Faastlane

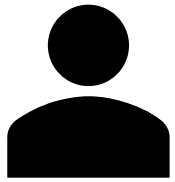
Accelerating Function-as-a-Service Workflows

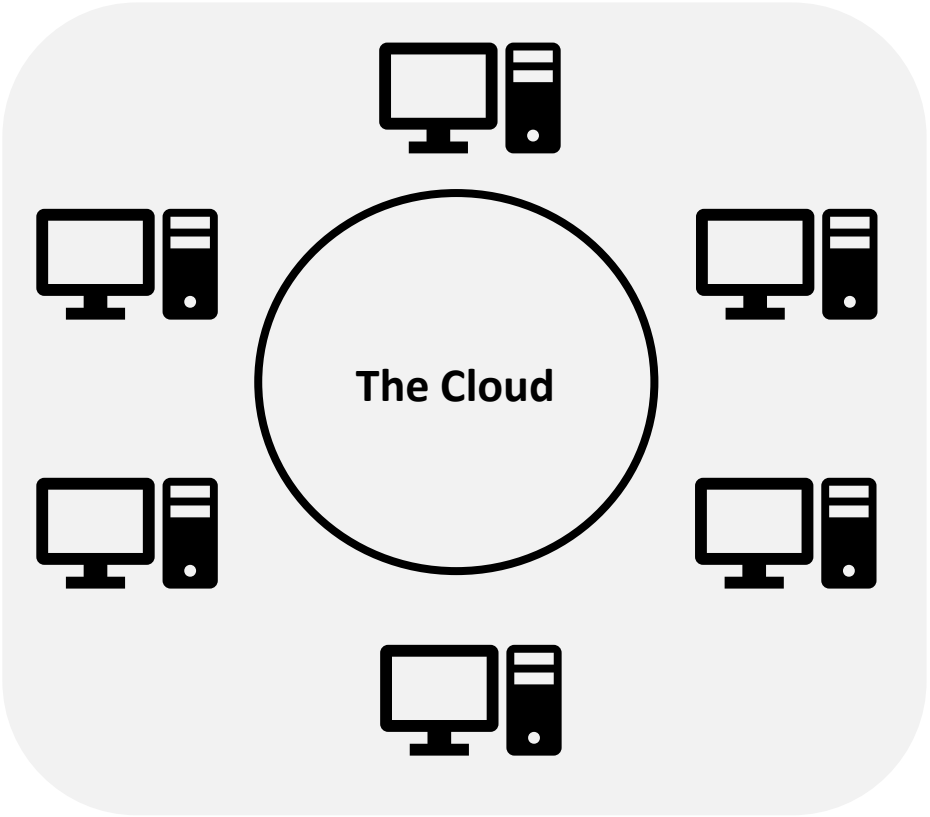
Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, Arkaprava Basu

Overview of Function-as-a-Service (FaaS)

Developer

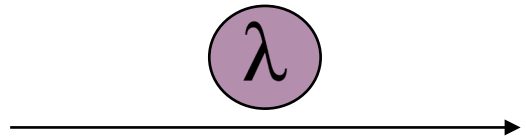
Client(s)

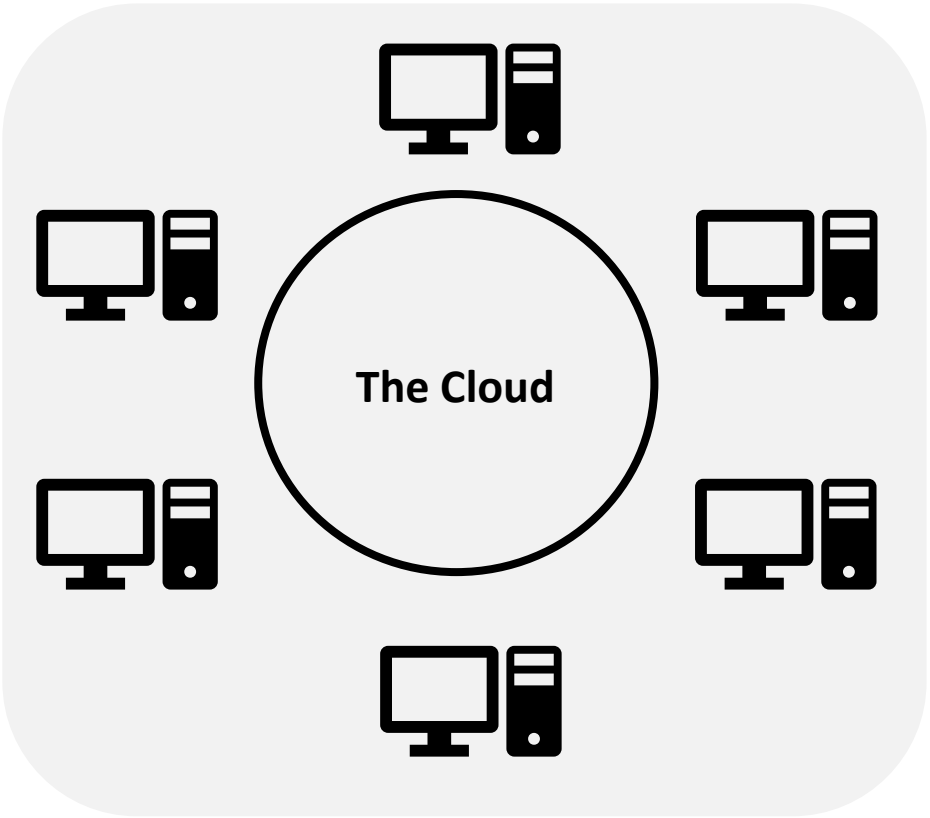




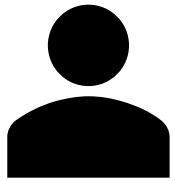
Overview of Function-as-a-Service (FaaS)

Developer



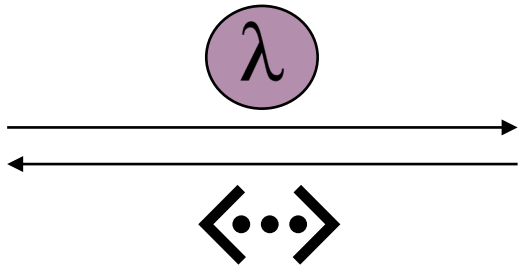


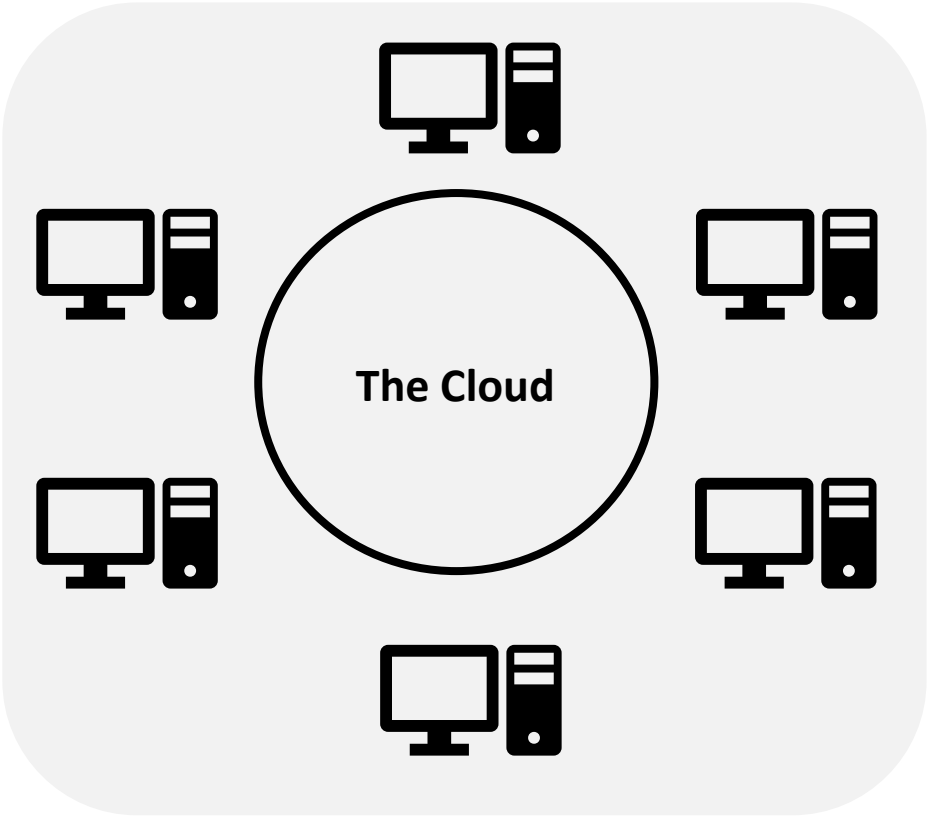
Client(s)



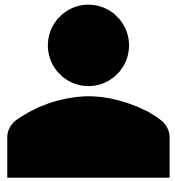
Overview of Function-as-a-Service (FaaS)

Developer

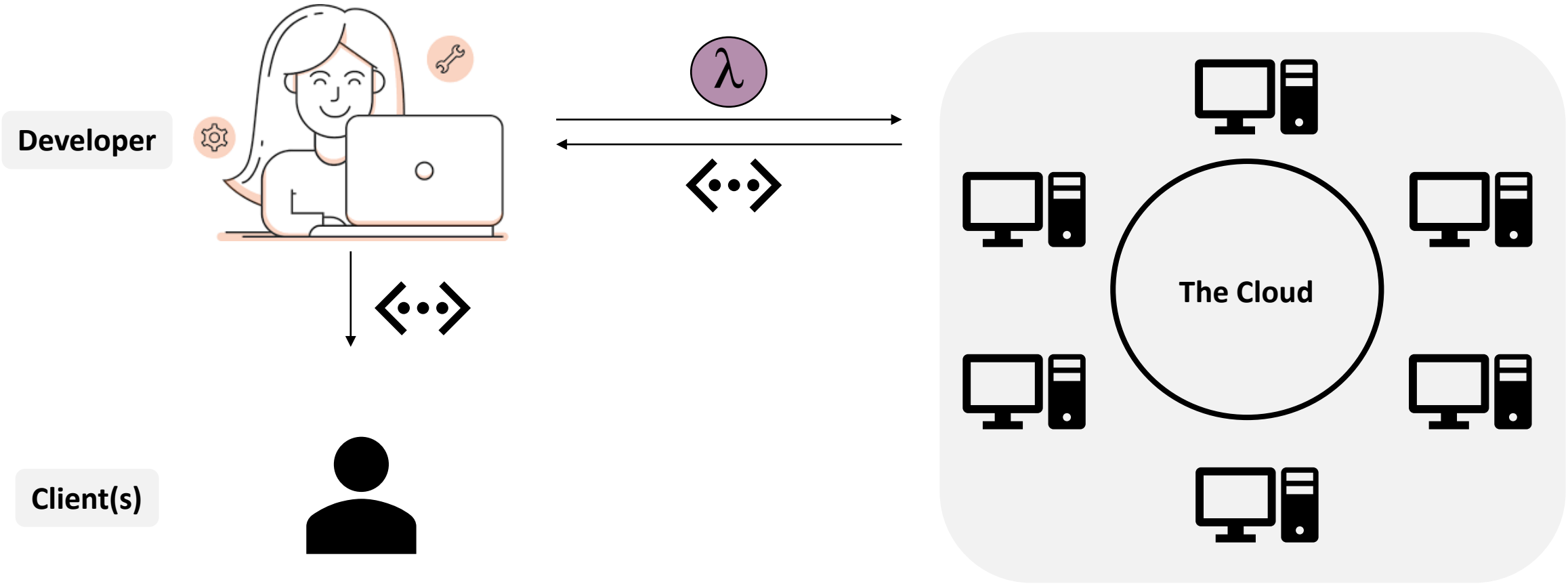




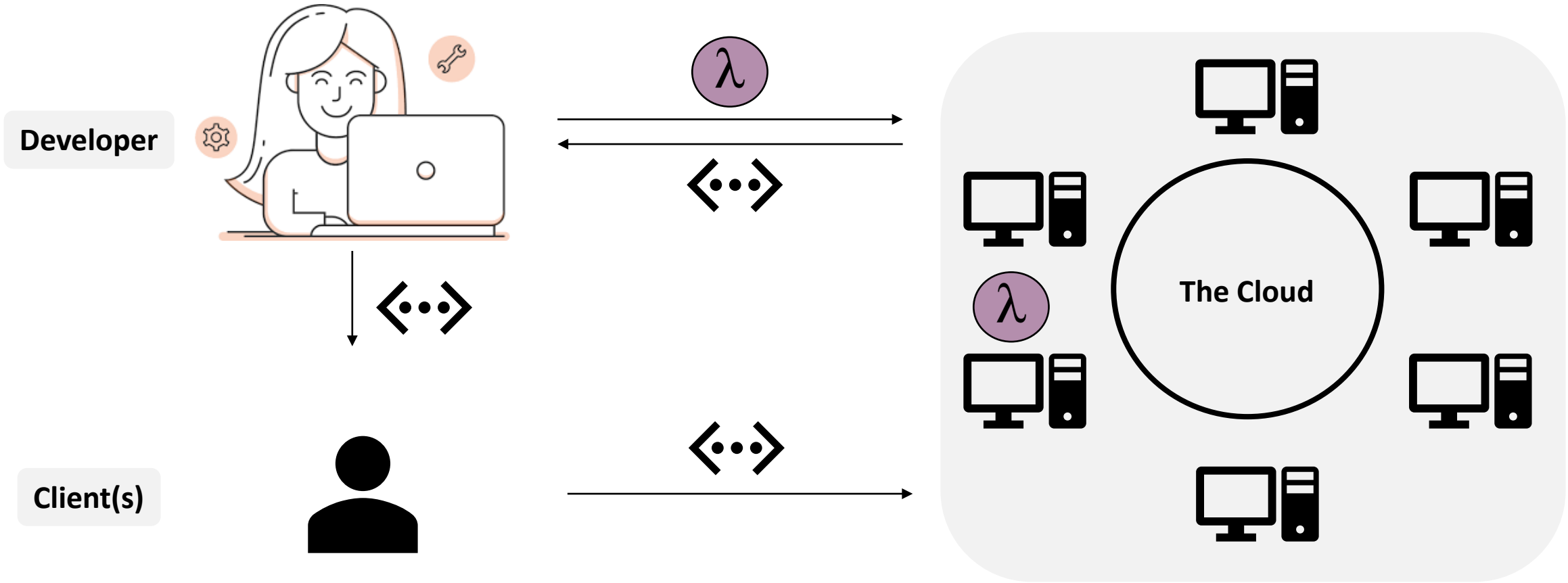
Client(s)



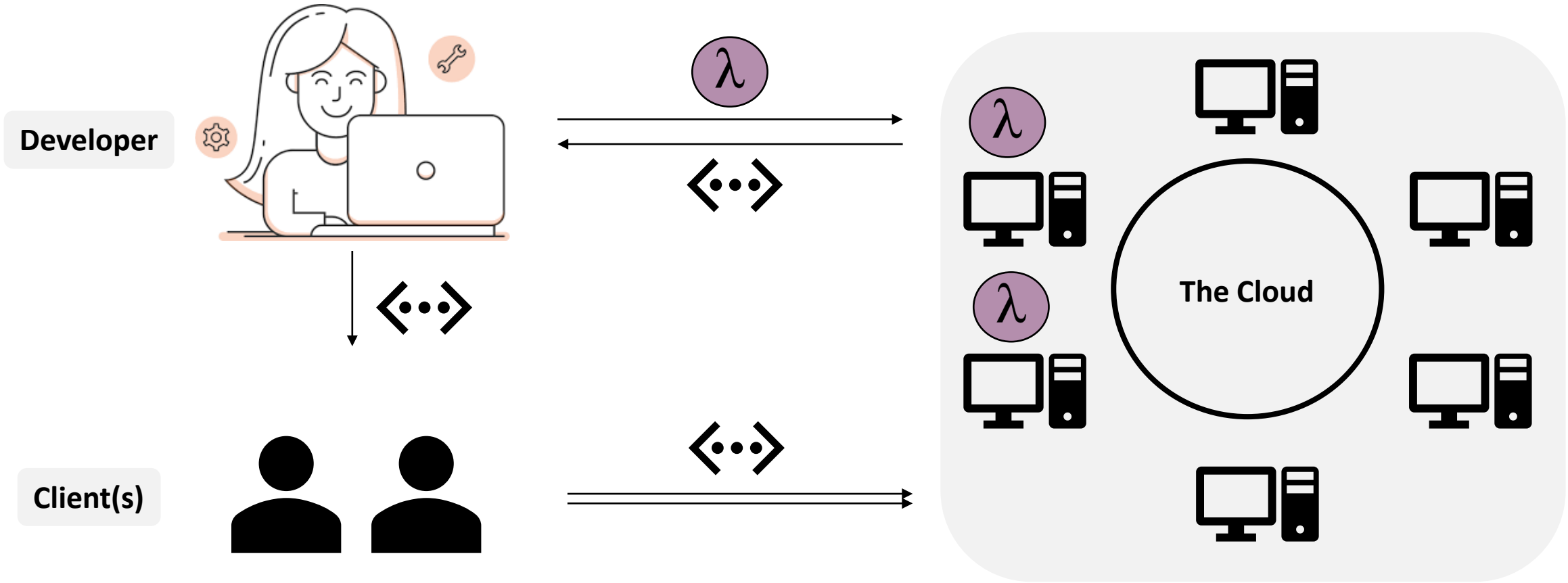
Overview of Function-as-a-Service (FaaS)



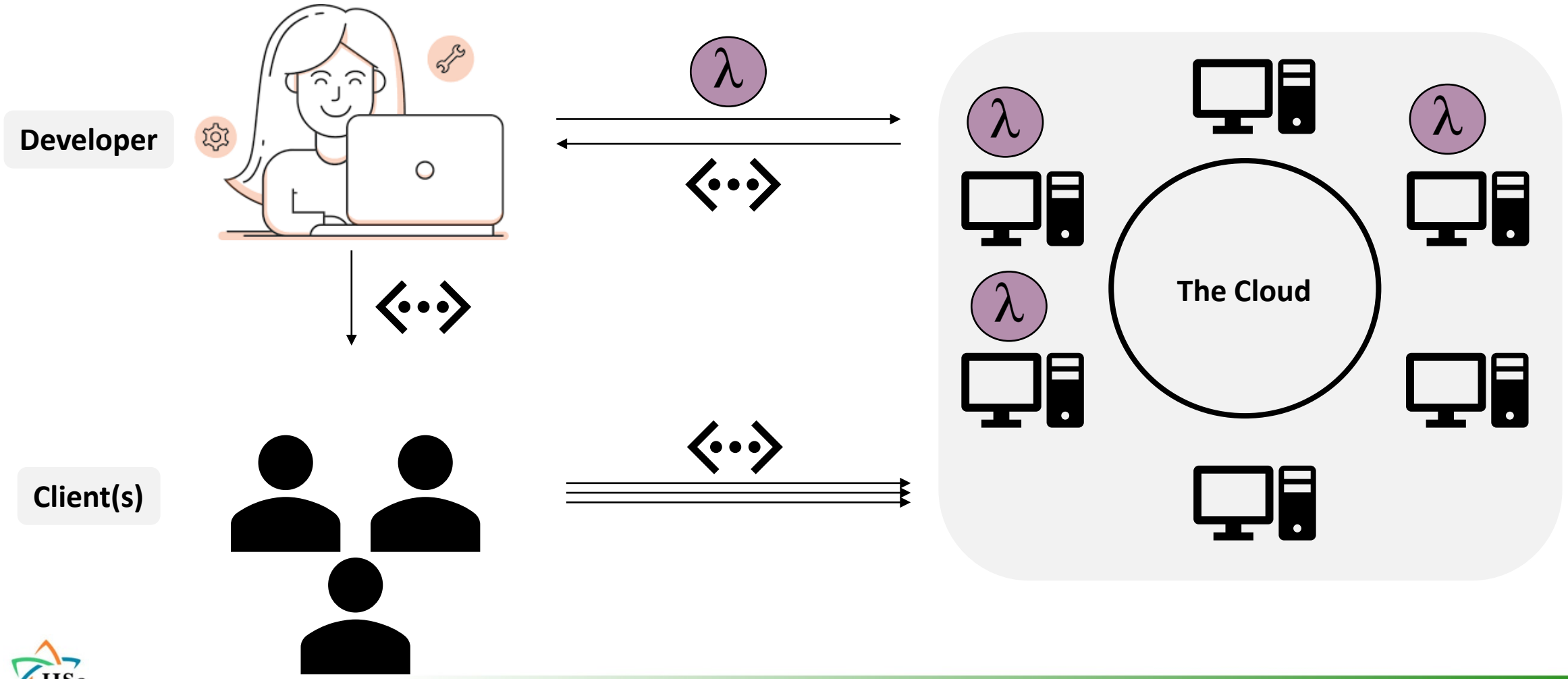
Overview of Function-as-a-Service (FaaS)



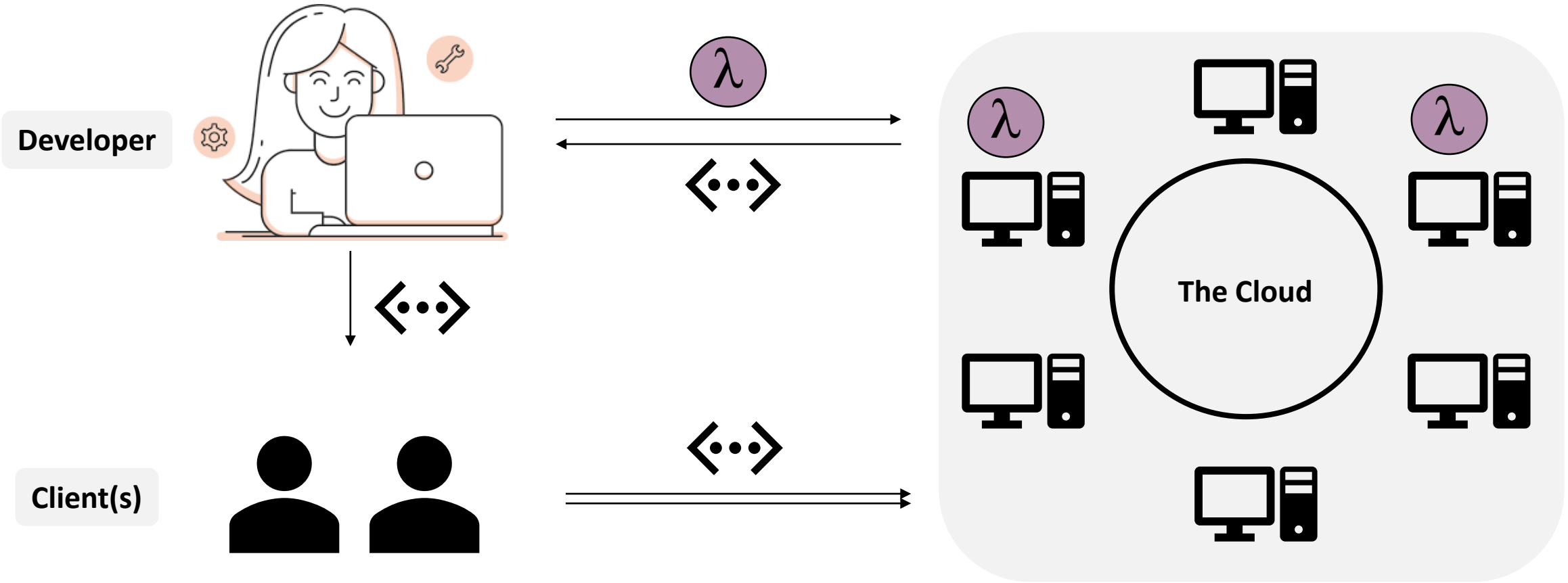
Overview of Function-as-a-Service (FaaS)



Overview of Function-as-a-Service (FaaS)

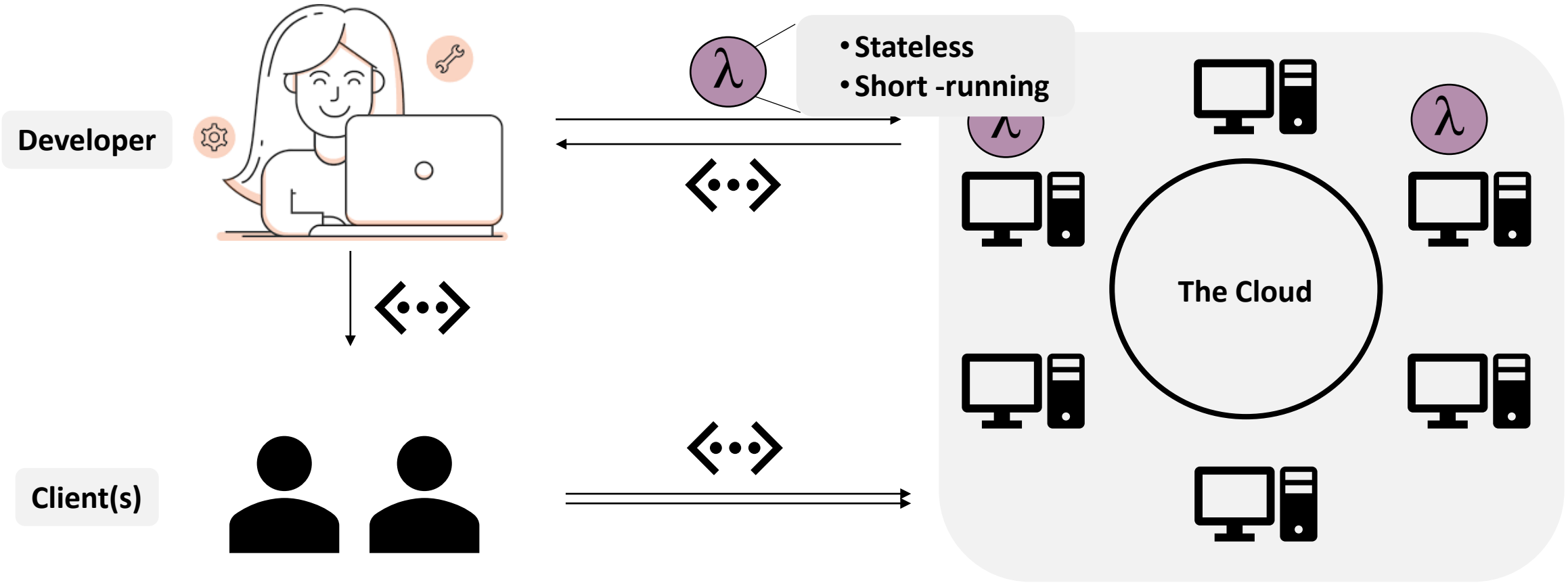


Overview of Function-as-a-Service (FaaS)



Unique FaaS features: Autoscaling, pay-per-use billing

Overview of Function-as-a-Service (FaaS)

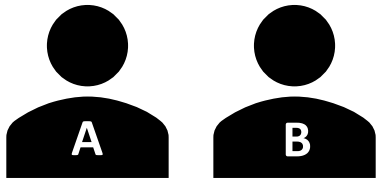


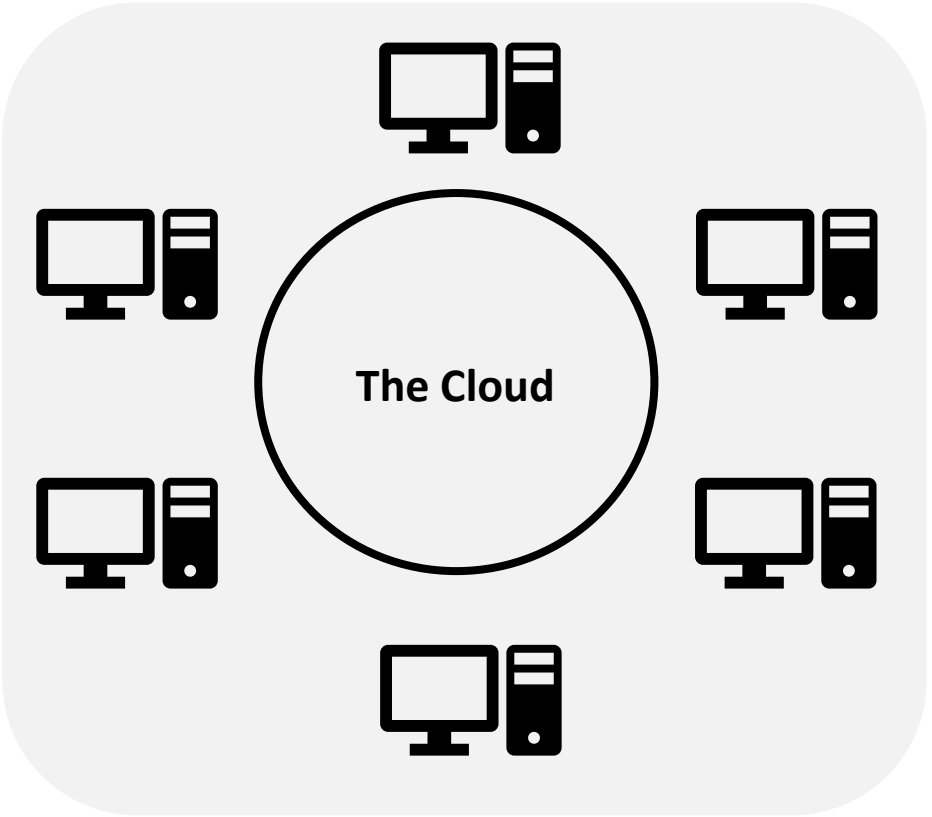
Unique FaaS features: Autoscaling, pay-per-use billing

FaaS workflows: Orchestrating multiple functions

Developer

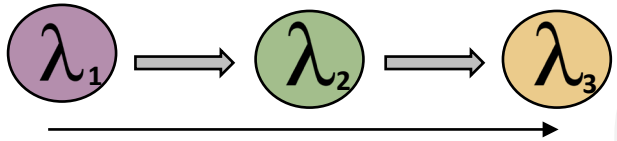
Clients

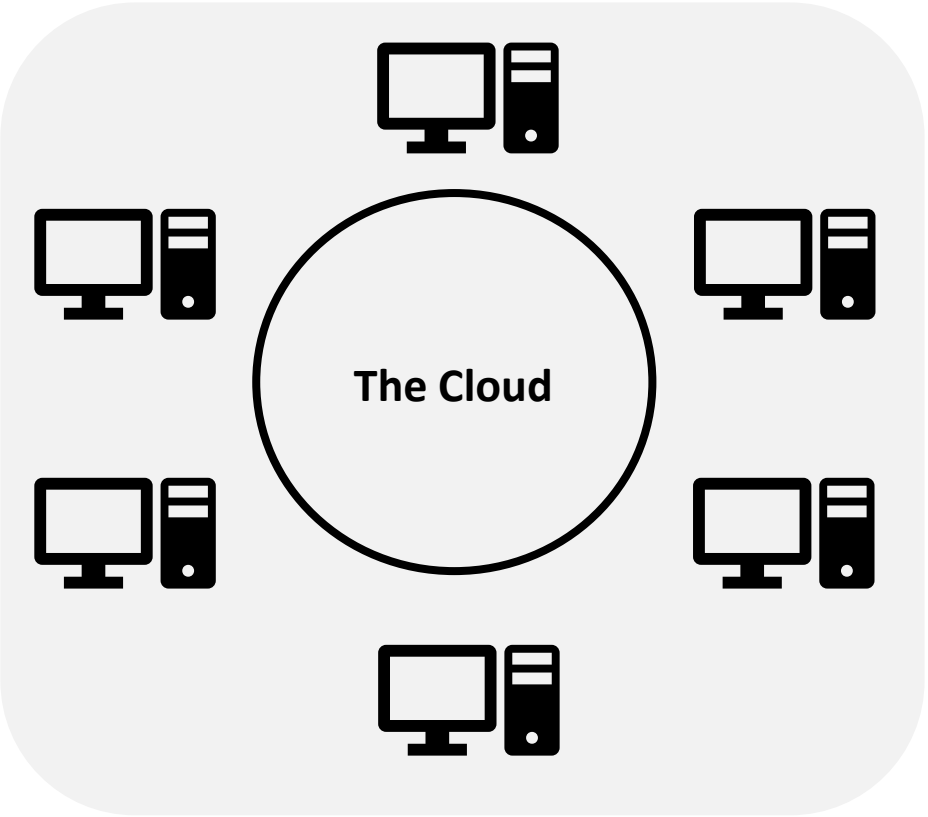




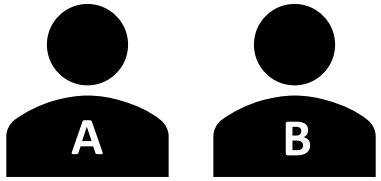
FaaS workflows: Orchestrating multiple functions

Developer



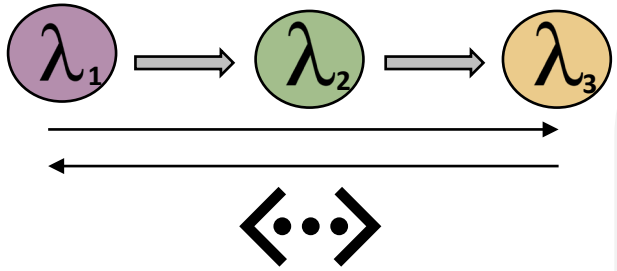


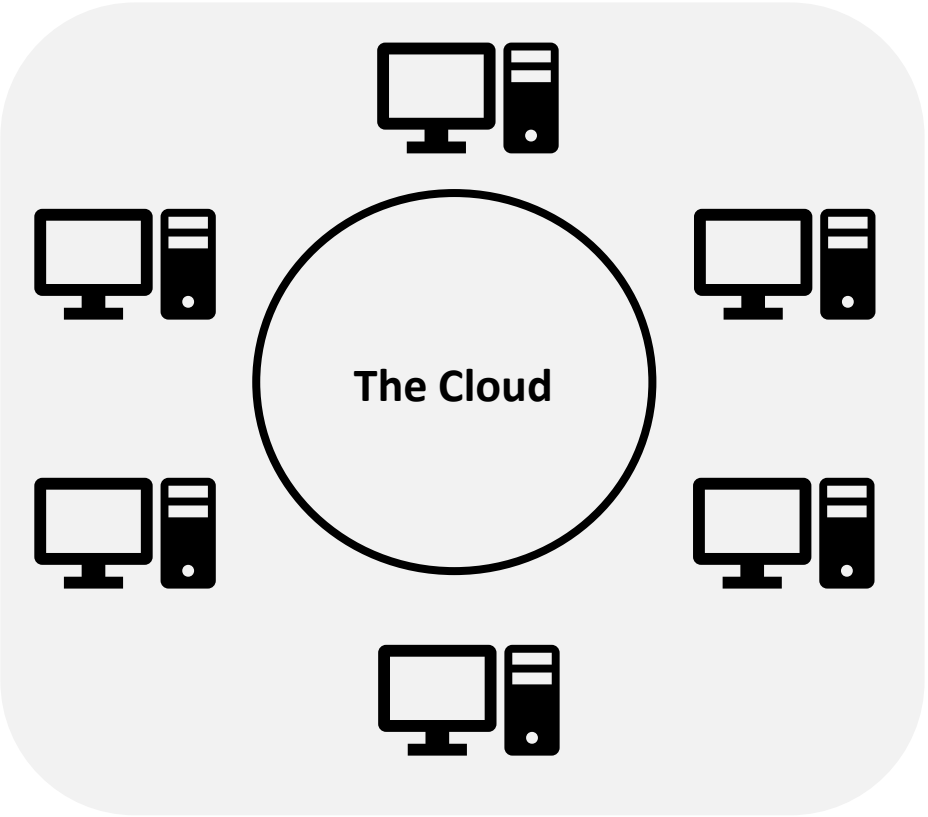
Clients



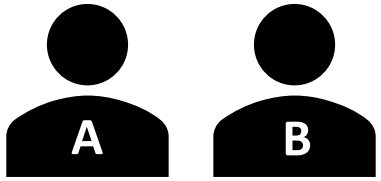
FaaS workflows: Orchestrating multiple functions

Developer

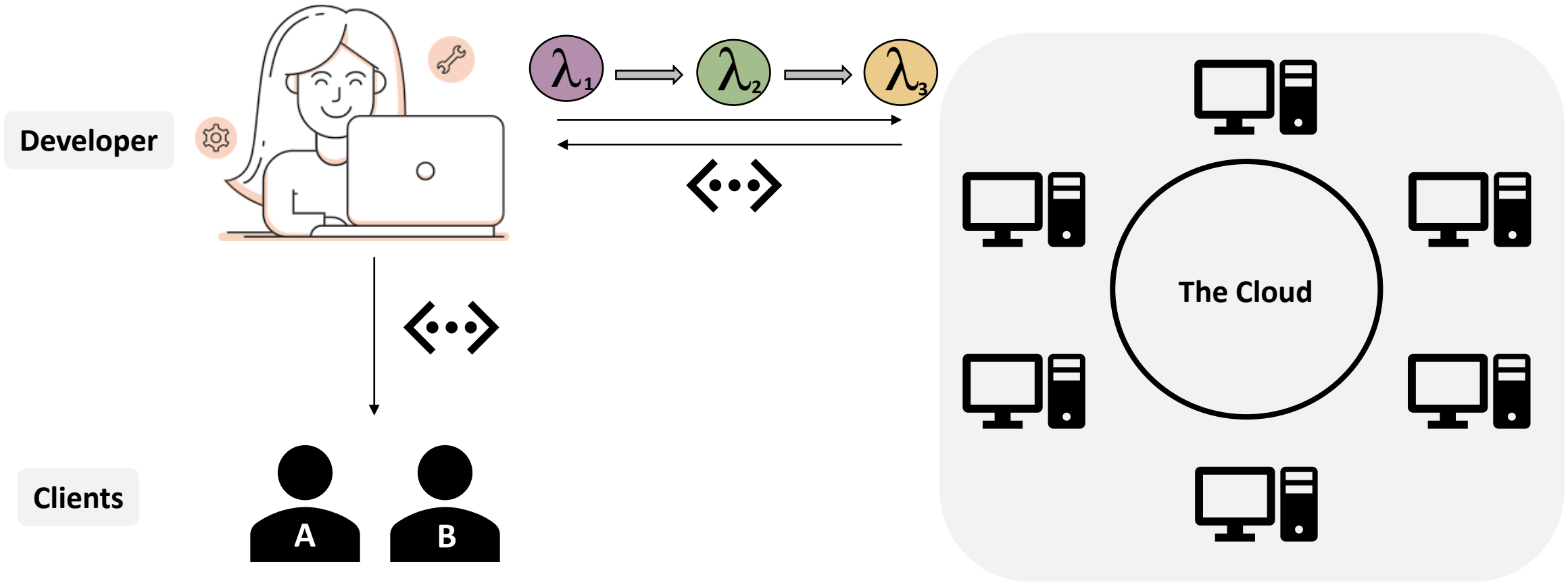




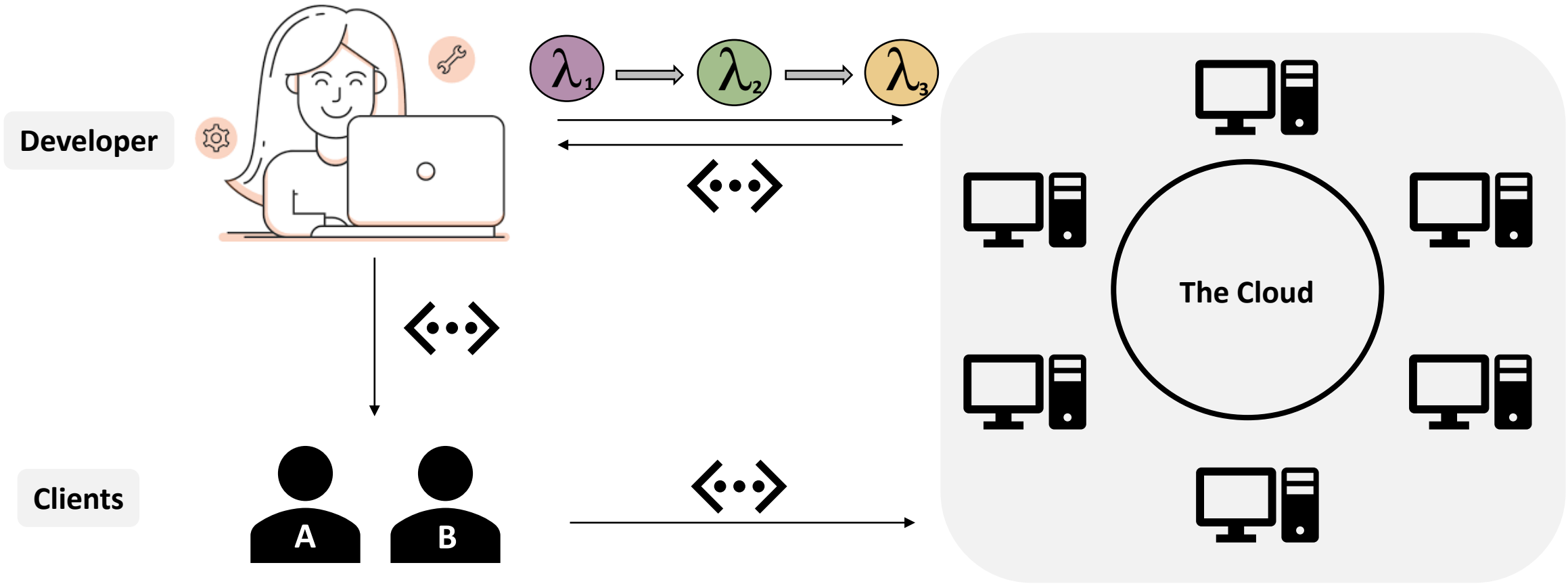
Clients



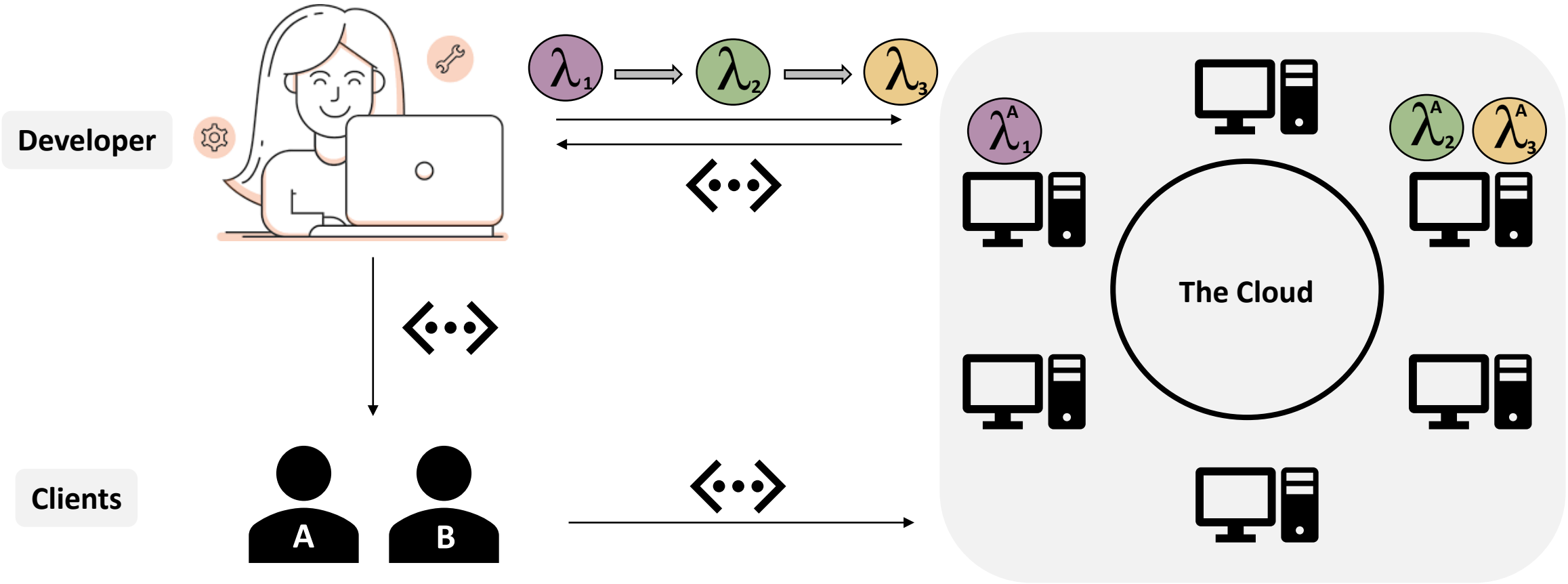
FaaS workflows: Orchestrating multiple functions



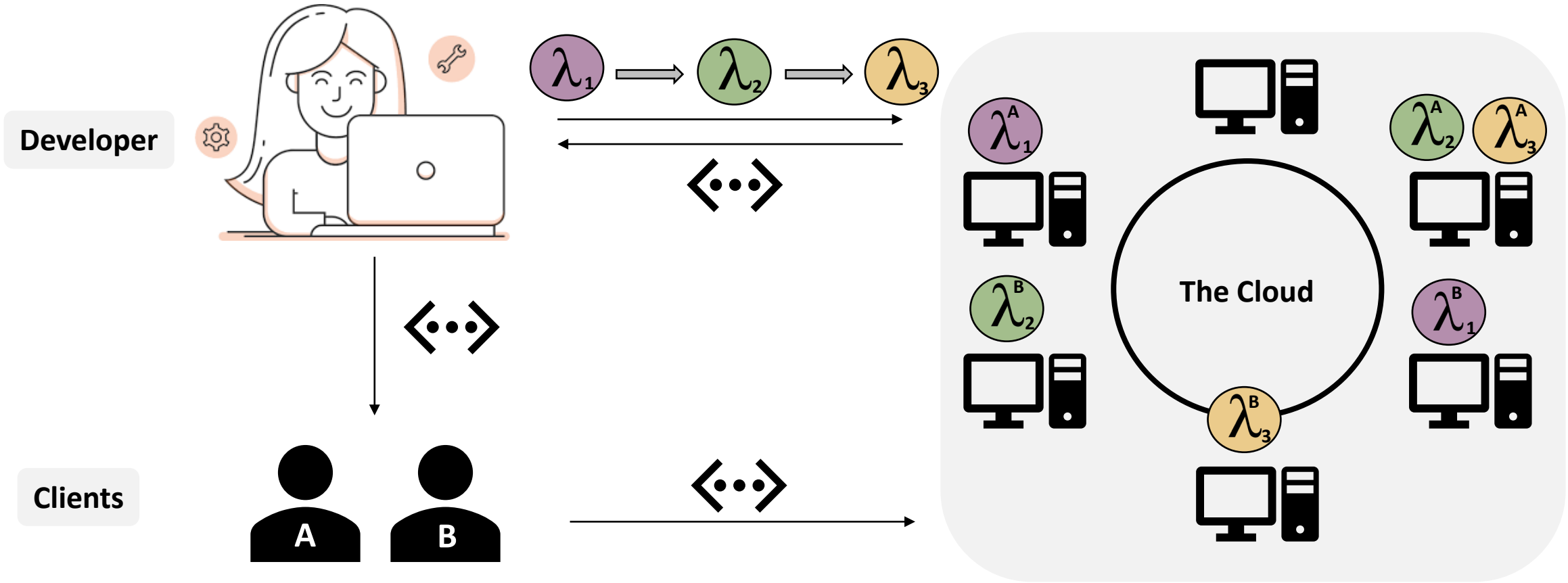
FaaS workflows: Orchestrating multiple functions



FaaS workflows: Orchestrating multiple functions

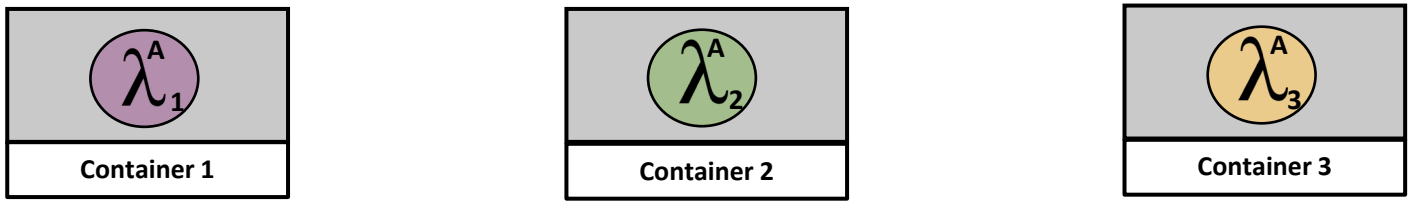


FaaS workflows: Orchestrating multiple functions



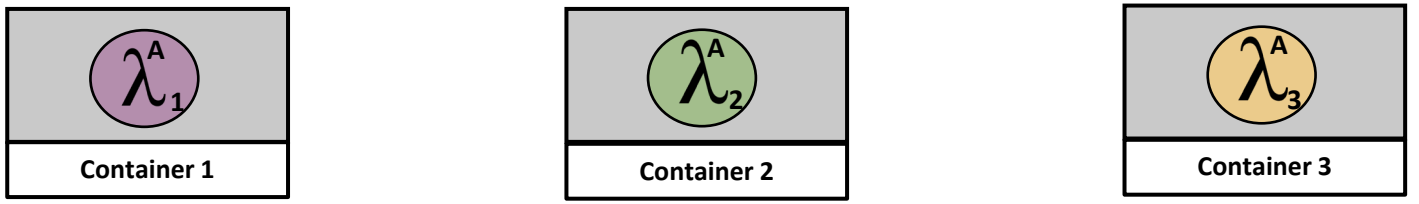
Function interactions in a workflow instance

Current commercial FaaS platforms



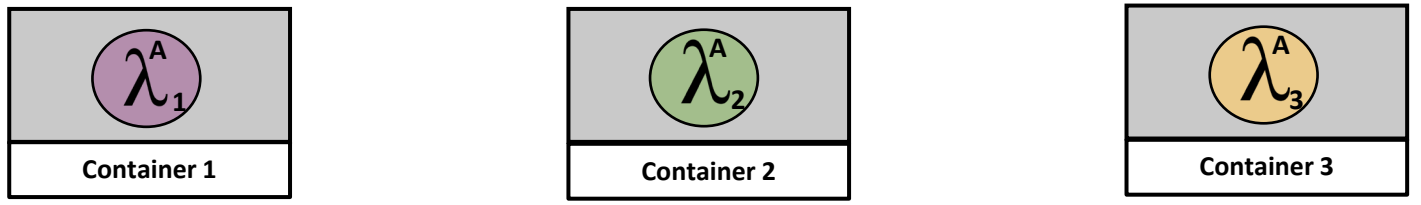
Function interactions in a workflow instance

Current commercial FaaS platforms



Function interactions in a workflow instance

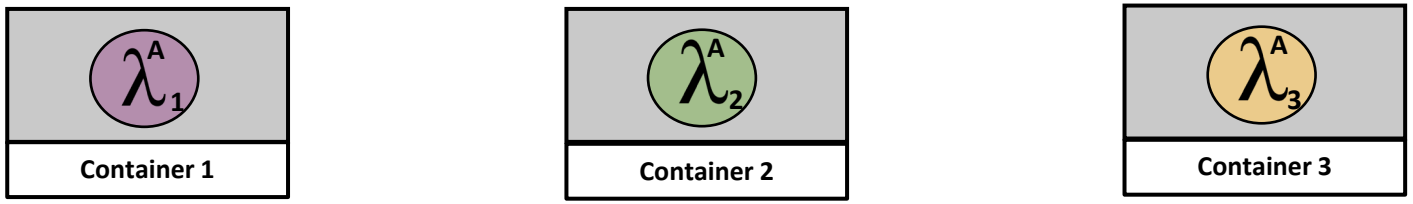
Current commercial FaaS platforms



Function interaction latency can be up to 96% of total execution time!

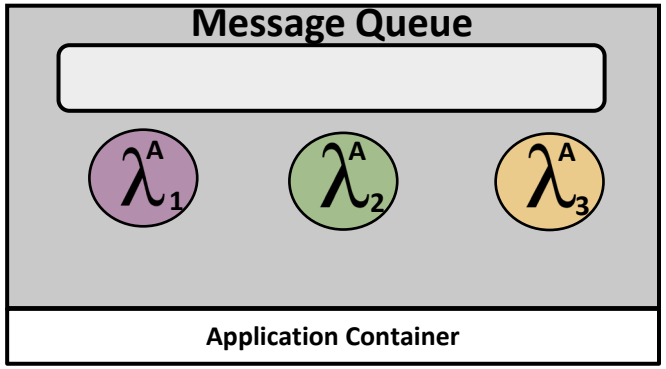
Function interactions in a workflow instance

Current commercial FaaS platforms



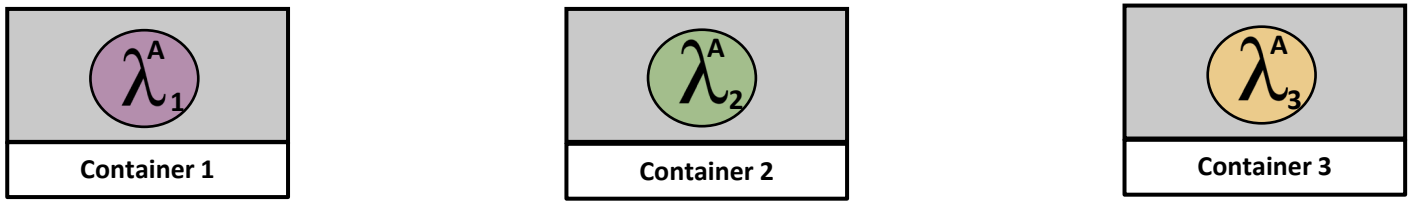
Function interaction latency can be up to 96% of total execution time!

SAND (Akkus et al., ATC '18)



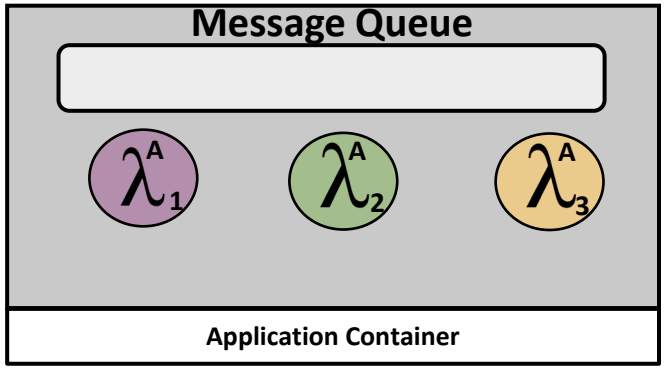
Function interactions in a workflow instance

Current commercial FaaS platforms

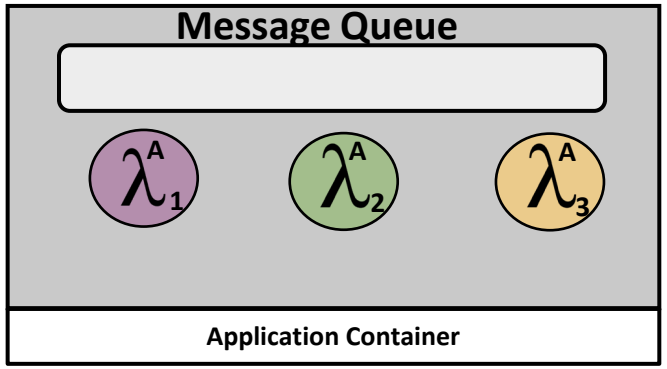


Function interaction latency can be up to 96% of total execution time!

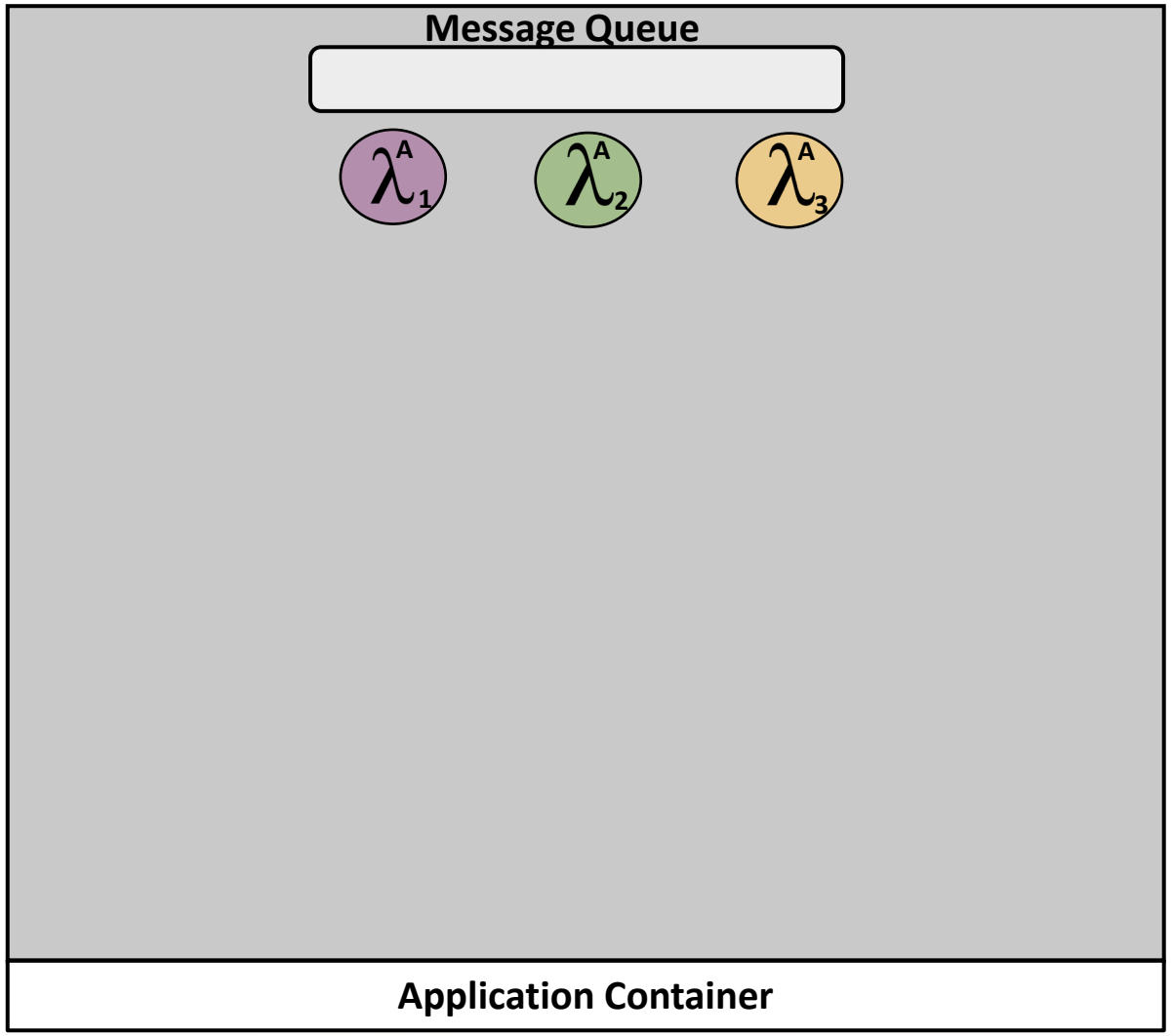
SAND (Akkus et al., ATC '18)



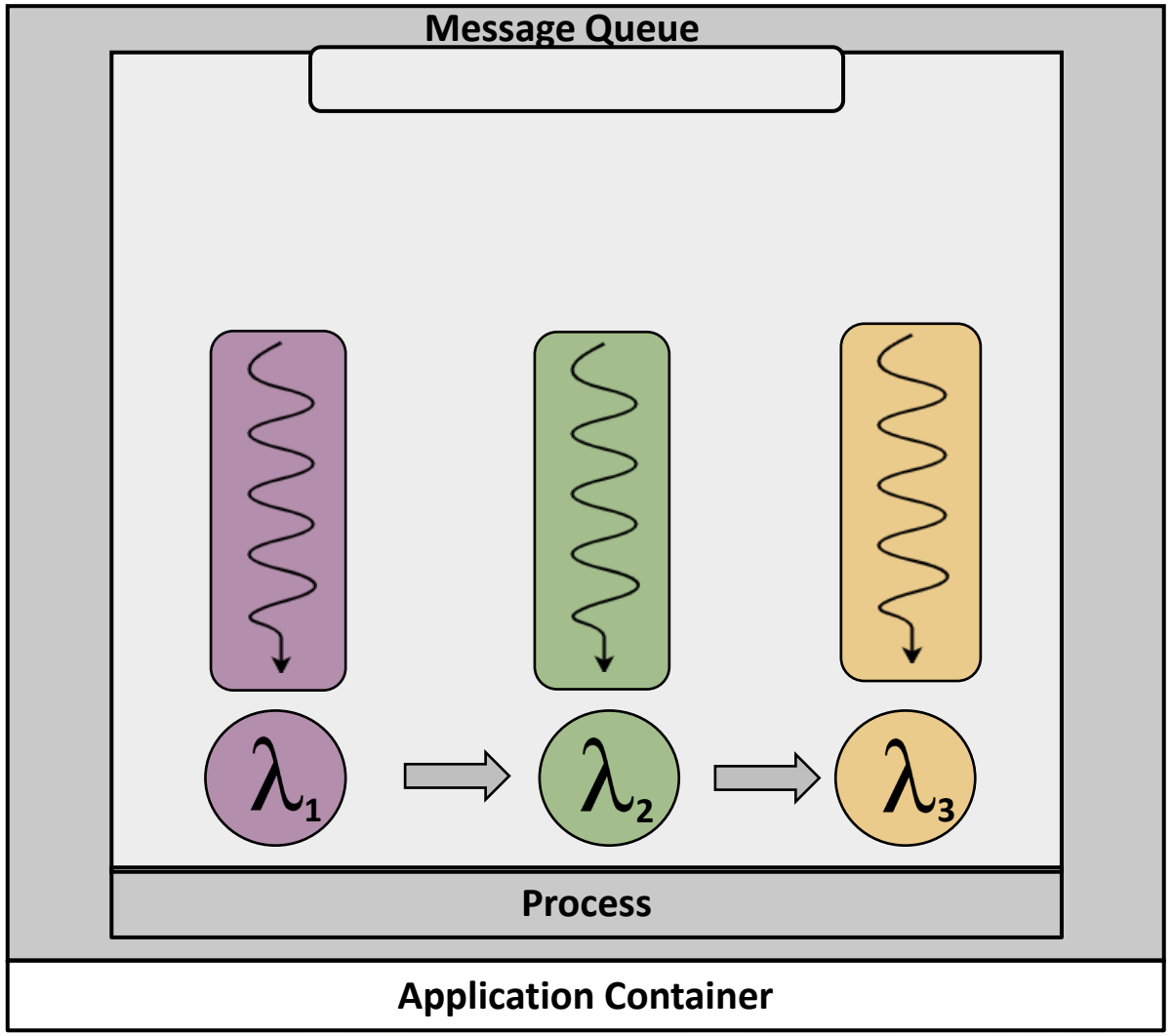
Faastlane: Running functions in threads



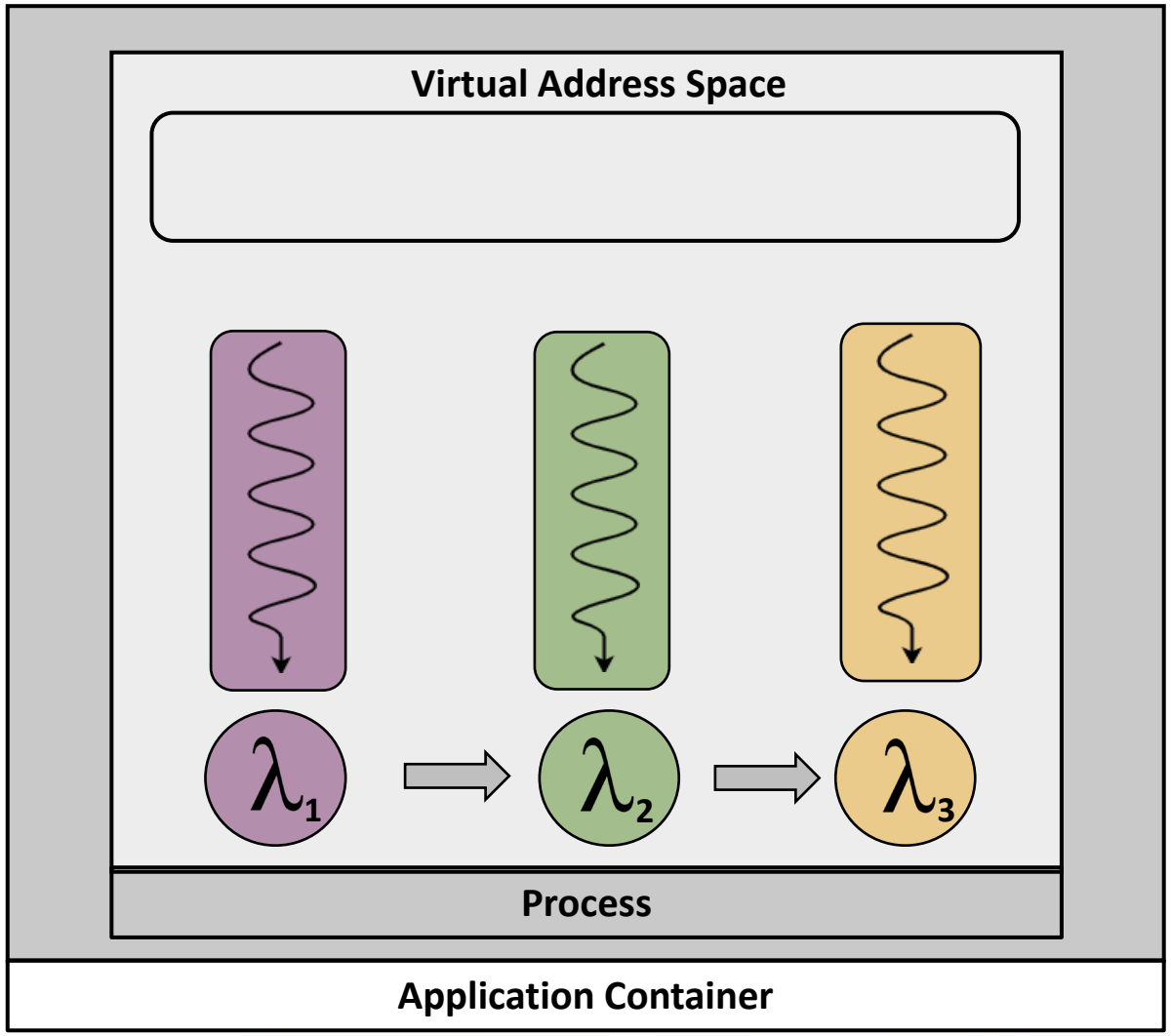
Faastlane: Running functions in threads



Faastlane: Running functions in threads

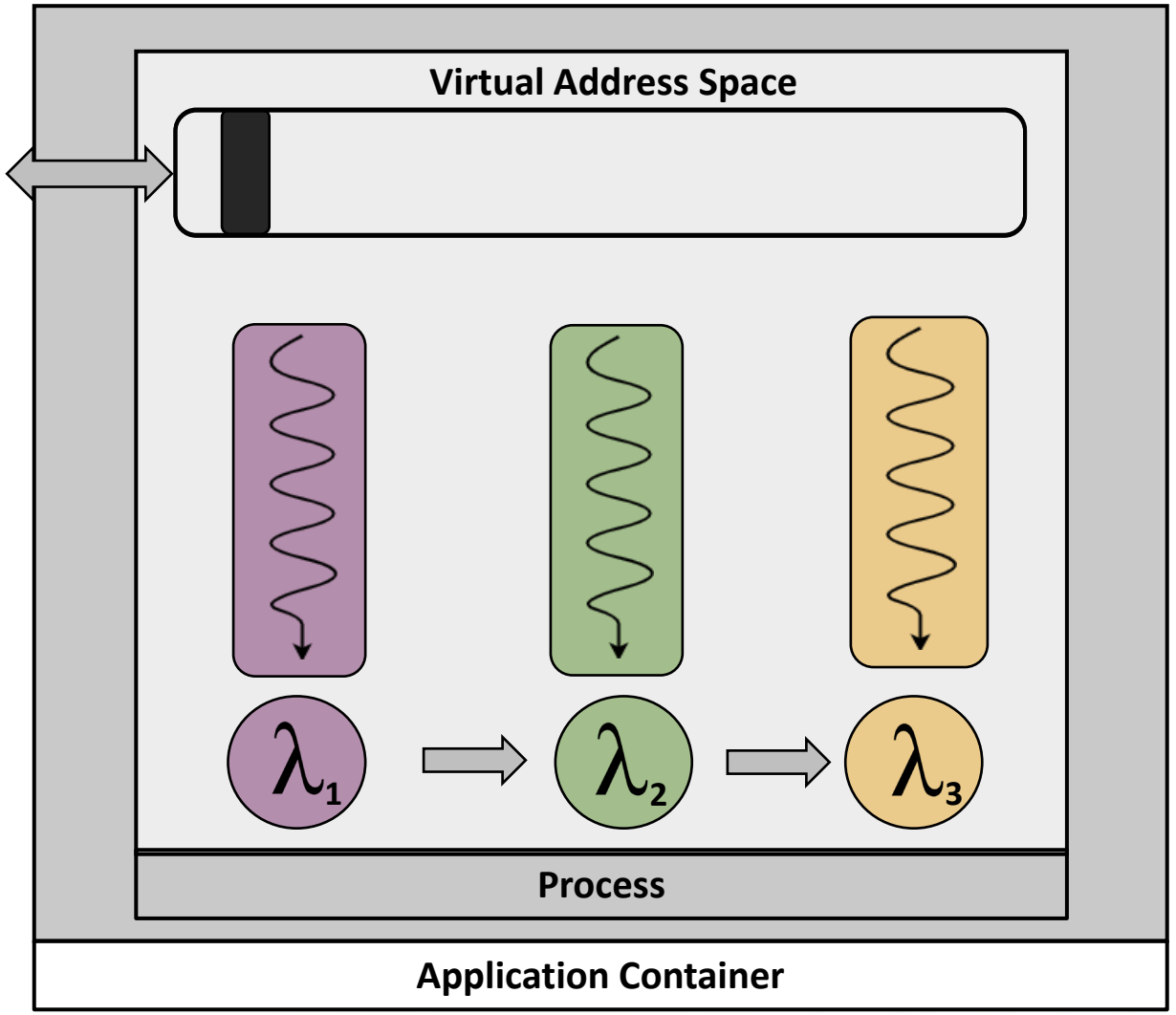


Faastlane: Running functions in threads



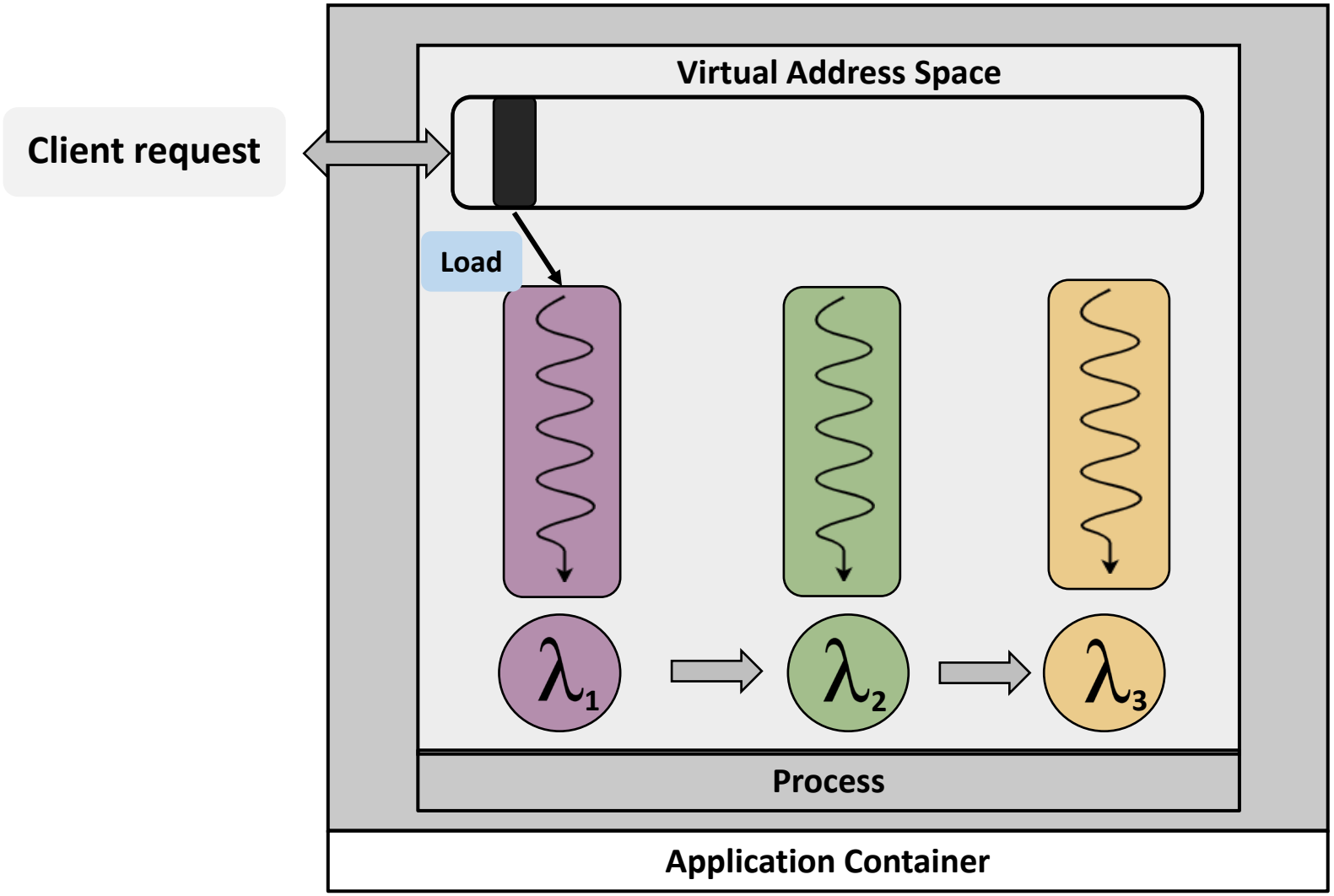
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



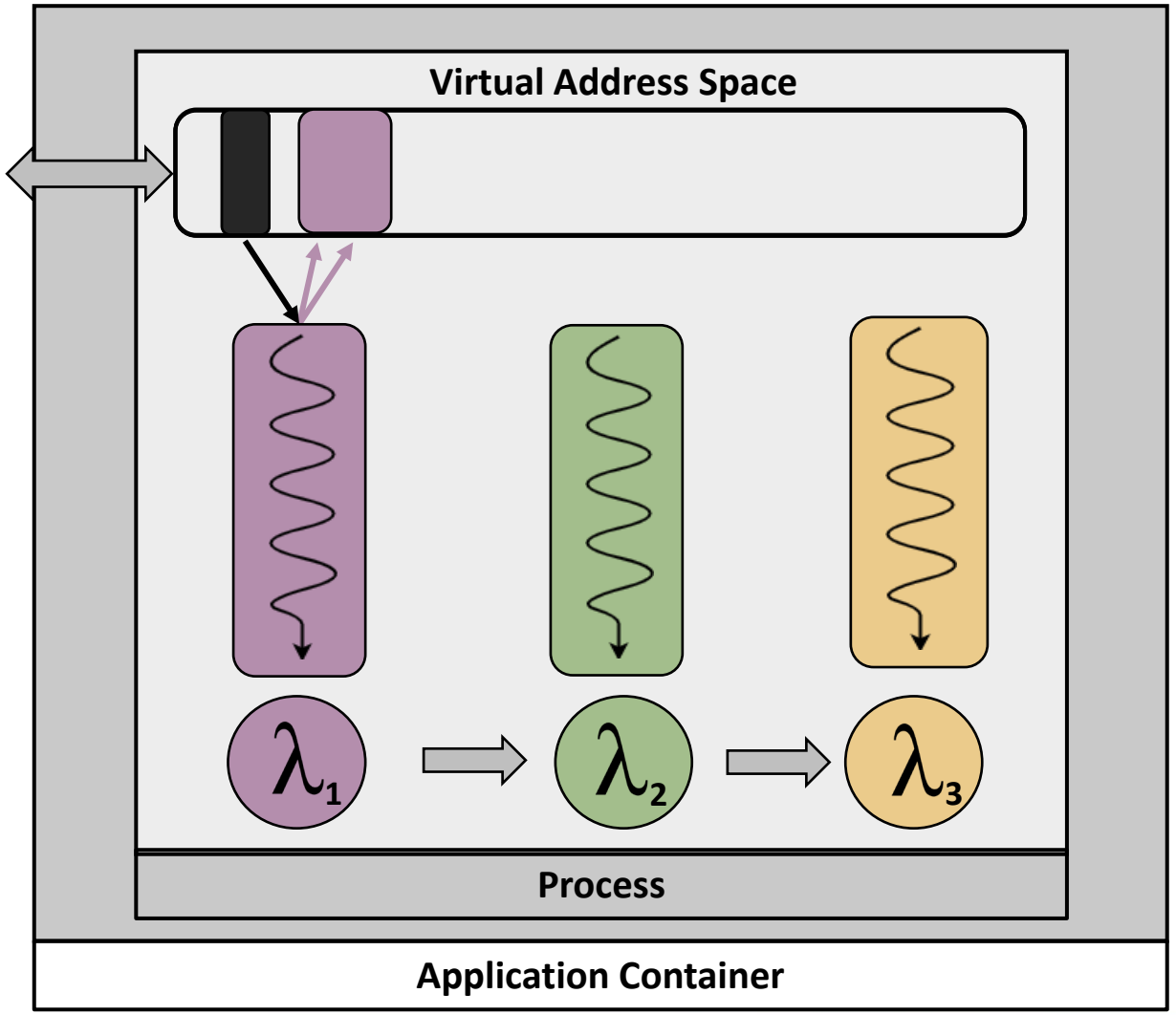
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



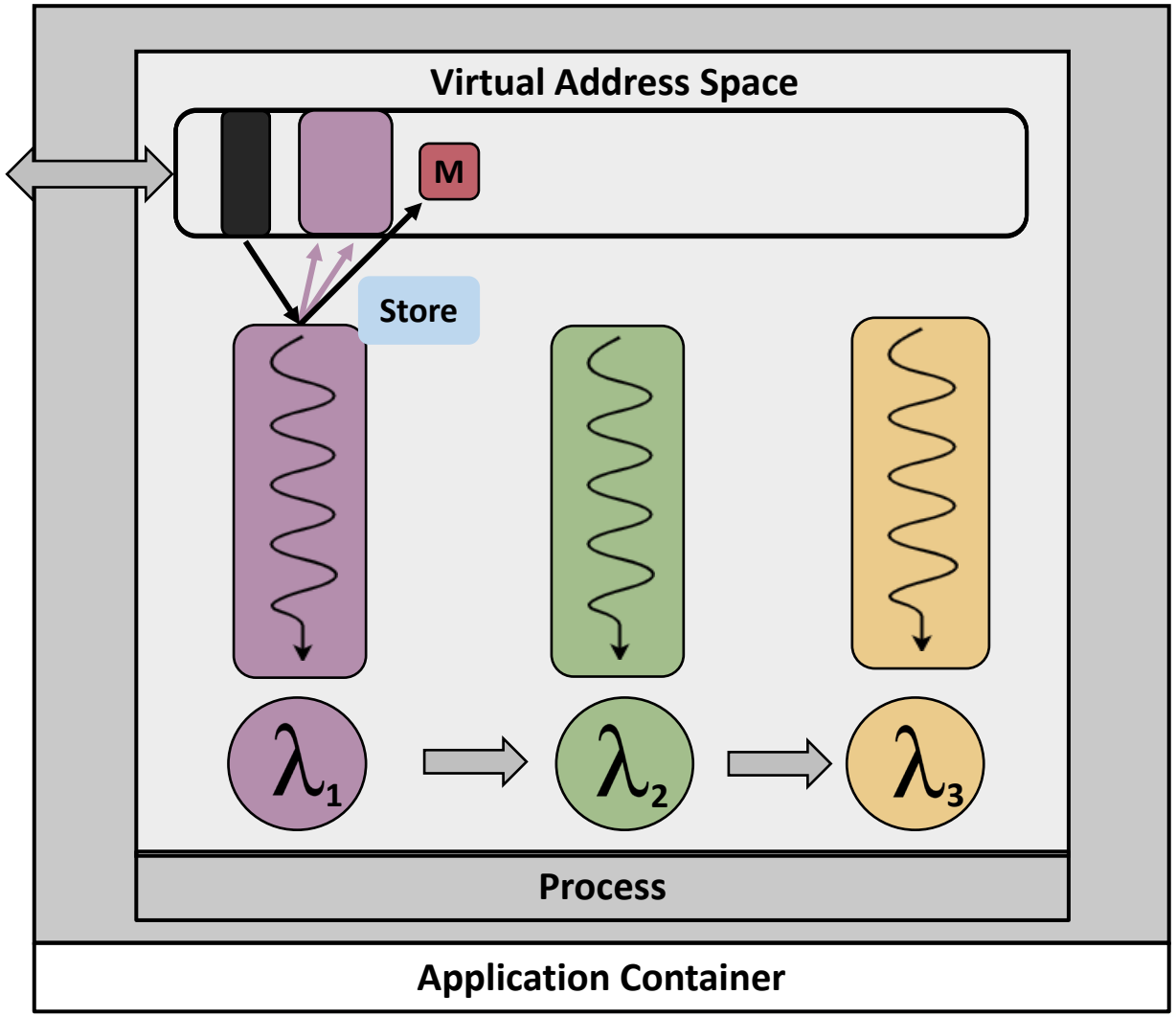
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



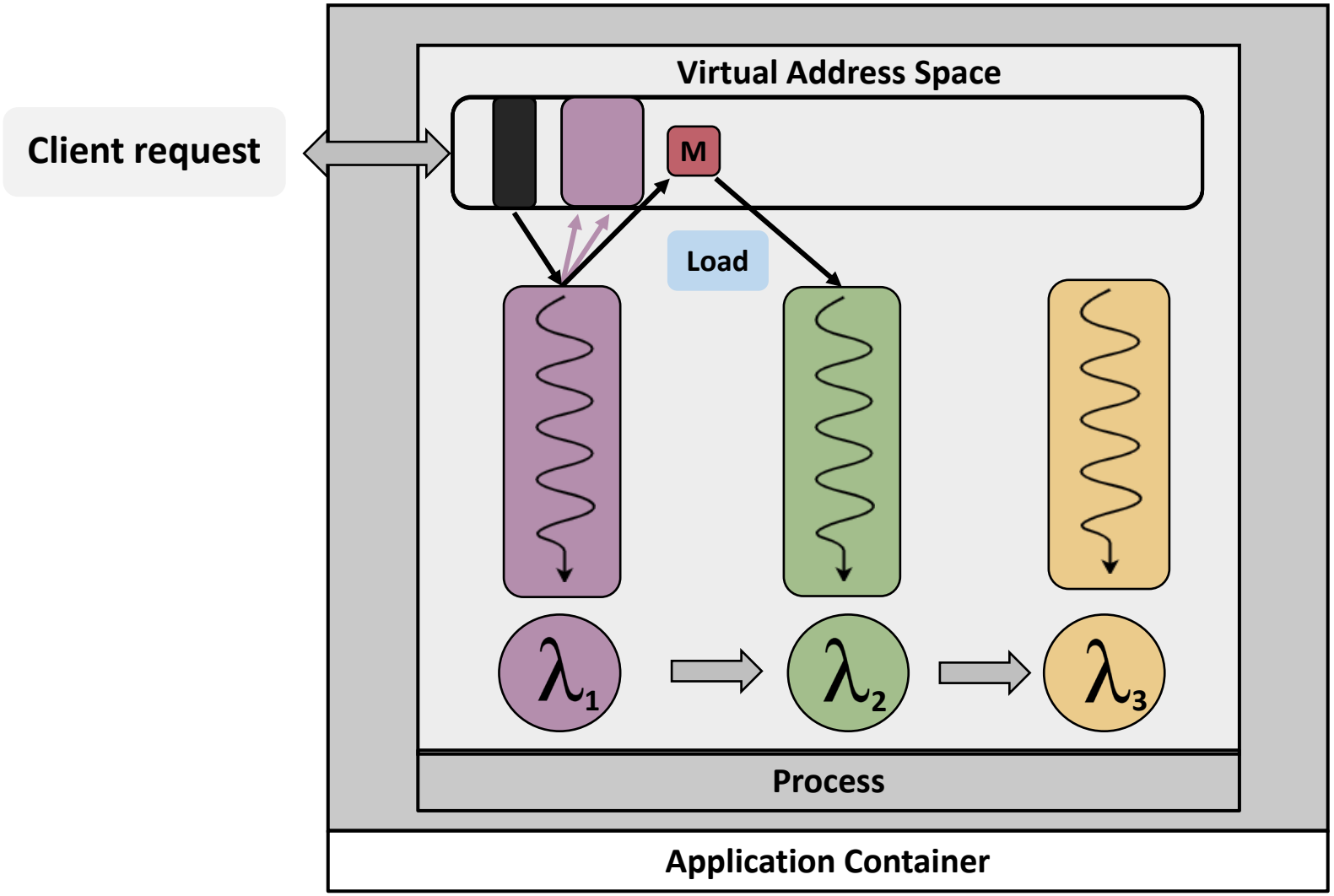
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



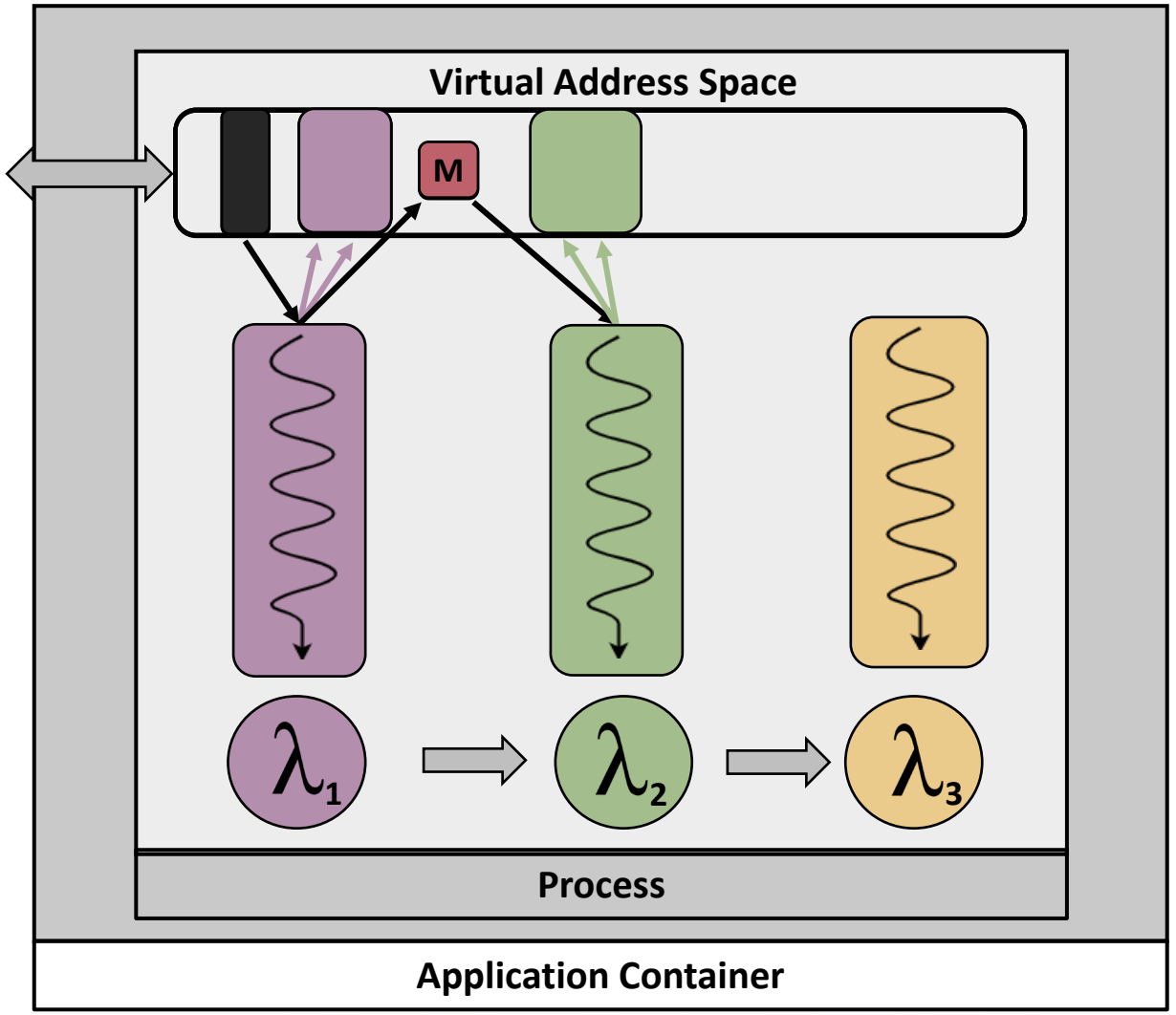
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



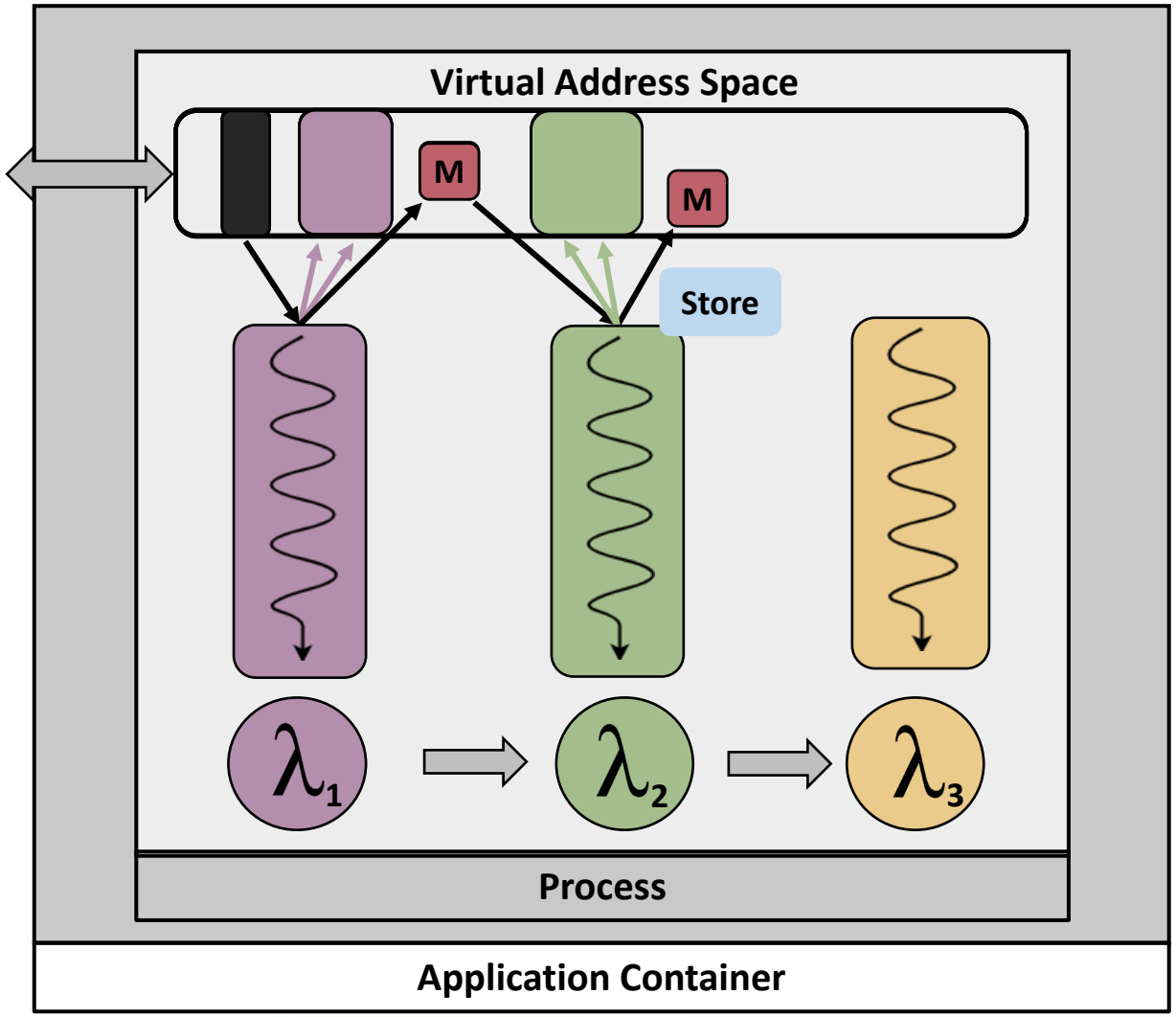
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



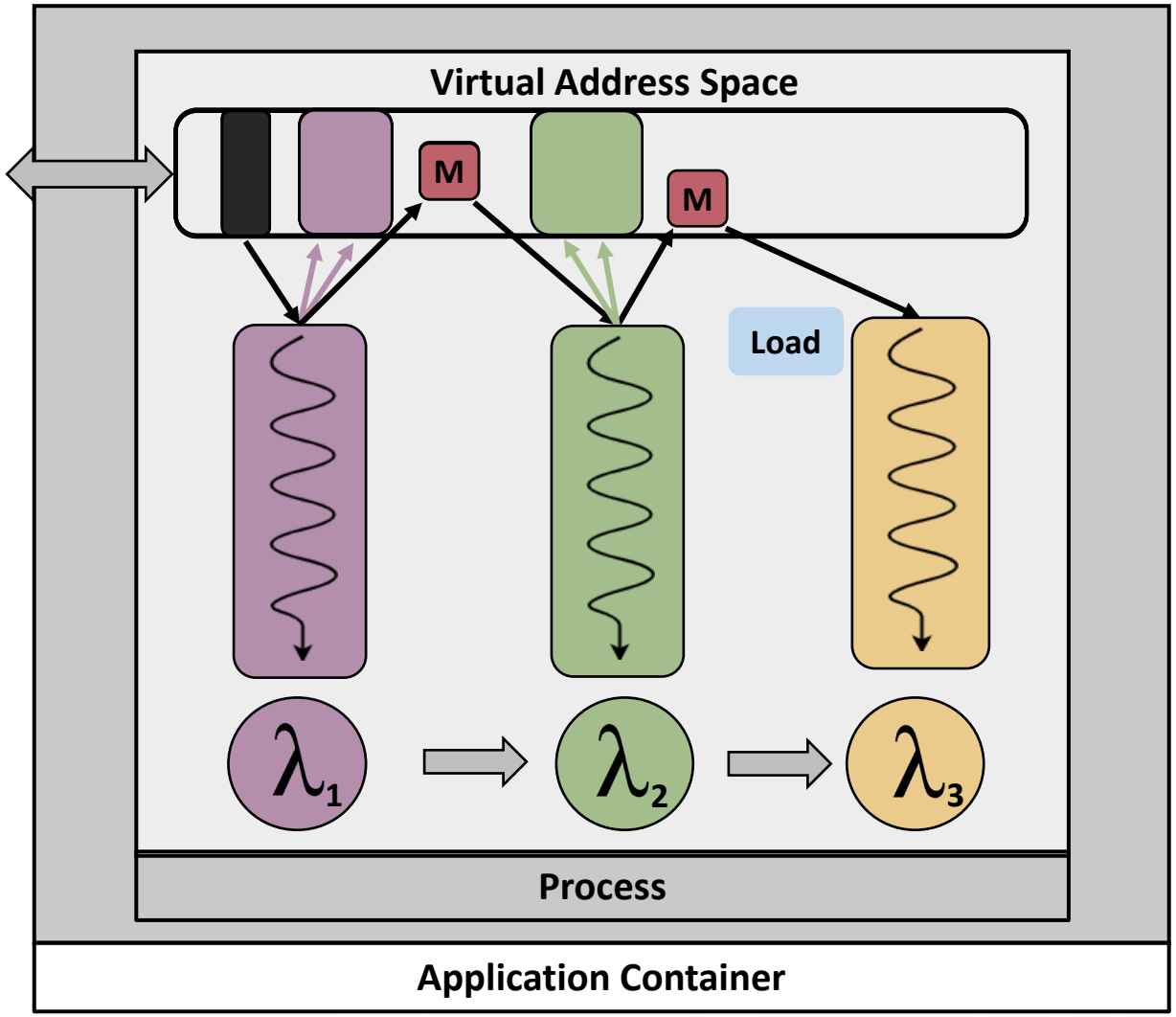
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



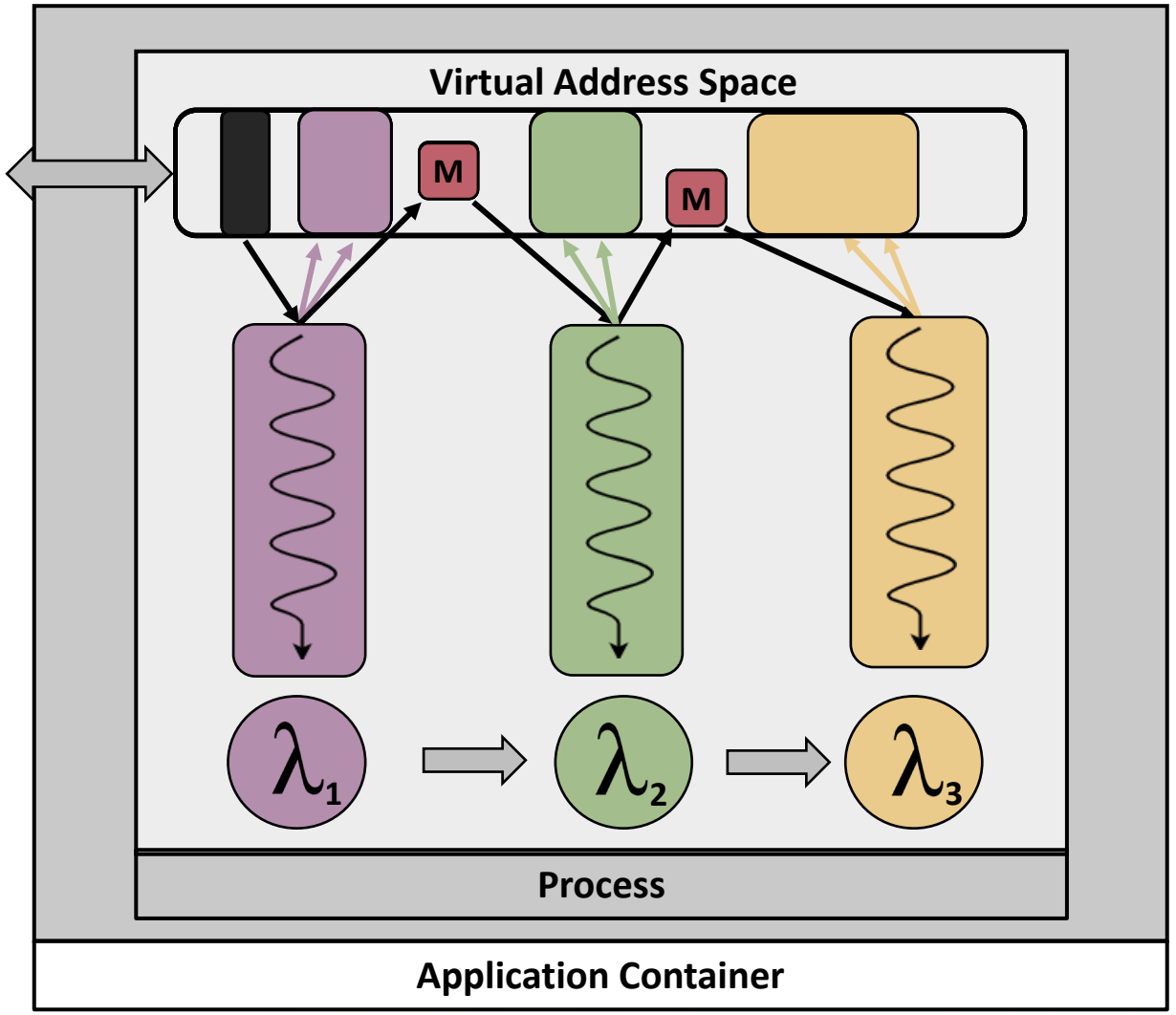
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



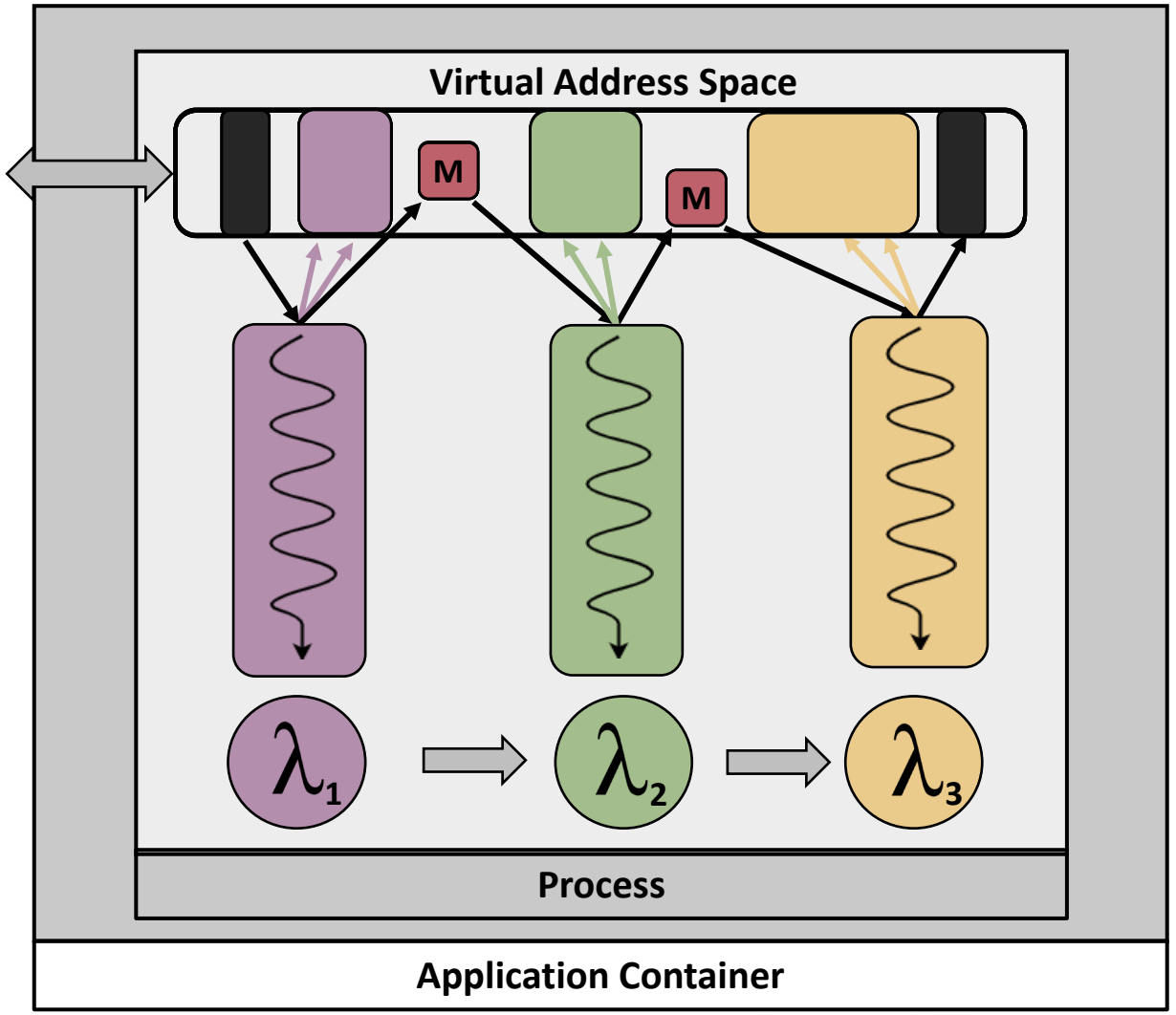
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



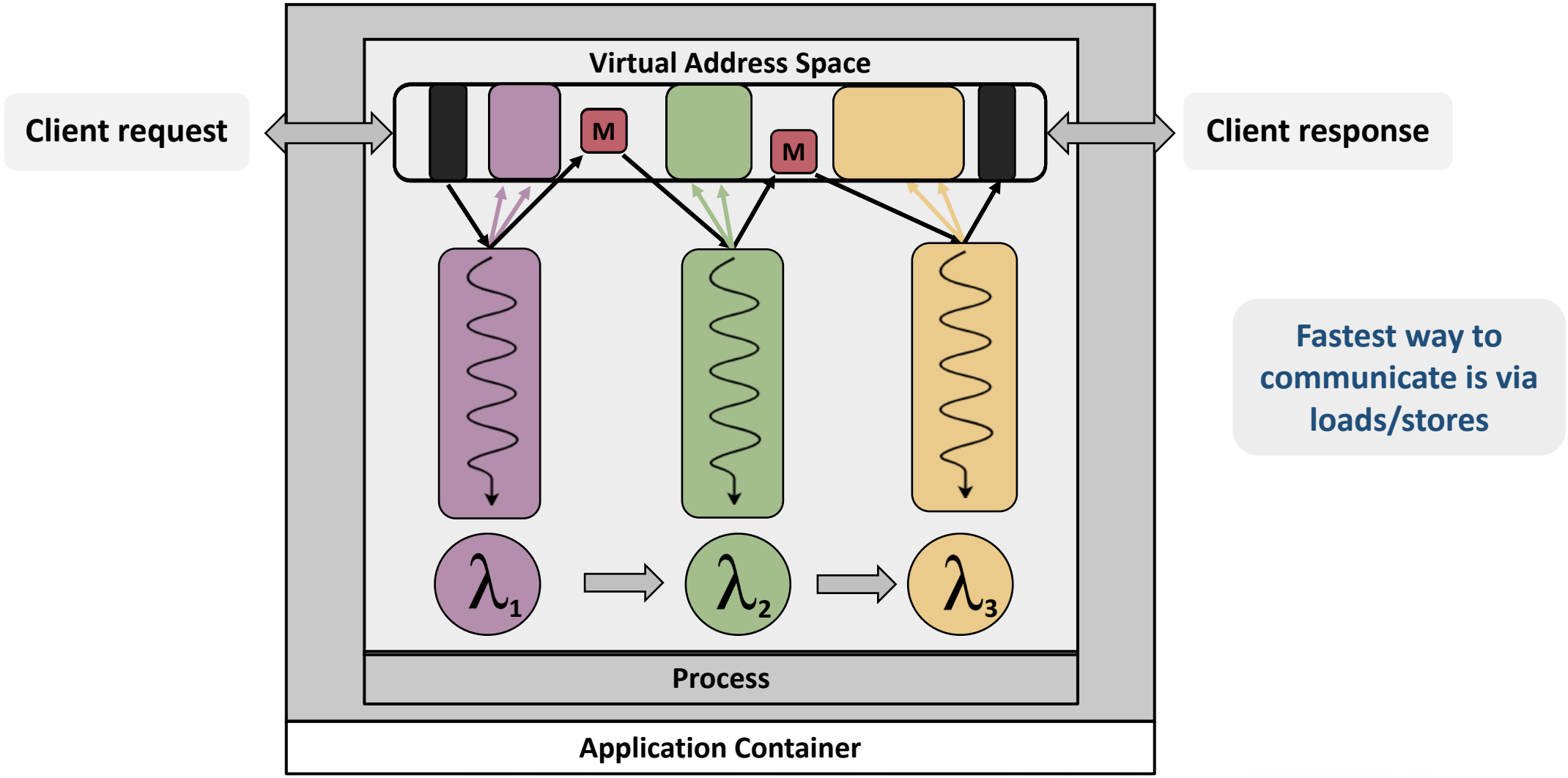
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



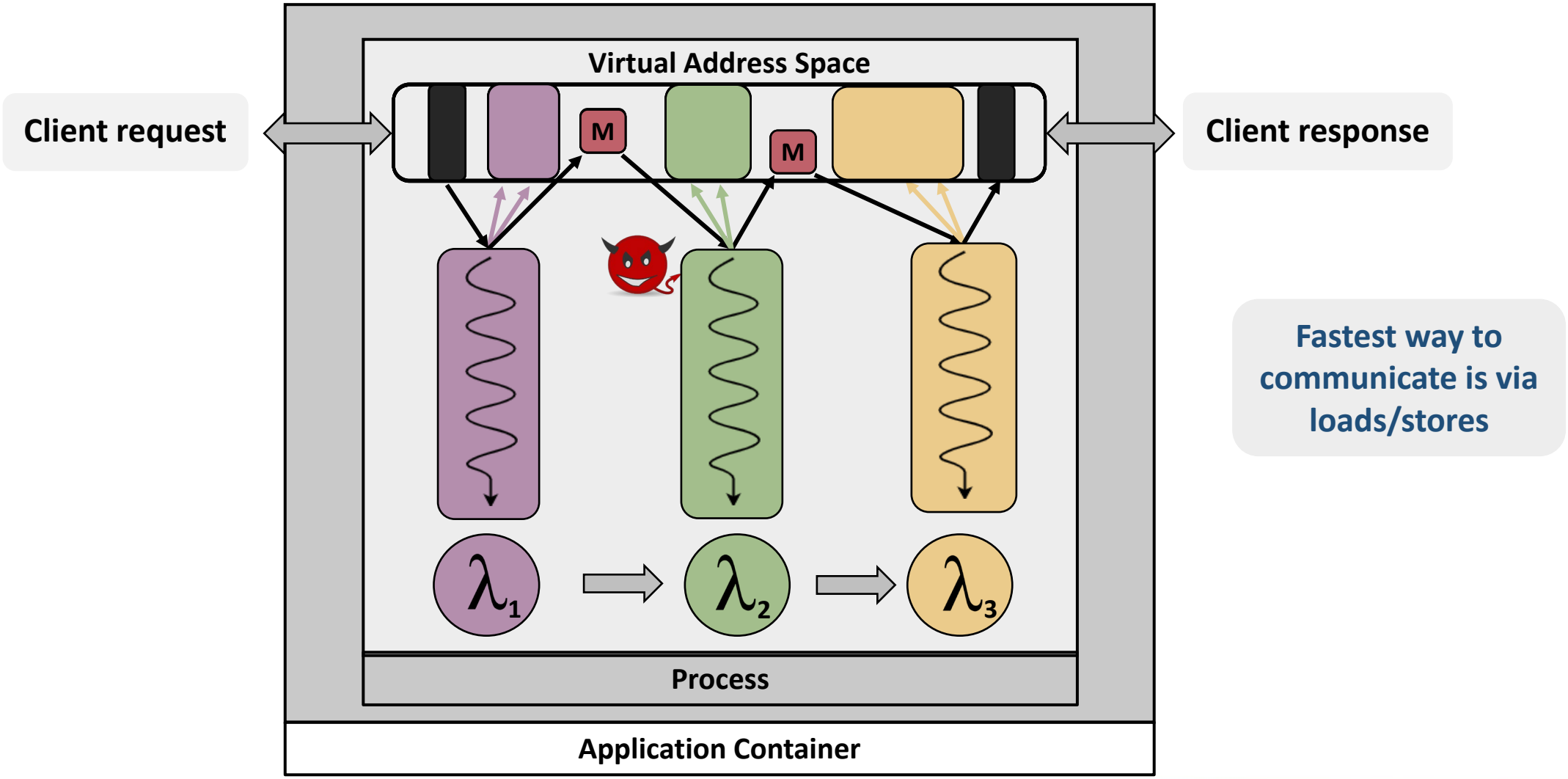
Fastest way to communicate is via loads/stores

Faastlane: Running functions in threads



Fastest way to communicate is via loads/stores

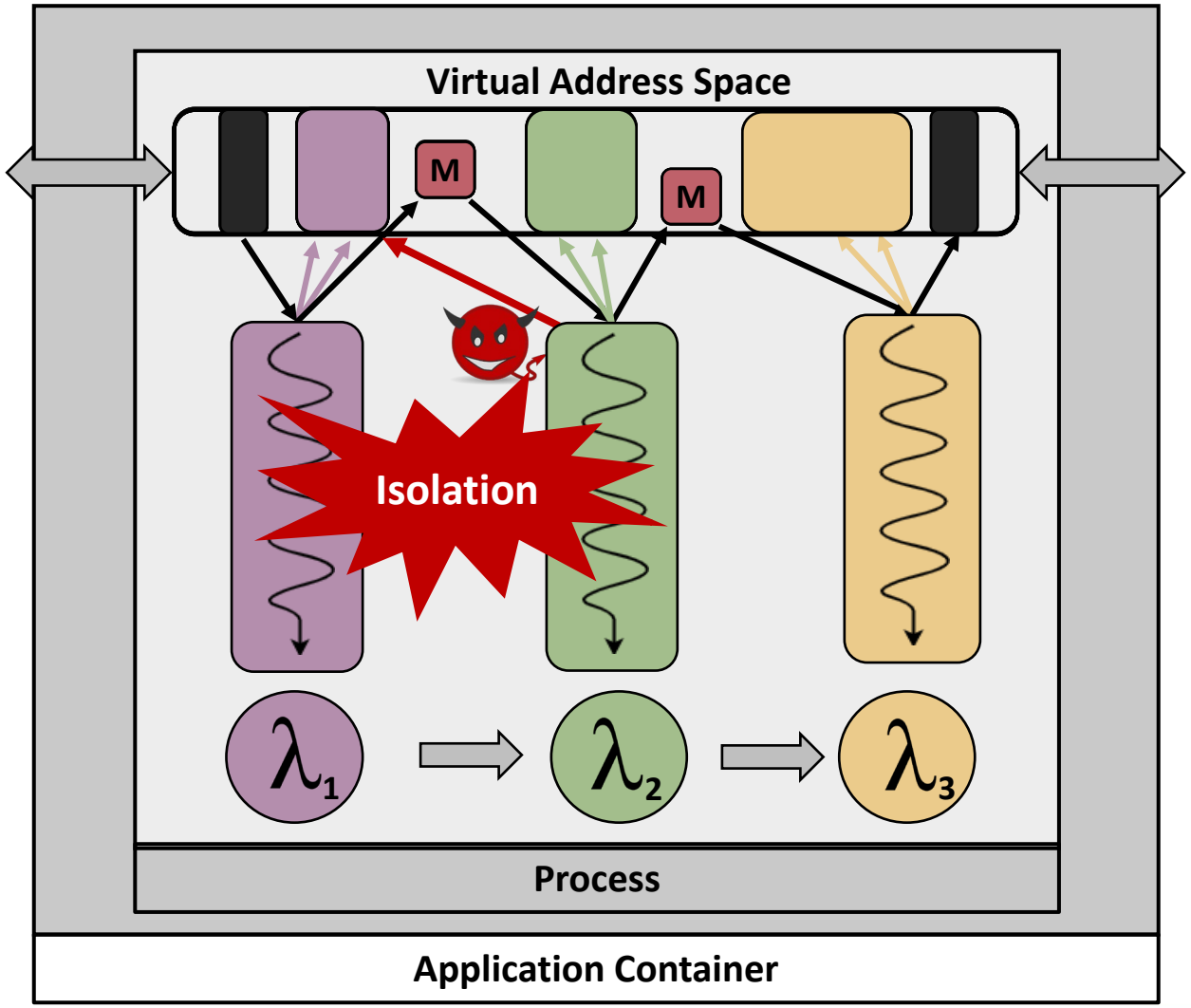
Faastlane: Running functions in threads



Fastest way to communicate is via loads/stores

Challenge 1: Protect function's private memory

Client request

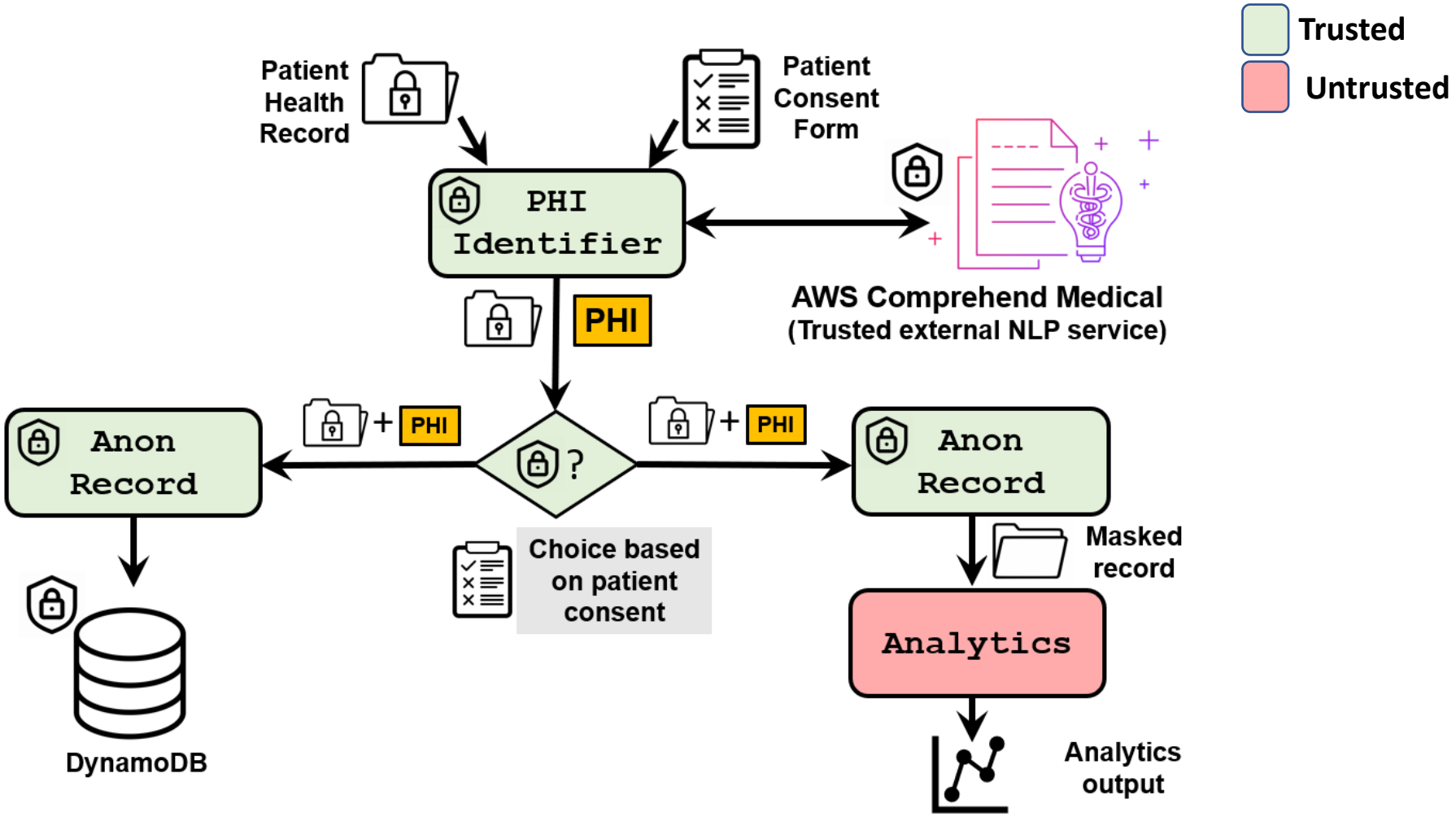


Client response

One function can access another's private data

Fastest way to communicate is via loads/stores

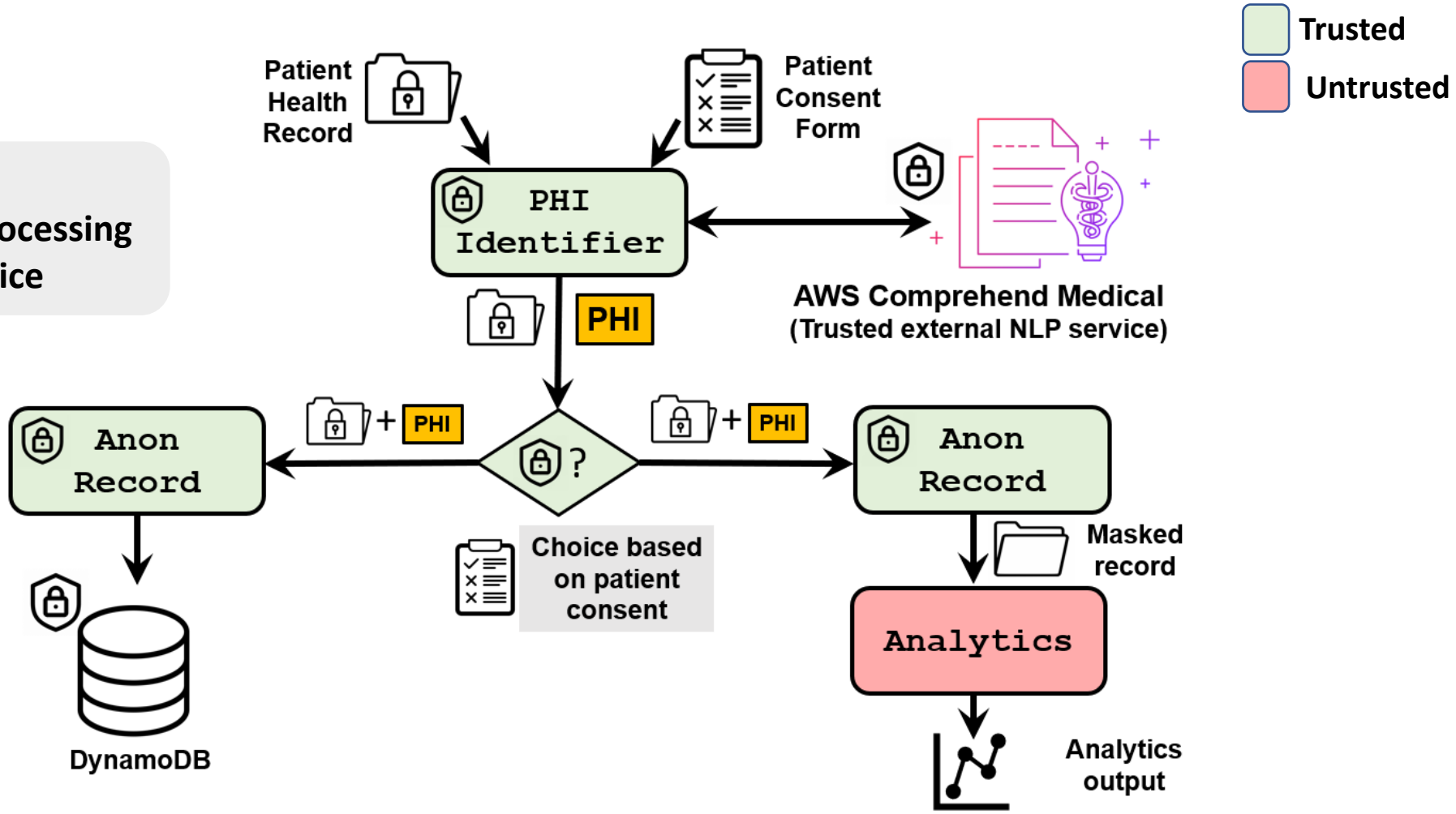
Healthcare analytics: A case for isolation



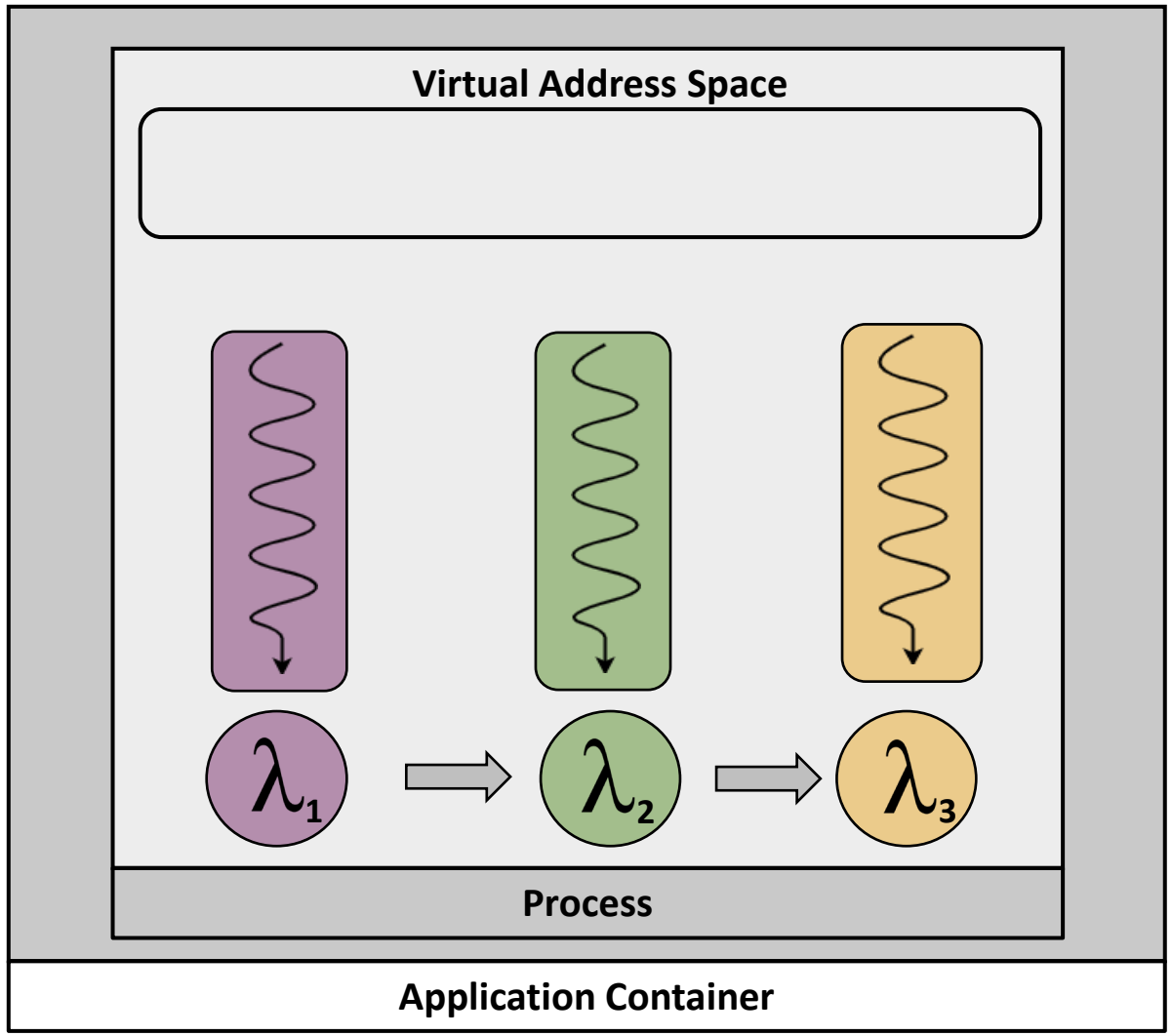
Healthcare analytics: A case for isolation

Other examples

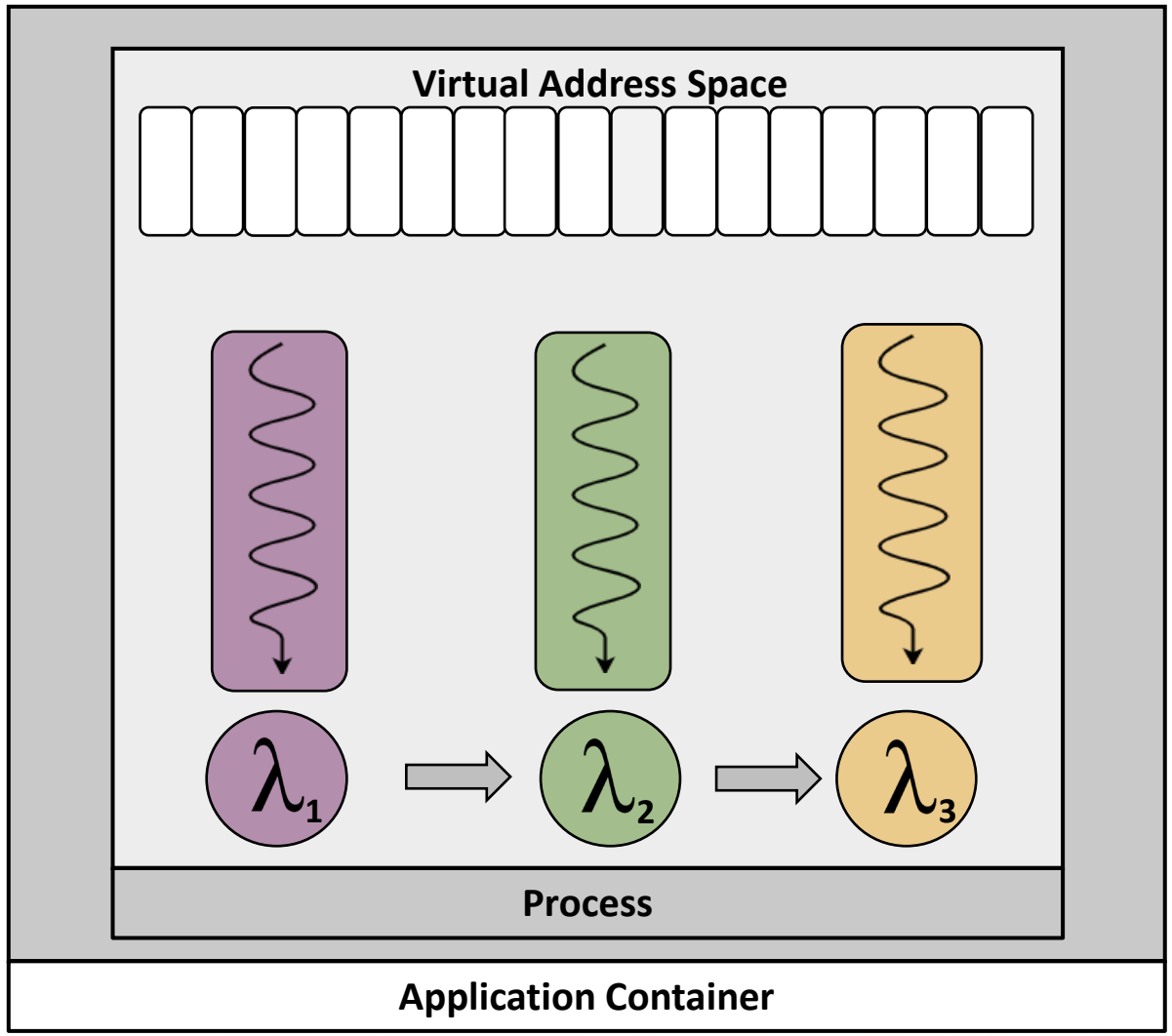
1. **Financial record processing**
2. **ML prediction service**



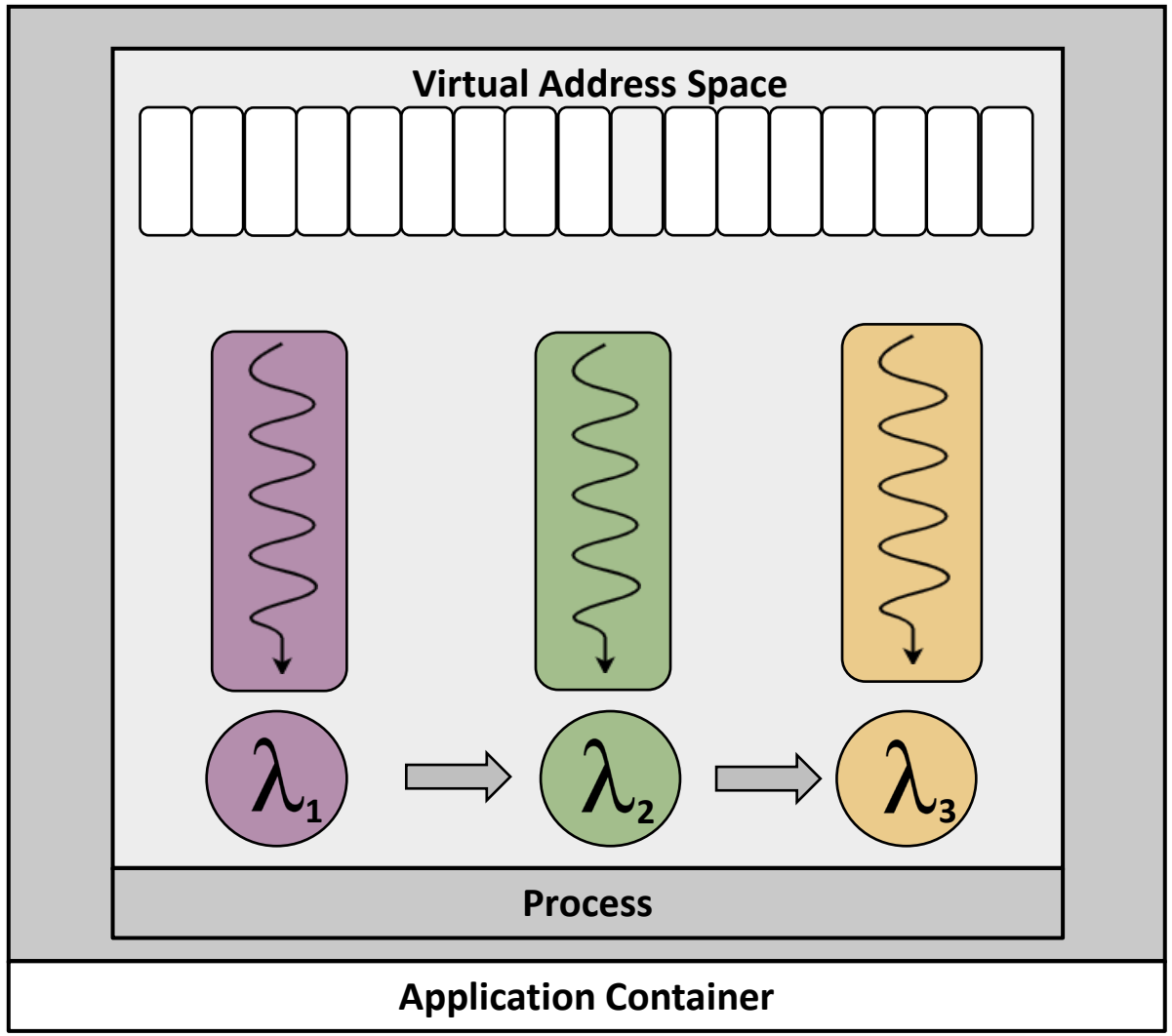
Solution 1: Lightweight isolation with Intel MPK



Solution 1: Lightweight isolation with Intel MPK

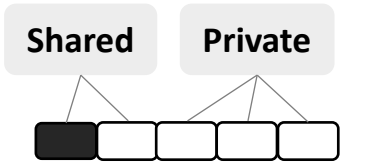


Solution 1: Lightweight isolation with Intel MPK

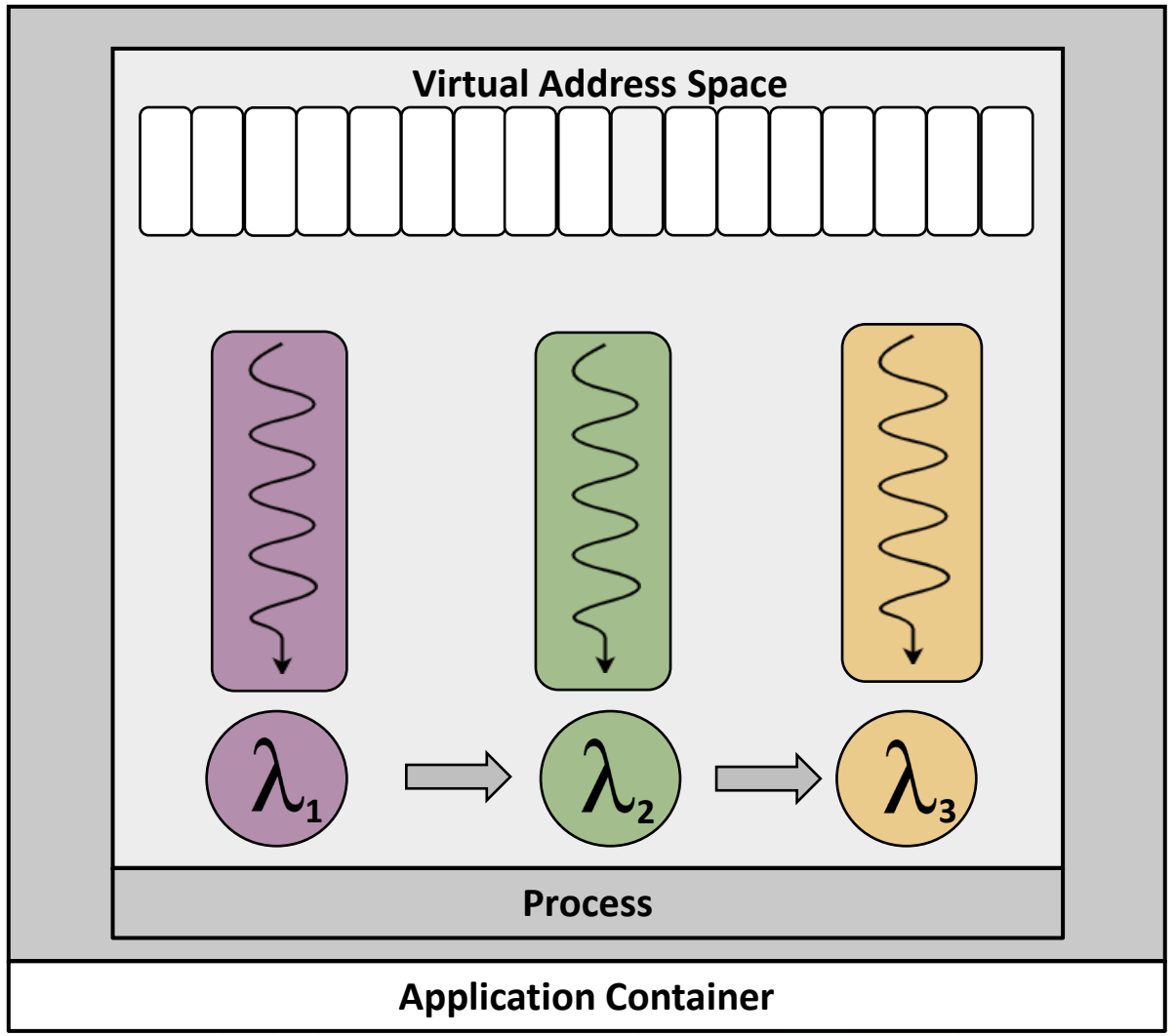


PKRU register
Protection key permissions

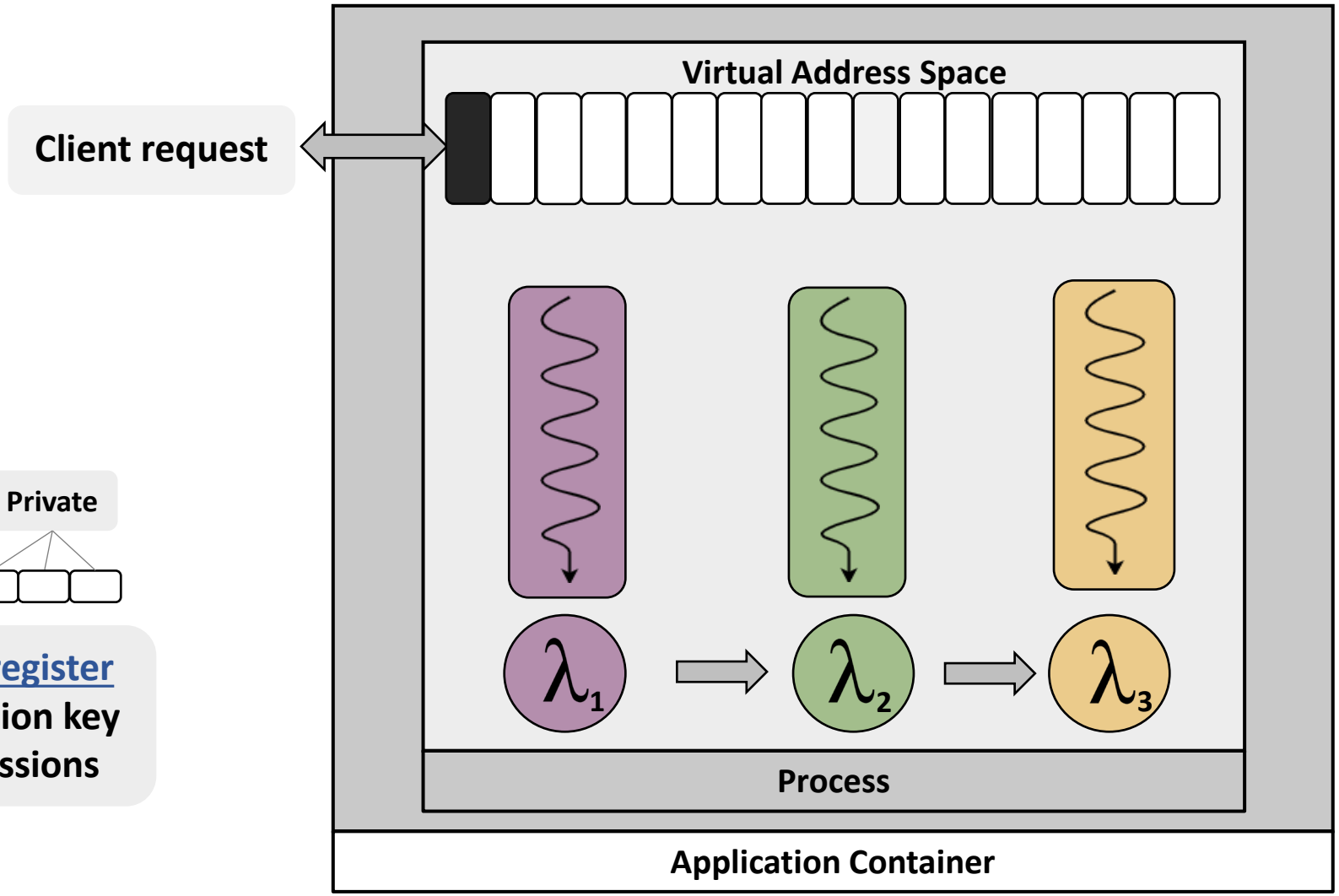
Solution 1: Lightweight isolation with Intel MPK

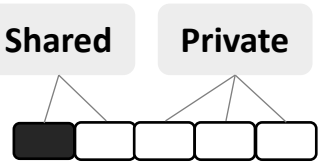


PKRU register
Protection key permissions



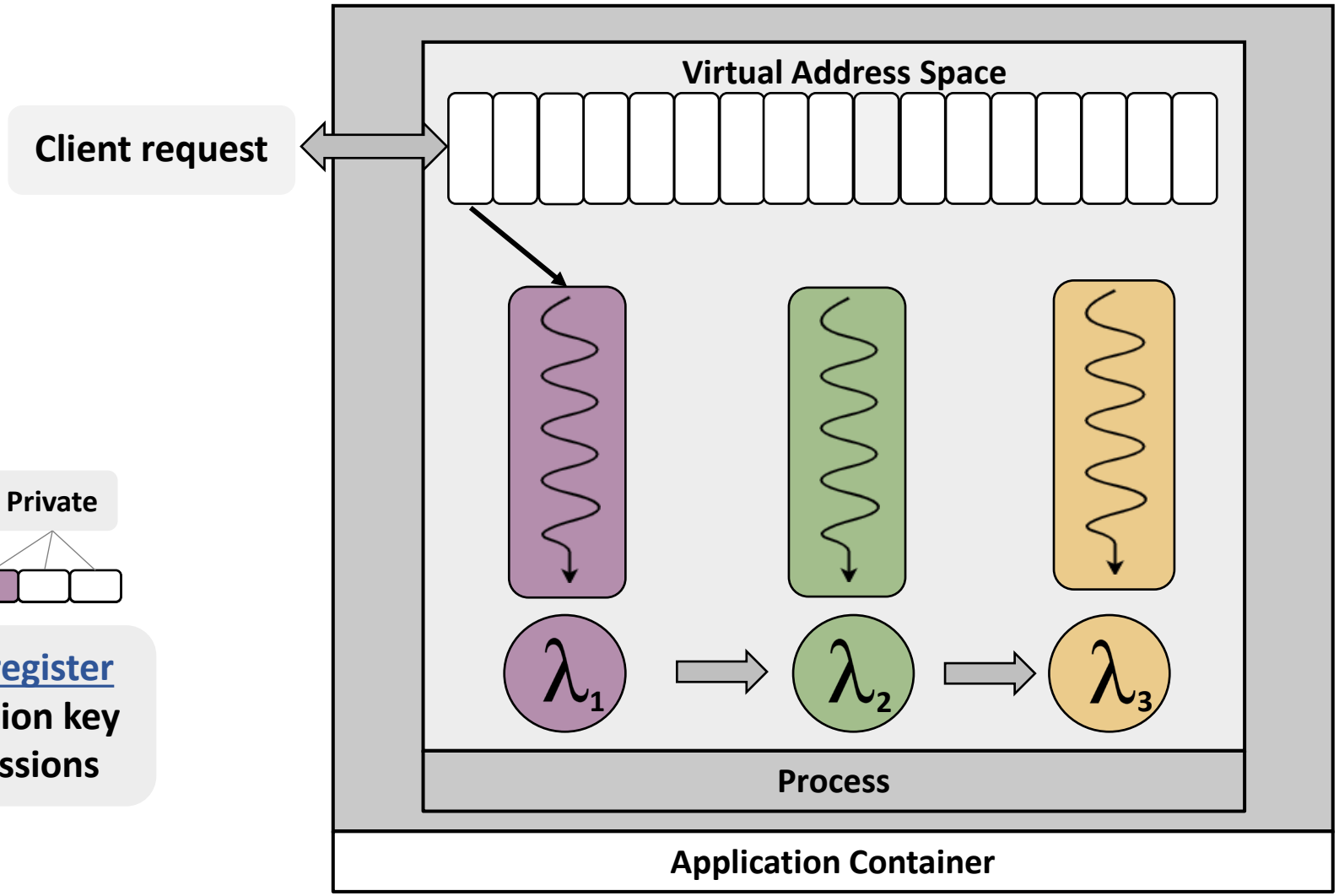
Solution 1: Lightweight isolation with Intel MPK

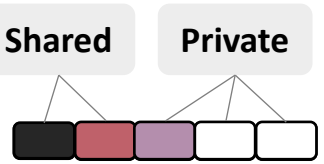




PKRU register
Protection key permissions

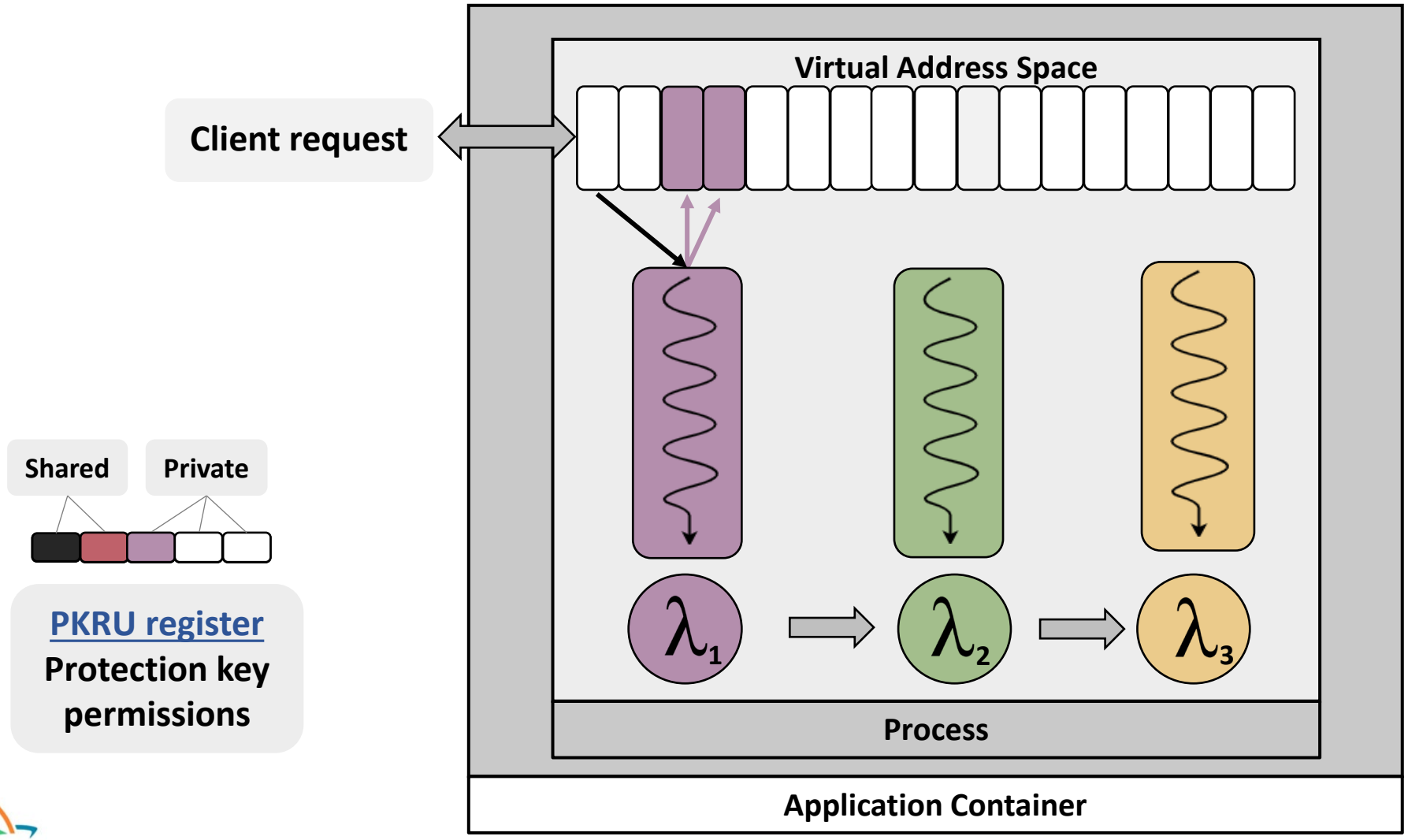
Solution 1: Lightweight isolation with Intel MPK



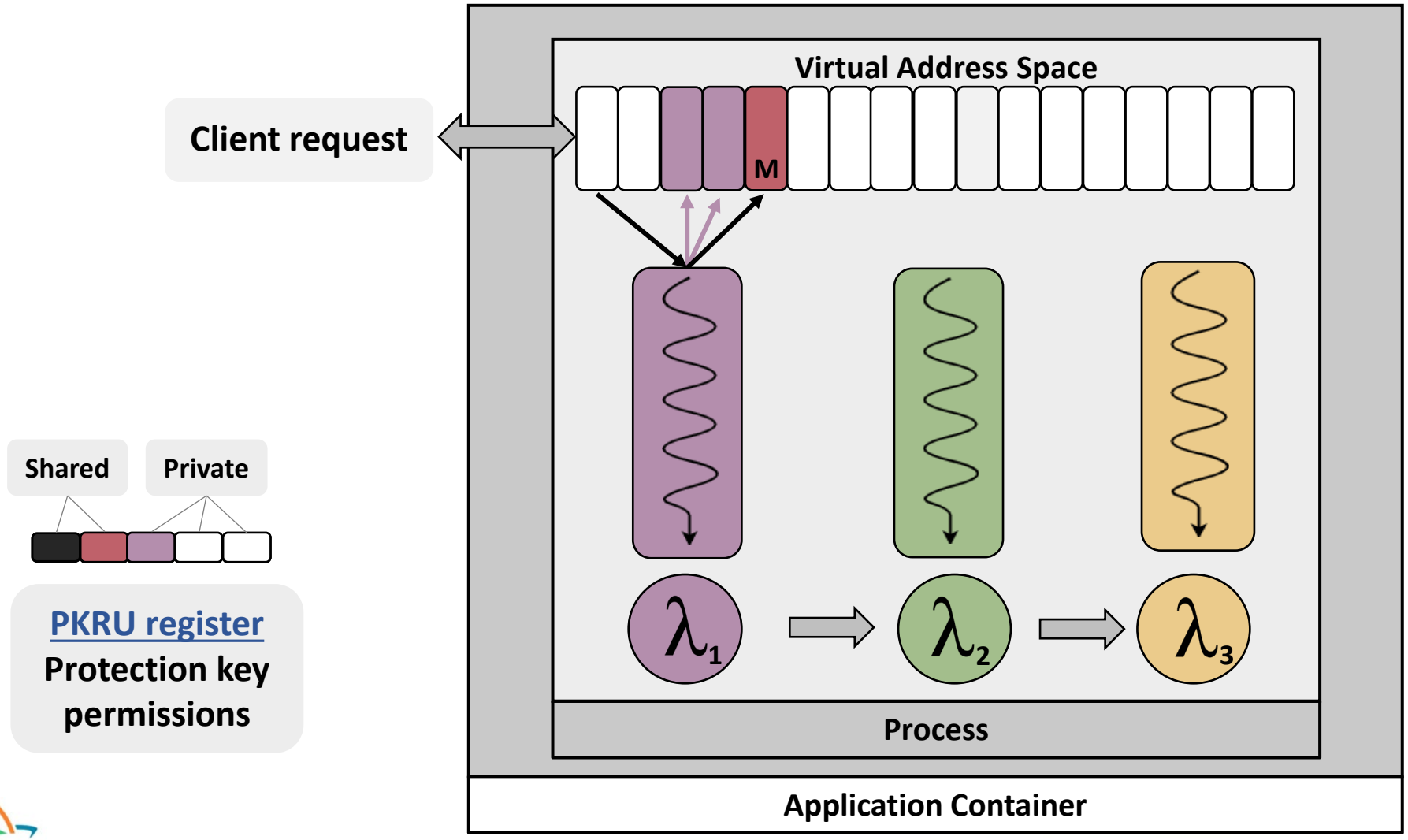


PKRU register
Protection key permissions

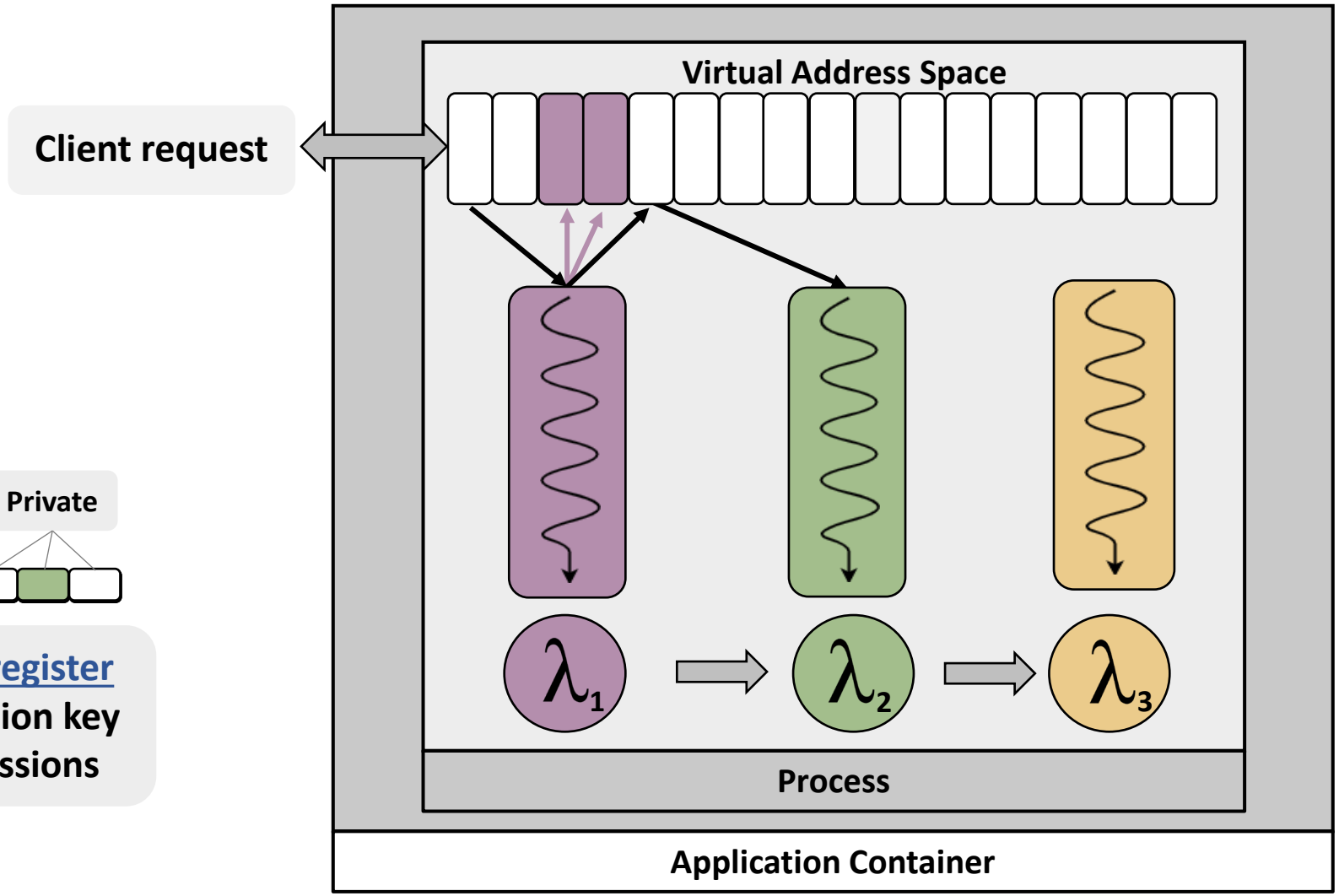
Solution 1: Lightweight isolation with Intel MPK

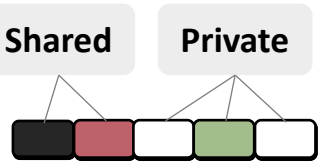


Solution 1: Lightweight isolation with Intel MPK



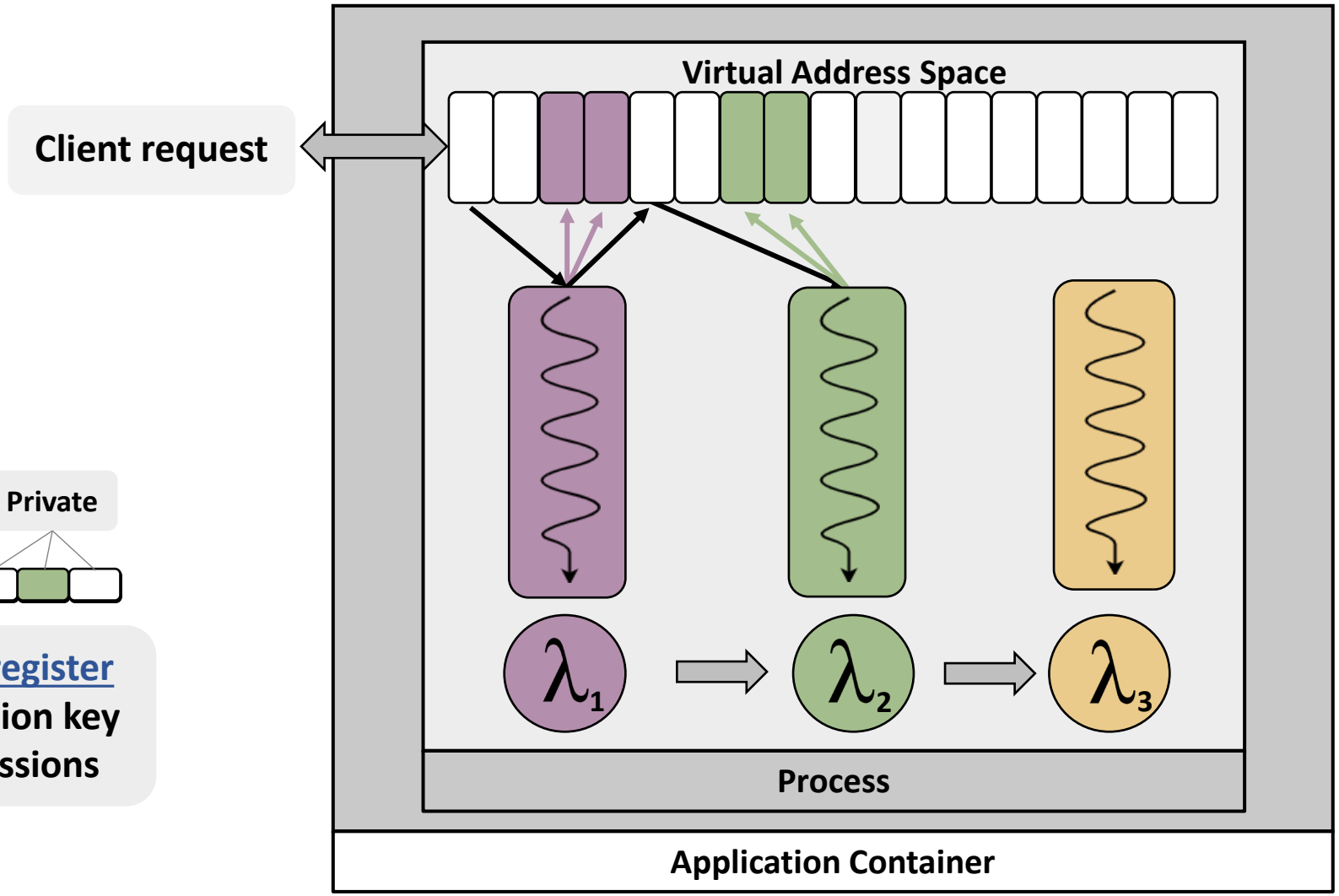
Solution 1: Lightweight isolation with Intel MPK

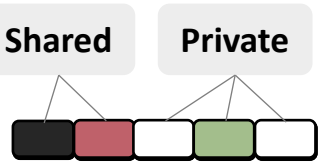




PKRU register
Protection key permissions

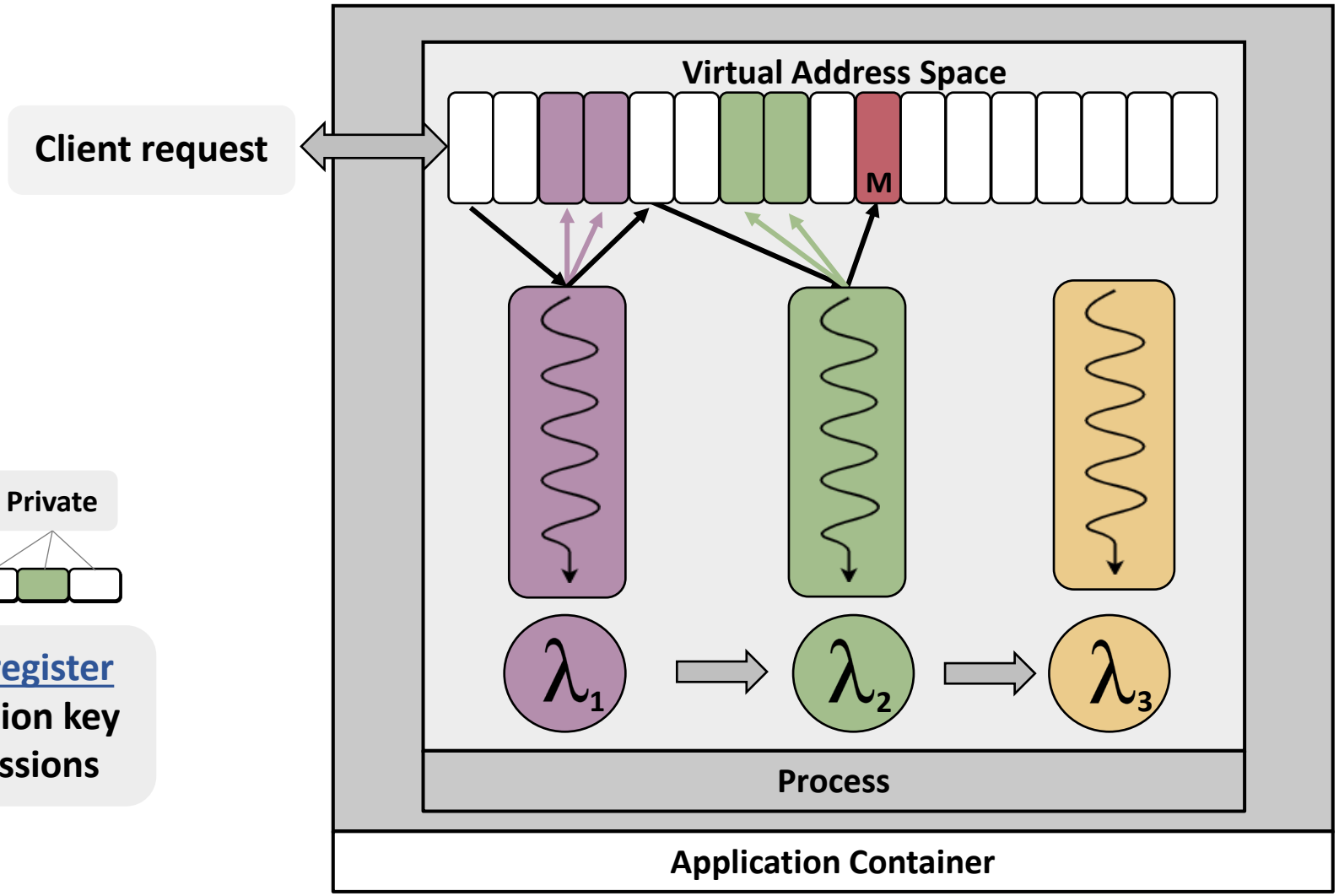
Solution 1: Lightweight isolation with Intel MPK

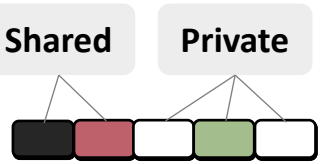




PKRU register
Protection key permissions

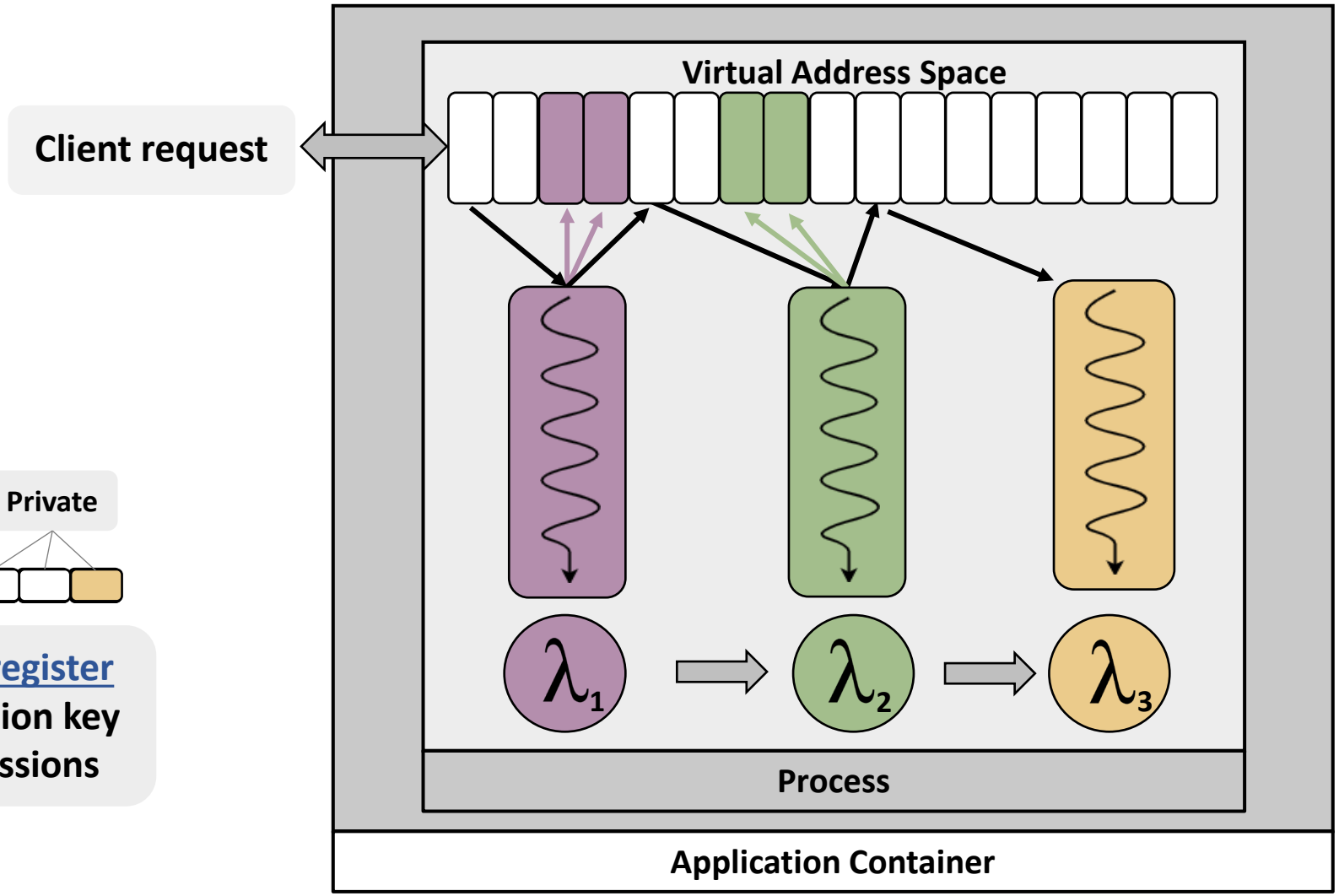
Solution 1: Lightweight isolation with Intel MPK

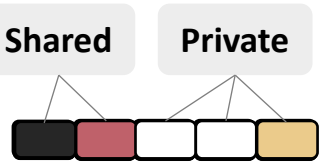




PKRU register
Protection key permissions

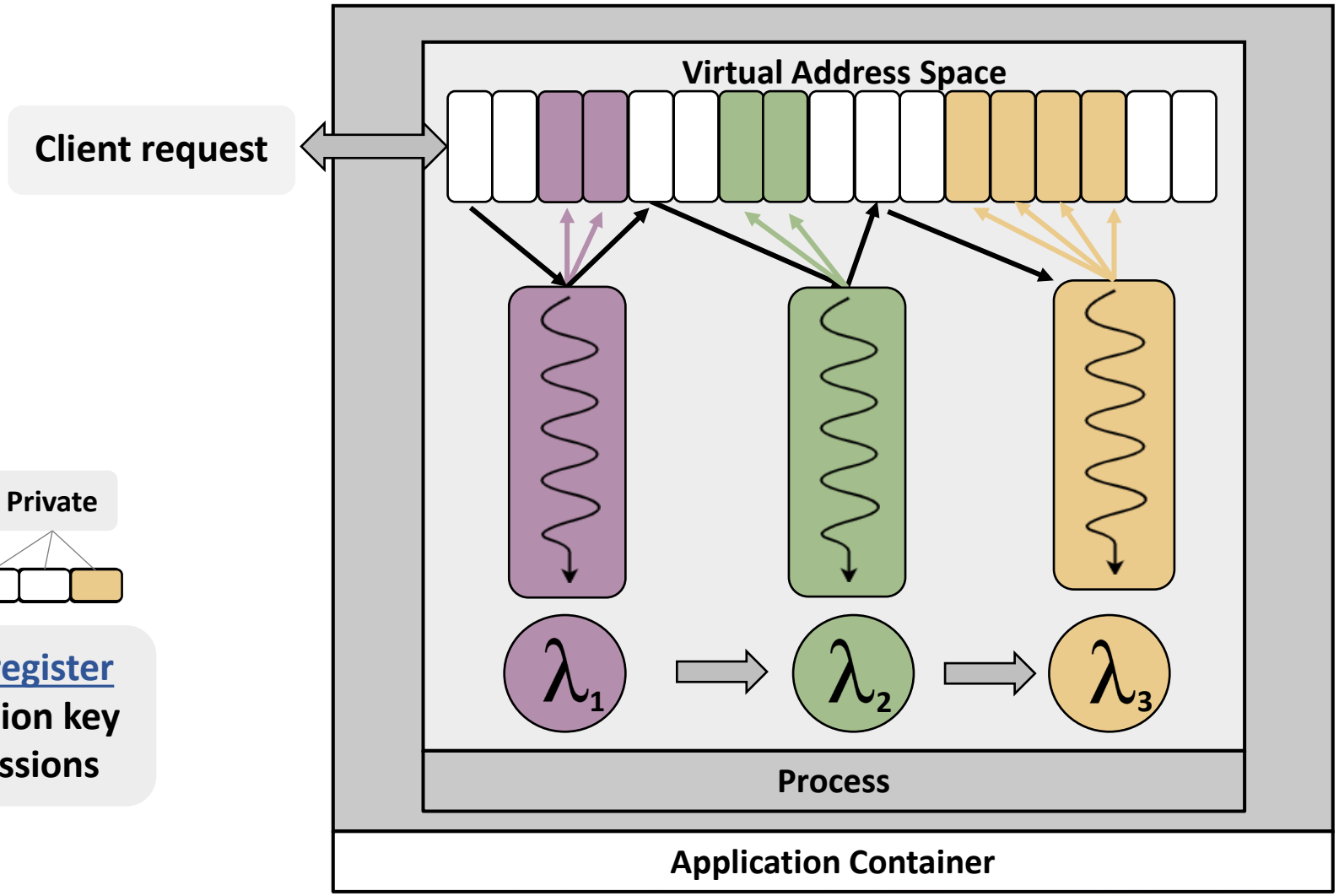
Solution 1: Lightweight isolation with Intel MPK

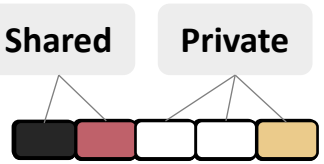




PKRU register
Protection key permissions

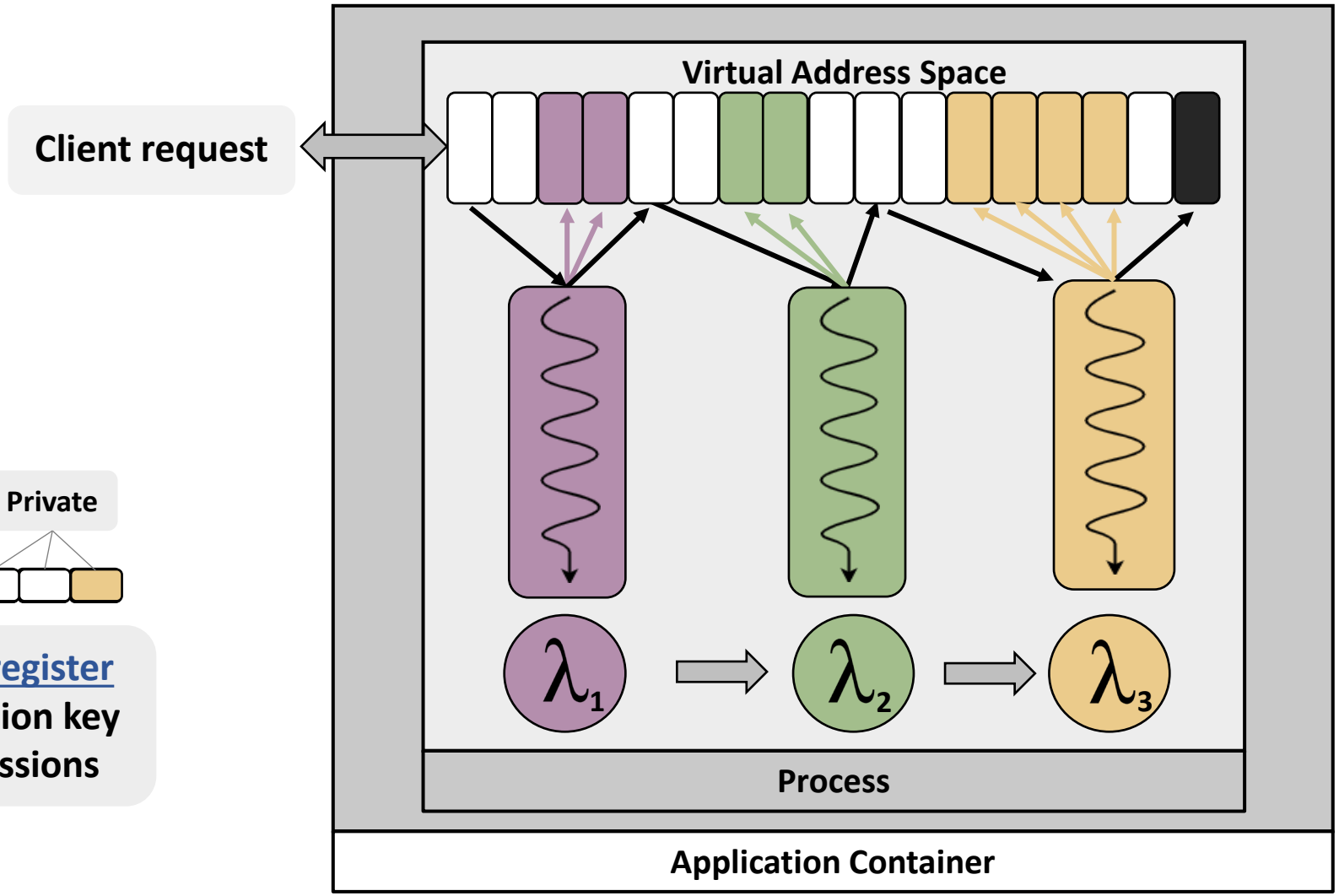
Solution 1: Lightweight isolation with Intel MPK

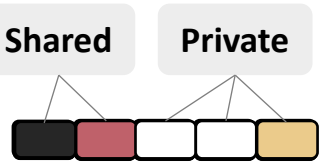




PKRU register
Protection key permissions

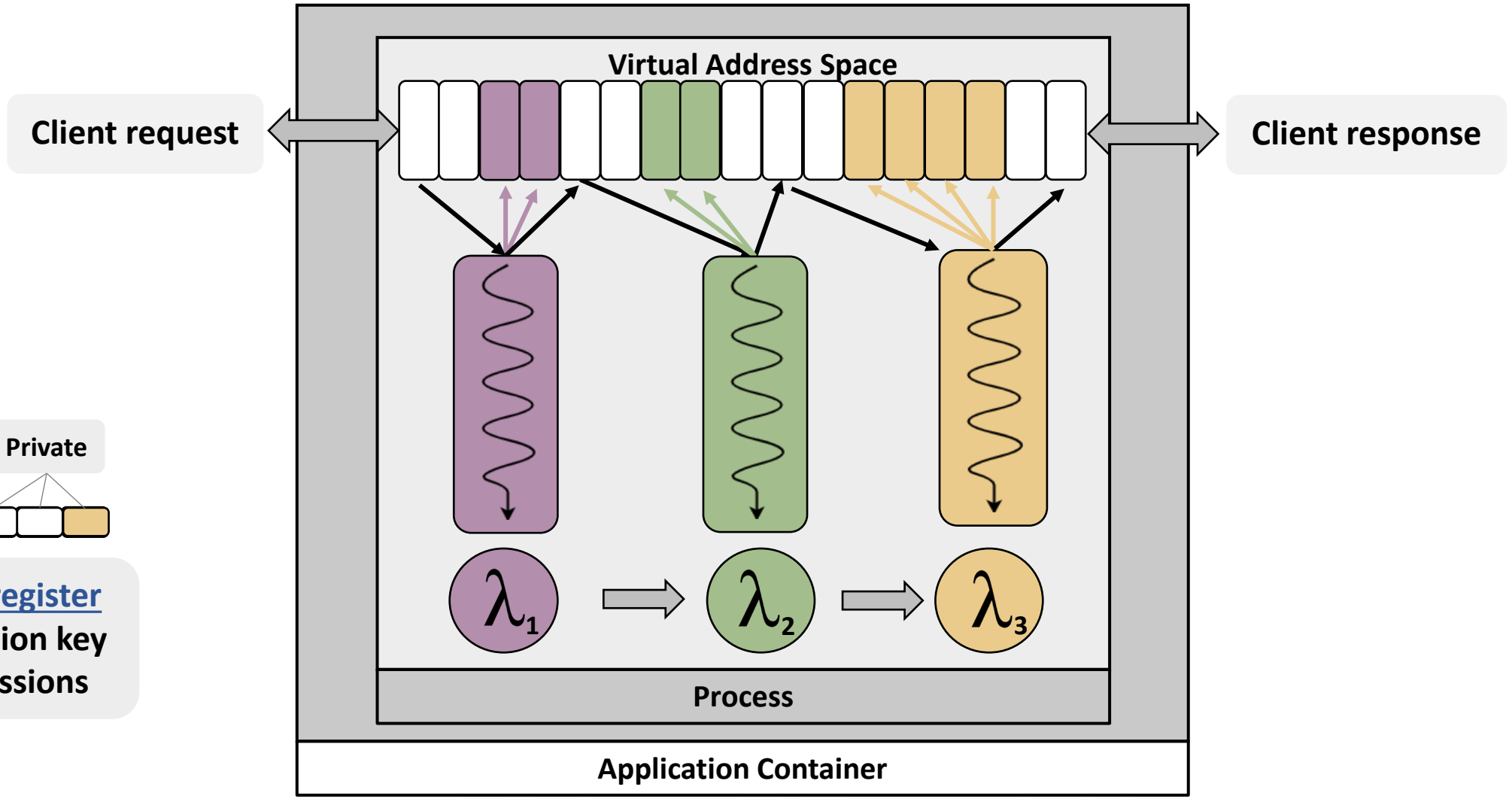
Solution 1: Lightweight isolation with Intel MPK



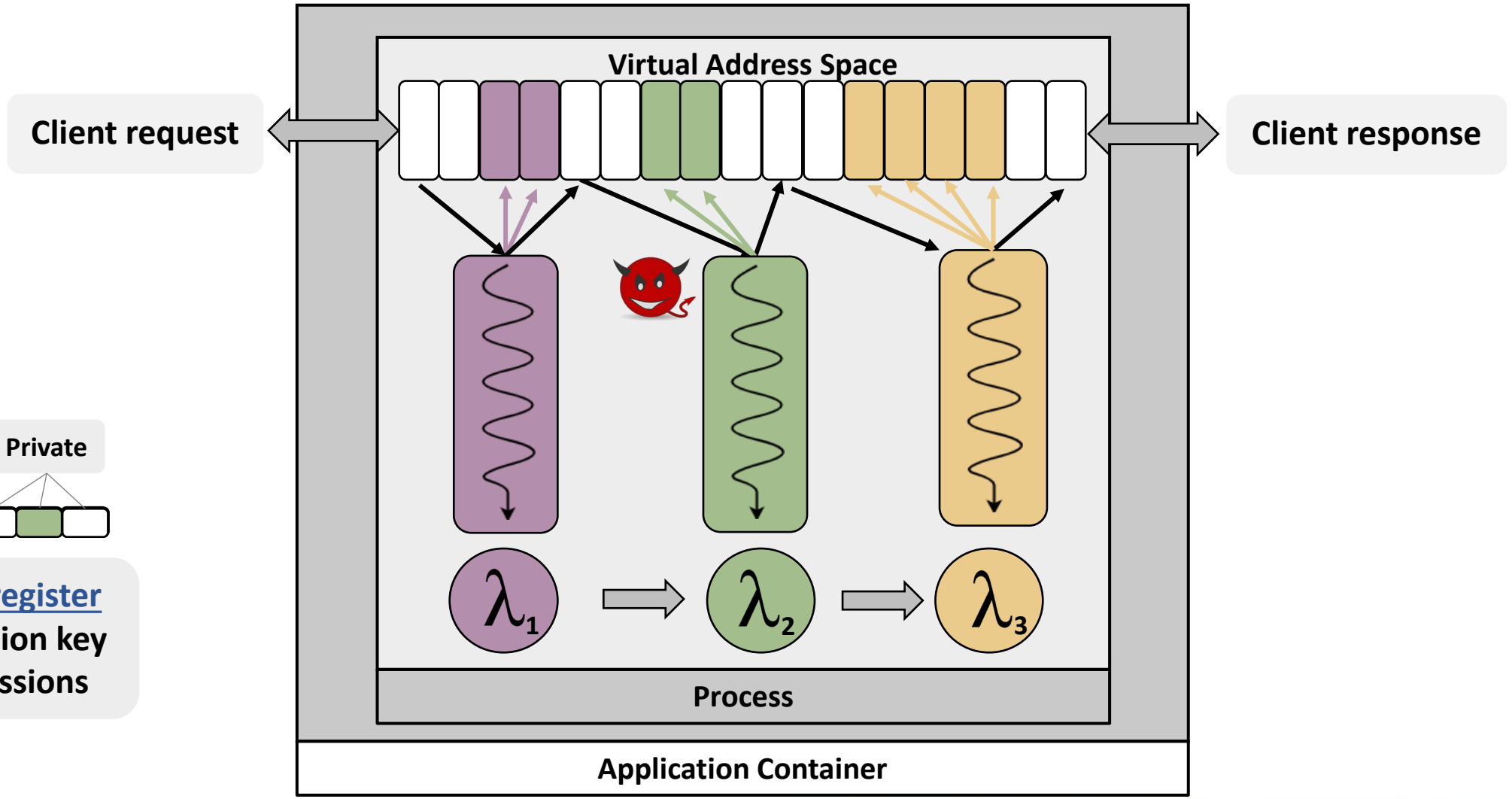


PKRU register
Protection key permissions

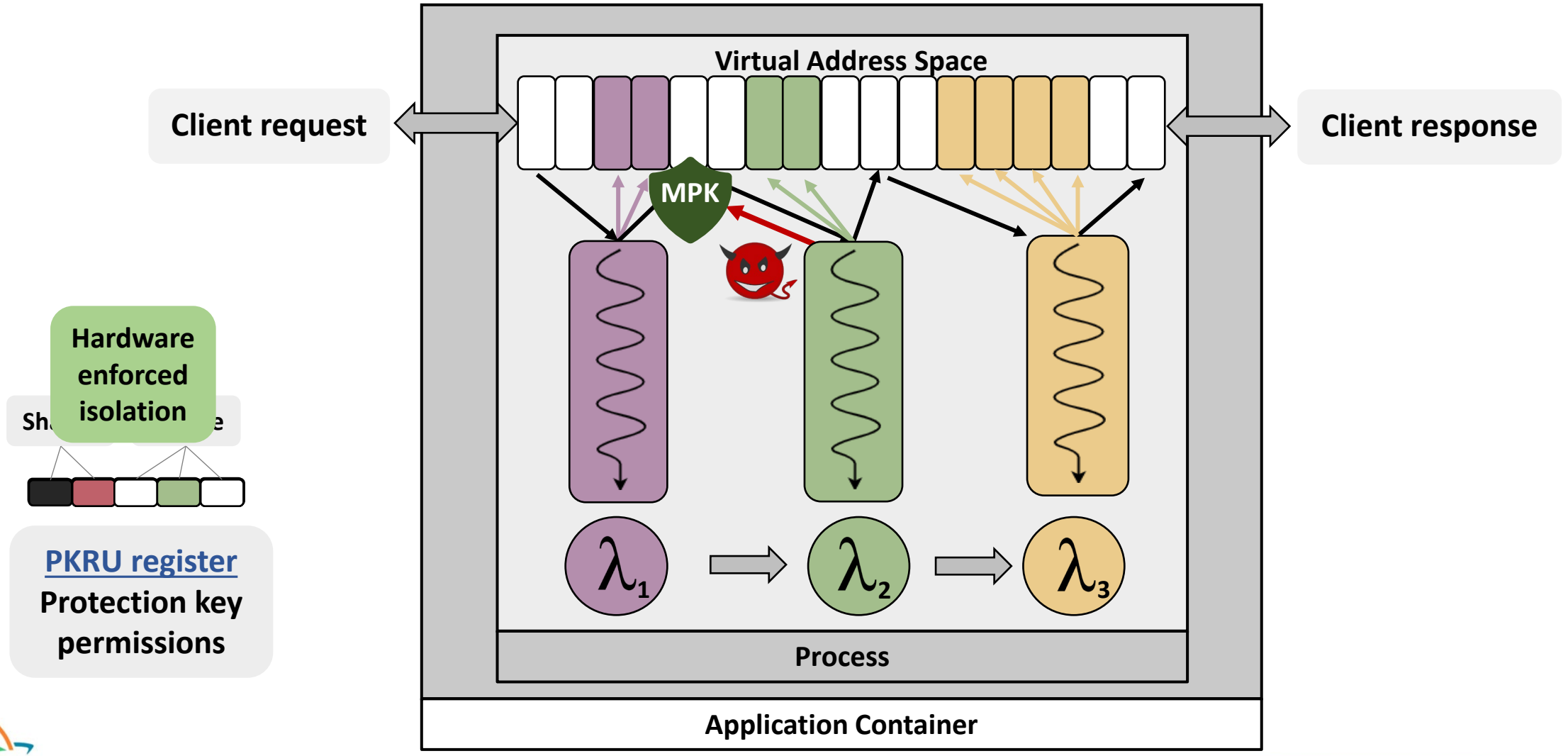
Solution 1: Lightweight isolation with Intel MPK



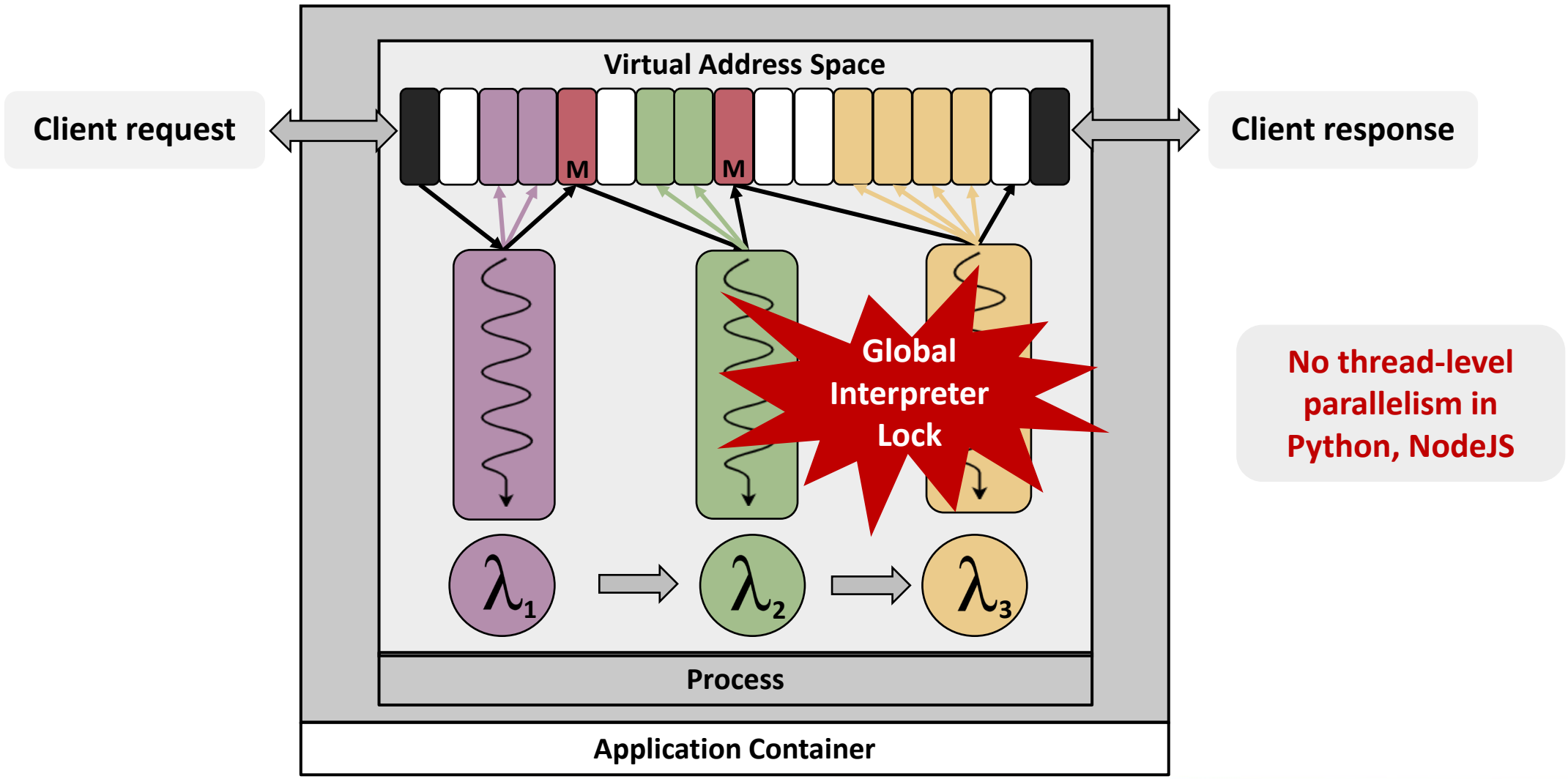
Solution 1: Lightweight isolation with Intel MPK



Solution 1: Lightweight isolation with Intel MPK

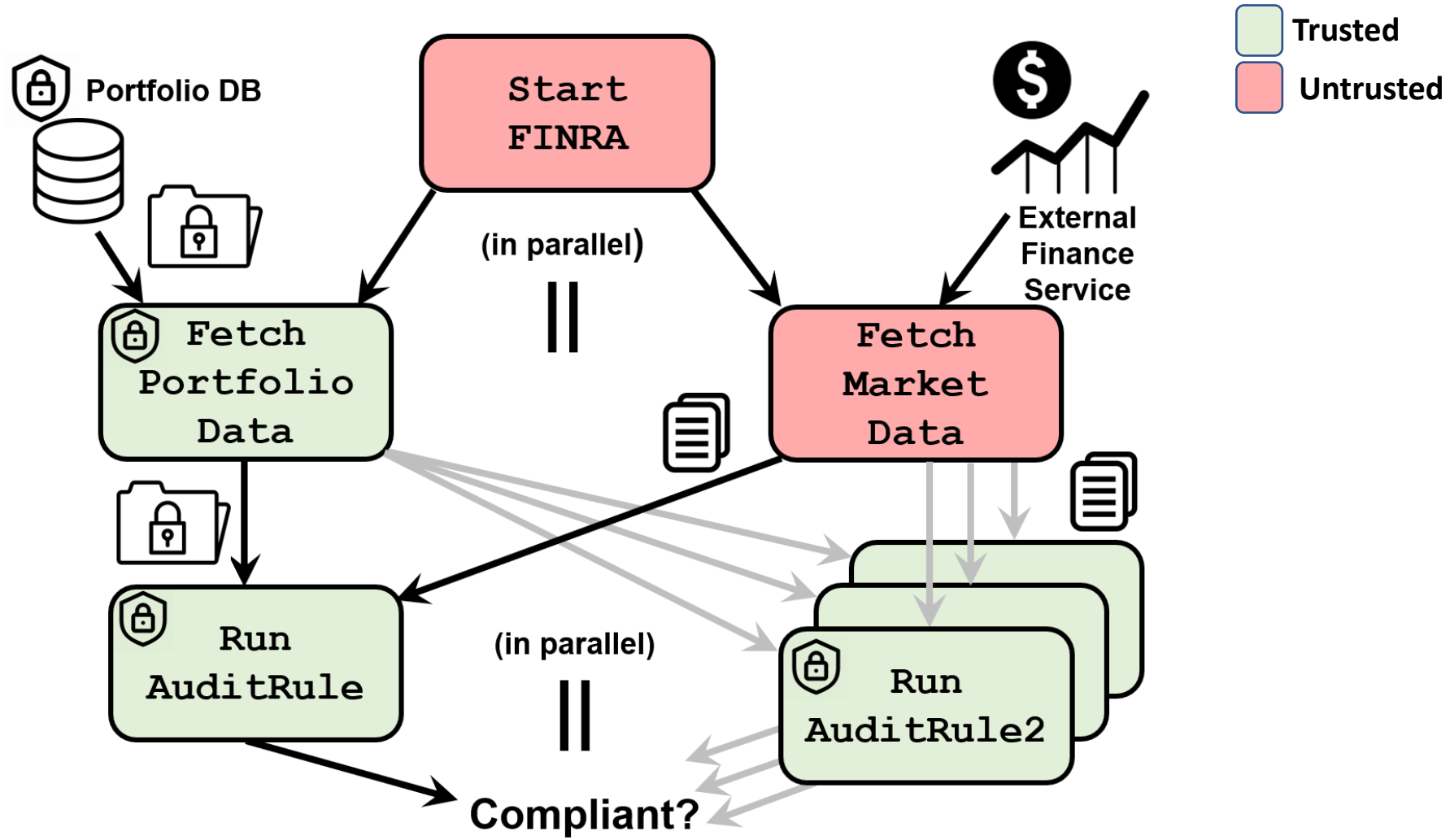


Challenge 2: No thread-level parallelism

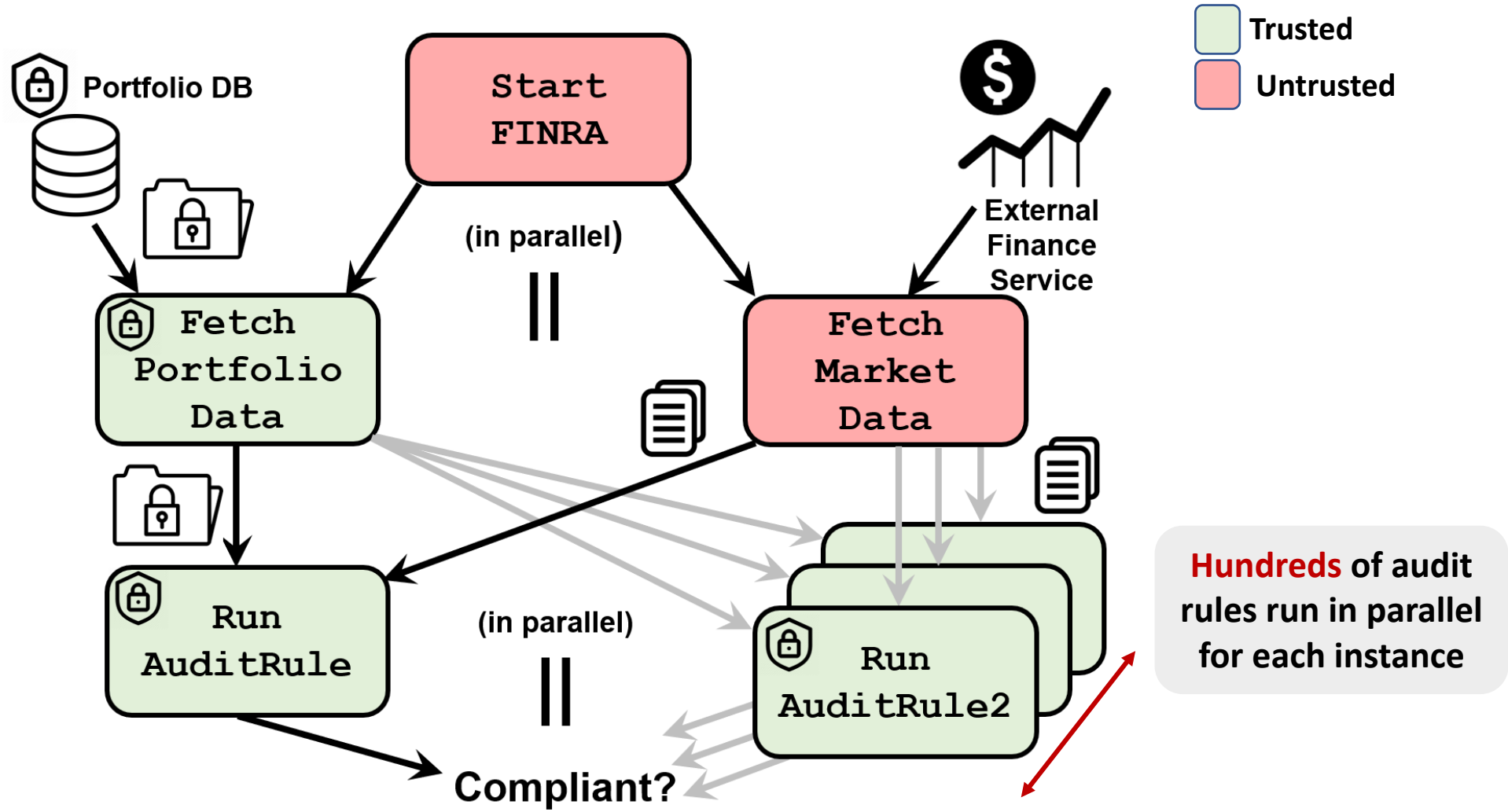


No thread-level parallelism in Python, NodeJS

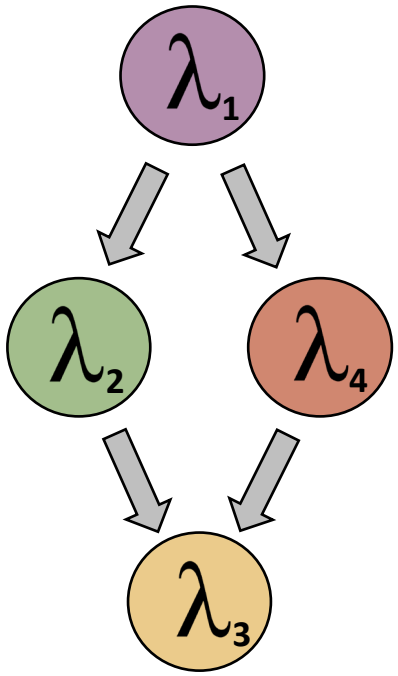
FINRA: Parallel functions in a workflow instance



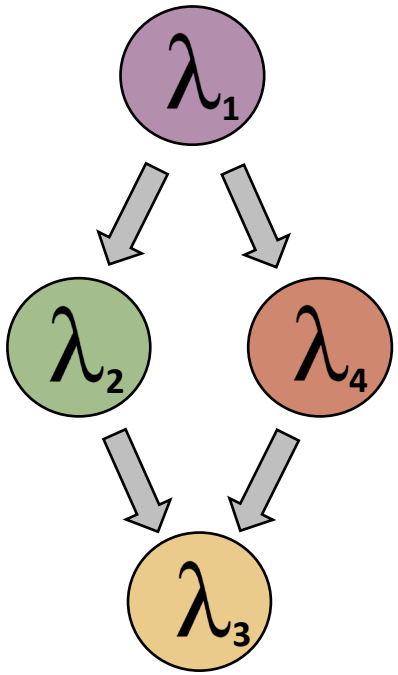
FINRA: Parallel functions in a workflow instance

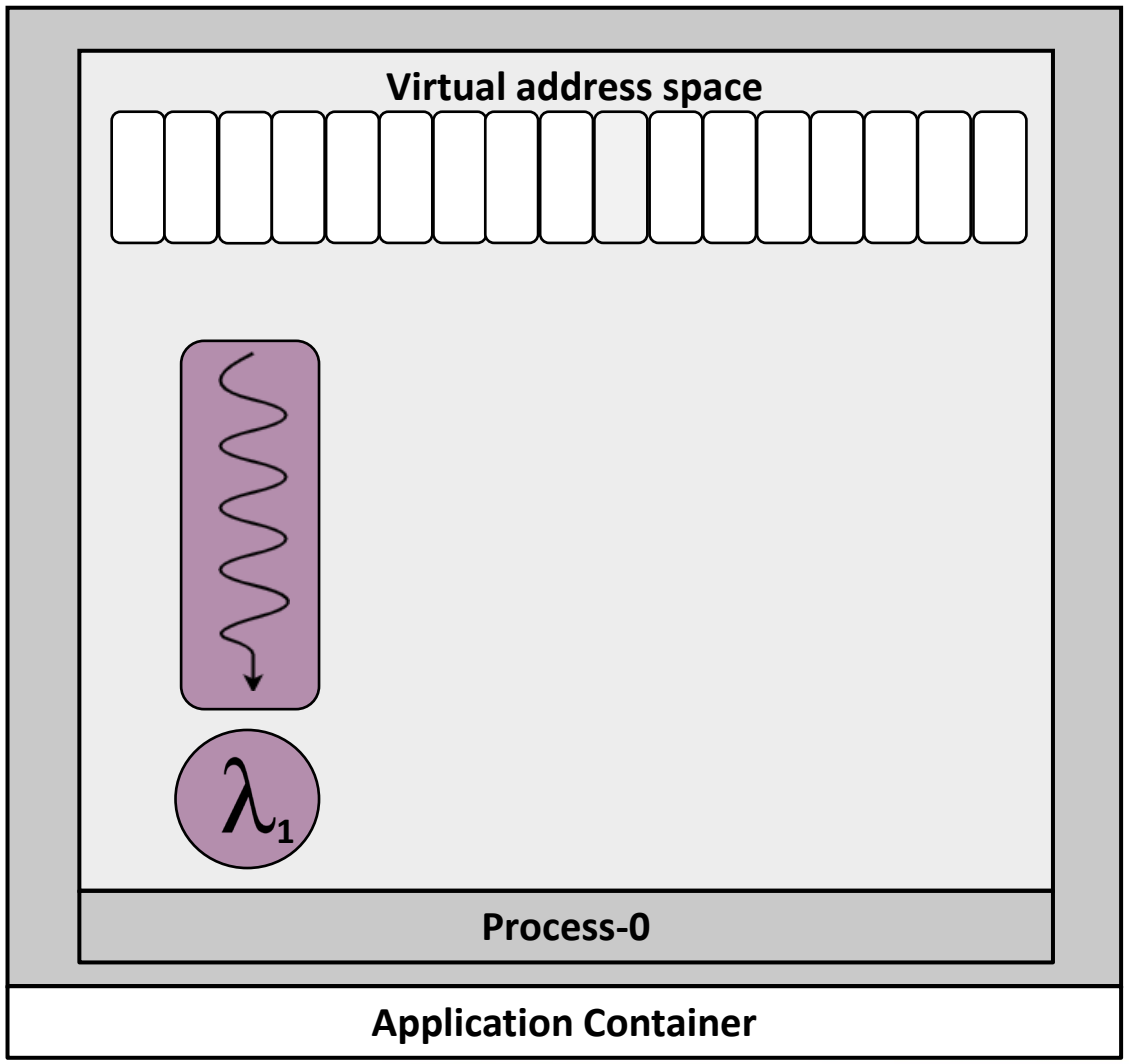


Solution 2: Dynamically switch to processes

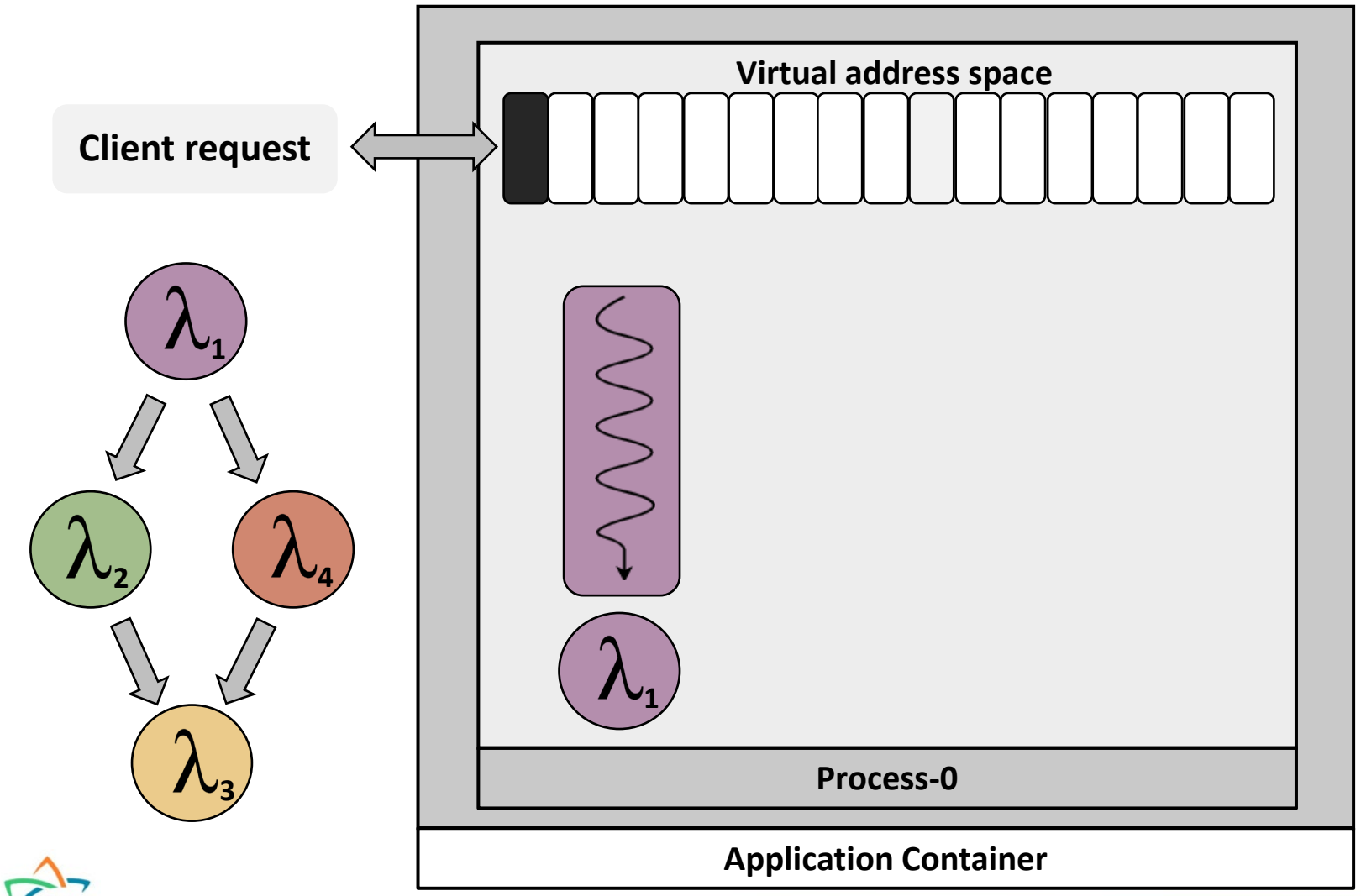


Solution 2: Dynamically switch to processes

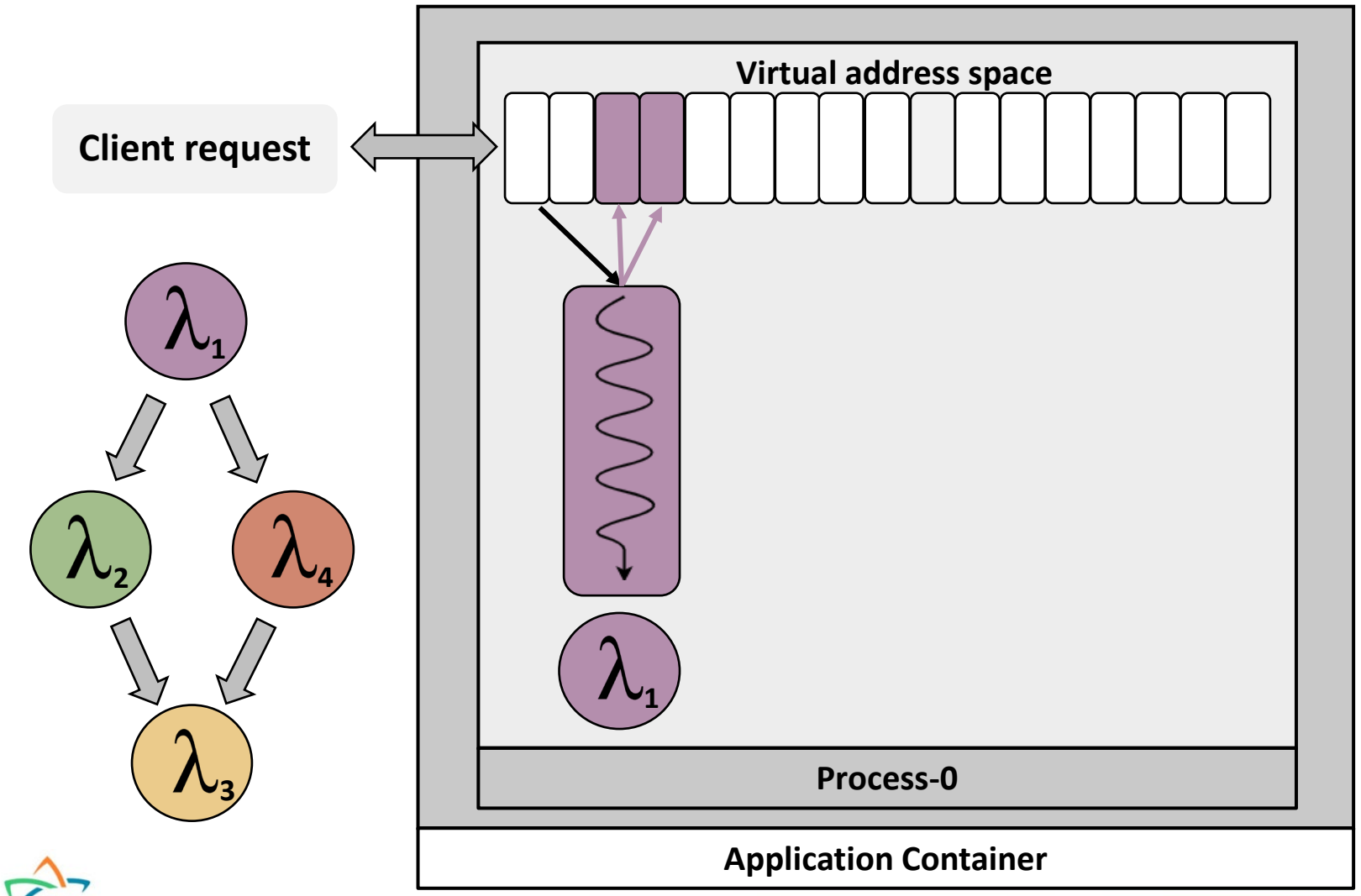




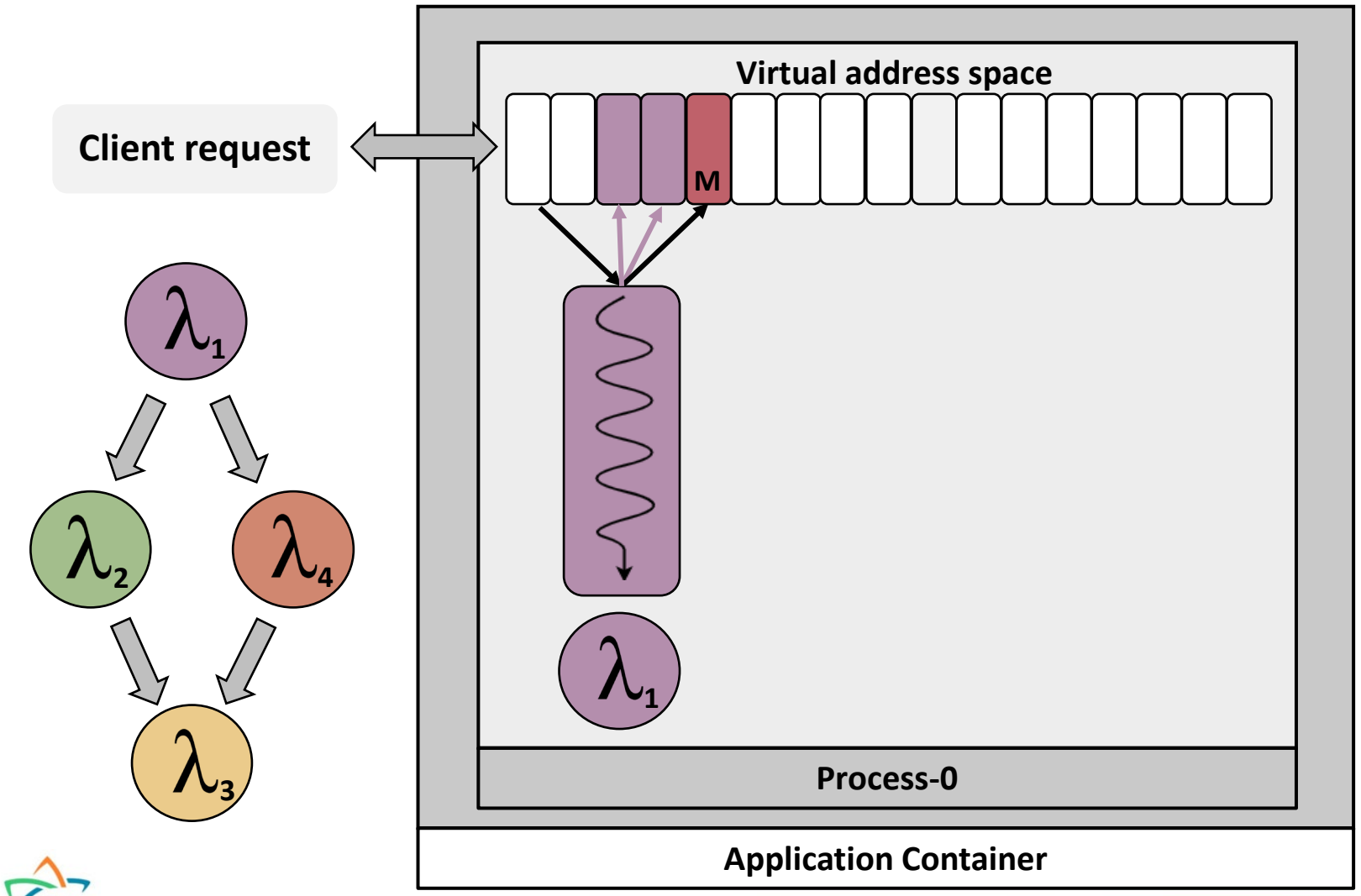
Solution 2: Dynamically switch to processes



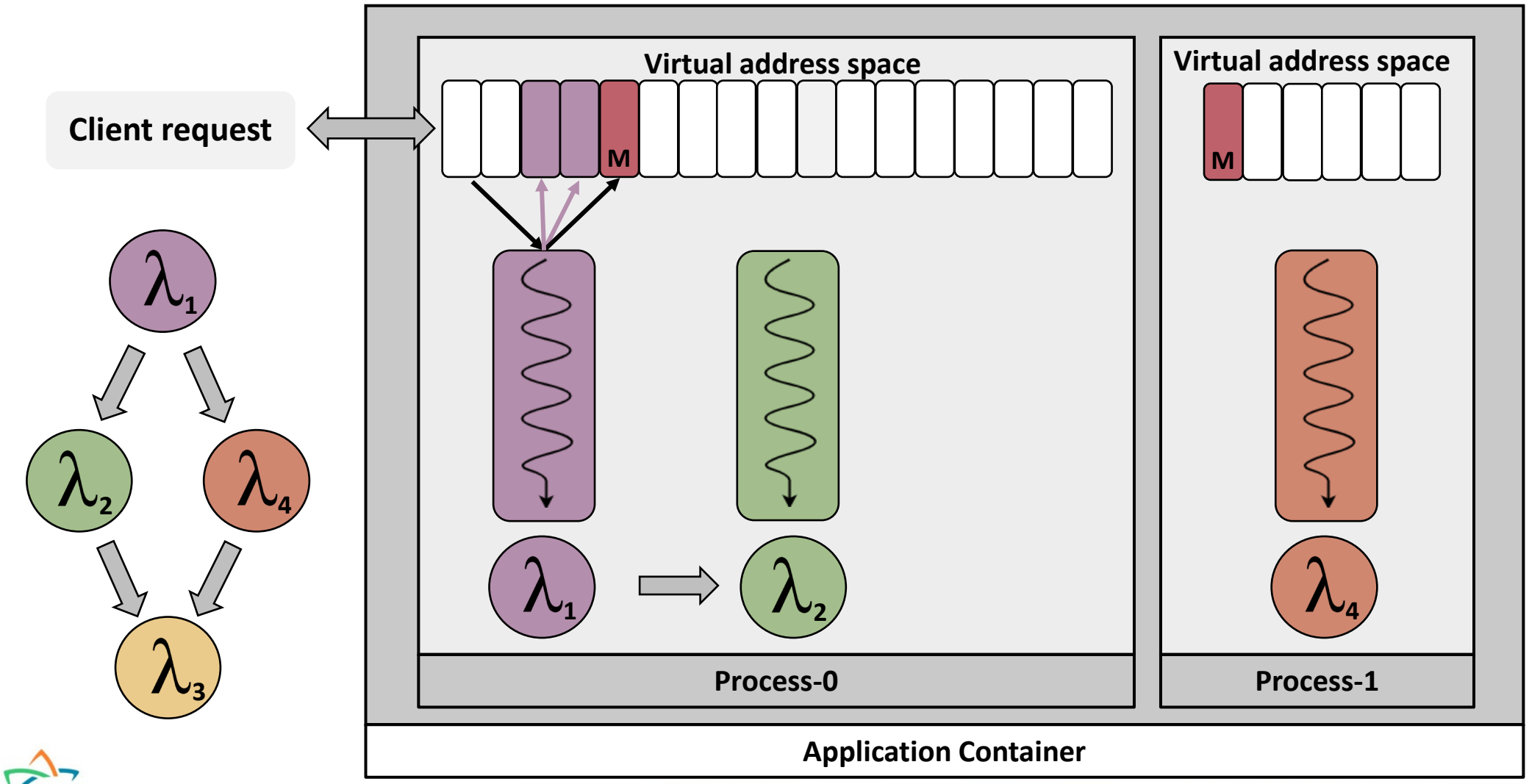
Solution 2: Dynamically switch to processes



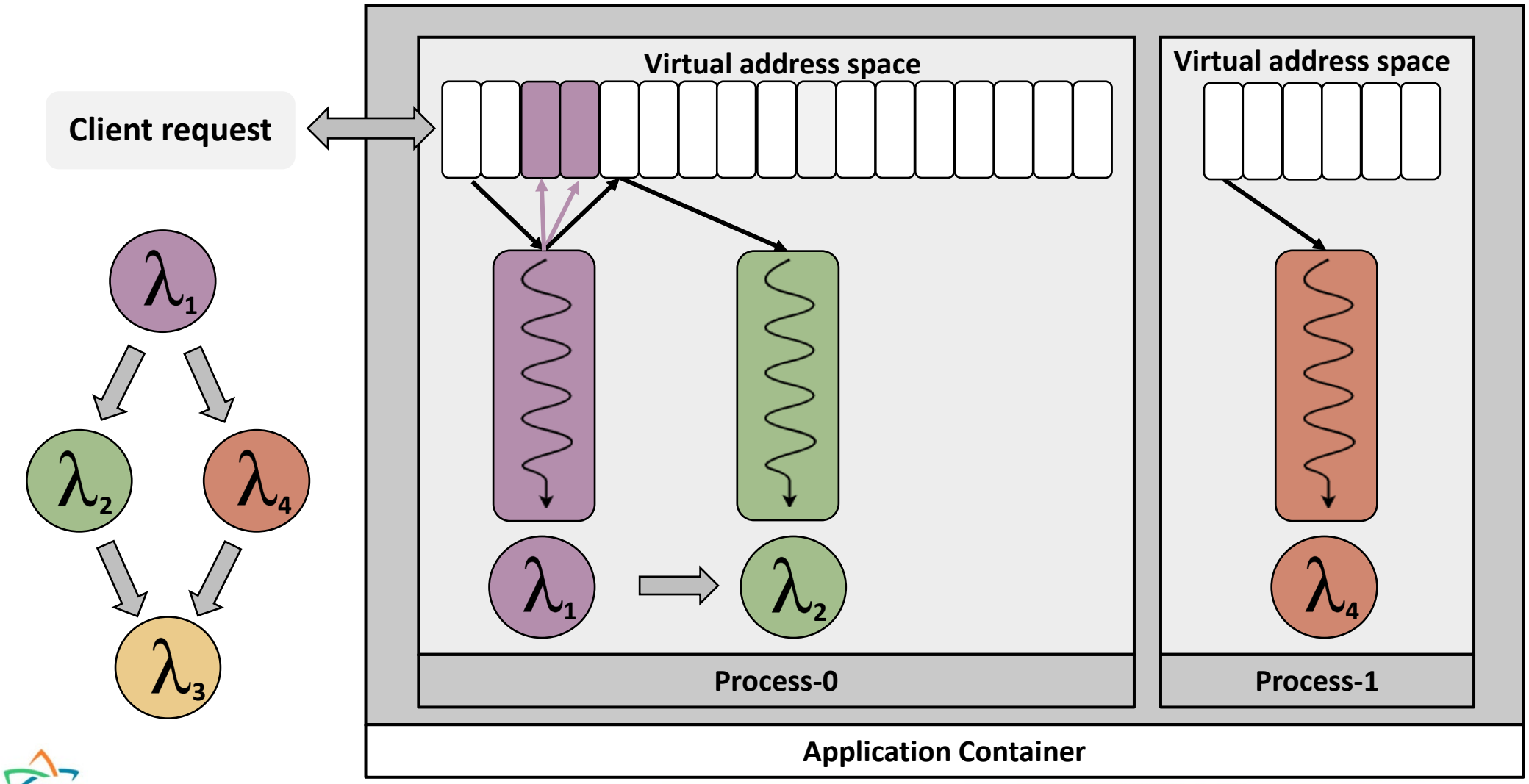
Solution 2: Dynamically switch to processes



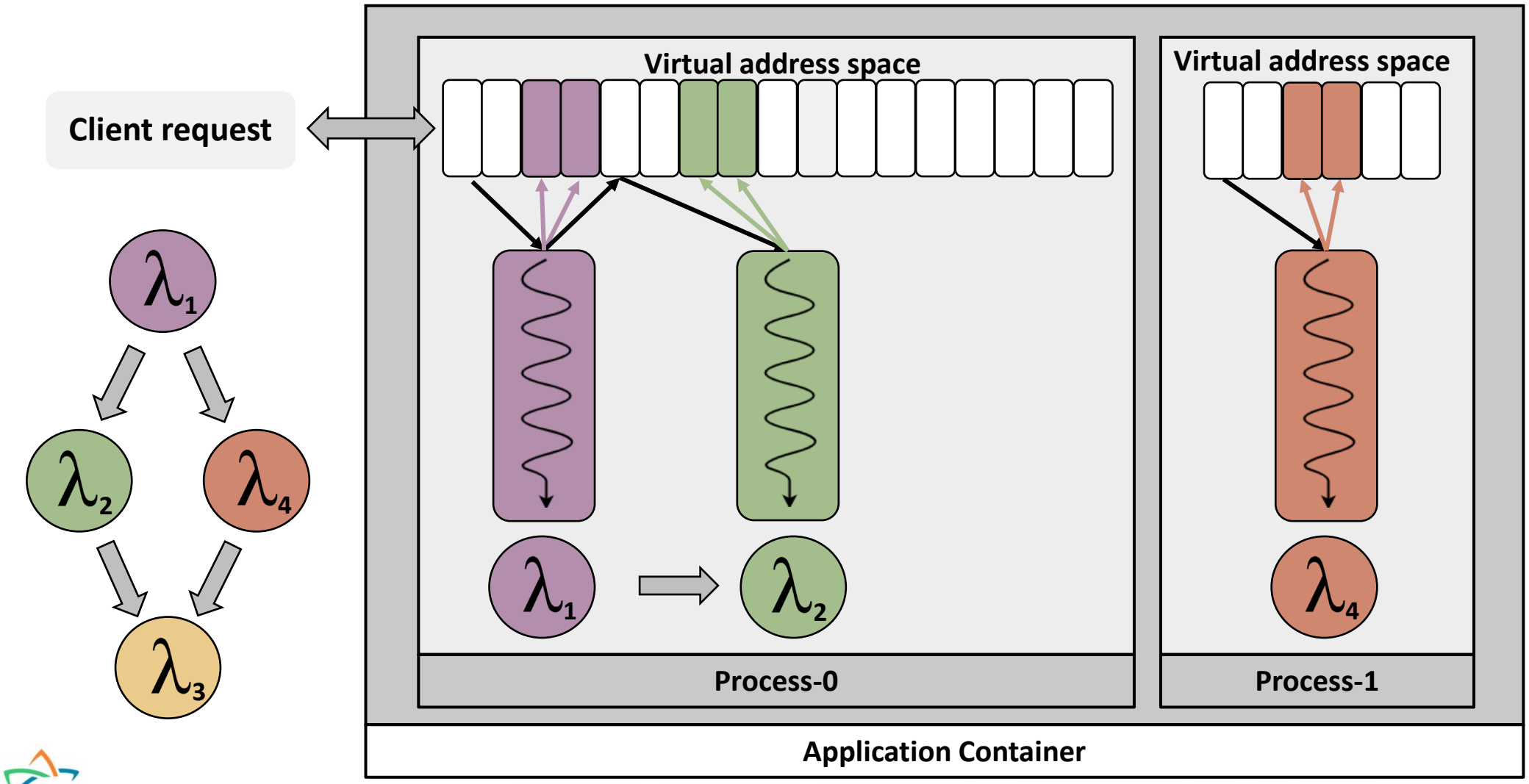
Solution 2: Dynamically switch to processes



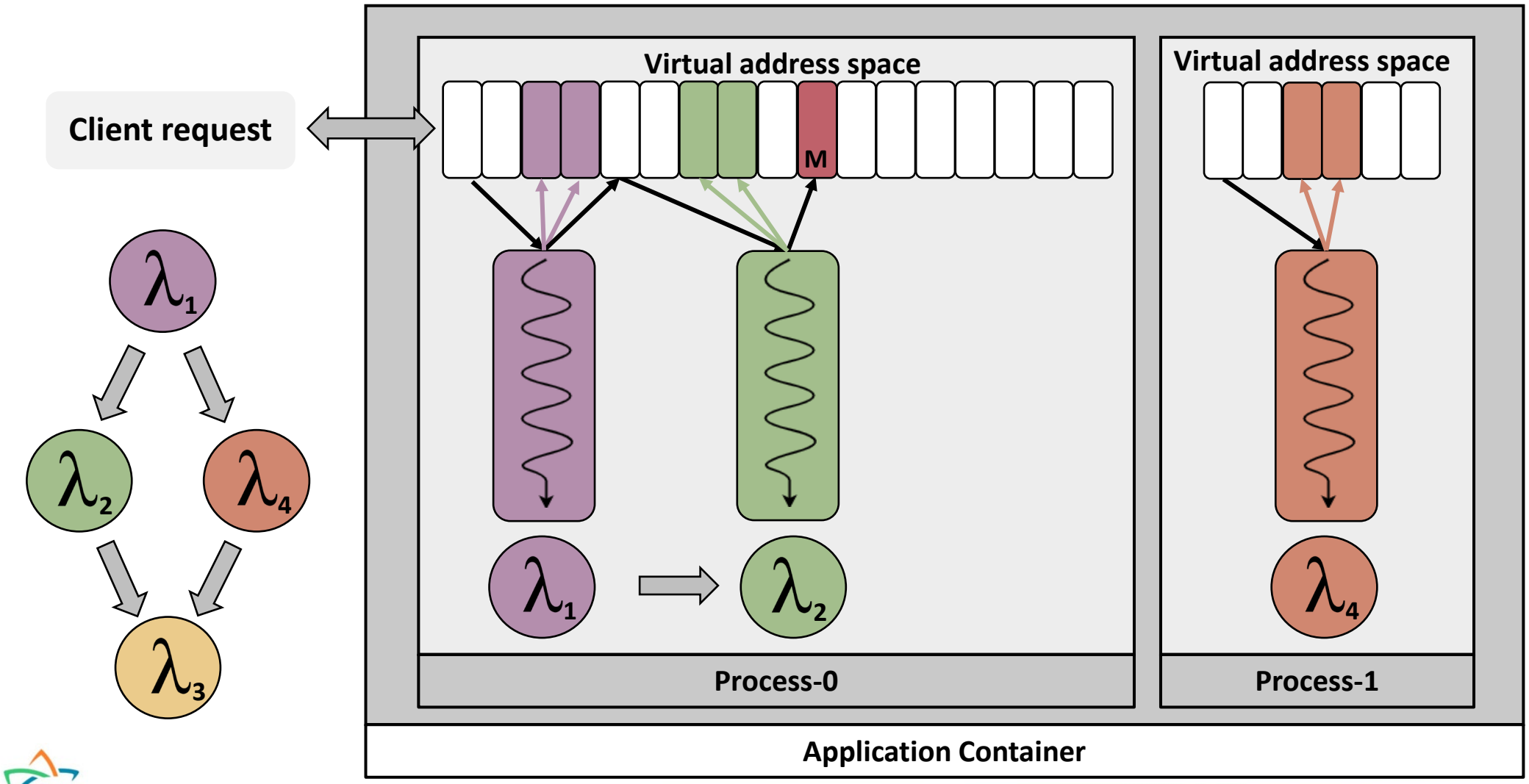
Solution 2: Dynamically switch to processes



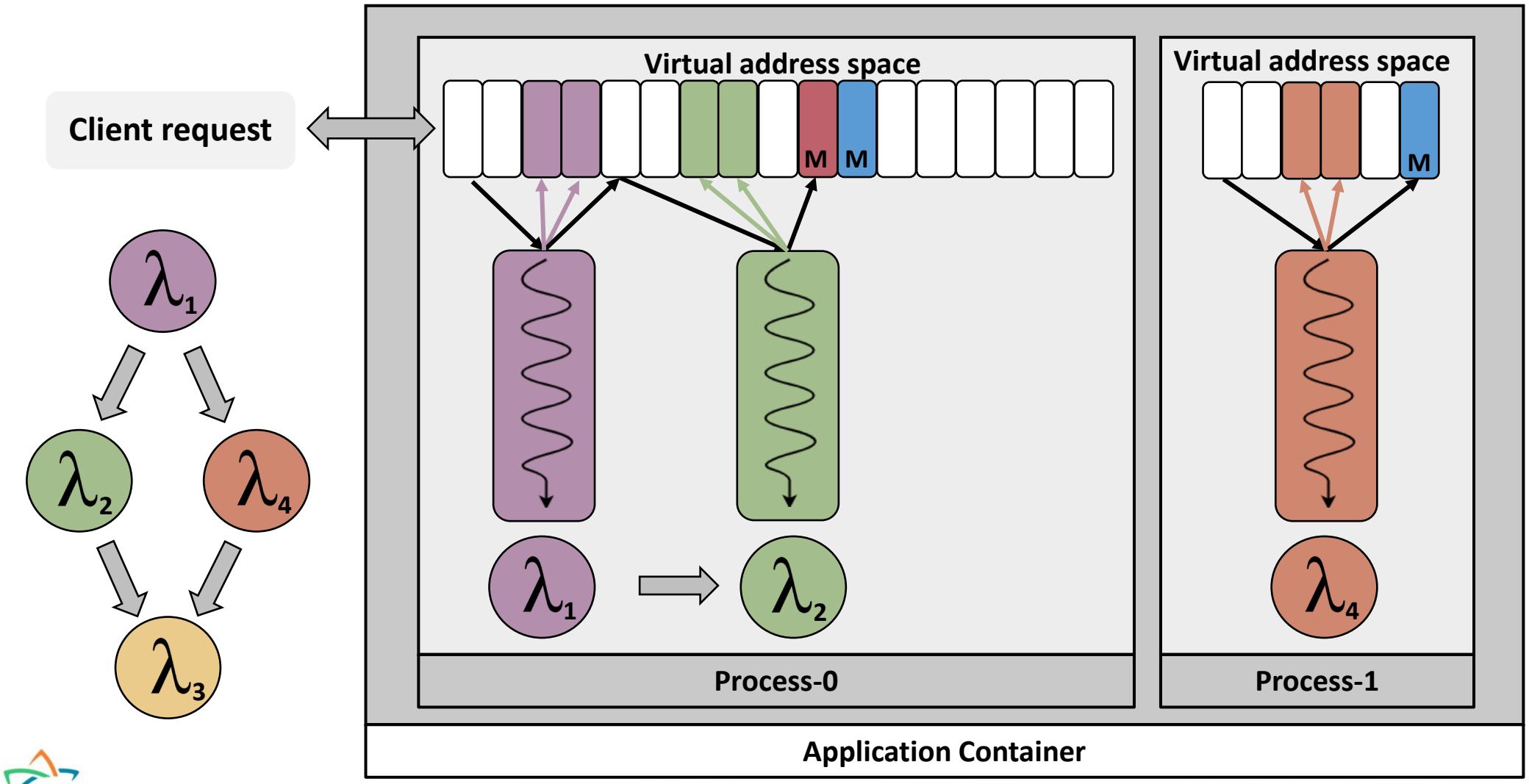
Solution 2: Dynamically switch to processes



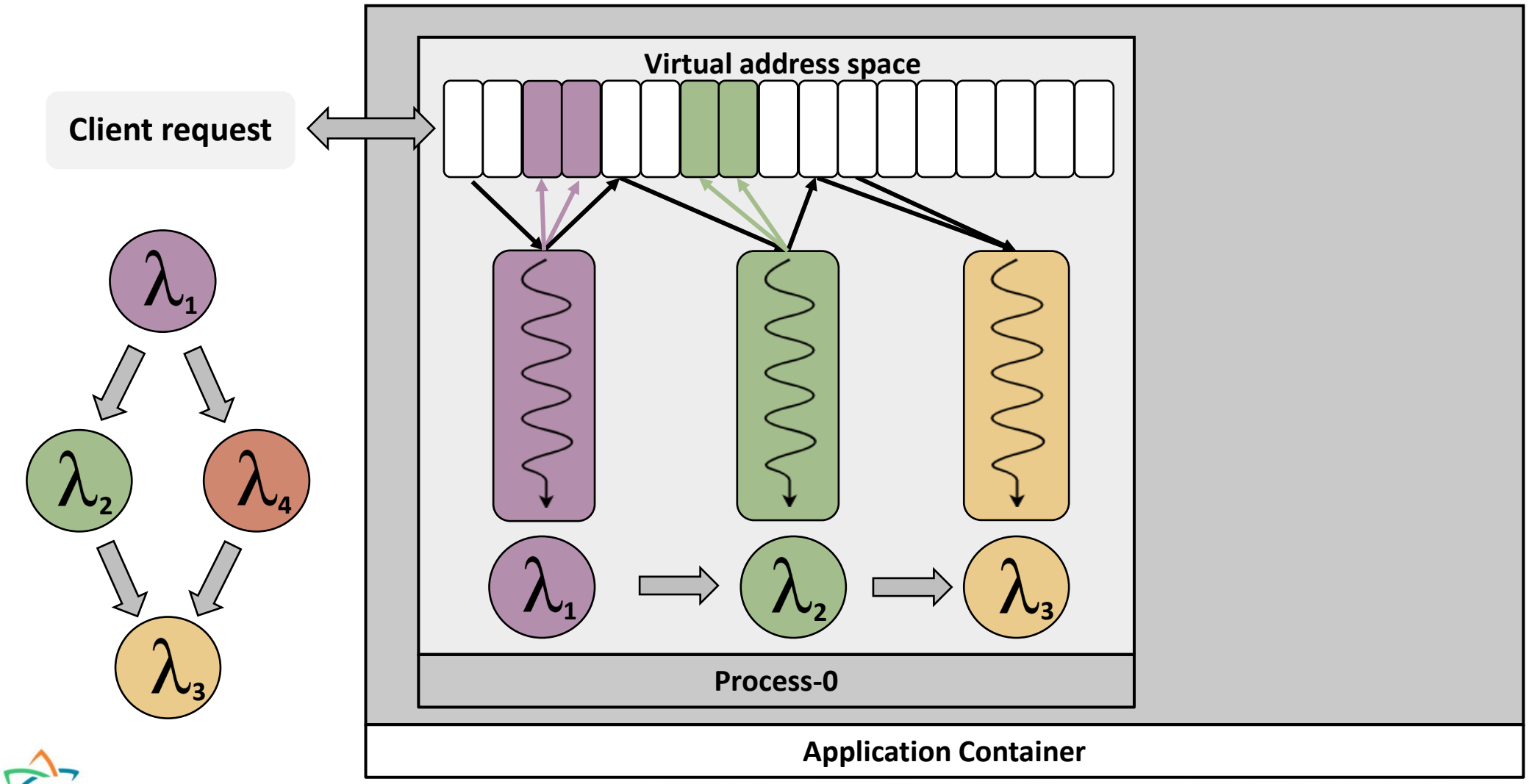
Solution 2: Dynamically switch to processes



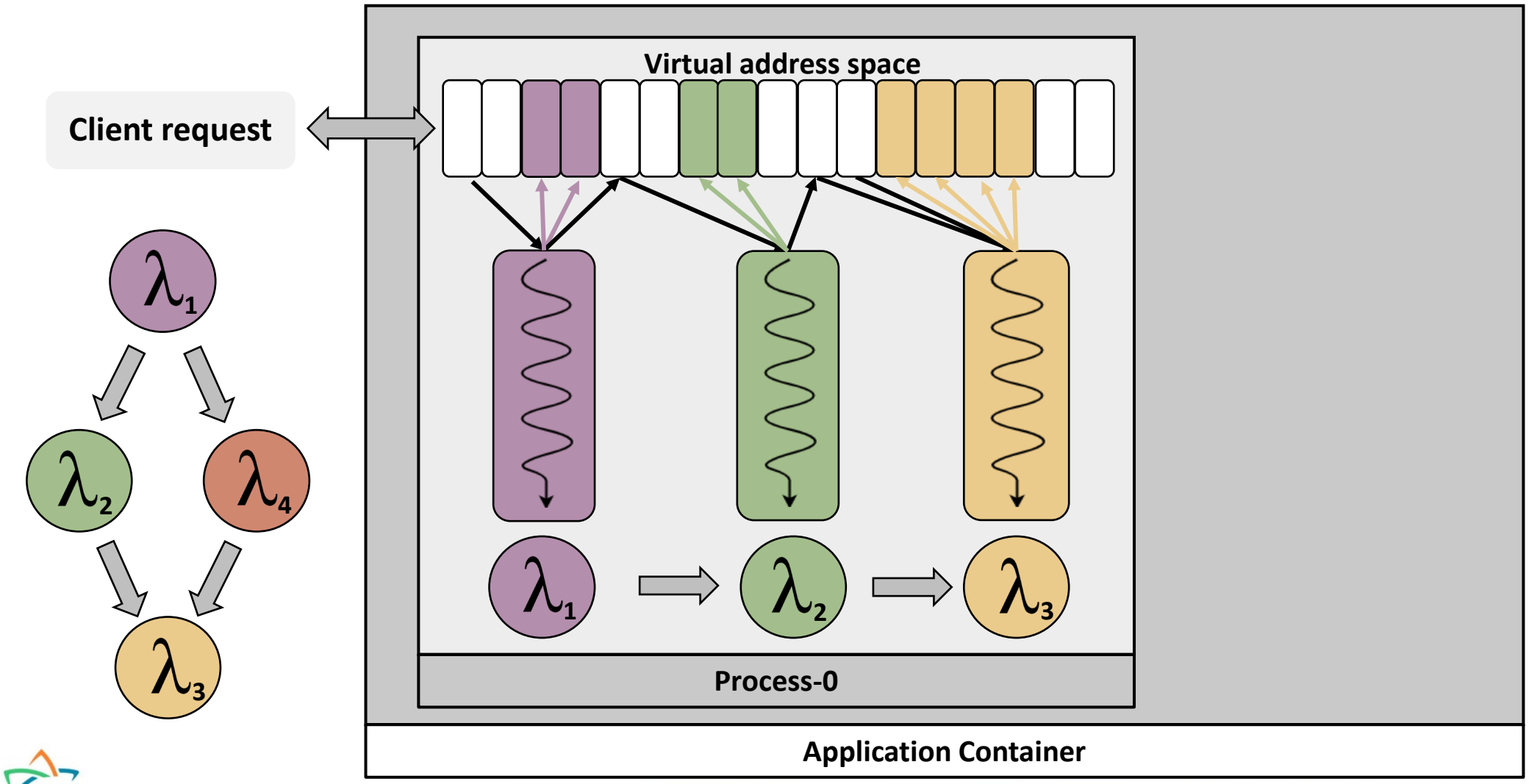
Solution 2: Dynamically switch to processes



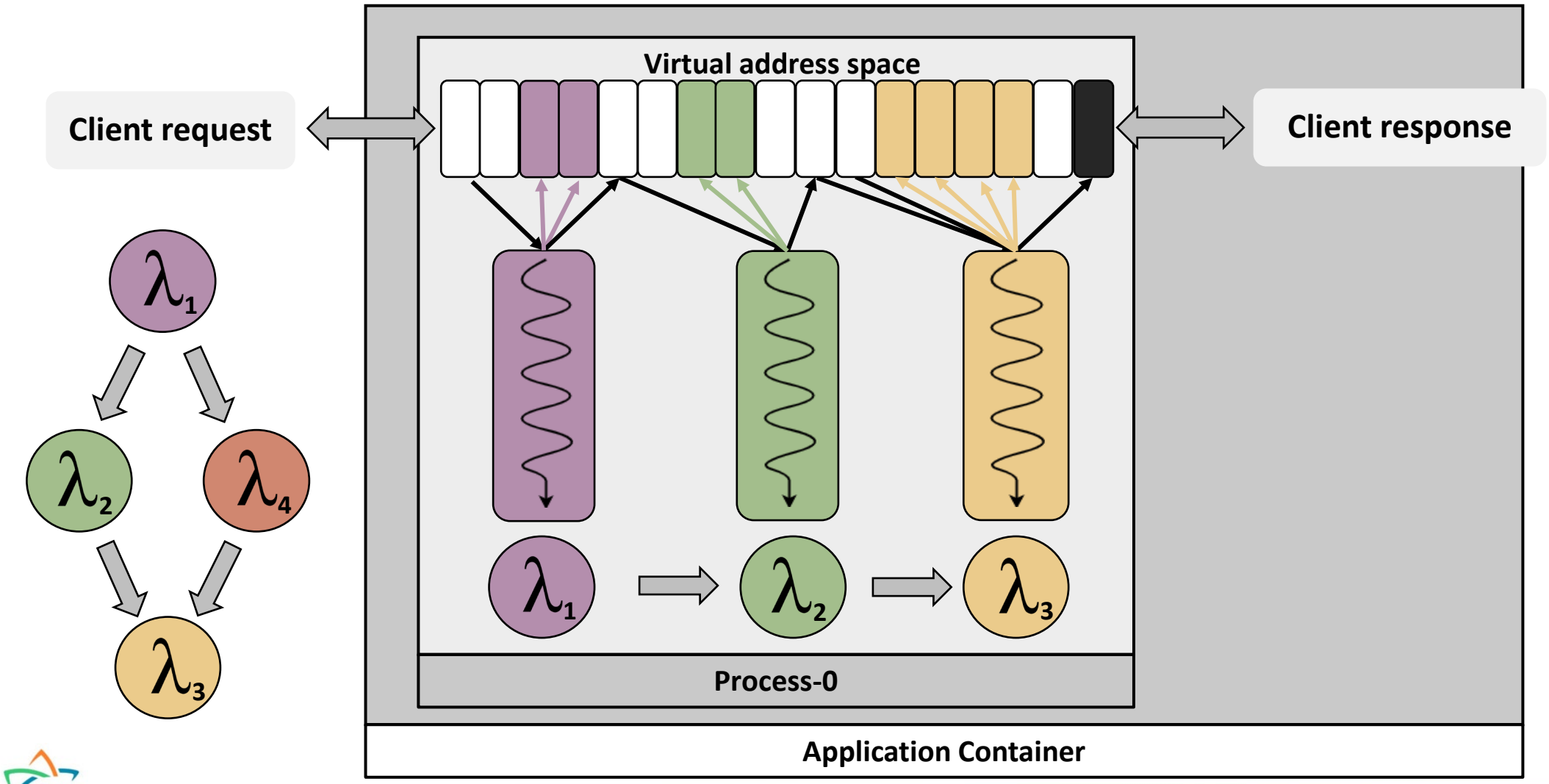
Solution 2: Dynamically switch to processes



Solution 2: Dynamically switch to processes

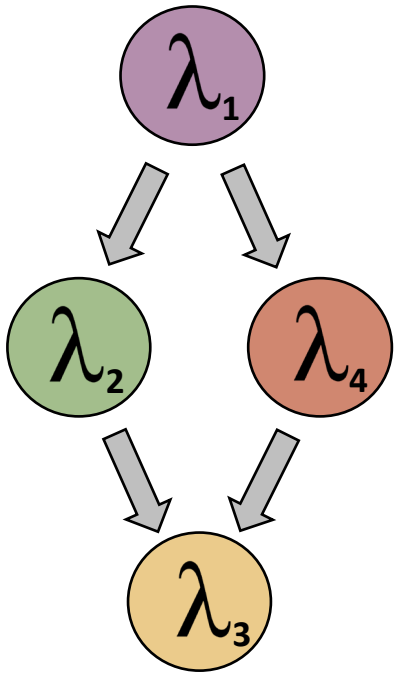


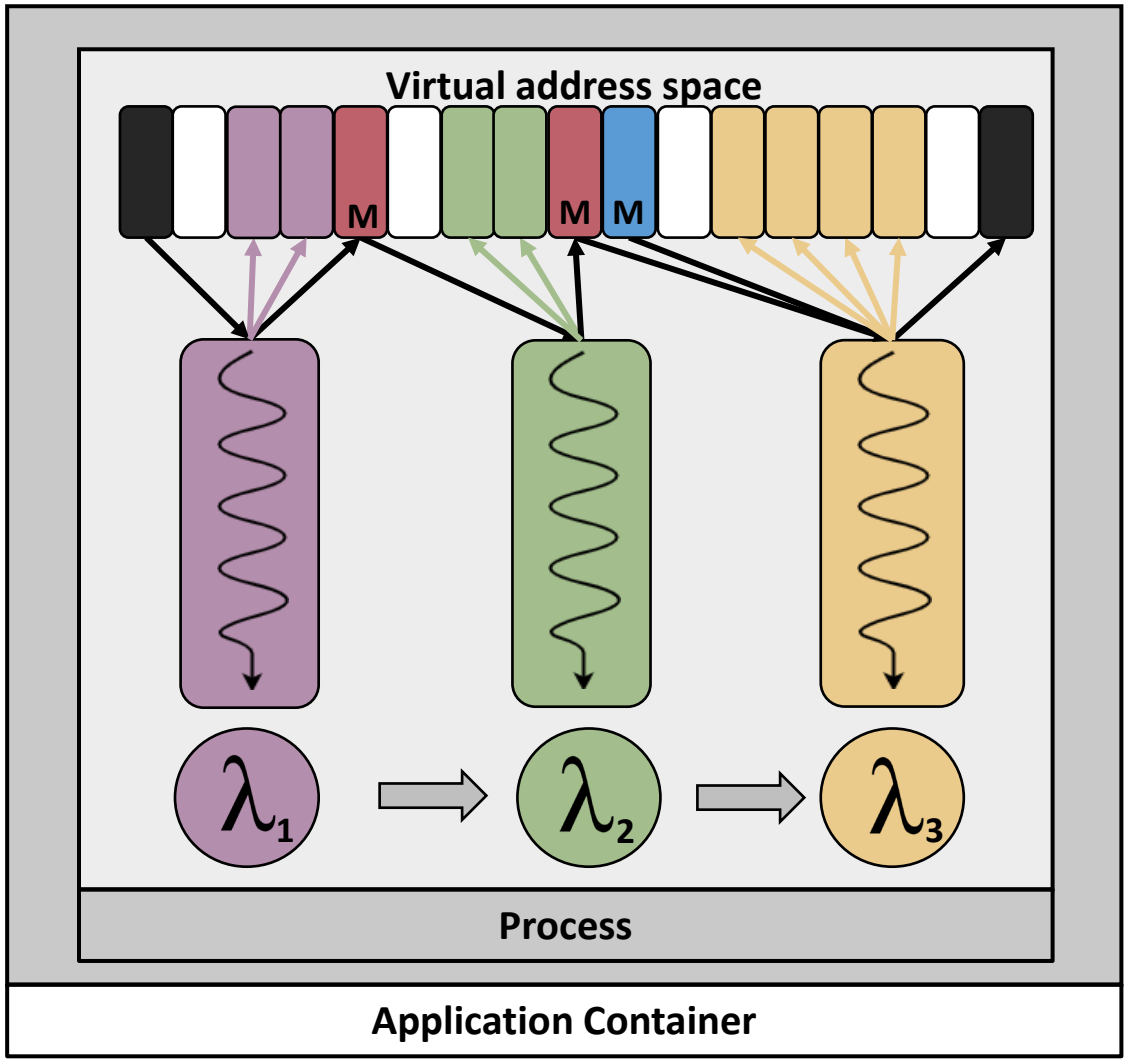
Solution 2: Dynamically switch to processes



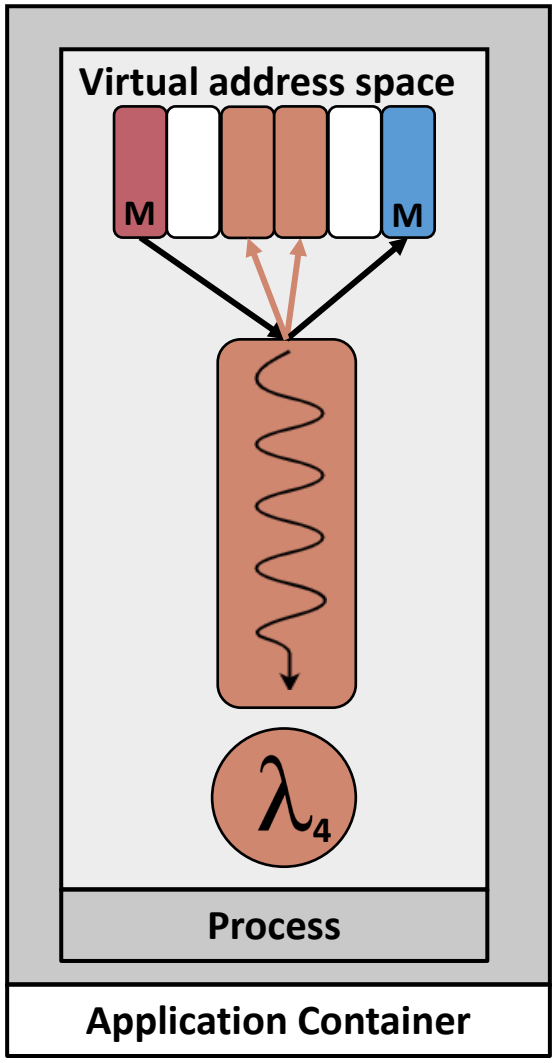
Solution 2: Smart scaling with containers

1 vCPU container



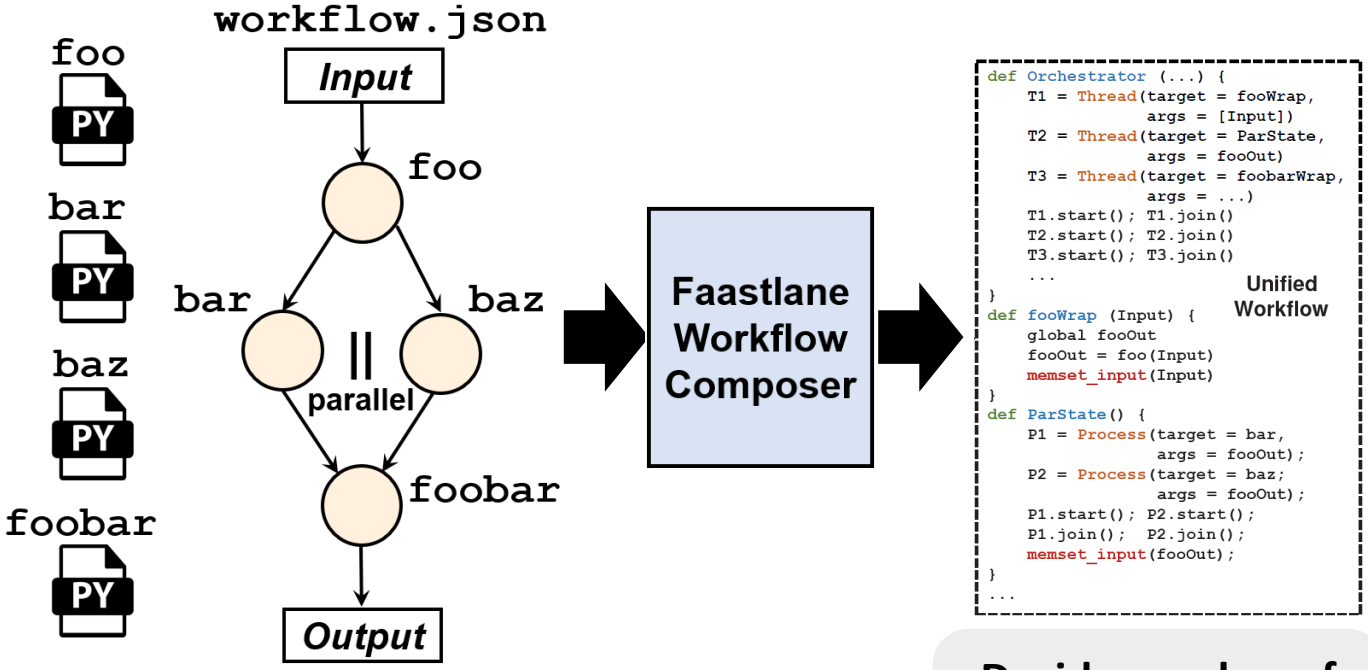


TCP



Faastlane: Putting it all together

Developer



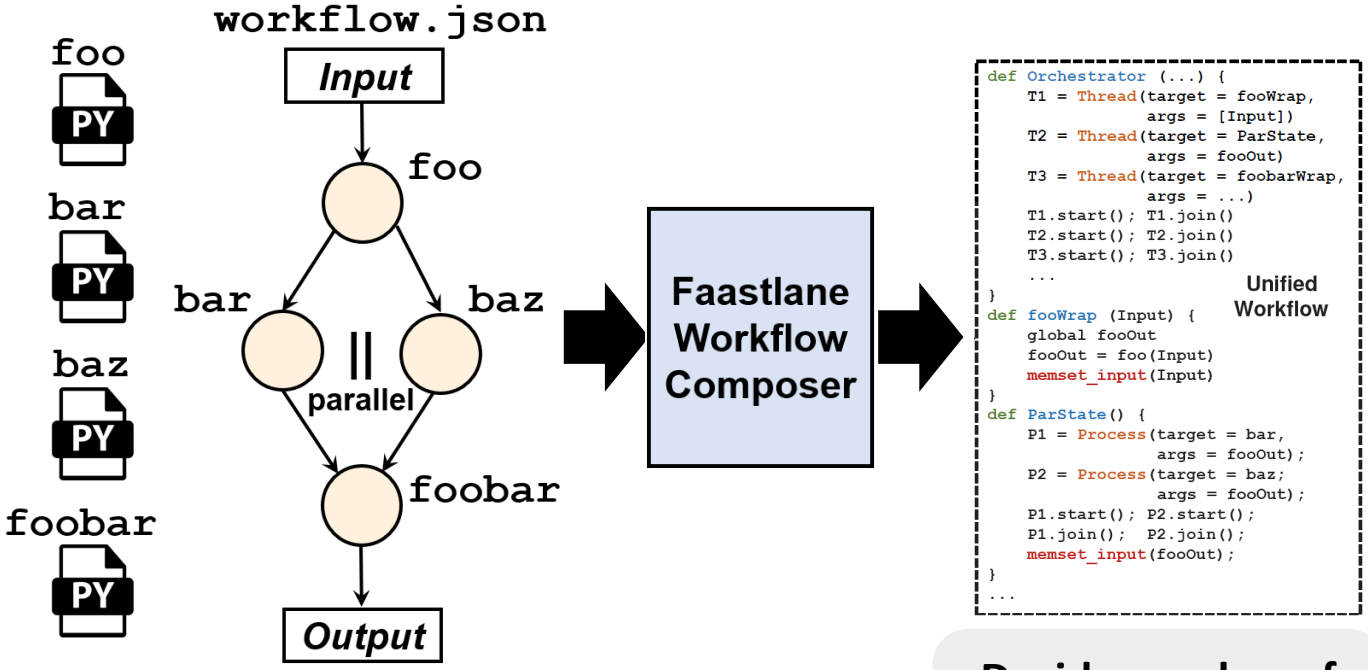

```
def Orchestrator (...) {
    T1 = Thread(target = fooWrap,
                args = [Input])
    T2 = Thread(target = ParState,
                args = fooOut)
    T3 = Thread(target = foobarWrap,
                args = ...)
    T1.start(); T1.join()
    T2.start(); T2.join()
    T3.start(); T3.join()
    ...
}
def fooWrap (Input) {
    global fooOut
    fooOut = foo(Input)
    memset_input(Input)
}
def ParState() {
    P1 = Process(target = bar,
                 args = fooOut);
    P2 = Process(target = baz;
                 args = fooOut);
    P1.start(); P2.start();
    P1.join();  P2.join();
    memset_input(fooOut);
}
...
```

Unified Workflow

Decide number of threads, processes and containers

Faastlane: Putting it all together

Developer



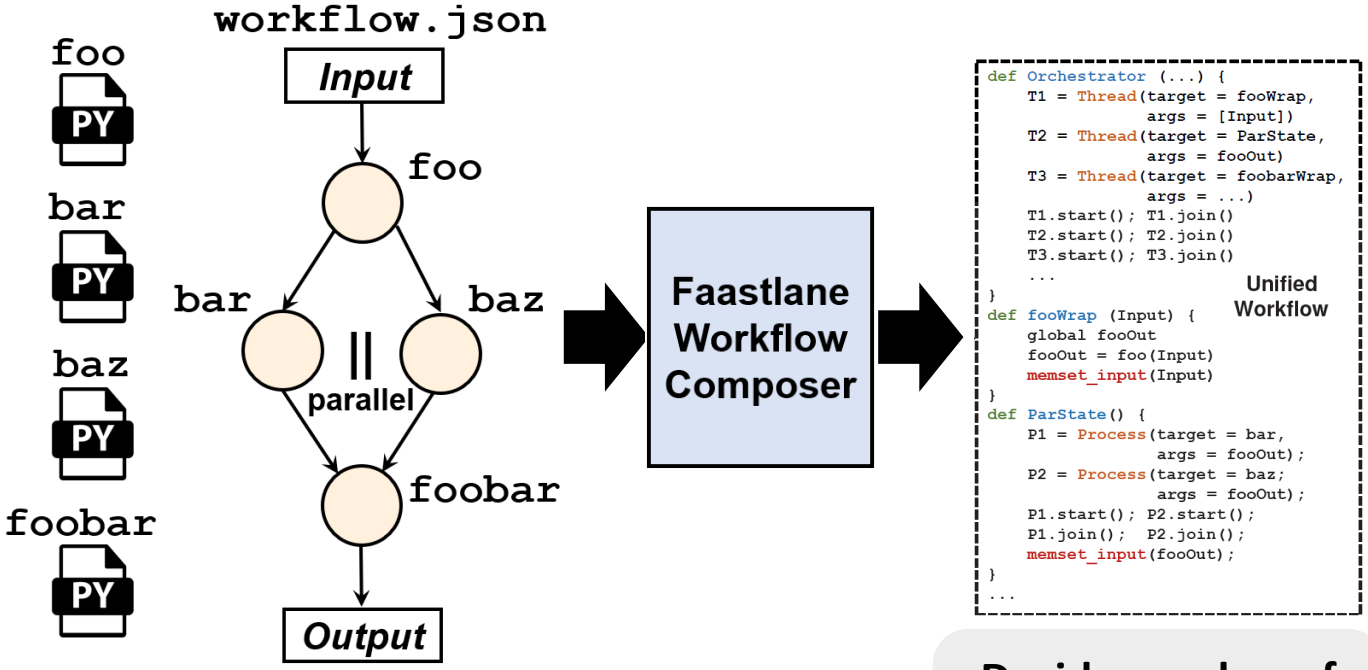

```
def Orchestrator (...) {
    T1 = Thread(target = fooWrap,
                args = [Input])
    T2 = Thread(target = ParState,
                args = fooOut)
    T3 = Thread(target = foobarWrap,
                args = ...)
    T1.start(); T1.join()
    T2.start(); T2.join()
    T3.start(); T3.join()
    ...
}
def fooWrap (Input) {
    global fooOut
    fooOut = foo(Input)
    memset_input(Input)
}
def ParState() {
    P1 = Process(target = bar,
                 args = fooOut);
    P2 = Process(target = baz;
                 args = fooOut);
    P1.start(); P2.start();
    P1.join();  P2.join();
    memset_input(fooOut);
}
...
```

Unified Workflow

Decide number of threads, processes and containers

Faastlane: Putting it all together

Developer



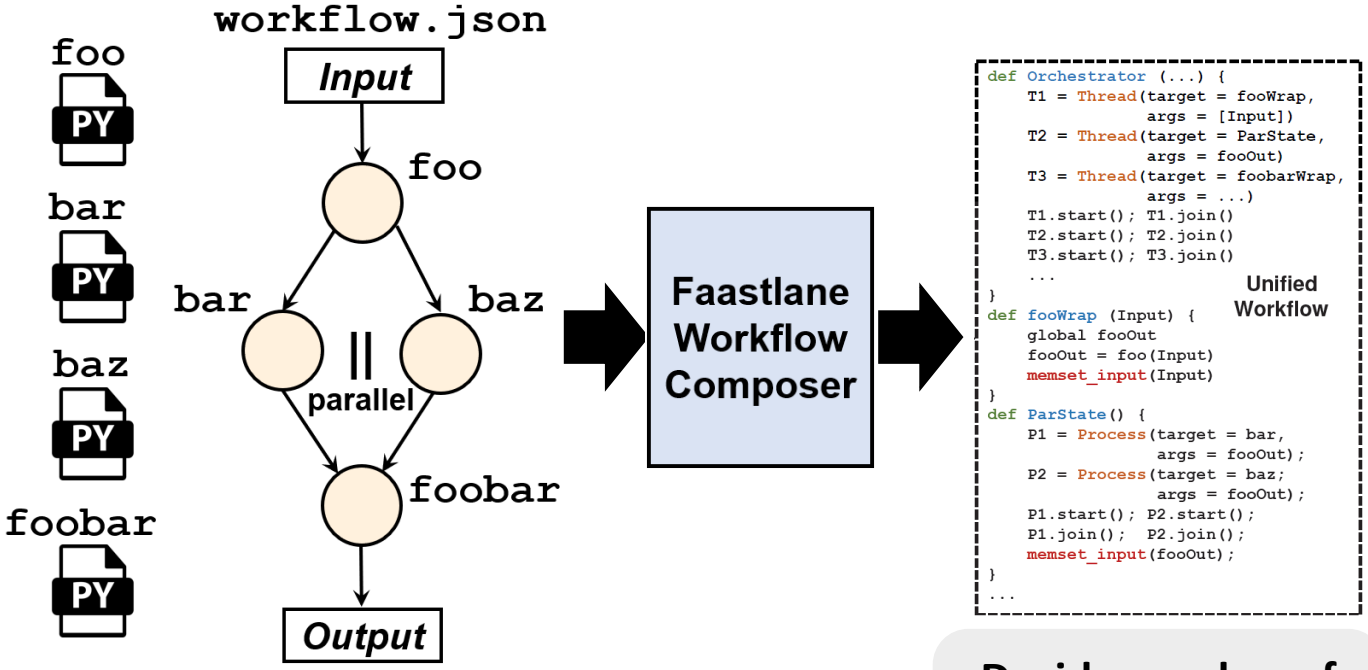

```
def Orchestrator (...) {
    T1 = Thread(target = fooWrap,
                args = [Input])
    T2 = Thread(target = ParState,
                args = fooOut)
    T3 = Thread(target = foobarWrap,
                args = ...)
    T1.start(); T1.join()
    T2.start(); T2.join()
    T3.start(); T3.join()
    ...
}
def fooWrap (Input) {
    global fooOut
    fooOut = foo(Input)
    memset_input(Input)
}
def ParState() {
    P1 = Process(target = bar,
                 args = fooOut);
    P2 = Process(target = baz;
                 args = fooOut);
    P1.start(); P2.start();
    P1.join();  P2.join();
    memset_input(fooOut);
}
...
```

Unified Workflow

Decide number of threads, processes and containers

Faastlane: Putting it all together

Developer



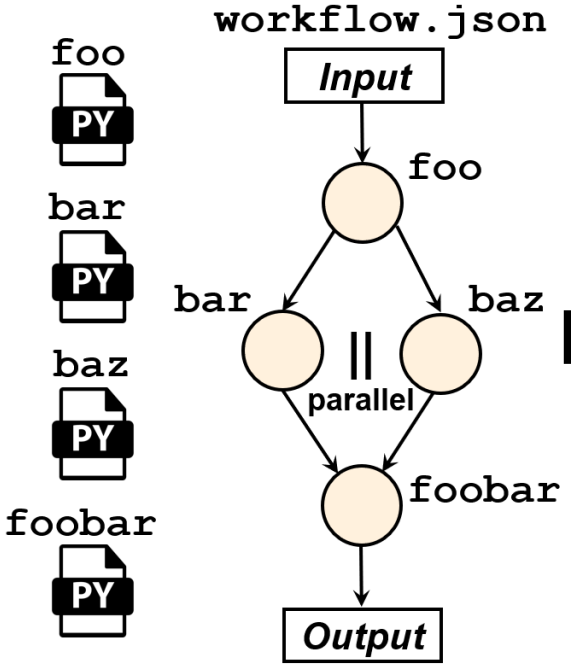

```
def Orchestrator (...) {
    T1 = Thread(target = fooWrap,
                args = [Input])
    T2 = Thread(target = ParState,
                args = fooOut)
    T3 = Thread(target = foobarWrap,
                args = ...)
    T1.start(); T1.join()
    T2.start(); T2.join()
    T3.start(); T3.join()
    ...
}
def fooWrap (Input) {
    global fooOut
    fooOut = foo(Input)
    memset_input(Input)
}
def ParState() {
    P1 = Process(target = bar,
                 args = fooOut);
    P2 = Process(target = baz;
                 args = fooOut);
    P1.start(); P2.start();
    P1.join();  P2.join();
    memset_input(fooOut);
}
...
```

Unified Workflow

Decide number of threads, processes and containers

Faastlane: Putting it all together

Developer



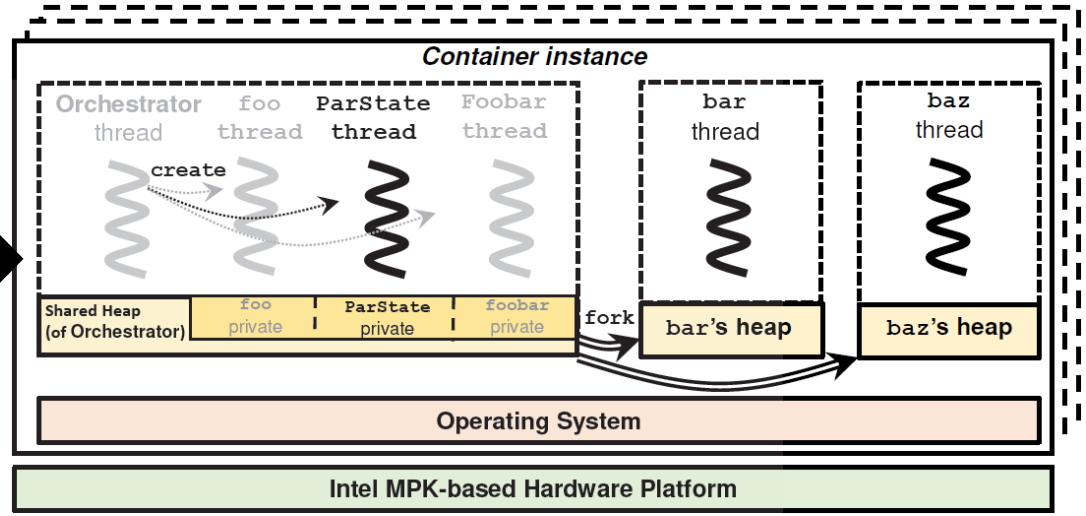
Faastlane Workflow Composer

```
def Orchestrator (...) {
    T1 = Thread(target = fooWrap,
                args = [Input])
    T2 = Thread(target = ParState,
                args = fooOut)
    T3 = Thread(target = foobarWrap,
                args = ...)
    T1.start(); T1.join()
    T2.start(); T2.join()
    T3.start(); T3.join()
    ...
}
def fooWrap (Input) {
    global fooOut
    fooOut = foo(Input)
    memset_input(Input)
}
def ParState() {
    P1 = Process(target = bar,
                 args = fooOut);
    P2 = Process(target = baz;
                 args = fooOut);
    P1.start(); P2.start();
    P1.join();  P2.join();
    memset_input(fooOut);
}
...
```

Unified Workflow

Decide number of threads, processes and containers

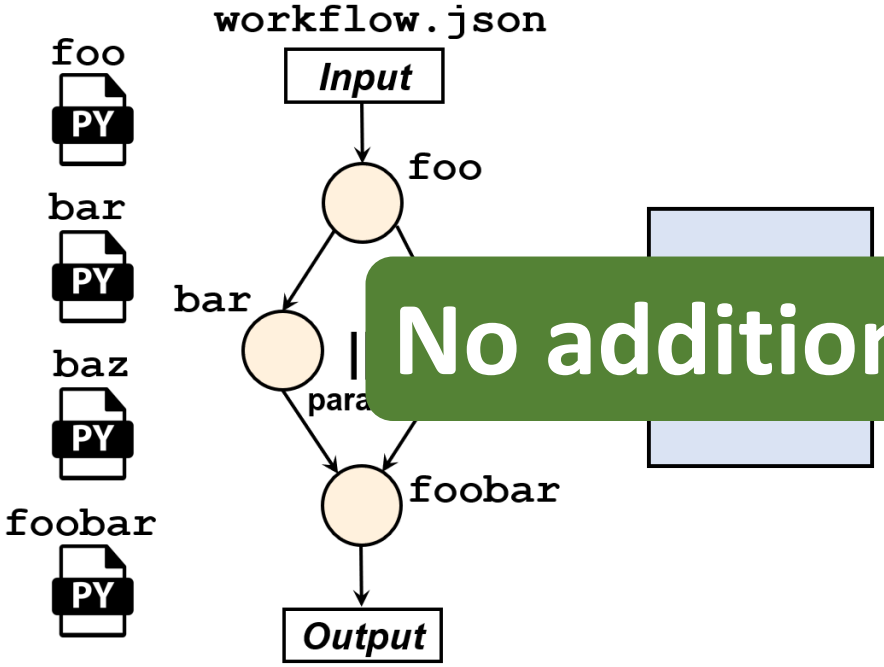
Cloud Provider



Faastlane's thread level memory manager

Faastlane: Putting it all together

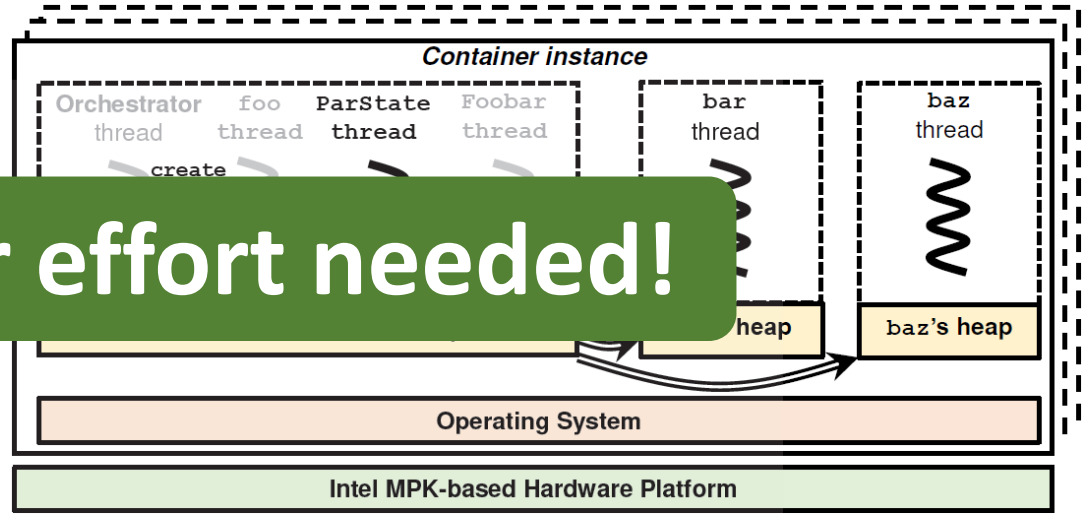
Developer




```
def Orchestrator (...) {
    T1 = Thread(target = fooWrap,
                args = [Input])
    T2 = Thread(target = ParState,
                args = fooOut)
    T3 = Thread(target = foobarWrap,
                args = ...)
    T1.start(); T1.join()
    T2.start(); T2.join()
    T3.start(); T3.join()
    P1 = Process(target = bar,
                 args = fooOut);
    P2 = Process(target = baz;
                 args = fooOut);
    P1.start(); P2.start();
    P1.join();  P2.join();
    memset_input(fooOut);
}
...
```

Decide number of threads, processes and containers

Cloud Provider



Faastlane's thread level memory manager

No additional developer effort needed!

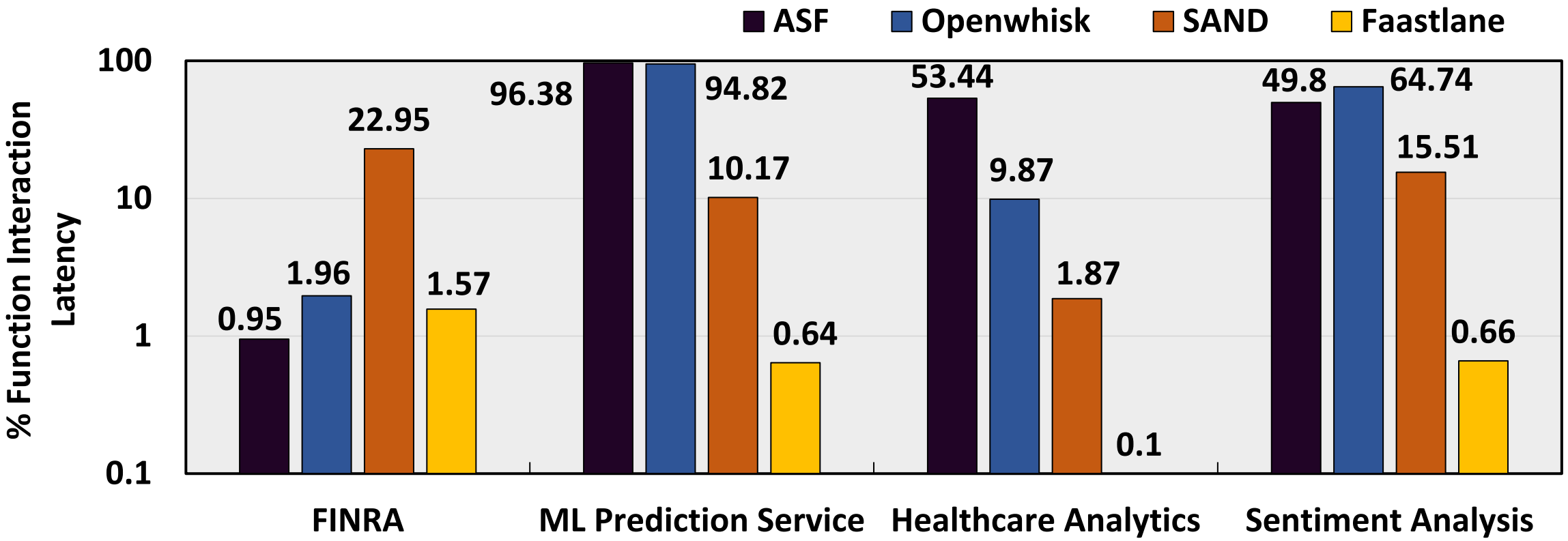
Evaluation

Experimental setup

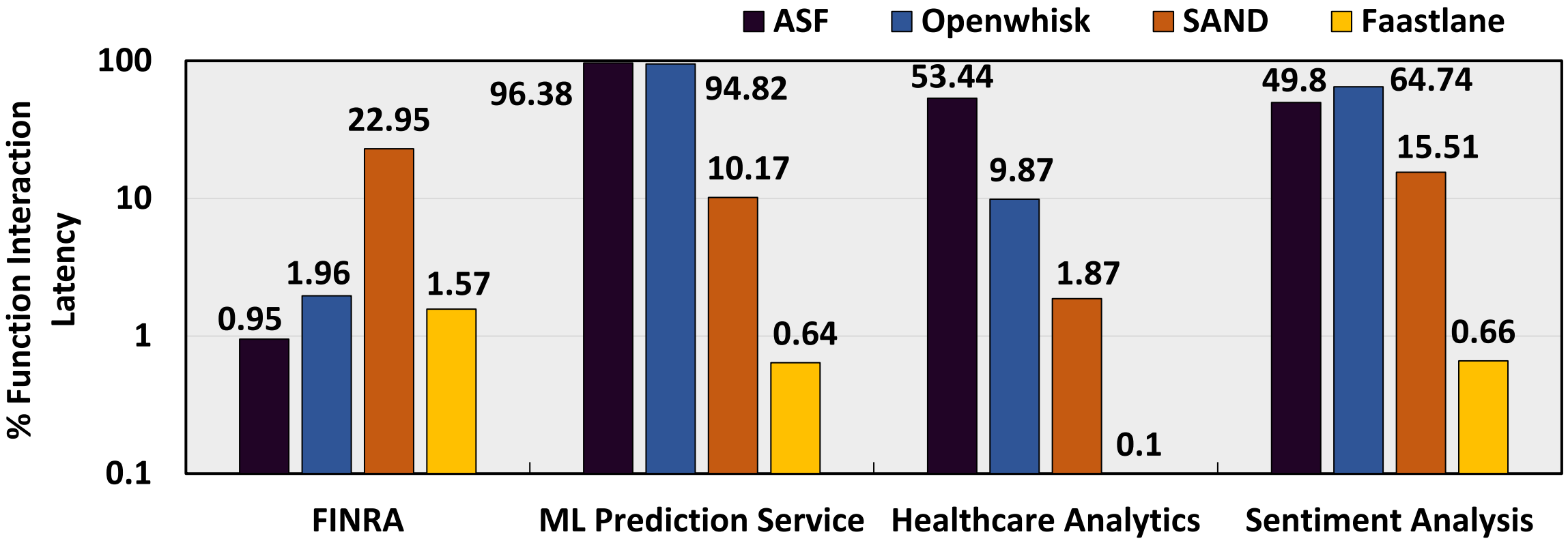
- **Hardware**
  - Intel Xeon, 36 core, 384 GB RAM
- **Software**
  - Unmodified Openwhisk framework
  - Custom python runtime with thread-level memory manager

Function interaction latency on FaaS platforms

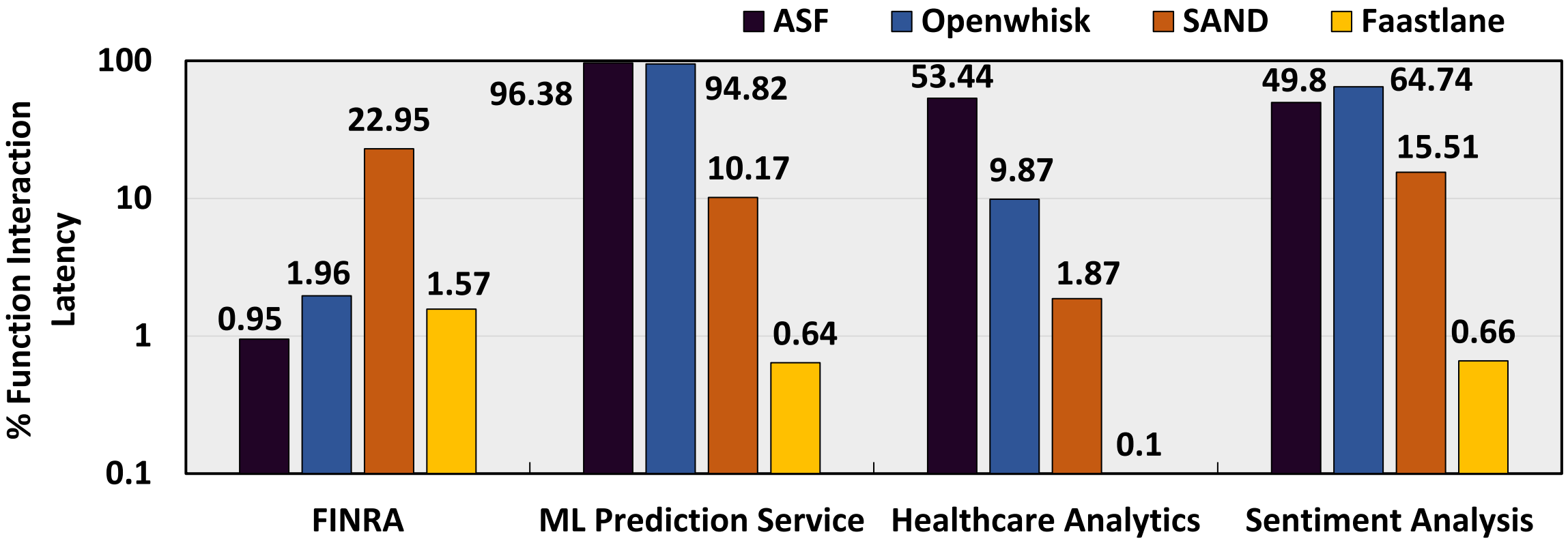
Function interaction latency on FaaS platforms



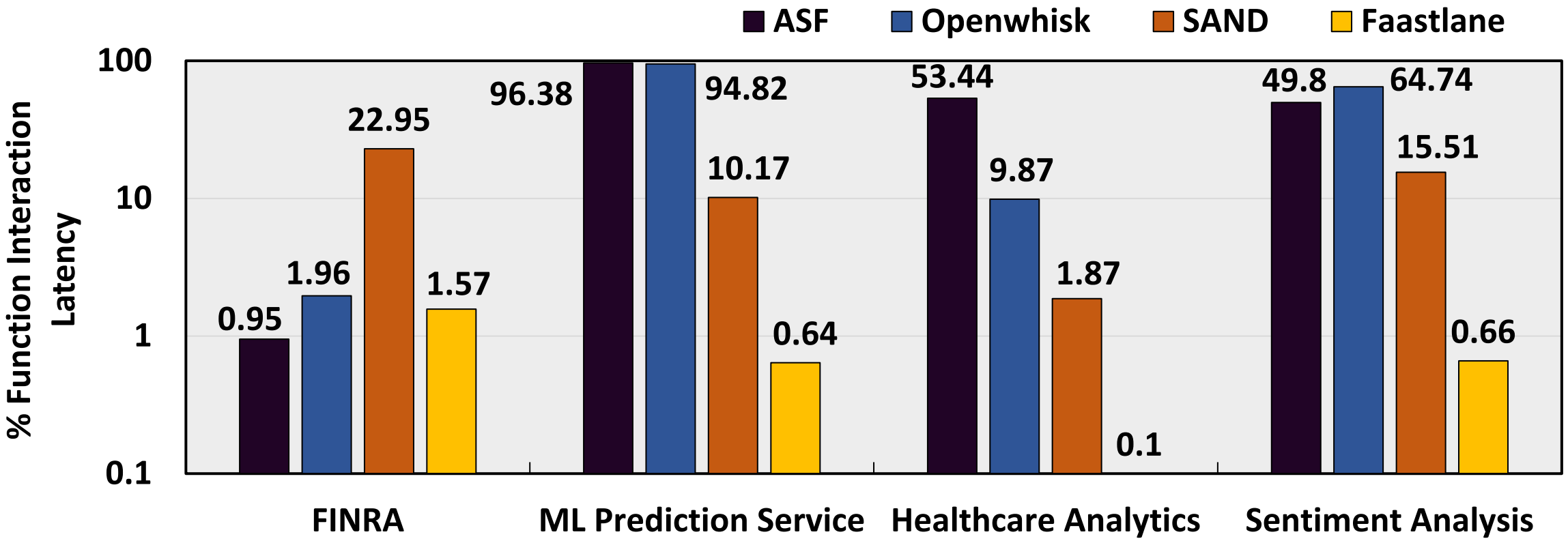
Function interaction latency on FaaS platforms



Function interaction latency on FaaS platforms

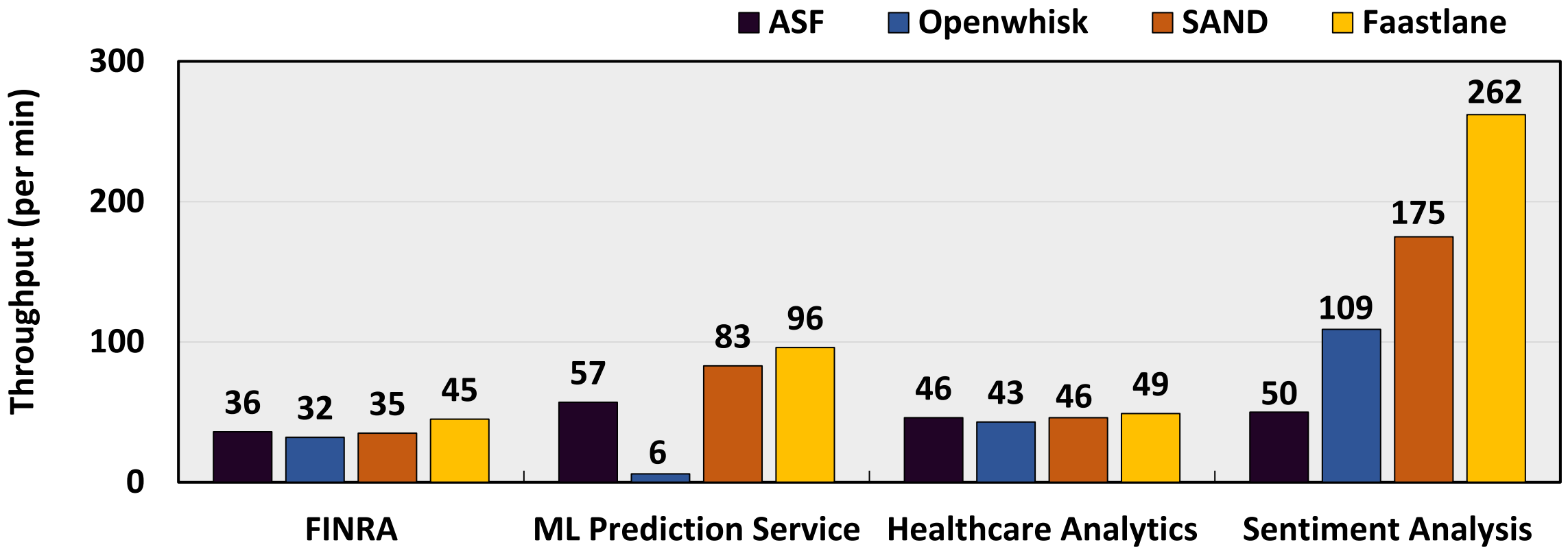


Function interaction latency on FaaS platforms

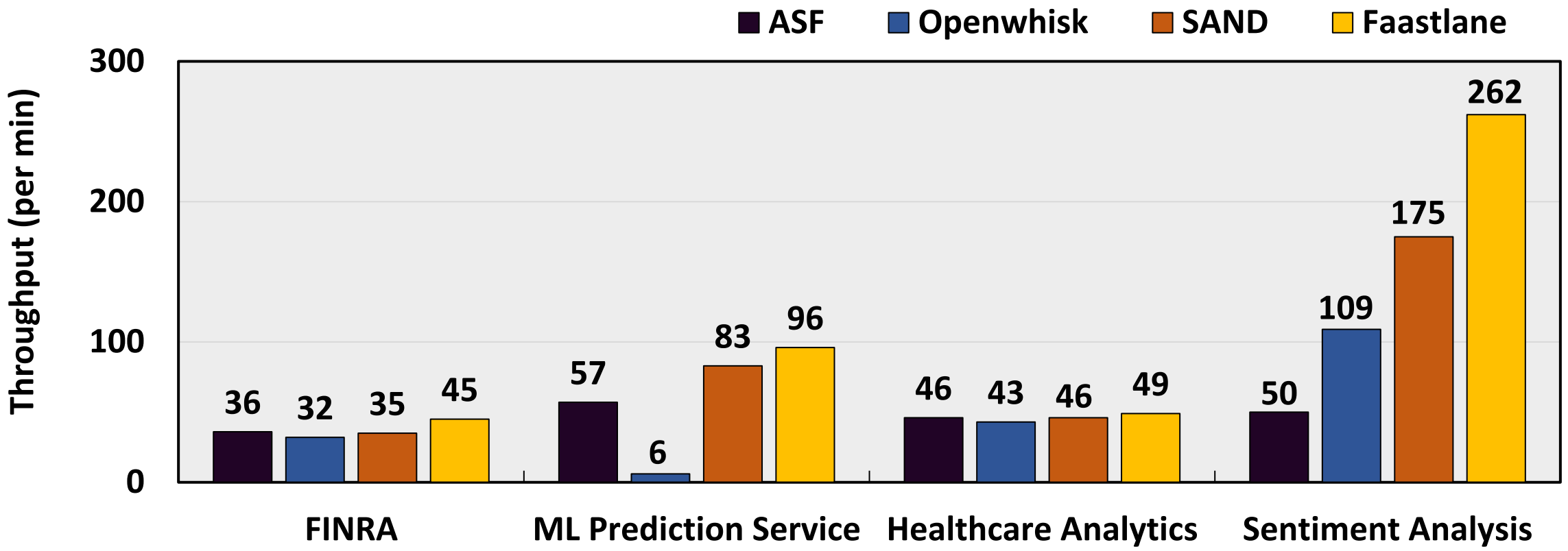


Request throughput on FaaS platforms

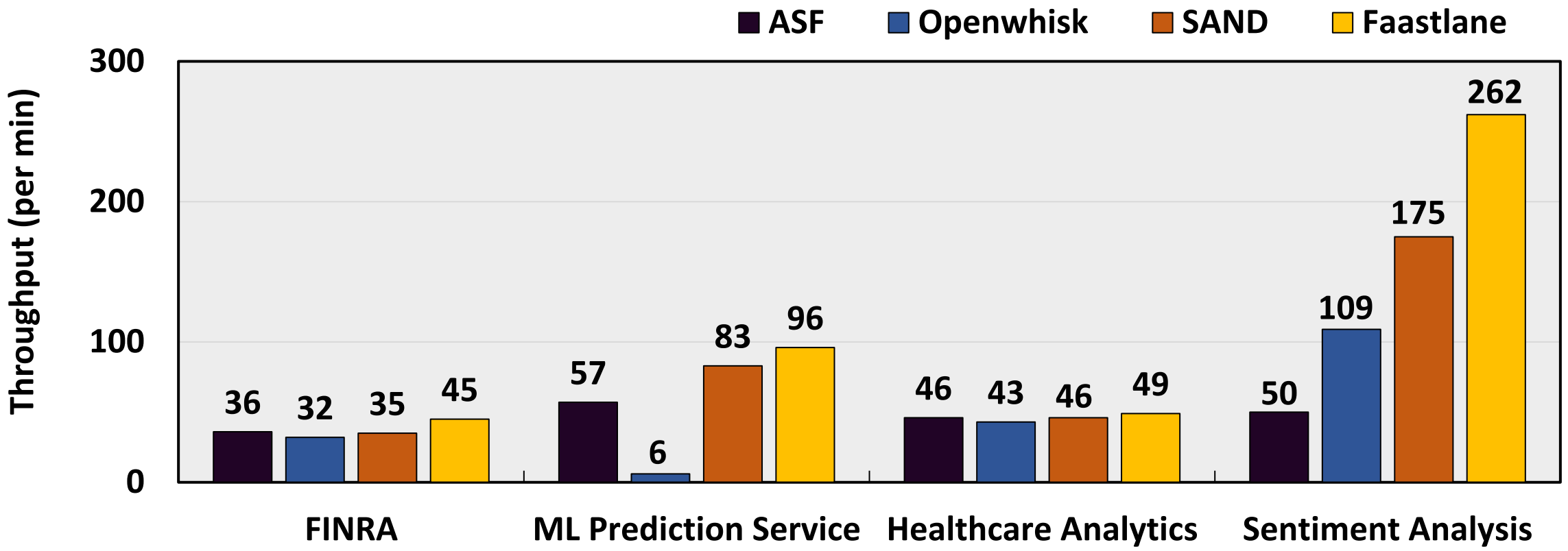
Request throughput on FaaS platforms



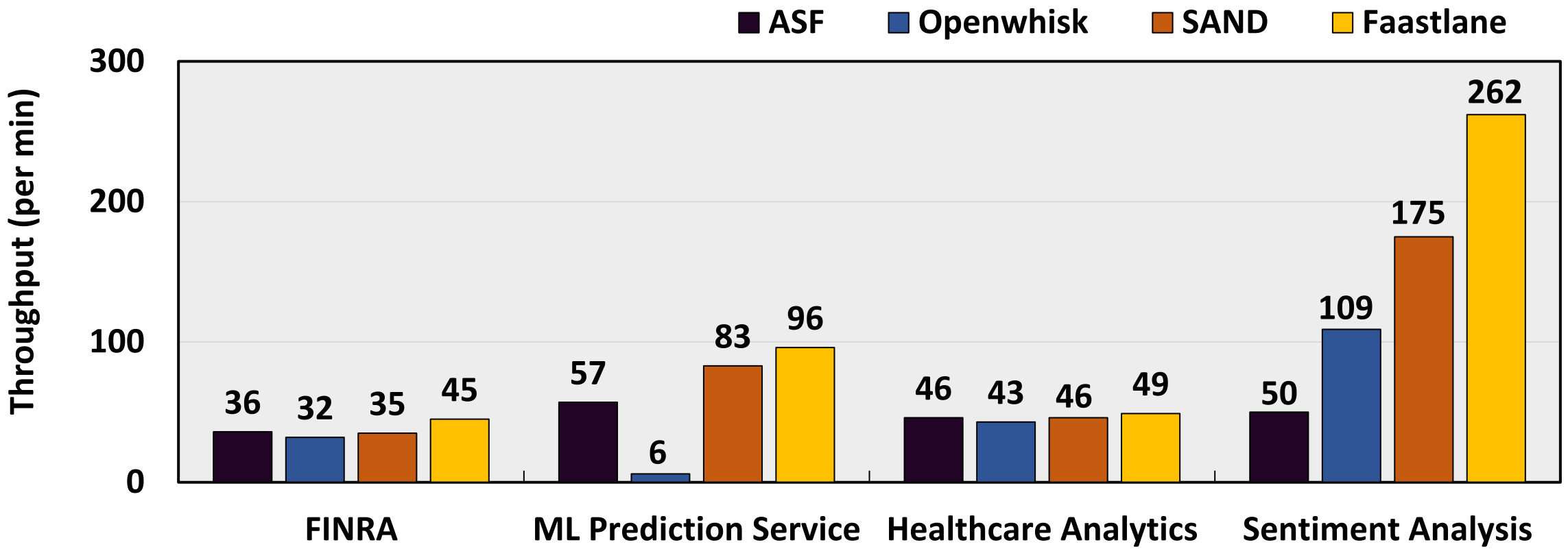
Request throughput on FaaS platforms



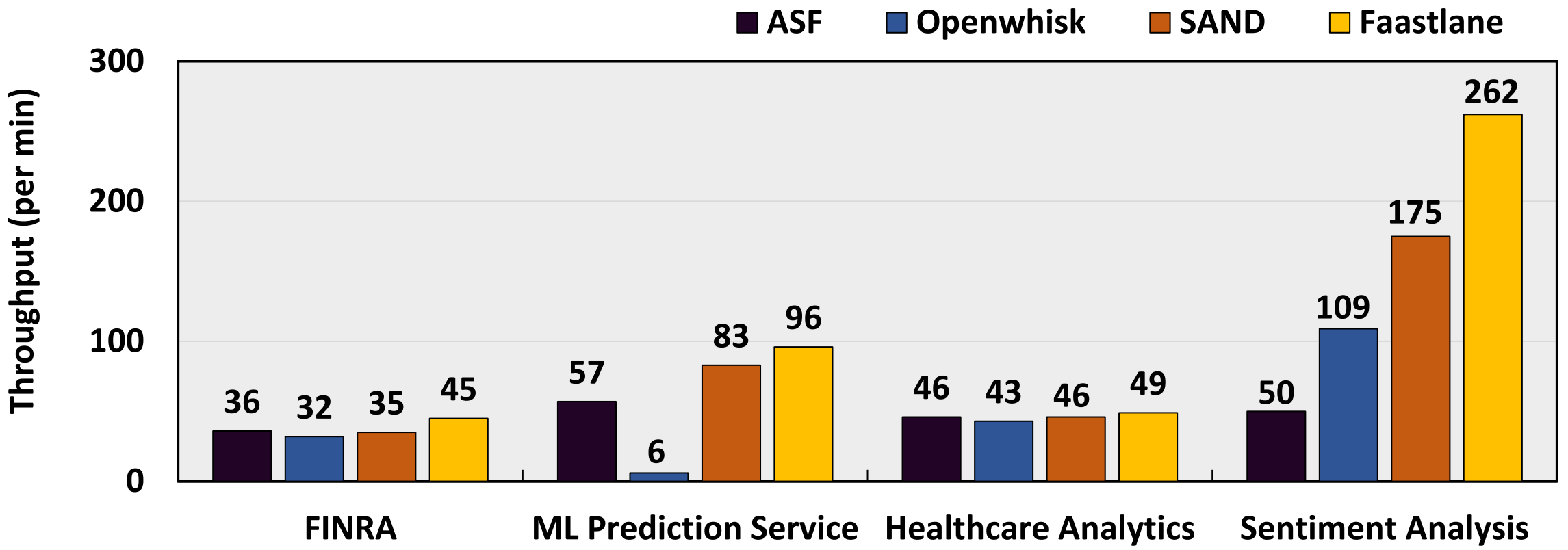
Request throughput on FaaS platforms



Request throughput on FaaS platforms



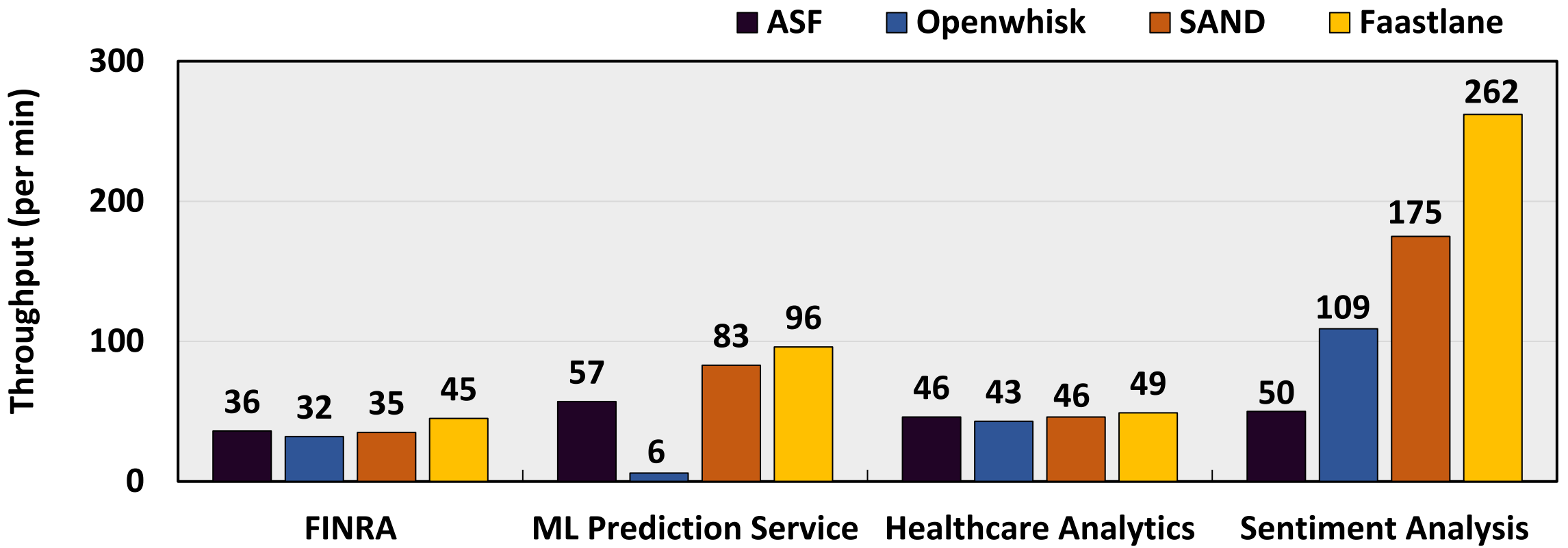
Request throughput on FaaS platforms



Faastlane over Openwhisk

Upto 16x better

Request throughput on FaaS platforms



Faastlane over Openwhisk

Upto 16x better

Faastlane over SAND

Upto 1.49x better

Req

ns

Faastlane: Accelerating Function-as-a-Service Workflows

Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, Arkaprava Basu
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

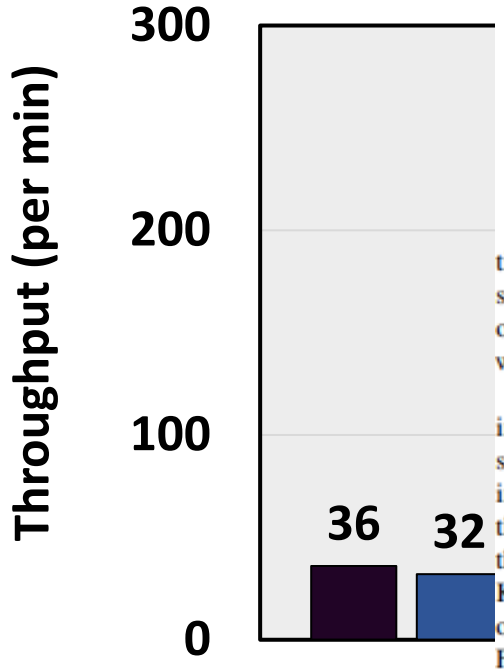
Abstract

In FaaS workflows, a set of functions implement application logic by interacting and exchanging data among themselves. Contemporary FaaS platforms execute each function of a workflow in separate containers. When functions in a workflow interact, the resulting latency slows execution.

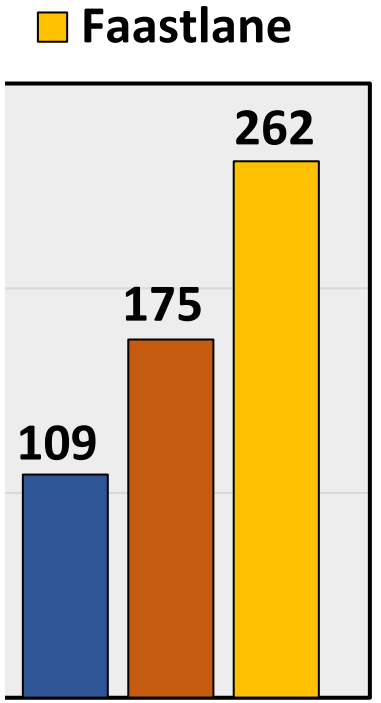
Faastlane minimizes function interaction latency by striving to execute functions of a workflow as threads within a single process of a container instance, which eases data sharing via simple load/store instructions. For FaaS workflows that operate on sensitive data, Faastlane provides lightweight thread-level isolation domains using Intel Memory Protection Keys (MPK). While threads ease sharing, implementations of languages such as Python and Node.js (widely used in FaaS applications) disallow concurrent execution of threads. Faastlane dynamically identifies opportunities for parallelism in FaaS workflows and fork processes (instead of threads) or spawns new container instances to concurrently execute parallel functions of a workflow. We implemented Faastlane atop Apache OpenWhisk and show that it accelerates workflow instances by up to 15×, and reduces function interaction latency by up to 99.95% compared to OpenWhisk.

FaaS shifts the responsibility of managing compute resources from the developer to the cloud provider. The cloud provider charges the developer (*i.e.*, cloud client) only for the resources (*e.g.*, execution time) used to execute functions in the application (workflow). Scaling is automatic for the developer—as the workload (*i.e.*, number of requests) increases, the provider spawns more instances of the workflow.

In contemporary FaaS offerings, each function, even those that belong to the *same workflow instance*, is executed on a separate container. This setup is ill-suited for many FaaS applications (*e.g.*, image- or text-processing) in which a workflow consists of multiple interacting functions. A key performance bottleneck is *function interaction latency*—the latency of copying *transient* state (*e.g.*, partially-processed images) across functions within a workflow instance. The problem is exacerbated when FaaS platforms limit the size of the directly communicable state across functions. For example, ASF limits the size of arguments that can be passed across functions to 32KB [35]. However, many applications (*e.g.*, image processing) may need to share larger objects [2]. They are forced to pass state across functions of a workflow instance via cloud storage services (*e.g.*, Amazon S3), which typically takes hun-



FI



iment Analysis

er

Conclusion

- Faastlane minimizes function interaction latency using threads

Conclusion

- Faastlane minimizes function interaction latency using threads
- Lightweight intra-process isolation through Intel MPK

Conclusion

- Faastlane minimizes function interaction latency using threads
- Lightweight intra-process isolation through Intel MPK
- Dynamically switching to processes to leverage parallelism

Conclusion

- Faastlane minimizes function interaction latency using threads
- Lightweight intra-process isolation through Intel MPK
- Dynamically switching to processes to leverage parallelism
- No additional developer effort needed

Thank You

Contact:
kjjswaroop@gmail.com
ajaynayak@iisc.ac.in
vg@iisc.ac.in
arkapravab@iisc.ac.in

https://github.com/csl-iisc/faastlane