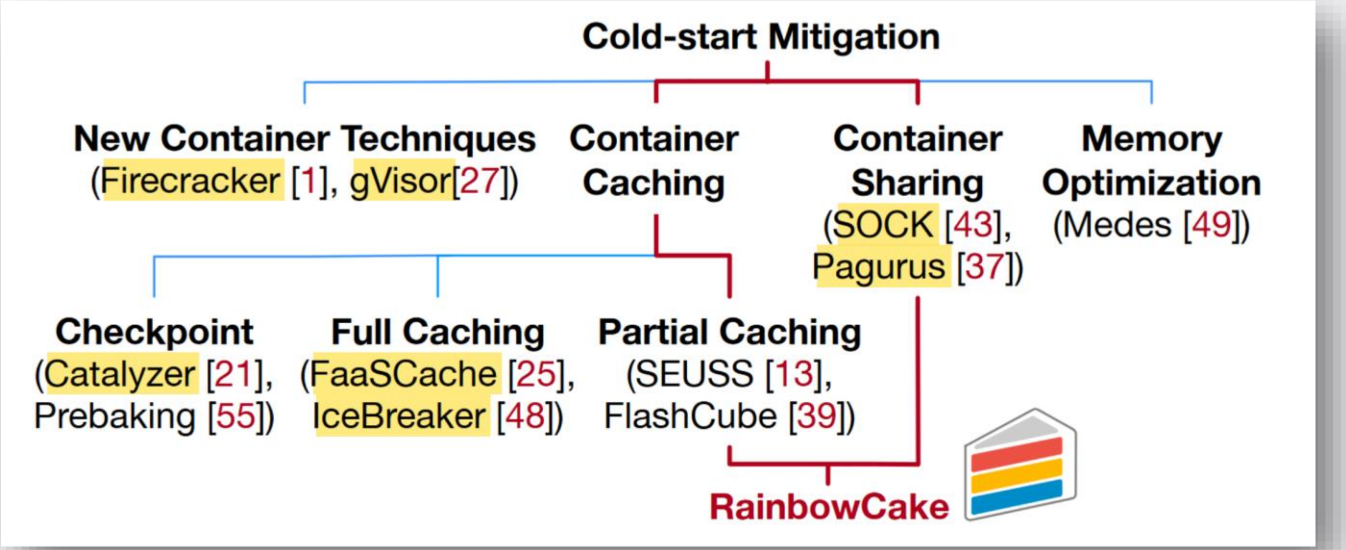
- Stage #1: Environment setup
- Stage #2: Language runtime initialization
- Stage #3: User deployment package loading



2. Existing Methods

Layer-wise Caching ← Partial Caching

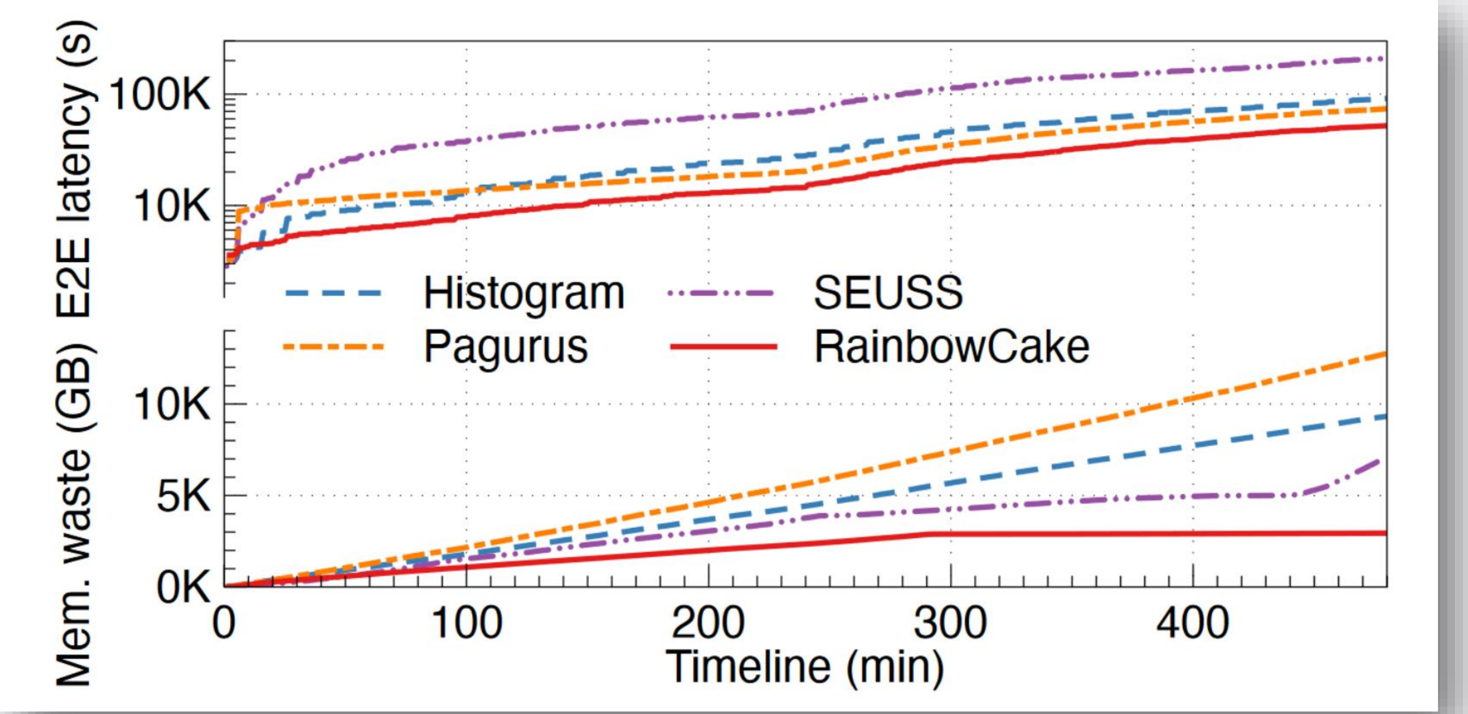
- **Bare** Layer ← Stage #1: Environment setup
- **Lang** Layer ← Stage #2: Language runtime initialization
- **User** Layer ← Stage #3: User deployment package loading



2. Existing Methods

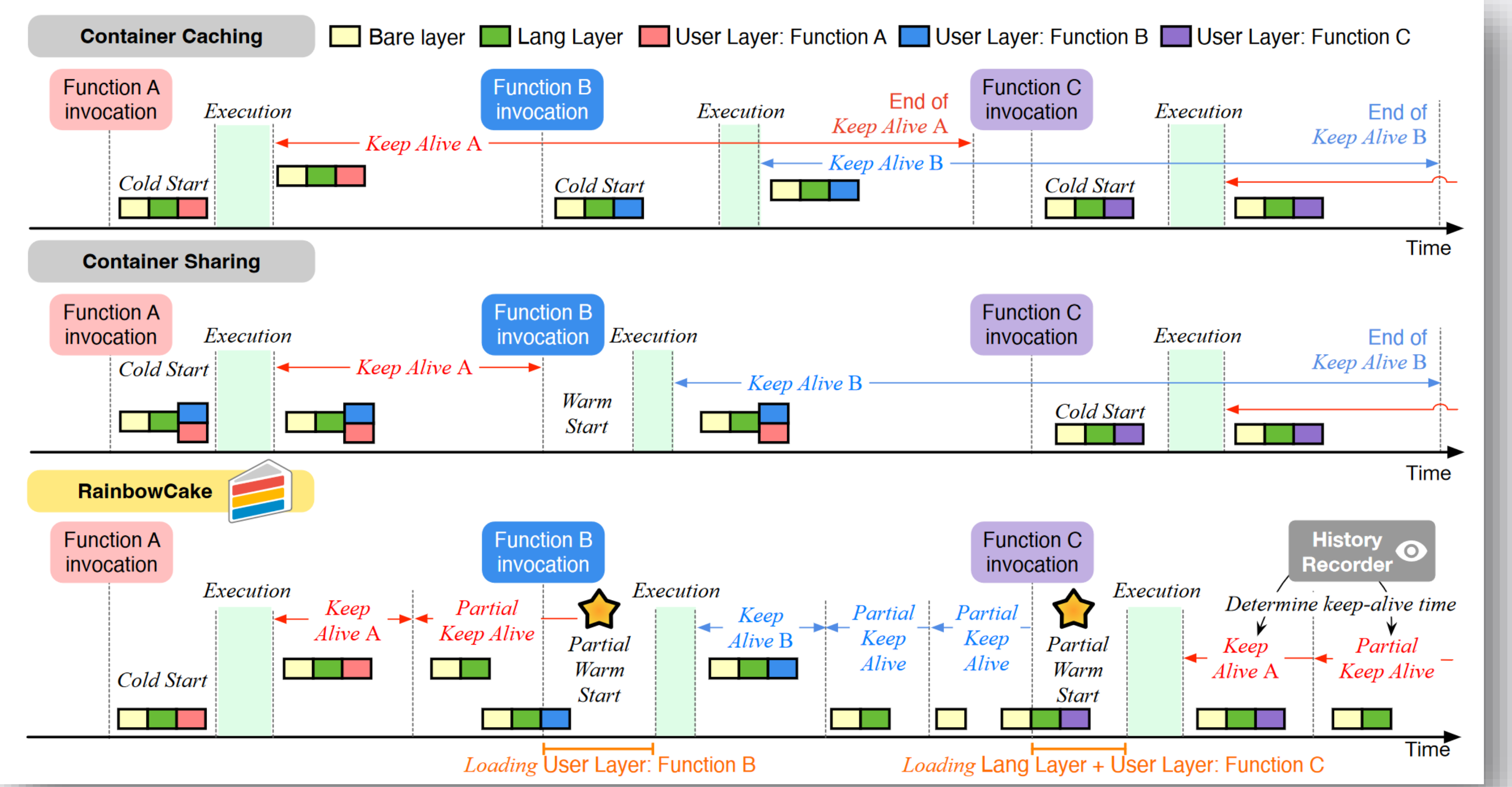
Latency-Mem Trade-offs

Histogram: Full-cache | SEUSS: Partial-cache | Pagurus: Container sharing



3. Solution

Layer-wise Caching Workflow



3. Solution

Intuitive idea

- Upon invocation, start a full container [Bare, Python, PyHelloWorld]
- Determine the TTL for the 3-layer container.
    - How to? **Cover future invocations** & **Balancing latency and mem overhead**.
- Upon timeout, peel of the user layer [Bare, Python]
- Determine the TTL for the 2-layer container.
- ...

3. Solution – Overview

Overview

- **Prewarm policy**
  - When should I prewarm the containers?
  - *How many?
- **Keep-alive policy**
  - How long should I keep the container alive, for specific container layers?
  - [Future invocation coverage] and [Balancing latency and mem overhead].
- ***Invocation policy**
  - Which container should I use?

3. Solution – IAT (Inter-Arrival Time)

IAT (Inter-Arrival Time)

To determine how long should I keep a container alive, I need to estimate when (interval) will the next invocation come.

3. Solution – IAT (Inter-Arrival Time)

Poisson Distribution Modeling

- For each function $f \in F$, $X_f \sim Poisson(\lambda_f)$. ($\lambda_f$ denotes average invocations per second)
- For each function that a layered container can transfer into, e.g., [Bare, Python] can transform into [Bare, Python, PyHelloWorld], [Bare, Python, PyMachineLearning], etc., $Y^{(k)} \sim Poisson(\lambda^{(k)}) = \sum_{f \in F^{(k)}} X_f$, where $\lambda^{(k)} = \sum_{f \in F^{(k)}} \lambda_f$.
- *↑ Poisson Distribution: Invocation pattern*
- *↓ Exponential Distribution: Invocation interval*
- $X^{(k)} \sim Exponential(\lambda^{(k)})$, and $CDF(x; \lambda^{(k)}) = 1 - e^{-\lambda^{(k)}x}, x \geq 0$
- $IAT(k, p) = CDF^{-1}(p; \lambda^{(k)})$

3. Solution – Cost Metrics

Cost Metrics

- Startup overhead: latency
- Wasted resource: mem overhead
- $C = \alpha \times C_{startup} + (1 - \alpha) \times C_{memory}$
- For each function instance, the startup latency and memory footprint is typically constant.
- Average startup latency $\bar{t}^k$, average memory occupation $\bar{m}^k$

3. Solution – Prewarm policy

Prewarm policy

- Schedule a prewarm event after IAT of time.
- If a warm **User** container exists, skip.
- **Each timestamp, at most 1 prewarmed User container for each function instance.*

Algorithm 1: *RainbowCake*'s Pre-warming

```
async def SchedulePrewarm(function_id, IAT):
    Sleep(IAT) /* Wait until next request */
    if Available(function_id) is False then
        /* Pre-warm if no warm ones */
        PrewarmContainer(function_id, type=User)
    else
        /* Skip if warm containers exist */
        pass
    return
while function invocation arrives do
    function_id ← function.get_id()
    next_IAT ← Poisson(function_id, type=User)
    /* Asynchronous execution */
    SchedulePrewarm(function_id, next_IAT)
```

/core/invoker/src/main/scala/org/apache/openwhisk/core/containerpool/ContainerPool.scala

3. Solution – Keep-alive policy

Keep-alive policy

- Upper-bound for TTL:
- $\alpha \times \bar{t}^k = (1 - \alpha) \times \bar{m}^k \beta$
- $TTL = \min(IAT, \beta)$
- **Does IAT fit for multi-prewarmed containers? Multiple User containers downgrade to Lang containers?*

Algorithm 2: *RainbowCake*'s Keep-alive

```
def ComputeTTL(container, IAT):
    t ← container.get_startup_latency()
    m ← container.get_memory_footprint()
    β ← (α × t)/((1 − α) × m) /* Equation 6 */
    return Min(IAT, β)
while container timeouts do
    function_id ← container.get_function_id()
    layer ← container.get_type()
    if layer is Bare then
        /* Bare containers timeout */
        container.kill()
    else
        /* User or Lang containers timeout */
        container.downgrade()
        layer ← container.get_type()
        next_IAT ← Poisson(function_id, layer)
        TTL ← ComputeTTL(container, next_IAT)
        SetContainerTimeout(container, TTL)
```

/core/invoker/src/main/scala/org/apache/openwhisk/core/containerpool/ContainerPool.scala

3. Solution – *Invocation policy

*Invocation policy

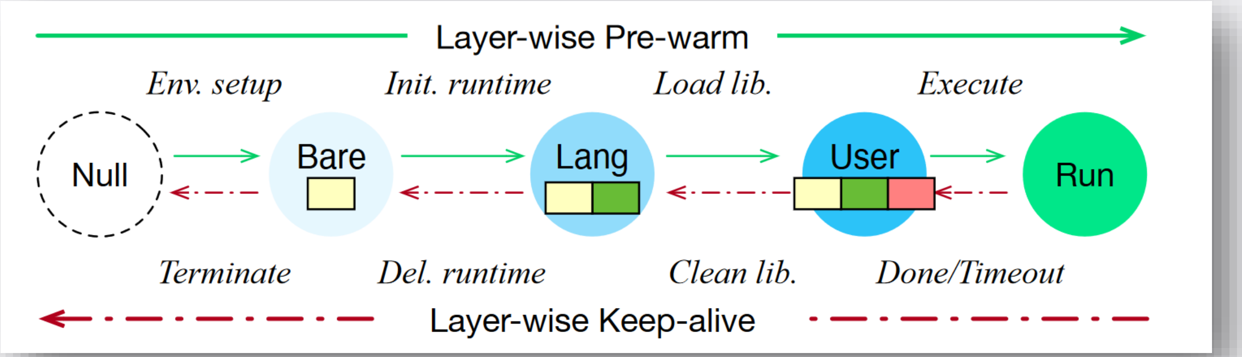
- User > Lang > Bare > Cold
- **Optimal? Preemption?*

/core/invoker/src/main/scala/org/apache/openwhisk/core/containerpool/ContainerPool.scala

3. Solution – Implementation

Implementation

- **OpenWhisk's Container System**
  - Akka Actor Library
  - Finite State Machine (FSM)
- **Layer-wise Policy Implementation**
  - HTTP */clean* handler for invoker containers



4. Result – Experimental Setup

Experimental Setup

- **Workloads**
  - Node.js, Python, Java
- **Invocation traces**
  - Azure traces
- **Baselines**
  - OpenWhisk: fixed 10 minutes keep-alive
  - Histogram: Full-cache, predicting inter-arrival time
  - FaaSCache: Full-cache
  - SEUSS: Partial-cache
  - Pagurus: Container sharing

4. Result – End-to-end latency

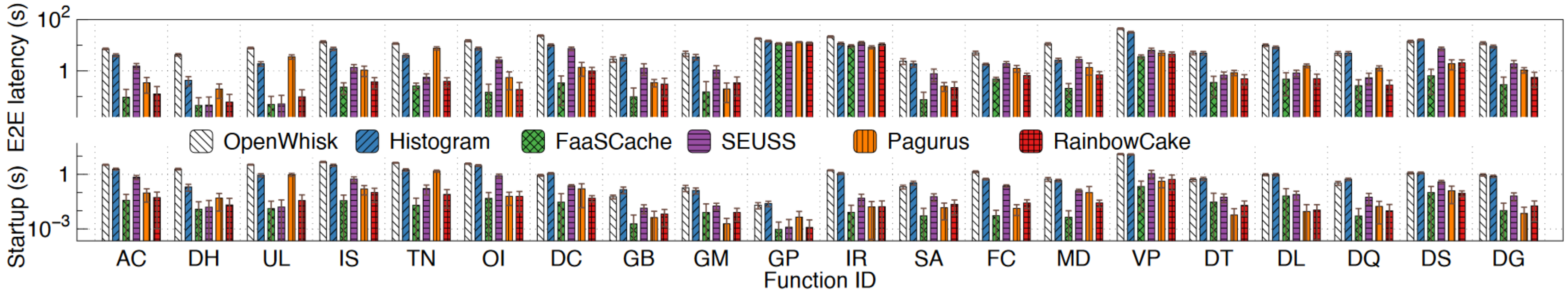


Figure 6. Average function startup and end-to-end latency of six baselines.

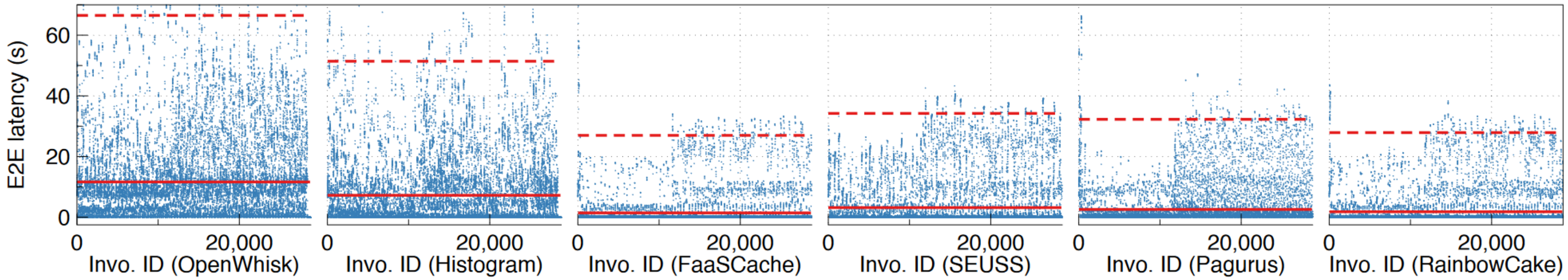


Figure 7. End-to-end latency of each invocation executed by six baselines. Red dash and solid lines represent the 99th percentile and average latency, respectively.

4. Result – Memory Waste

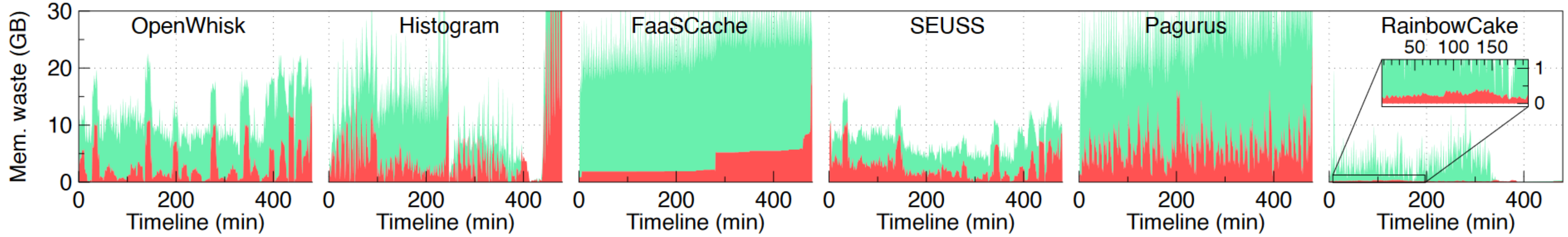


Figure 8. Timeline of wasted memory of six baselines. Green and red shadows represent memory wasted but eventually hit and memory wasted never hit by invocations, respectively.

4. Result – Ablation Study

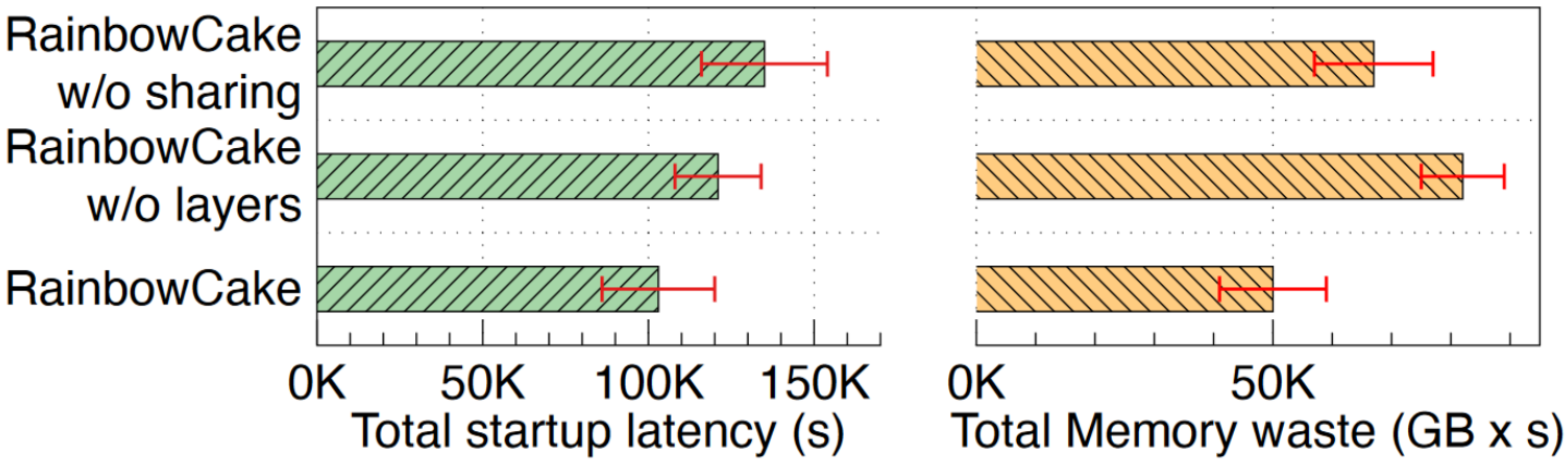


Figure 9. Ablation study of *RainbowCake*.

4. Result – Performance Source Analysis

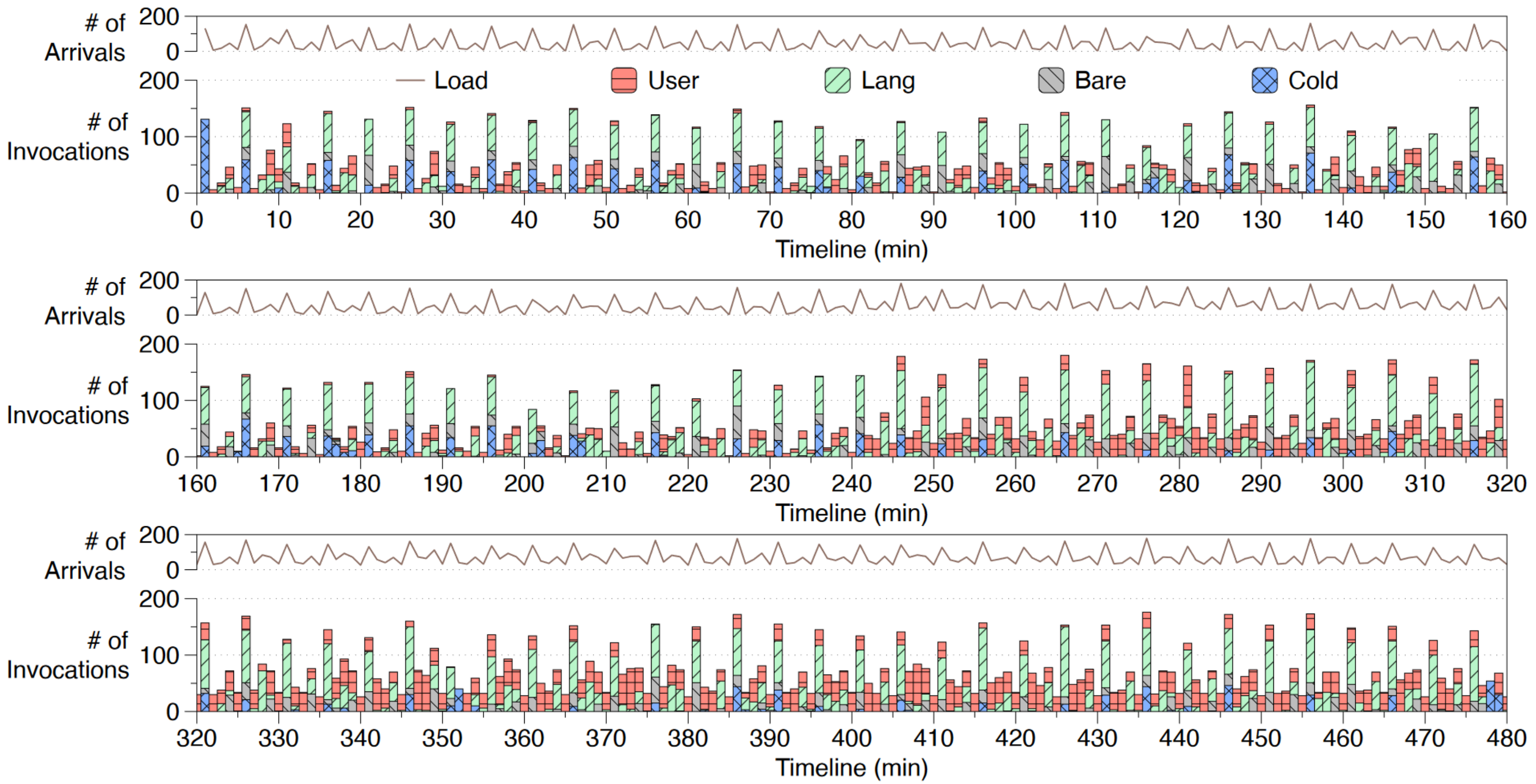


Figure 10. Invocation arrivals and startup timeline of the 8-hour trace set with *RainbowCake*.

4. Result – Sensitivity Analysis

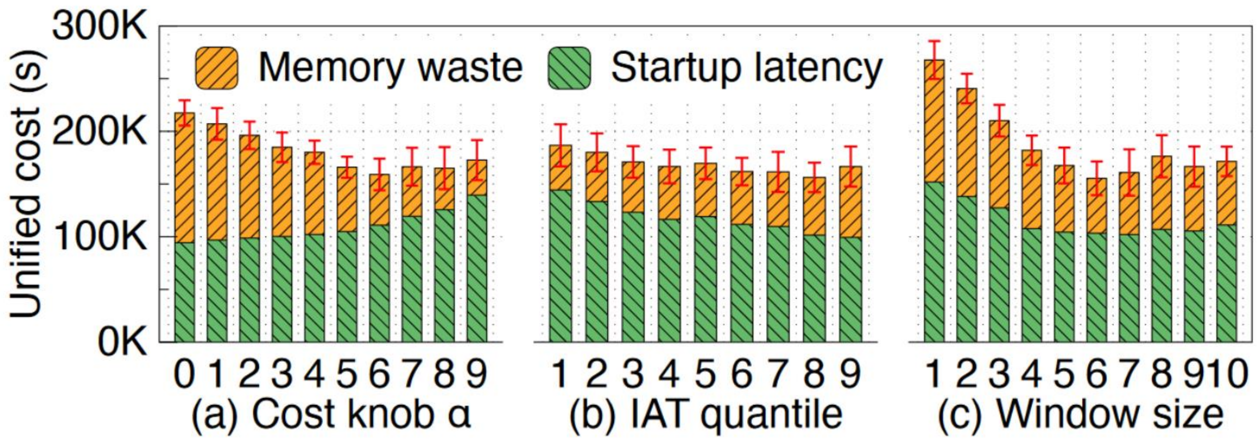
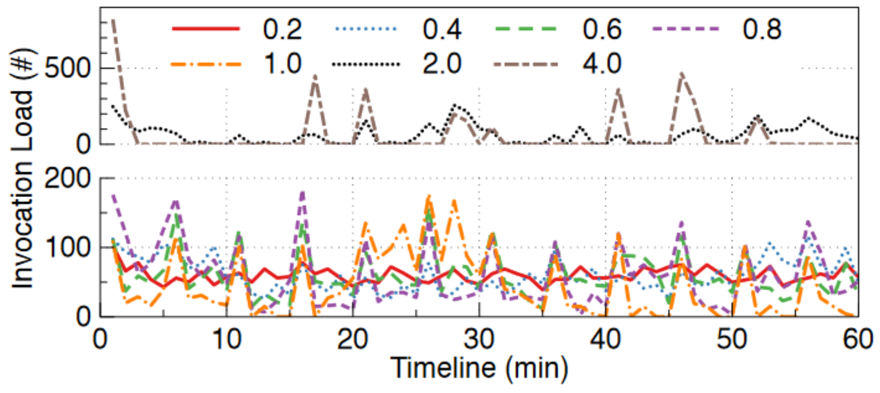


Figure 11. Sensitivity analysis of *RainbowCake*'s total wasting cost and total startup cost for knob parameter α (0.990 to 0.999), IAT quantile p (0.1 to 0.9), and invocation sliding window n (1 to 10).

4. Result – Robustness to Burstiness



(a) Trace timelines of different IAT CVs

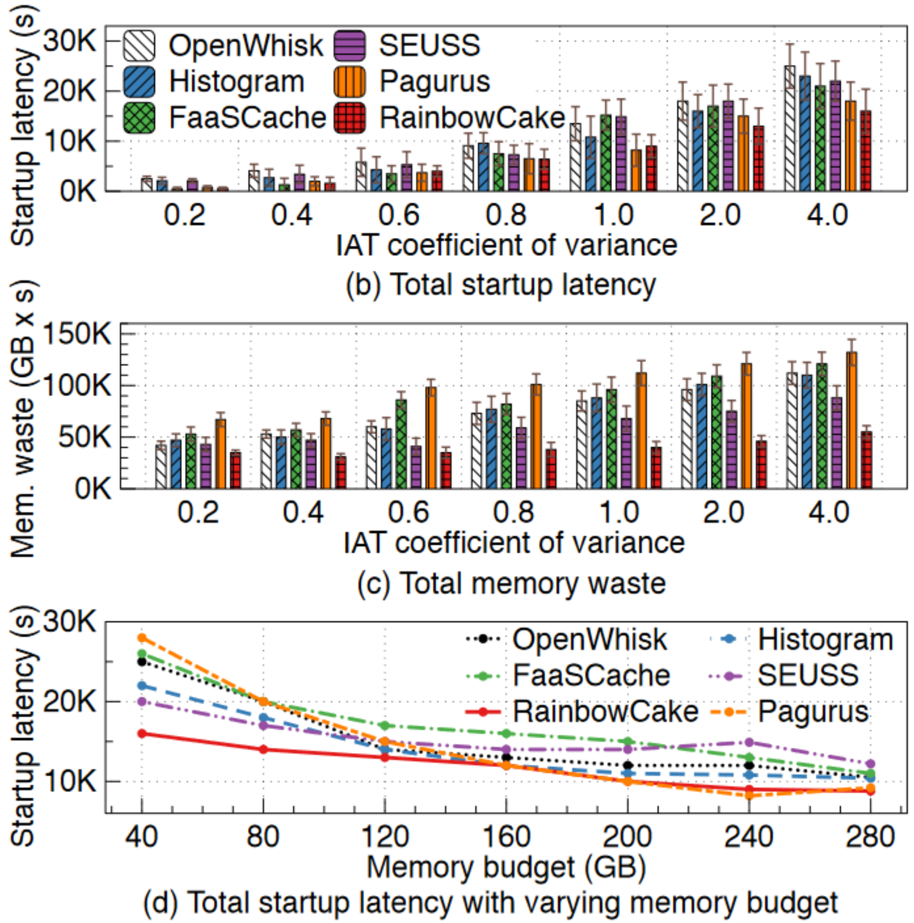


Figure 12. Robustness to burstiness and limited memory budgets of six baselines.

4. Result – Overheads

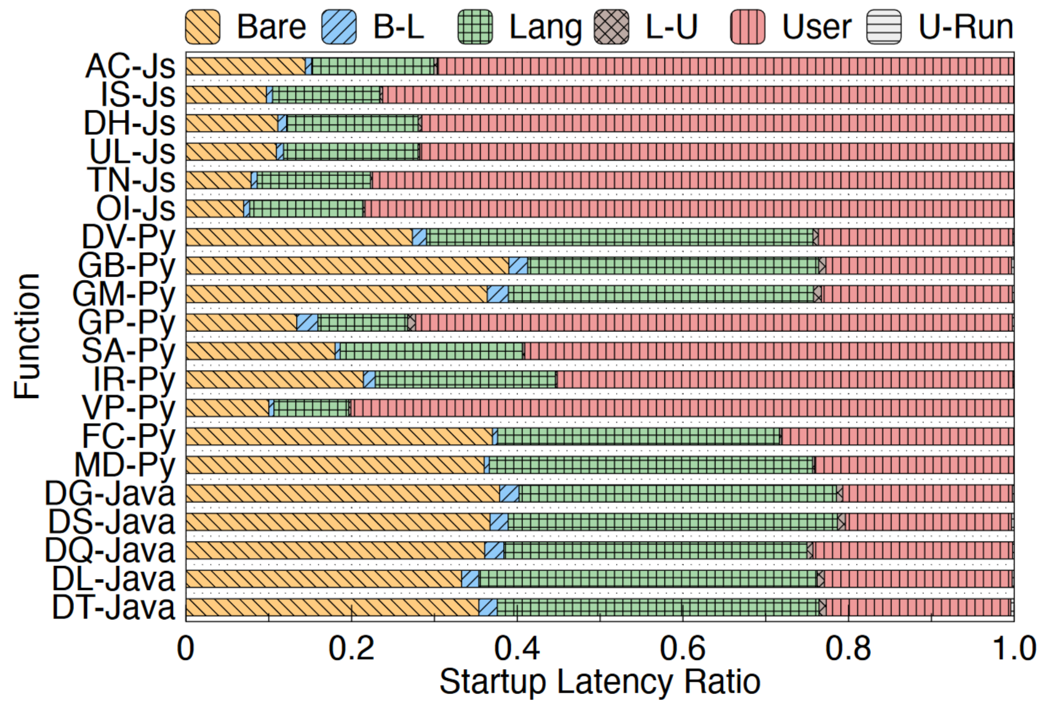


Figure 14. Relative ratio of startup latency breakdown of 20 functions used in the evaluation, including three layers and inter-transition overheads: Bare-to-Lang (B-L), Lang-to-User (L-U), and User-to-Run (U-Run).