

Going FaaSter



Cost-Performance optimisations
of Serverless on Kubernetes

Erwin van Eyk



The central trade-off in serverless computing

High Performance

“Infinite” scaling

High availability

Low latency

Low Cost

No costs when idle

No operational cost

Granular billing

How can we optimize the performance-cost trade-off?

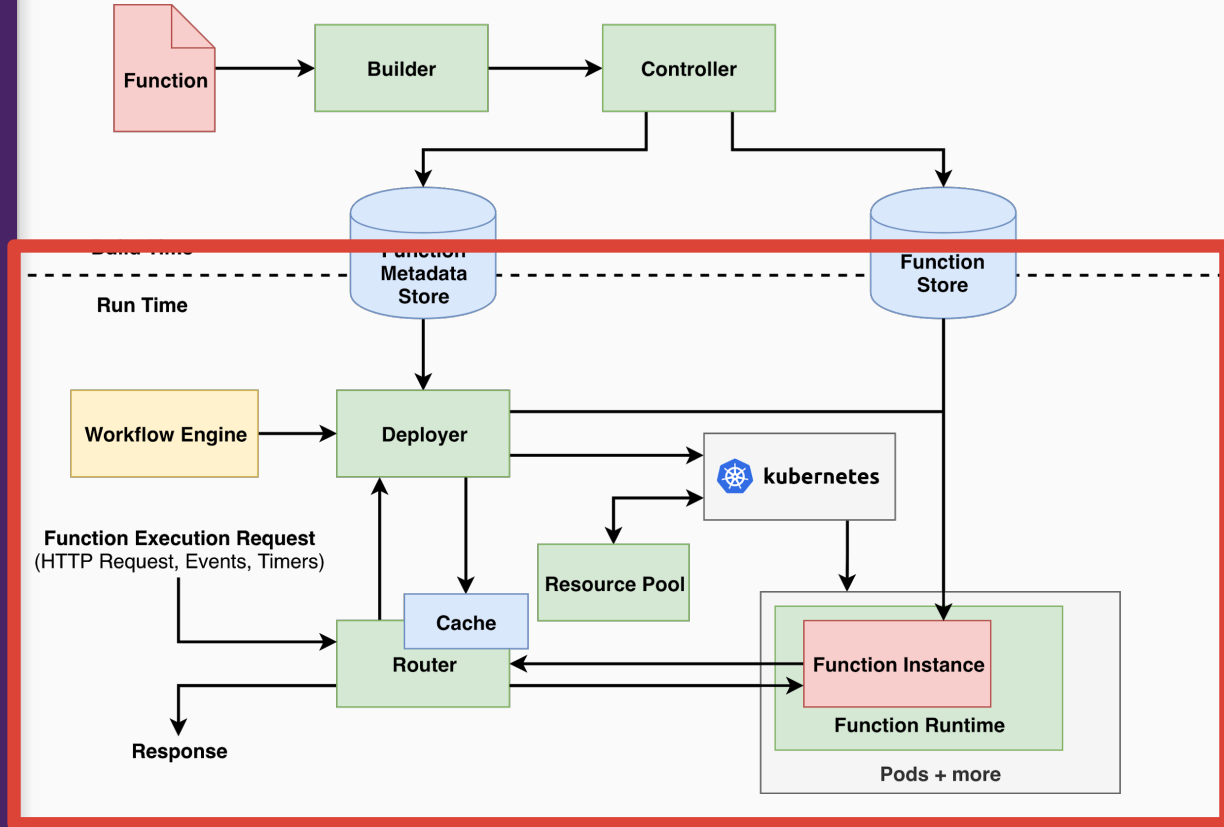
Anatomy of a FaaS platform

Build time

Developer creates, manages the functions

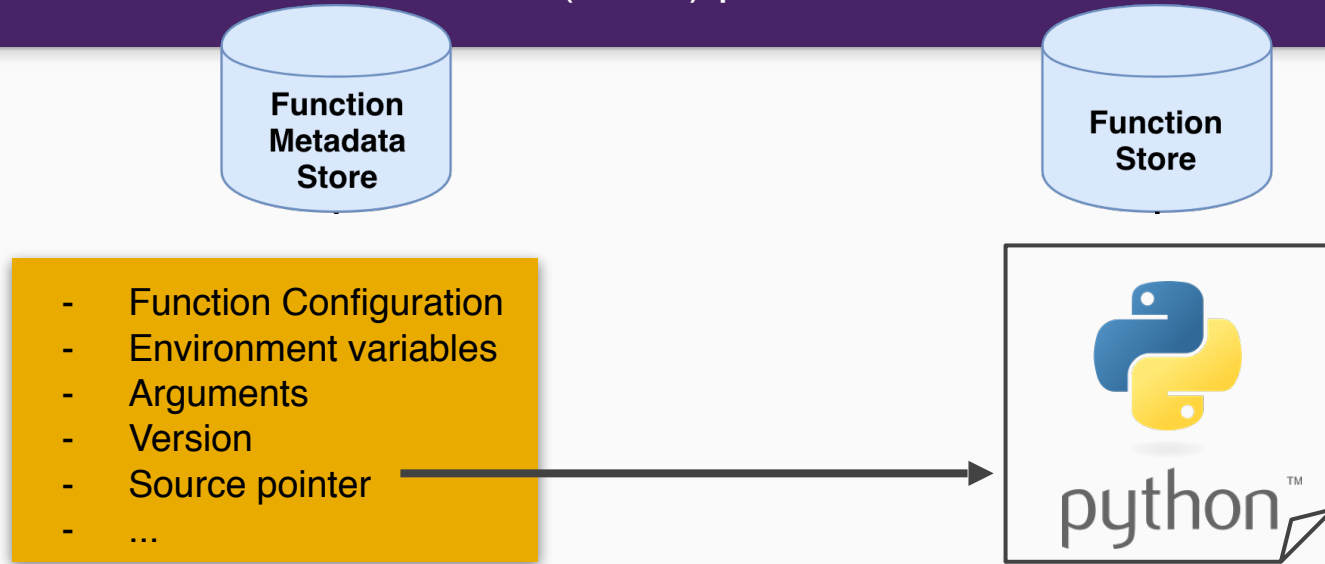
Run time

User and external systems events trigger function executions.

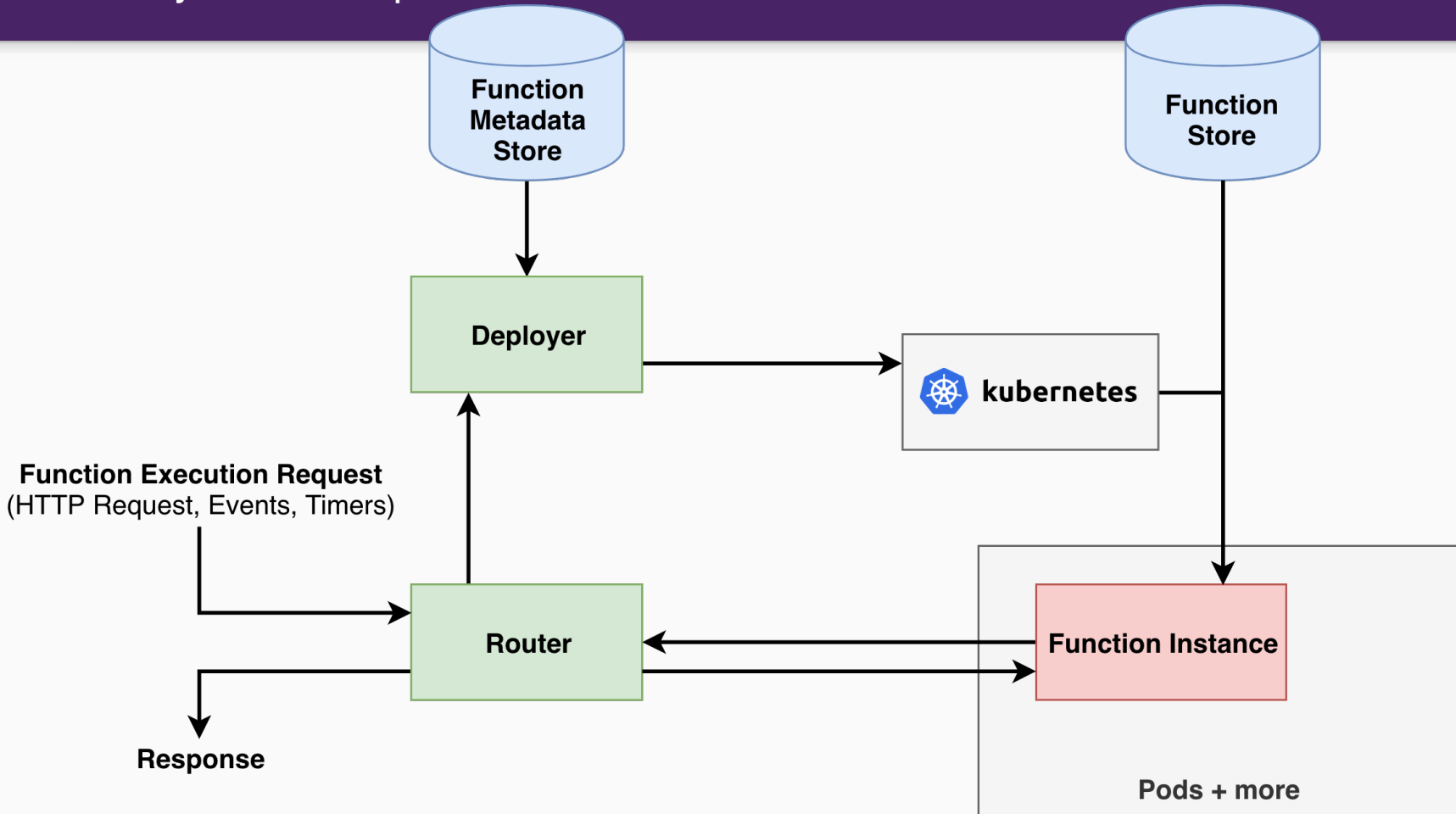


van Eyk, E., Iosup, A., Seif, S., & Thömmes, M. (2017, December). The SPEC cloud group's research vision on FaaS and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing* (pp. 1-4). ACM.

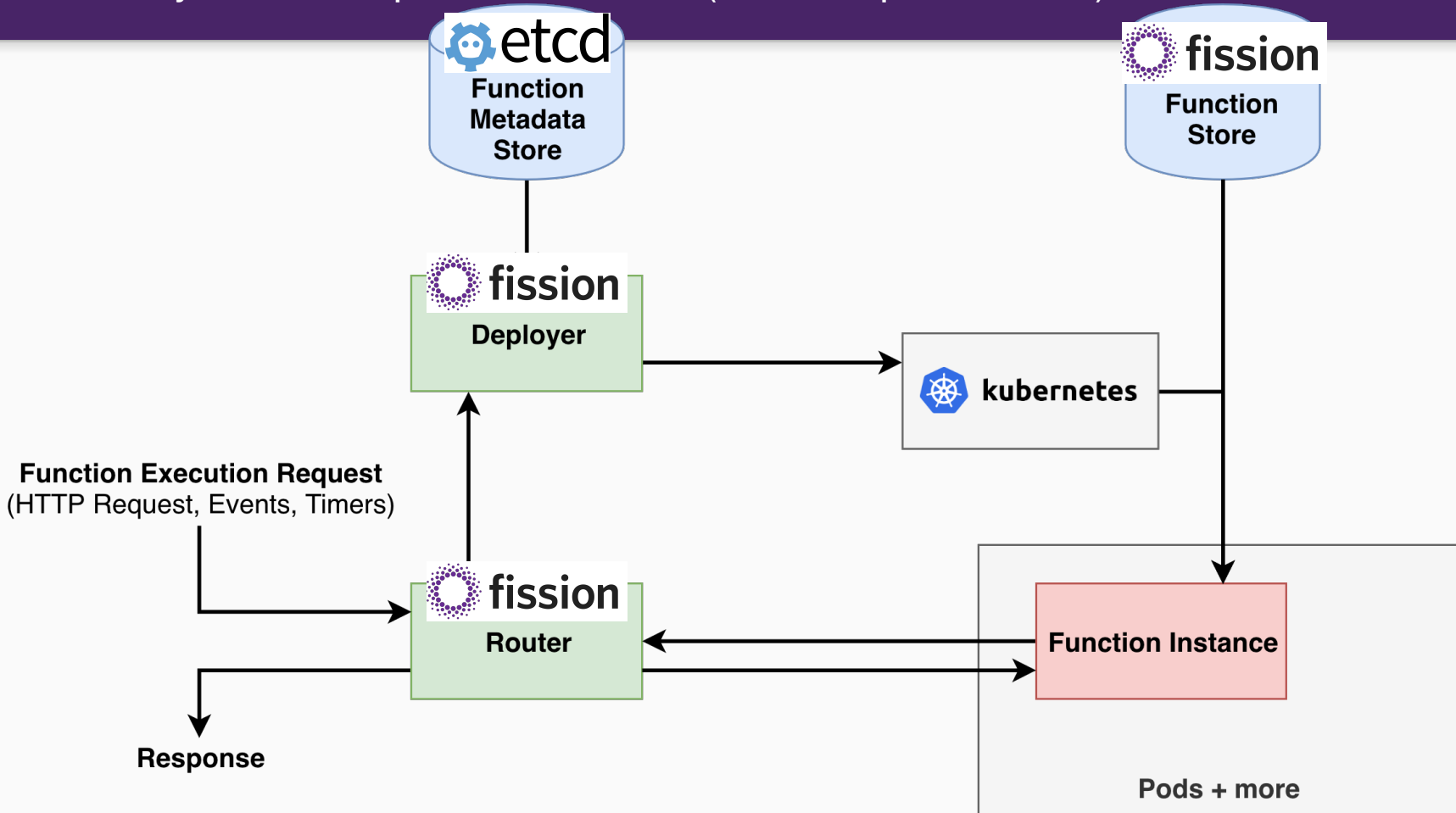
Anatomy of a Functions-as-a-Service (FaaS) platform



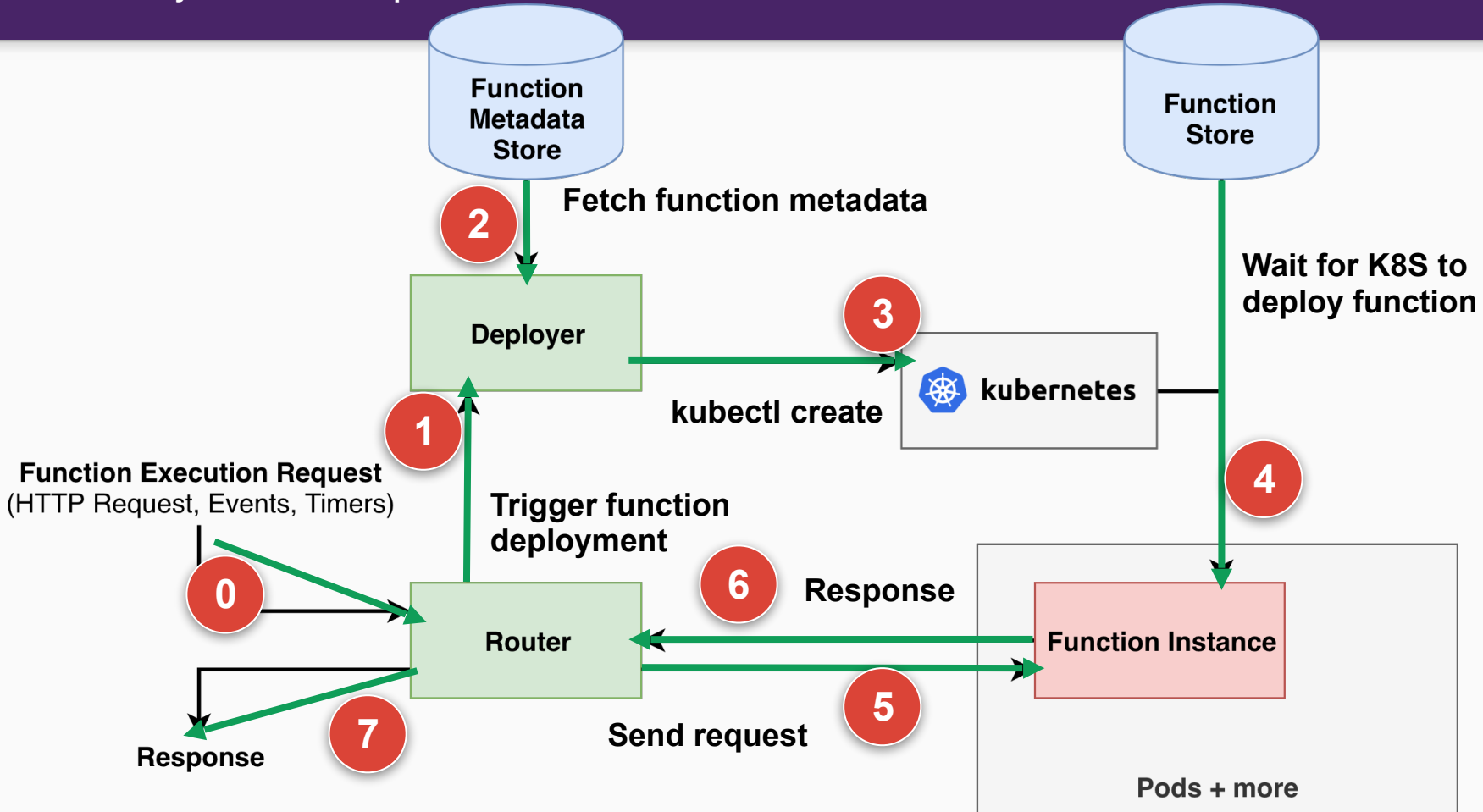
Anatomy of a FaaS platform



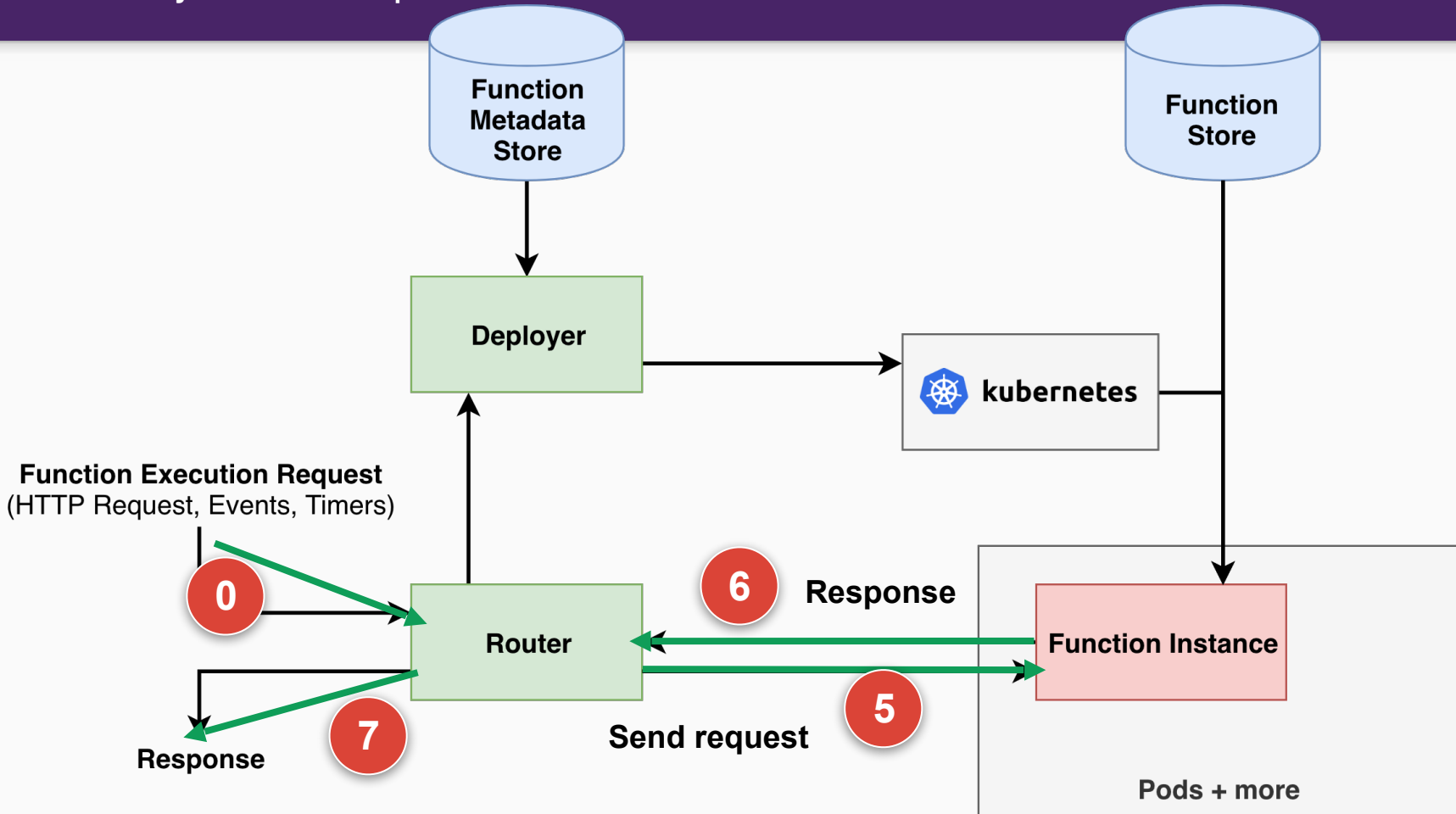
Anatomy of a FaaS platform: Fission (without optimizations)



Anatomy of a FaaS platform: cold start



Anatomy of a FaaS platform: warm execution



Cold Start

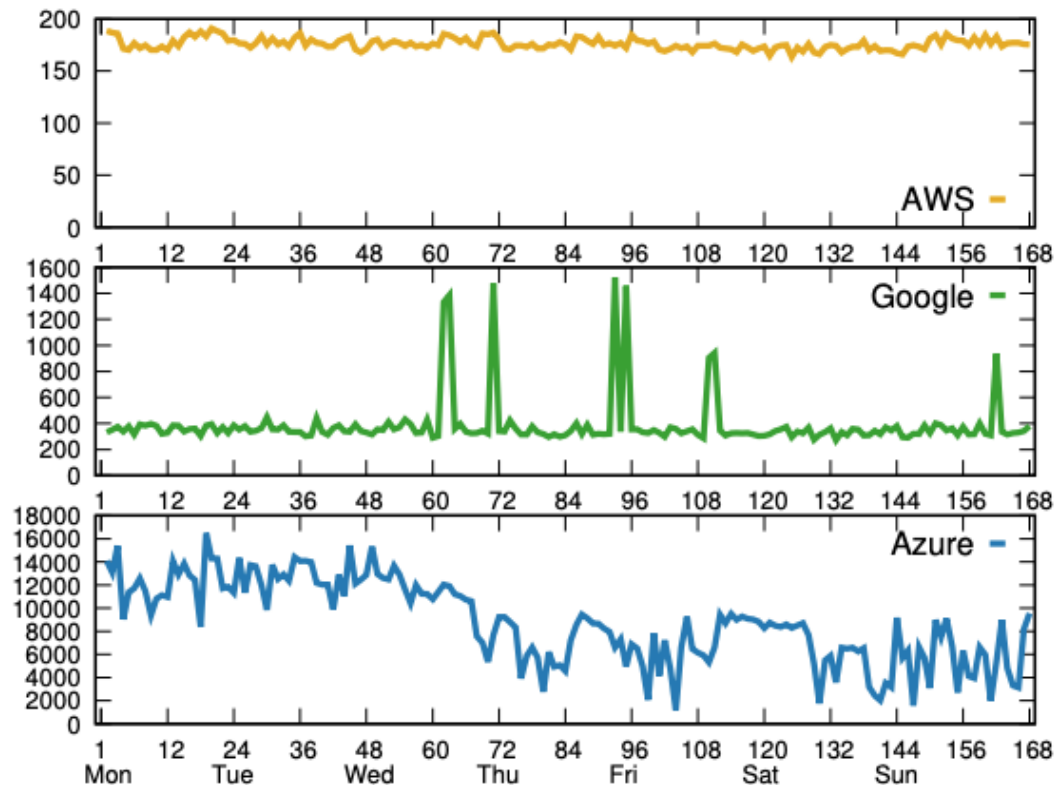


Warm Execution



Cold starts matter!

Coldstart latency (in ms) over 168 hours



180 ms

500 ms

3600 ms

How do FaaS platforms improve their performance?

And, at what cost?

1. Function resource reusing
2. Function runtime pooling
3. Function prefetching
4. Function prewarming

Optimization 1

Function Resource Reusing

**Trigger
deployer**

Fetch function
metadata

Deploy pod

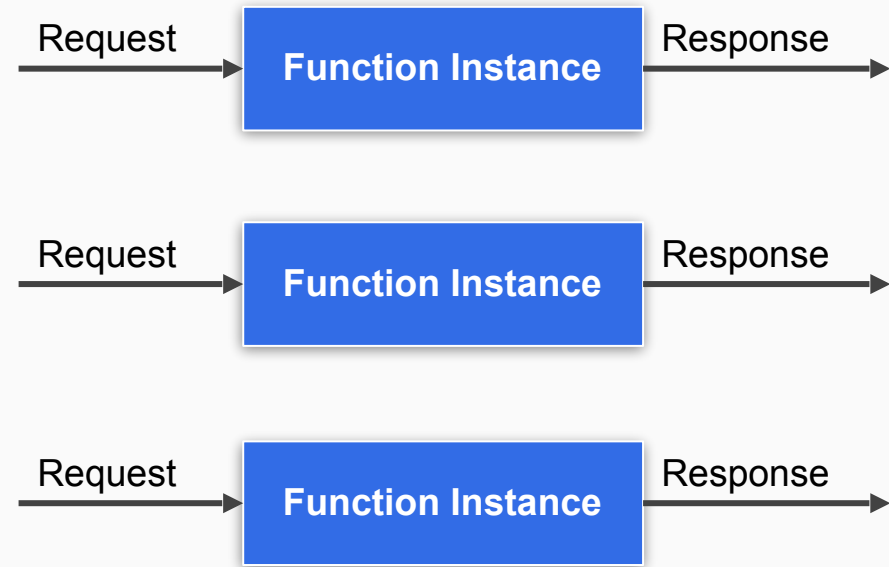
Fetch
function

Deploy function

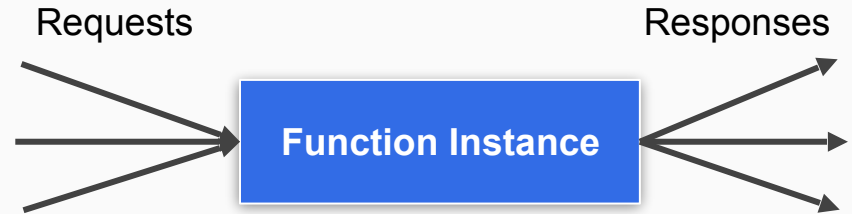
**Route
request**

Function
Execution

Function Isolation vs. Function Reuse



Full Isolation



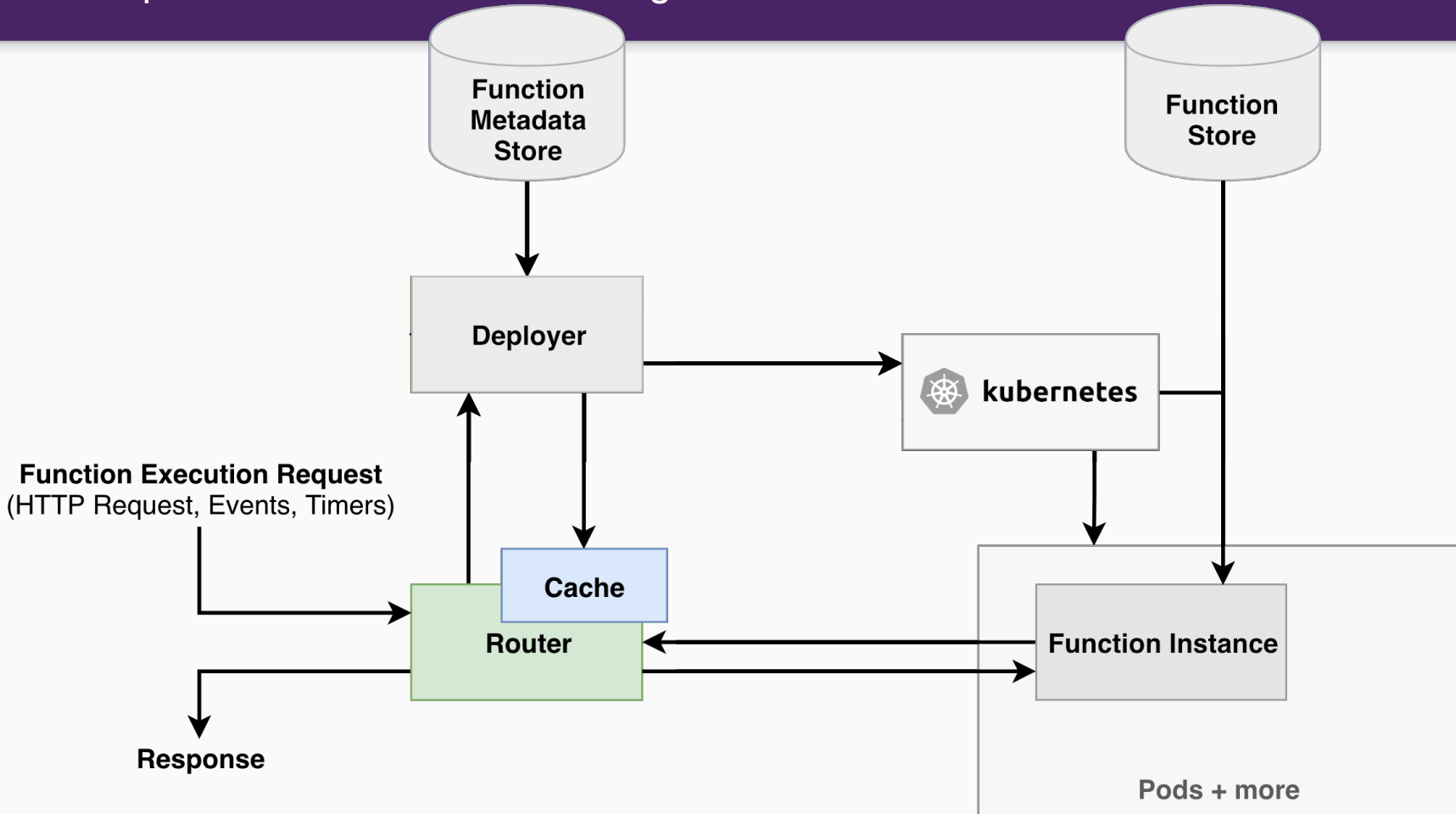
Full resource reuse



Function resource reusing in practice

- Why performance isolation:
 - Performance variability
- In practice: all FaaS platforms reuse resources
 - Per-user binpacking
 - Functions are isolated
 - Function executions share resources

FaaS platform with function reusing



Trade-off: how long to keep functions alive?

- To reuse functions we have to keep them alive.
- Keep-alive in practice:
 - AWS: ~6 hours
 - Google: ~6 hours
 - Azure: 1-4 days

Long keep-alive

short keep-alives

More warm executions

Less idle resources



Optimization 2

Function Runtime Pooling

Trigger
deployer

Fetch function
metadata

Deploy pod

Fetch
function

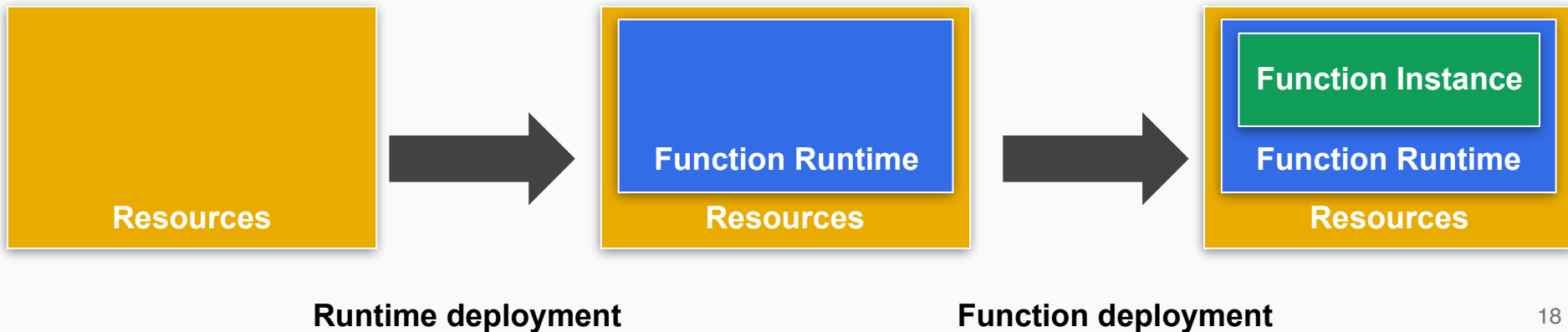
Deploy function

Route
request

Function
Execution

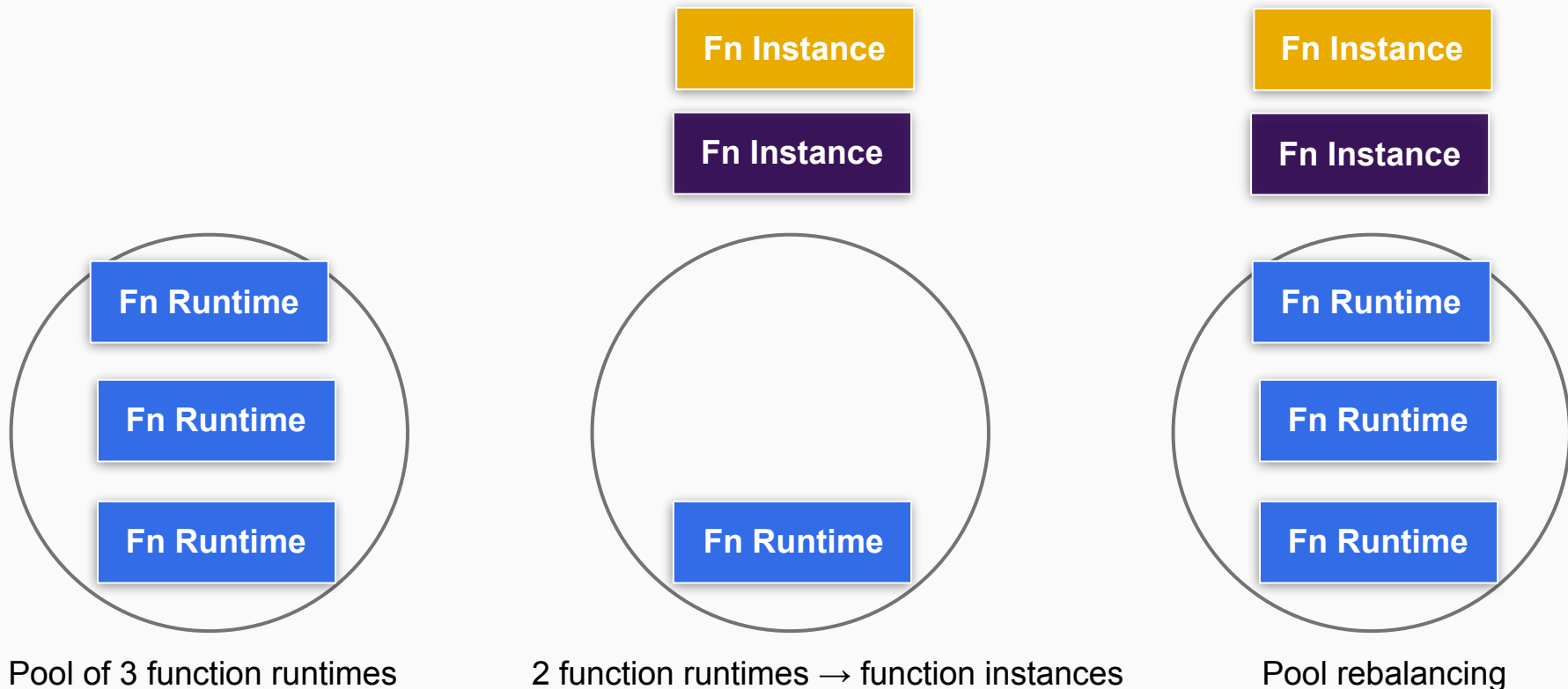
Function Instance = Runtime + Function

- Insight: function instances consist out of two parts
 - **Function-specific code:** user-provided business logic.
 - **Runtime:** operational logic, monitoring, health checks...
- Divide the deployment process into 2 stages:
 - Deploy the runtime → unspecialized runtime or stem cell
 - Deploy the function to the runtime → specialized function

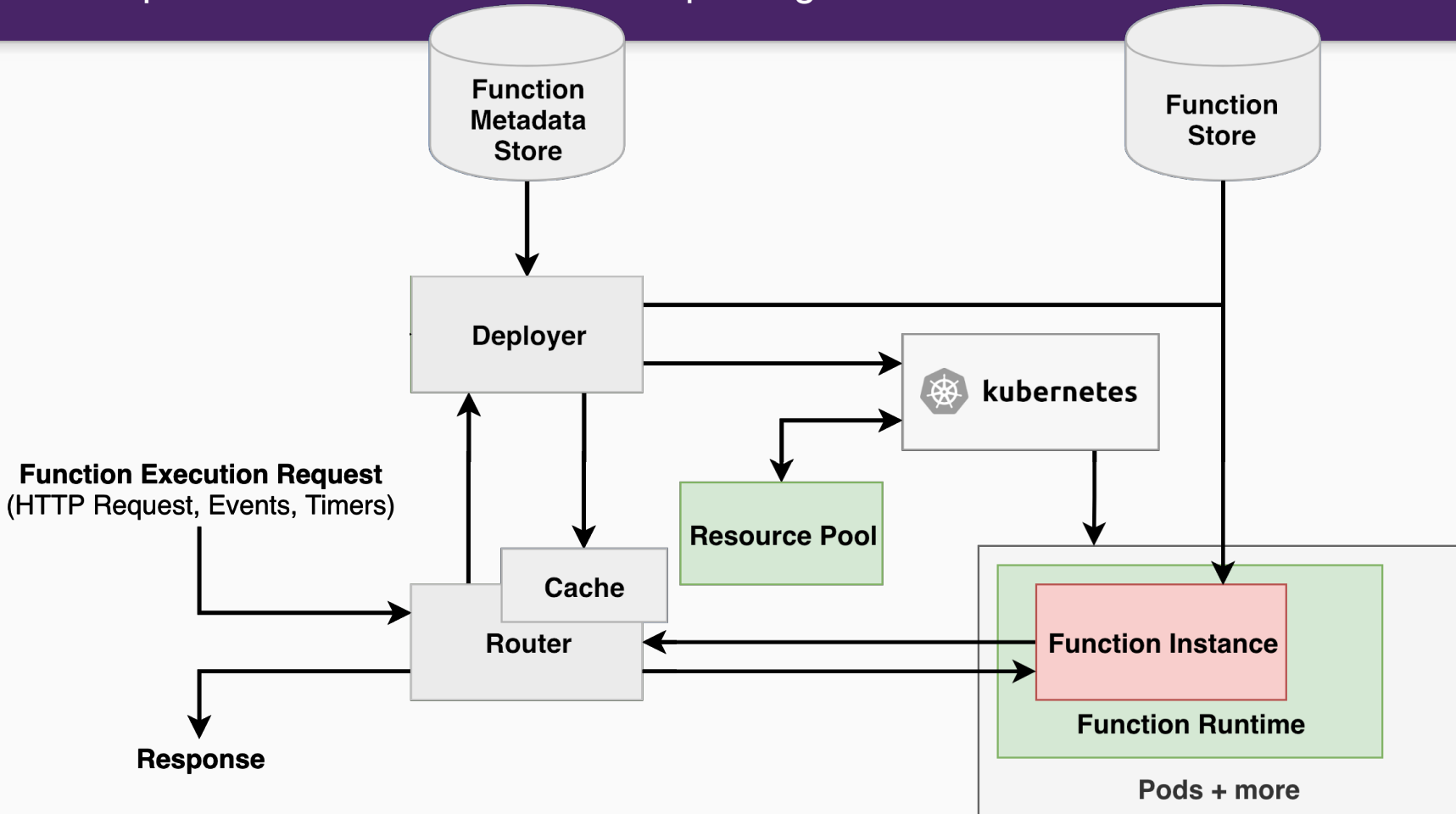


Resource Pooling

- Common in many domains (e.g. thread pools)



FaaS platform with function runtime pooling



Trade-off: how big should the pool?

Large pool

Handle high concurrency

Increases resource overhead

Minimal pool

Fast pool exhaustion

Minimize pool; less idle resources

Performance

Minimize cost



Optimization 3

Function Prefetching

Trigger
deployer

Fetch function
metadata

Deploy pod

**Fetch
function**

Deploy function

Route
request

Function
Execution

Fetch function sources proactively and place them near resources to reduce function transfer latency

- Software flow has a big impact on cold start durations
 - Function sources (10s of MBs) have to be retrieved and transferred to the resources
- Especially important for geo-distributed and edge use cases
 - AWS Lambda@edge
 - Cloudflare

Prefetching



Remote Storage



Cluster-level



Rack/Machine-level



Function-level

Prefetching

Higher latency
Less storage costs



Remote Storage



Cluster-level



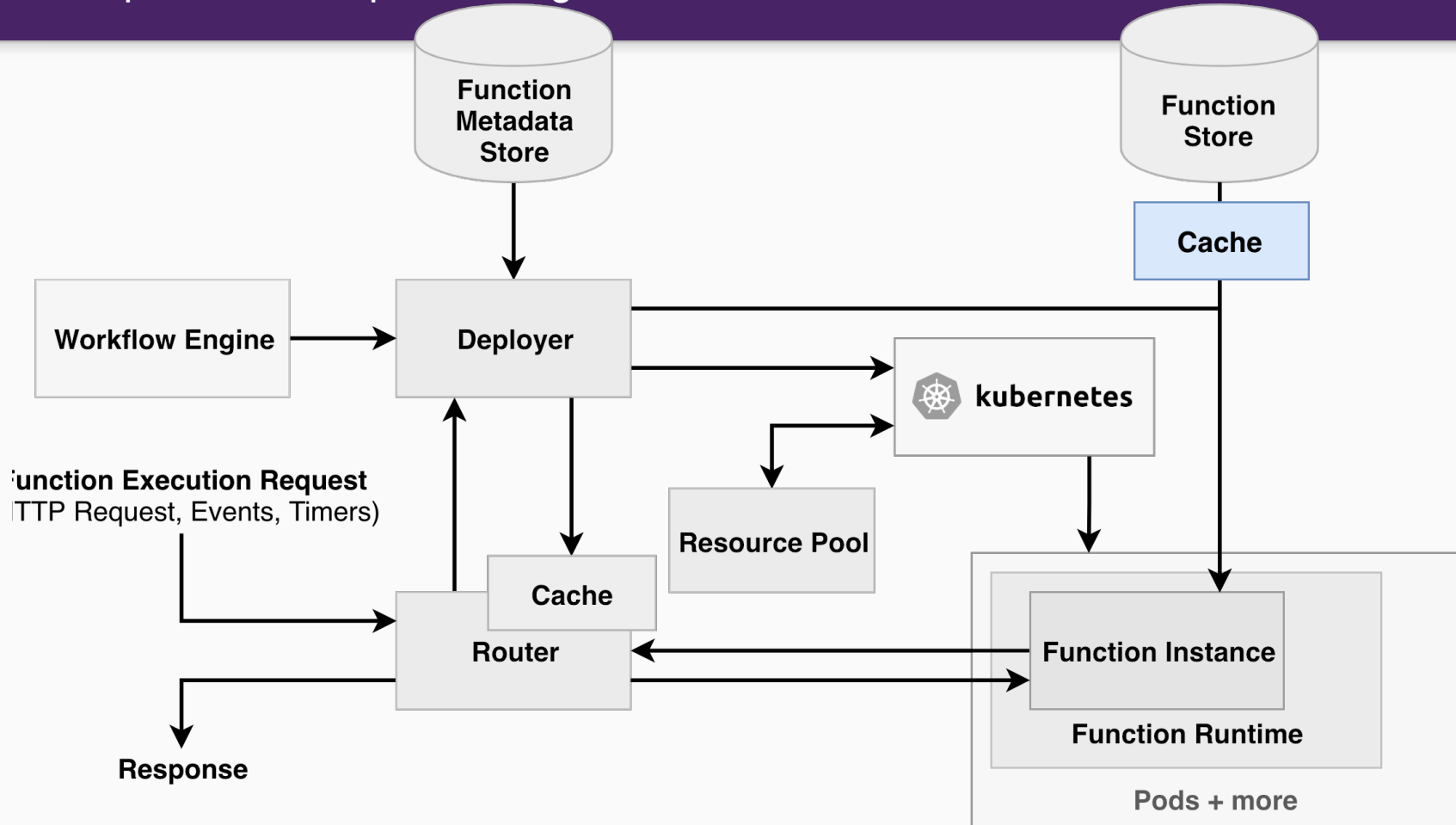
Rack/Machine-level

Lower latency
More storage costs



Function-level

FaaS platform with prefetching



Optimization 4

Function Prewarming

**Trigger
deployer**

Fetch function
metadata

Deploy pod

Fetch
function

Deploy function

Route
request

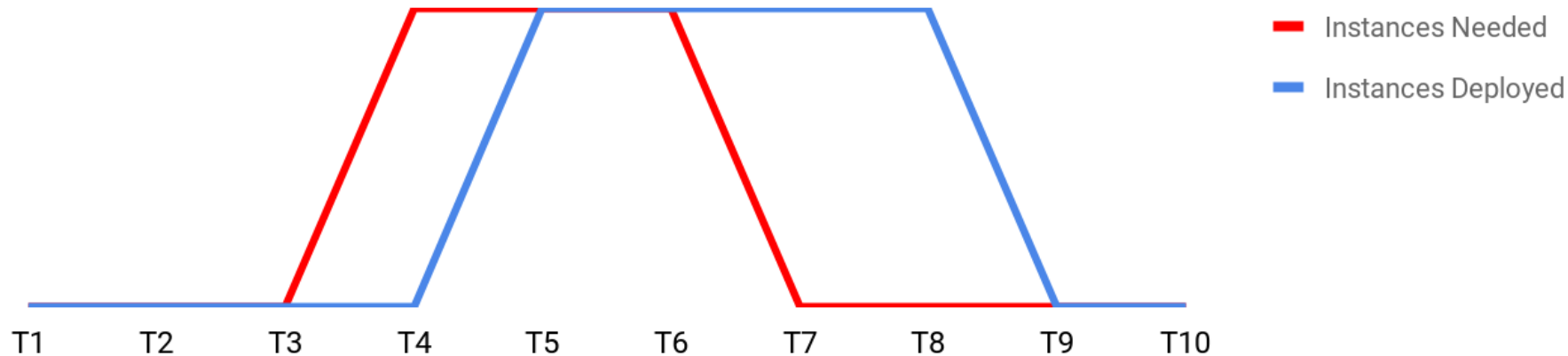
Function
Execution

Function prewarming

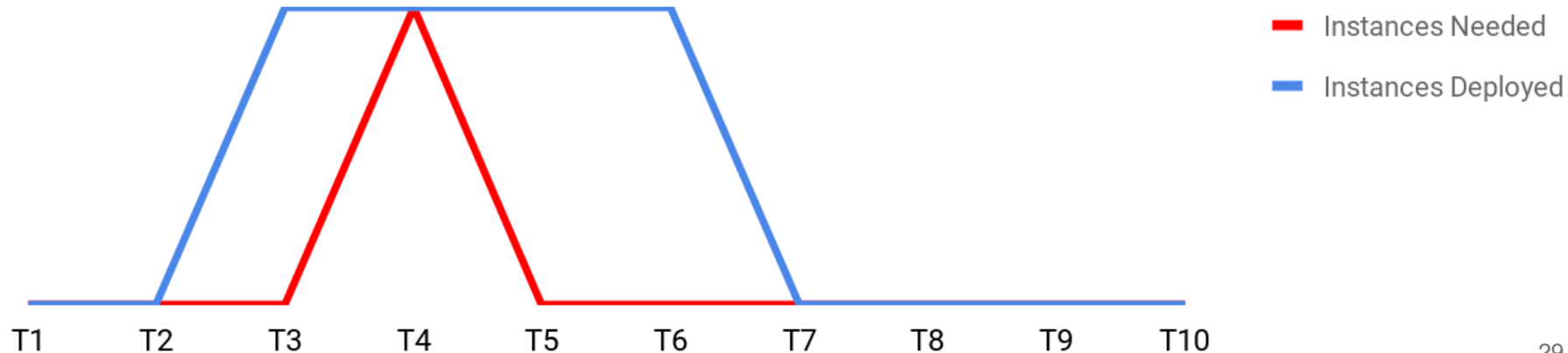
Anticipate function executions by deploying functions predictively.

- Prewarming or predictive scheduling in other domains:
 - CPU branch predictor
 - Proactive autoscalers
 - Predictive caches

Without prewarming



With prewarming



Predicting function executions is hard...

Active field of research (autoscaling, predictive caches...)

Common approaches

1. Runtime analysis
 - Rule-based
 - Pattern recognition and machine learning
 - Artificial intelligence
2. Exploit additional information of functions
 - Dependency knowledge in function compositions
 - Interval triggers

... and involves a trade-off.

Optimistic prewarming

Low threshold

Misprediction: resources wasted

Pessimistic prewarming

High threshold

Misprediction: no prewarm

More performance due to prewarming

Less costs due to less mispredicted prewarming



Lambda Warmer: Optimize AWS Lambda Function Cold Starts

Posted in [Serverless](#) on July 14, 2018



At a recent AWS Startup Day event in Boston, MA, [Chris Munns](#), the Senior Developer Advocate for Serverless at AWS, discussed Lambda [cold starts](#) and how to mitigate them. According to Chris (although he acknowledge that it is a “hack”) using the **CloudWatch Events “ping”** method is really the only way to do it right now. He gave a number of really good tips to pre-warm your functions “correctly”:

- Don’t ping more often than every 5 minutes
- Invoke the function directly (i.e. don’t use API Gateway to invoke it)

Hi, I’m Jeremy Daly. Thanks for visiting, I hope you find the information on this site useful. Be sure to follow me on Twitter and Github for the latest updates and [projects](#) I’m working on.

 Follow @jeremy_daly

 Follow @jeremydaly

Need help with your project? Learn more [about me](#) and my [consulting](#) services.

off-by-one

Sign up for my **WEEKLY email newsletter** that focuses on using

... and involves a trade-off.

Optimistic prewarming

Low threshold

Misprediction: resources wasted

Pessimistic prewarming

High threshold

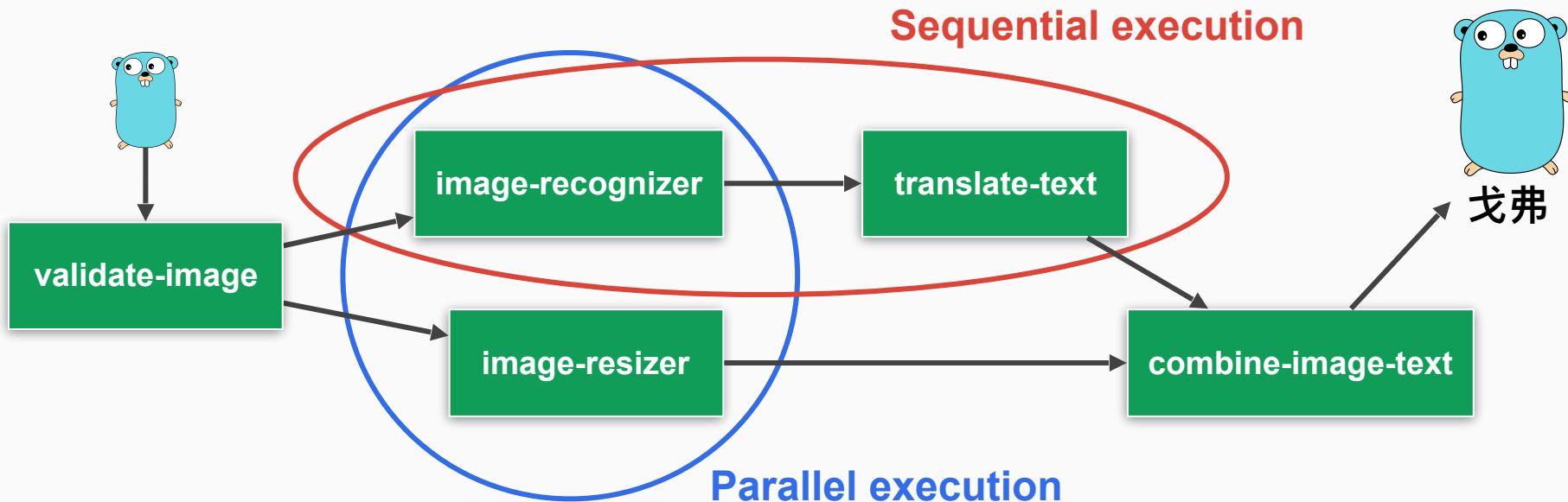
Misprediction: no prewarm

More performance due to prewarming

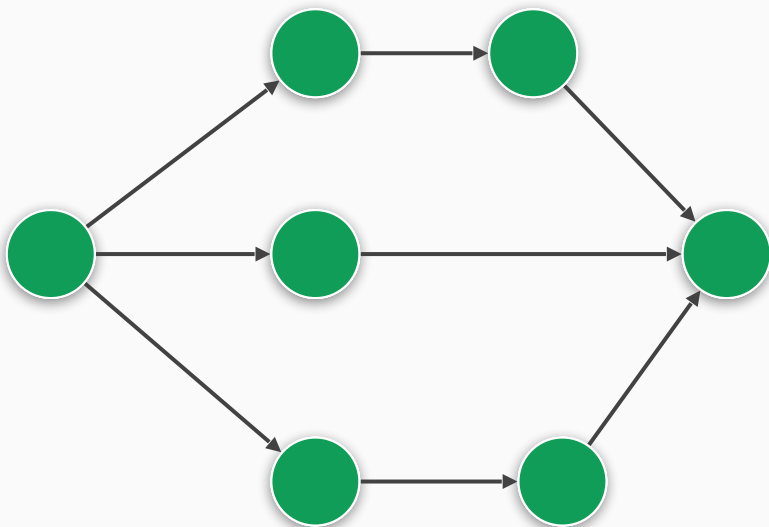
Less costs due to less mispredicted prewarming



- Connect existing functions into complex function compositions
- Workflow engine takes care of the plumbing and provides fully monitorable, fault-tolerant function compositions with low overhead.



Fission Workflows supports **horizon-based prewarming**



-  Finished
-  Started
-  Prewarmed
-  Not started

Conclusion

Four techniques key to performance in serverless:

1. Function resource reusing
2. Function runtime pooling
3. Function prefetching
4. Function prewarming

Each makes a trade-off between performance and cost.

Serverless: Pay not just for what you use - pay for what you need.

Thanks!



Slack

Twitter

<http://fission.io> + <https://github.com/fission>

<http://fission.io/workflows>

<http://slack.fission.io/>

@fissionio

Erwin van Eyk

Software Engineer @ Platform9 Systems

Chair @ SPEC CLOUD RG Serverless

Researcher @ AtLarge Research

@erwinvaneyk

erwin@platform9.com

<https://erwinvaneyk.nl>

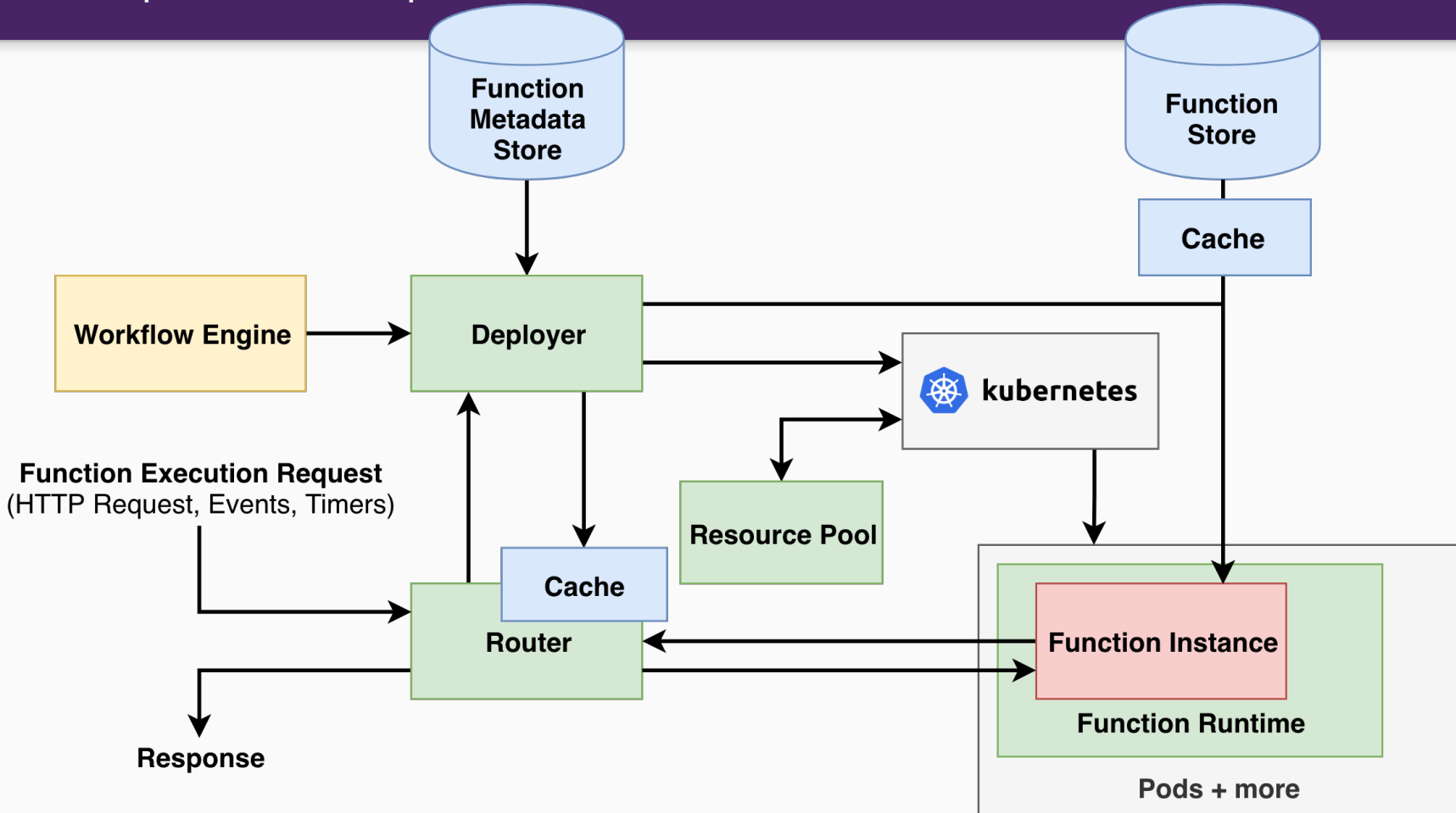


Additional Slides

What is next?

- Function and execution scheduling
- Workload-based predictions
- Comprehensive Benchmarks
- Performance Overhead Reductions
- Explicit Performance vs. Cost trade-offs

FaaS platform with optimizations



Fission Workflows supports initial **horizon-based prewarming**

