TEXAS
The University of Texas at Austin

Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices

Zhipeng Jia, Emmett Witchel
University of Texas at Austin

Motivation: Two Trends in Cloud Computing

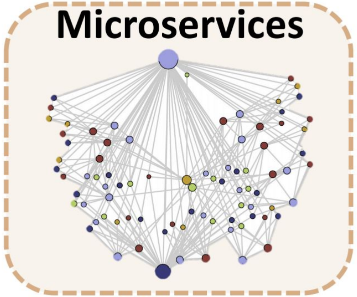
Serverless functions / Function as a service (FaaS)

- User provides *stateless* functions, that are executed on cloud provider's infrastructure
- Benefits: elasticity, and pay-as-you-go billing

AWS Lambda

Microservices

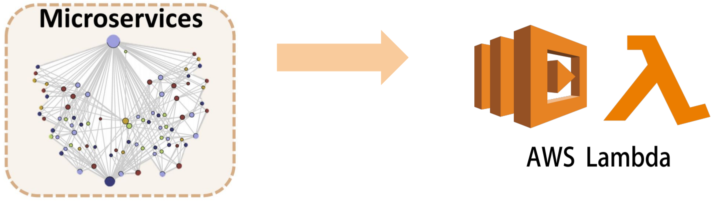
- Organize online applications with *single-purpose*, *loosely-coupled* microservices
- Benefits: composable software design



Motivation: Serverless Microservices

- Microservices are mostly implemented as RPC servers
- Stateless RPC handlers naturally fit in the FaaS paradigm

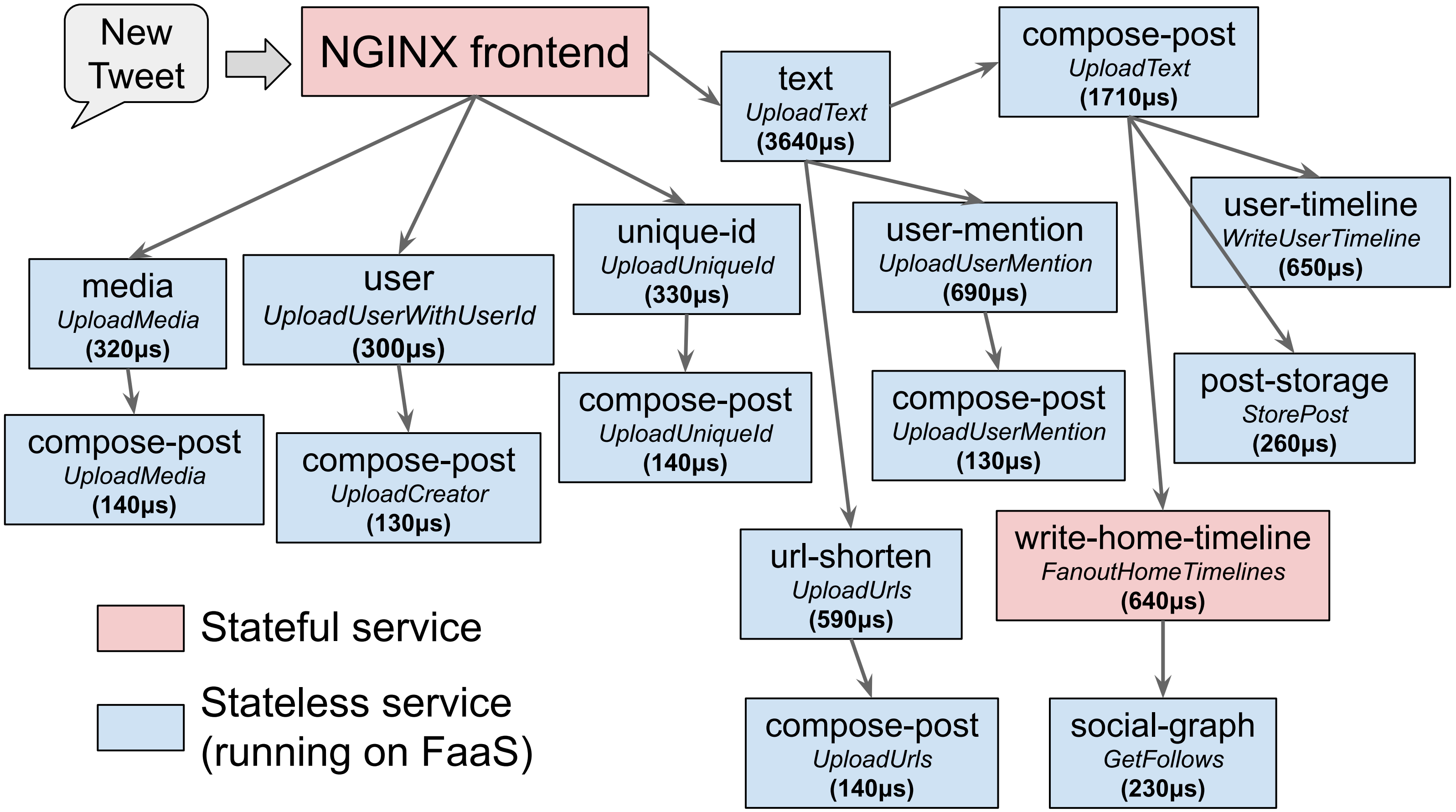
RPC servers *stateless RPC handlers*



But not performant !!

	RPC servers	AWS Lambda
median latency	2.34ms	26.94ms (**11.5x**)
tail latency	6.48ms	160.8ms (**24.8x**)

SocialNetwork microservices from DeathStarBench [ASPLOS ‘19]
Running under light load (100 QPS)



New Tweet

NGINX frontend

text *UploadText* **(3640µs)**

compose-post *UploadText* **(1710µs)**

media *UploadMedia* **(320µs)**

compose-post *UploadMedia* **(140µs)**

user *UploadUserWithUserId* **(300µs)**

compose-post *UploadCreator* **(130µs)**

unique-id *UploadUniqueId* **(330µs)**

compose-post *UploadUniqueId* **(140µs)**

user-mention *UploadUserMention* **(690µs)**

compose-post *UploadUserMention* **(130µs)**

url-shorten *UploadUrls* **(590µs)**

compose-post *UploadUrls* **(140µs)**

user-timeline *WriteUserTimeline* **(650µs)**

post-storage *StorePost* **(260µs)**

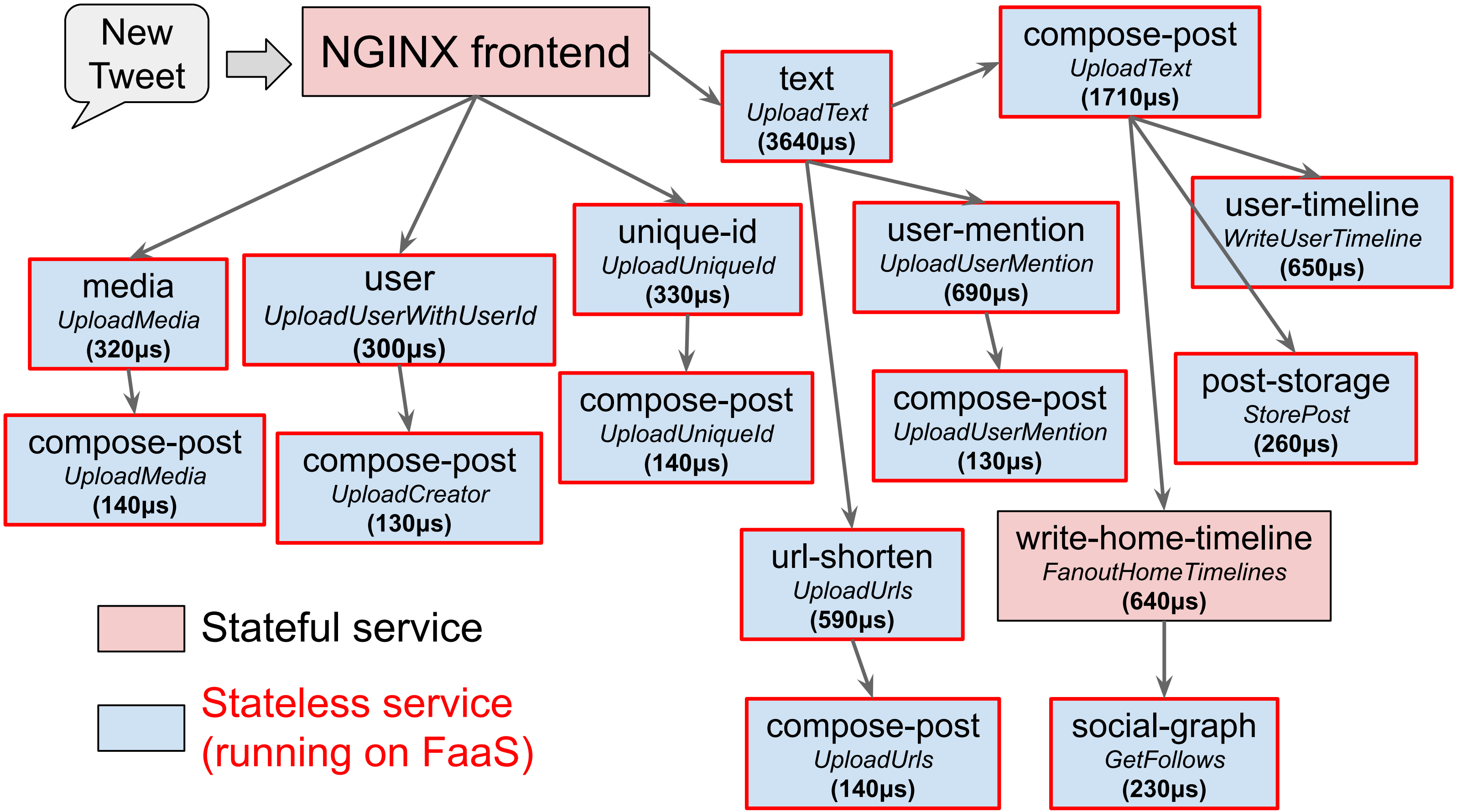
write-home-timeline *FanoutHomeTimelines* **(640µs)**

social-graph *GetFollows* **(230µs)**

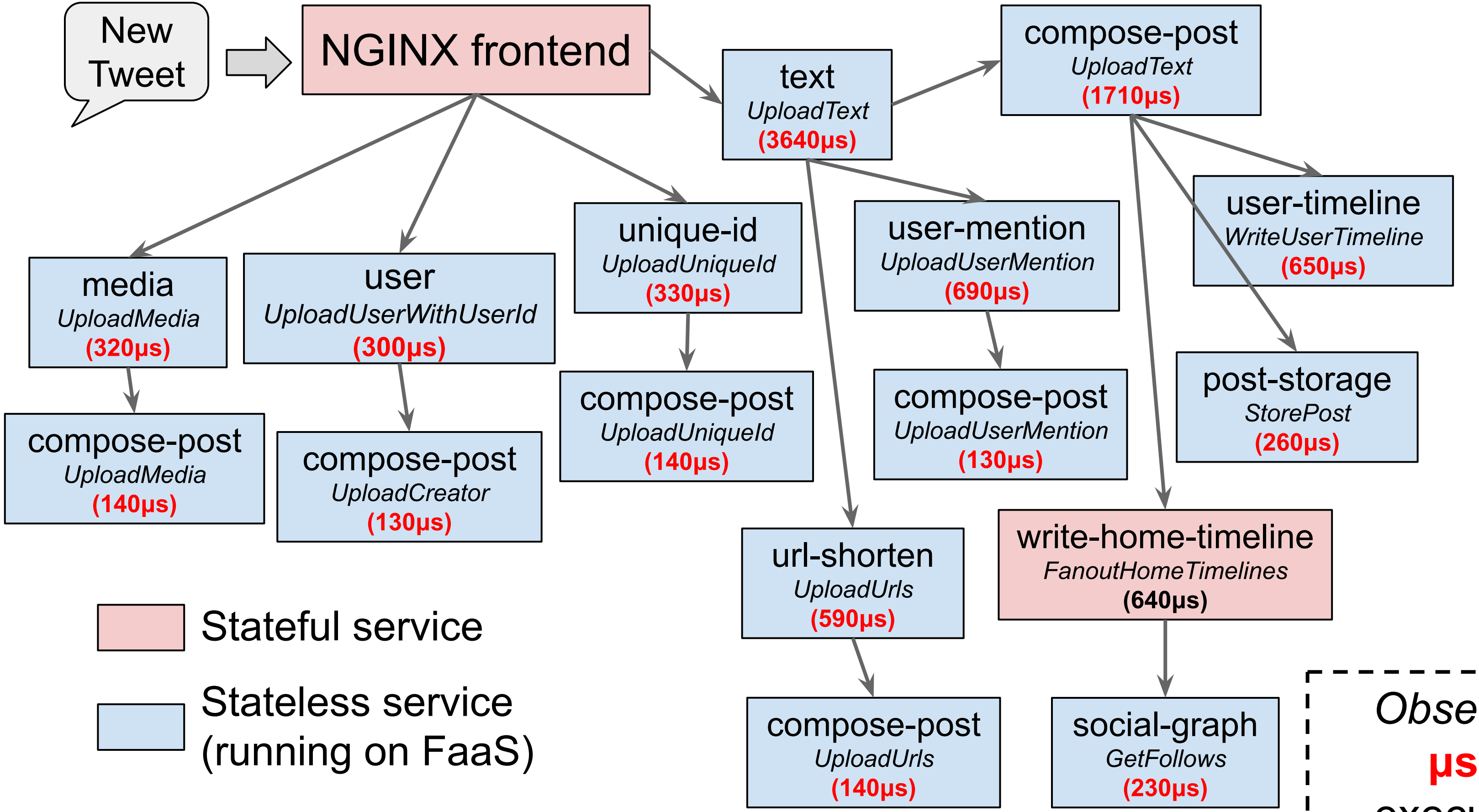
🟥 Stateful service

🟦 Stateless service (running on FaaS)

RPC trace from SocialNetwork microservices

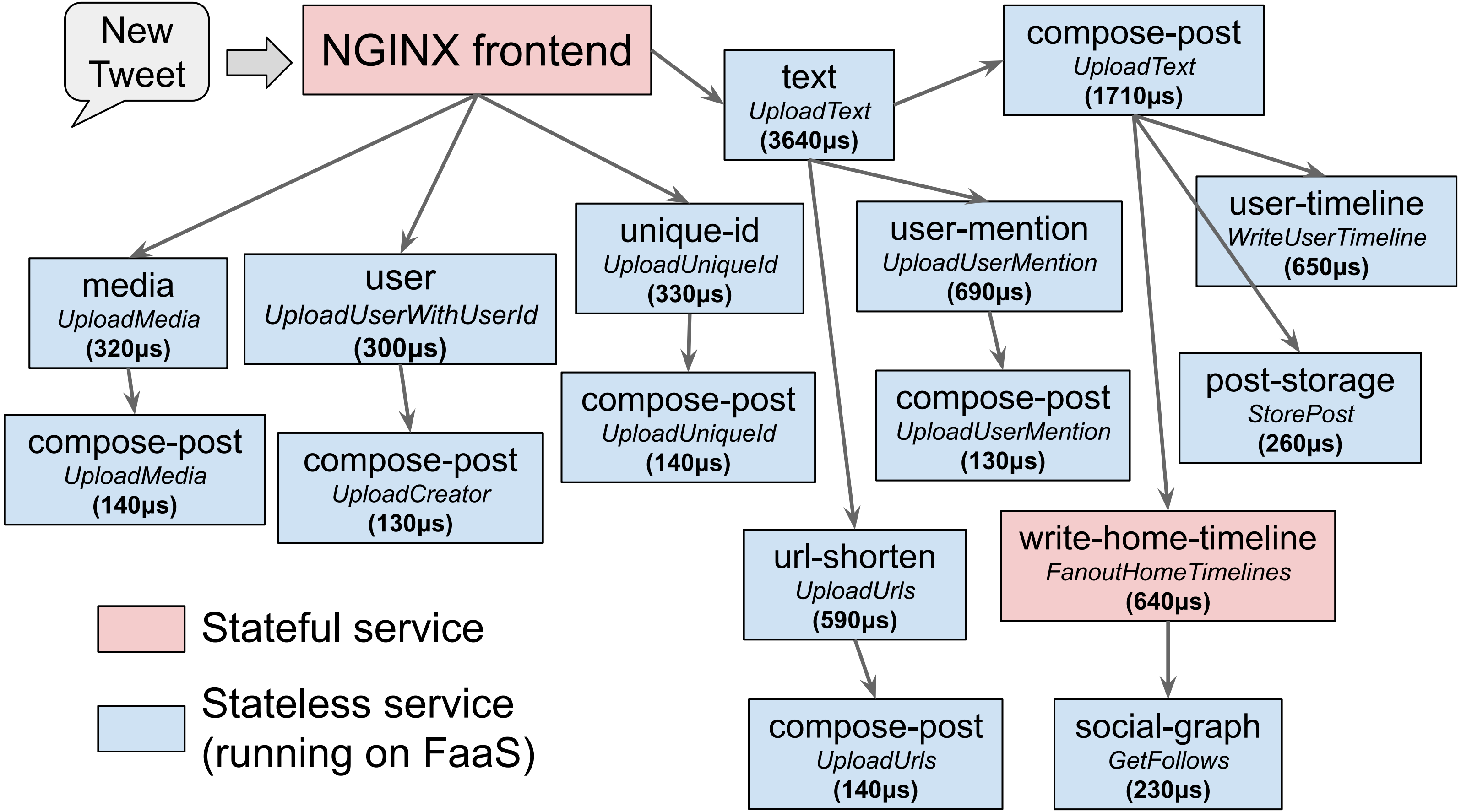


RPC trace from SocialNetwork microservices



Observation 1:
μs-scale
execution time

RPC trace from SocialNetwork microservices



New Tweet

NGINX frontend

media *UploadMedia* **(320μs)**

compose-post *UploadMedia* **(140μs)**

user *UploadUserWithUserId* **(300μs)**

compose-post *UploadCreator* **(130μs)**

unique-id *UploadUniqueId* **(330μs)**

compose-post *UploadUniqueId* **(140μs)**

text *UploadText* **(3640μs)**

user-mention *UploadUserMention* **(690μs)**

compose-post *UploadUserMention* **(130μs)**

url-shorten *UploadUrls* **(590μs)**

compose-post *UploadUrls* **(140μs)**

compose-post *UploadText* **(1710μs)**

user-timeline *WriteUserTimeline* **(650μs)**

post-storage *StorePost* **(260μs)**

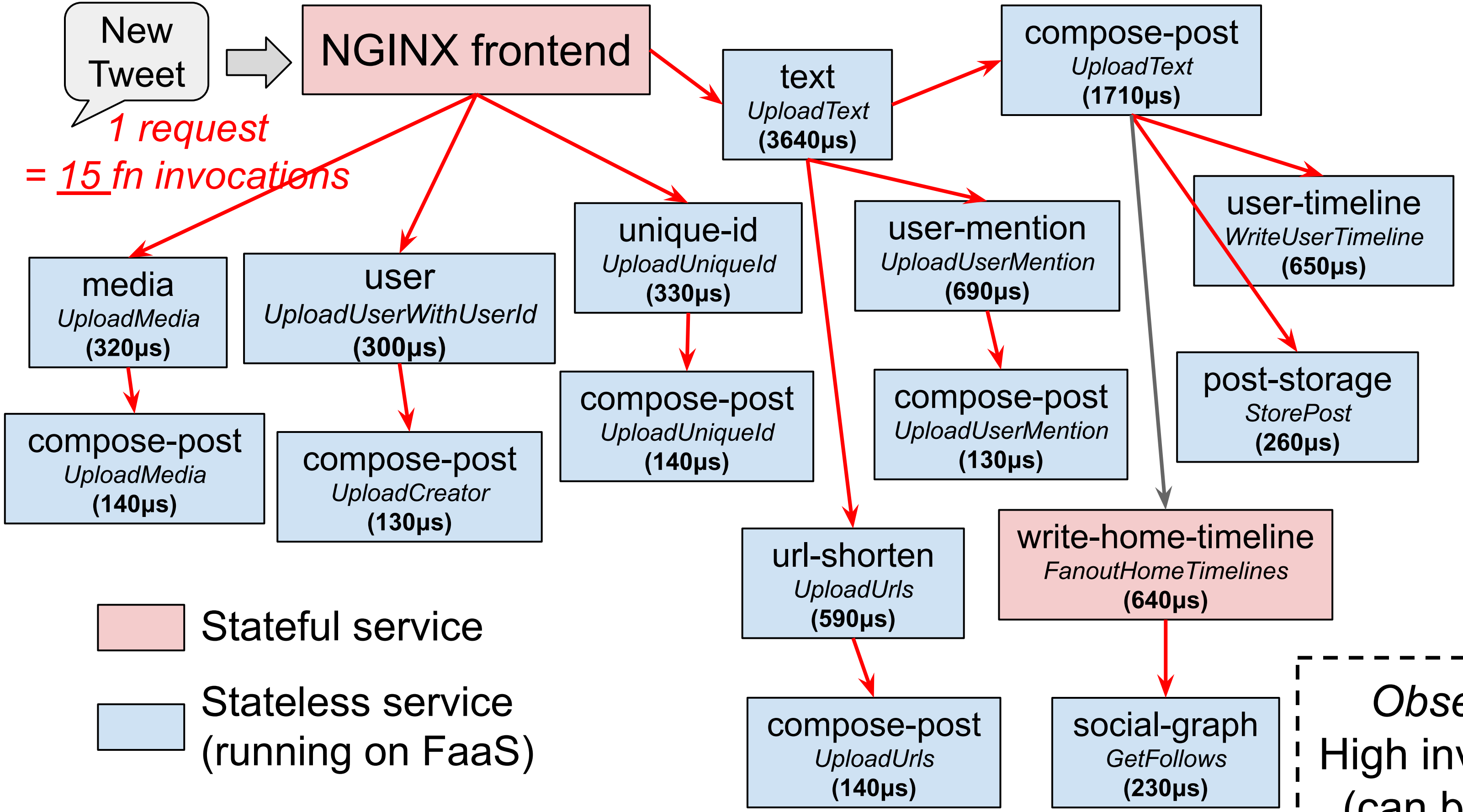
write-home-timeline *FanoutHomeTimelines* **(640μs)**

social-graph *GetFollows* **(230μs)**

🟥 Stateful service

🟦 Stateless service (running on FaaS)

RPC trace from SocialNetwork microservices



Observation 2:
High invocation rate
(can be >**100K/s**)

RPC trace from SocialNetwork microservices

Performance Goals for Nightcore

- Observation 1: **μs-scale** execution time
- Observation 2: high invocation rate (>**100K/s**)

	Function Execution Time	**Invocation Latency**	**Invocation Rate**
Current FaaS runtime	>100ms	1-10s of ms	<10K/min

Current FaaS workloads: video processing, distributed compilation, data analytics, etc.

Invocation latency: duration between function request and the start of function execution

Performance Goals for Nightcore

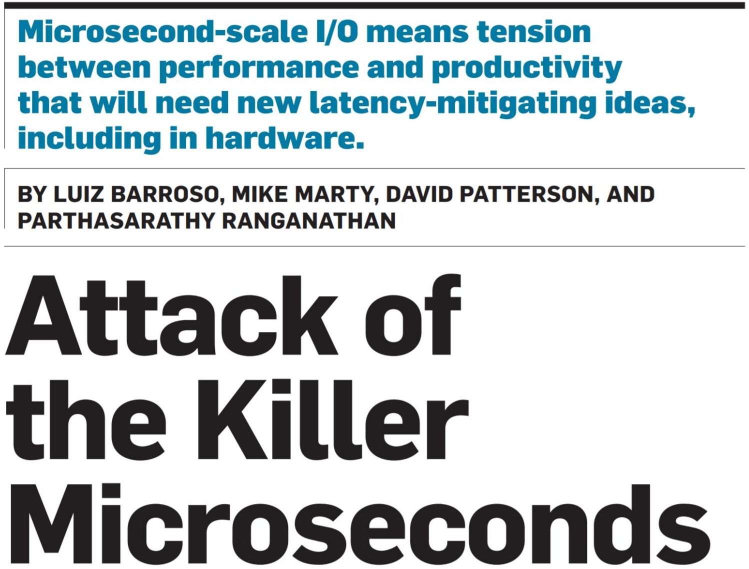
- Observation 1: **µs-scale** execution time
- Observation 2: high invocation rate (>**100K/s**)

	Function Execution Time	**Invocation Latency**	**Invocation Rate**
Current FaaS runtime	>100ms	1-10s of ms	<10K/min
FaaS runtime for microservices	100s of µs	<100µs	>100K/s

Our performance goals

Invocation latency: duration between function request and the start of function execution

Nightcore's Goals are Challenging Because We Are Vulnerable to *Killer Microseconds*



Microsecond-scale I/O means tension between performance and productivity that will need new latency-mitigating ideas, including in hardware.

BY LUIZ BARROSO, MIKE MARTY, DAVID PATTERSON, AND PARTHASARATHY RANGANATHAN

Attack of the Killer Microseconds

Communications of the ACM | March 2017

Microsecond-scale events:

- Networking
- TCP/IP stack
- RPC protocol
- Context switch
- Thread scheduling
- ……

Where hides our *killer microseconds*?

Nightcore Design

Hunting for “the killer microseconds” in the regime of FaaS

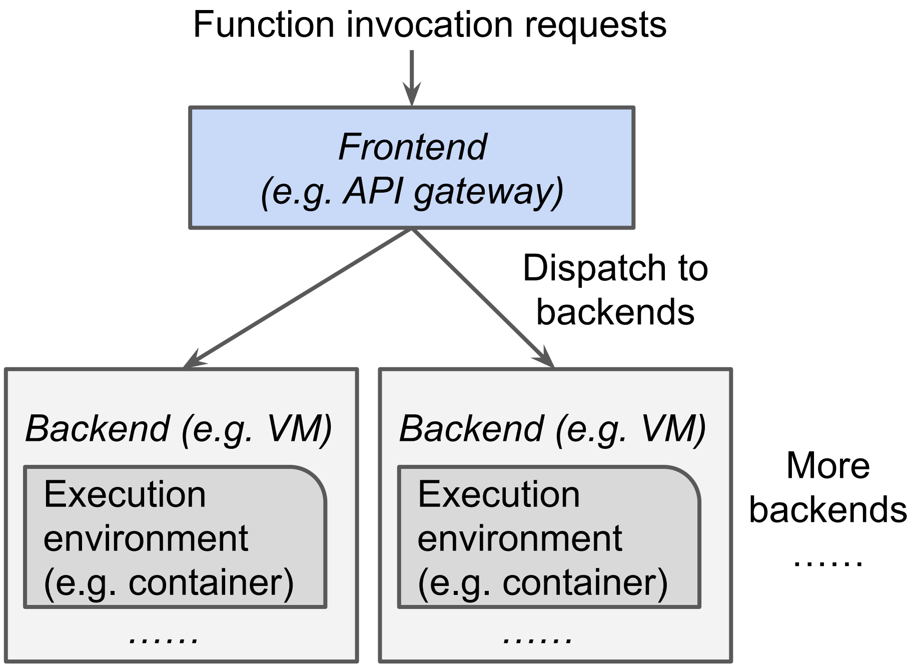
Nightcore’s Techniques

- Optimizing locality of internal function calls
- High optimizations for local I/Os
  - Low-latency message channels
  - Event-driven concurrency
- Managing per-microservice concurrency to mitigate load variation

Nightcore’s Techniques

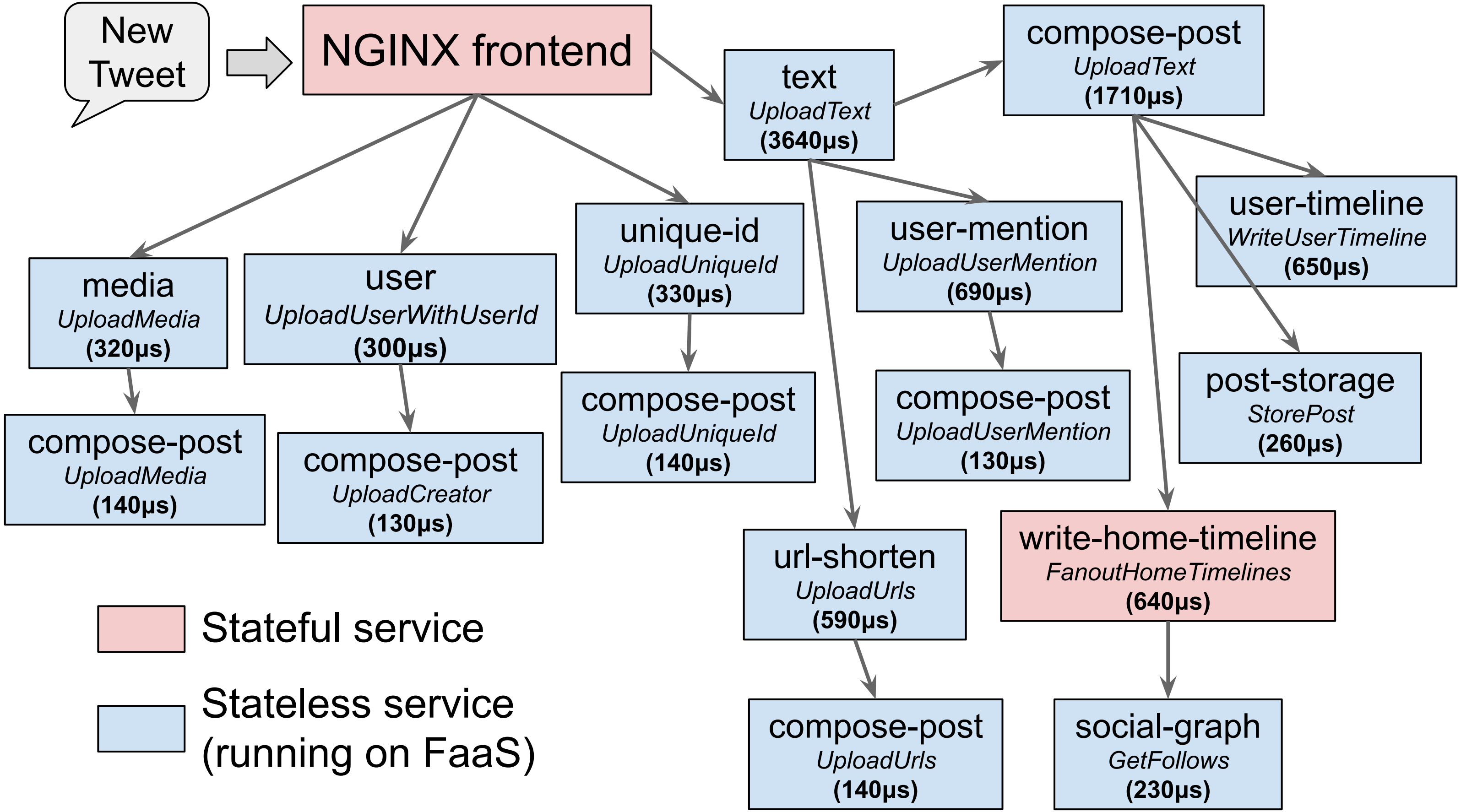
- Optimizing locality of internal function calls
- High optimizations for local I/Os
  - Low-latency message channels
  - Event-driven concurrency
- Managing per-microservice concurrency to mitigate load variation

High-Level Design of a FaaS Runtime



Separation of frontend and backend

- Adopted by Apache OpenWhisk and OpenFaaS
- Scaling the system by adding backends



New Tweet

NGINX frontend

media *UploadMedia* **(320μs)**

compose-post *UploadMedia* **(140μs)**

user *UploadUserWithUserId* **(300μs)**

compose-post *UploadCreator* **(130μs)**

unique-id *UploadUniqueId* **(330μs)**

compose-post *UploadUniqueId* **(140μs)**

text *UploadText* **(3640μs)**

user-mention *UploadUserMention* **(690μs)**

compose-post *UploadUserMention* **(130μs)**

url-shorten *UploadUrls* **(590μs)**

compose-post *UploadUrls* **(140μs)**

compose-post *UploadText* **(1710μs)**

user-timeline *WriteUserTimeline* **(650μs)**

post-storage *StorePost* **(260μs)**

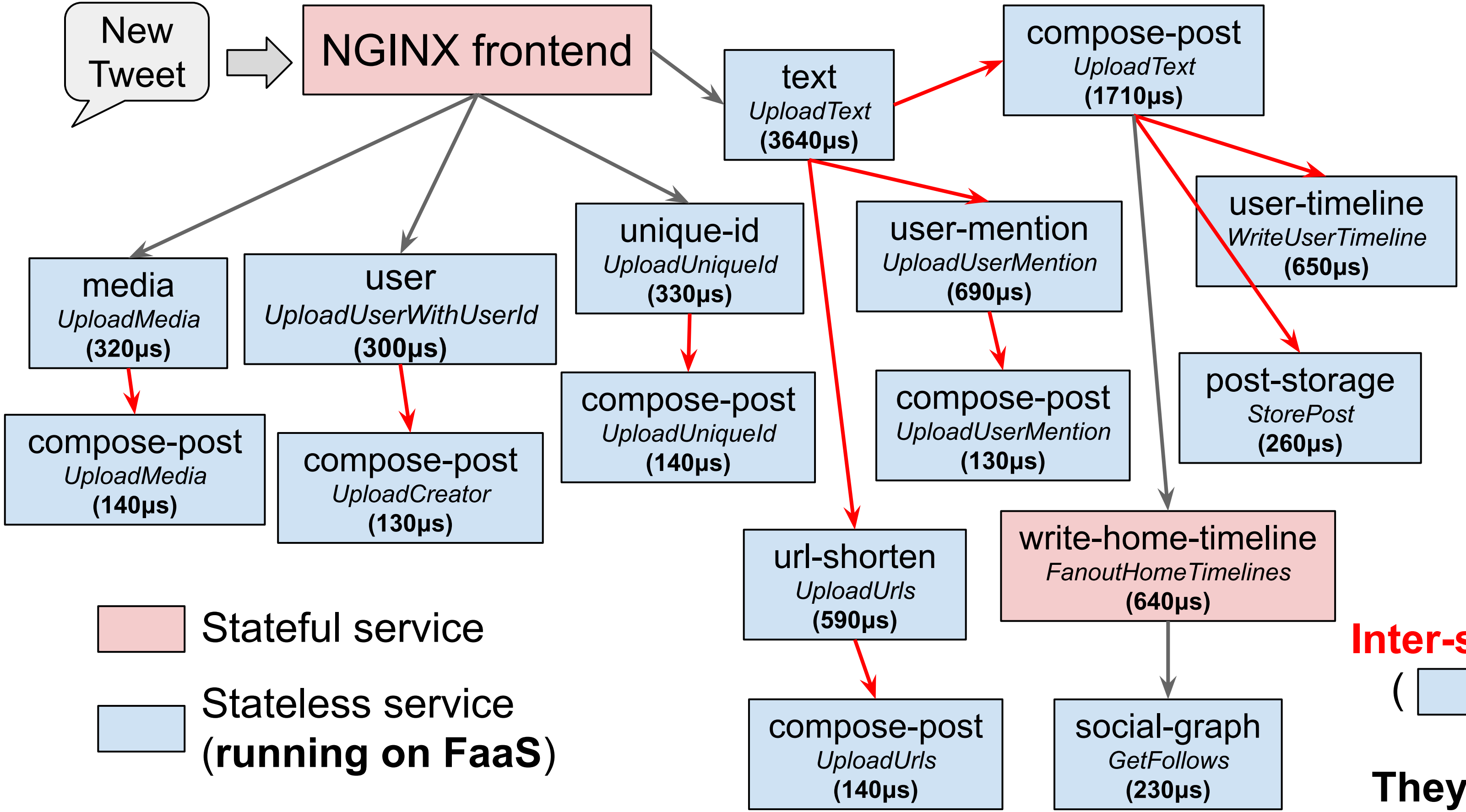
write-home-timeline *FanoutHomeTimelines* **(640μs)**

social-graph *GetFollows* **(230μs)**

🟥 Stateful service

🟦 Stateless service (running on FaaS)

RPC trace from SocialNetwork microservices



New Tweet

NGINX frontend

text
UploadText
(3640µs)

compose-post
UploadText
(1710µs)

media
UploadMedia
(320µs)

user
UploadUserWithUserId
(300µs)

unique-id
UploadUniqueId
(330µs)

user-mention
UploadUserMention
(690µs)

user-timeline
WriteUserTimeline
(650µs)

compose-post
UploadMedia
(140µs)

compose-post
UploadCreator
(130µs)

compose-post
UploadUniqueId
(140µs)

compose-post
UploadUserMention
(130µs)

post-storage
StorePost
(260µs)

url-shorten
UploadUrls
(590µs)

write-home-timeline
FanoutHomeTimelines
(640µs)

compose-post
UploadUrls
(140µs)

social-graph
GetFollows
(230µs)

🟥 Stateful service

🟦 Stateless service
(**running on FaaS**)

Inter-service calls
(🟦 to 🟦)

They're <u>internal</u>
w.r.t. FaaS system

RPC trace from SocialNetwork microservices

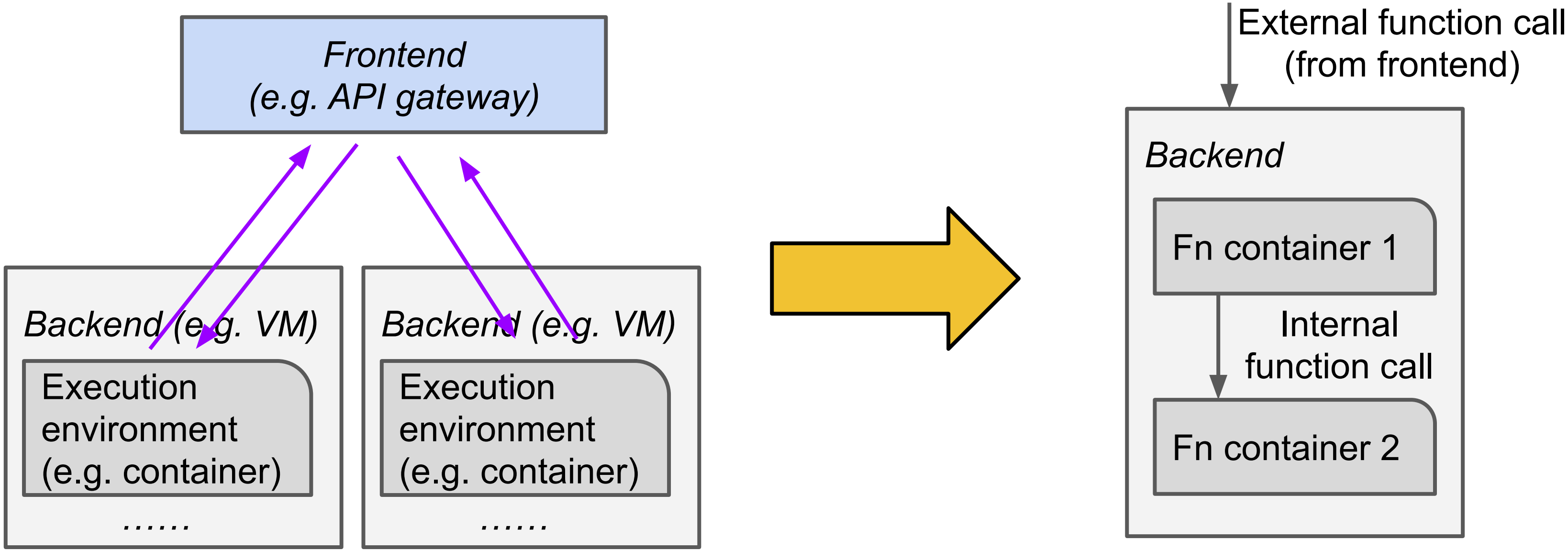
Observation: High Ratio of Internal Calls

Function calls that are internal w.r.t. FaaS system

Frequent in microservices

Microservice workloads	Social Network		Movie Reviewing	Hotel Reservation	Hipster Shop
	write	mixed			
% of internal fn calls	**66.7%**	**62.3%**	**69.2%**	**79.2%**	**85.1%**

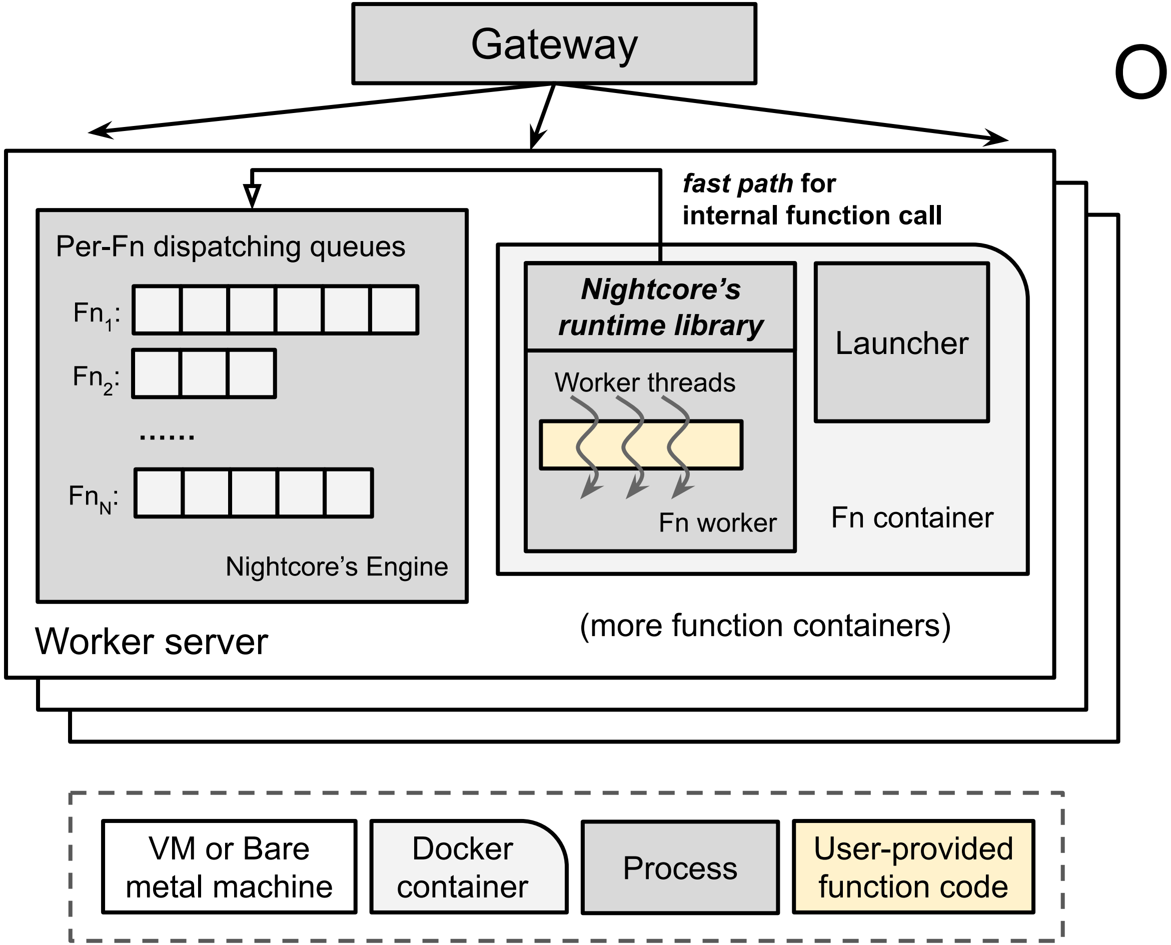
Optimizing Locality for Internal Function Calls



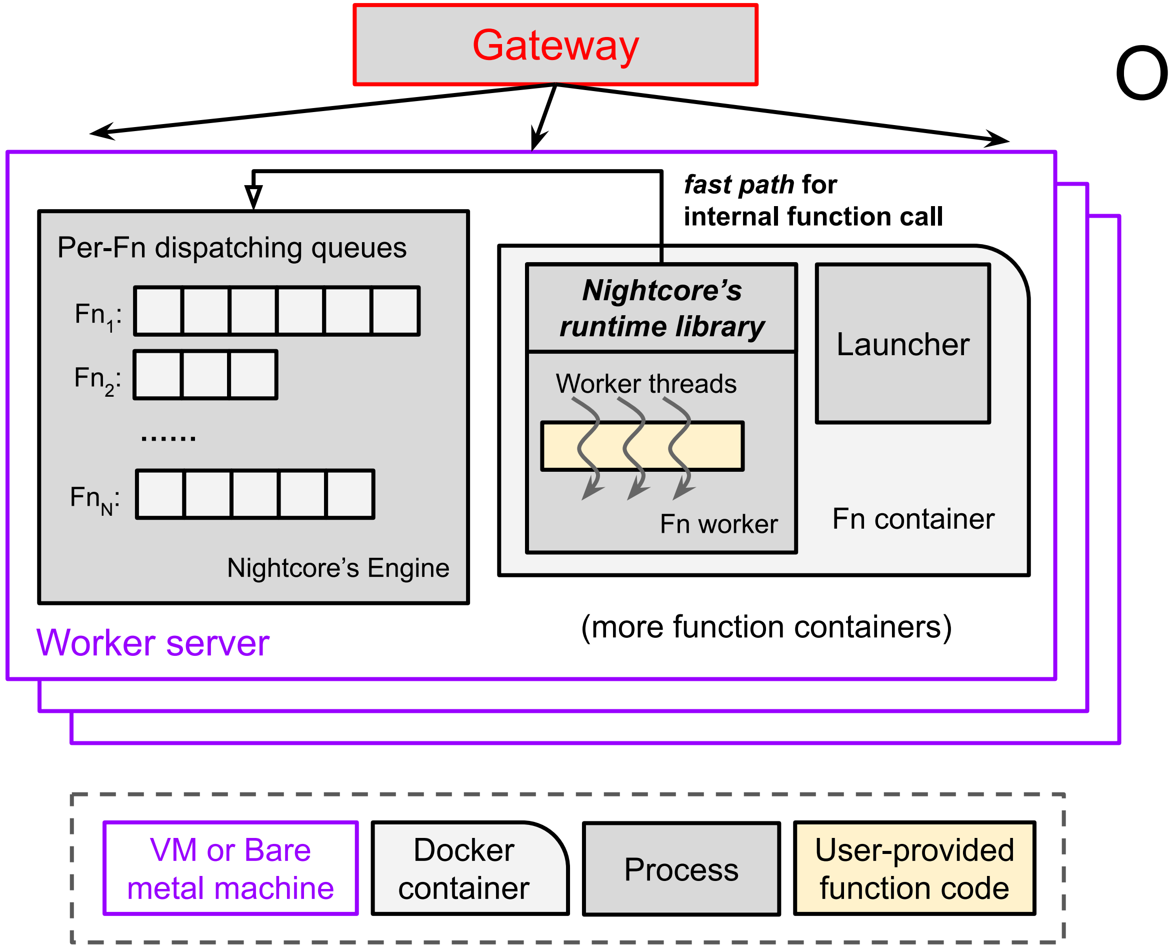
Internal function calls always go through frontend

Skip frontend for internal function calls

Overview of Nightcore

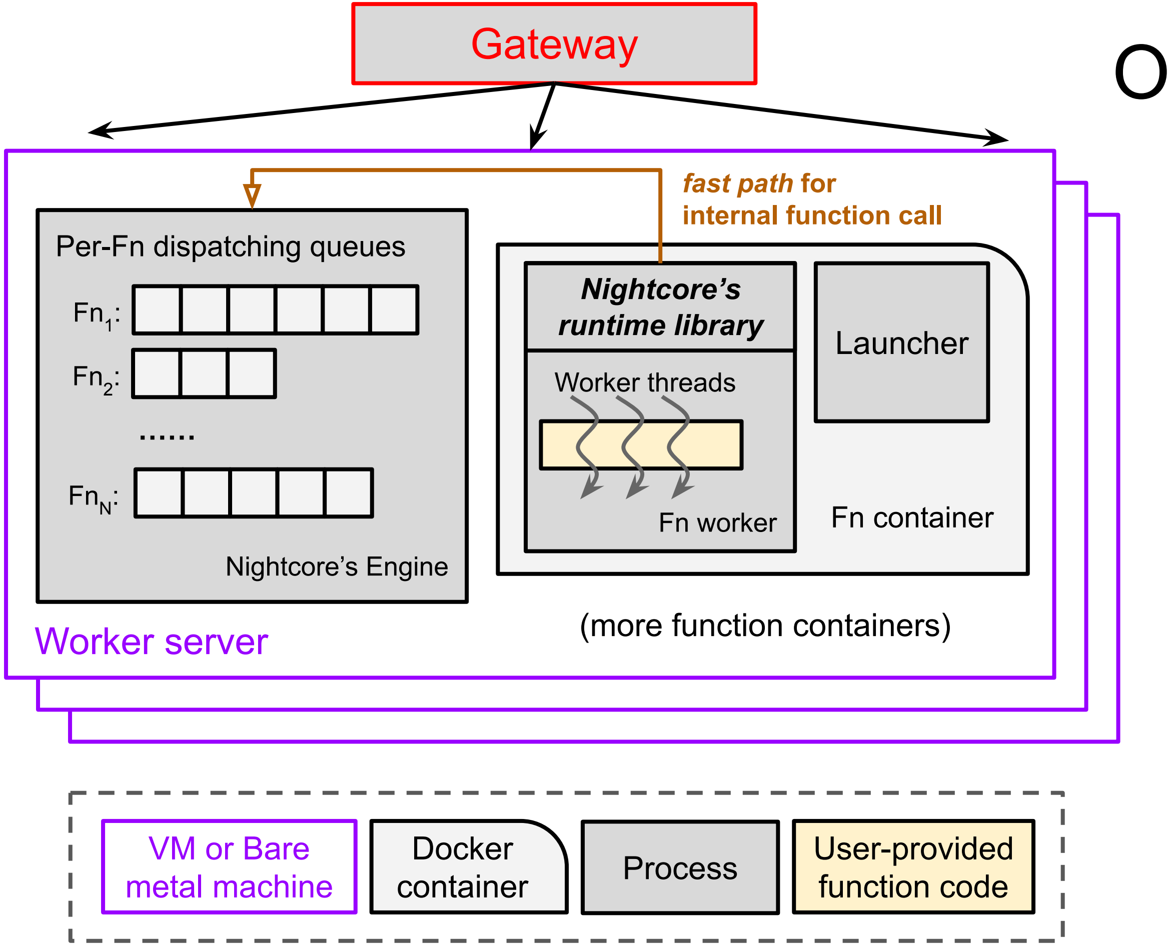


Overview of Nightcore



Frontend and *Backend*

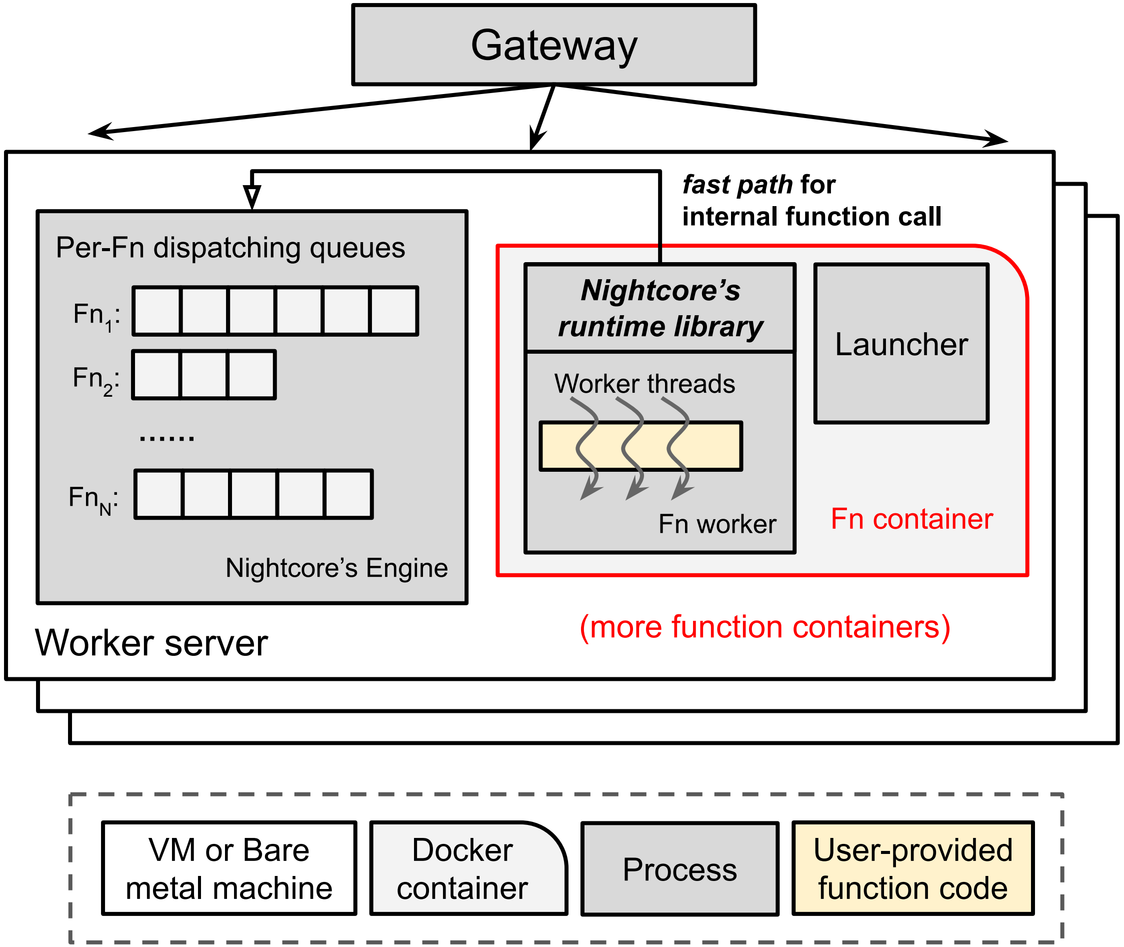
Overview of Nightcore

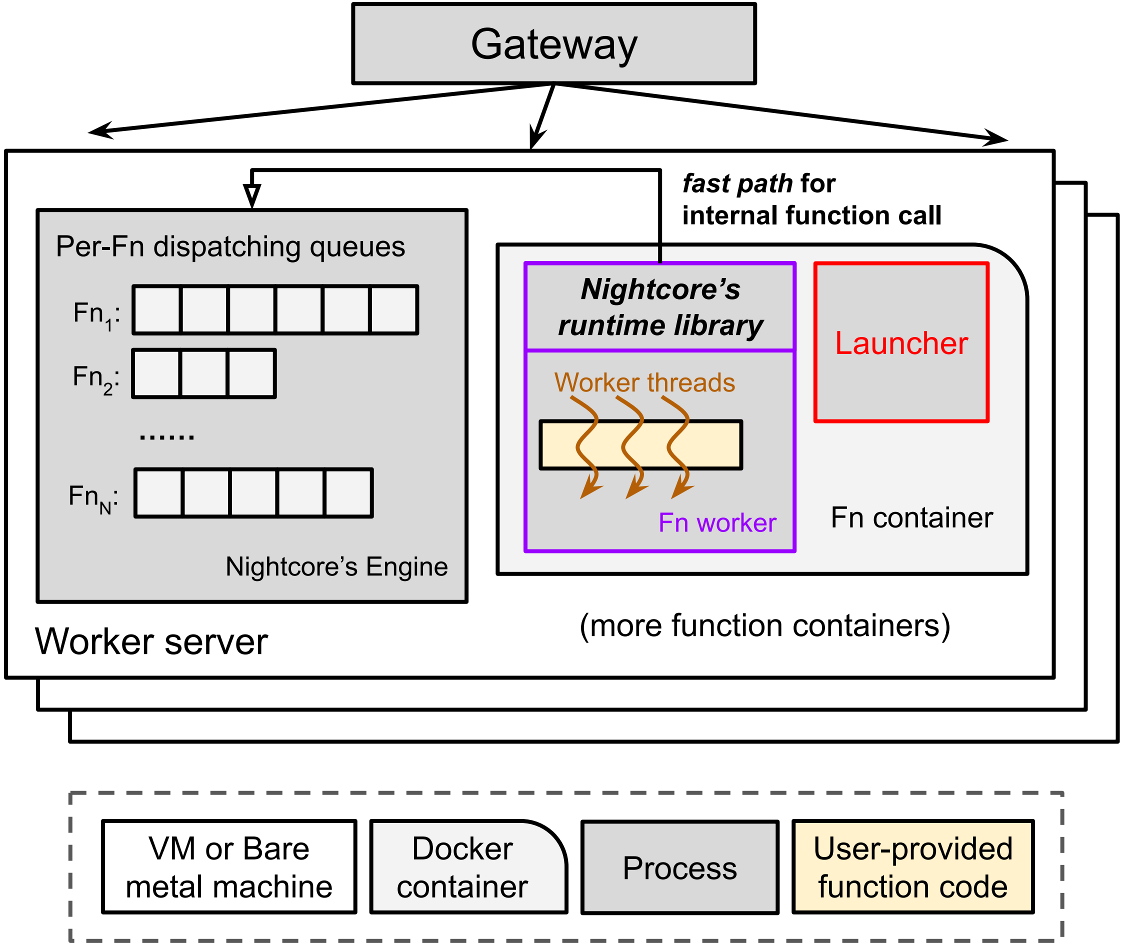


Fast path for
internal function calls

Function Containers

Execution environments for serverless functions





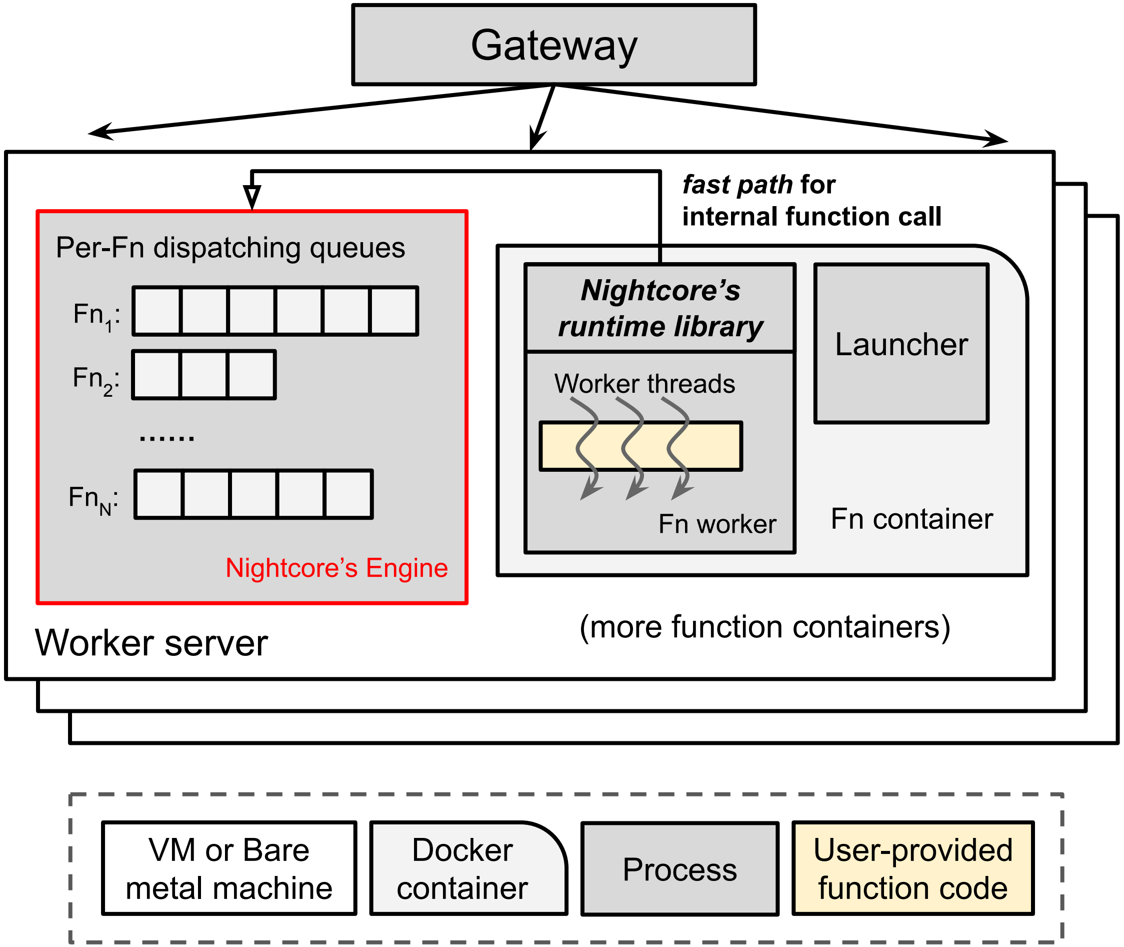
Function Containers

Execution environments for serverless functions

Launcher launches new function workers, and worker threads

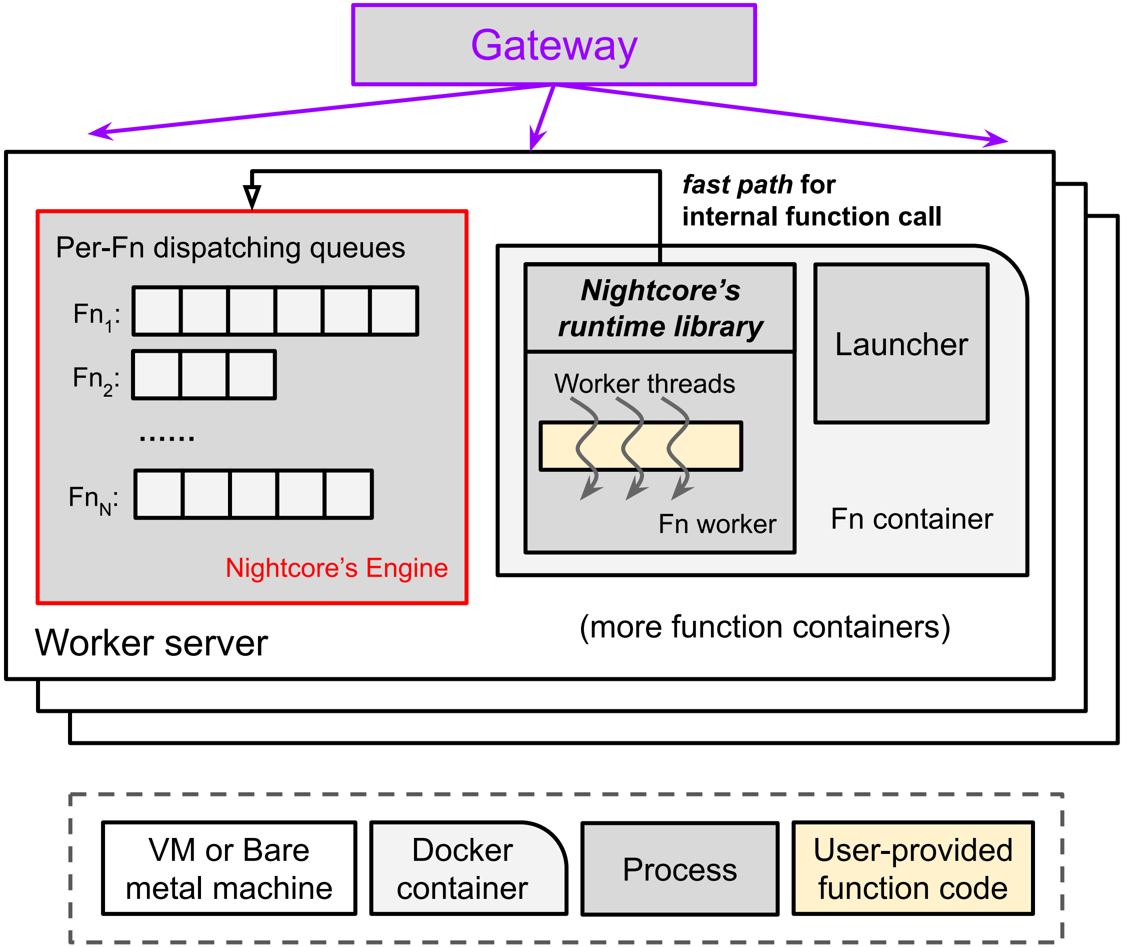
Nightcore’s Engine

The main Nightcore process running on each worker server



Nightcore's Engine

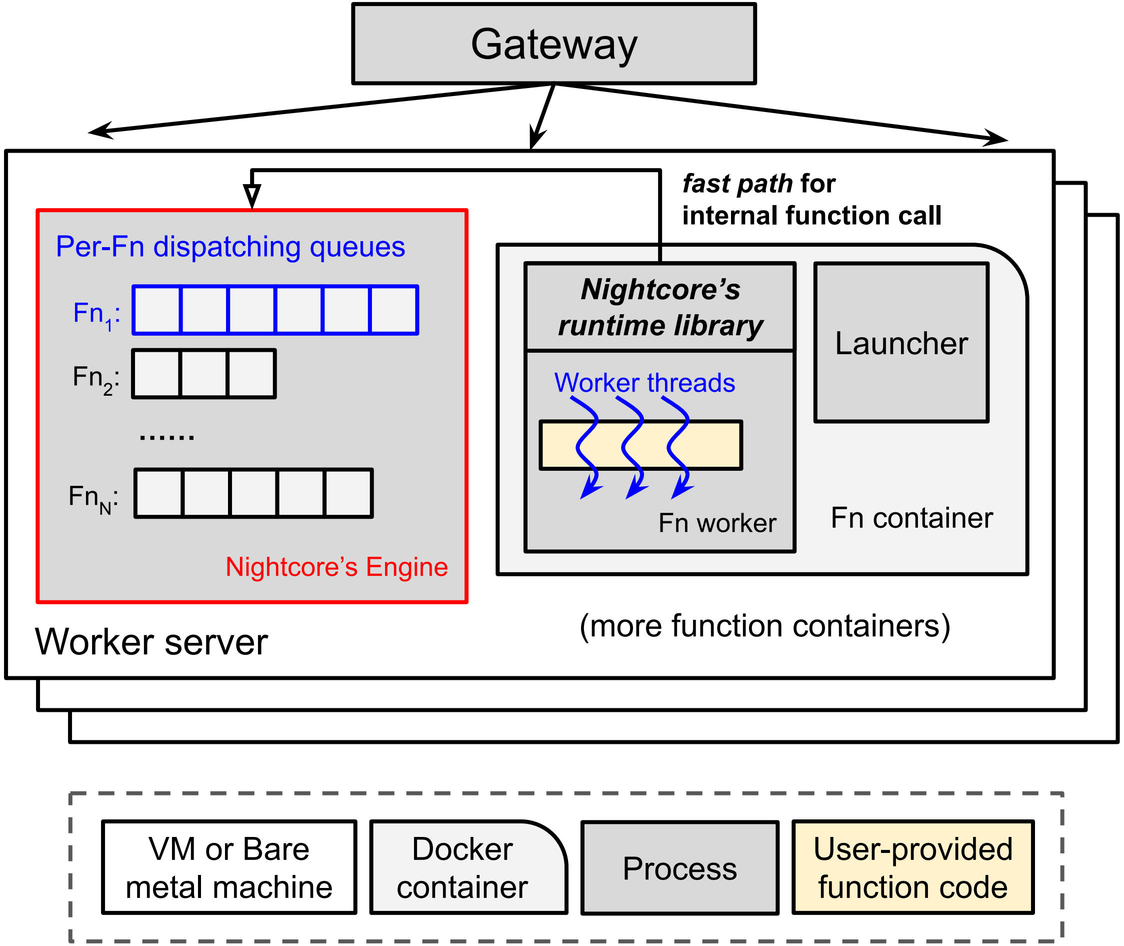
The main Nightcore process running on each worker server



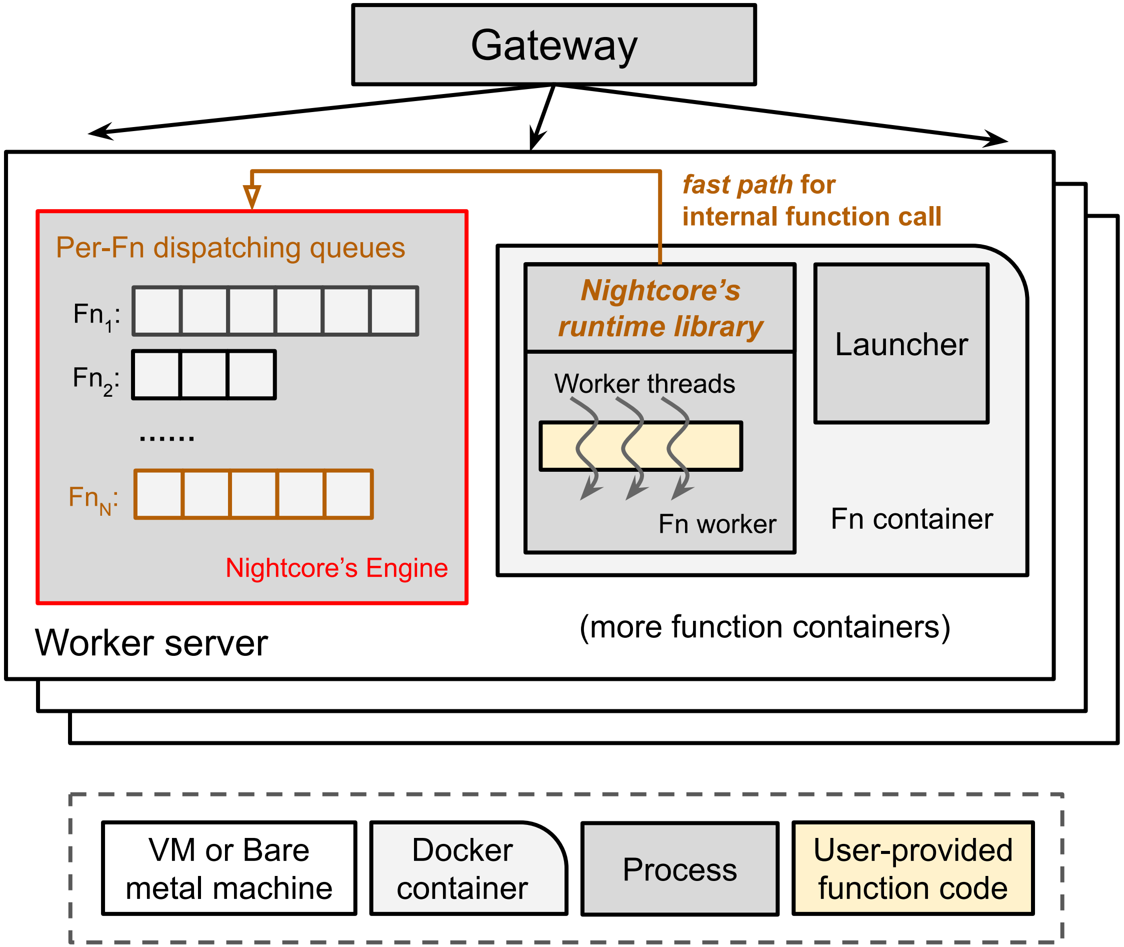
Receive *external* function requests from Gateway

Nightcore's Engine

The main Nightcore process running on each worker server



Dispatch function requests to worker threads

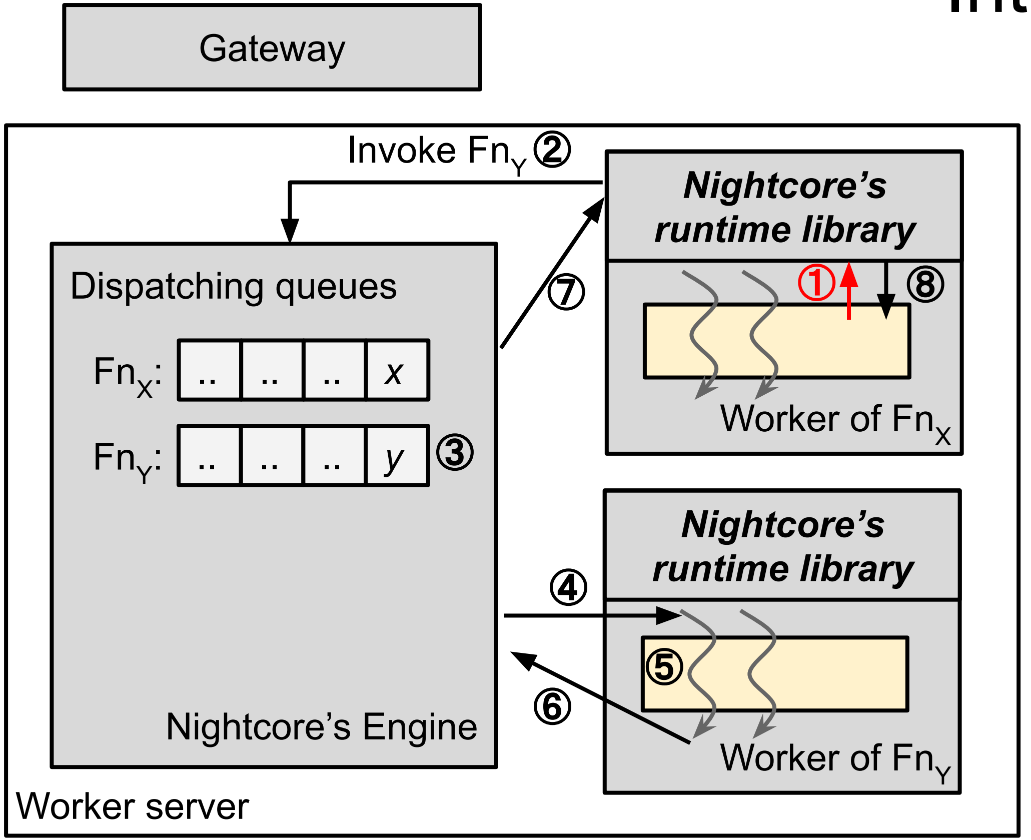


Nightcore's Engine

The main Nightcore process running on each worker server

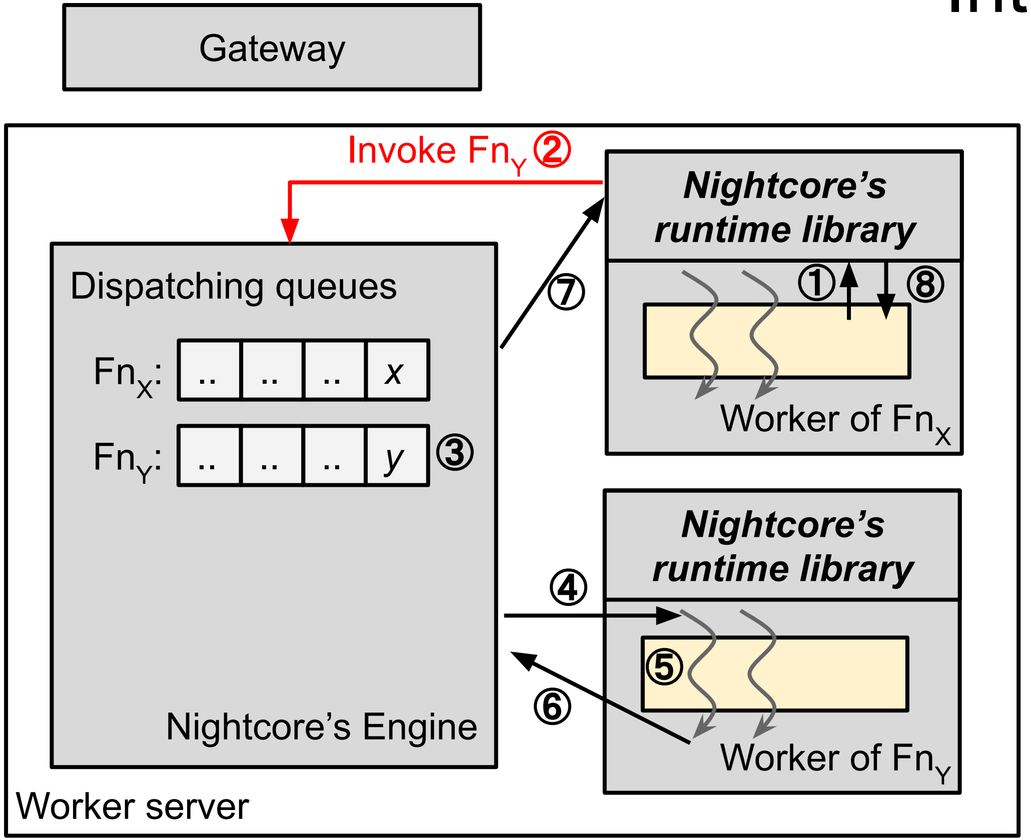
Fast path for internal function calls

Internal Function Request



① Fn_y invoked via Nightcore's runtime API

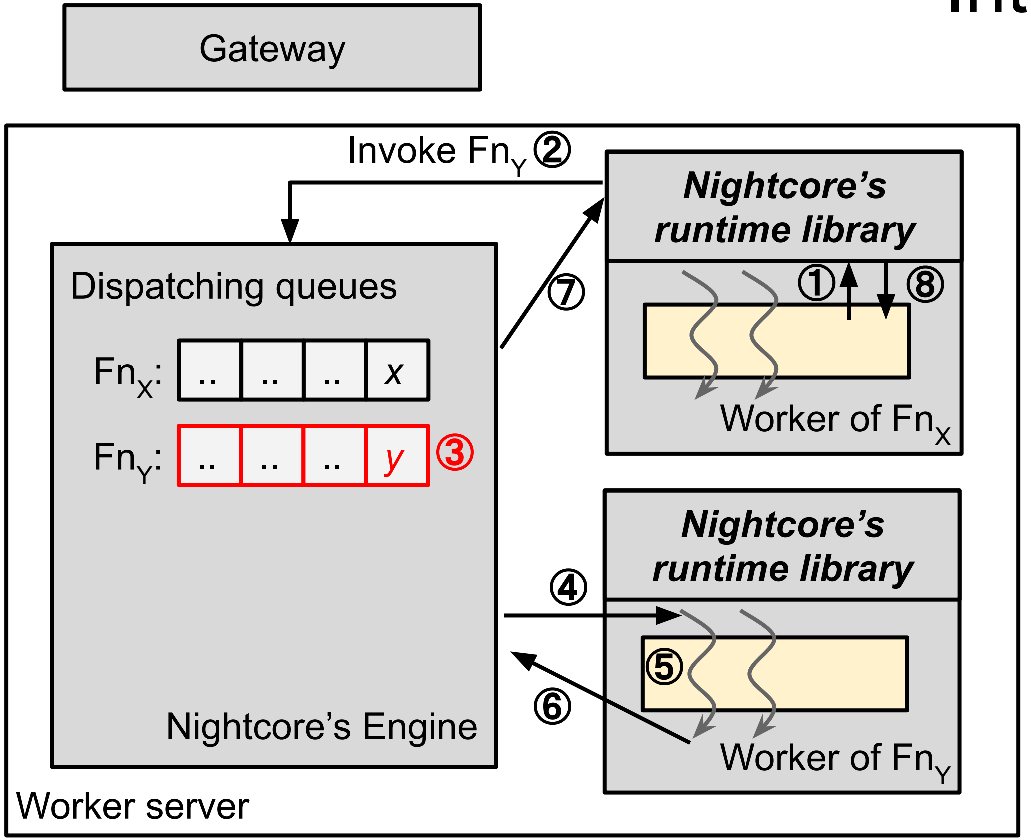
Internal Function Request


Gateway
Invoke Fn_Y ②
Nightcore's runtime library
①
⑧
Worker of Fn_X
⑦
Dispatching queues
Fn_X: x
Fn_Y: .. .. .. y
③
Nightcore's Engine
④
⑥
Nightcore's runtime library
⑤
Worker of Fn_Y
Worker server

① Fn_y invoked via Nightcore's runtime API

② Req_y sent to Nightcore's engine

Internal Function Request

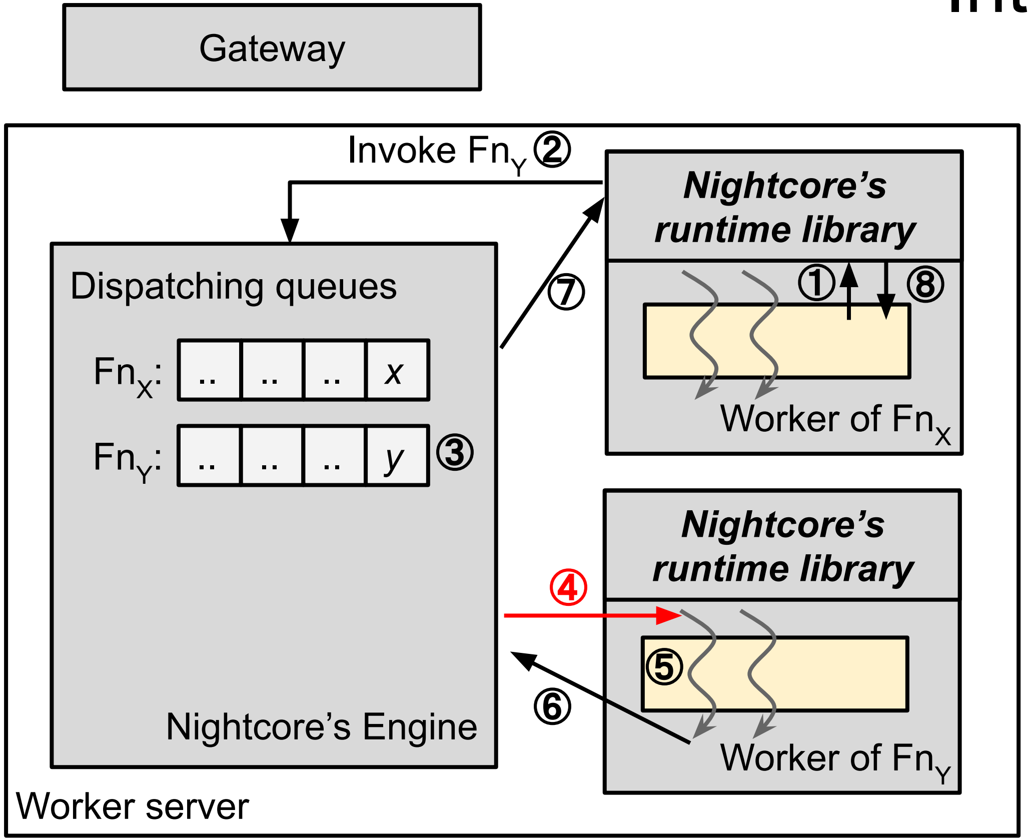


① Fn_y invoked via Nightcore's runtime API

② Req_y sent to Nightcore's engine

③ Place req_y in the dispatching queue

Internal Function Request



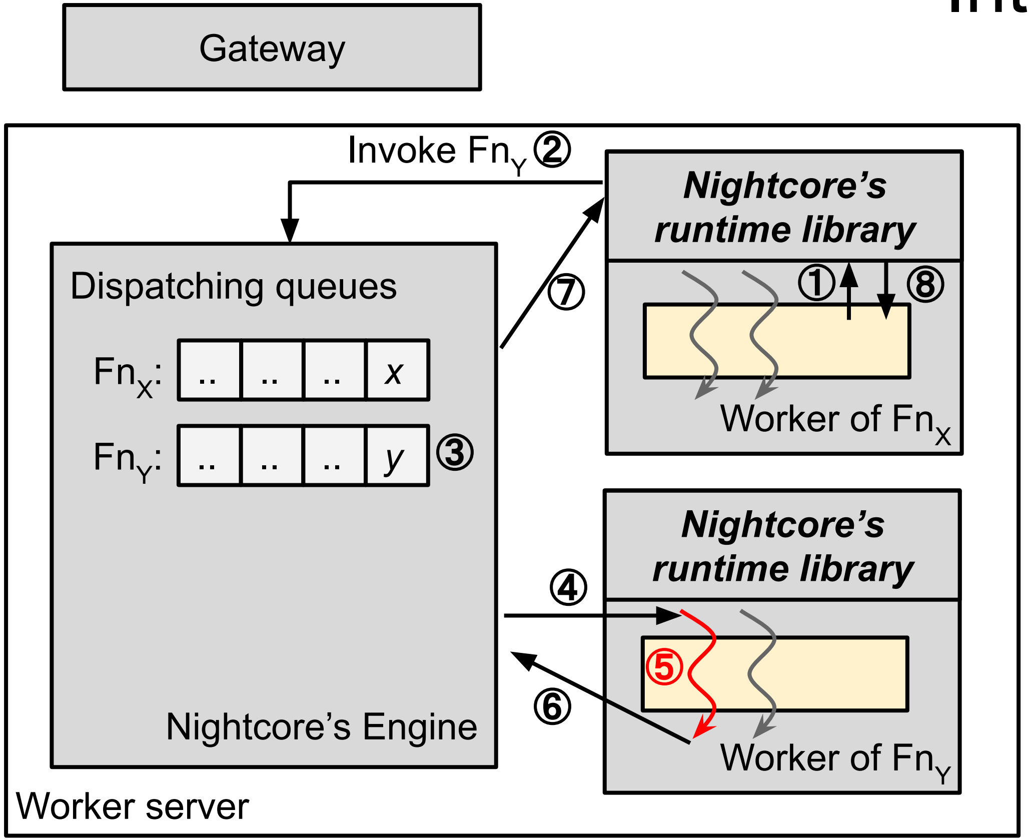
① Fn$_y$ invoked via Nightcore's runtime API

② Req$_y$ sent to Nightcore's engine

③ Place req$_y$ in the dispatching queue

④ Dispatch Req$_y$ to worker of Fn$_y$

Internal Function Request



① Fn$_y$ invoked via Nightcore's runtime API

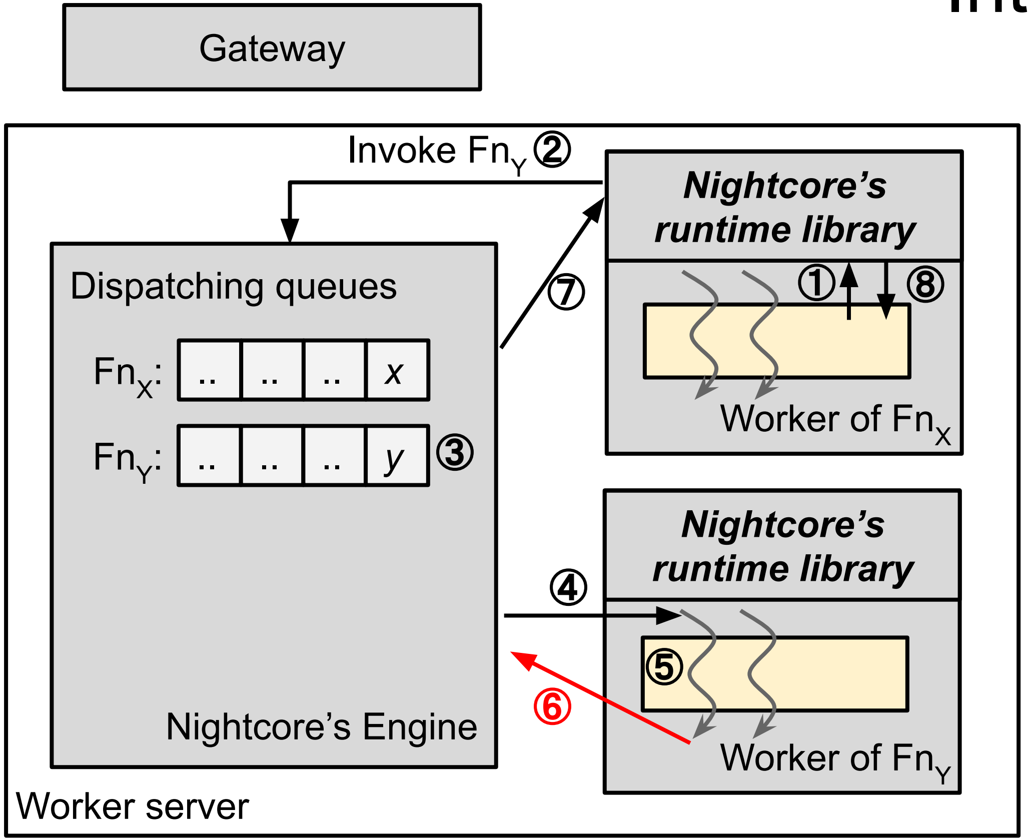
② Req$_y$ sent to Nightcore's engine

③ Place req$_y$ in the dispatching queue

④ Dispatch Req$_y$ to worker of Fn$_y$

⑤ Worker thread executes code of Fn$_y$

Internal Function Request



① Fn_y invoked via Nightcore's runtime API

② Req_y sent to Nightcore's engine

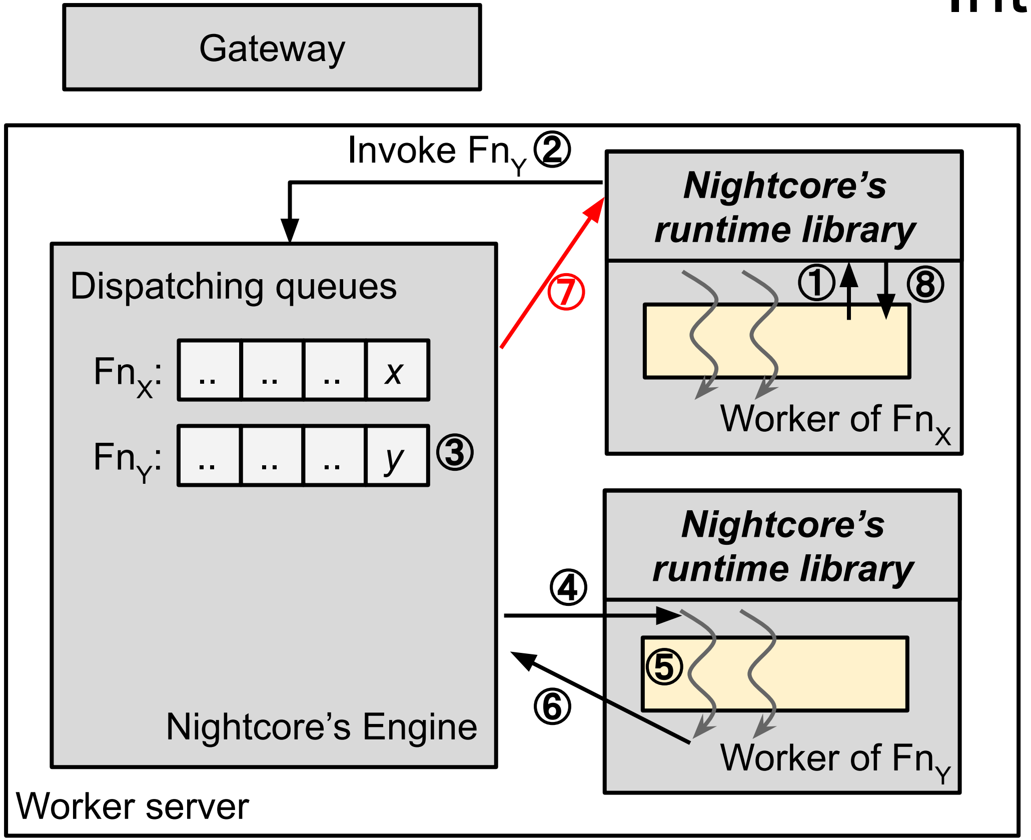
③ Place req_y in the dispatching queue

④ Dispatch Req_y to worker of Fn_y

⑤ Worker thread executes code of Fn_y

⑥ Execution of req_y completed

Internal Function Request



① Fn_y invoked via Nightcore's runtime API

② Req_y sent to Nightcore's engine

③ Place req_y in the dispatching queue

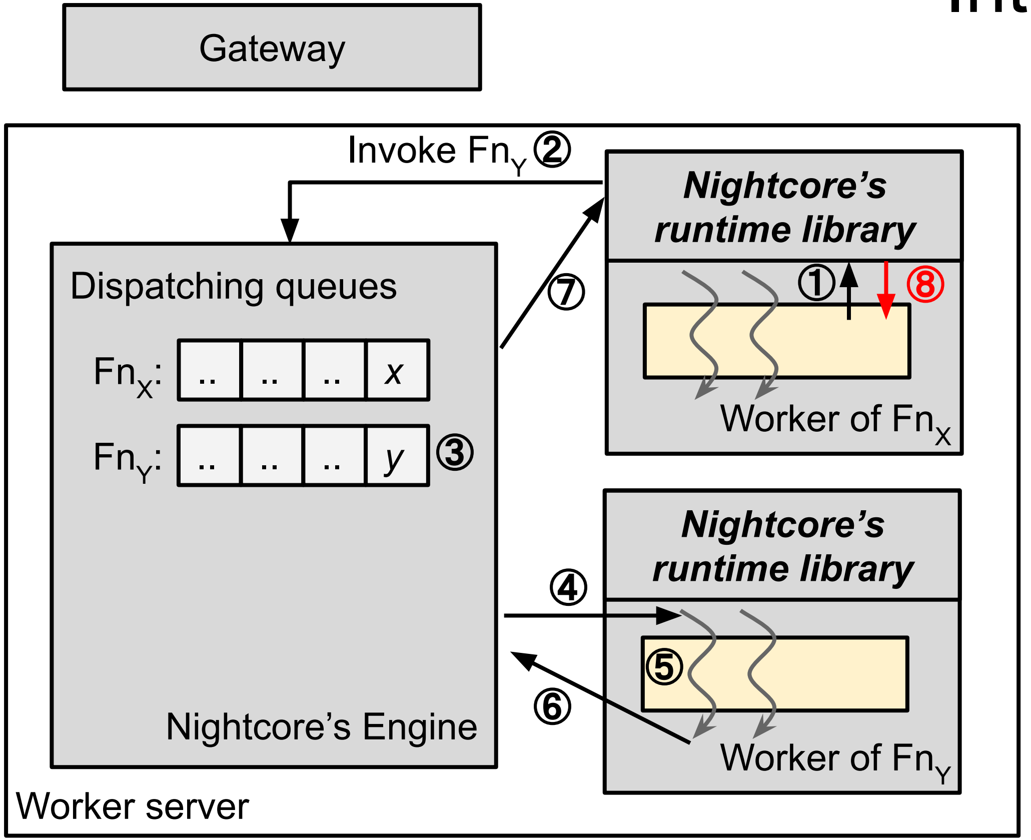
④ Dispatch Req_y to worker of Fn_y

⑤ Worker thread executes code of Fn_y

⑥ Execution of req_y completed

⑦ Send output back to worker of Fn_x

Internal Function Request



① Fn_y invoked via Nightcore's runtime API

② Req_y sent to Nightcore's engine

③ Place req_y in the dispatching queue

④ Dispatch Req_y to worker of Fn_y

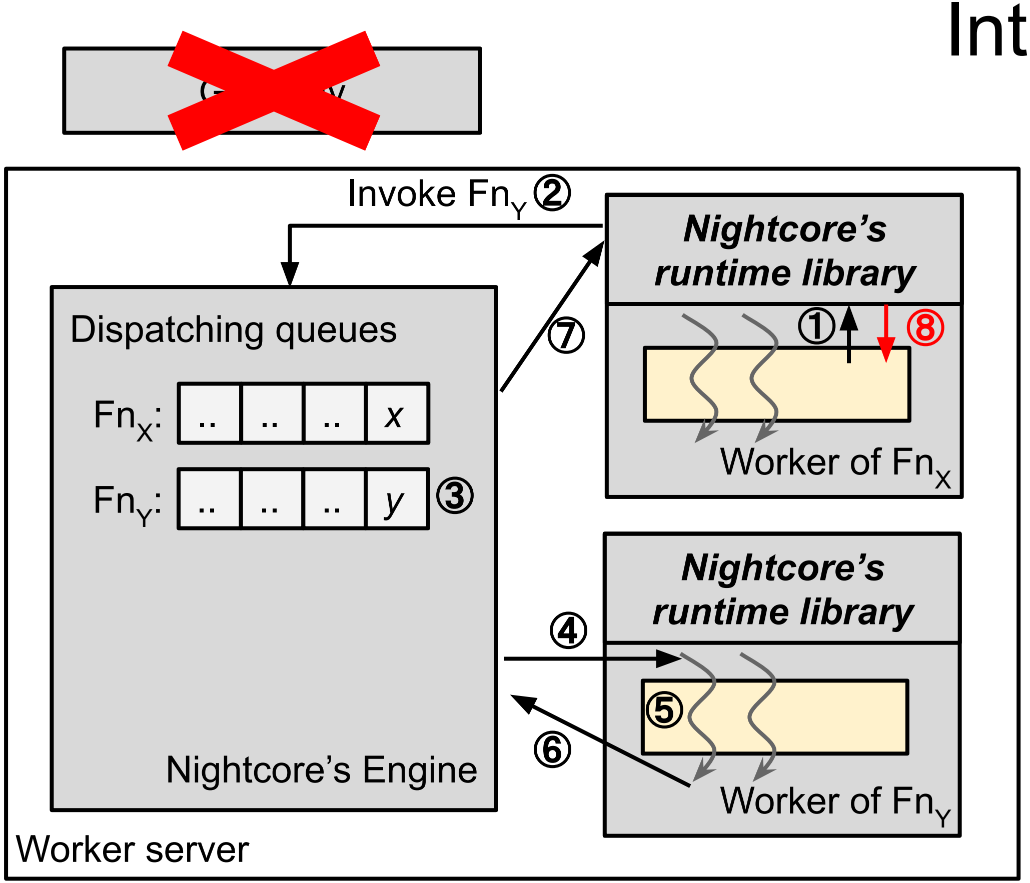
⑤ Worker thread executes code of Fn_y

⑥ Execution of req_y completed

⑦ Send output back to worker of Fn_x

⑧ Execution flow returns back to code of Fn_x

Internal Function Request



① Fn_y invoked via Nightcore's runtime API

② Req_y sent to Nightcore's engine

③ Place req_y in the dispatching queue

④ Dispatch Req_y to worker of Fn_y

⑤ Worker thread executes code of Fn_y

⑥ Execution of req_y completed

⑦ Send output back to worker of Fn_x

⑧ Execution flow returns back to code of Fn_x

Nightcore’s Techniques

- Optimizing locality of internal function calls
- High optimizations for local I/Os
    - Low-latency message channels
    - Event-driven concurrency
- Managing per-microservice concurrency to mitigate load variation

Nightcore’s Low-Latency Message Channel

We need IPC primitive for function worker I/Os

- One straightforward option —a feature-rich RPC framework like gRPC
- But wait, RPC protocols have μs-scale overheads (*killer microseconds*!)

Nightcore builds its own message channels for best performance

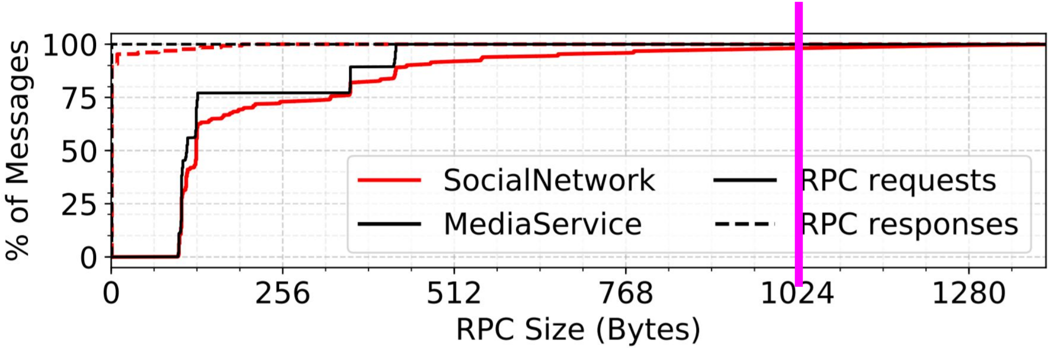
- Built on top of OS pipes
- Transmit fixed-size 1KB messages

Deliver messages in **3.4**μs

- In contrast, gRPC over Unix sockets takes 13μs for sending 1KB RPC payloads

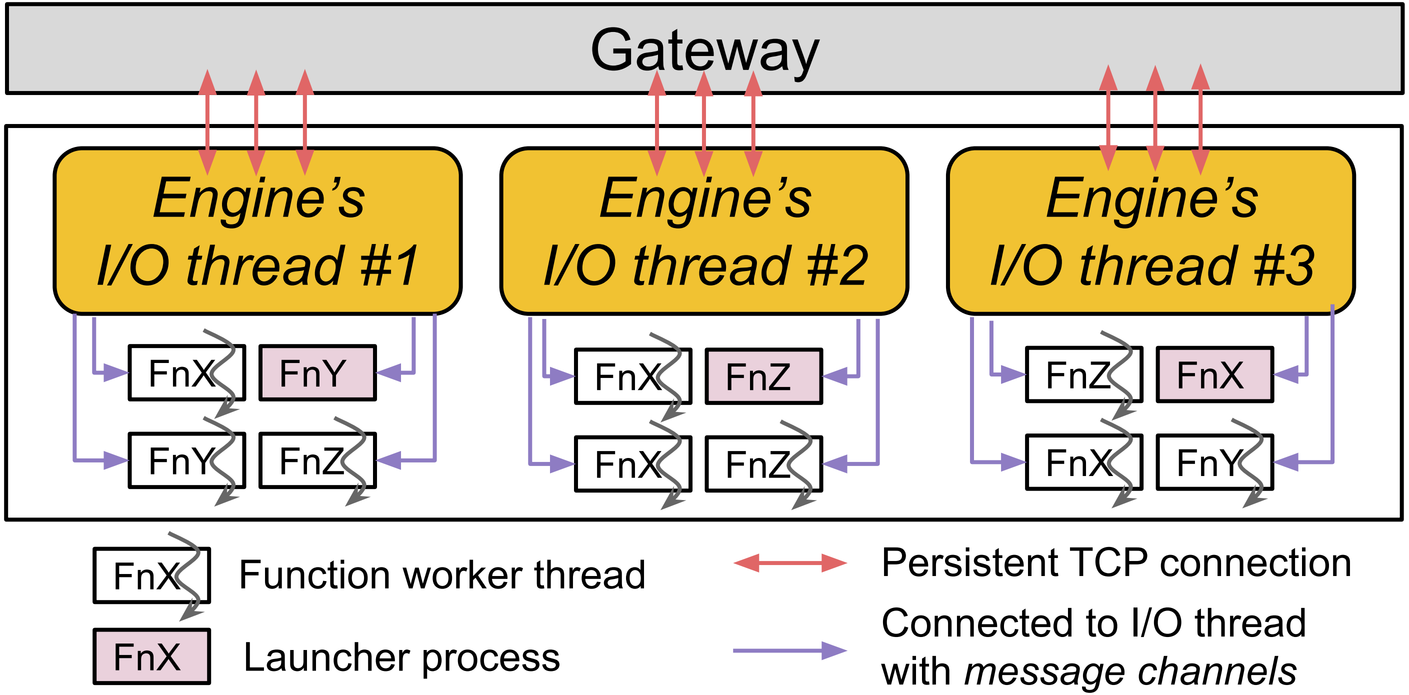
Nightcore’s Low-Latency Message Channel

Why choosing 1KB as the message size?



Distribution of RPC sizes across microservices in DeathStarBench

Event-Driven Concurrency for Best Efficiency



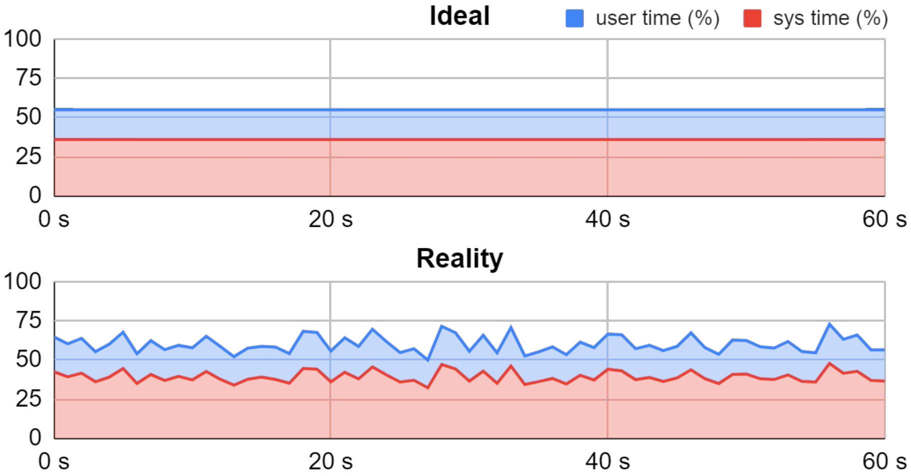
Small number of I/O threads

4 threads are sufficient for an invocation rate of 100K/s

Nightcore’s Techniques

- Optimizing locality of internal function calls
- High optimizations for local I/Os
  - Low-latency message channels
  - Event-driven concurrency
- Managing per-microservice concurrency to mitigate load variation

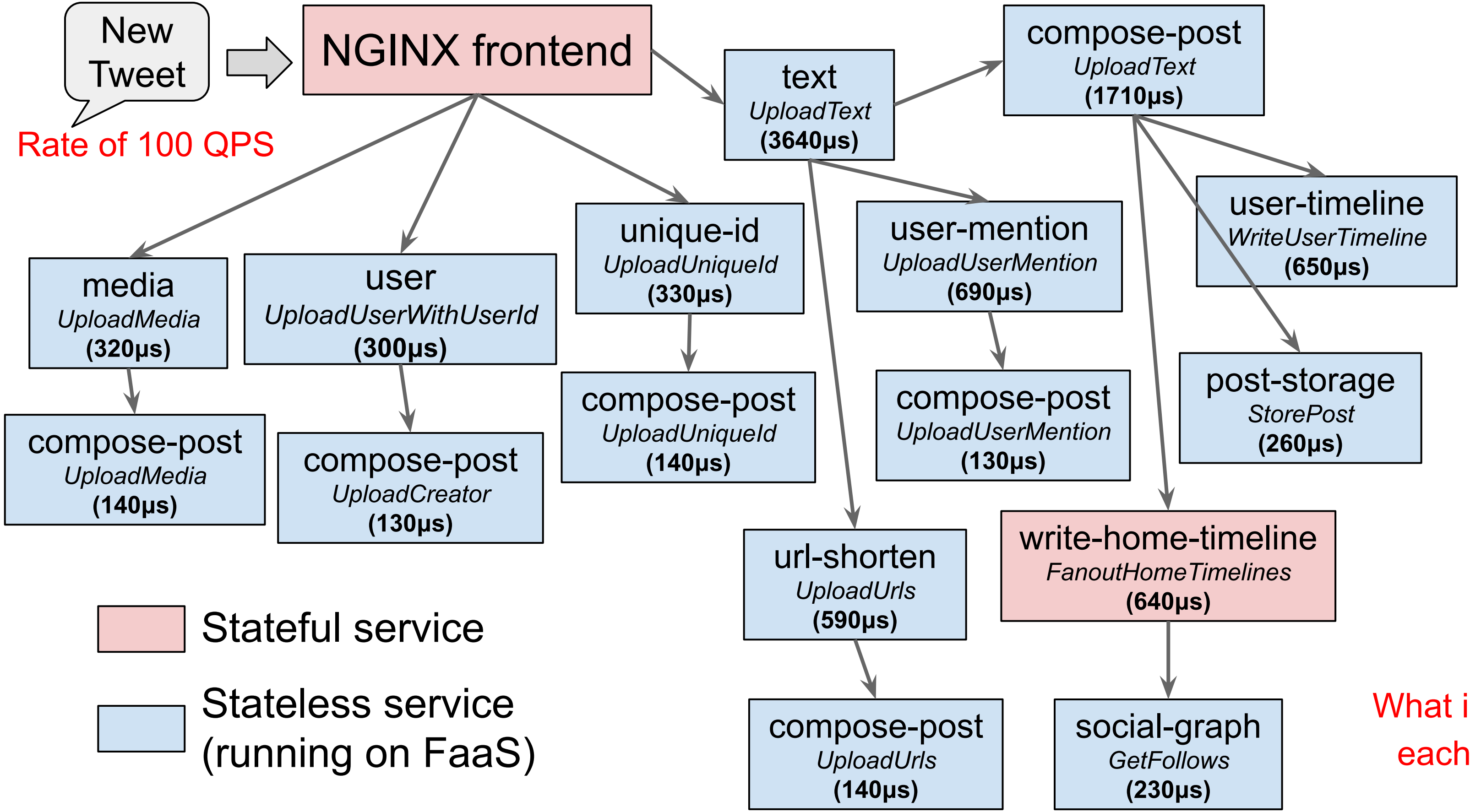
Internal Load Variations within Microservices



Timeline of CPU utilization
Running SocialNetwork microservices at
a ***fixed*** request rate

Why this happens?

Stage-based nature of microservices
→ Complex internal load dynamics



Rate of 100 QPS

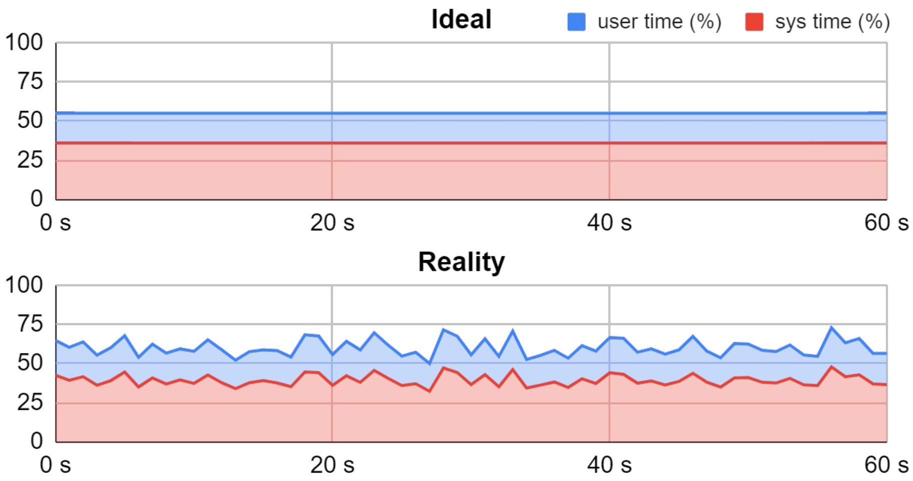
🟥 Stateful service

🟦 Stateless service (running on FaaS)

What is the load for each 🟦 ???

RPC trace from SocialNetwork microservices

Internal Load Variations within Microservices



Timeline of CPU utilization
Running SocialNetwork microservices at
a ***fixed*** request rate

Why this happens?

Stage-based nature of microservices
→ Complex internal load dynamics

Overusing concurrency for bursty load
→ Worse overall performance

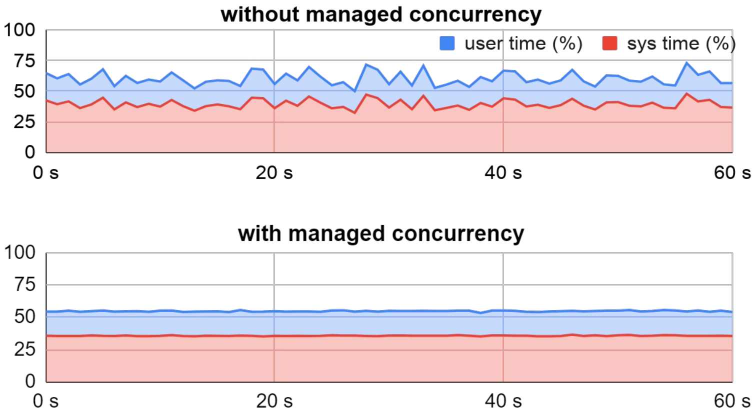
Nightcore’s Managed Concurrency

Per-function concurrency target

- Limiting concurrent execution
  → Prevent overuse of concurrency
- Dynamically computed with input load

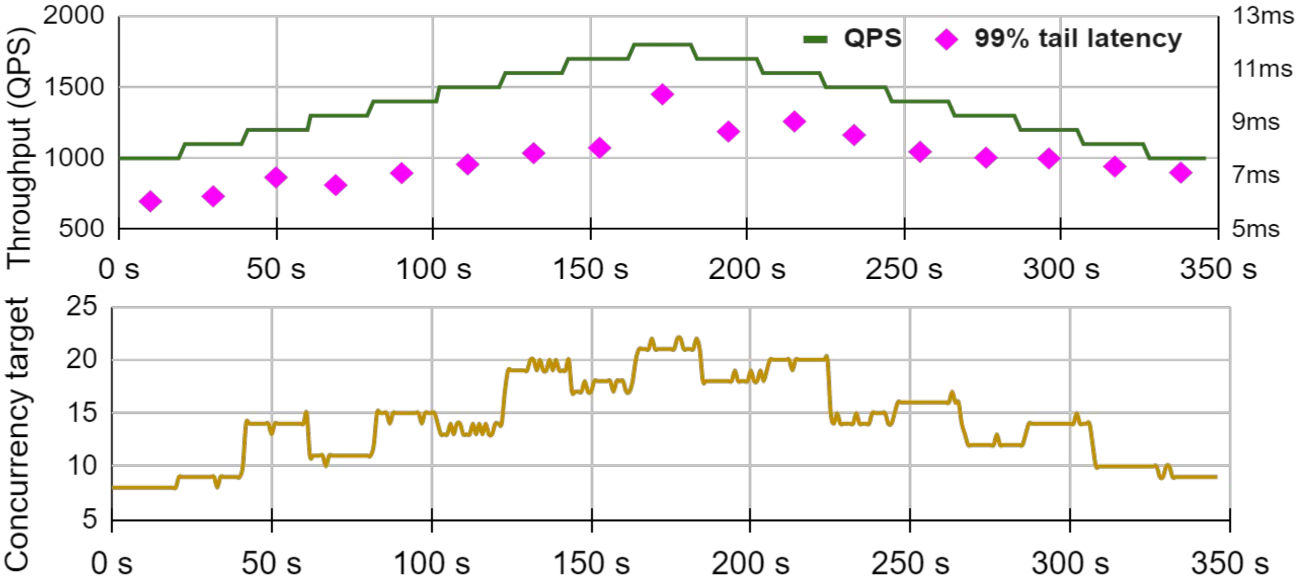
$$(\text{concurrency target}) = \underbrace{(\text{invocation rate}) \times (\text{function execution time})}_{\textit{Computed by exponential weight average}}$$

Timeline of CPU utilization
Running SocialNetwork microservices at
a ***fixed*** request rate



“Flatten the curve”

Nightcore's Managed Concurrency



Adaptive to load changes

Finally, Do We Achieve Our Performance Goals?

FaaS Systems	**Invocation Latency**		
	50th	***99th***	***99.9th***
AWS Lambda	10.4ms	25.8ms	59.9ms
OpenFaaS	1.09ms	3.66ms	5.54ms
Nightcore (external function calls)	285µs	536µs	855µs
Nightcore (internal function calls)	39µs	107µs	154µs

Evaluation

A nightcore edit is a cover track that speeds up the pitch and time of its source material by 10–30%.

Benchmark Workloads

DeathStarBench [ASPLOS ‘19]

- SocialNetwork
- MovieReviewing
- HotelReservation

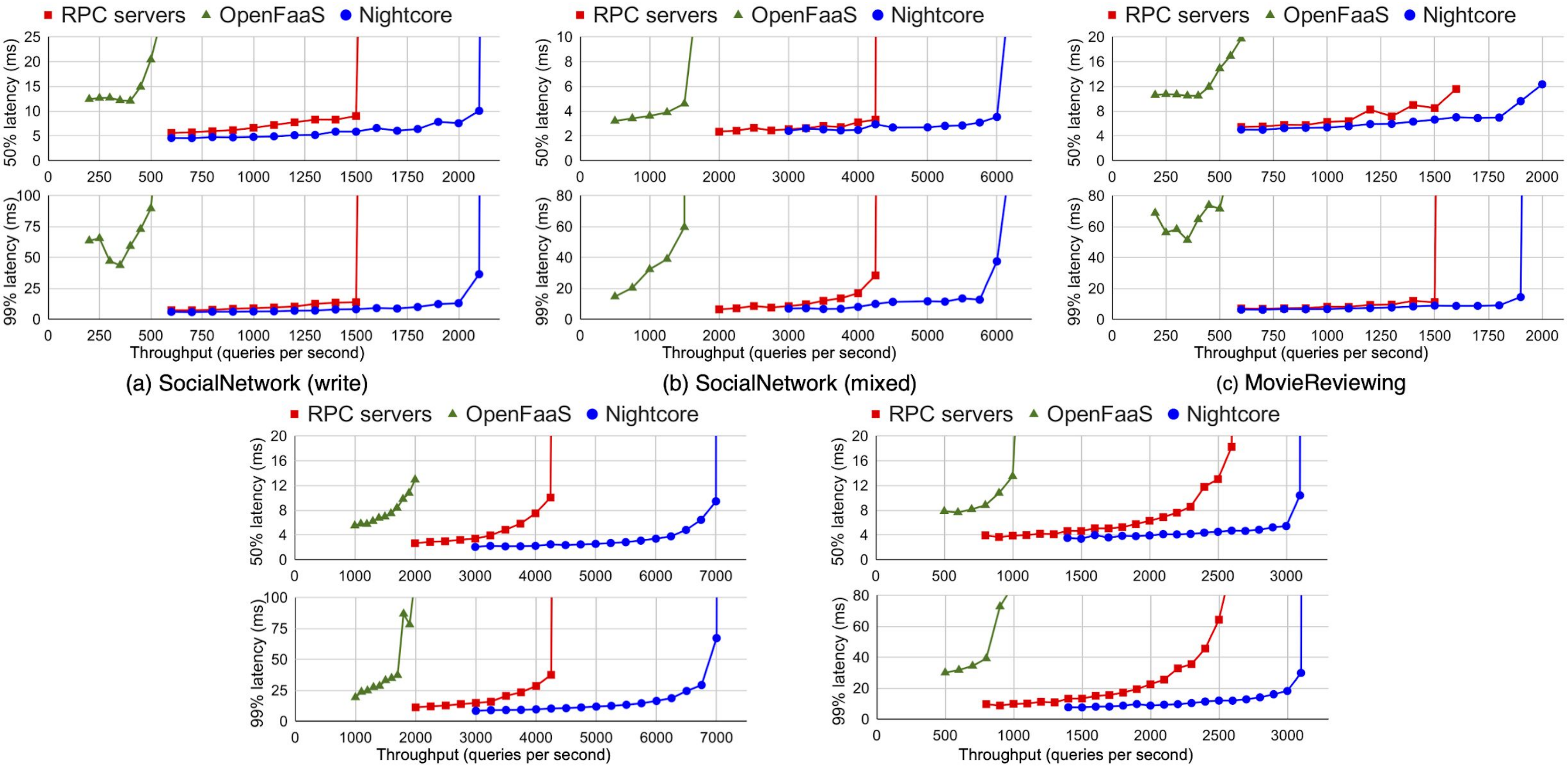
Google’s HipsterShop microservices

	Ported services	**RPC framework**	**Languages**
Social Network	11	Apache Thrift	C++
Movie Reviewing	12	Apache Thrift	C++
Hotel Reservation	11	gRPC	Go
HipsterShop	13	gRPC	Go, Node.js, Python

Systems for Comparison

RPC servers — non-serverless deployment of microservices

OpenFaaS — FaaS system deployed in the same way as Nightcore



(a) SocialNetwork (write)

(b) SocialNetwork (mixed)

(c) MovieReviewing

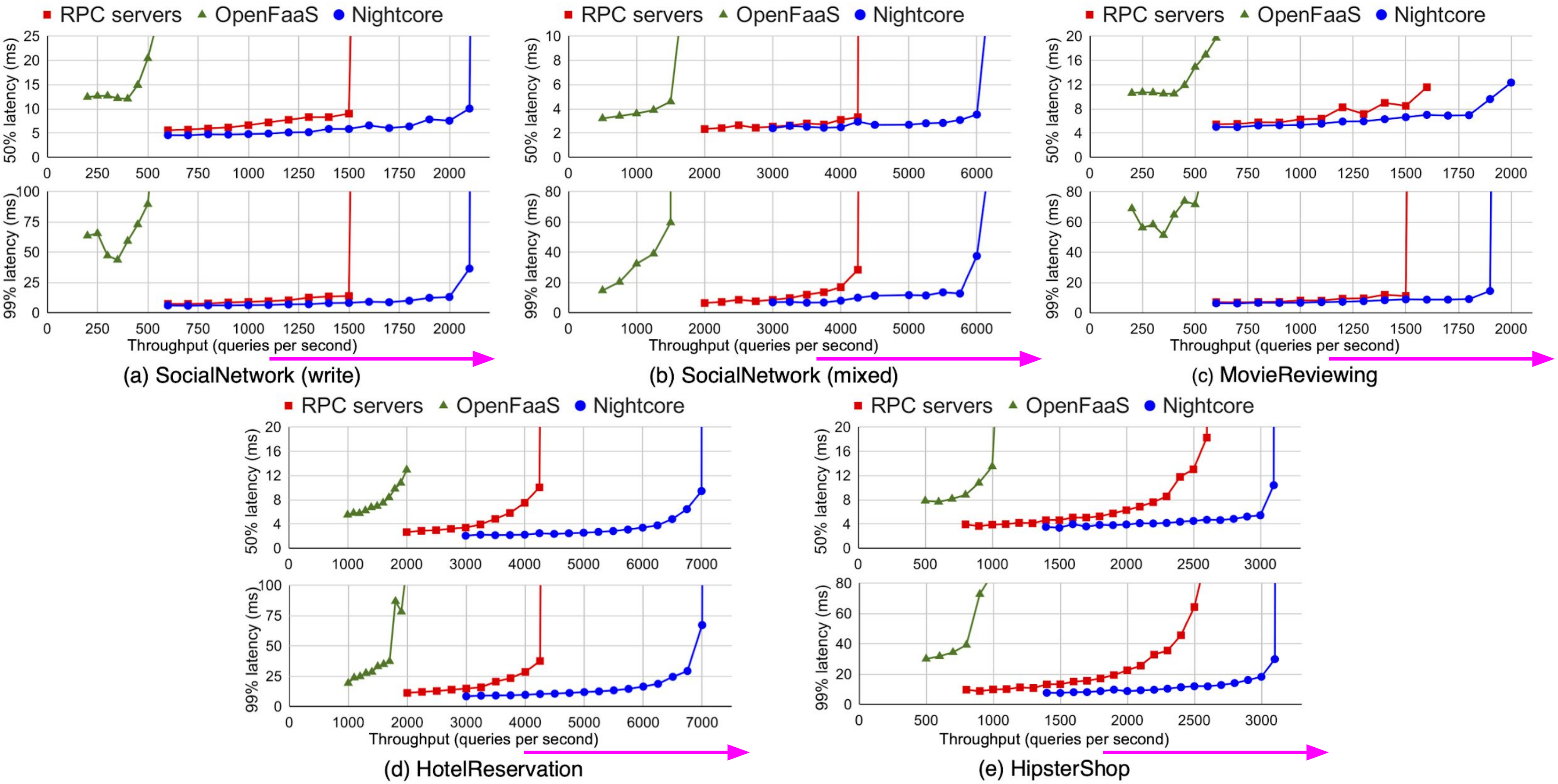
(d) HotelReservation

(e) HipsterShop

Single-Server Experiment

OpenFaaS and Nightcore: one worker VM runs all functions

RPC servers: one VM runs all RPC servers



(a) SocialNetwork (write)

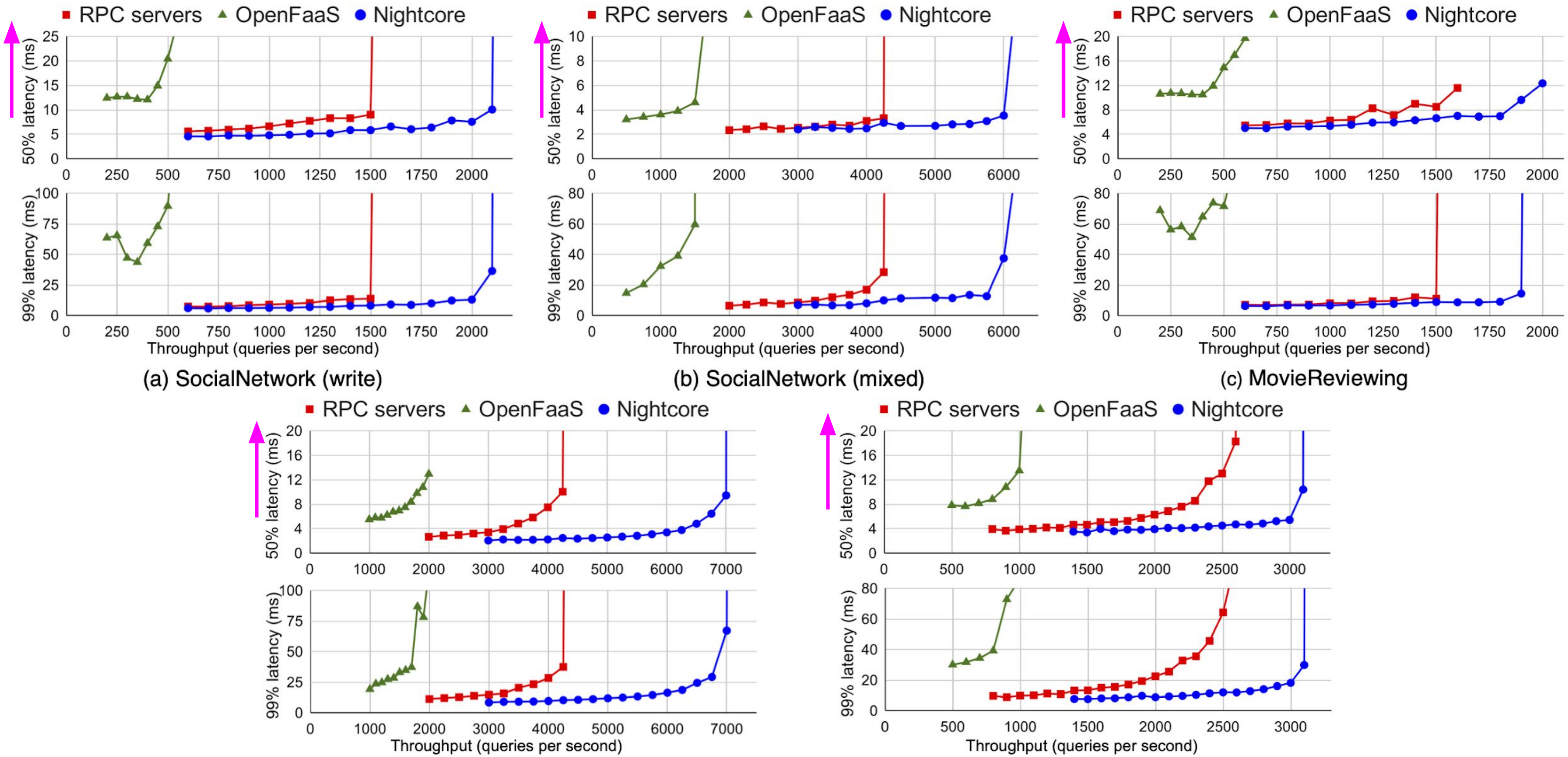
(b) SocialNetwork (mixed)

(c) MovieReviewing

(d) HotelReservation

(e) HipsterShop

X-axis: throughput (QPS)



(a) SocialNetwork (write)

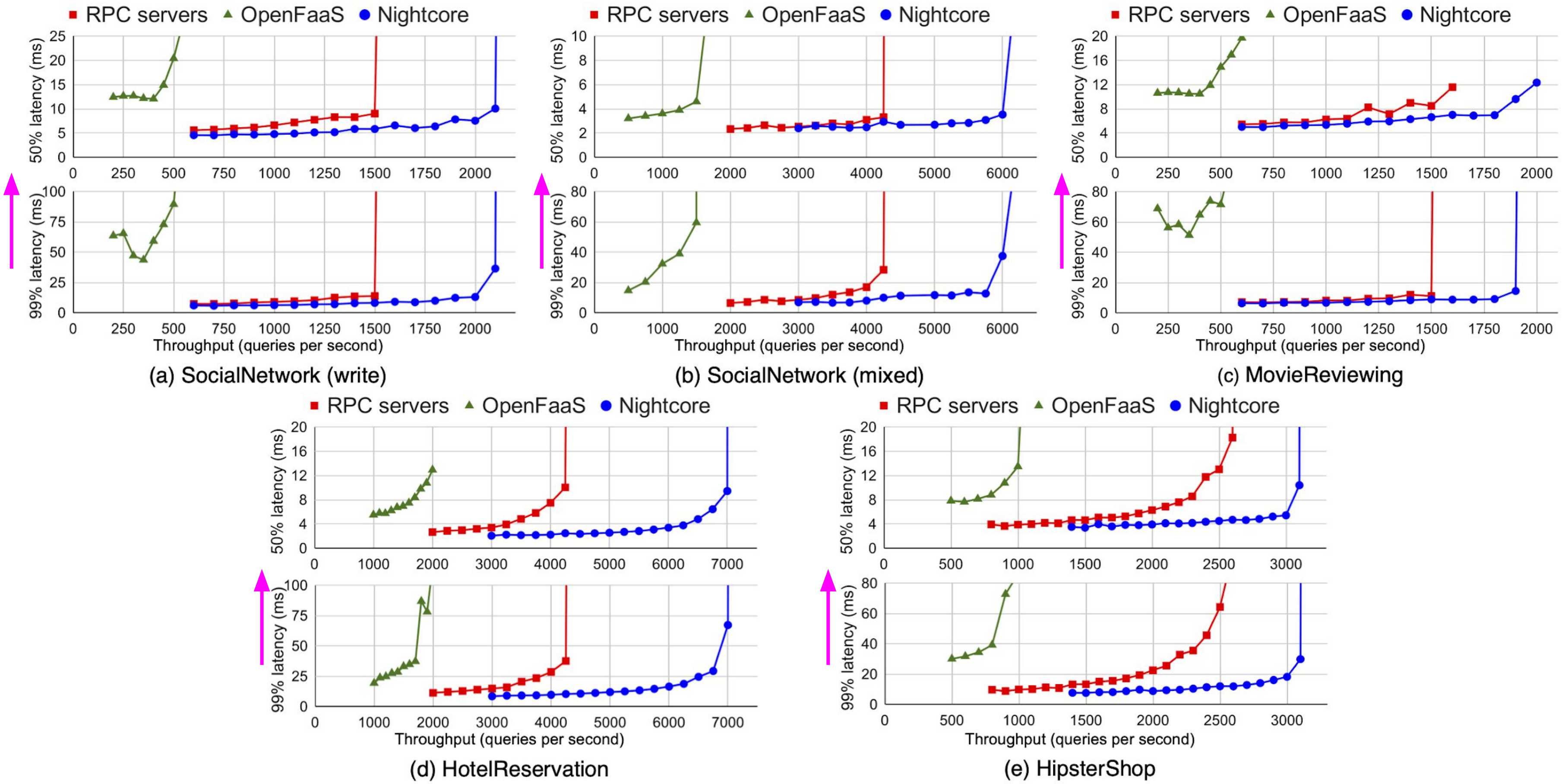
(b) SocialNetwork (mixed)

(c) MovieReviewing

(d) HotelReservation

(e) HipsterShop

Upper chart: median latency



(a) SocialNetwork (write)

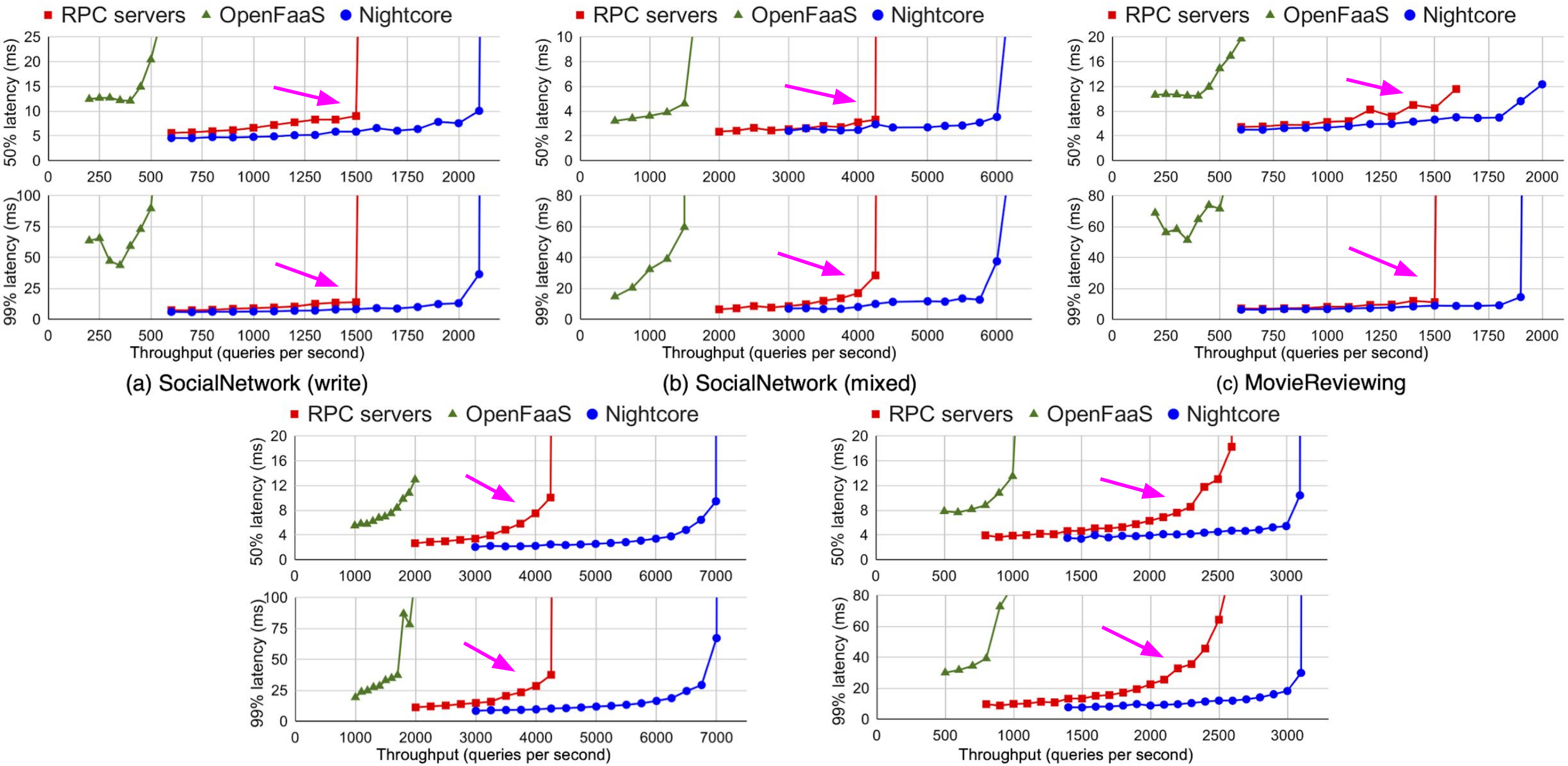
(b) SocialNetwork (mixed)

(c) MovieReviewing

(d) HotelReservation

(e) HipsterShop

Lower chart: tail latency



(a) SocialNetwork (write)

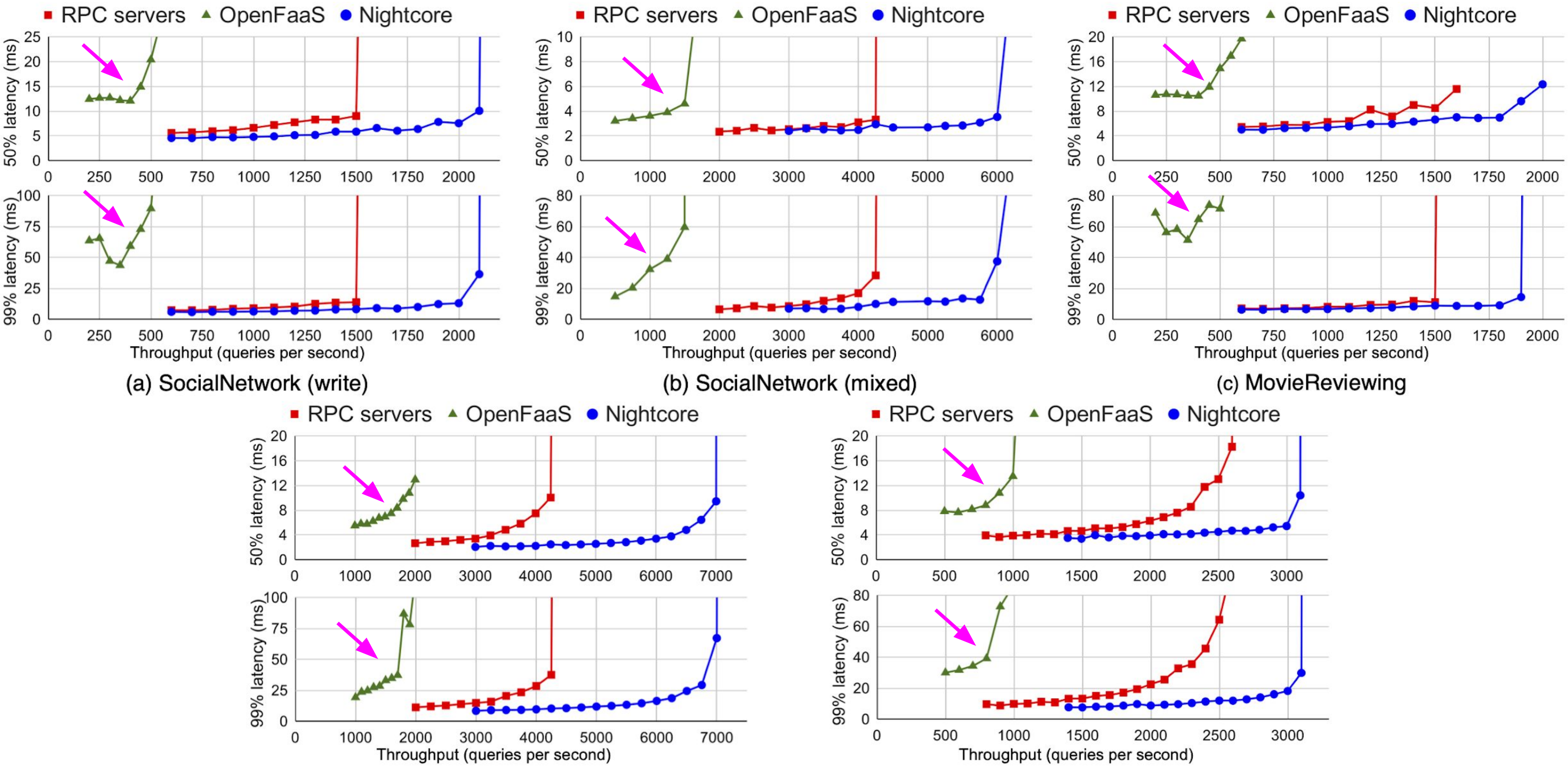
(b) SocialNetwork (mixed)

(c) MovieReviewing

(d) HotelReservation

(e) HipsterShop

RPC servers —**the ordinary choice for microservices**



(a) SocialNetwork (write)

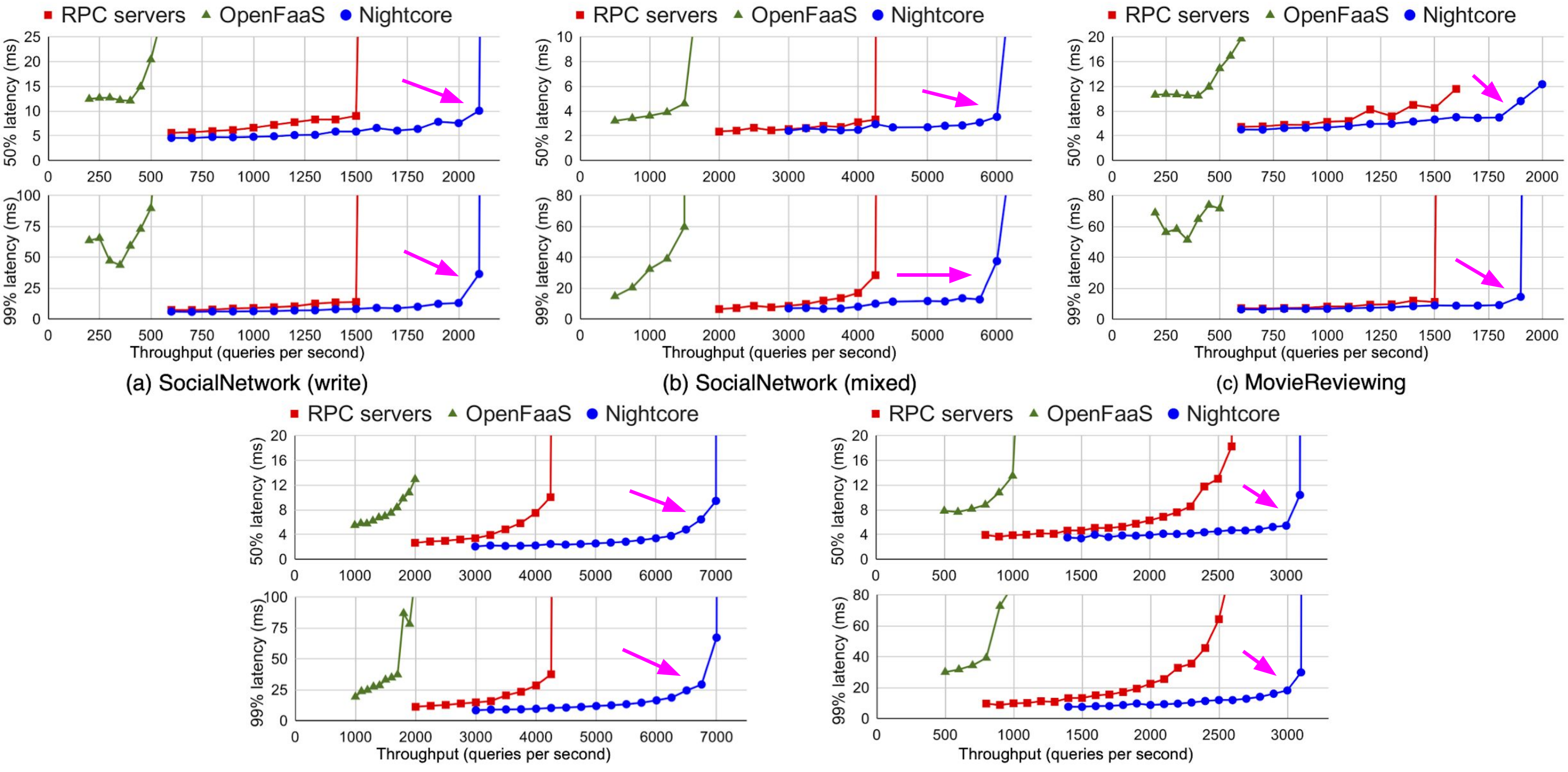
(b) SocialNetwork (mixed)

(c) MovieReviewing

(d) HotelReservation

(e) HipsterShop

OpenFaaS **—microservices on FaaS, but a worse choice**



(a) SocialNetwork (write)

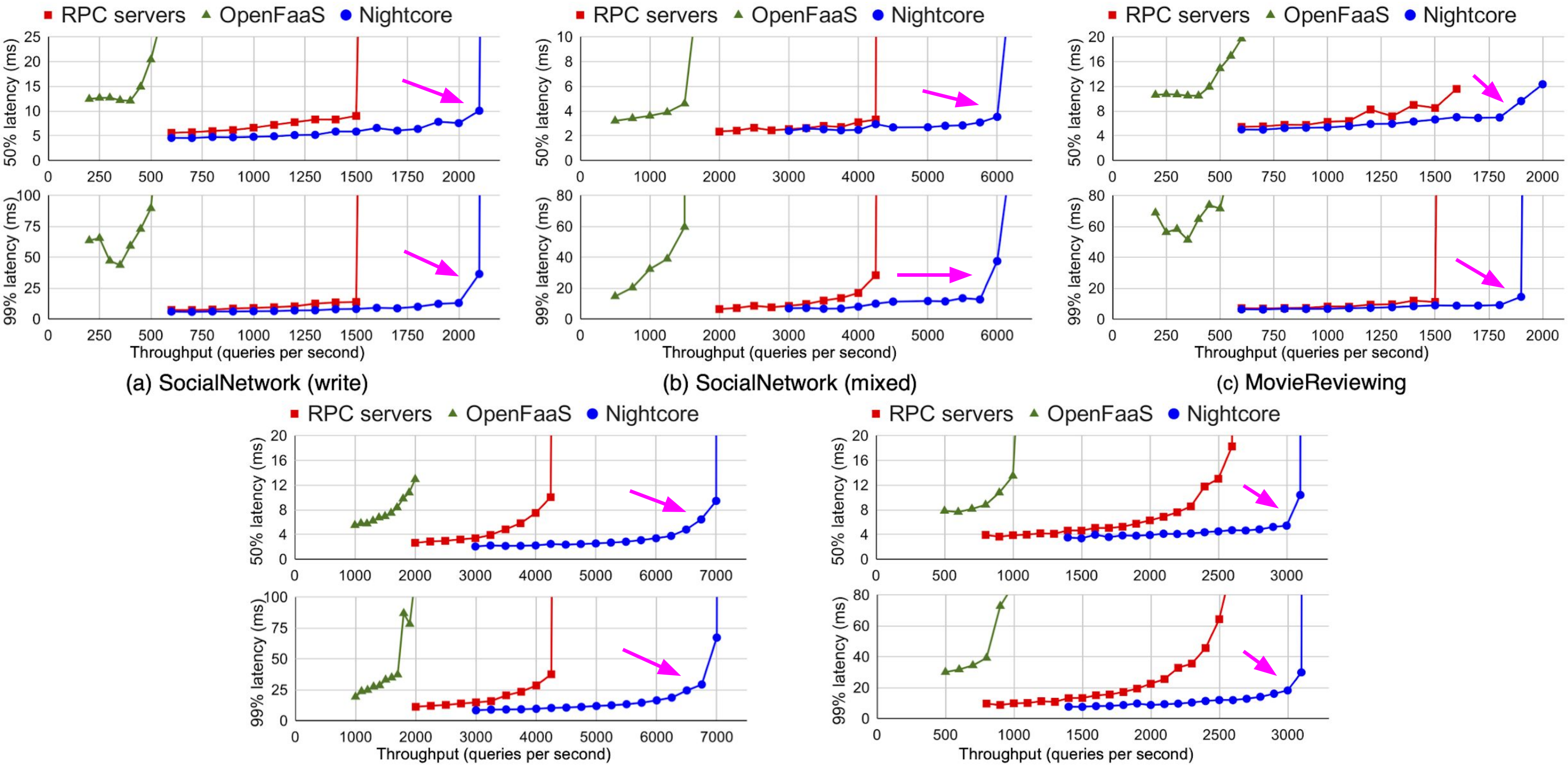
(b) SocialNetwork (mixed)

(c) MovieReviewing

(d) HotelReservation

(e) HipsterShop

Nightcore —**let FaaS shine for microservices**



(a) SocialNetwork (write)

(b) SocialNetwork (mixed)

(c) MovieReviewing

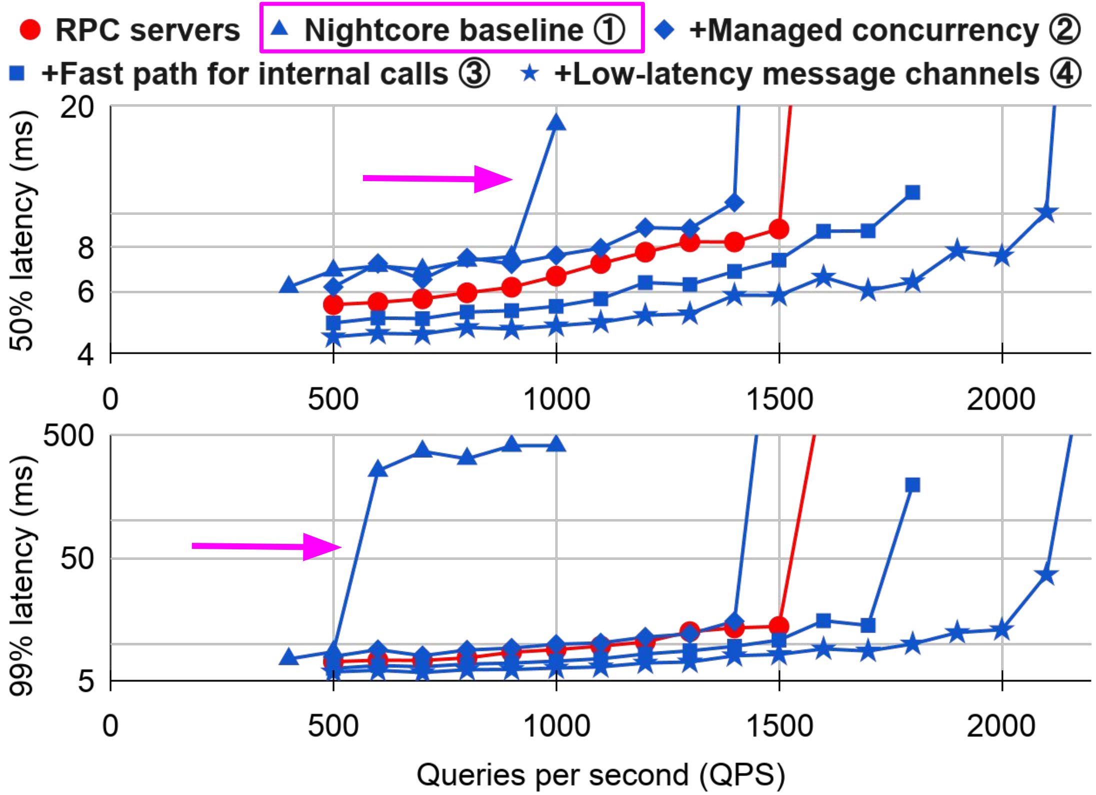
(d) HotelReservation

(e) HipsterShop

Nightcore v.s. RPC servers

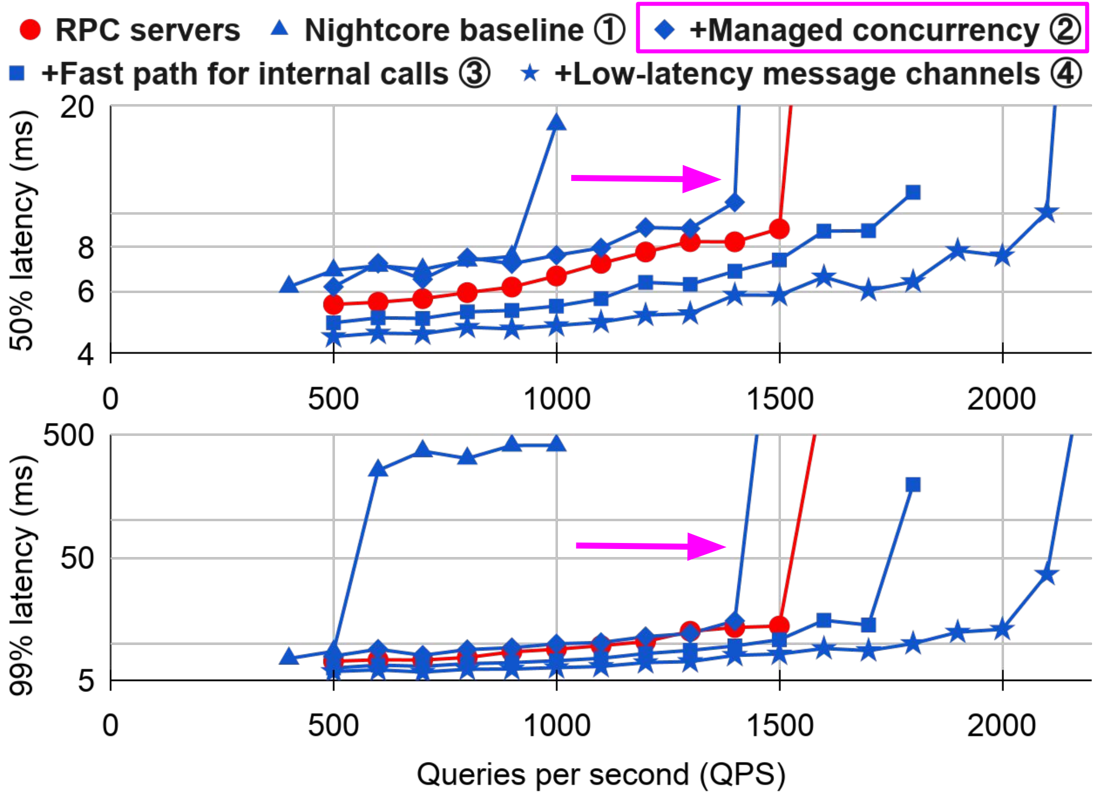
1.27x to **1.59x** higher throughput
up to **34%** reduction in tail latency

Performance Evaluation of Nightcore Designs



1/3 throughput of RPC servers

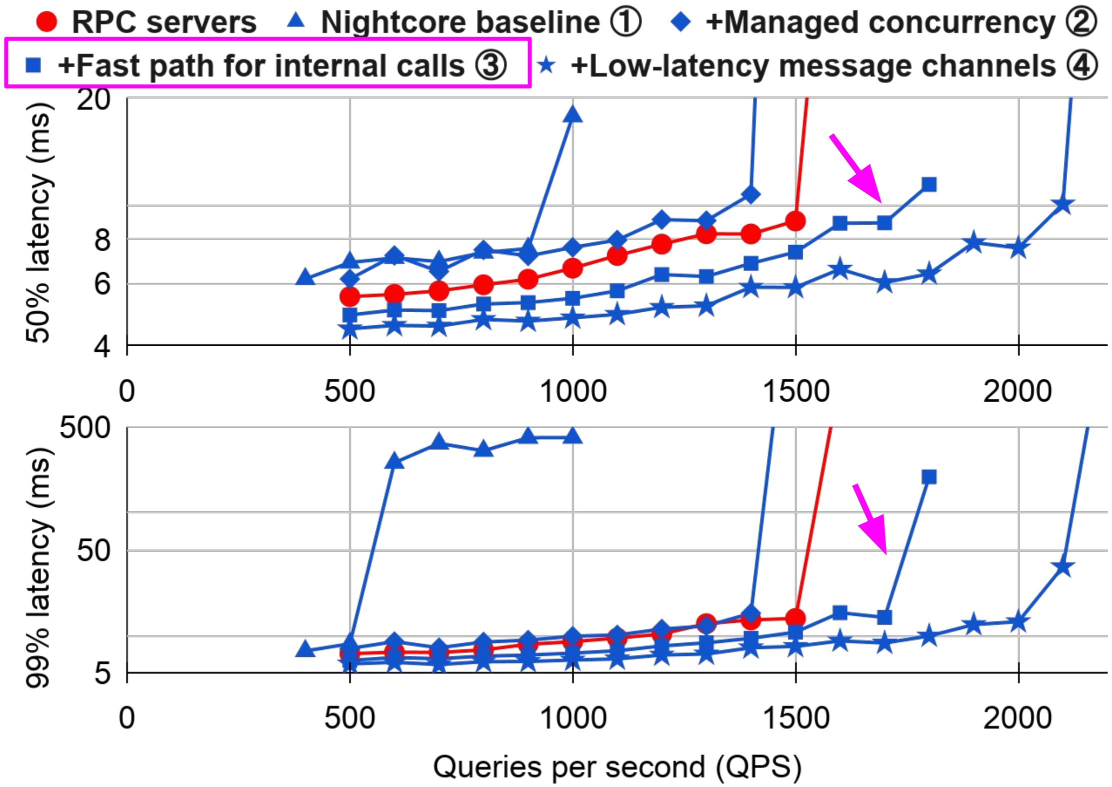
Performance Evaluation of Nightcore Designs



throughput closed to RPC servers

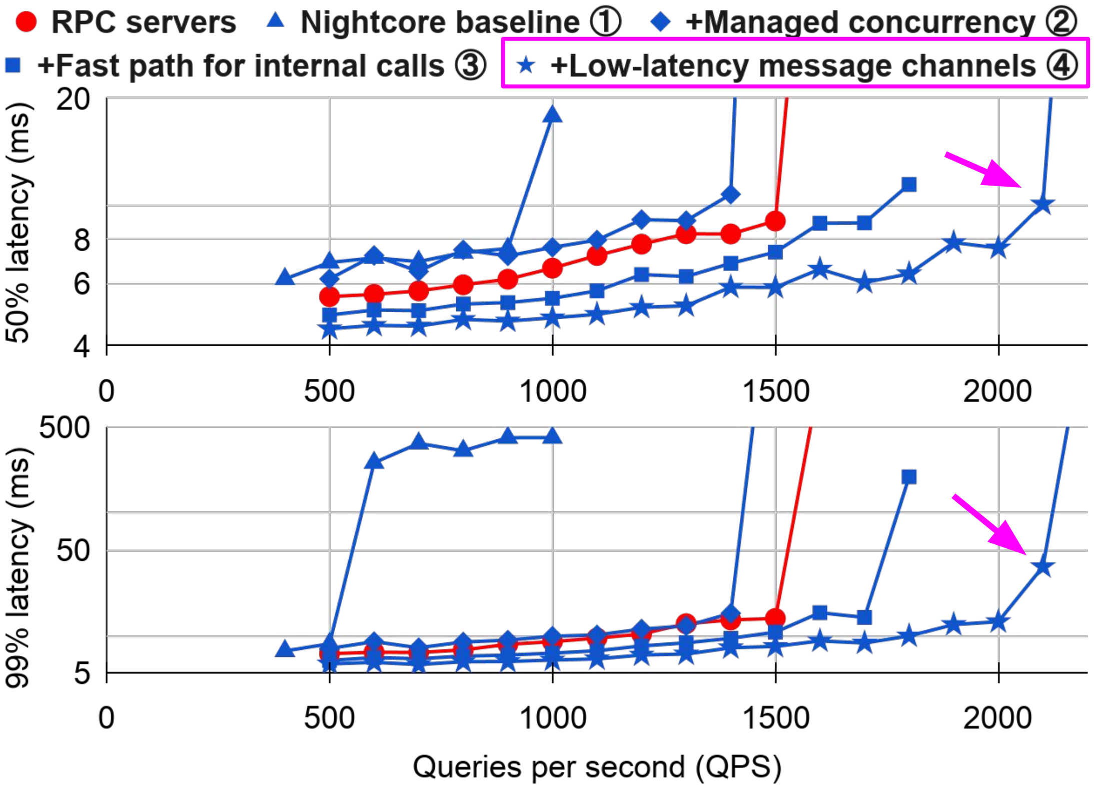
much better tail latency

Performance Evaluation of Nightcore Designs



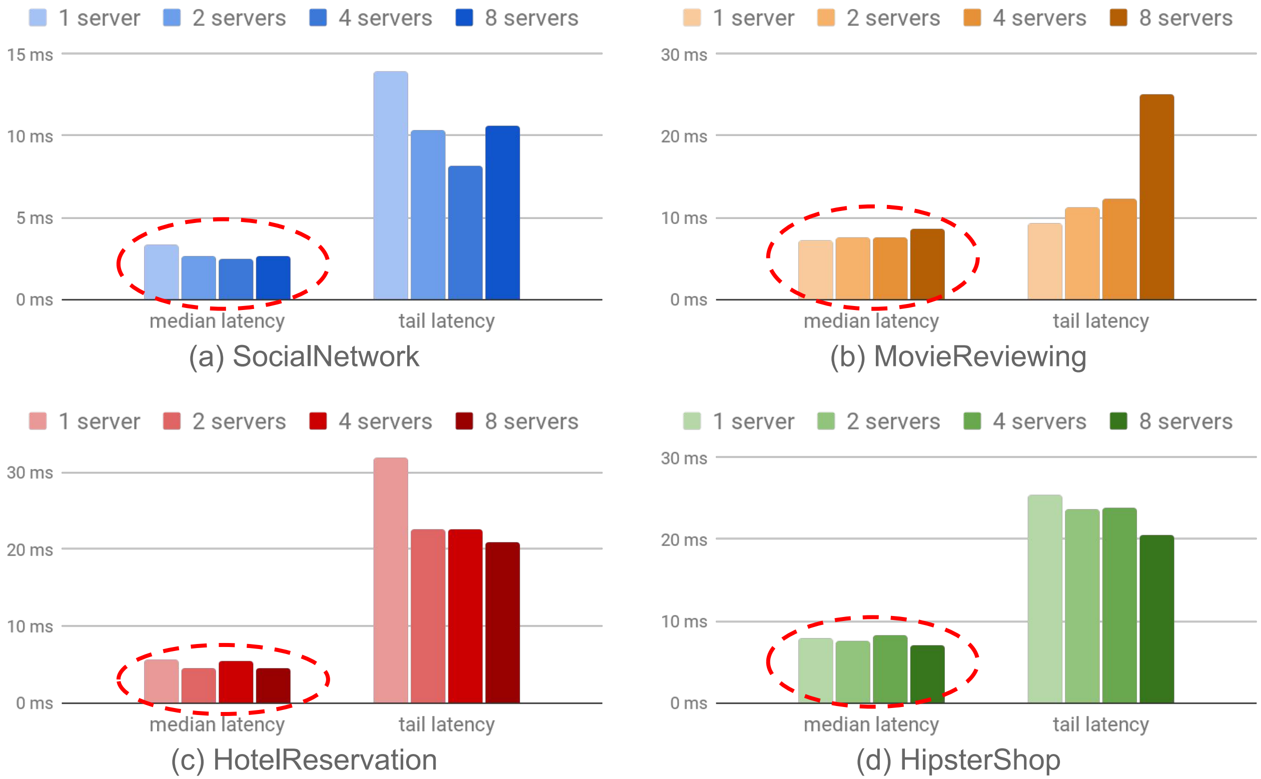
slightly better than RPC servers

Performance Evaluation of Nightcore Designs



1.33x higher throughput than RPC servers

Weak Scaling of Nightcore



(a) SocialNetwork

(b) MovieReviewing

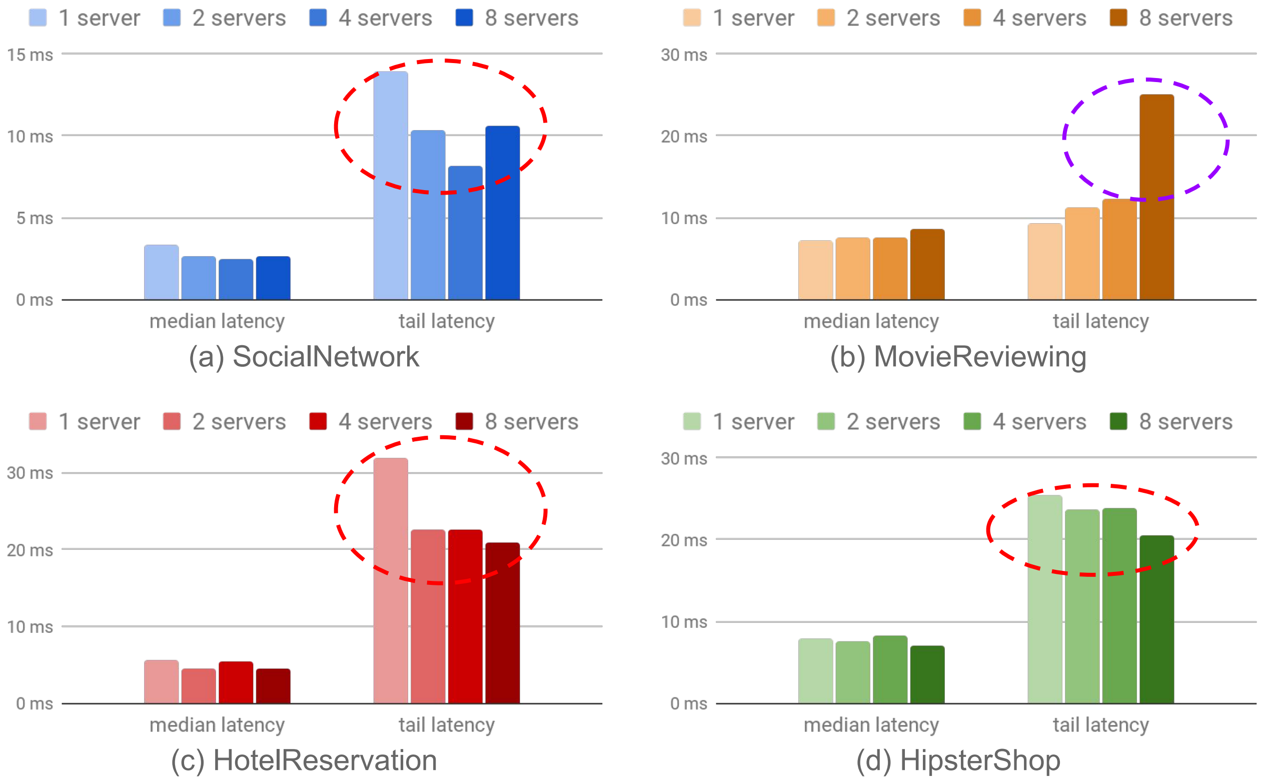
(c) HotelReservation

(d) HipsterShop

Note: *N* servers run *N* times of the request load of 1 server

Similar median latency with more servers

Weak Scaling of Nightcore



(a) SocialNetwork

(b) MovieReviewing

(c) HotelReservation

(d) HipsterShop

Note: *N* servers run *N* times of the request load of 1 server

Similar (or better) tail latency with more servers

Except MovieReviewing with 8 servers
But we see a similar spike in tail latencies when using 8 RPC servers

Comparison (8 Servers)

RPC servers as the baseline (1.0x)

	Throughput (higher is better)			**Tail Latency** (lower is better)	
	OpenFaaS	*Nightcore*		*OpenFaaS*	*Nightcore*
SocialNetwork	0.29x	1.33x		3.40x	0.34x
MovieReviewing	0.30x	1.36x		4.44x	0.98x
HotelReservation	0.28x	2.93x		0.96x	1.06x
HipsterShop	0.38x	1.87x		1.80x	0.31x

Comparison (8 Servers)

RPC servers as the baseline (1.0x)

	Throughput (higher is better)		**Tail Latency** (lower is better)	
	OpenFaaS	*Nightcore*	***OpenFaaS***	*Nightcore*
SocialNetwork	**0.29x**	1.33x	**3.40x**	0.34x
MovieReviewing	**0.30x**	1.36x	**4.44x**	0.98x
HotelReservation	**0.28x**	2.93x	**0.96x**	1.06x
HipsterShop	**0.38x**	1.87x	**1.80x**	0.31x

OpenFaaS v.s. RPC servers

28% to **38%** of throughput

increase tail latency by up to **4.4x**

☹

Comparison (8 Servers)

RPC servers as the baseline (1.0x)

	Throughput (higher is better)			**Tail Latency** (lower is better)	
	OpenFaaS	***Nightcore***		*OpenFaaS*	***Nightcore***
SocialNetwork	0.29x	**1.33x**		3.40x	**0.34x**
MovieReviewing	0.30x	**1.36x**		4.44x	**0.98x**
HotelReservation	0.28x	**2.93x**		0.96x	**1.06x**
HipsterShop	0.38x	**1.87x**		1.80x	**0.31x**

Nightcore v.s. RPC servers

1.4x to **2.9x** higher throughput

up to **69%** reduction in tail latency

😊

Conclusion

Nightcore is a FaaS runtime for μs-scale microservices

Nightcore includes diverse techniques to eliminate μs-scale overheads

Nightcore achieves **1.4x–2.9x** higher throughput than containerized RPC servers, and up to **69%** reduction in tail latency

Nightcore is open source at
github.com/ut-osa/nightcore

“Make it fast, rather than general or powerful”

(Butler W. Lampson, *Hints for Computer System Design*)