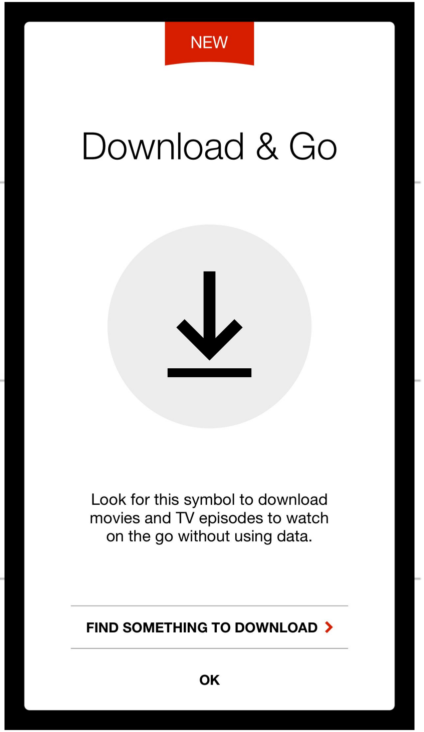
Netflix Play API

Why we built an Evolutionary Architecture

Suudhan Rangarajan (@suudhan)
Senior Software Engineer

NETFLIX


NEW

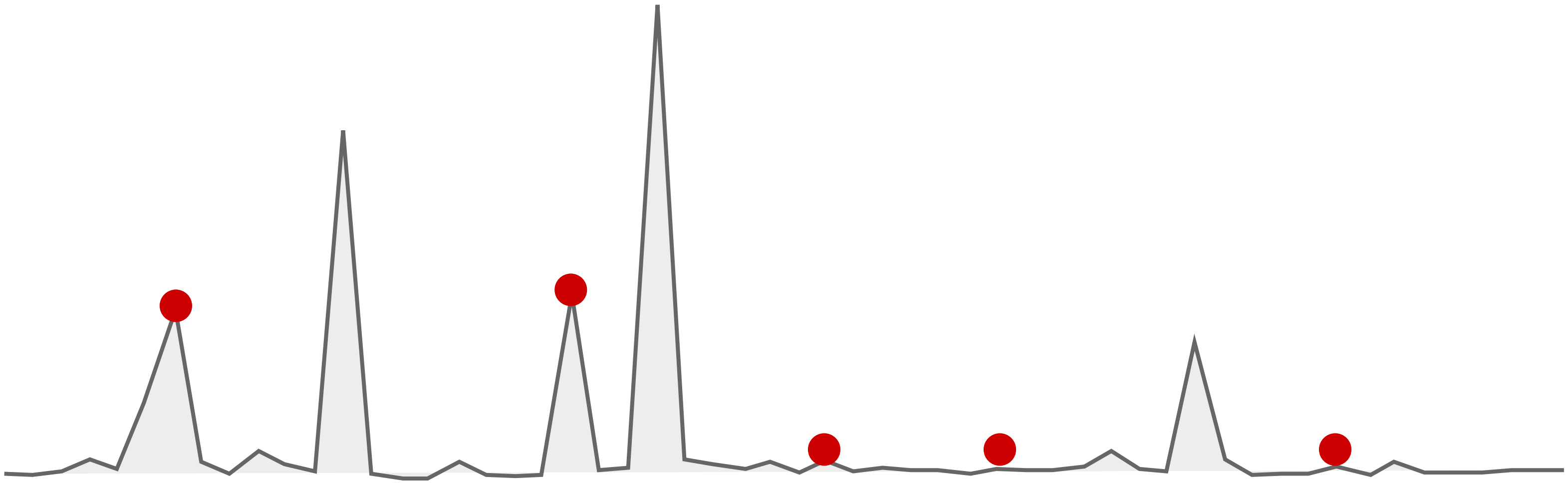
Download & Go

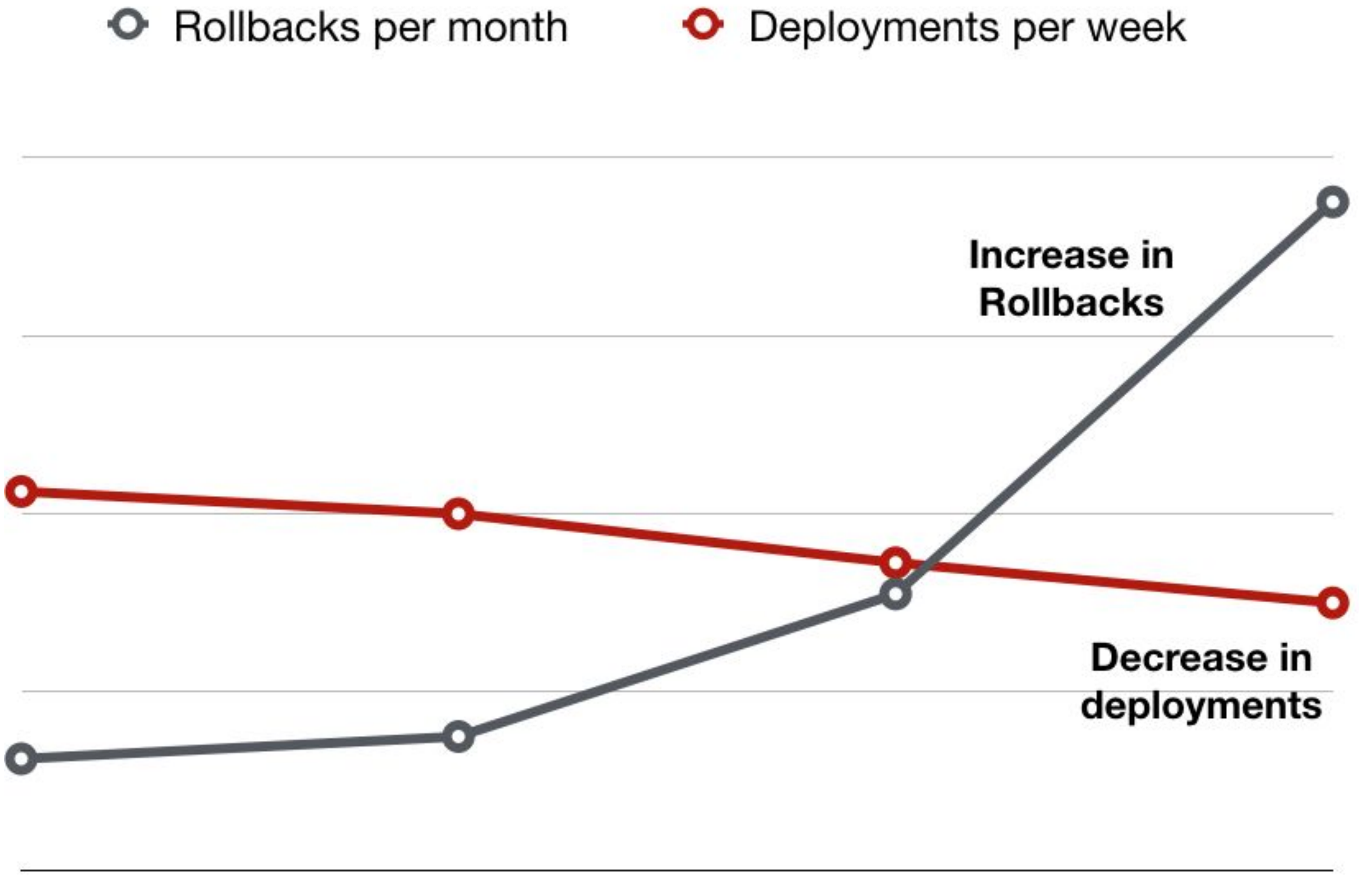
Look for this symbol to download movies and TV episodes to watch on the go without using data.

FIND SOMETHING TO DOWNLOAD >

OK

Q1 2016 Q2 2016 Q3 2016 Q4 2016




Rollbacks per month
Deployments per week
Increase in Rollbacks
Decrease in deployments

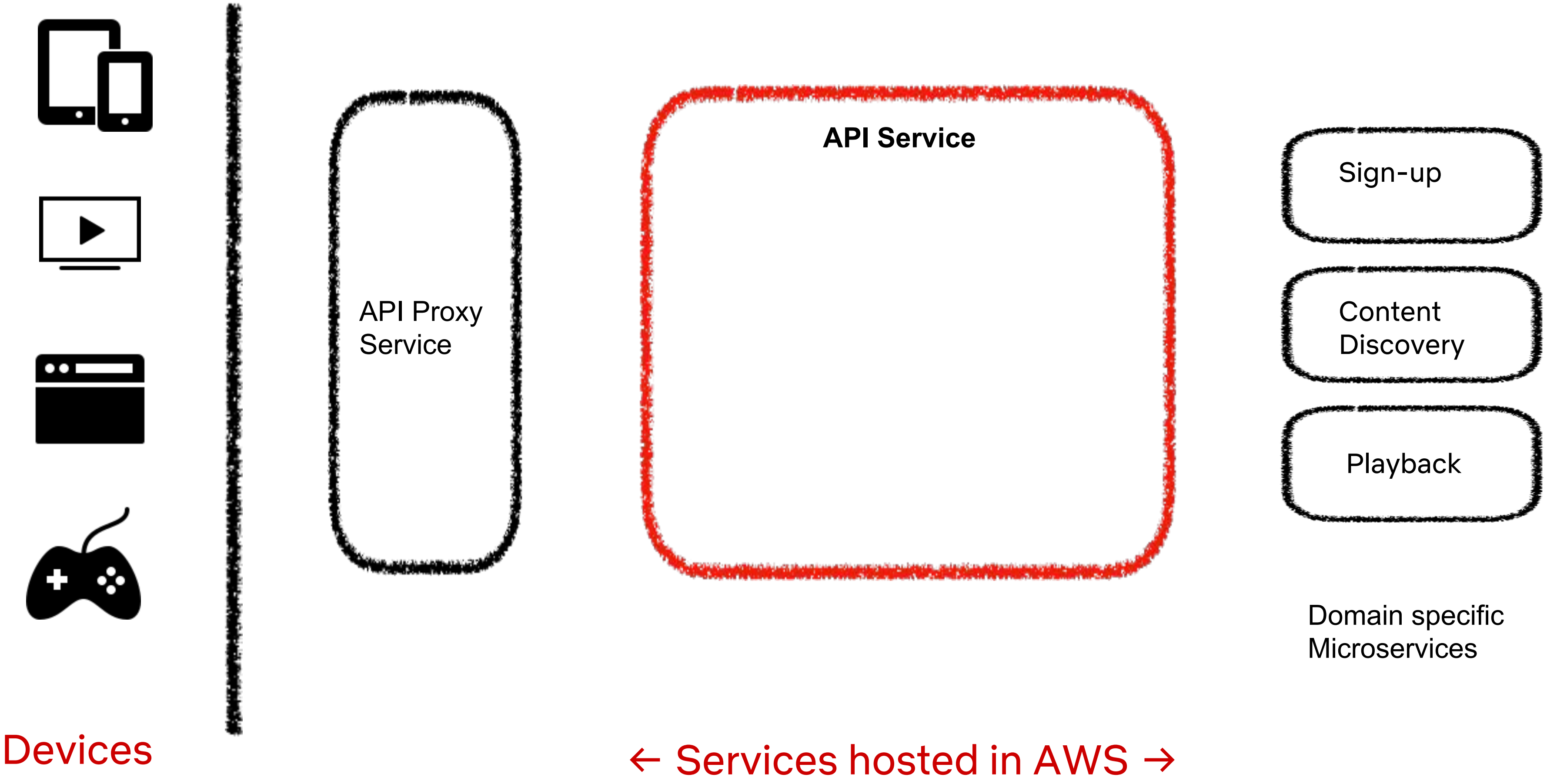
Netflix Play API

Why we built an Evolutionary Architecture

Suudhan Rangarajan (@suudhan)
Senior Software Engineer

NETFLIX

Previous Architecture Workflow



API Proxy Service

API Service

Sign-up

Content Discovery

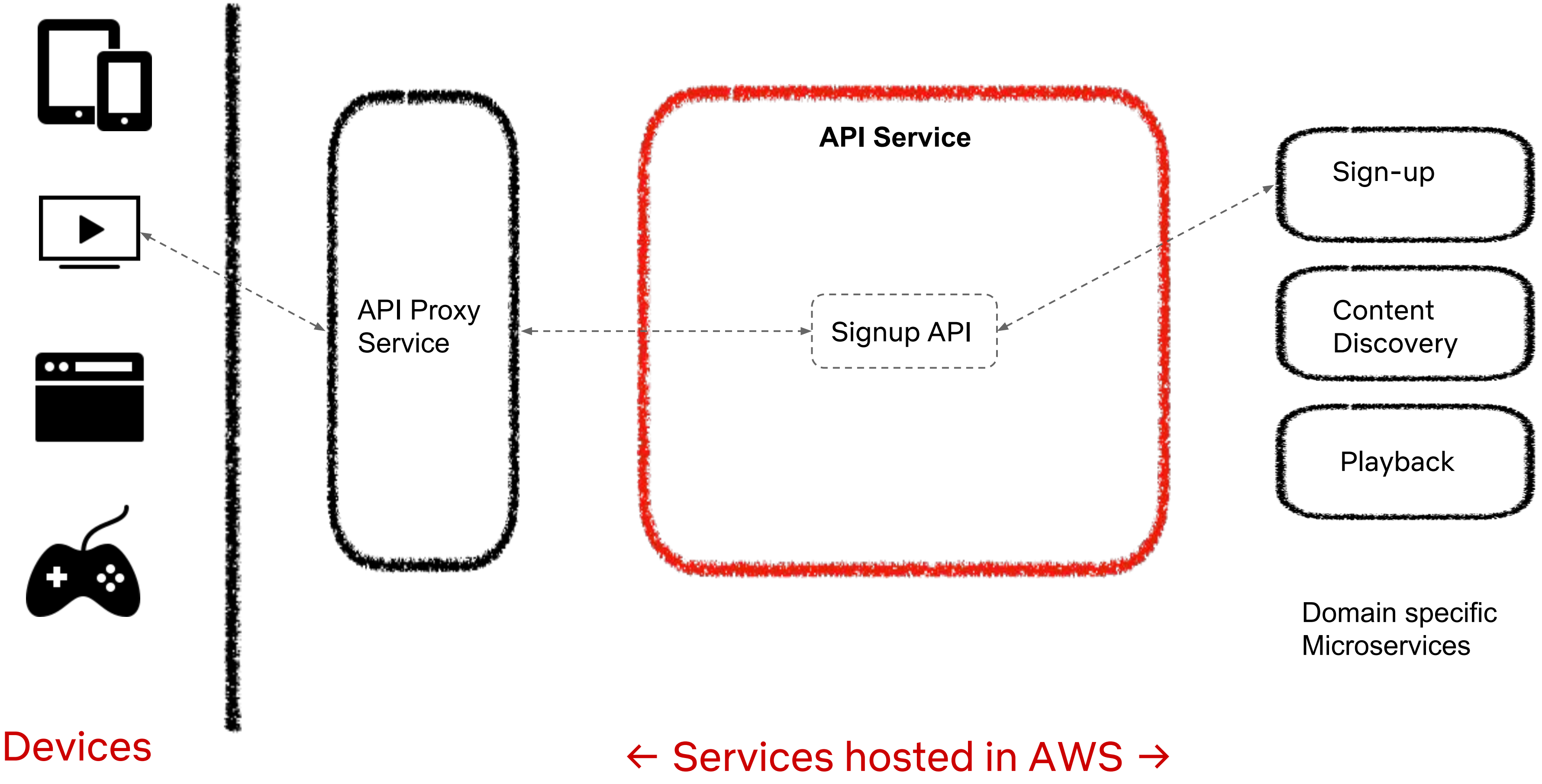
Playback

Domain specific Microservices

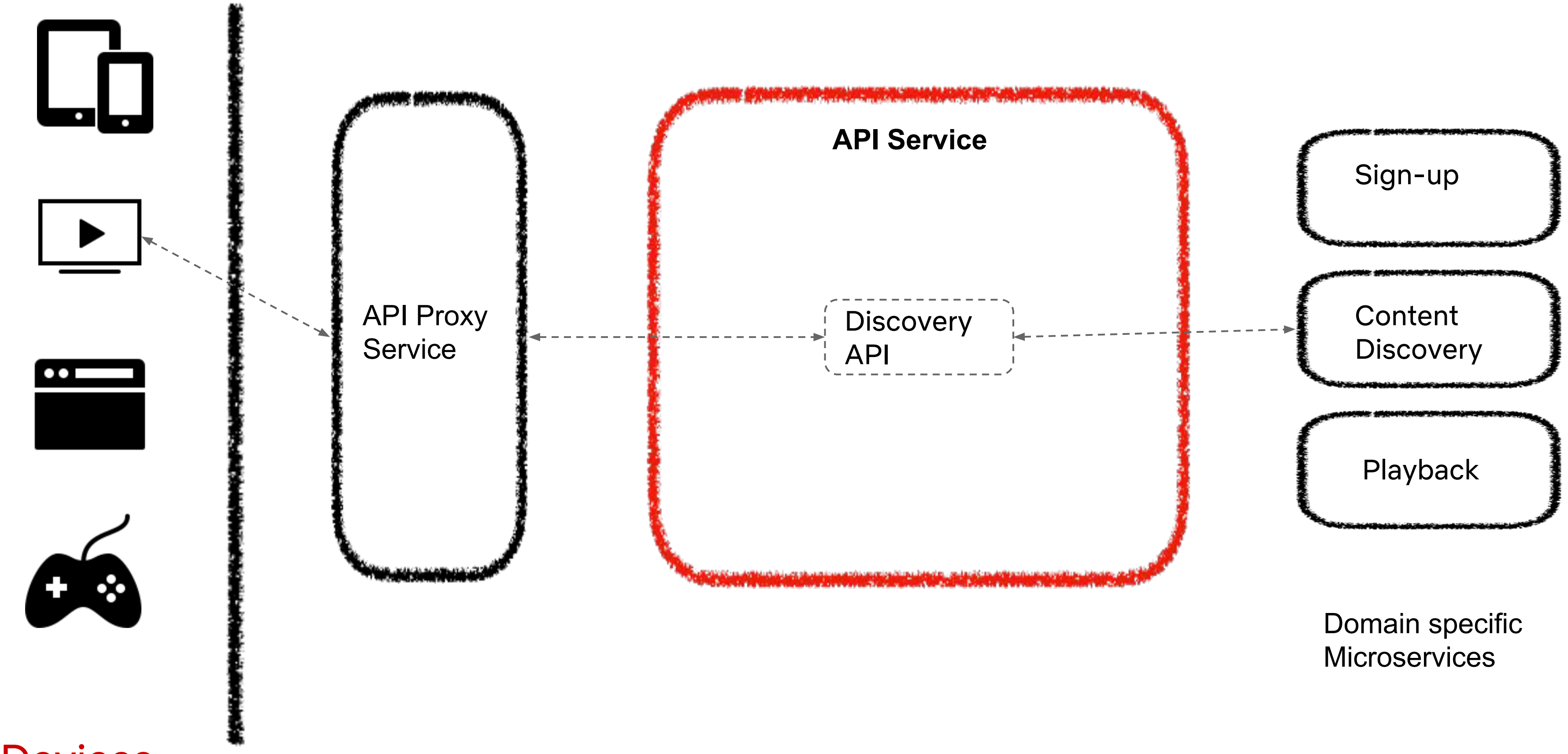
Devices

← Services hosted in AWS →

Signup Workflow



Content Discovery Workflow

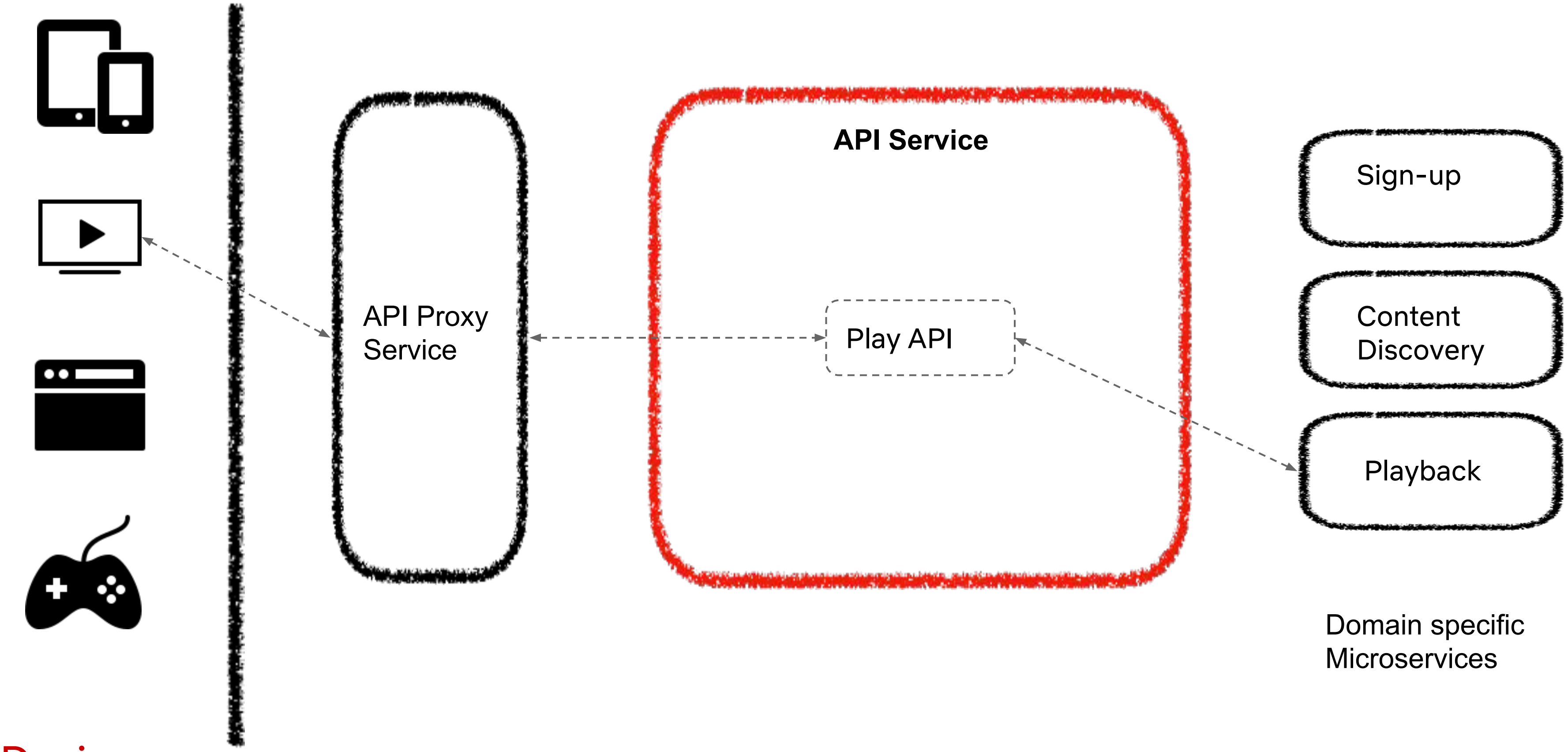


Domain specific Microservices

Devices

← Services hosted in AWS →

Playback Workflow



API Proxy Service

API Service

Play API

Sign-up

Content Discovery

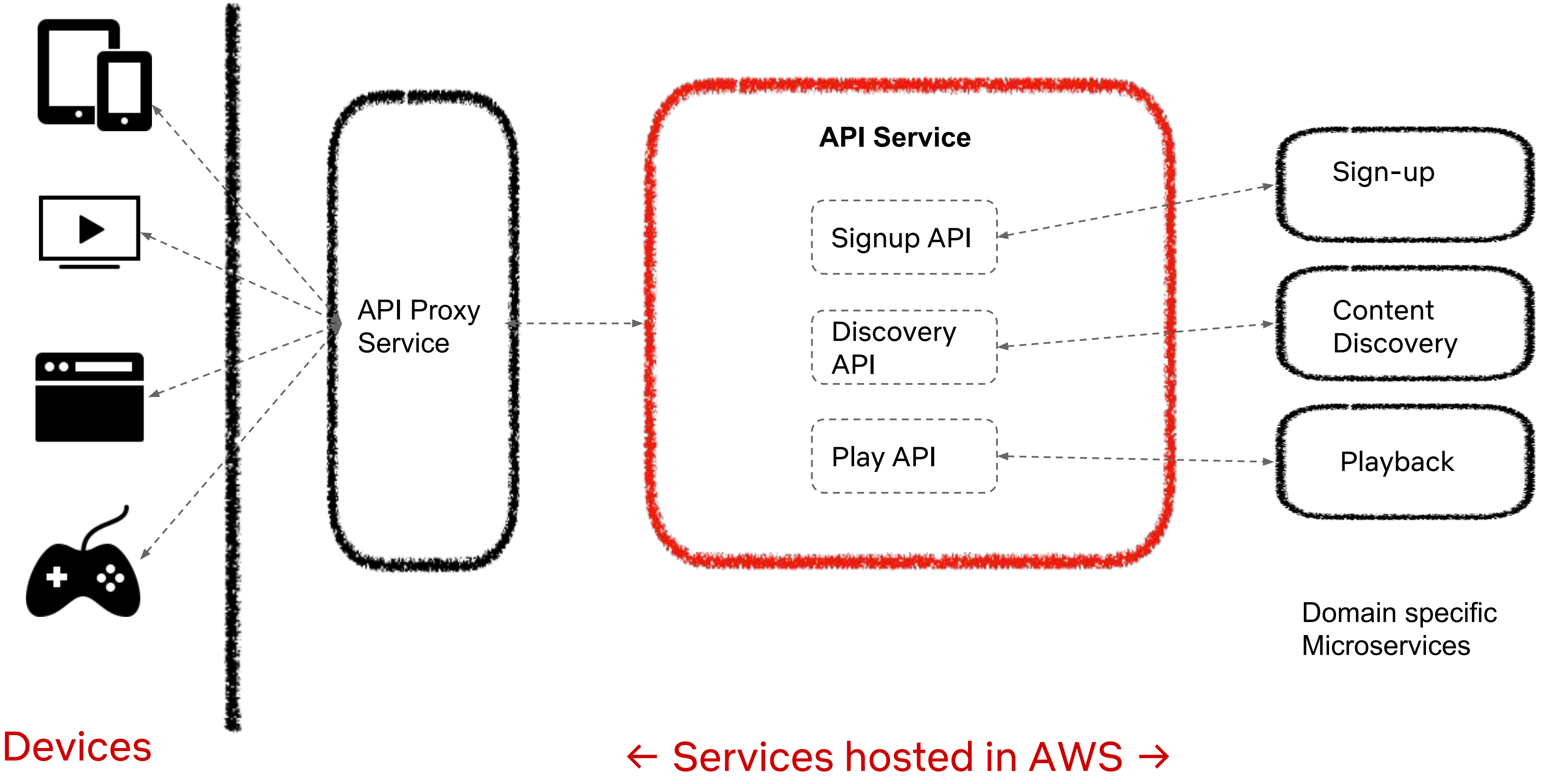
Playback

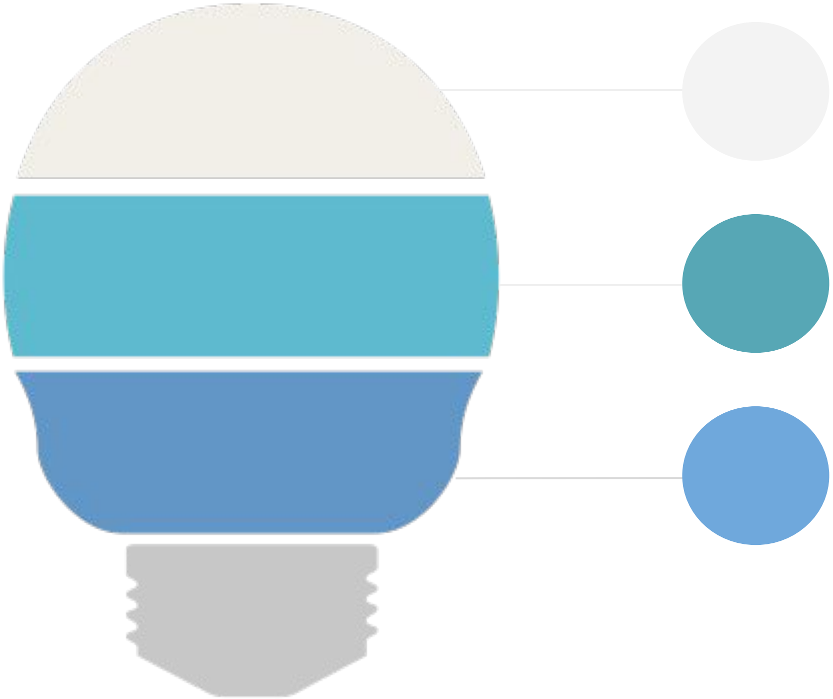
Domain specific Microservices

Devices

← Services hosted in AWS →

Previous Architecture

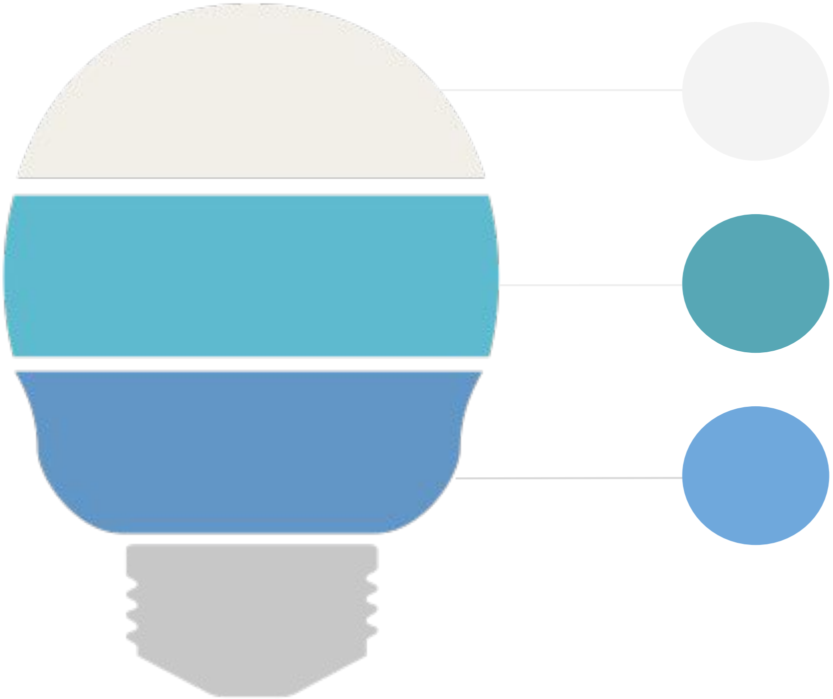




Identity

Type 1/2 Decisions

Evolvability

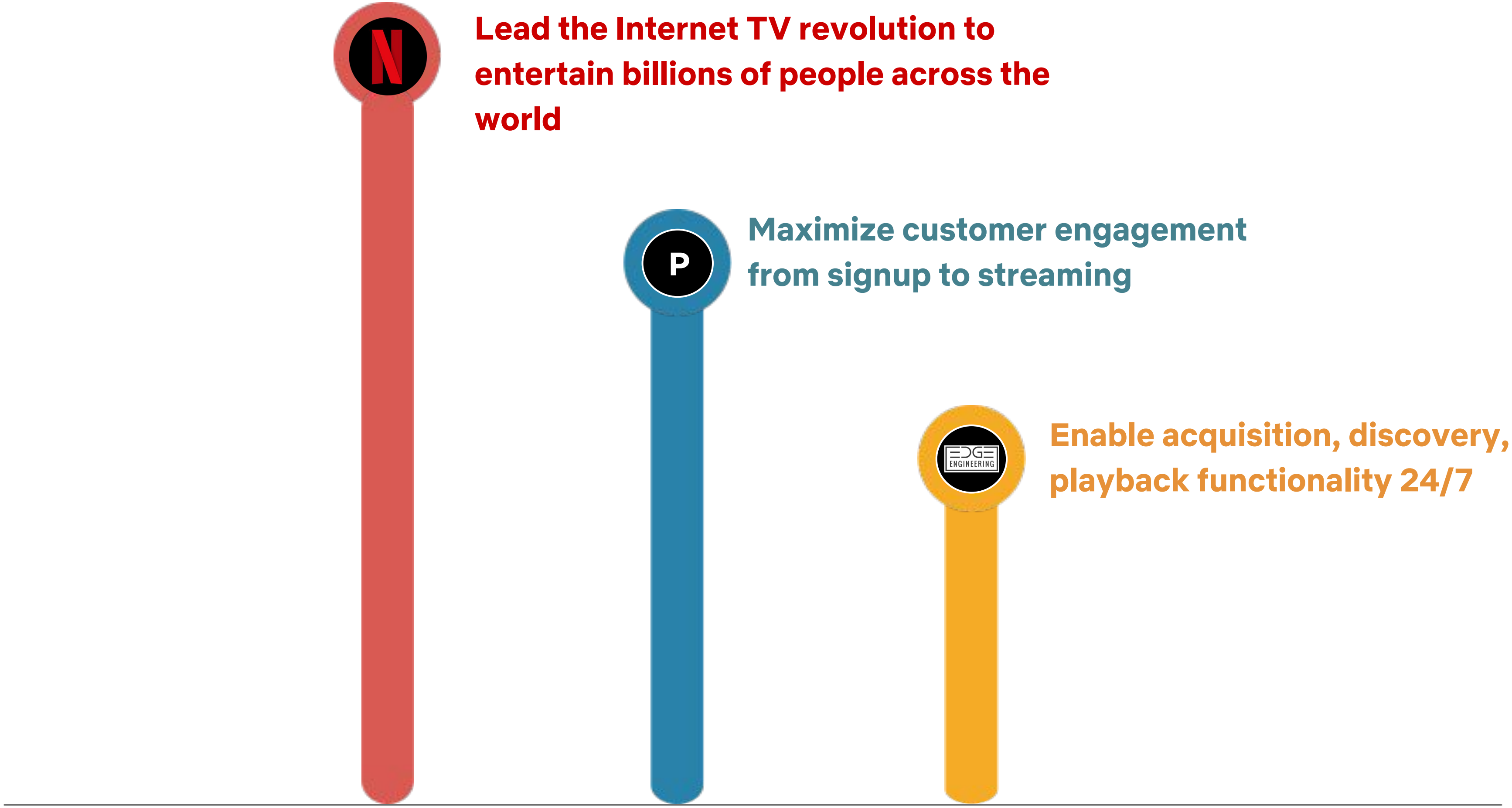


Identity

Type 1/2 Decisions

Evolvability

Start with WHY: Ask why your service exists



Lead the Internet TV revolution to entertain billions of people across the world

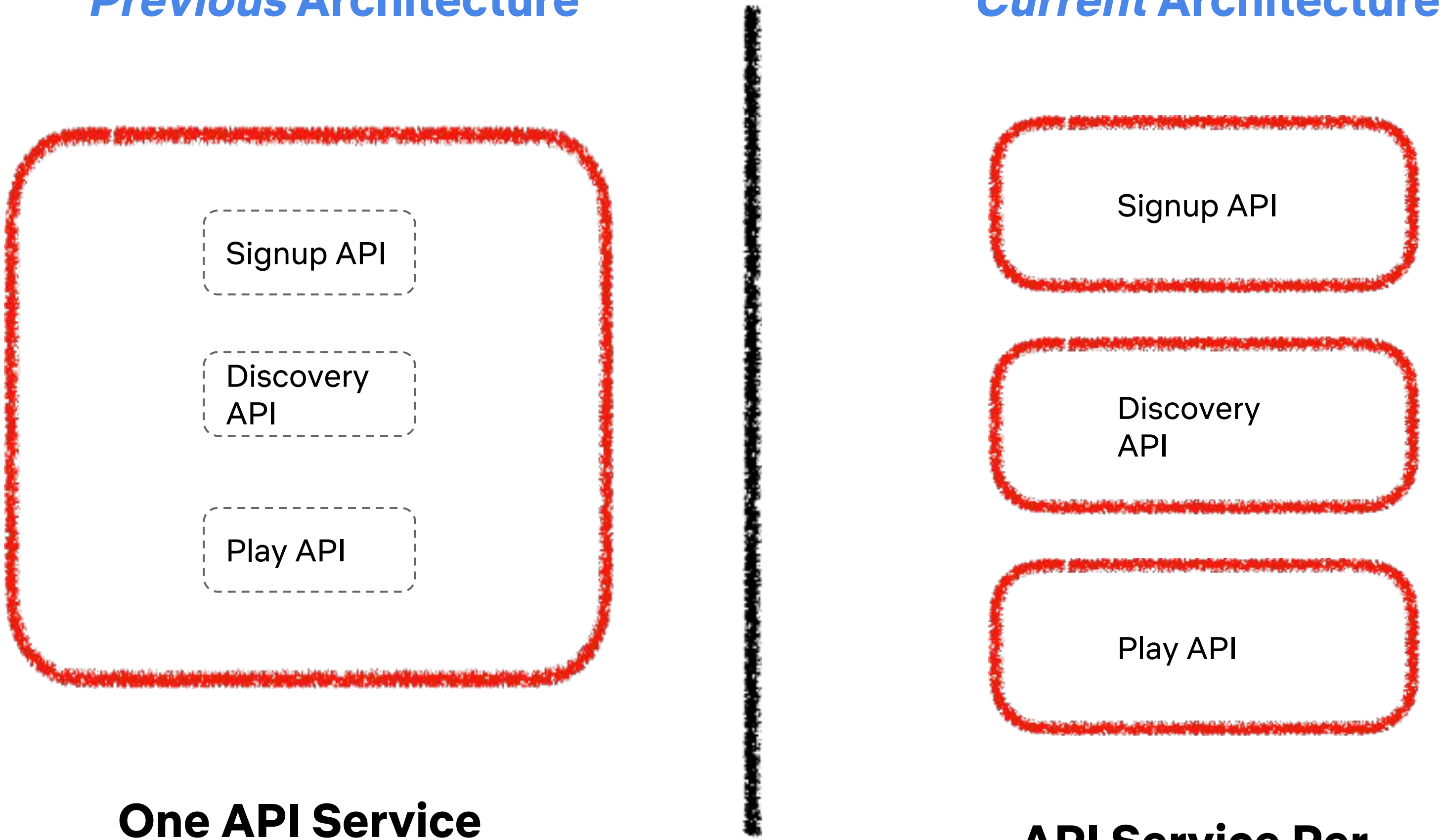
Maximize customer engagement from signup to streaming

Enable acquisition, discovery, playback functionality 24/7

API Identity: Deliver Acquisition, Discovery and Playback functions with high availability

Single Responsibility Principle: Be wary of multiple-identities rolled up into a single service

Previous Architecture



Signup API

Discovery API

Play API

One API Service

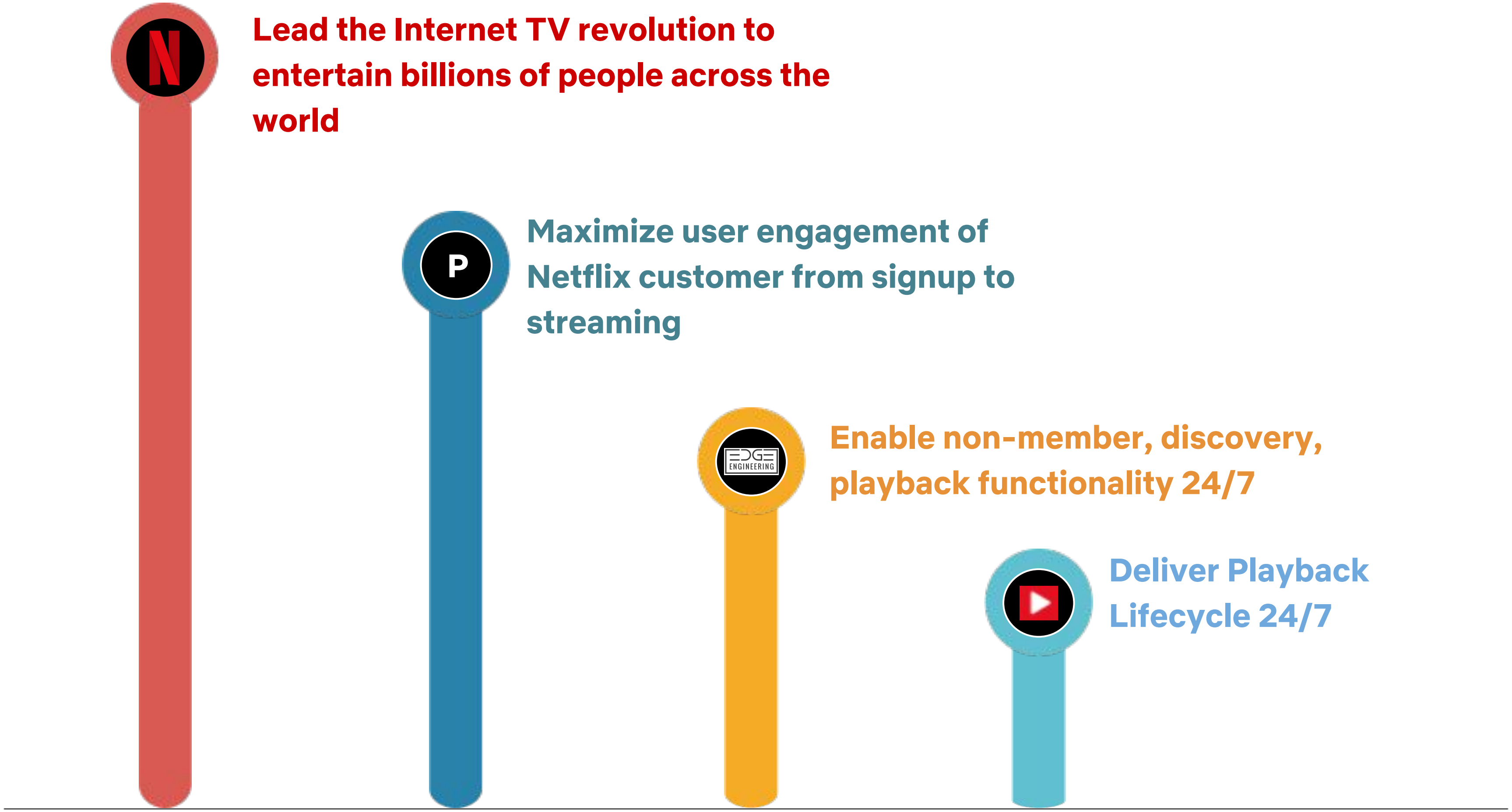
Current Architecture

Signup API

Discovery API

Play API

API Service Per function

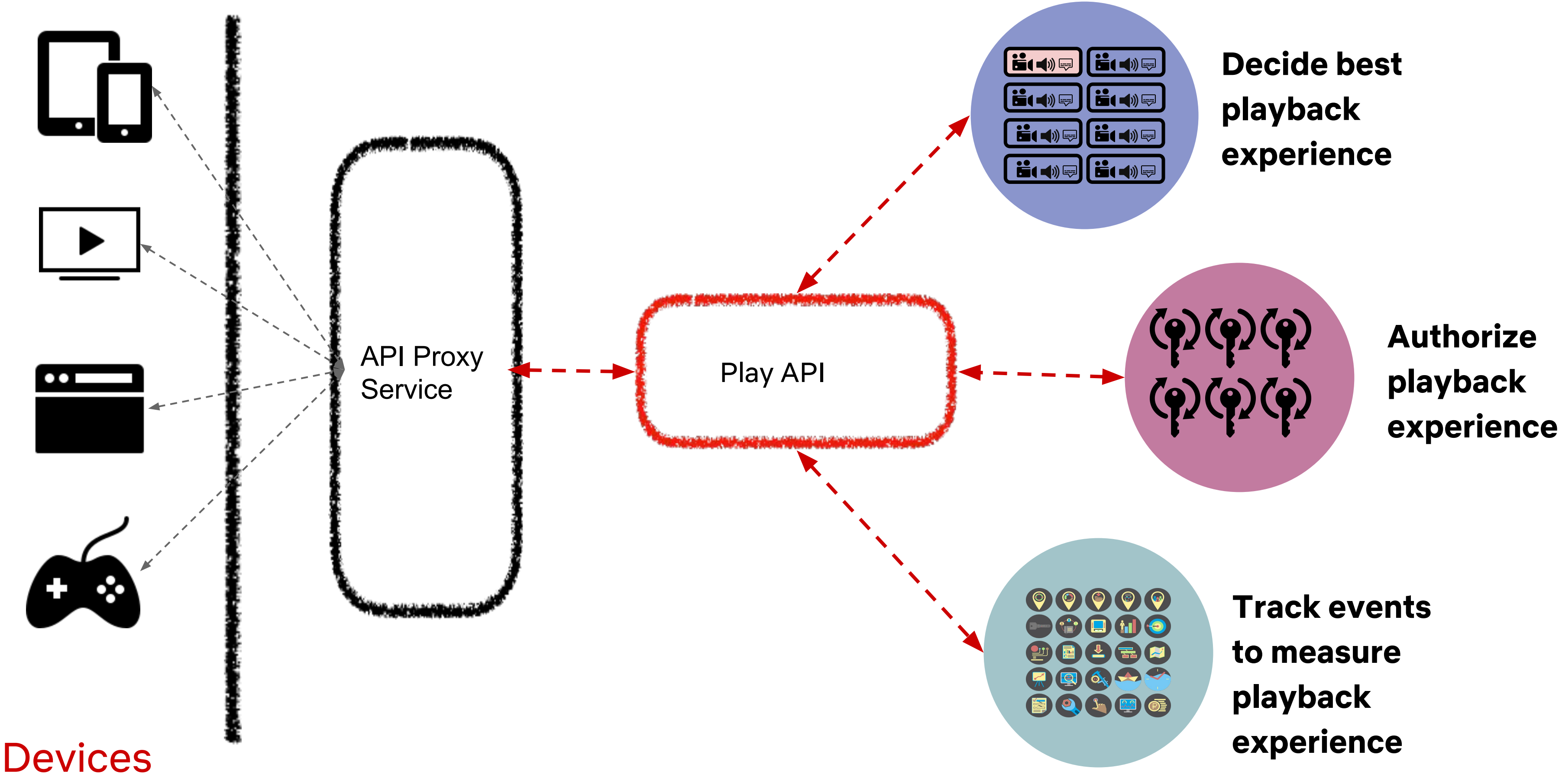


Lead the Internet TV revolution to entertain billions of people across the world

Maximize user engagement of Netflix customer from signup to streaming

Enable non-member, discovery, playback functionality 24/7

Deliver Playback Lifecycle 24/7



API Proxy Service

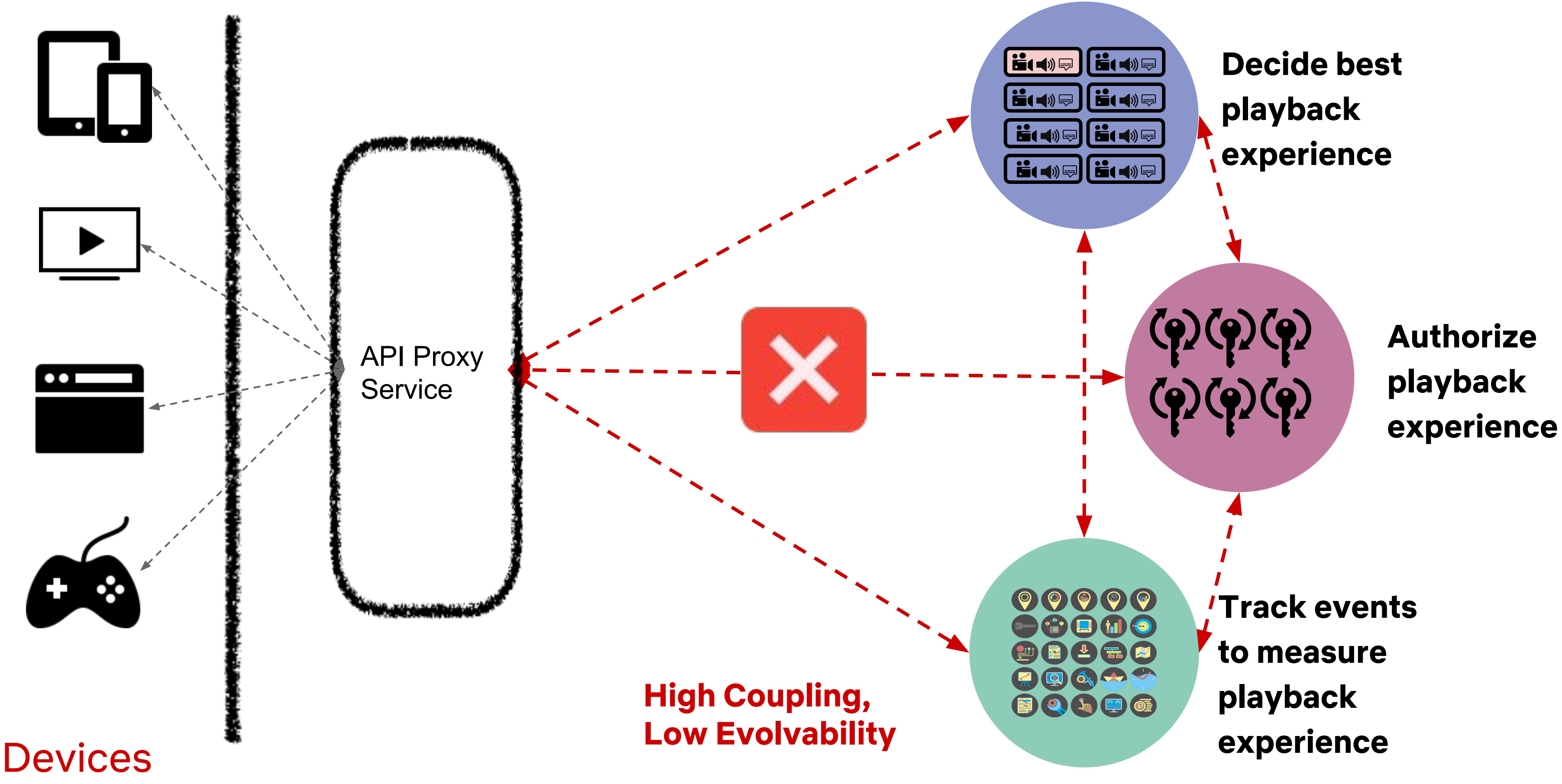
Play API

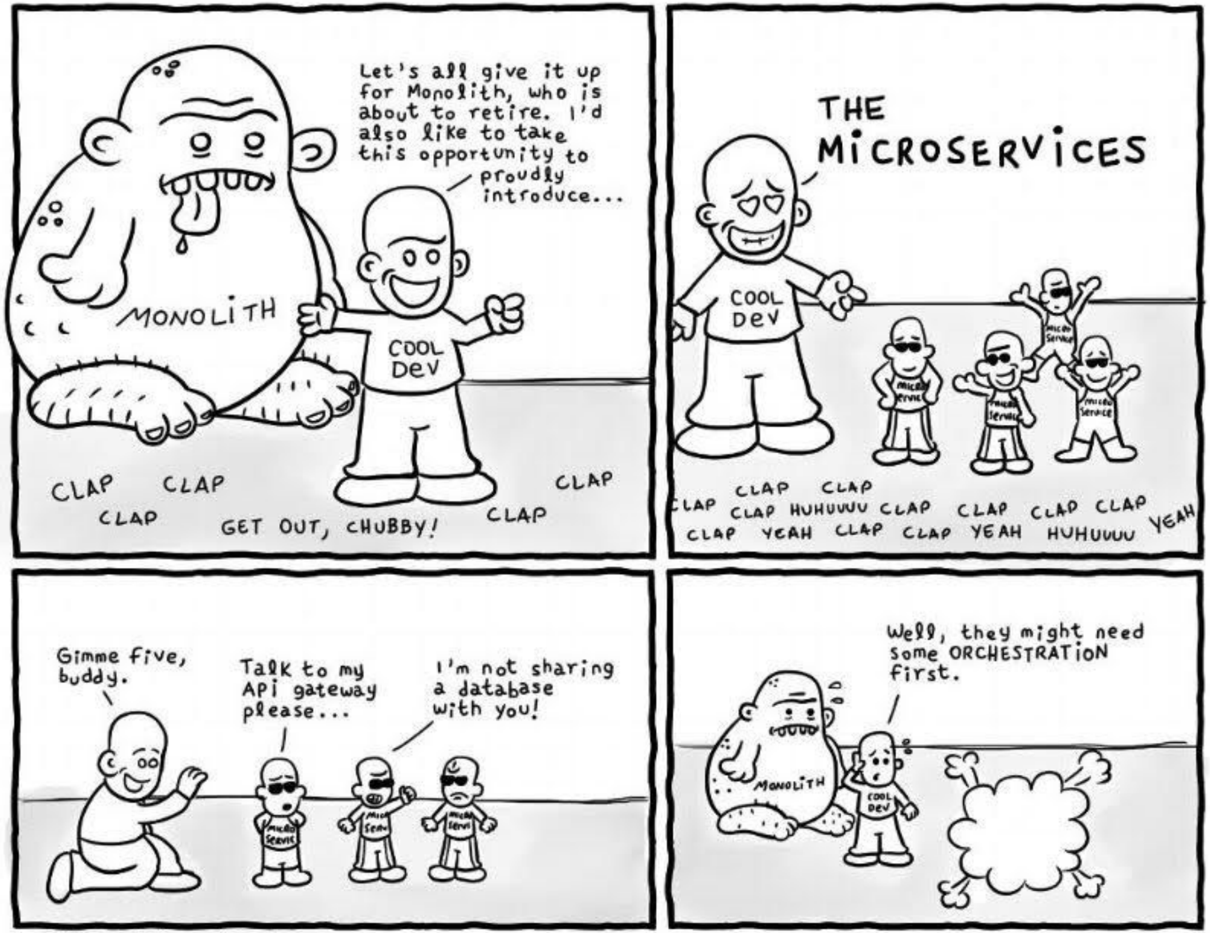
Decide best playback experience

Authorize playback experience

Track events to measure playback experience

Devices

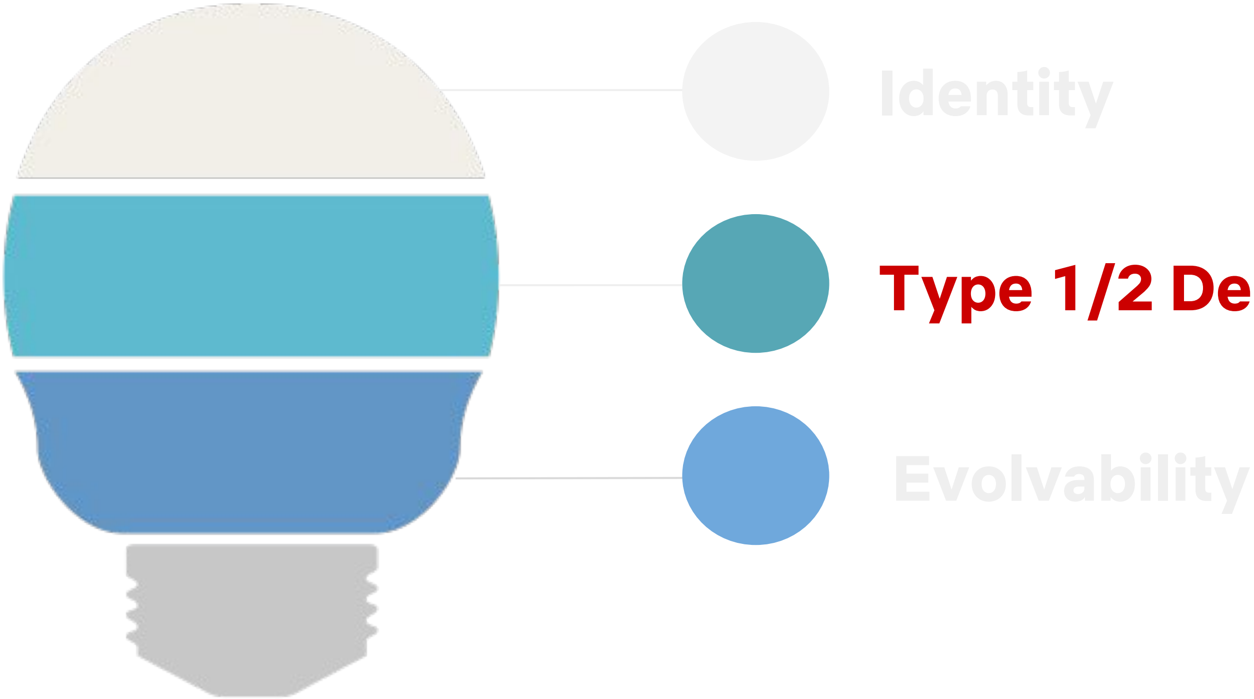

API Proxy Service
Decide best playback experience
Authorize playback experience
Track events to measure playback experience
High Coupling, Low Evolvability
Devices


Let's all give it up for Monolith, who is about to retire. I'd also like to take this opportunity to proudly introduce...
MONOLITH
COOL DEV
CLAP CLAP CLAP
GET OUT, CHUBBY!
CLAP CLAP
THE MICROSERVICES
COOL DEV
MICRO SERVICE
CLAP CLAP CLAP CLAP HUHUUUU CLAP CLAP CLAP CLAP YEAH
CLAP YEAH CLAP CLAP YEAH HUHUUUU
Gimme five, buddy.
Talk to my API gateway please...
I'm not sharing a database with you!
Well, they might need some ORCHESTRATION first.
MONOLITH
COOL DEV

Daniel Stori {turnoff.us}

Play API Identity: Orchestrate Playback Lifecycle with stable abstractions

Guiding Principle: We believe in a simple singular identity for our services. The identity relates to and complements the identities of the company, organization, team and its peer services


Identity
Type 1/2 Decisions
Evolvability

“Some decisions are consequential and irreversible or nearly irreversible – one-way doors – and these decisions must be made methodically, carefully, slowly, with great deliberation and consultation [...] We can call these Type 1 decisions…”

Quote from Jeff Bezos

"...But most decisions aren't like that – they are changeable, reversible – they're two-way doors. If you've made a suboptimal Type 2 decision, you don't have to live with the consequences for that long [...] Type 2 decisions can and should be made quickly by high judgment individuals or small groups."

Quote from Jeff Bezos

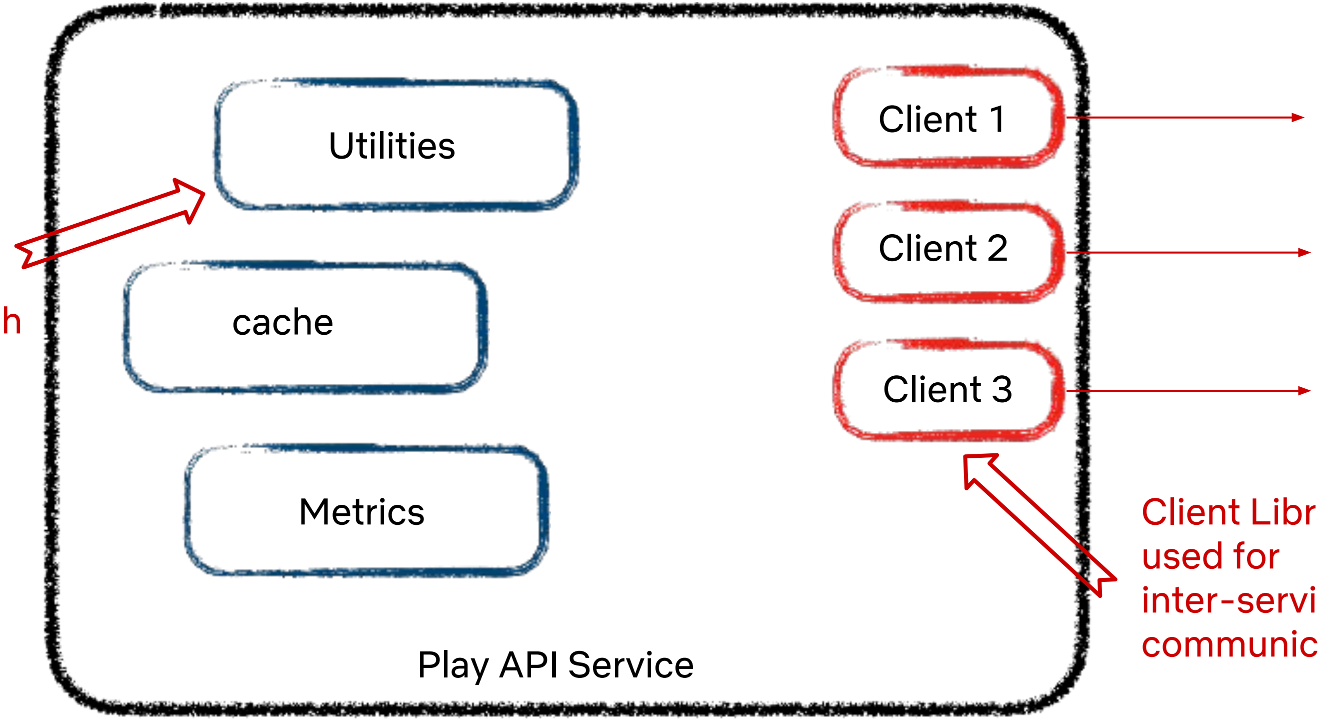
Three Type 1 Decisions to Consider

Appropriate Coupling

Synchronous & Asynchronous

Data Architecture

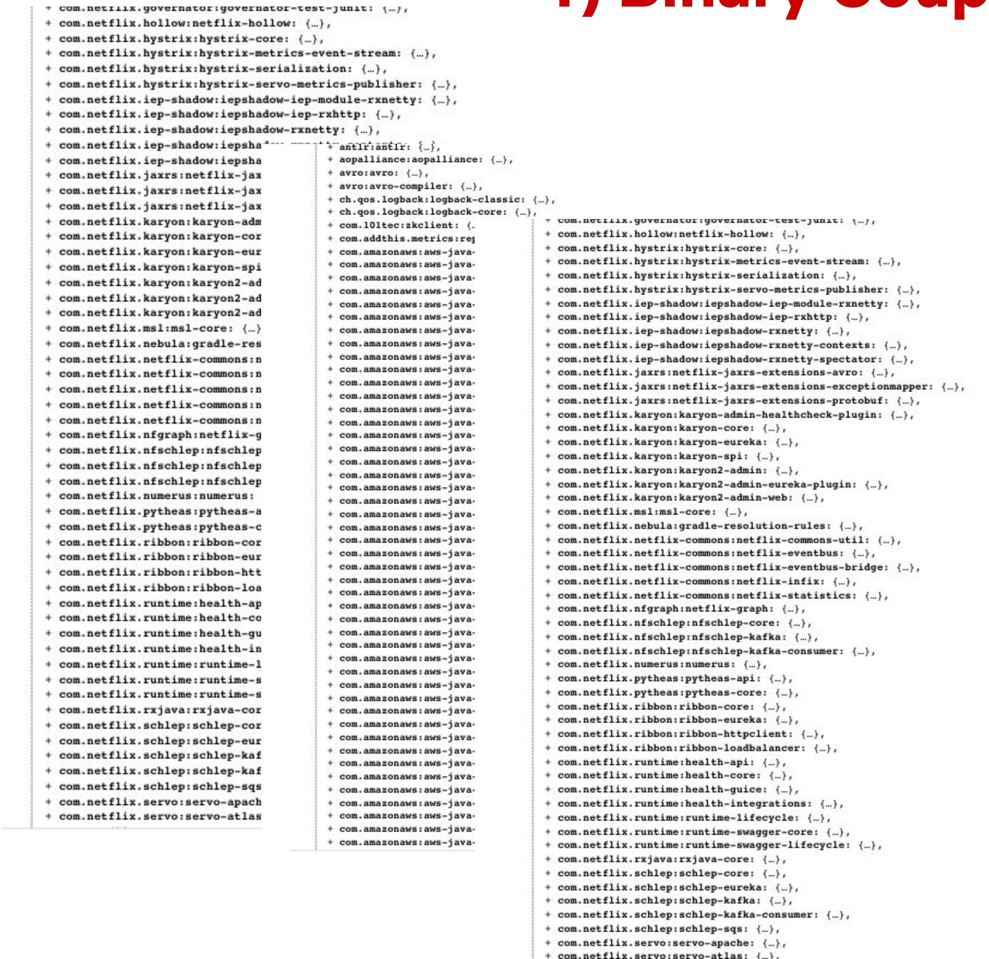
Two types of Shared Libraries



Shared Libraries with common functions

Client Libraries used for inter-service communications

1) Binary Coupling



```
+ com.netflix.governator:governator-test-junit: {…},
+ com.netflix.hollow:netflix-hollow: {…},
+ com.netflix.hystrix:hystrix-core: {…},
+ com.netflix.hystrix:hystrix-metrics-event-stream: {…},
+ com.netflix.hystrix:hystrix-serialization: {…},
+ com.netflix.hystrix:hystrix-servo-metrics-publisher: {…},
+ com.netflix.iep-shadow:iepshadow-iep-module-rxnetty: {…},
+ com.netflix.iep-shadow:iepshadow-iep-rxhttp: {…},
+ com.netflix.iep-shadow:iepshadow-rxnetty: {…},
+ com.netflix.iep-shadow:iepsha
+ com.netflix.iep-shadow:iepsha
+ com.netflix.jaxrs:netflix-jax
+ com.netflix.jaxrs:netflix-jax
+ com.netflix.jaxrs:netflix-jax
+ com.netflix.karyon:karyon-adm
+ com.netflix.karyon:karyon-cor
+ com.netflix.karyon:karyon-eur
+ com.netflix.karyon:karyon-spi
+ com.netflix.karyon:karyon2-ad
+ com.netflix.karyon:karyon2-ad
+ com.netflix.karyon:karyon2-ad
+ com.netflix.msl:msl-core: {…}
+ com.netflix.nebula:gradle-res
+ com.netflix.netflix-commons:n
+ com.netflix.netflix-commons:n
+ com.netflix.netflix-commons:n
+ com.netflix.netflix-commons:n
+ com.netflix.netflix-commons:n
+ com.netflix.nfgraph:netflix-g
+ com.netflix.nfschlep:nfschlep
+ com.netflix.nfschlep:nfschlep
+ com.netflix.nfschlep:nfschlep
+ com.netflix.numerus:numerus:
+ com.netflix.pytheas:pytheas-a
+ com.netflix.pytheas:pytheas-c
+ com.netflix.ribbon:ribbon-cor
+ com.netflix.ribbon:ribbon-eur
+ com.netflix.ribbon:ribbon-htt
+ com.netflix.ribbon:ribbon-loa
+ com.netflix.runtime:health-ap
+ com.netflix.runtime:health-co
+ com.netflix.runtime:health-gu
+ com.netflix.runtime:health-in
+ com.netflix.runtime:runtime-l
+ com.netflix.runtime:runtime-s
+ com.netflix.runtime:runtime-s
+ com.netflix.rxjava:rxjava-cor
+ com.netflix.schlep:schlep-cor
+ com.netflix.schlep:schlep-eur
+ com.netflix.schlep:schlep-kaf
+ com.netflix.schlep:schlep-kaf
+ com.netflix.schlep:schlep-sqs
+ com.netflix.servo:servo-apach
+ com.netflix.servo:servo-atlas
```

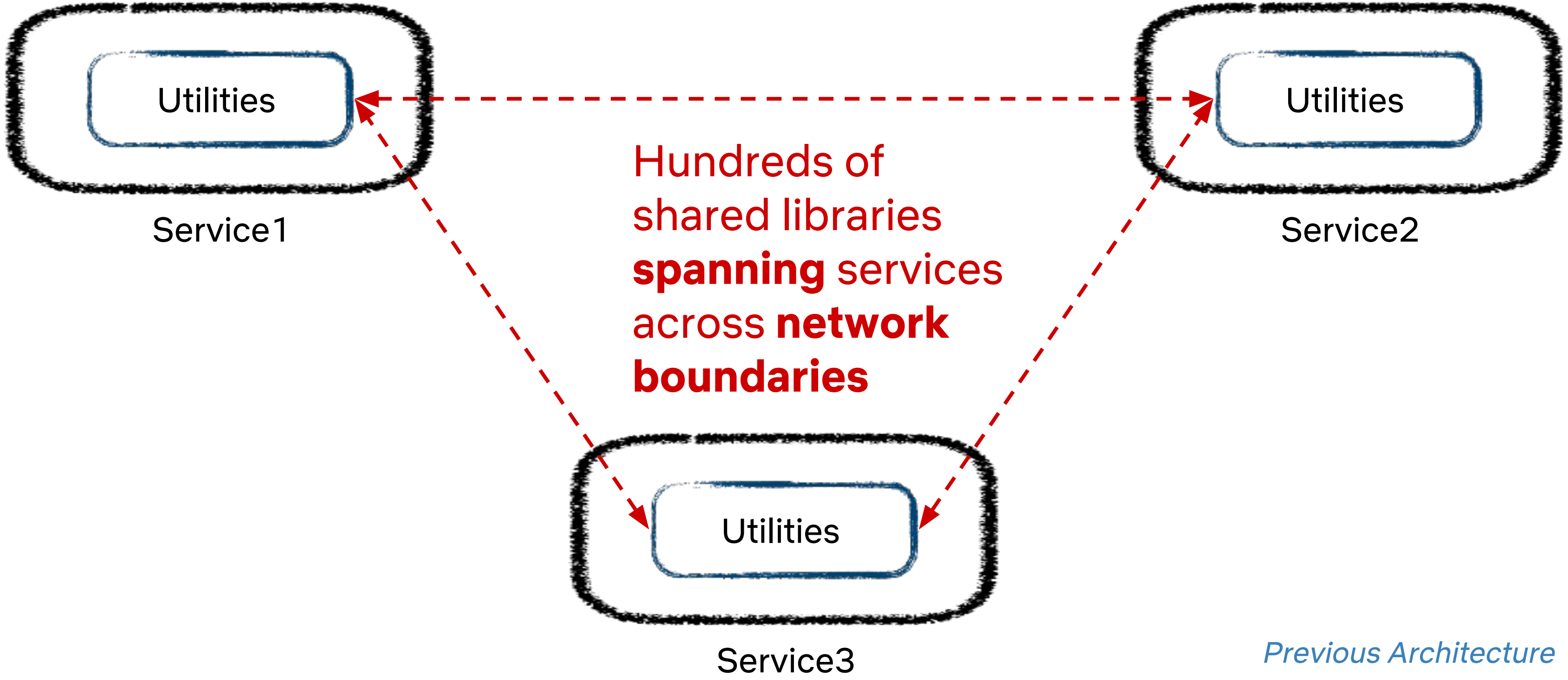
```
+ antlr:antlr: {…},
+ aopalliance:aopalliance: {…},
+ avro:avro: {…},
+ avro:avro-compiler: {…},
+ ch.qos.logback:logback-classic: {…},
+ ch.qos.logback:logback-core: {…},
+ com.101tec:zkclient: {
+ com.addthis.metrics:re
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
+ com.amazonaws:aws-java-
```

```
+ com.netflix.governator:governator-test-junit: {…},
+ com.netflix.hollow:netflix-hollow: {…},
+ com.netflix.hystrix:hystrix-core: {…},
+ com.netflix.hystrix:hystrix-metrics-event-stream: {…},
+ com.netflix.hystrix:hystrix-serialization: {…},
+ com.netflix.hystrix:hystrix-servo-metrics-publisher: {…},
+ com.netflix.iep-shadow:iepshadow-iep-module-rxnetty: {…},
+ com.netflix.iep-shadow:iepshadow-iep-rxhttp: {…},
+ com.netflix.iep-shadow:iepshadow-rxnetty: {…},
+ com.netflix.iep-shadow:iepshadow-rxnetty-contexts: {…},
+ com.netflix.iep-shadow:iepshadow-rxnetty-spectator: {…},
+ com.netflix.jaxrs:netflix-jaxrs-extensions-avro: {…},
+ com.netflix.jaxrs:netflix-jaxrs-extensions-exceptionmapper: {…},
+ com.netflix.jaxrs:netflix-jaxrs-extensions-protobuf: {…},
+ com.netflix.karyon:karyon-admin-healthcheck-plugin: {…},
+ com.netflix.karyon:karyon-core: {…},
+ com.netflix.karyon:karyon-eureka: {…},
+ com.netflix.karyon:karyon-spi: {…},
+ com.netflix.karyon:karyon2-admin: {…},
+ com.netflix.karyon:karyon2-admin-eureka-plugin: {…},
+ com.netflix.karyon:karyon2-admin-web: {…},
+ com.netflix.msl:msl-core: {…},
+ com.netflix.nebula:gradle-resolution-rules: {…},
+ com.netflix.netflix-commons:netflix-commons-util: {…},
+ com.netflix.netflix-commons:netflix-eventbus: {…},
+ com.netflix.netflix-commons:netflix-eventbus-bridge: {…},
+ com.netflix.netflix-commons:netflix-infix: {…},
+ com.netflix.netflix-commons:netflix-statistics: {…},
+ com.netflix.nfgraph:netflix-graph: {…},
+ com.netflix.nfschlep:nfschlep-core: {…},
+ com.netflix.nfschlep:nfschlep-kafka: {…},
+ com.netflix.nfschlep:nfschlep-kafka-consumer: {…},
+ com.netflix.numerus:numerus: {…},
+ com.netflix.pytheas:pytheas-api: {…},
+ com.netflix.pytheas:pytheas-core: {…},
+ com.netflix.ribbon:ribbon-core: {…},
+ com.netflix.ribbon:ribbon-eureka: {…},
+ com.netflix.ribbon:ribbon-httpclient: {…},
+ com.netflix.ribbon:ribbon-loadbalancer: {…},
+ com.netflix.runtime:health-api: {…},
+ com.netflix.runtime:health-core: {…},
+ com.netflix.runtime:health-guice: {…},
+ com.netflix.runtime:health-integrations: {…},
+ com.netflix.runtime:runtime-lifecycle: {…},
+ com.netflix.runtime:runtime-swagger-core: {…},
+ com.netflix.runtime:runtime-swagger-lifecycle: {…},
+ com.netflix.rxjava:rxjava-core: {…},
+ com.netflix.schlep:schlep-core: {…},
+ com.netflix.schlep:schlep-eureka: {…},
+ com.netflix.schlep:schlep-kafka: {…},
+ com.netflix.schlep:schlep-kafka-consumer: {…},
+ com.netflix.schlep:schlep-sqs: {…},
+ com.netflix.servo:servo-apache: {…},
+ com.netflix.servo:servo-atlas: {…},
```

“Thick” shared libraries with 100s of dependent libraries (e.g. utilities jar)

Previous Architecture

Binary coupling => Distributed Monolith

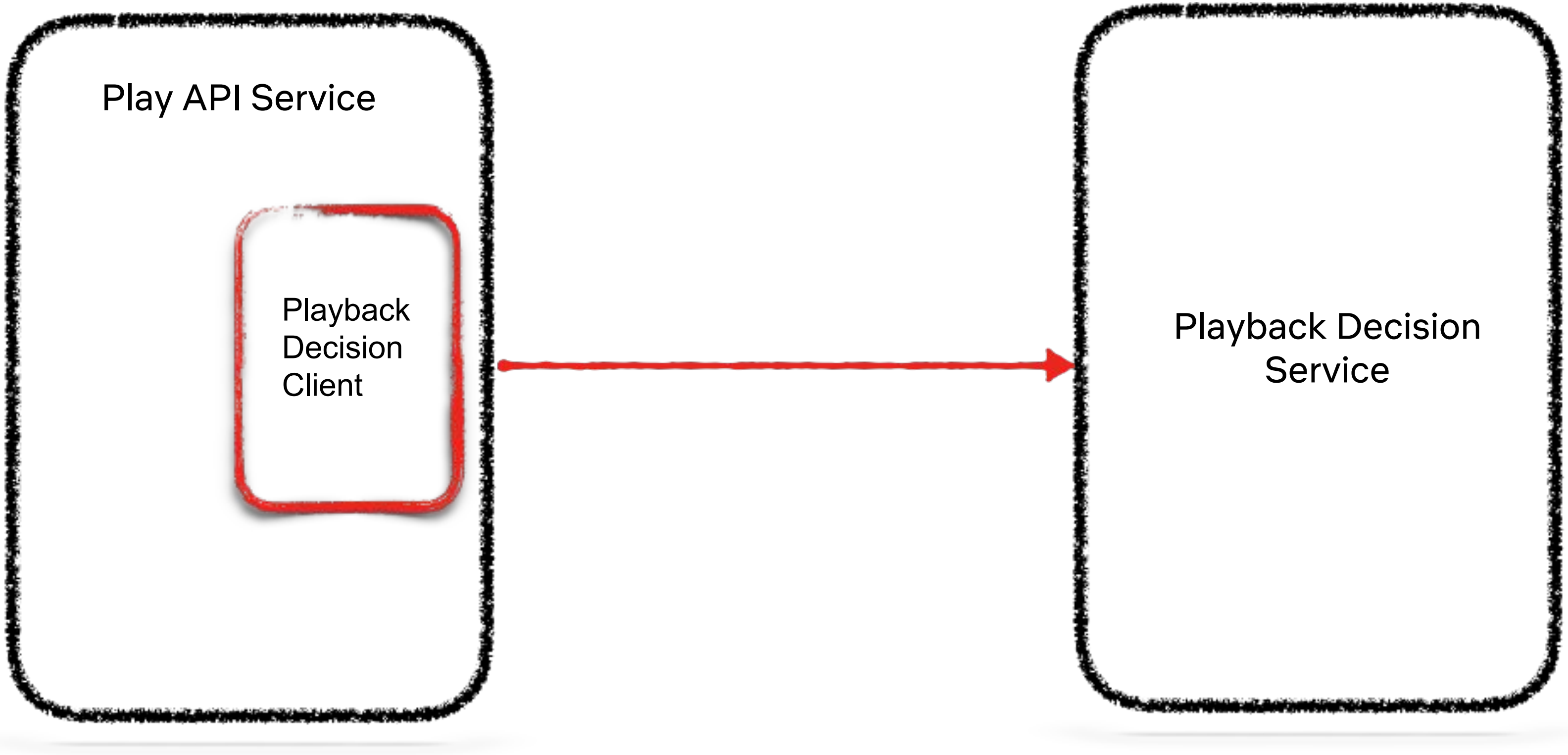


Hundreds of shared libraries **spanning** services across **network boundaries**

Previous Architecture

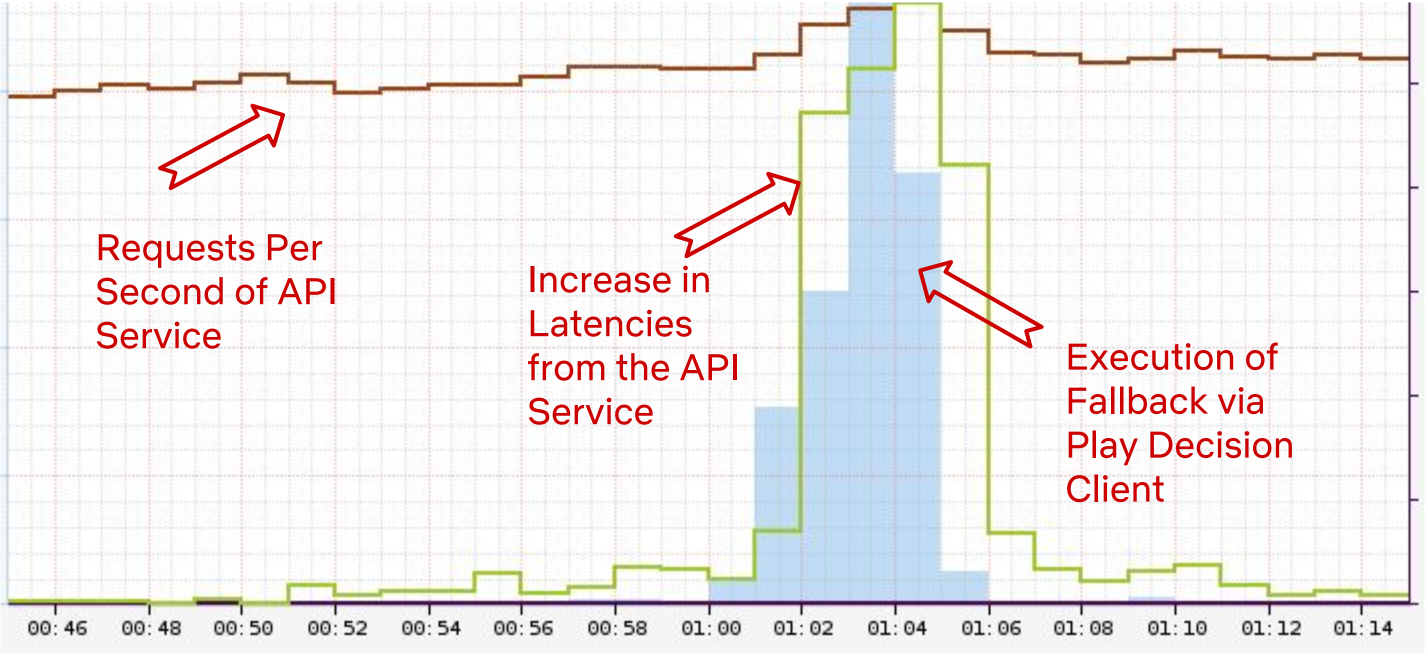
“The evils of too much coupling between services are far worse than the problems caused by code duplication”

- Sam Newman (Building Microservices)



Previous Architecture

Clients with heavy Fallbacks

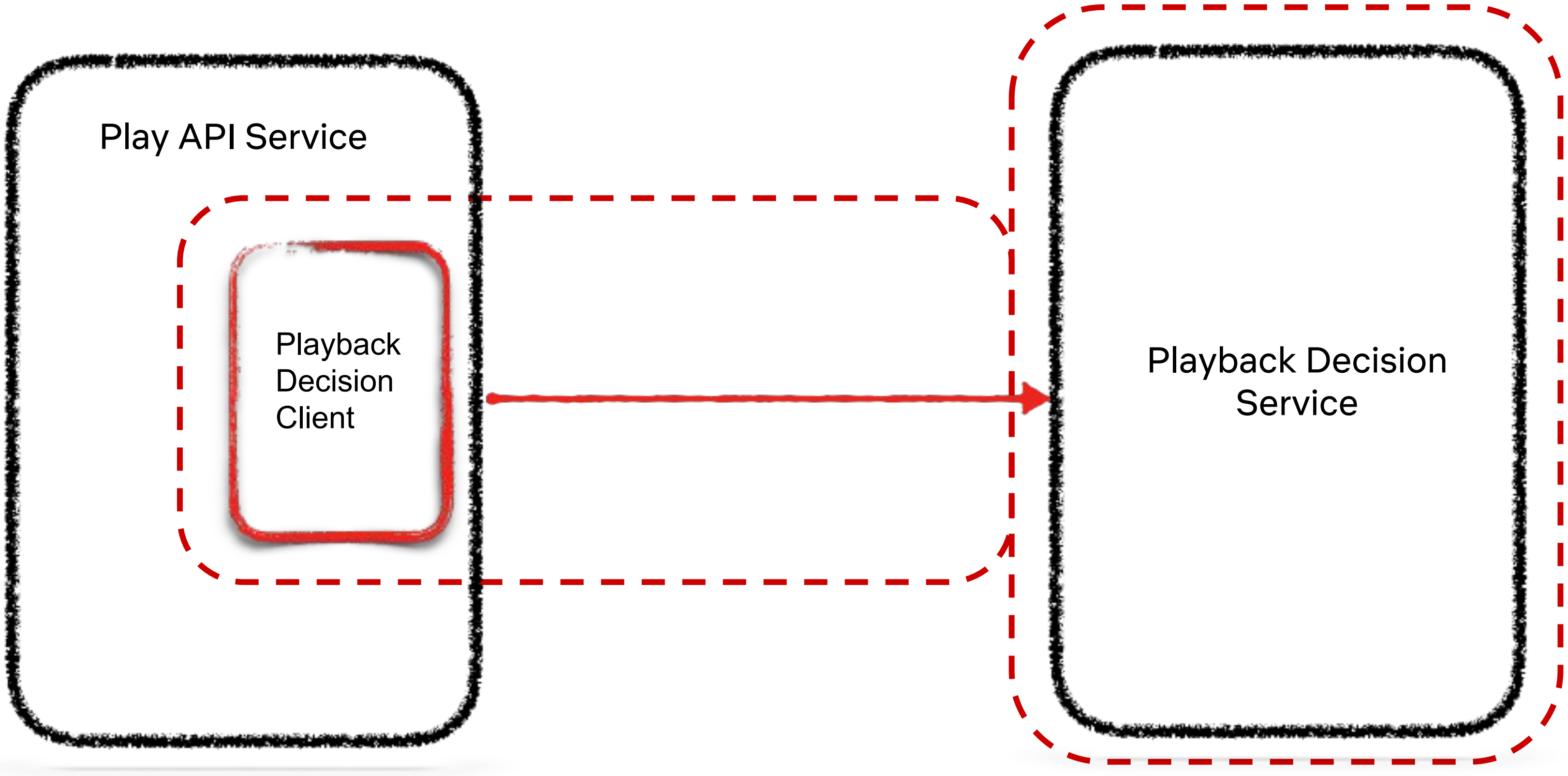


Requests Per Second of API Service

Increase in Latencies from the API Service

Execution of Fallback via Play Decision Client

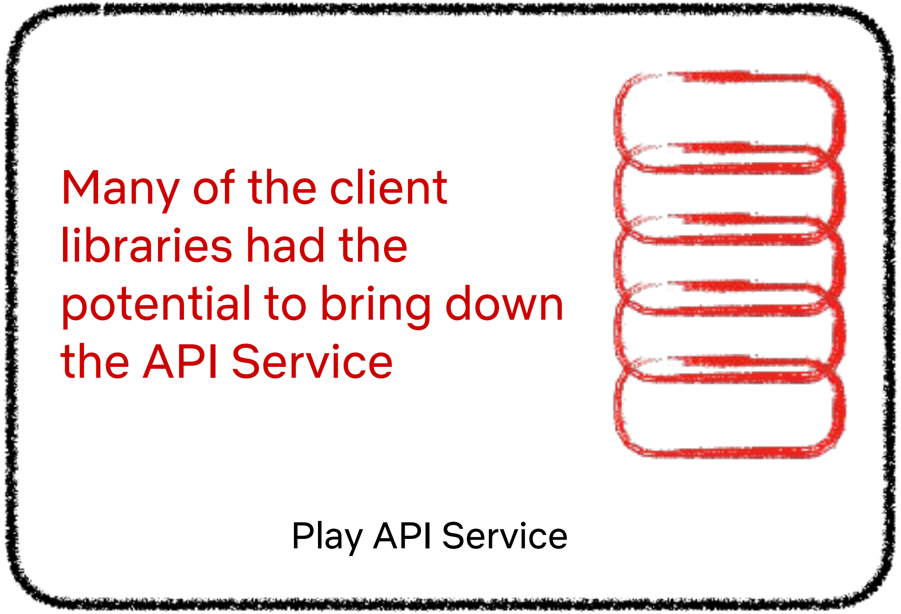
2) Operational Coupling



Previous Architecture

“Operational Coupling” might be an ok choice, if some services/teams are not yet ready to own and operate a highly available service.

Operational Coupling impacts Availability

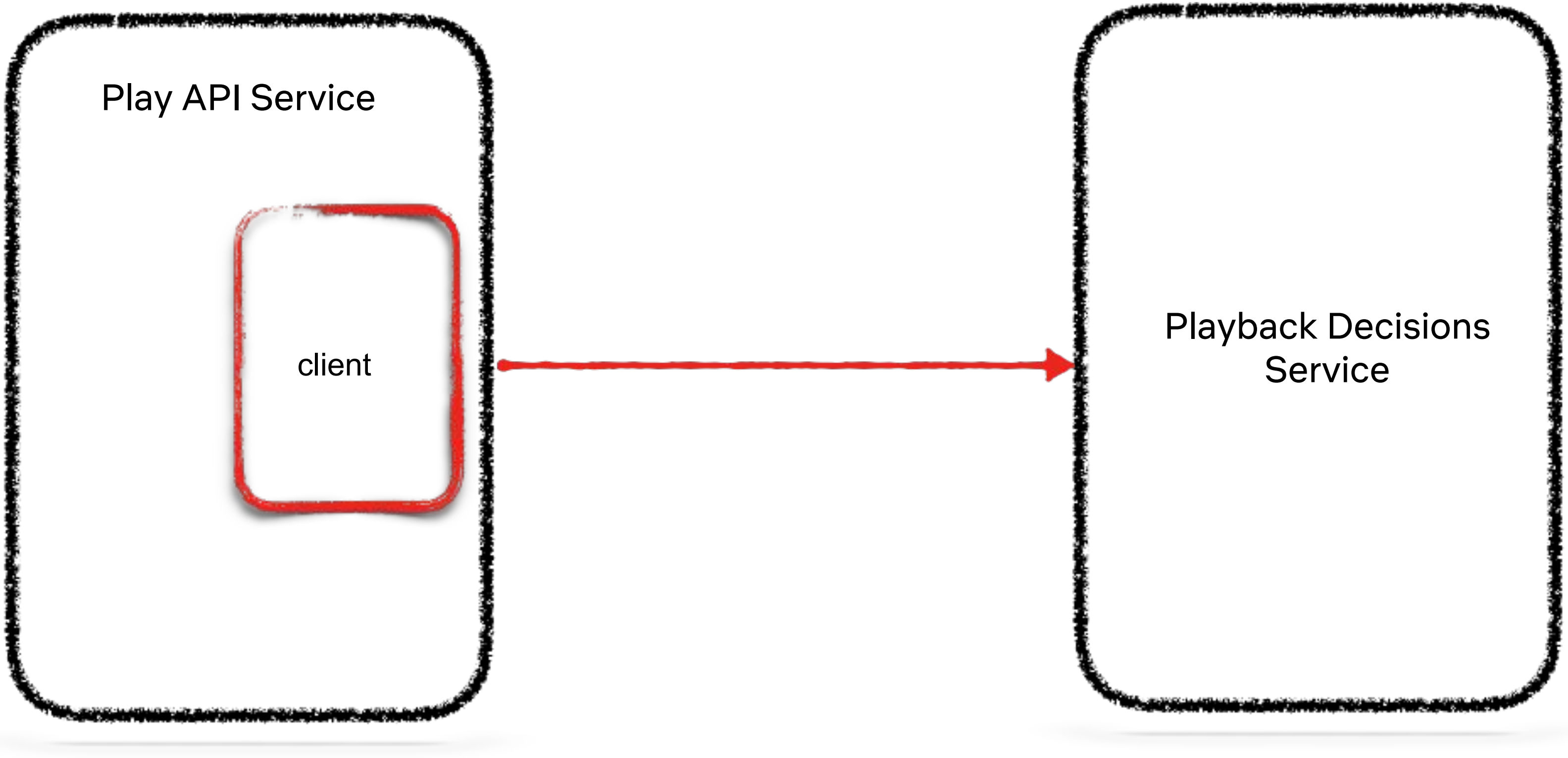


Many of the client libraries had the potential to bring down the API Service

Play API Service

Previous Architecture

3) Language Coupling

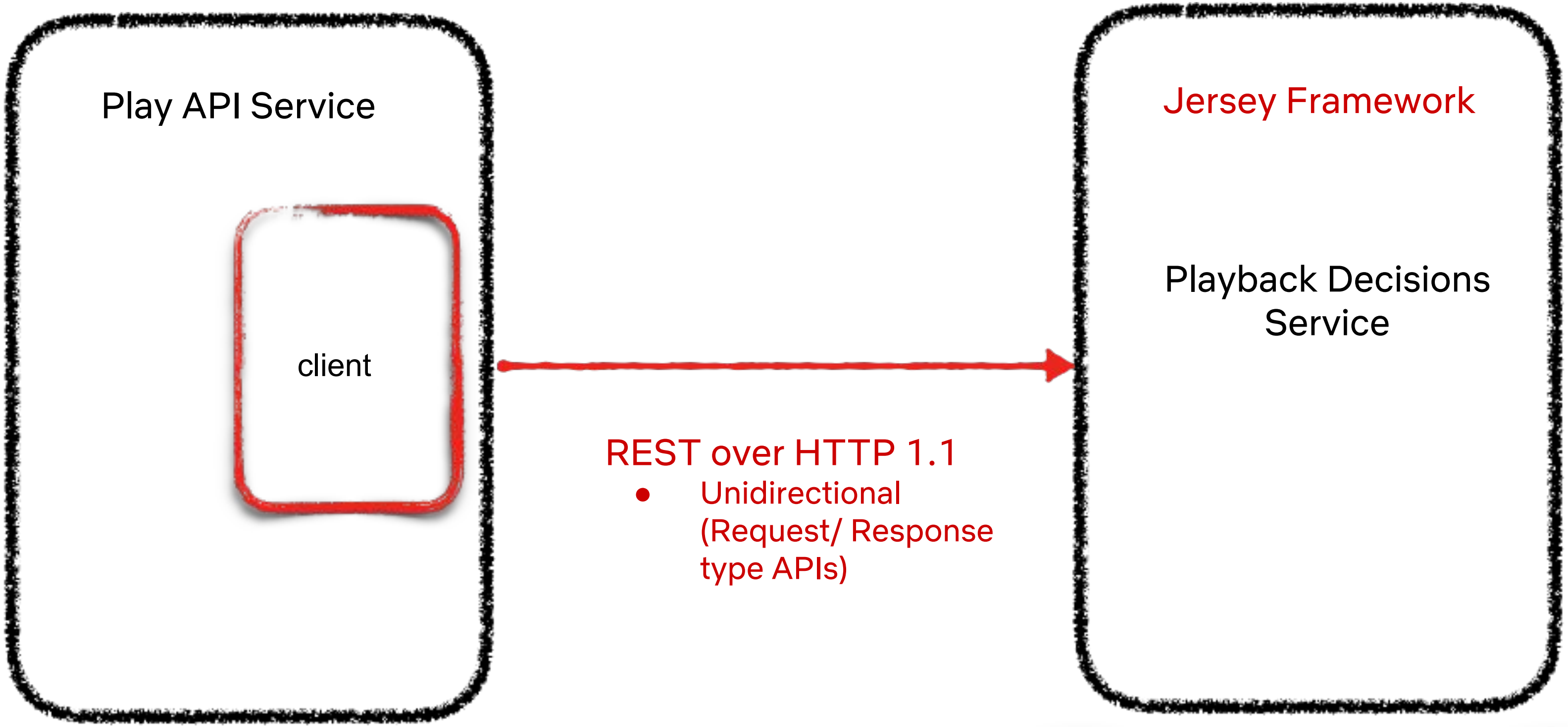


Java

Java

Previous Architecture

Communication Protocol



REST over HTTP 1.1

- Unidirectional (Request/ Response type APIs)

Previous Architecture

Requirements

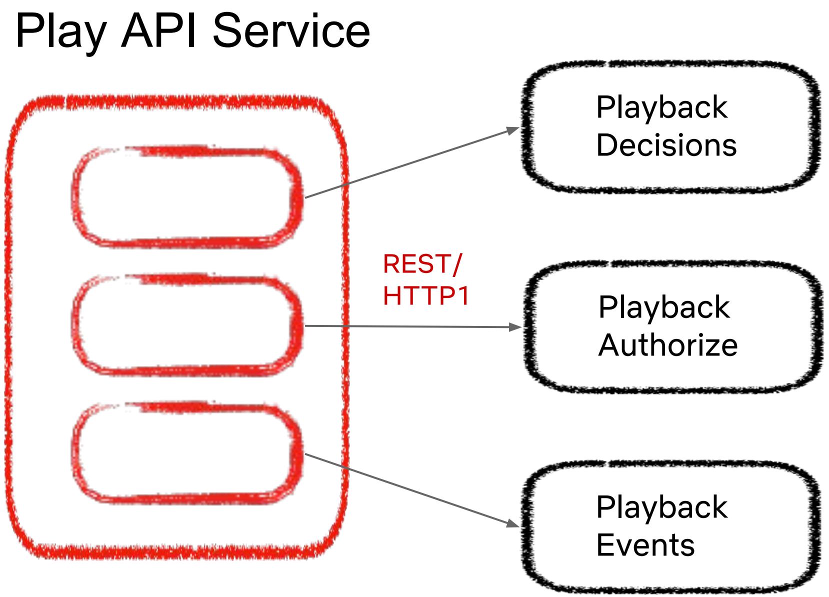
- Operationally “thin” Clients
- No or limited shared libraries
- Auto-generated clients for Polyglot support
- Bi-Directional Communication

REST vs RPC

- At Netflix, most use-cases were modelled as Request/Response
  - REST was a simple and easy way of communicating between services; so choice of REST was more incidental rather than intentional
- Most of the services were not following RESTful principles.
  - The URL didn’t represent a unique resource, instead the parameters passed in the call determined the response - effectively made them a RPC call
- So we were agnostic to REST vs RPC as long as it meets our requirements

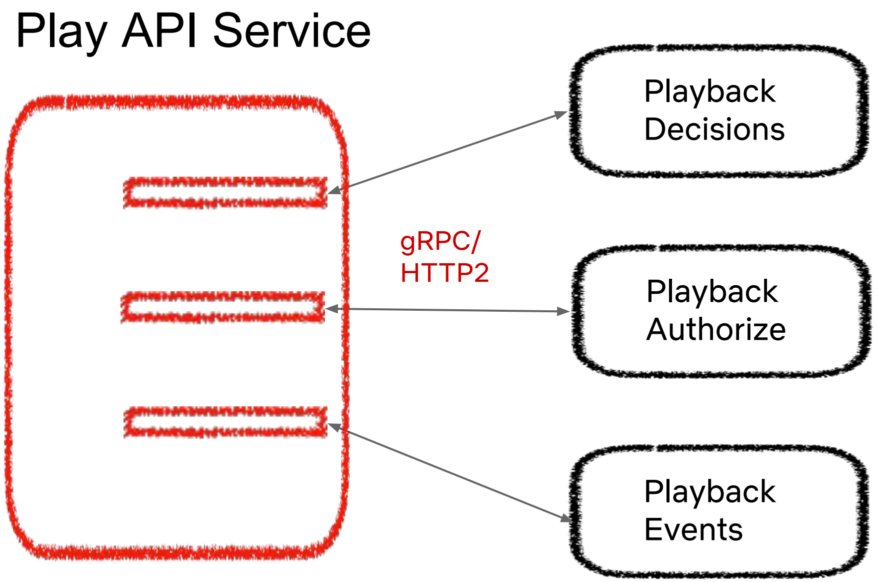
GRPC

Previous Architecture



1) Operationally Coupled Clients
2) High Binary Coupling
3) Only Java
4) Unidirectional communication

Current Architecture

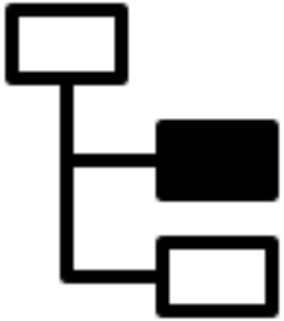


1) Minimal Operational Coupling
2) Limited Binary Coupling
3) Beyond Java
4) Beyond Request/ Response

Type 1 Decision: Appropriate Coupling

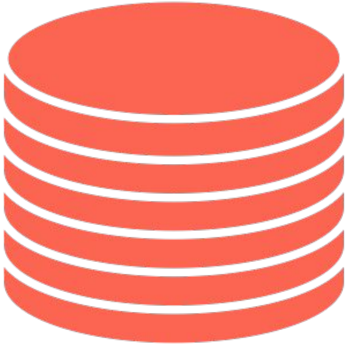
Consider “thin” auto-generated clients with bi-directional communication and minimize code reuse across service boundaries

Three Type 1 Decisions to Consider



✓ Appropriate Coupling

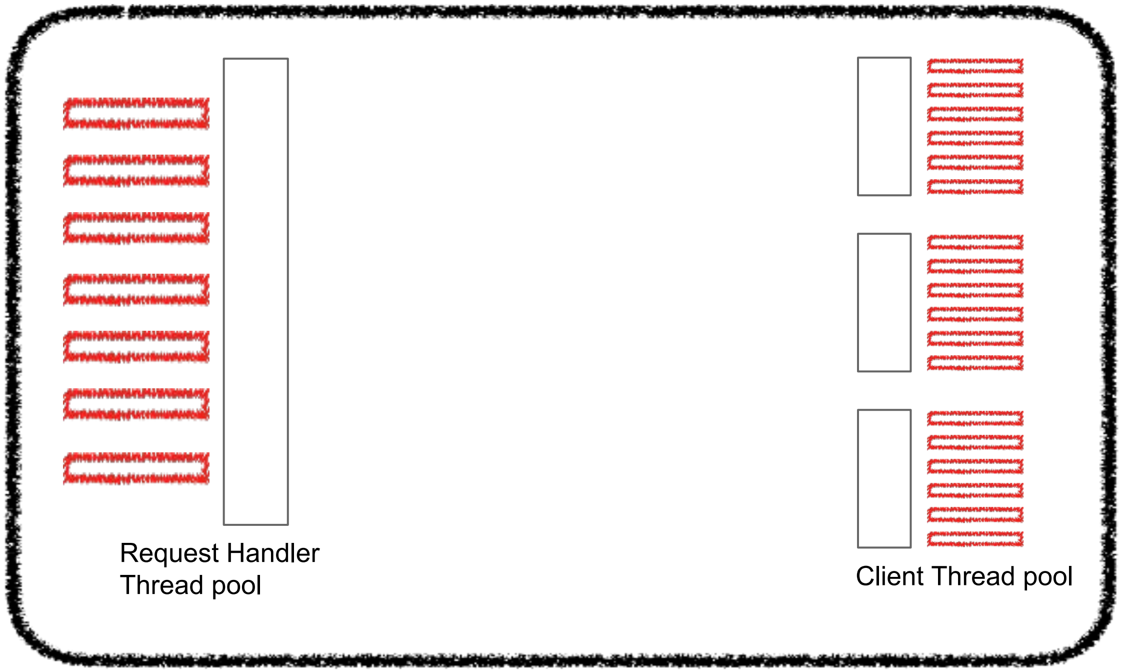
Synchronous vs Asynchronous



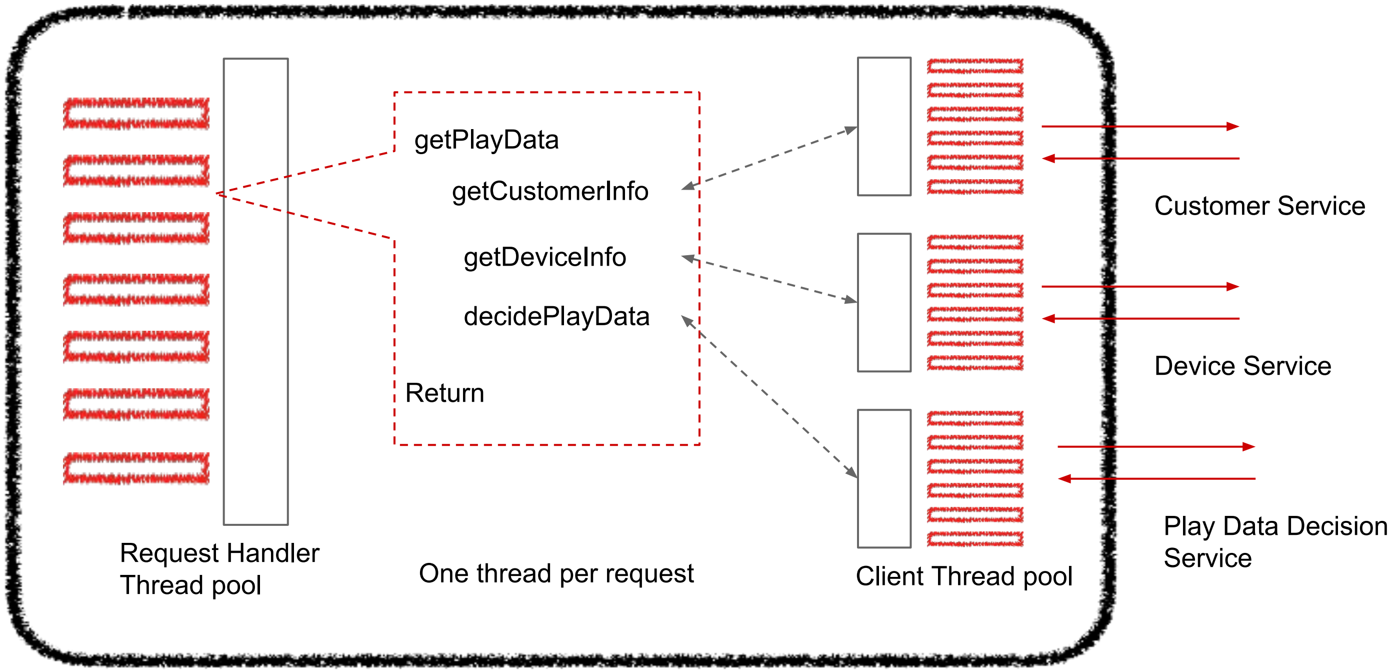
Data Architecture

```
PlayData getPlayData(string customerId, string titleId,
string deviceId){
    CustomerInfo custInfo = getCustomerInfo(customerId);
    DeviceInfo deviceInfo = getDeviceInfo(deviceId);
    PlayData playdata = decidePlayData(custInfo,
deviceInfo, titleId);
    return playdata;
}
```

Typical Synchronous Architecture



Typical Synchronous Architecture



getPlayData

getCustomerInfo

getDeviceInfo

decidePlayData

Return

Request Handler
Thread pool

One thread per request

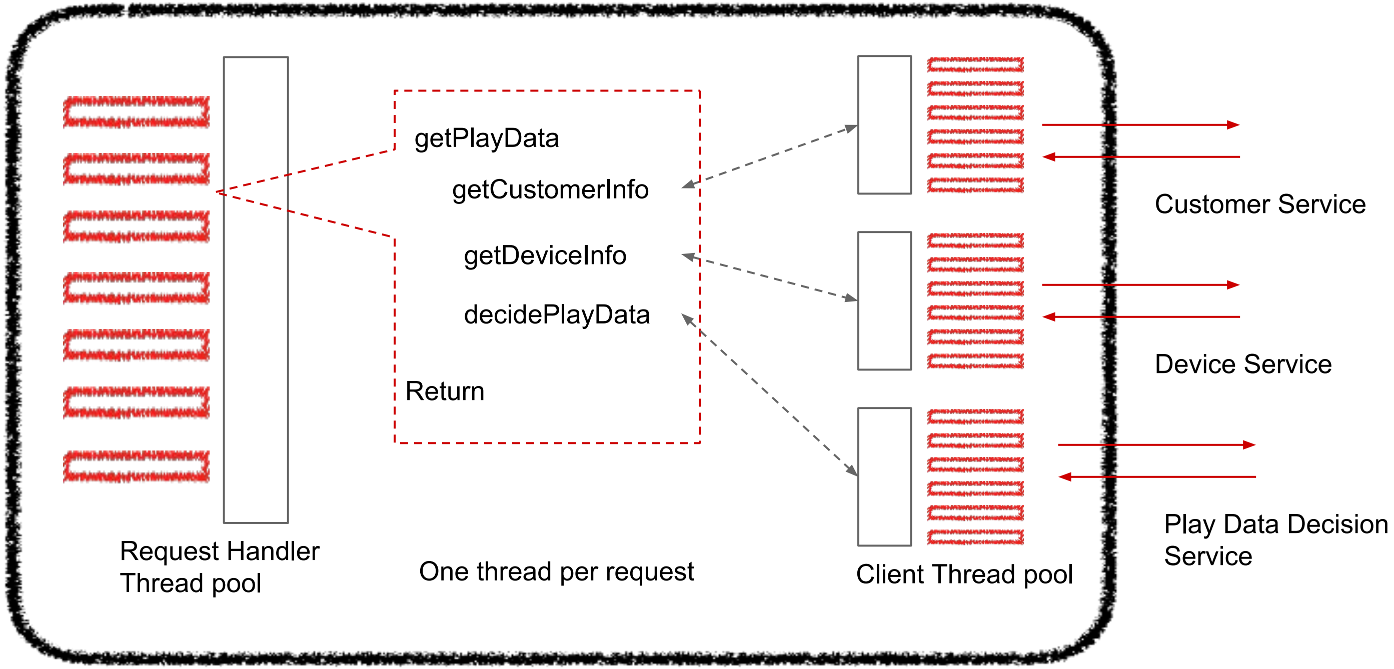
Client Thread pool

Customer Service

Device Service

Play Data Decision
Service

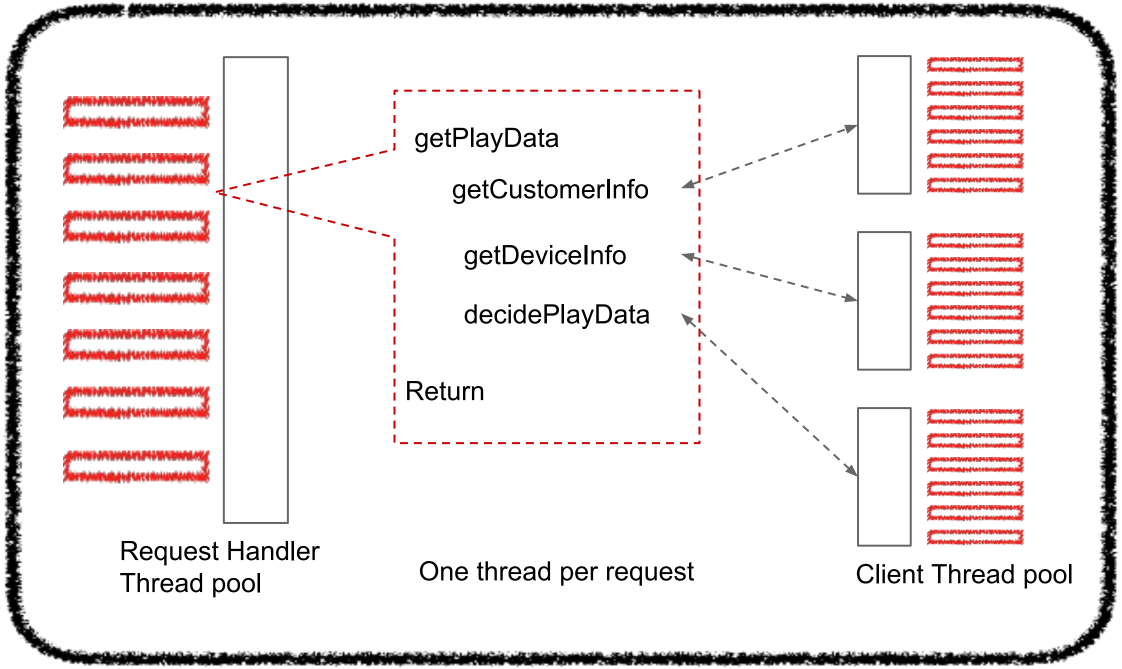
Typical Synchronous Architecture



Blocking Request Handler

Blocking Client I/O

Typical Synchronous Architecture



Blocking Request Handler

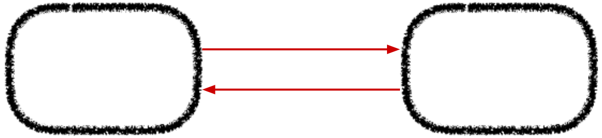
Blocking Client I/O

➕ Works for Simple Request/Response

➕ Works for Limited Clients

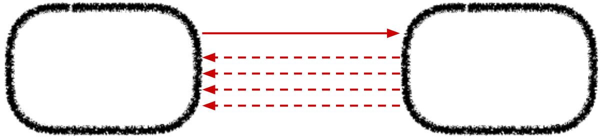
Beyond Request/Response

One Request - One Response



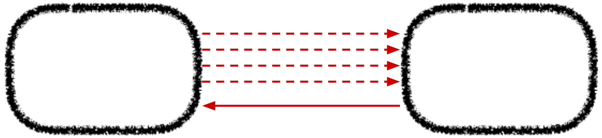
Request Play-data for Title X
Receive Play-data for Title X

One Request - Stream Response



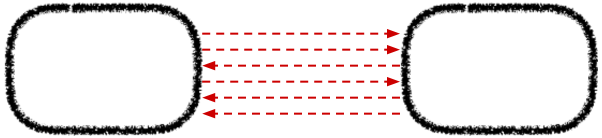
Request Play-data for Titles X,Y,Z
Receive Play-data for Title X
Receive Play-data for Title Y
Receive Play-data for Title Z

Stream Request - One Response



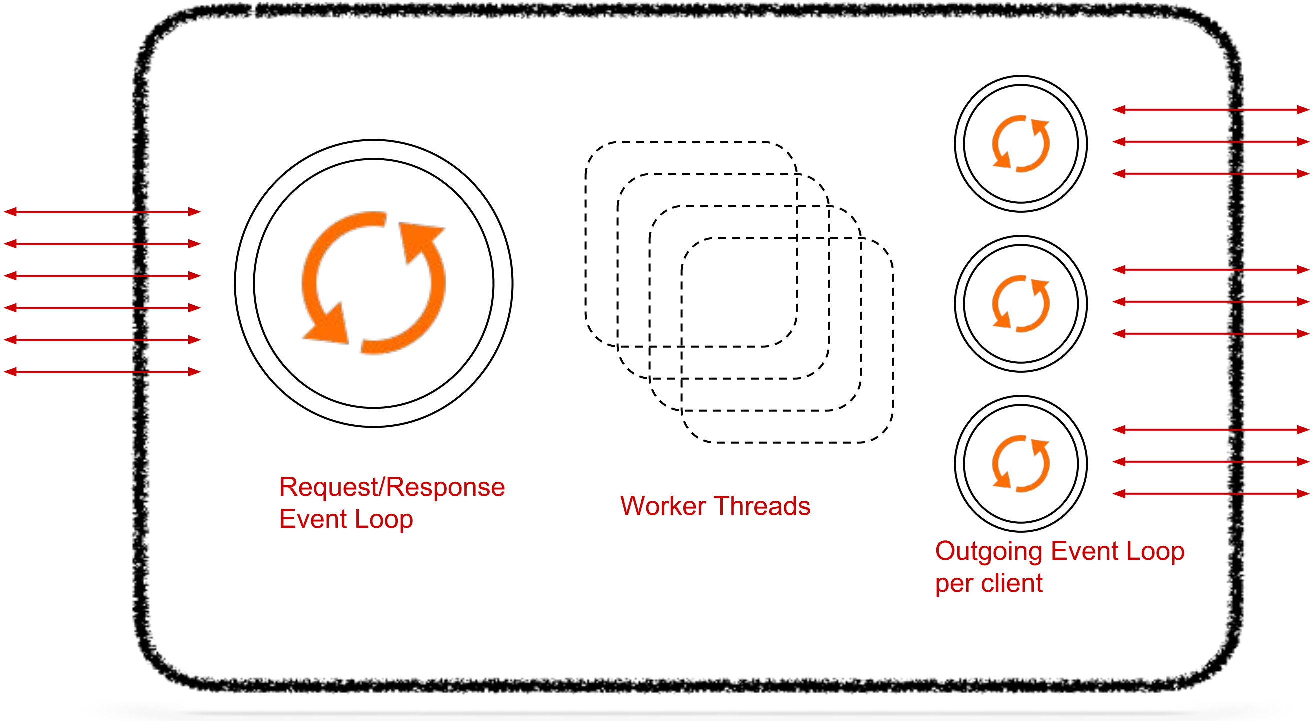
Request Play-data for Title X
Request Play-data for Title Y
Request Play-data for Title Z
Receive Play-data for Titles X,Y,Z

Stream Request - Stream Response



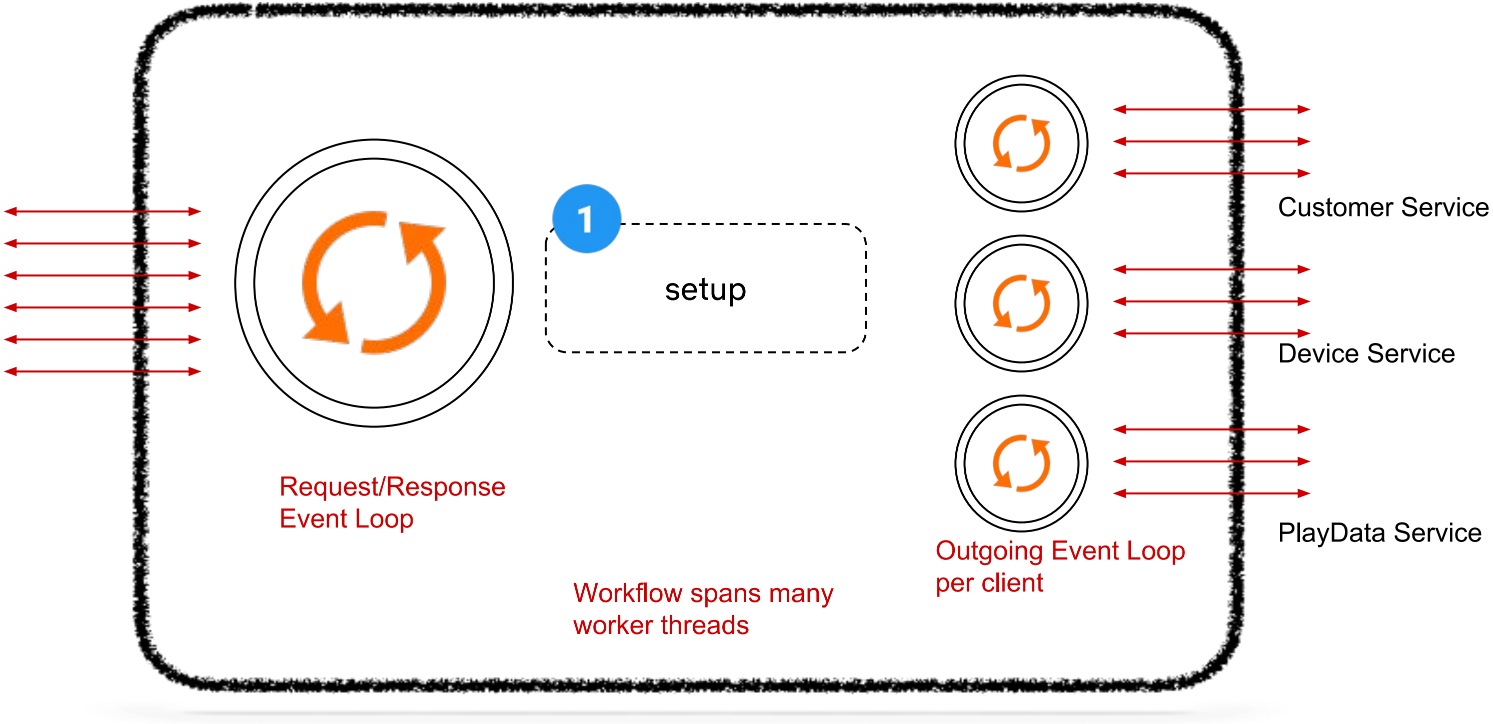
Request Play-data for Title X
Request Play-data for Title Y
Receive Play-data for Title X
Get Play-data for Title Z
Receive Play-data for Title Y
Receive Play-data for Title Z

Asynchronous Architecture

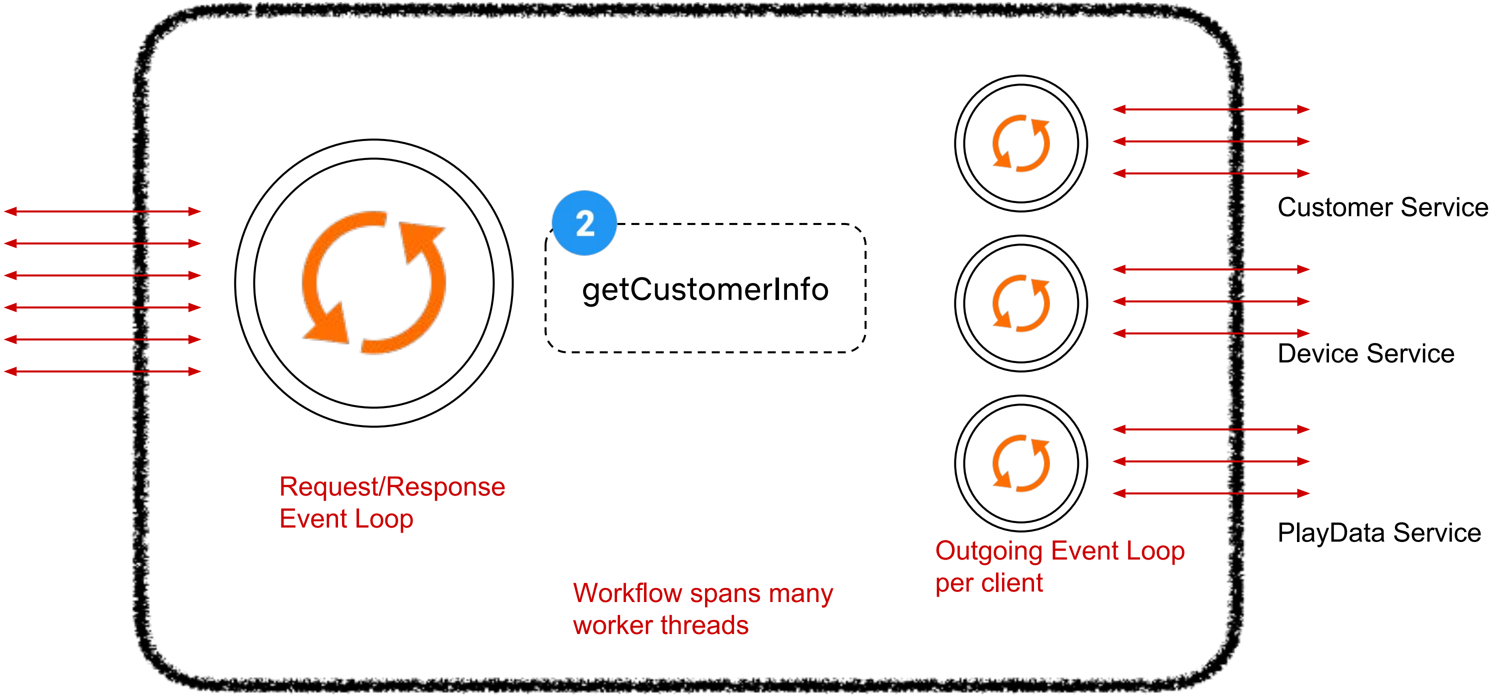


```
PlayData getPlayData(string customerId, string titleId,
string deviceId){
   Zip(getCustomerInfo(customerId),
        getDeviceInfo(deviceId),
        (custInfo, deviceInfo) ->
          return decidePlayData(custInfo, deviceInfo,
       titleId)
       );
}
```

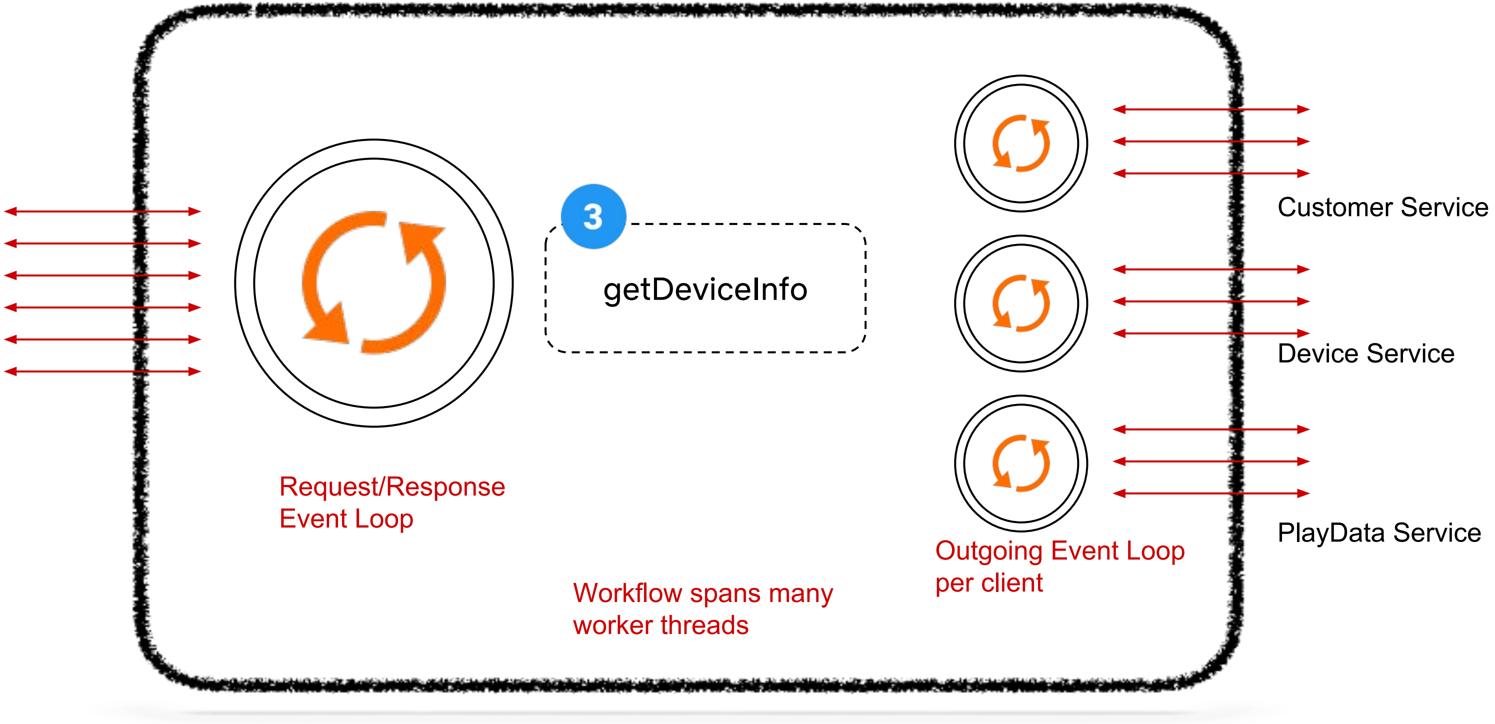
Asynchronous Architecture



Asynchronous Architecture



Asynchronous Architecture



3

getDeviceInfo

Customer Service

Device Service

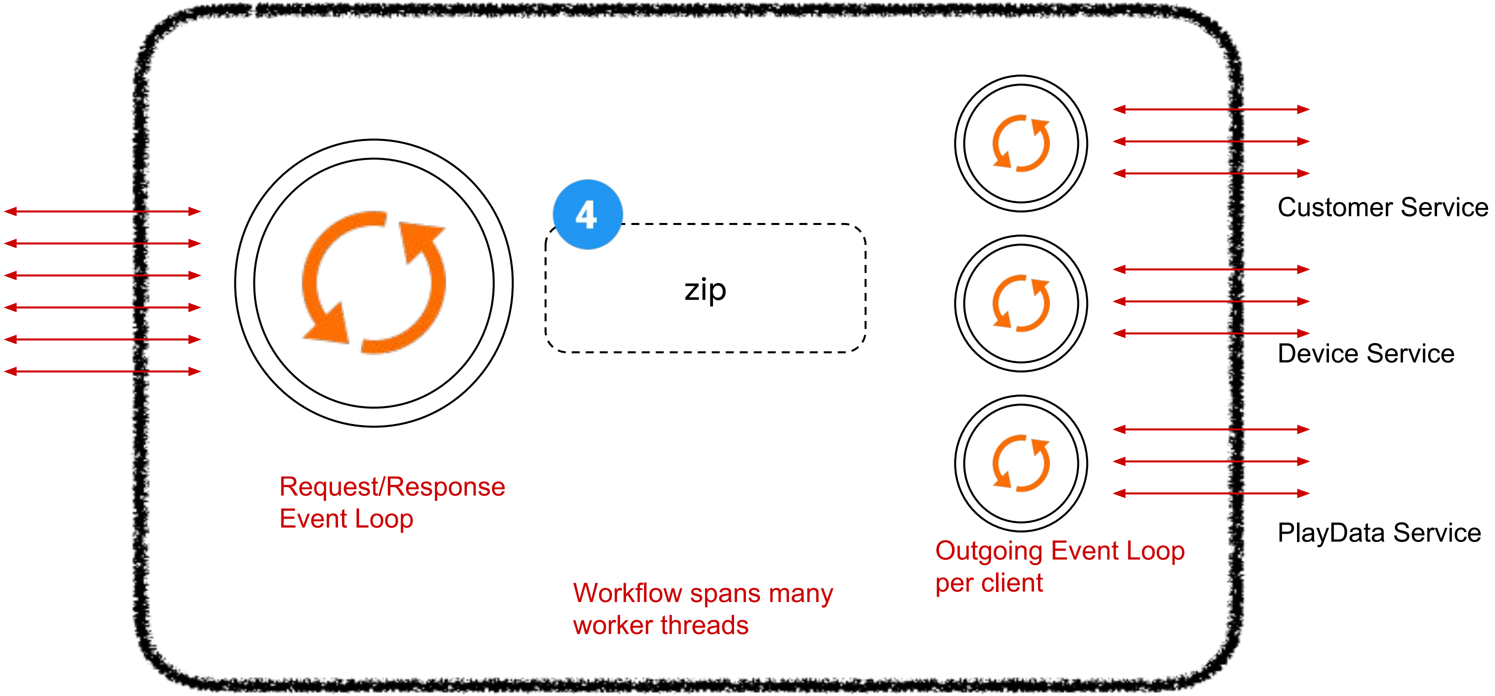
PlayData Service

Request/Response Event Loop

Workflow spans many worker threads

Outgoing Event Loop per client

Asynchronous Architecture

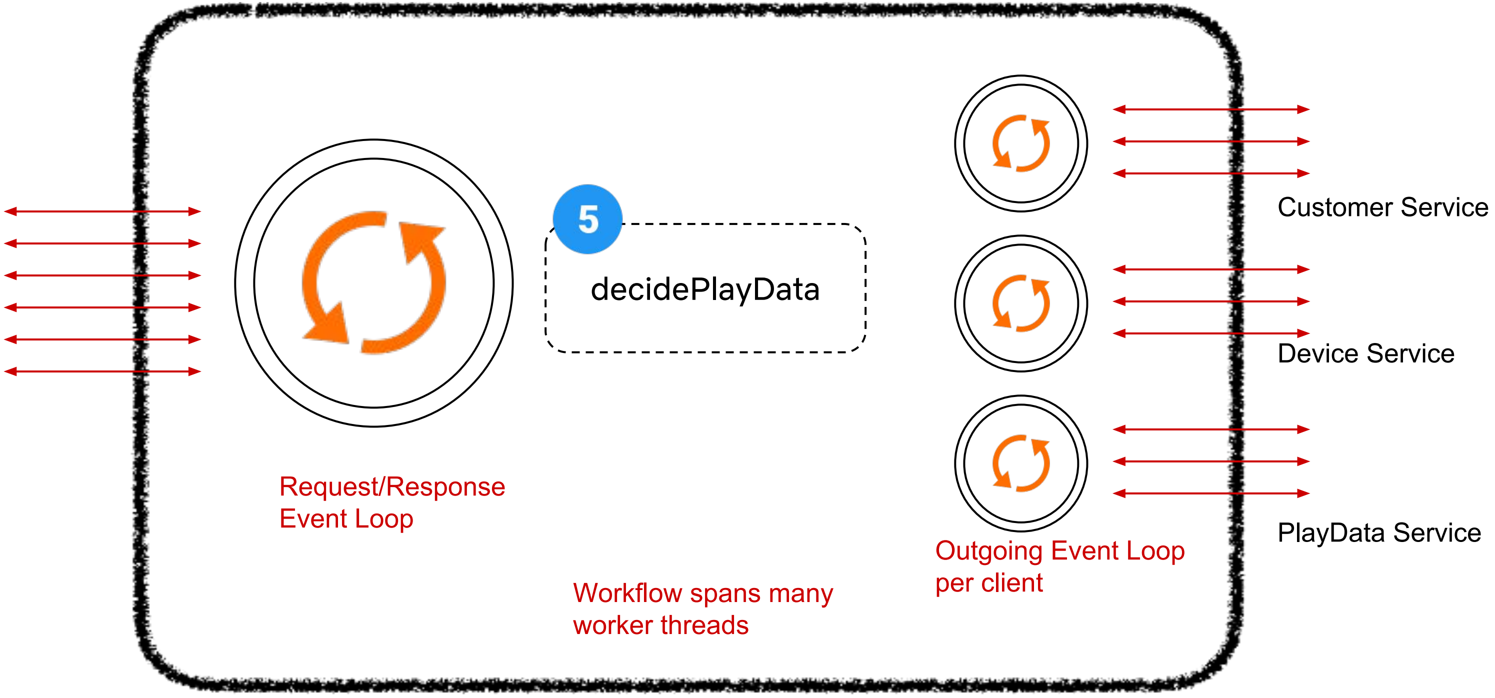


Request/Response Event Loop

Workflow spans many worker threads

Outgoing Event Loop per client

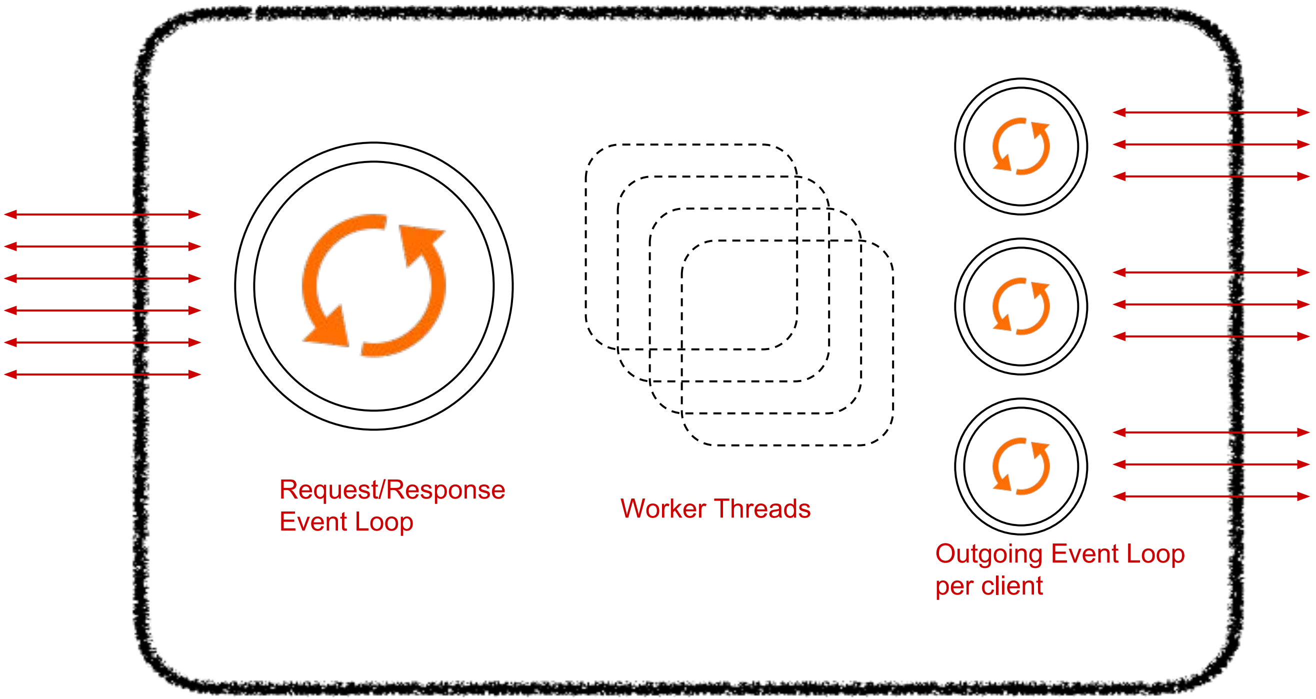
Asynchronous Architecture



Workflow spans multiple threads

- All context is passed as messages from one processing unit to another.
- If we need to follow and reason about a request, we need to build tools to capture and reassemble the order of execution units
- None of the calls can block

Asynchronous Architecture



Request/Response Event Loop

Worker Threads

Outgoing Event Loop per client

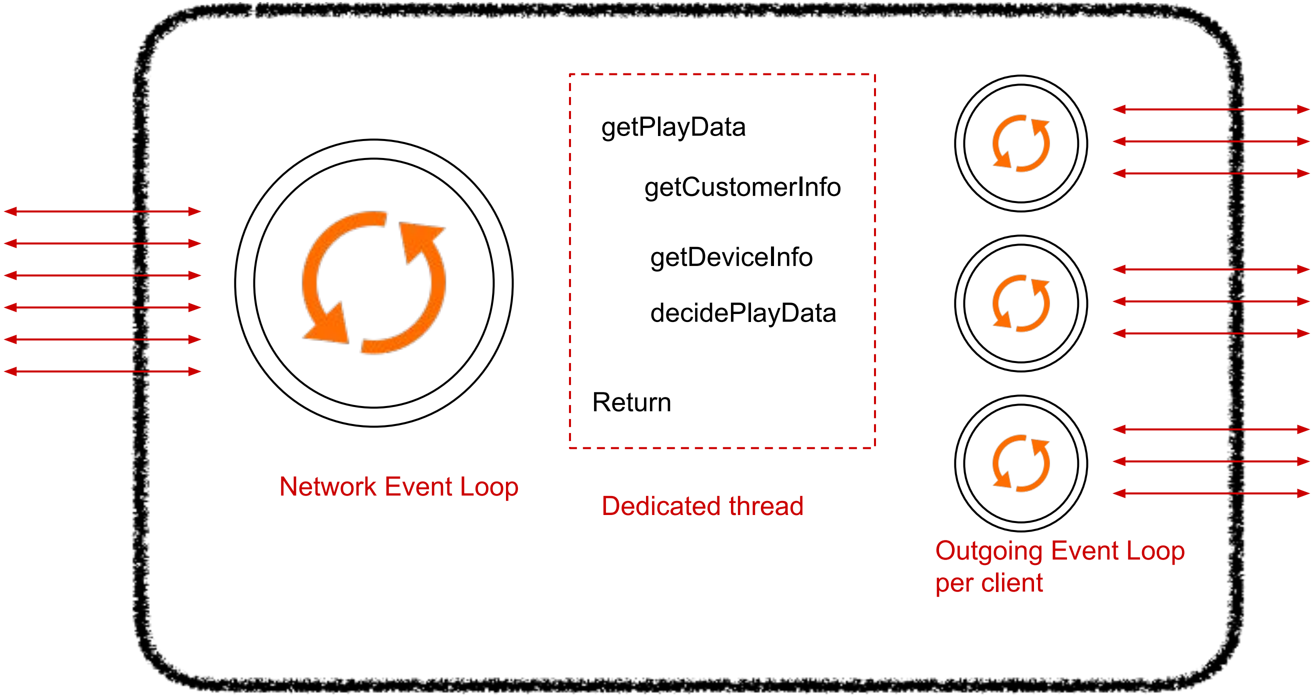
Asynchronous Request Handler

Non-Blocking I/O

Synchrony

Ask: Do you really have a need beyond Request/Response?

Synchronous Execution + Asynchronous I/O



getPlayData

getCustomerInfo

getDeviceInfo

decidePlayData

Return

Network Event Loop

Dedicated thread

Outgoing Event Loop per client

Blocking Request Handler

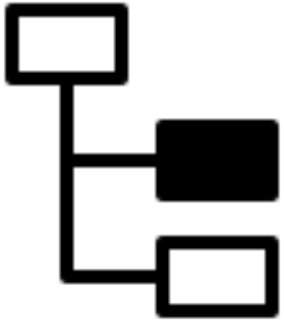
Non-Blocking I/O

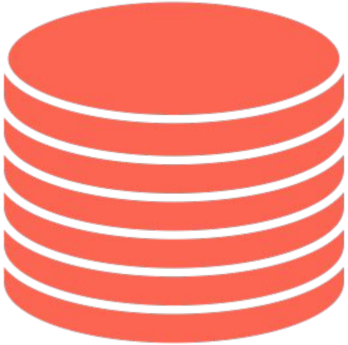
Current Architecture

Type 1 Decision: Synchronous vs Asynchronous

If most of your APIs fit the Request/Response pattern, consider a synchronous request handler, with nonblocking I/O

Three Type 1 Decisions to Consider





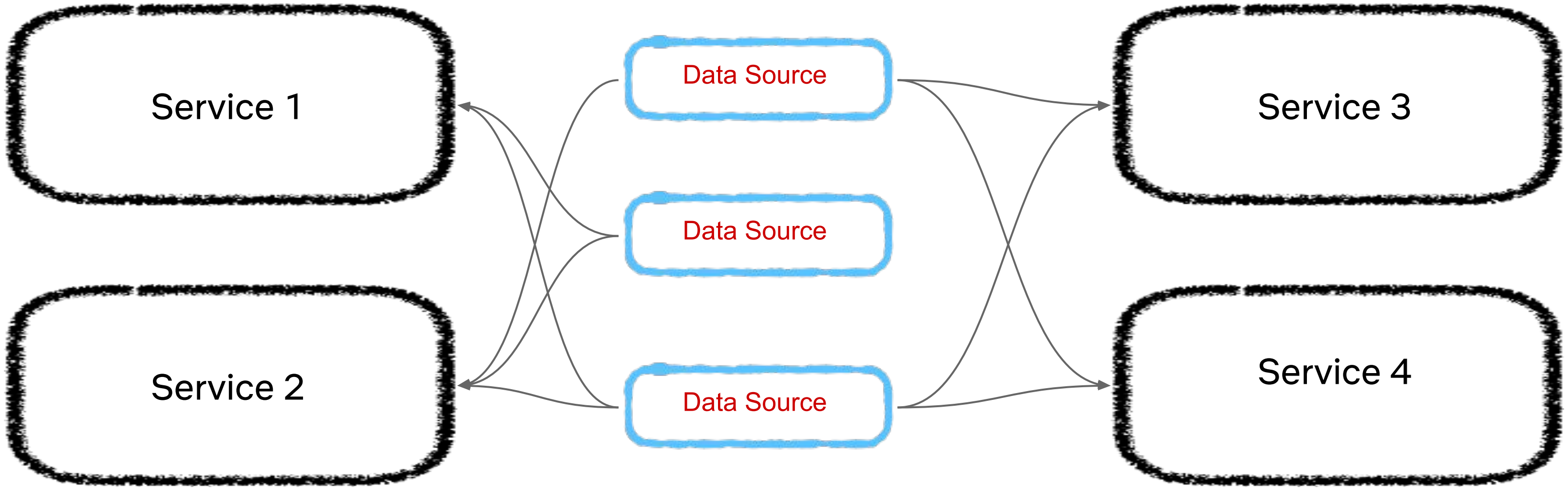
✓ Appropriate Coupling

✓ Synchronous vs Asynchronous

Data Architecture

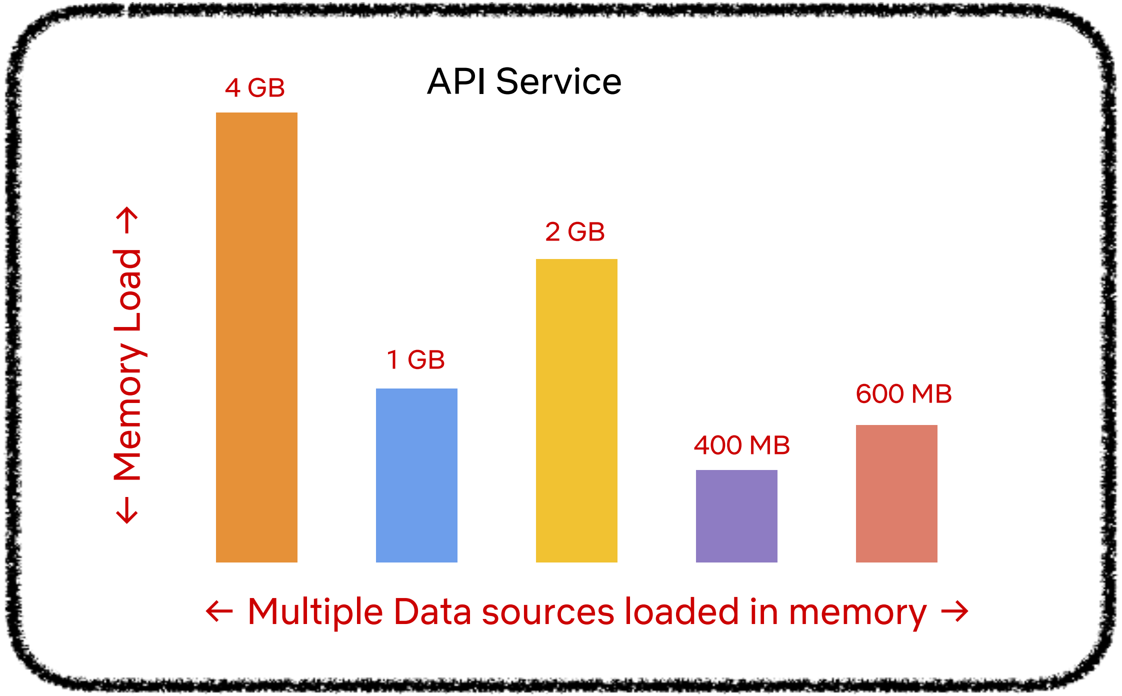
Without an intentional Data Architecture, Data becomes its own monolith

What a Data Monolith looks like



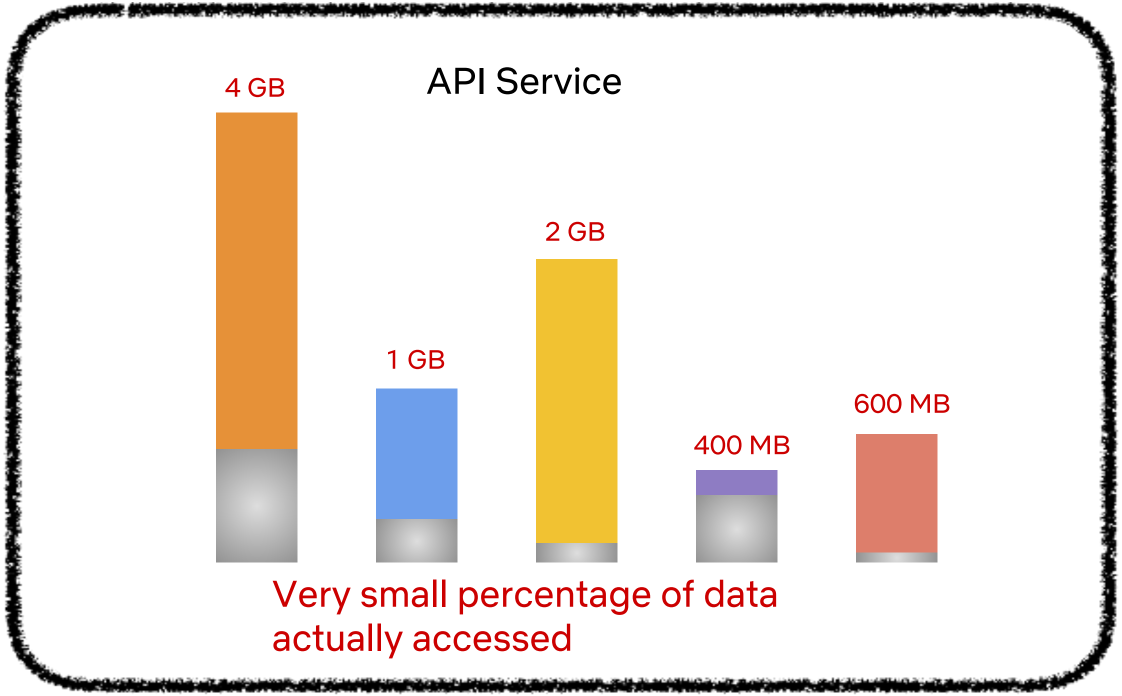
Previous Architecture

What a Data Monolith looks like



Previous Architecture

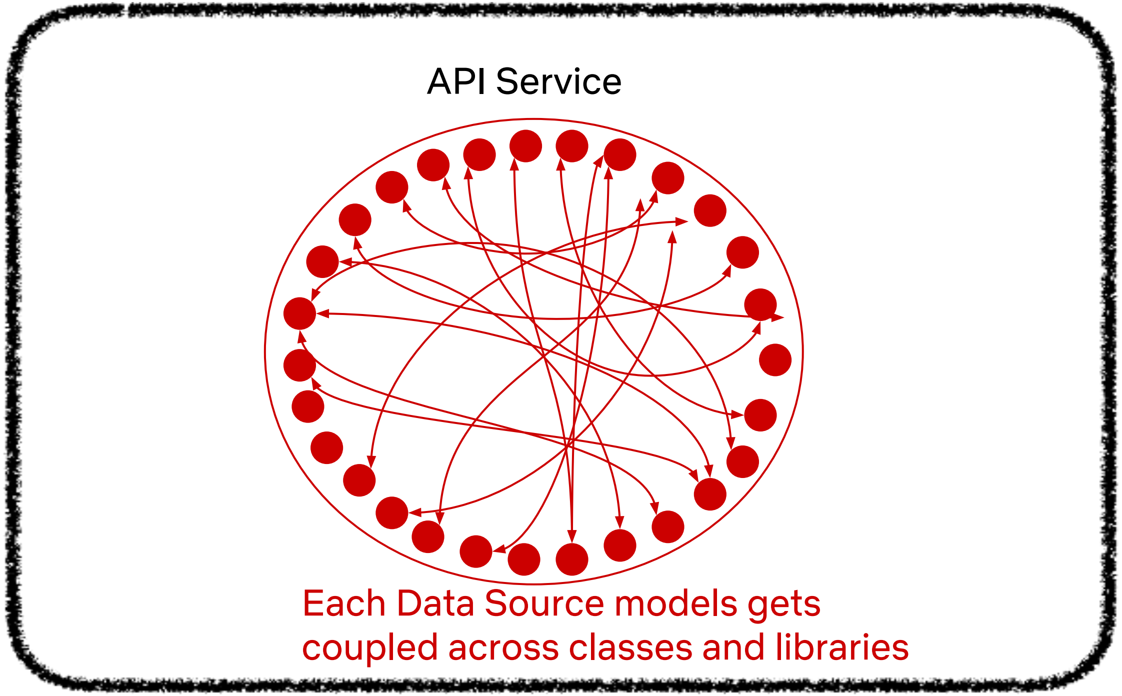
What a Data Monolith looks like



Very small percentage of data actually accessed

Previous Architecture

What a Data Monolith looks like



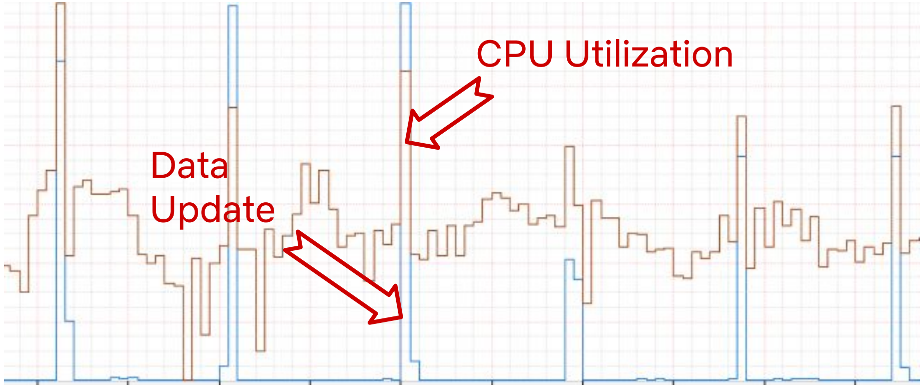
API Service

Each Data Source models gets
coupled across classes and libraries

Previous Architecture

What a Data Monolith looks like

API Service



CPU Utilization

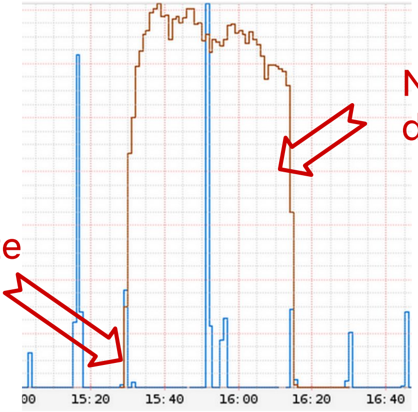
Data Update

Unpredictable Performance Characteristics

Previous Architecture

What a Data Monolith looks like

API Service



Data Update

Netflix was down

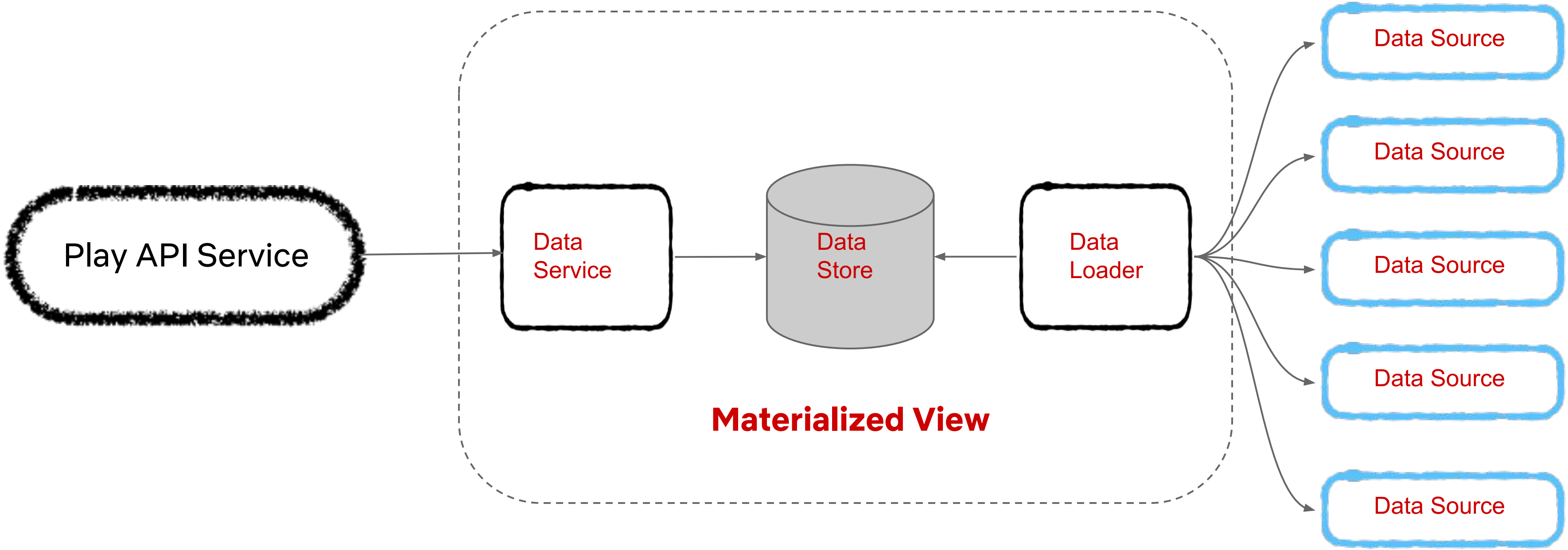
Potential to bring down the service

Previous Architecture

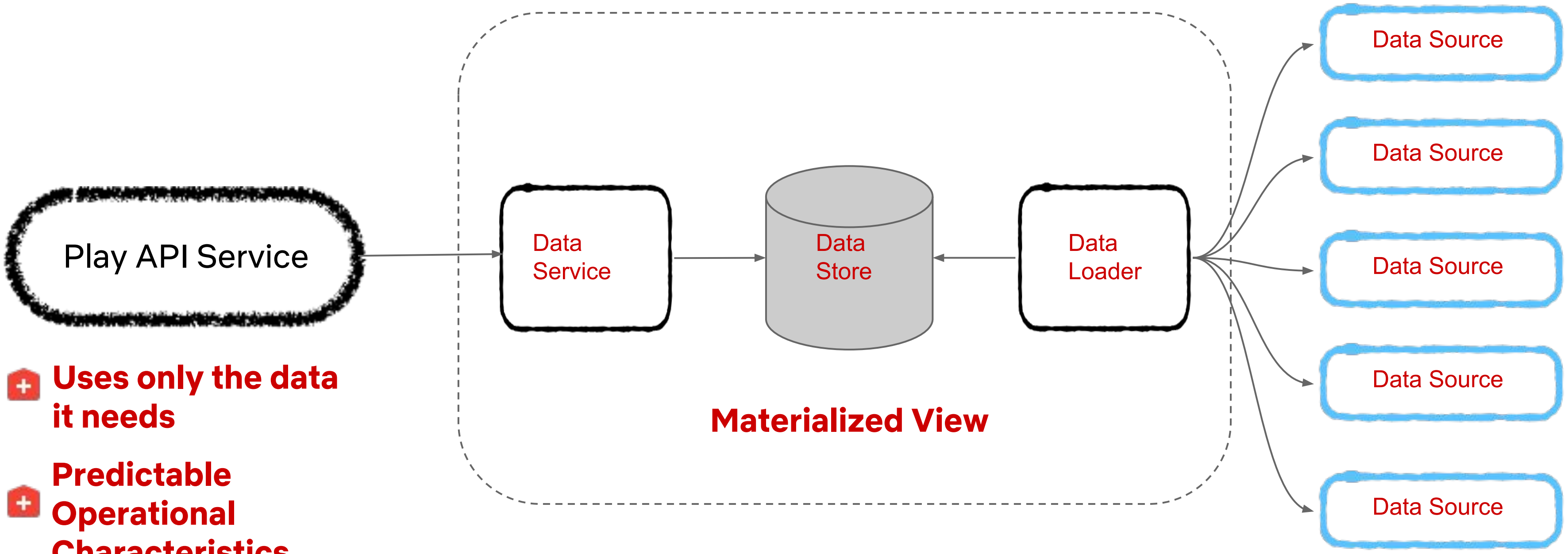
"All problems in computer science can be solved by another level of indirection."

David Wheeler

(World's first Comp Sci PhD)



Current Architecture



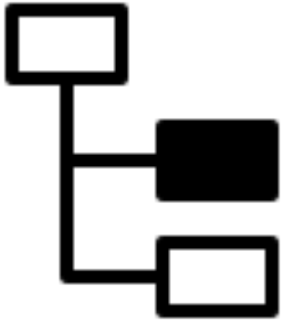
- **Uses only the data it needs**
- **Predictable Operational Characteristics**
- **Reduced Dependency chain**

Current Architecture

Type 1 Decision: Data Architecture

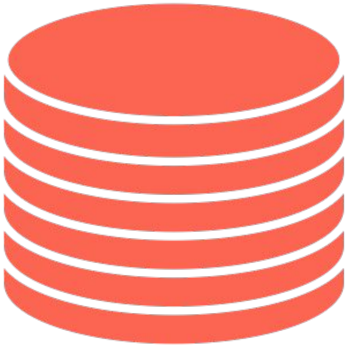
Isolate Data from the Service. At the very least, ensure that data sources are accessed via a layer of abstraction, so that it leaves room for extension later

Three Type 1 Decisions to Consider



✓ Appropriate Coupling

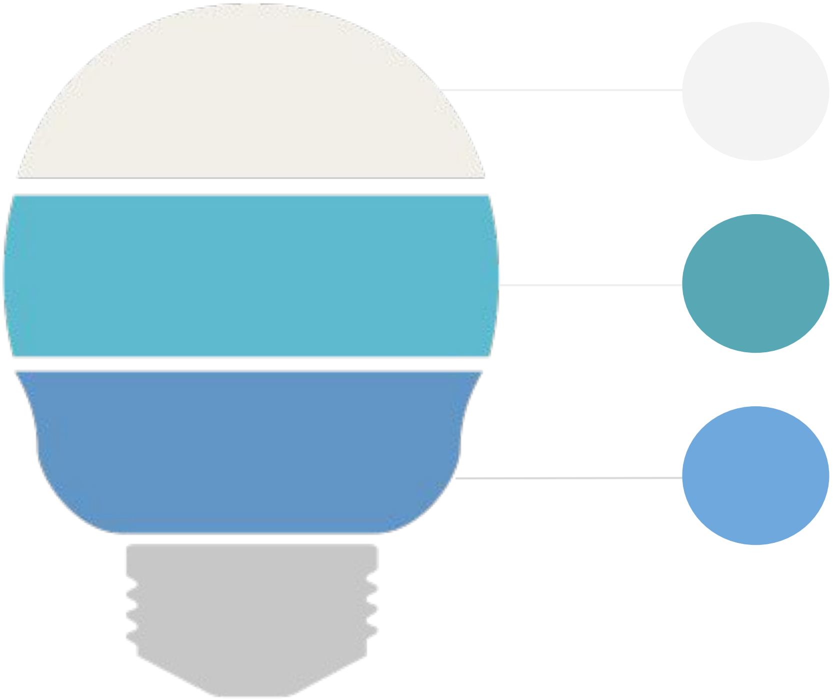
✓ Synchrony



✓ Data Architecture

For Type 2 decisions, choose a path, experiment and iterate

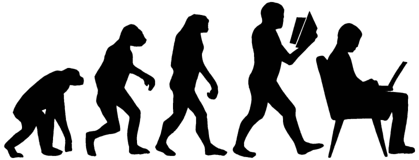
Guiding Principle: **Identify your Type 1 and Type 2 decisions; Spend 80% of your time debating and aligning on Type 1 Decisions**



Identity

Type 1/2 Decisions

Evolvability

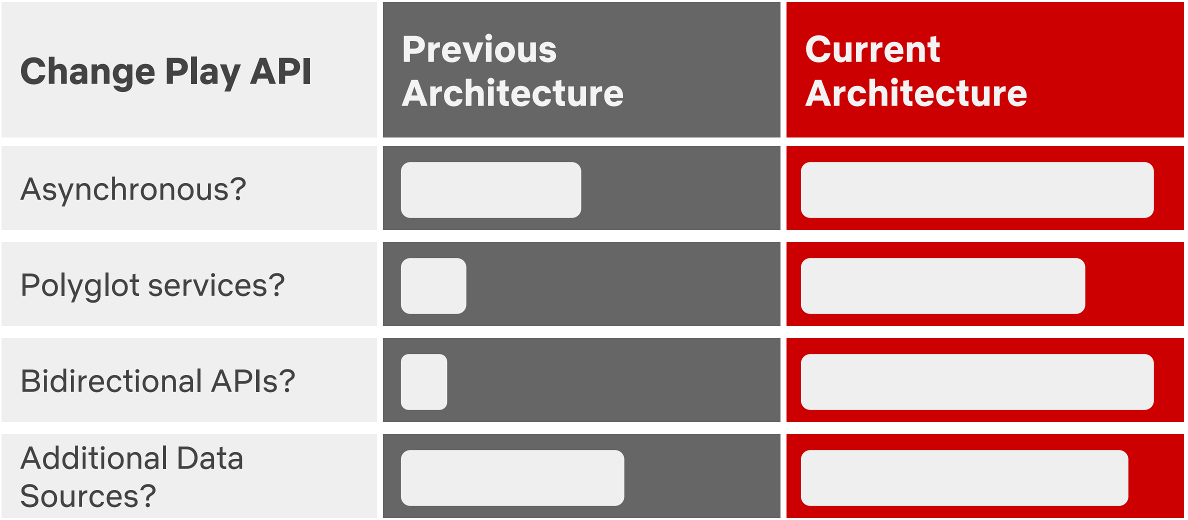


An Evolutionary Architecture supports guided and incremental change as first principle among multiple dimensions

- ThoughtWorks

Choosing a microservices architecture with appropriate coupling allows us to evolve across multiple dimensions

How evolvable are the Type 1 decisions



Known Unknowns

Change Play API	**Previous Architecture**	**Current Architecture**
Asynchronous?		
Polyglot services?		
Bidirectional APIs?		
Additional Data Sources?		

Potential Type 1 decisions in the future?

Change Play API	Previous Architecture	Current Architecture
Containers?	□	?
Serverless?	□	?

And we fully expect that there will be Unknown Unknowns

As we evolve, how to ensure we are not breaking our original goals?

Use Fitness Functions to guide change

High Availability

Low Latency

Evolvability

Simplicity

Scalable

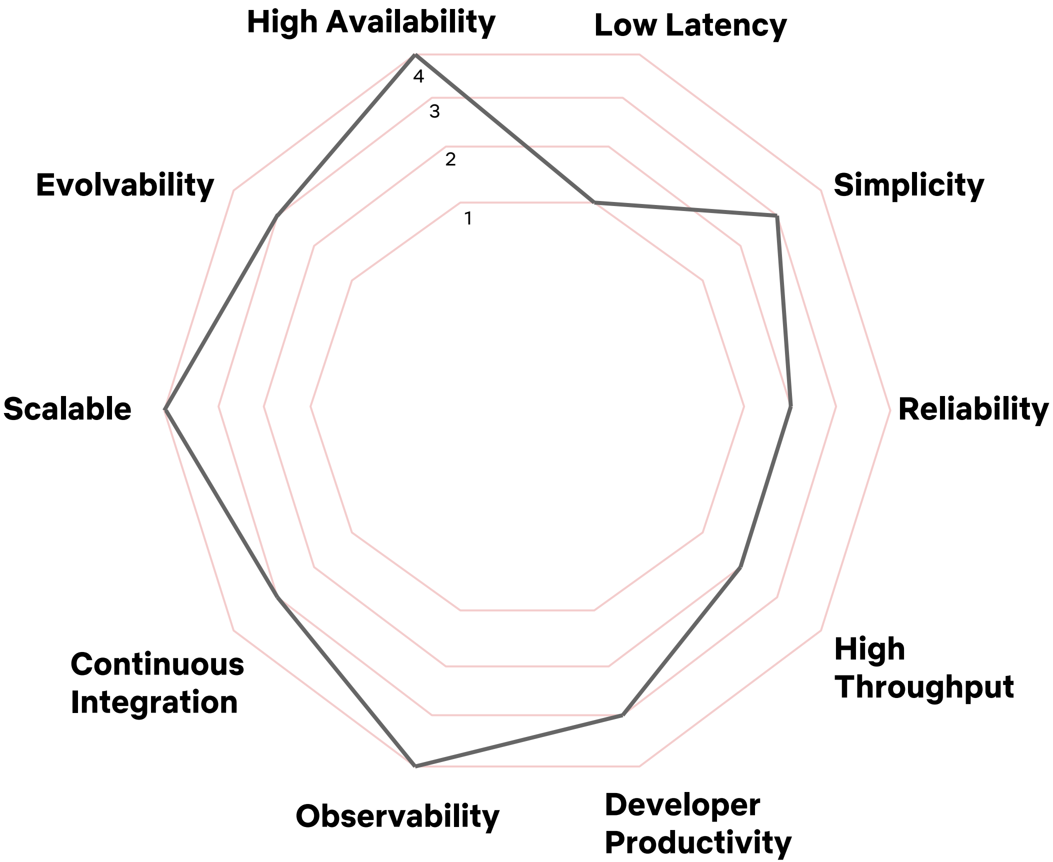
Reliability

Continuous Integration

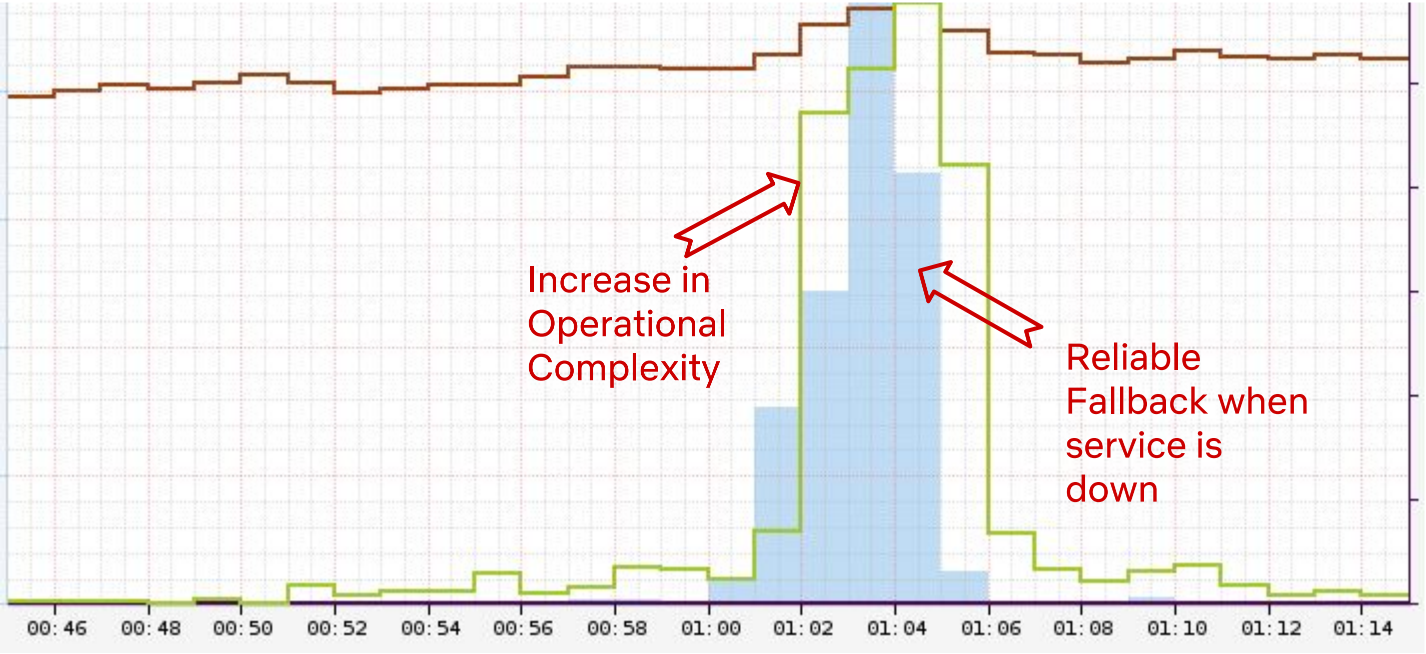
High Throughput

Observability

Developer Productivity


High Availability
Low Latency
Simplicity
Reliability
High Throughput
Developer Productivity
Observability
Continuous Integration
Scalable
Evolvability
4
3
2
1

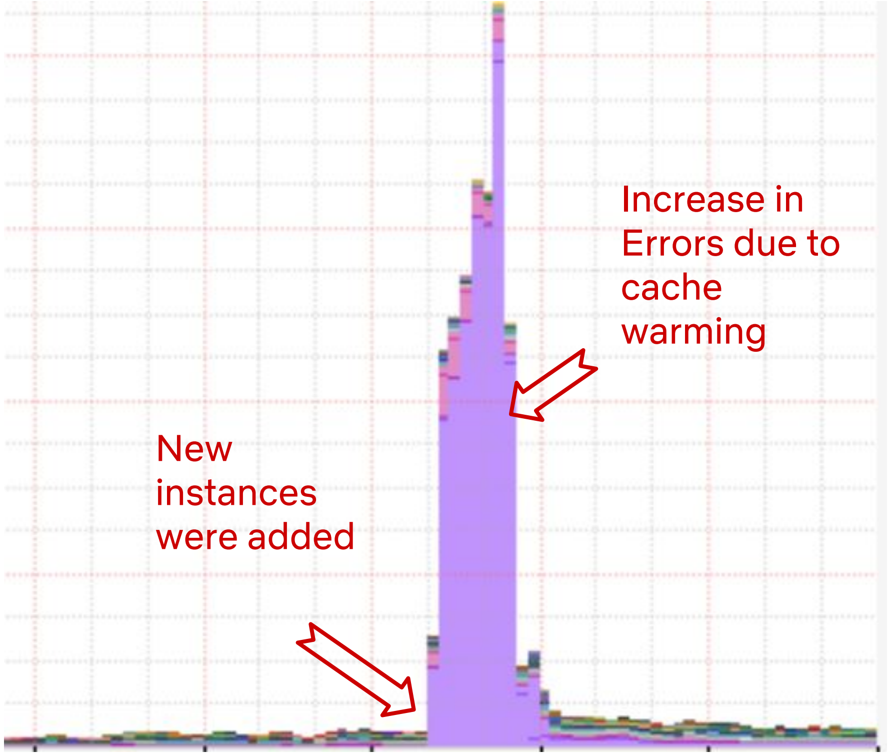
Why Simplicity over Reliability?



Increase in Operational Complexity

Reliable Fallback when service is down

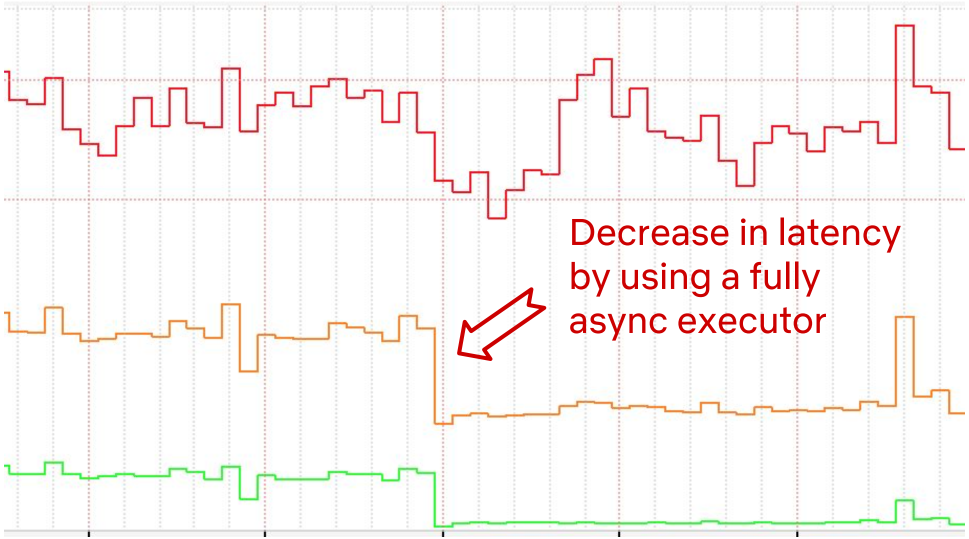
Why Scalability over Throughput?



New instances were added

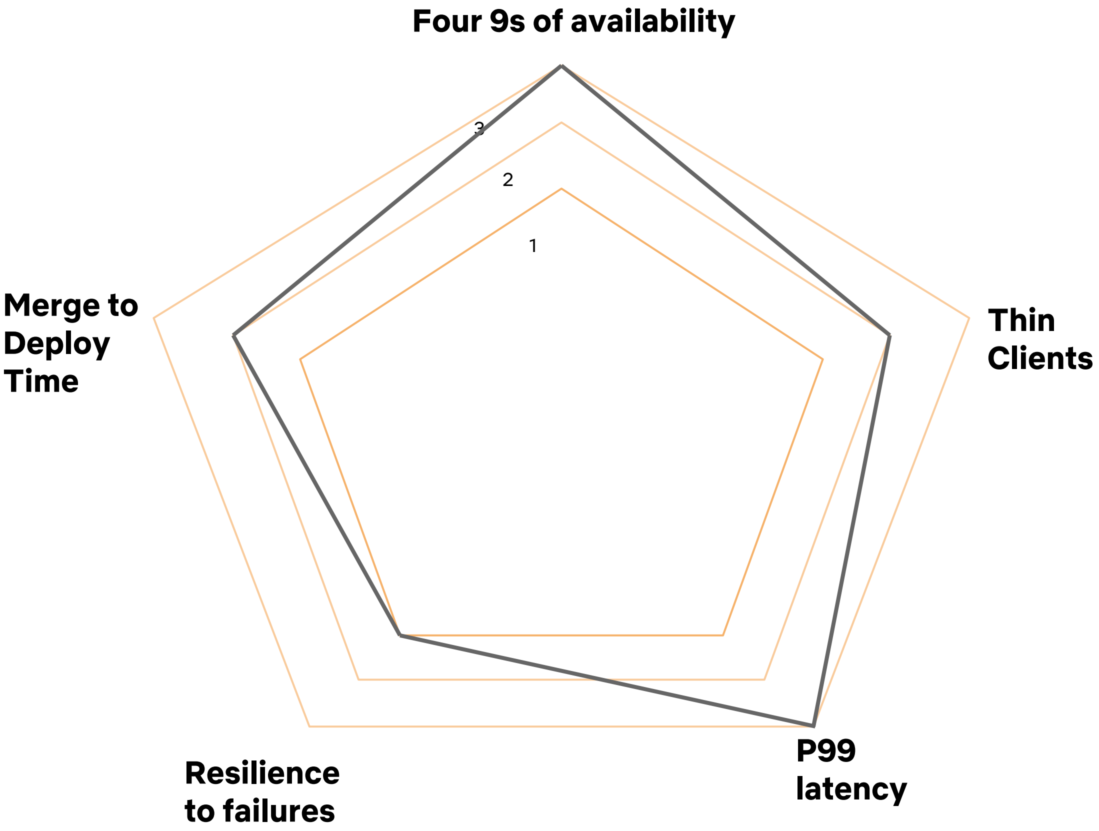
Increase in Errors due to cache warming

Why Observability over Latency?



Decrease in latency by using a fully async executor

Cost of Async: Loss in Observability


Four 9s of availability
3
2
1
Thin Clients
P99 latency
Resilience to failures
Merge to Deploy Time

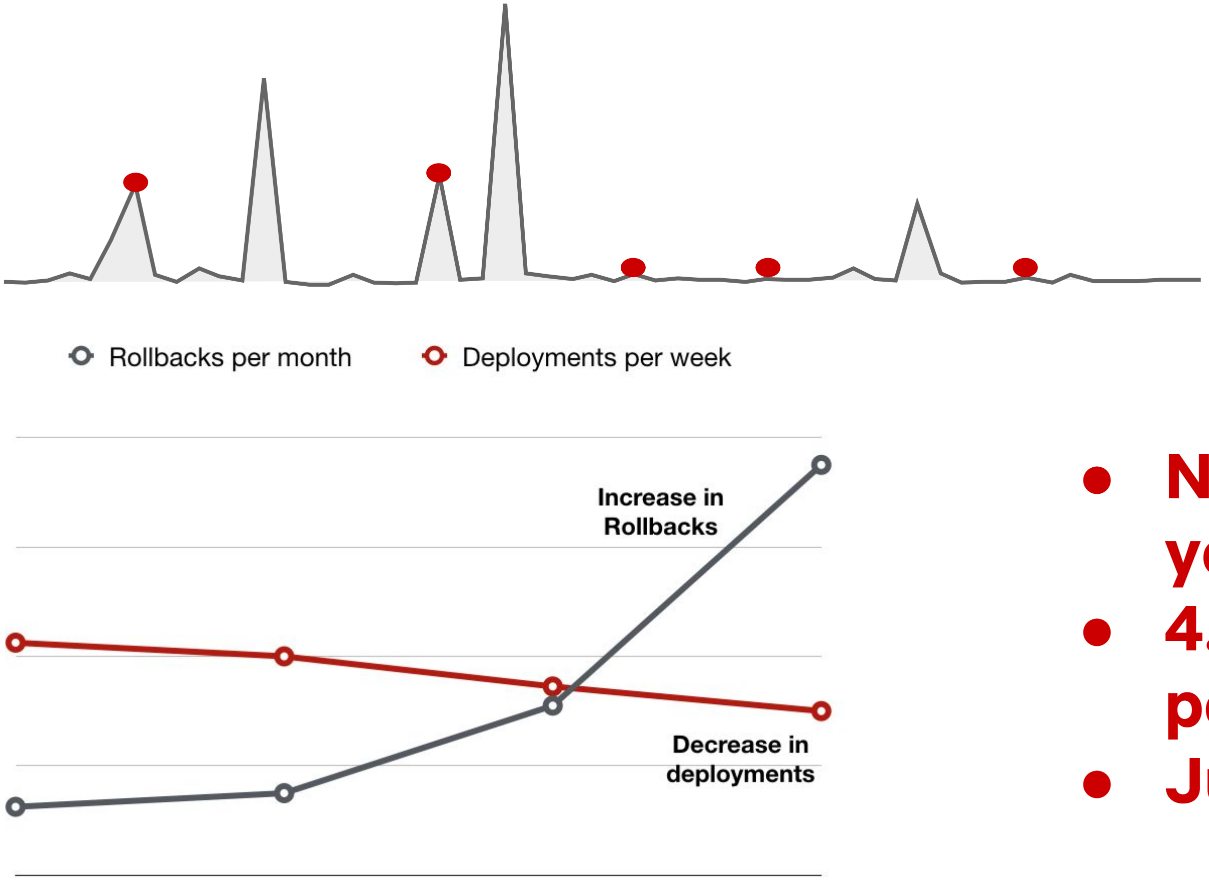
Guiding Principle: Define Fitness functions to act as your guide for architectural evolution

Previous Architecture

- Multiple Identities
- Operational Coupling
- Binary Coupling
- Synchronous communication
- Only Java
- Data Monolith

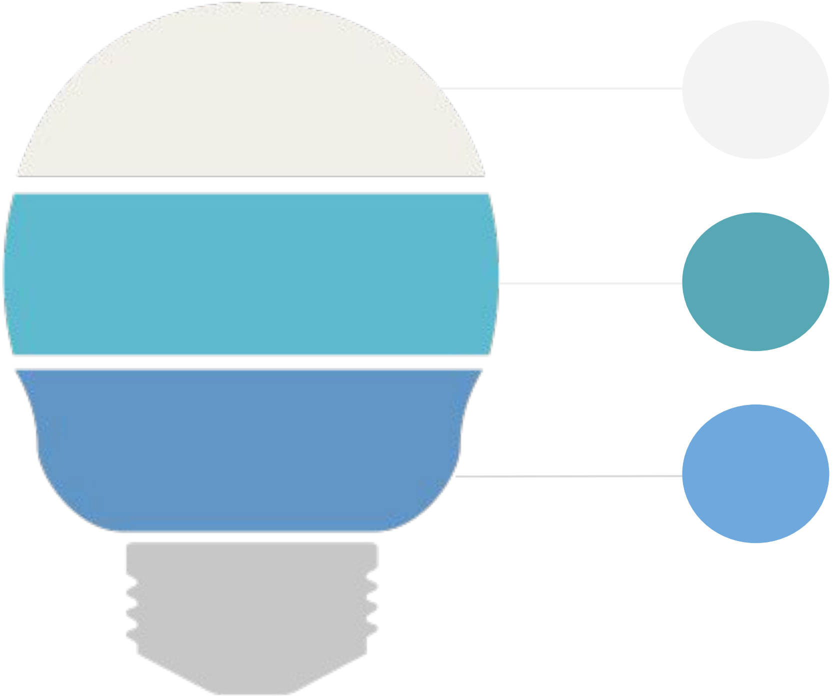
Current Architecture

- Singular Identities
- Operational Isolation
- No Binary Coupling
- Asynchronous communication
- Beyond Java
- Explicit Data Architecture
- Guided Fitness Functions



- No incidents in a year
- 4.5 deployments per week
- Just two rollbacks!

Build a Evolutionary Architecture



Identity

Type 1/2 Decisions

Evolvability