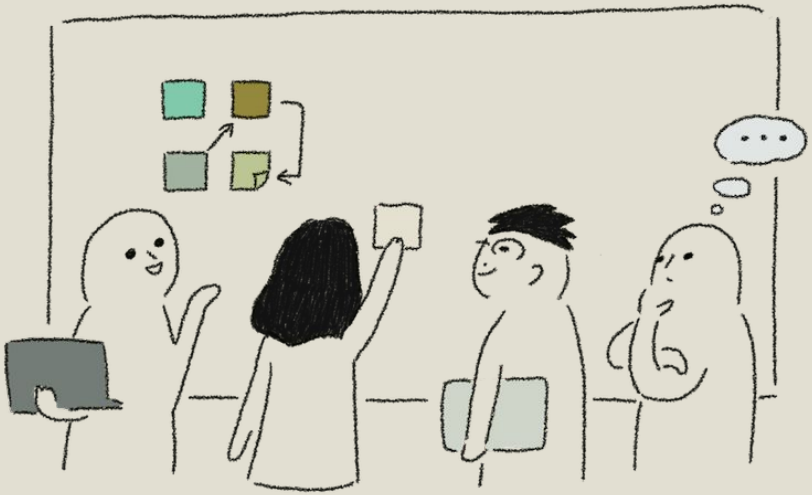
DATE 20250414

Session 7



backend

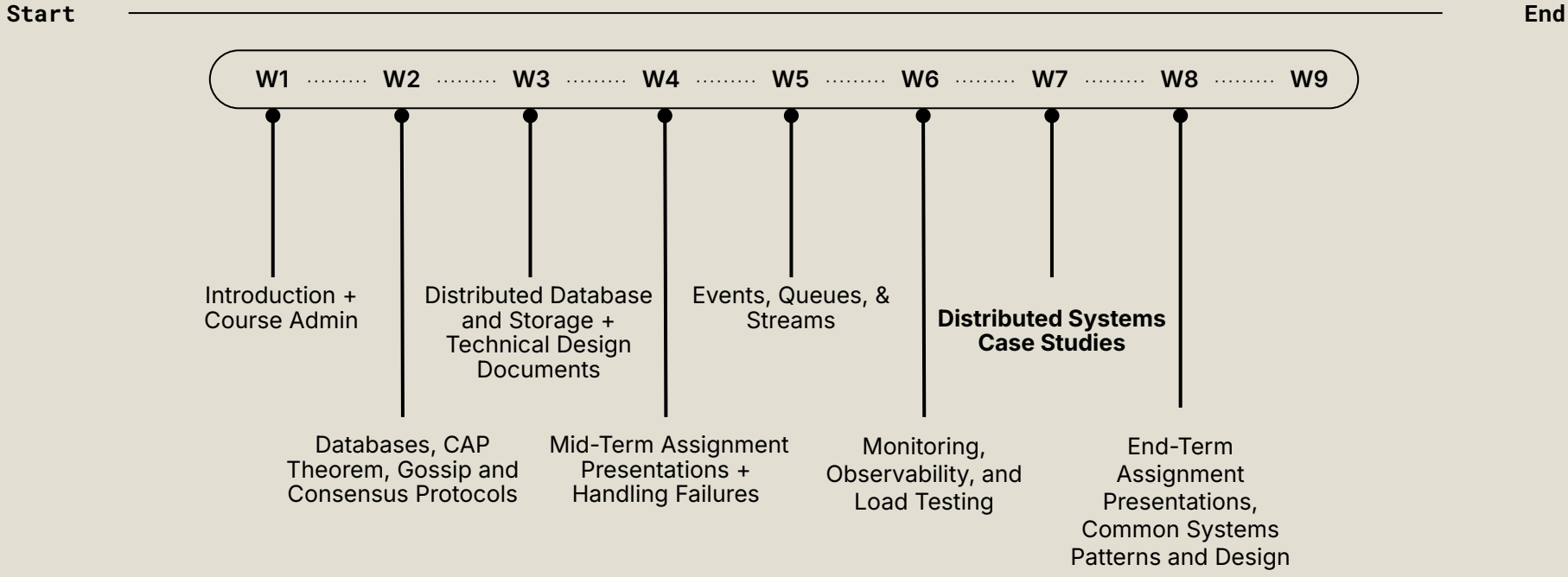
cohort #0 by open camp

#7's agenda

1. admin matters (if any)
2. microservices case study
3. load balancing case study
4. developer experience case study
5. w7 assignment

admin matters

Curriculum: https://opencamp-cc.github.io/backend-curriculum/

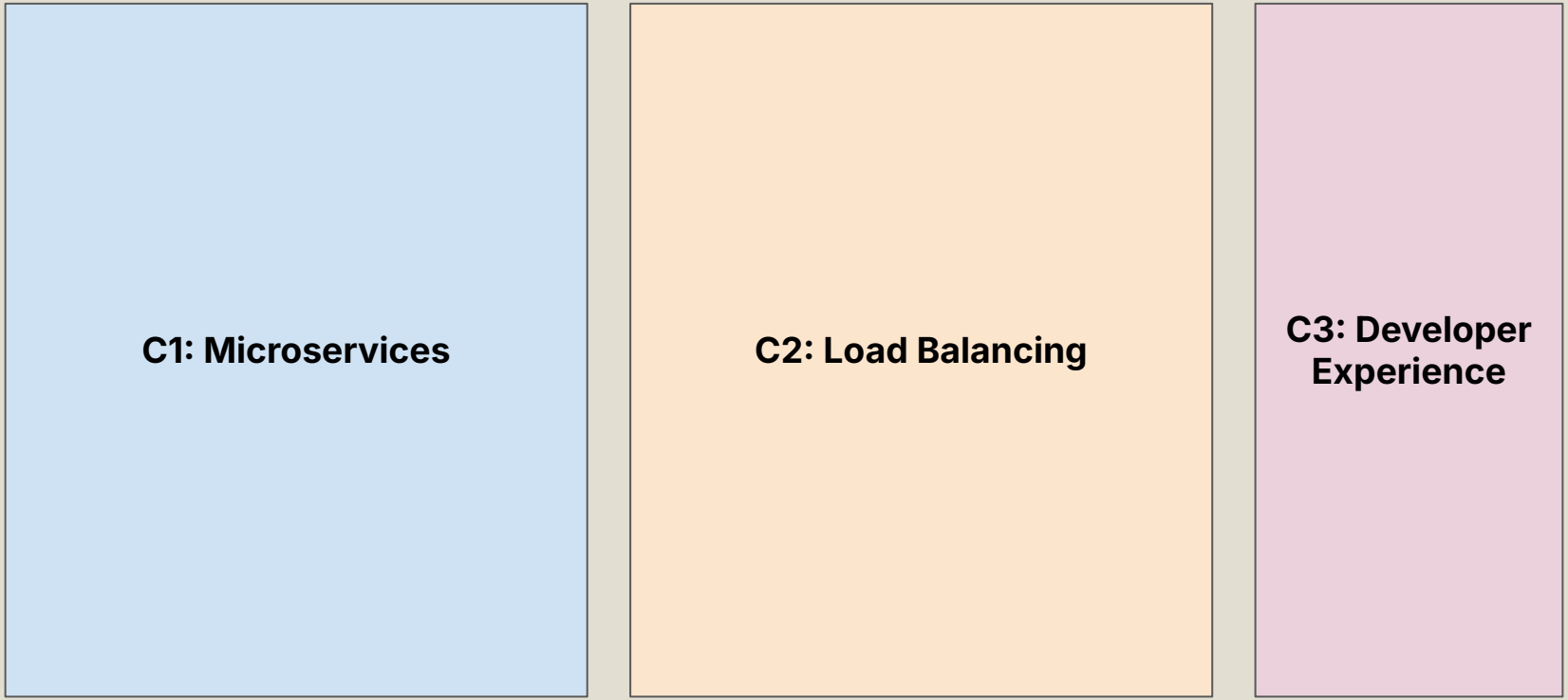


W5 + W6 Assignment

- Load Testing or Monitoring:
  - Load Testing w/ k6: 2 persons

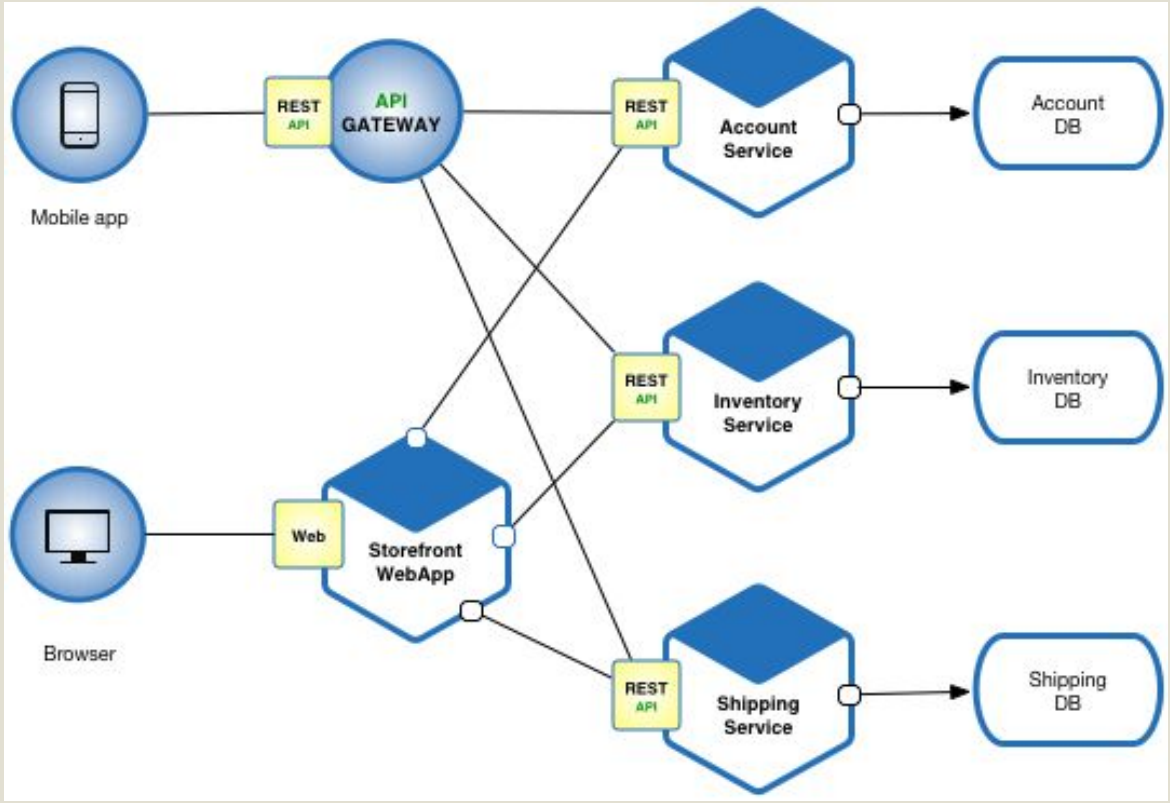
- Queues:
  - RabbitMQ: 2 persons
  - Kafka: 1 person

- Still catching up: 4 persons



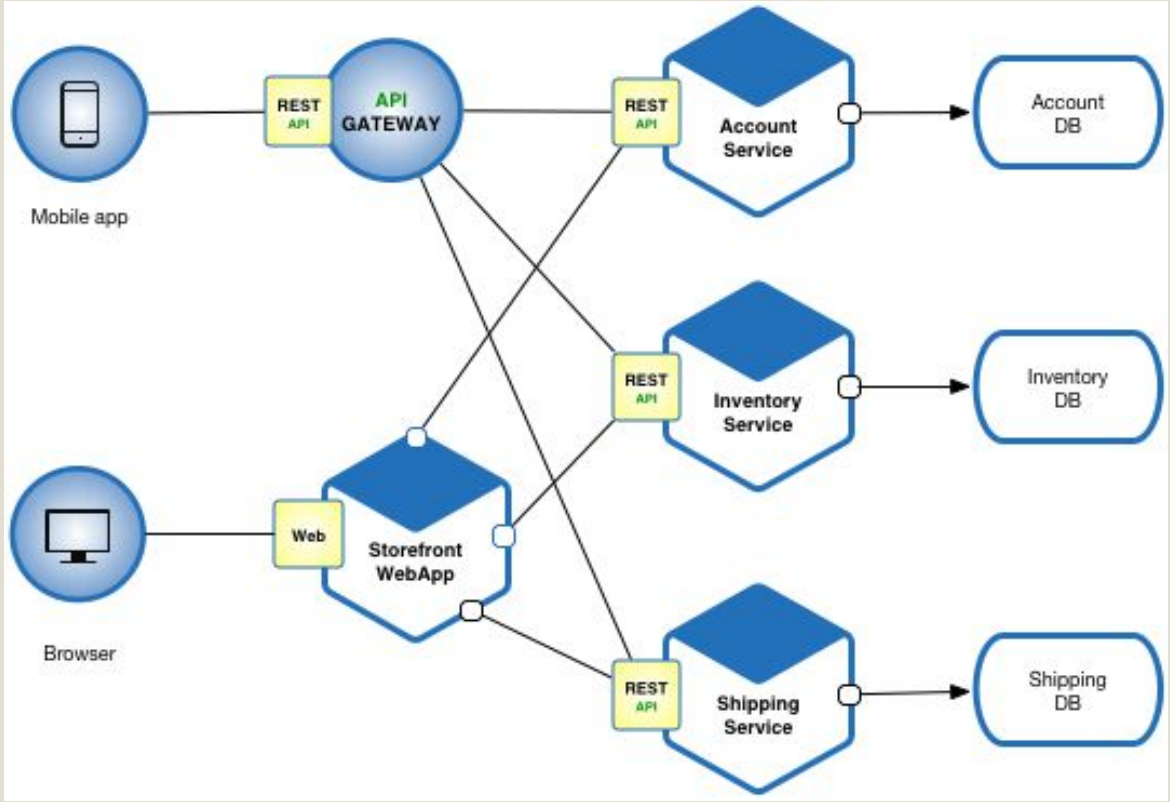
Three Case Studies Today

7.1 microservice case study



Microservice Design

https://microservices.io/patterns/microservices.html



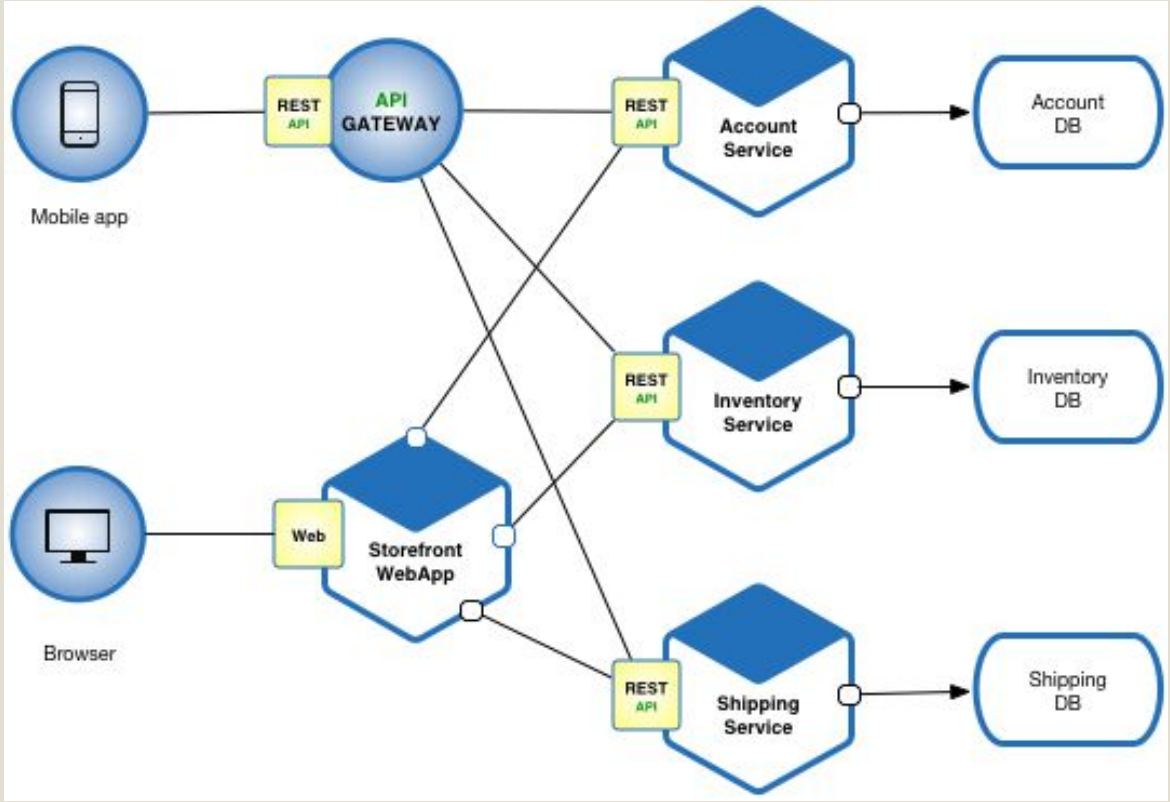
1. Services can be deployed and upgraded without affecting the whole system

2. Scale independent services based on workload

3. Can support different tech stacks instead of all in one language

Microservice Design: Some Benefits

https://microservices.io/patterns/microservices.html



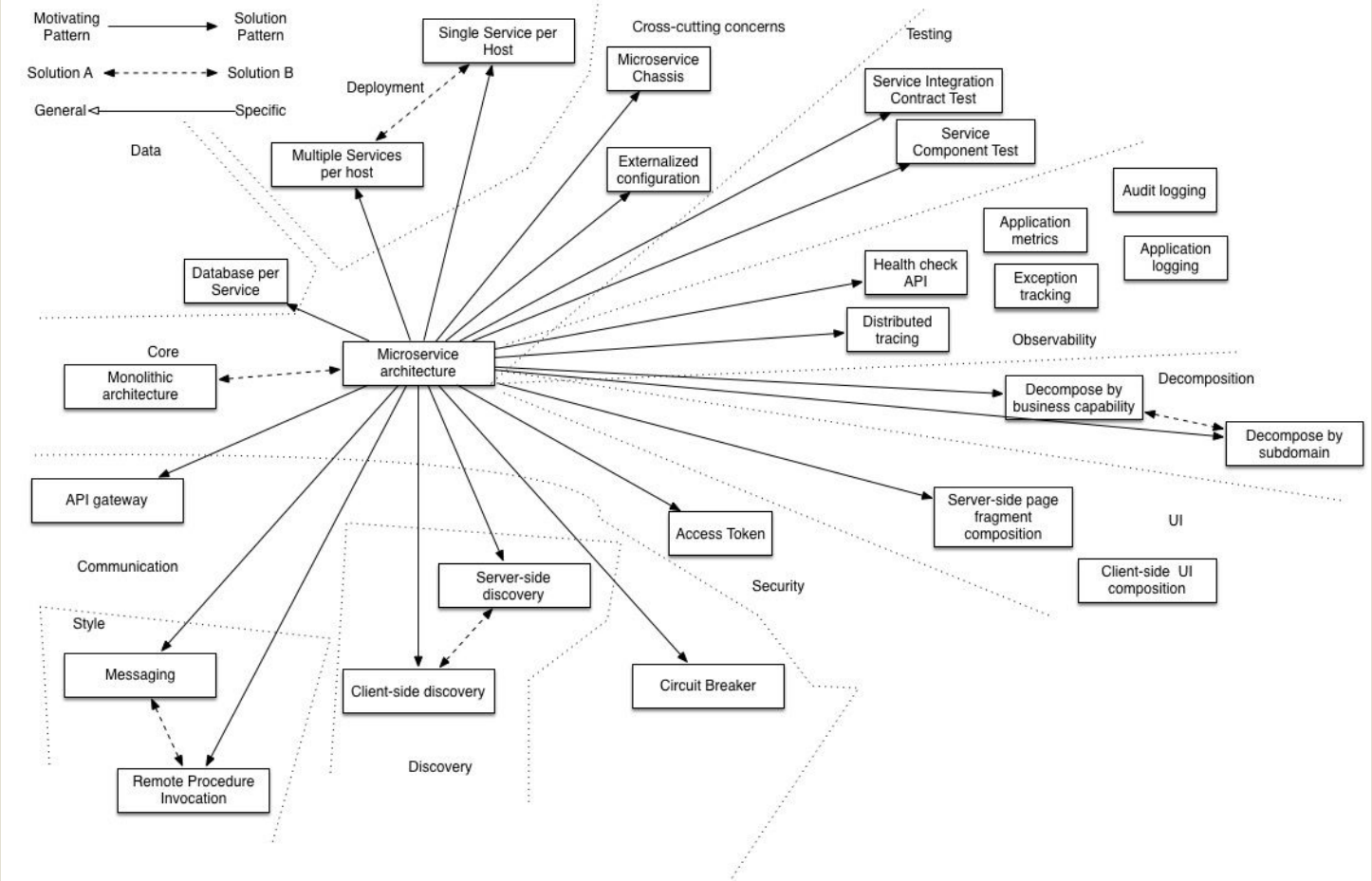
1. More overhead, more monitoring and deployment tooling

2. Complex services means harder to trace root causes

3. Tight integration between services might result in “Distributed Monolith”

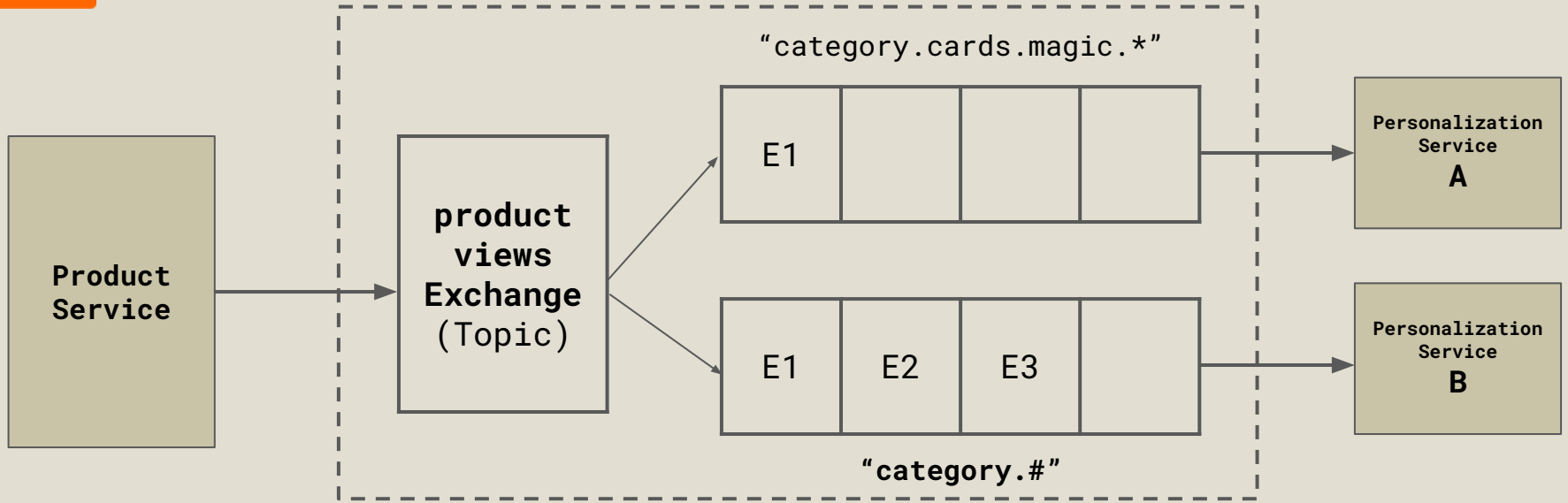
Microservice Design: Some Drawbacks

https://microservices.io/patterns/microservices.html



Microservice Design: Related Patterns

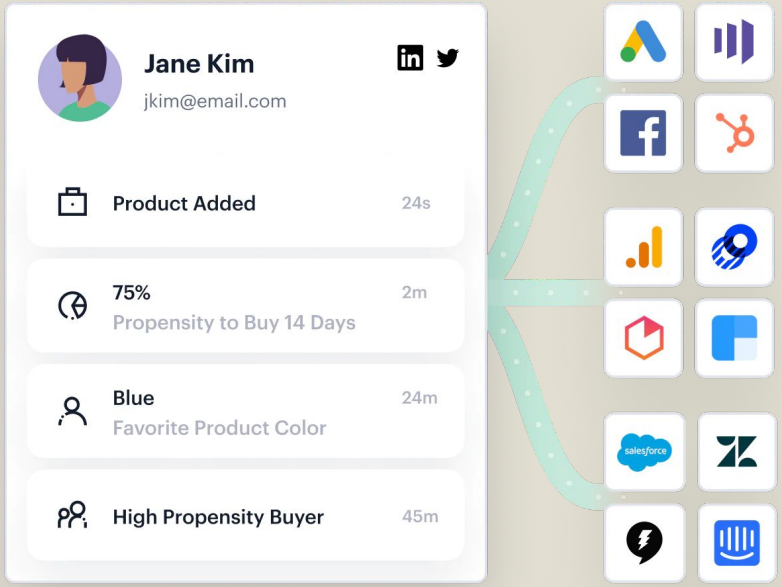
https://microservices.io/patterns/microservices.html



Services May Communicate via Queues Too

```
analytics.track(`Product Added`, {
  name: `Organic Banana`,
  price: 0.38
})
```

```
SELECT *
FROM website.product_added
WHERE name LIKE `%banana%`
LIMIT 1
```

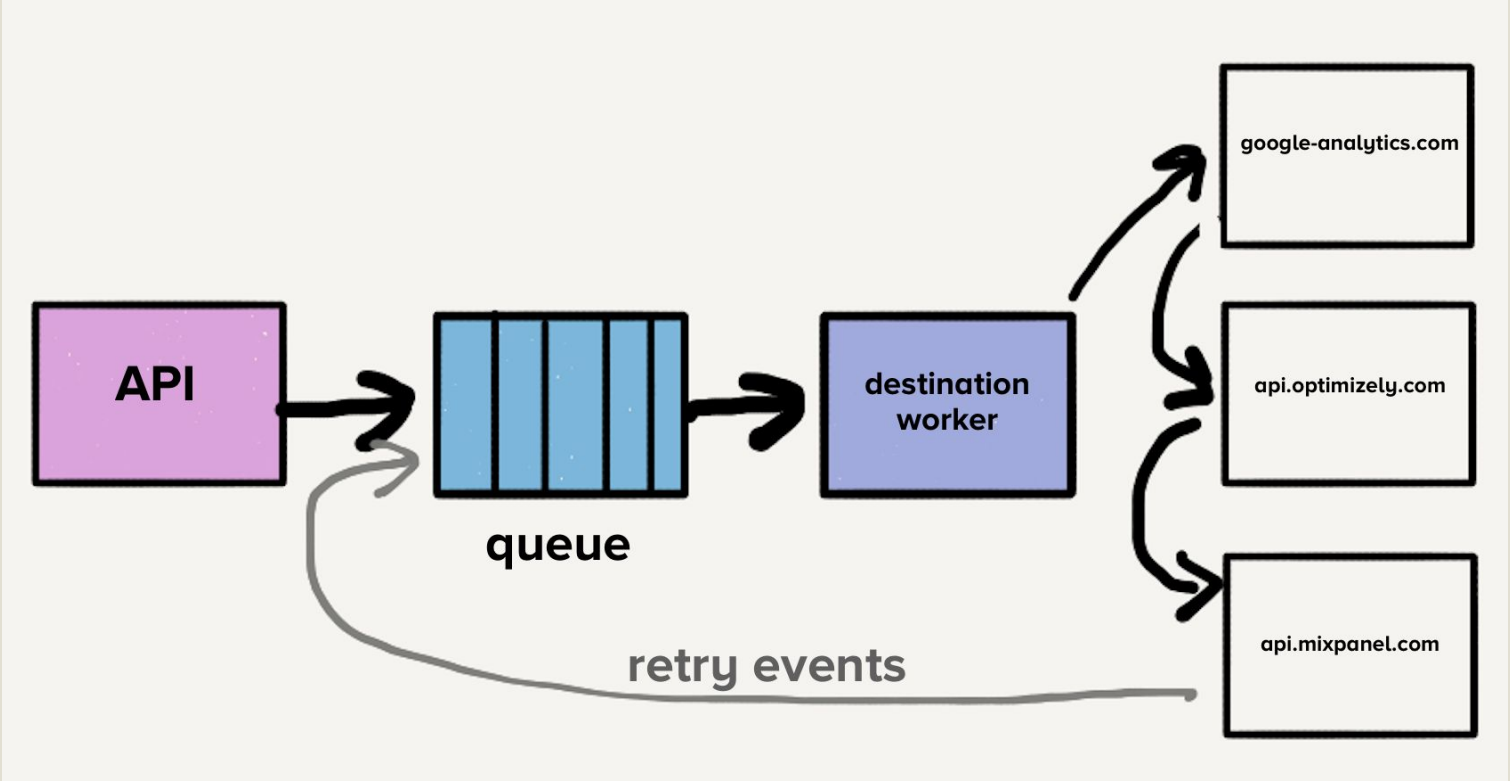


Case Study: Segment

https://segment.com/product/

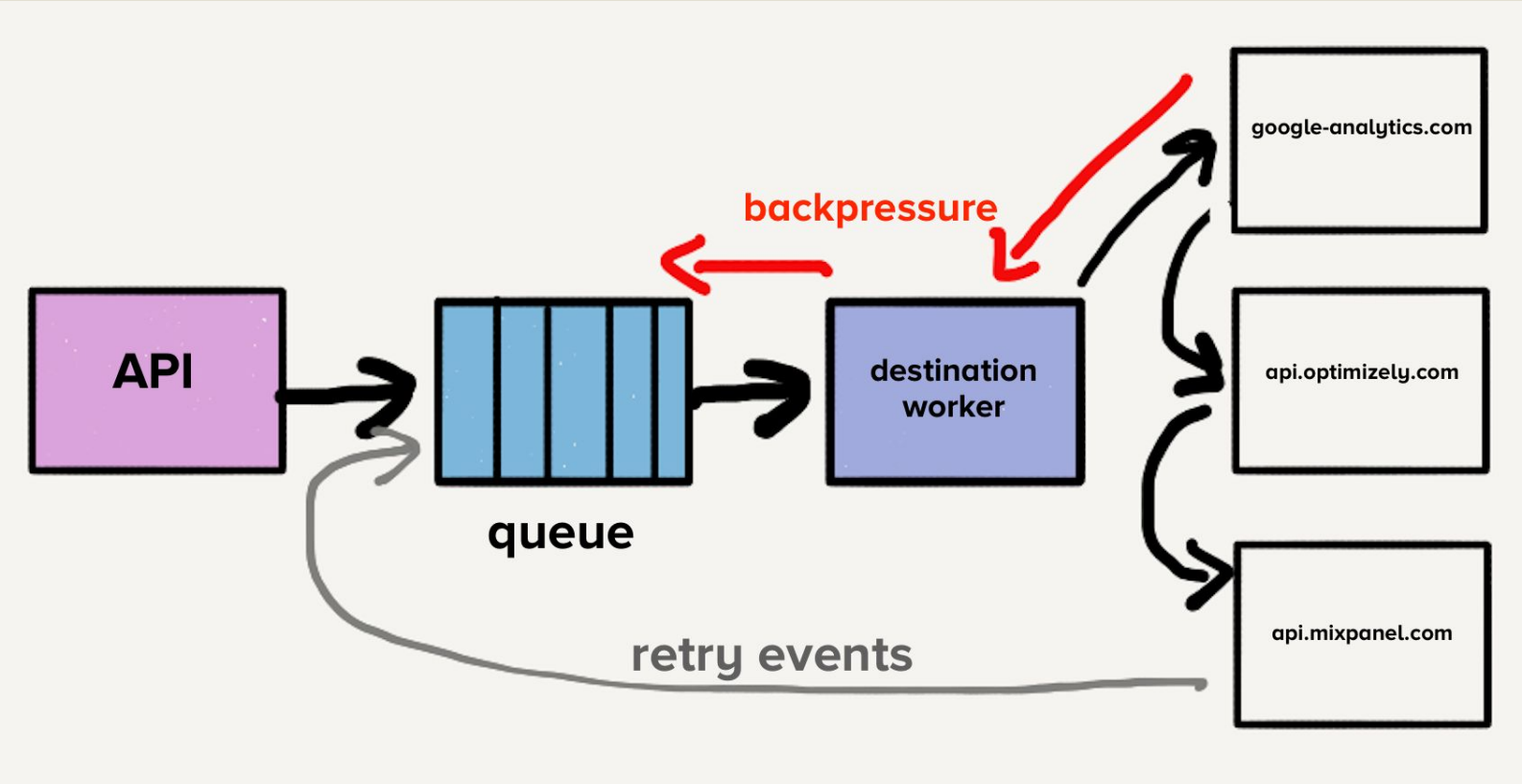
C1: Segment Connections

https://segment.com/product/



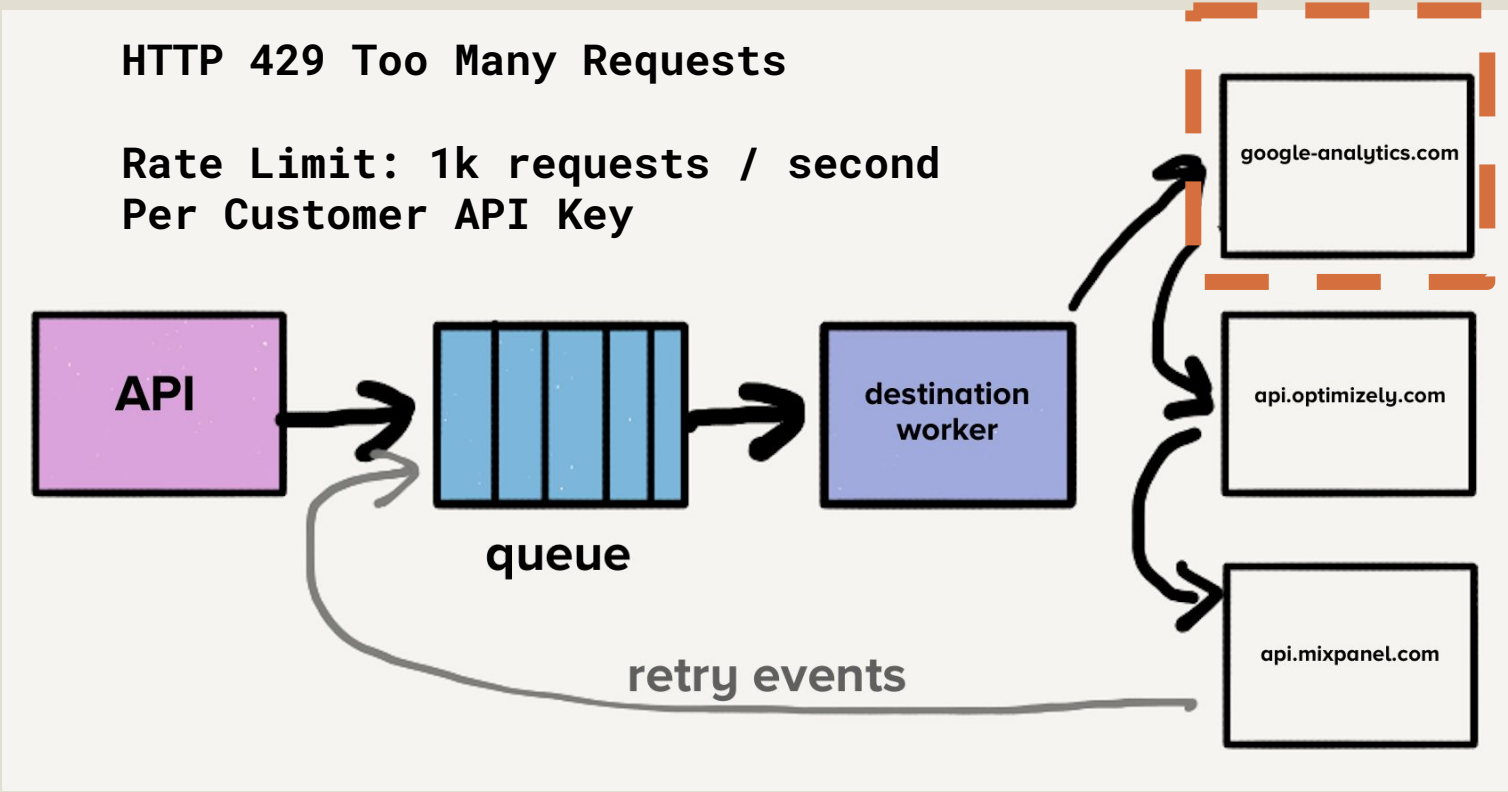
Goodbye Microservices

https://segment.com/blog/goodbye-microservices/



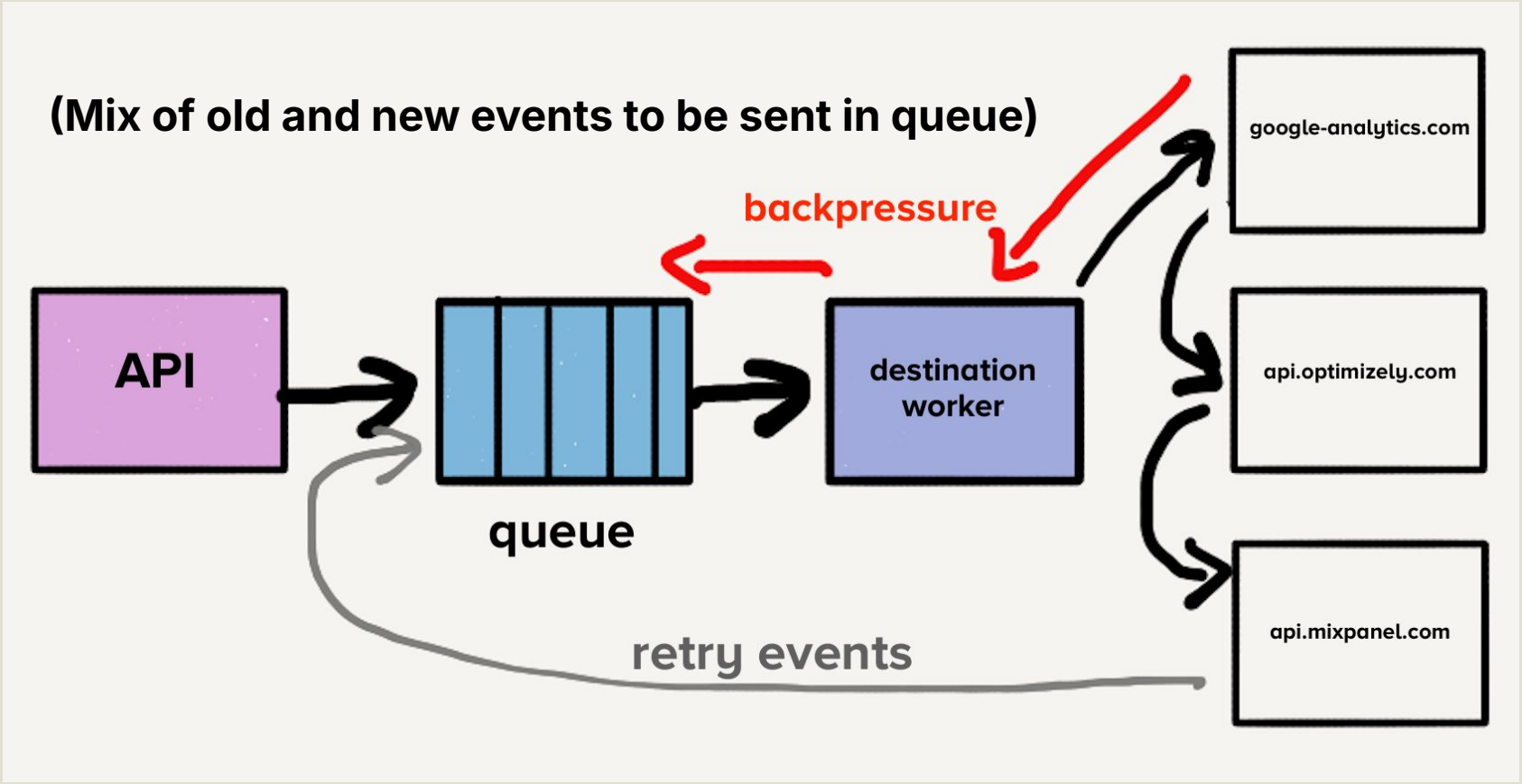
Backpressure: Resulting from Retries

https://segment.com/blog/goodbye-microservices/



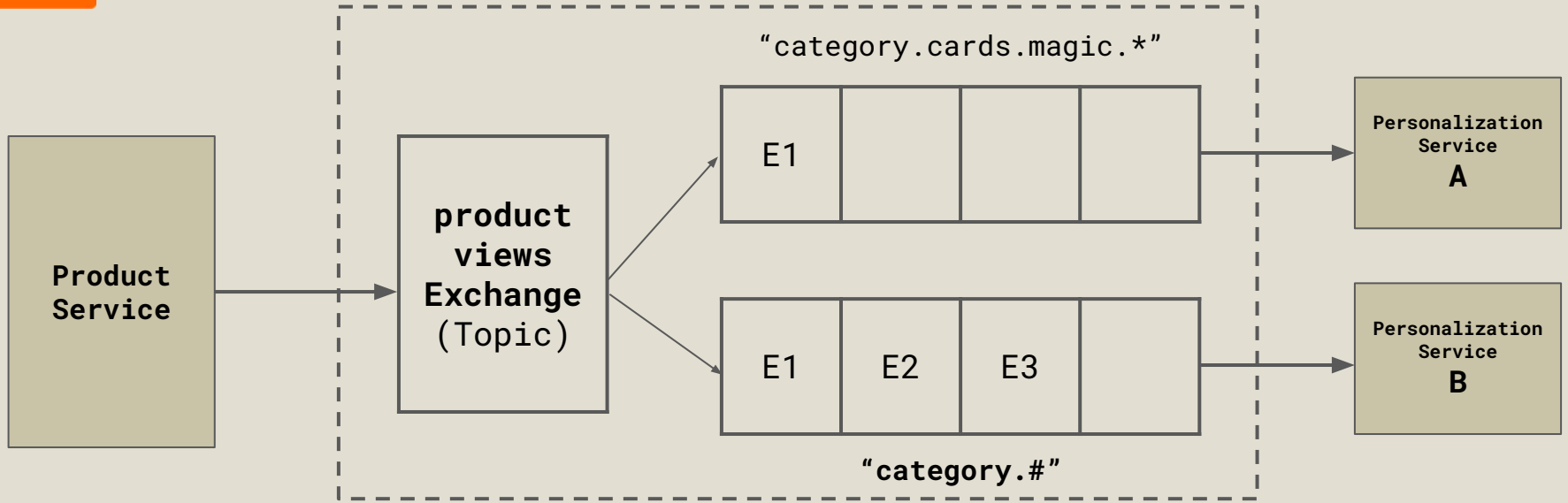
Rate Limited by External Services

https://segment.com/blog/goodbye-microservices/

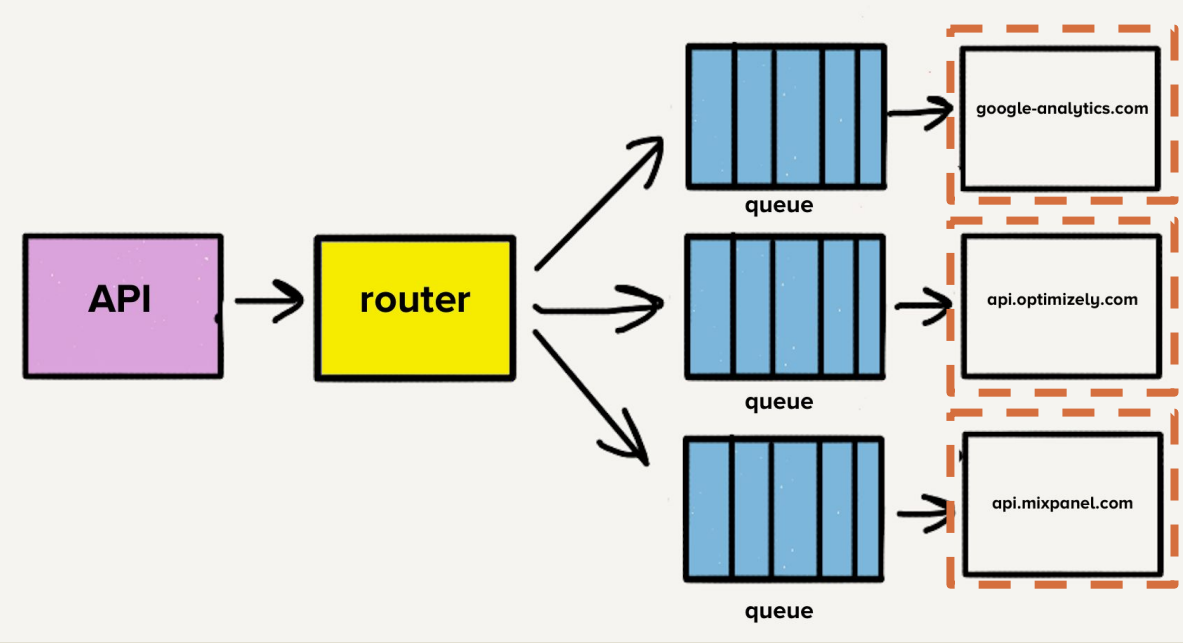


Backpressure: Resulting from Retries

https://segment.com/blog/goodbye-microservices/



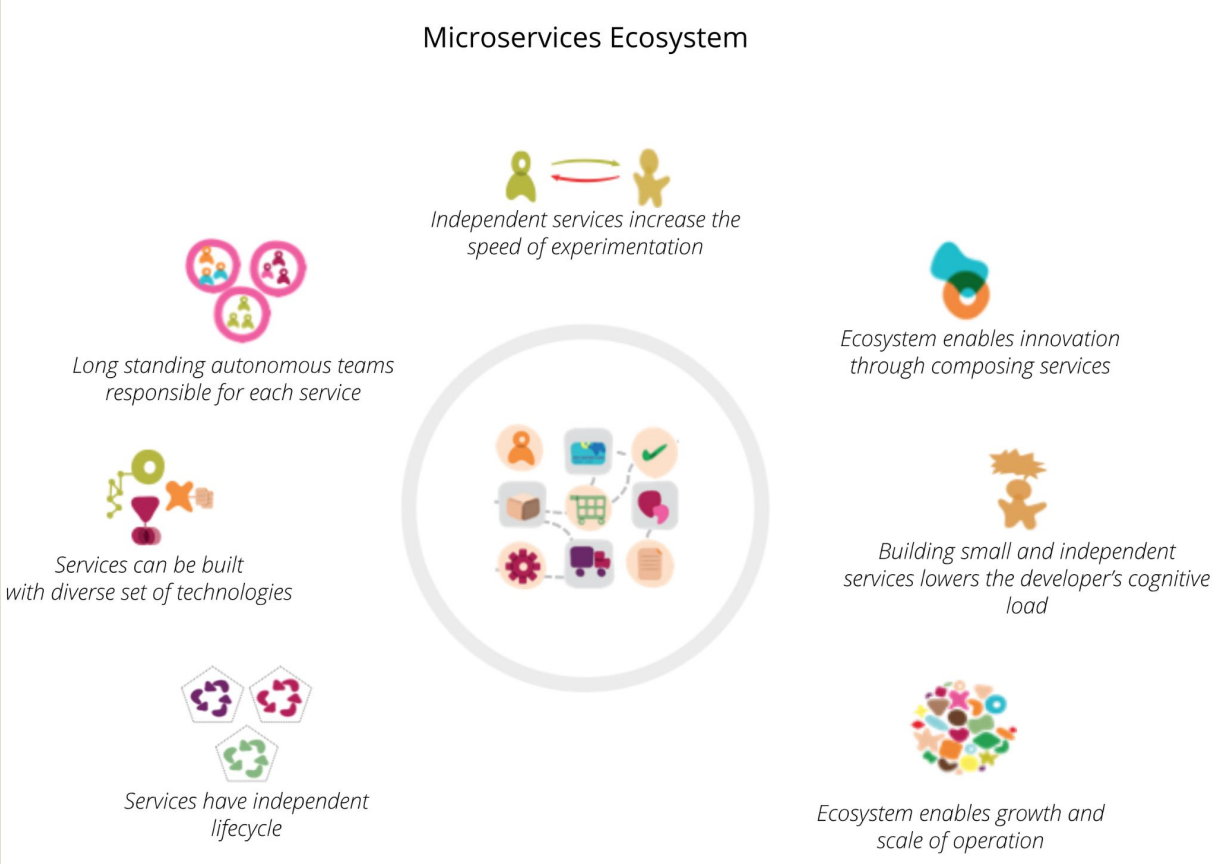
Split into Different Queues



Each worker is a microservice that consumes the queue and send requests to a vendor.

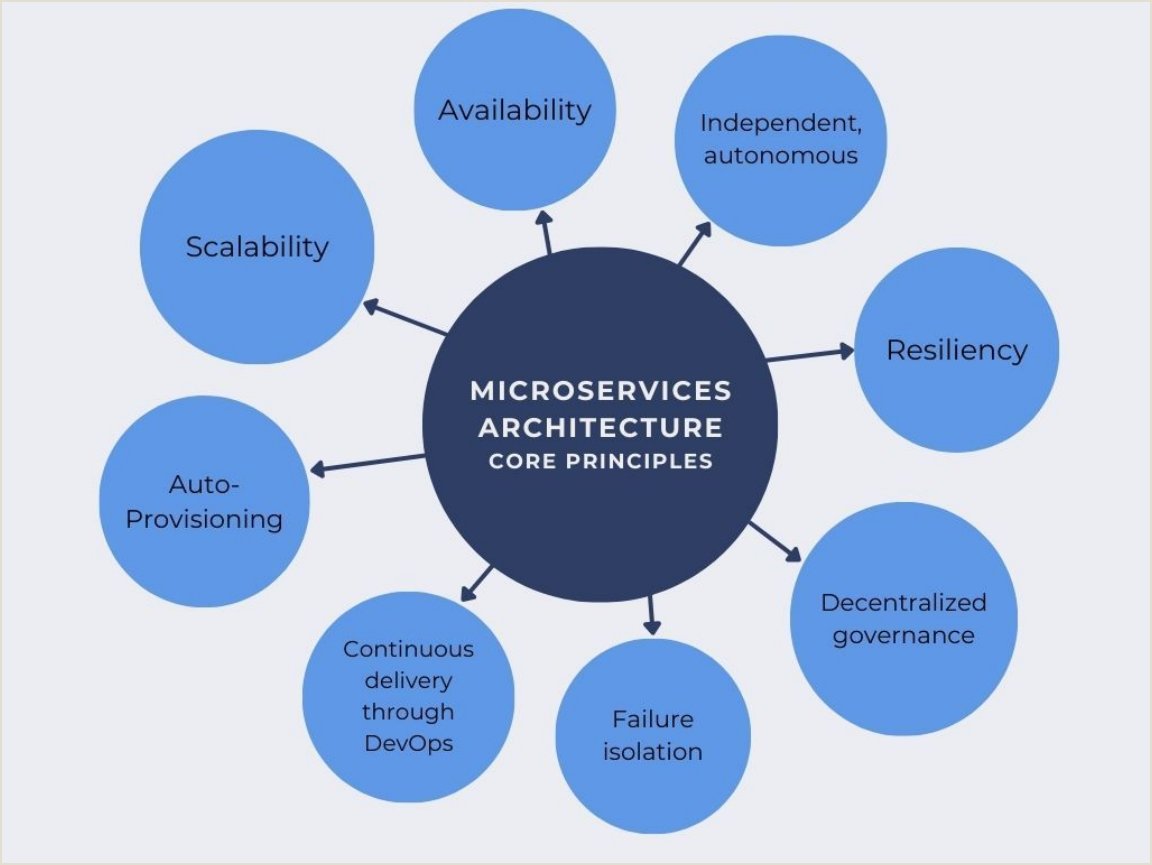
Backpressure: Resulting from Retries

https://segment.com/blog/goodbye-microservices/



How to break a Monolith into Microservices

https://martinfowler.com/articles/break-monolith-into-microservices.html



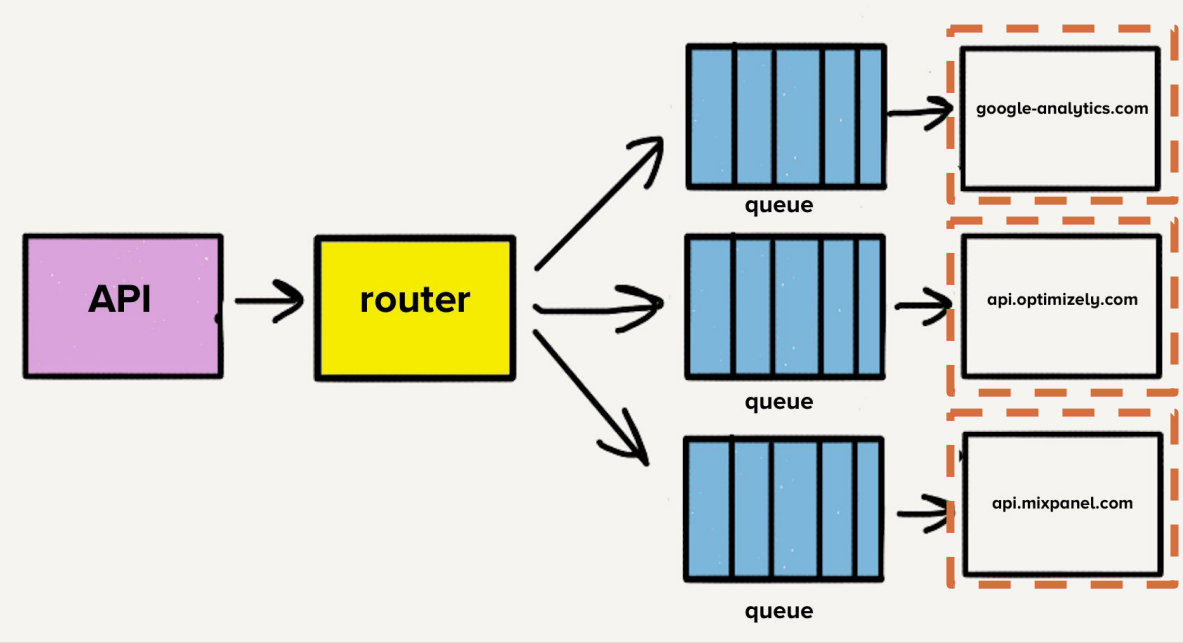
Microservices Design Patterns: Essential Architecture and Design Guide

https://dzone.com/articles/design-patterns-for-microservices

Benefits	Downsides
Strong Module Boundaries: Microservices reinforce modular structure, which is particularly important for larger teams. **Independent Deployment**: Simple services are easier to deploy, and since they are autonomous, are less likely to cause system failures when they go wrong. **Technology Diversity**: With microservices you can mix multiple languages, development frameworks and data-storage technologies.	**Distribution**: Distributed systems are harder to program, since remote calls are slow and are always at risk of failure. **Eventual Consistency**: Maintaining strong consistency is extremely difficult for a distributed system, which means everyone has to manage eventual consistency. **Operational Complexity**: You need a mature operations team to manage lots of services, which are being redeployed regularly.

Microservices: Trade-offs

https://martinfowler.com/articles/microservice-trade-offs.html



Ideal:
- Independent
- Single responsibility
- Scale horizontally

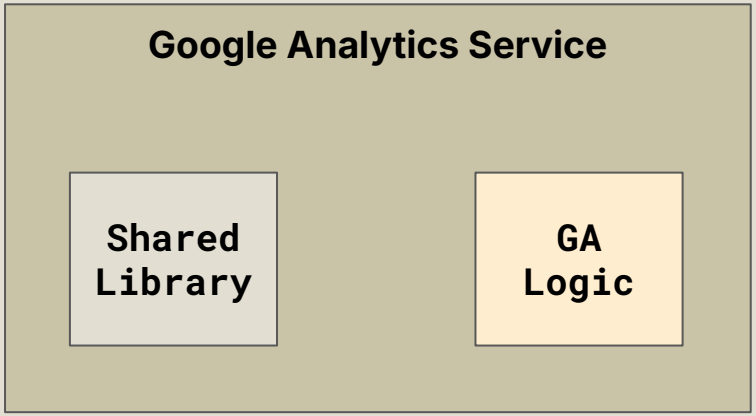
Workers are Ideal Microservices

https://segment.com/blog/goodbye-microservices/

Shared Library

- Used across all services

- Contains shared logic such as Queue processing, retries, etc.



Shared Library: Reuse Code

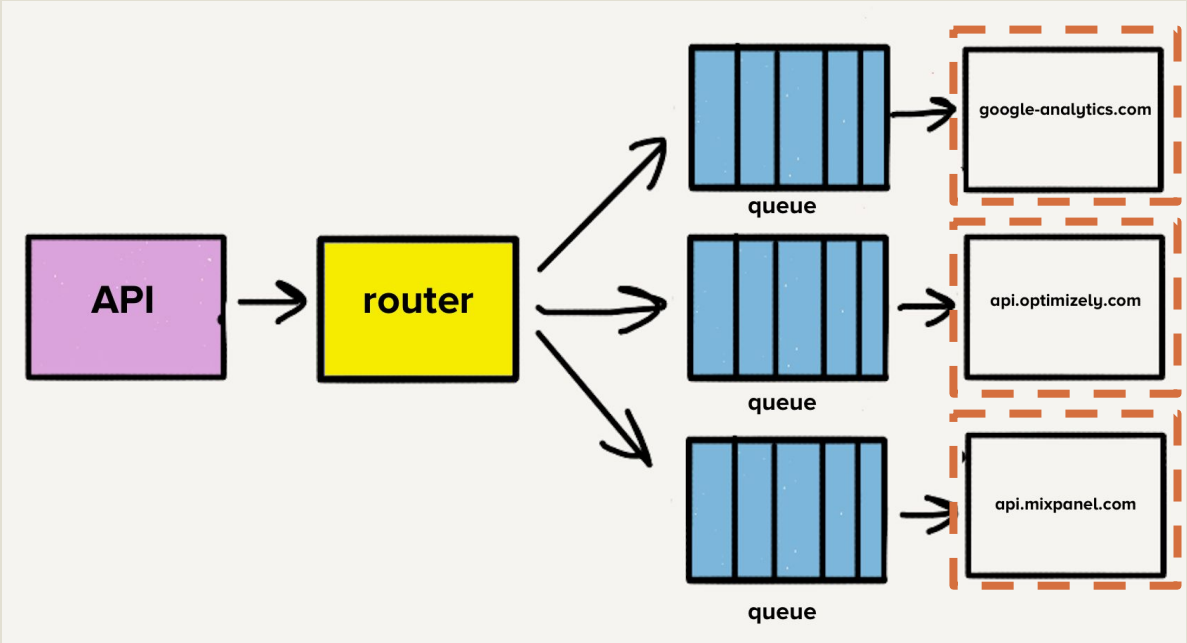
“As time went on, we added over 50 new destinations, and that meant 50 new repos. *[Note: Their total was 140 services!]*

Instead of enabling us to move faster, **the small team found themselves mired in exploding complexity**. Essential benefits of this architecture became burdens. As our velocity plummeted, our defect rate exploded.

Eventually, the team found themselves unable to make headway, with **3 full-time engineers spending most of their time just keeping the system alive**. Something had to change. This post is the story of how we took a step back and embraced an approach that aligned well with our product requirements and needs of the team.”

Moved back to Monolith

https://segment.com/blog/goodbye-microservices/



What Went Wrong?

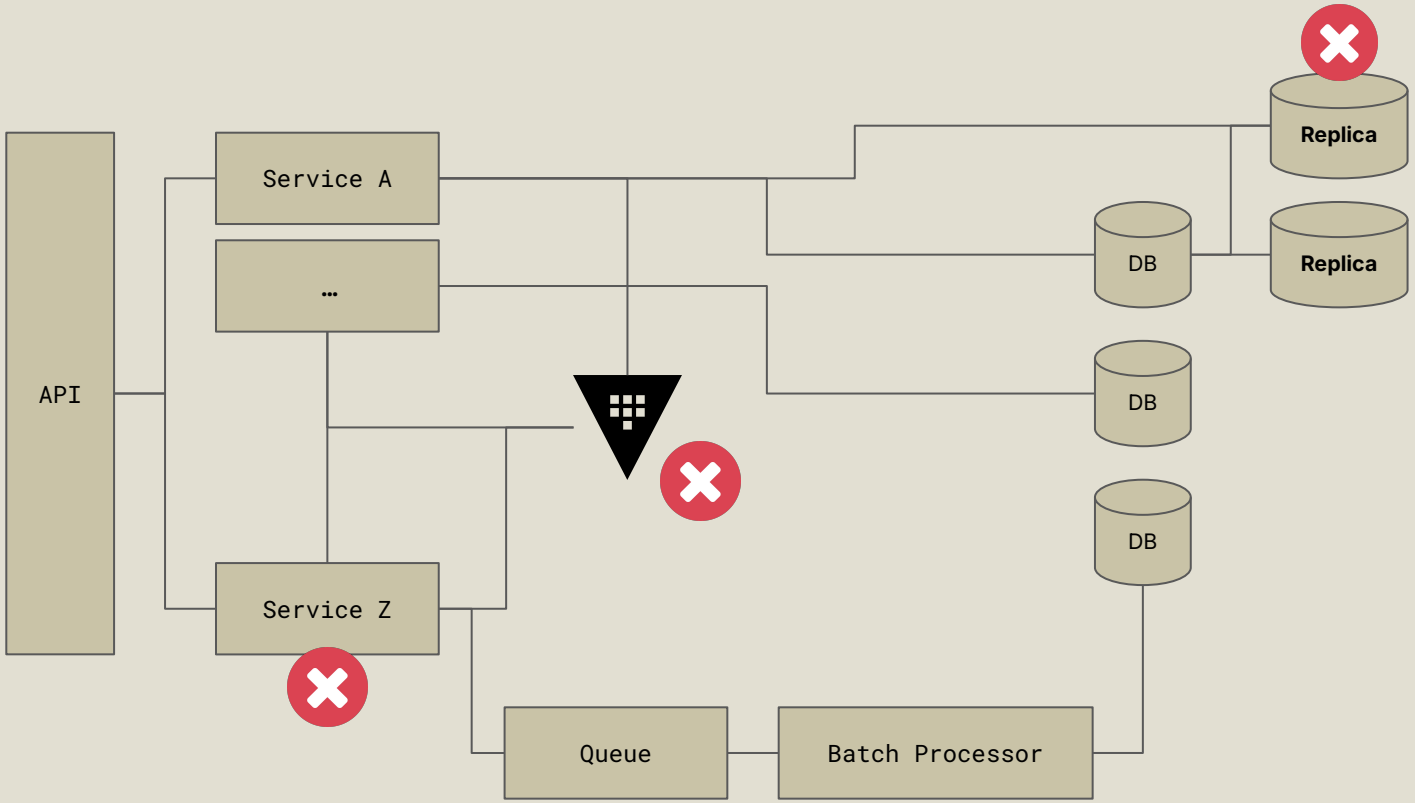
"... Eventually, the team found themselves unable to make headway, with **3 full-time engineers spending most of their time just keeping the system alive**."

[Remember: 140 services in total!]

Downsides to Microservices:

Operational Complexity: You need a mature operations team to manage lots of services, which are being redeployed regularly.

Problem 1: Team Size vs No. of Services



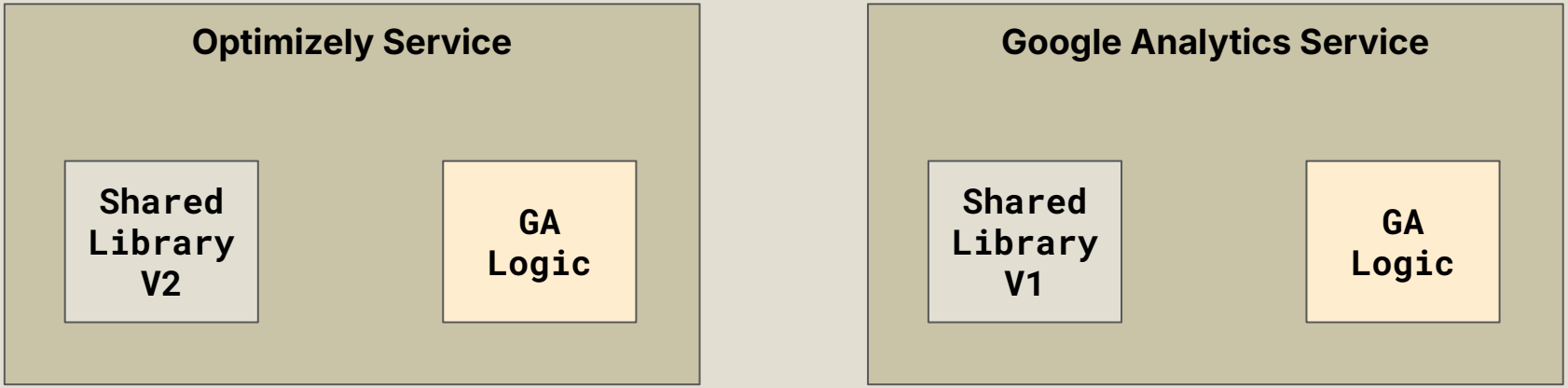
New Tools = Higher Overhead, Increased Risks

“The shared libraries made building new destinations quick. The familiarity brought by a uniform set of shared functionality made maintenance less of a headache.

However, a new problem began to arise. **Testing and deploying changes to these shared libraries impacted all of our destinations.** It began to require considerable time and effort to maintain.

Making changes to improve our libraries, knowing we’d have to test and deploy dozens of services, was a risky proposition. When pressed for time, engineers would only include the updated versions of these libraries on a single destination’s codebase.”

Problem 2: Maintenance Cost



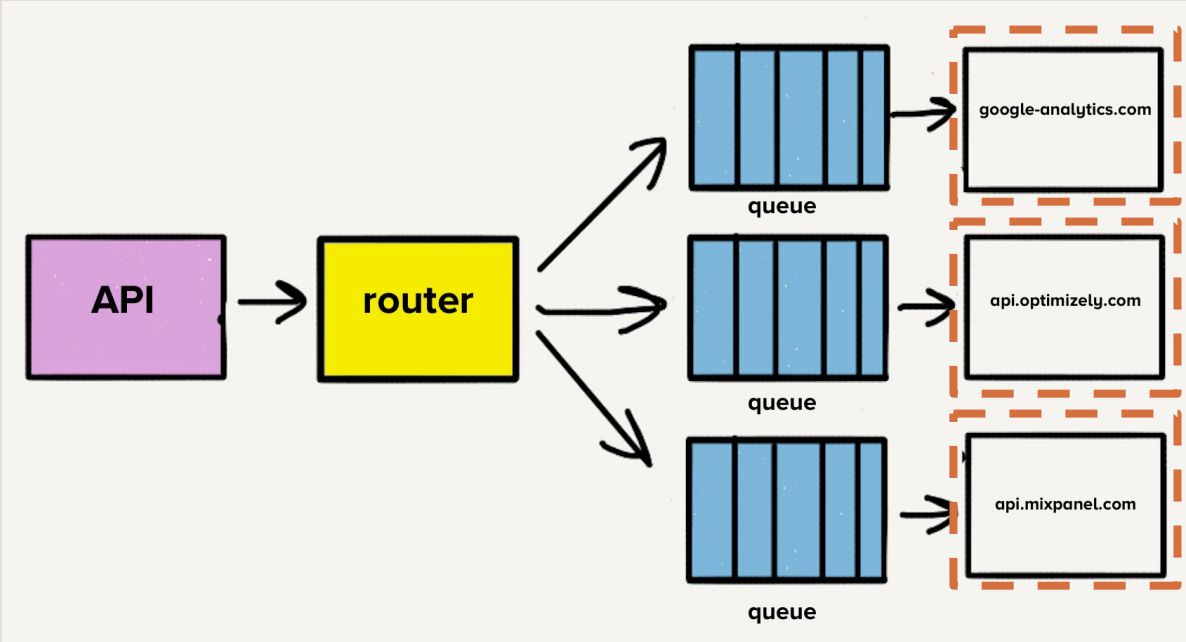
Updating 140 Services is Quite Challenging

“**While we did have auto-scaling implemented, each service had a distinct blend of required CPU and memory resources**, which made tuning the auto-scaling configuration more art than science.

The number of destinations continued to grow rapidly, with the team adding three destinations per month on average, which meant more repos, more queues, and more services. **With our microservice architecture, our operational overhead increased linearly with each added destination**...

The overhead from managing all of these services was a huge tax on our team. We were literally losing sleep over it since it was common for the on-call engineer to get paged to deal with load spikes.”

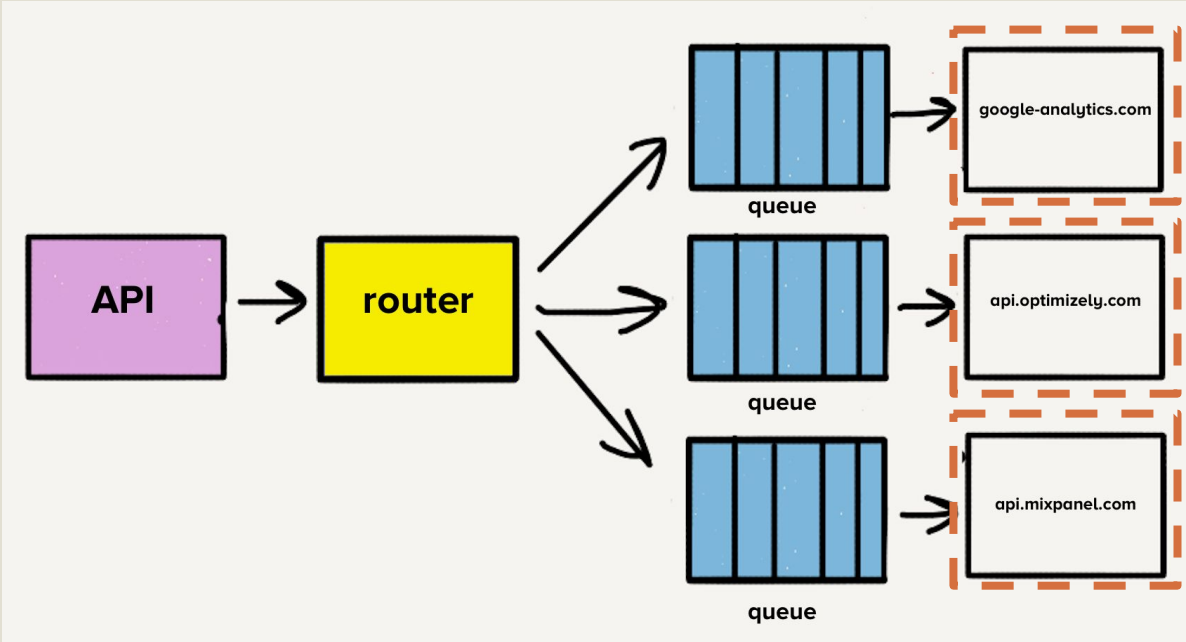
Problem 3: Load Spikes



3,000 events / min

500 events / min

Customer A: Load Pattern

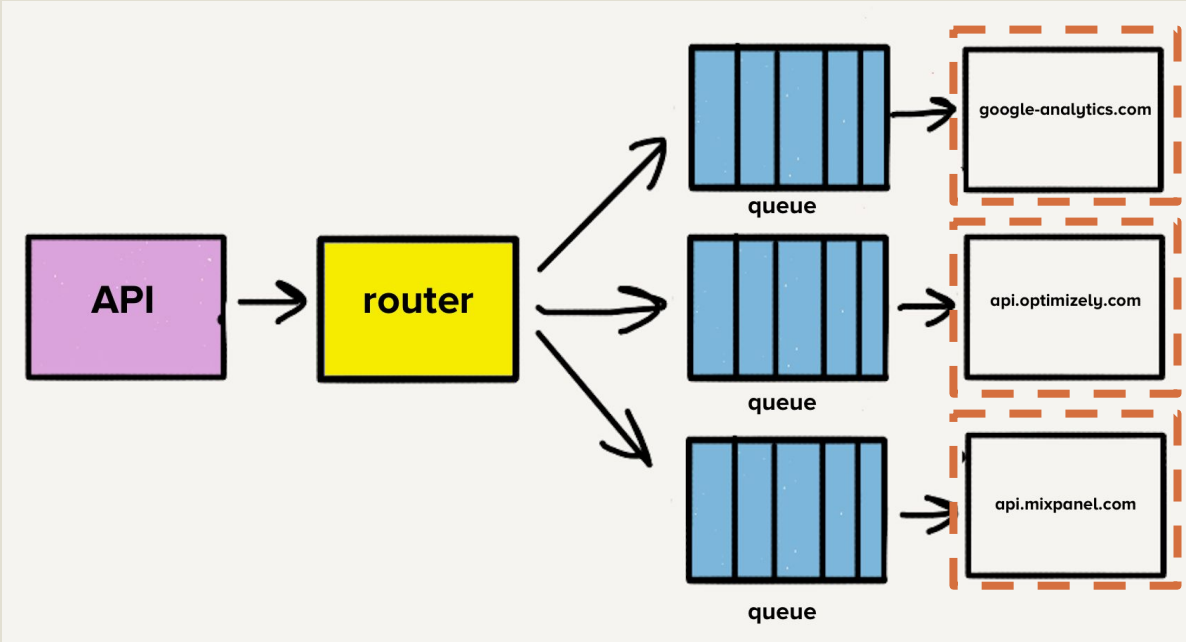


15,000 events / min

10,000 events / min

9,000 events / min

Customer B: Load Pattern

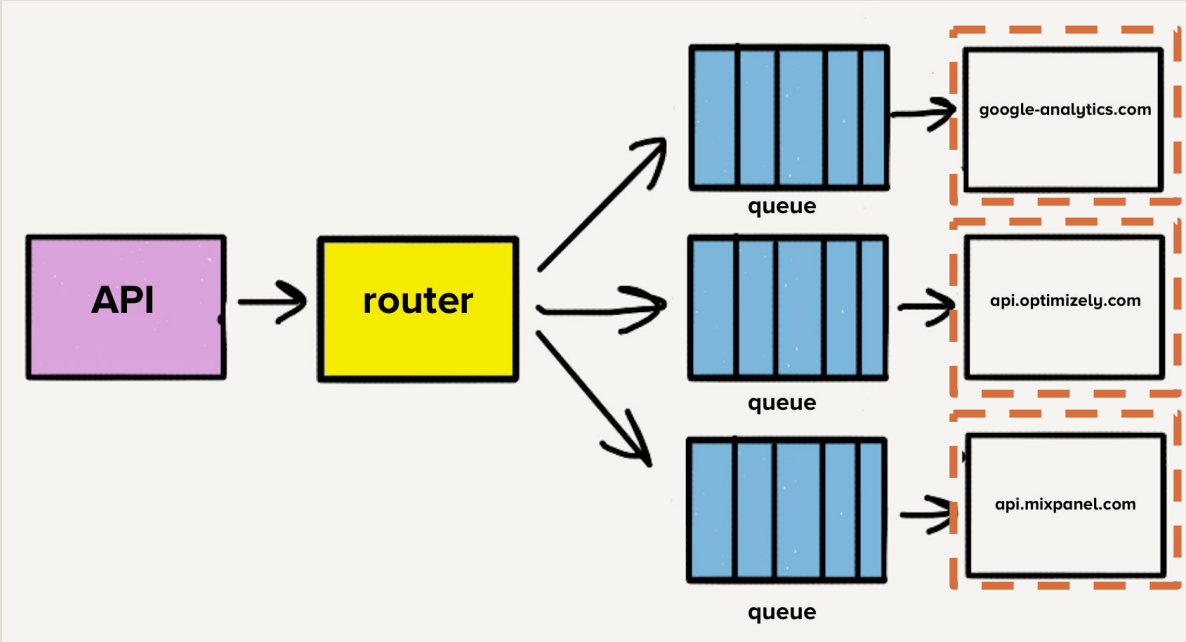


0 events / min

0 events / min

0 events / min

Customer C: Quiet Time



25,000 events / min

10,000 events / min

10,000 events / min

Customer C: Push Notifications Sent

For a simple system with 3 services, 3 components:

Probability of any single part working = 1 - 0.001 = 0.999 (99.9%)

Probability of all parts working = (0.999)^6 ≈ 0.994 (99.4%)

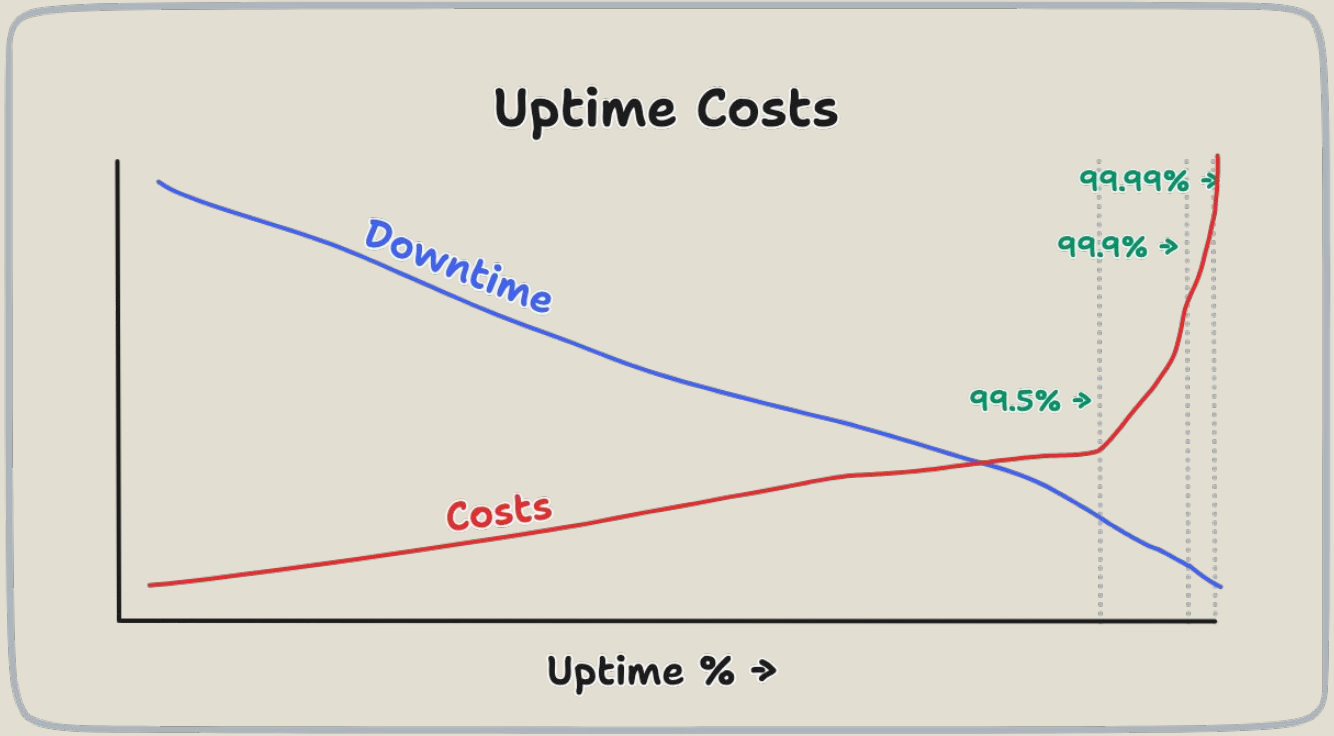
Probability of at least one part failing = 1 - 0.994 = 0.006 (0.6%)

~0.6% chance of experiencing some downtime

For a complex system with 140 services, 140 components:

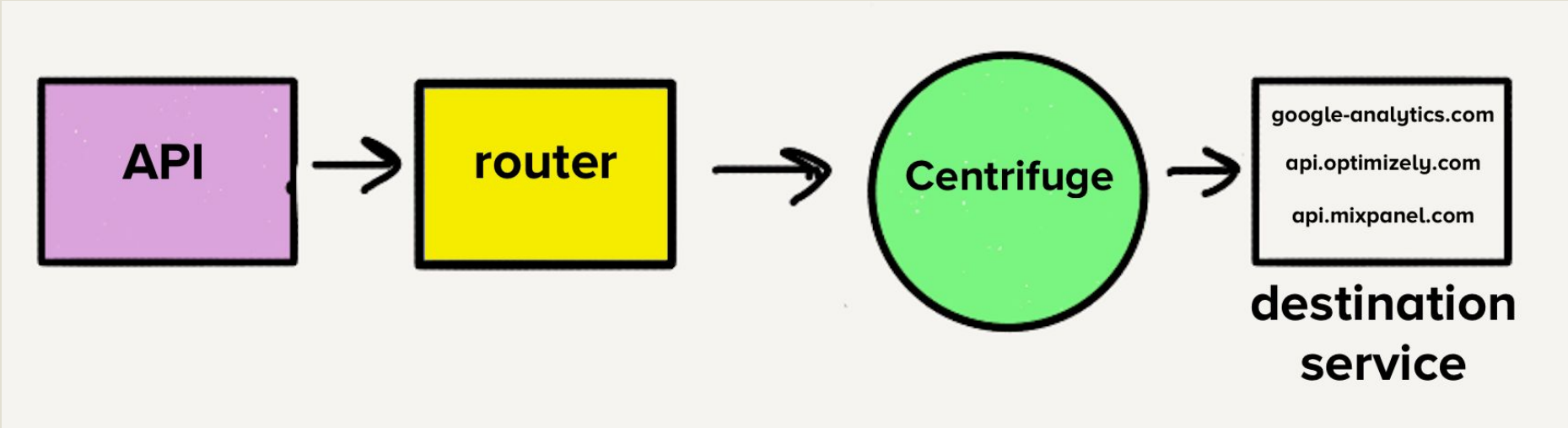
~24.4% chance of experiencing some downtime

IF: 0.1% Chance of Any Component Failing in a Month



Uptime Guarantees – A Pragmatic Perspective

https://world.hey.com/itzy/uptime-guarantees-a-pragmatic-perspective-736d7ea4



Solution: Merge 140 Microservices back into 1 Service

https://segment.com/blog/goodbye-microservices/

“Our initial microservice architecture worked for a time, solving the immediate performance issues in our pipeline by isolating the destinations from each other. However, we weren’t set up to scale. We lacked the proper tooling for testing and deploying the microservices when bulk updates were needed. As a result, our developer productivity quickly declined.

The proof was in the improved velocity. When our microservice architecture was still in place, we made 32 improvements to our shared libraries. One year later, we’ve made 46 improvements.

The change also benefited our operational story. With every destination living in one service, we had a good mix of CPU and memory-intense destinations, which made scaling the service to meet demand significantly easier. **The large worker pool can absorb spikes in load, so we no longer get paged for destinations that process small amounts of load.**”

Happy Ending (?)

https://segment.com/blog/goodbye-microservices/

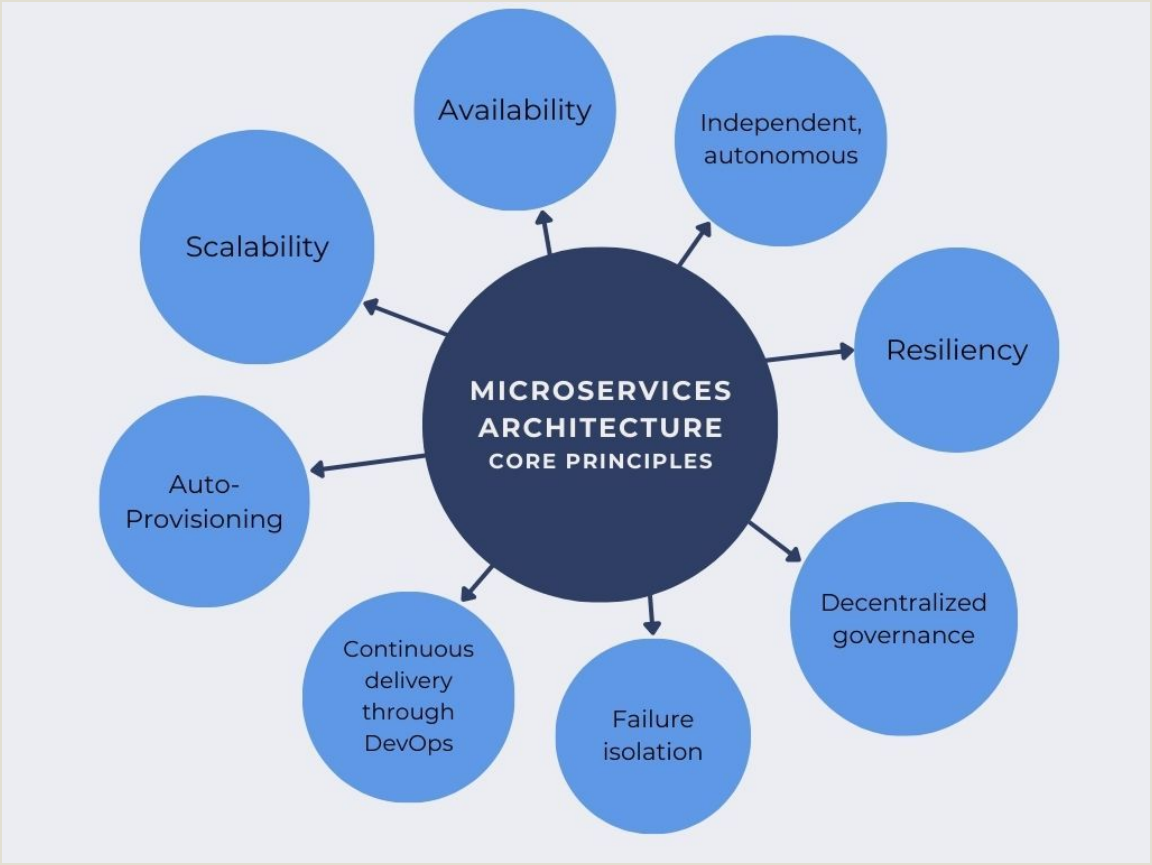
1. **Fault isolation is difficult.** With everything running in a monolith, if a bug is introduced in one destination that causes the service to crash, the service will crash for all destinations. We have comprehensive automated testing in place, but tests can only get you so far. We are currently working on a much more robust way to prevent one destination from taking down the entire service while still keeping all the destinations in a monolith.

2. **In-memory caching is less effective.** Previously, with one service per destination, our low traffic destinations only had a handful of processes, which meant their in-memory caches of control plane data would stay hot. Now that cache is spread thinly across 3000+ processes so it's much less likely to be hit. We could use something like Redis to solve for this, but then that's another point of scaling for which we'd have to account. In the end, we accepted this loss of efficiency given the substantial operational benefits.

3. **Updating the version of a dependency may break multiple destinations.** While moving everything to one repo solved the previous dependency mess we were in, it means that if we want to use the newest version of a library, we'll potentially have to update other destinations to work with the newer version. In our opinion though, the simplicity of this approach is worth the trade-off. And with our comprehensive automated test suite, we can quickly see what breaks with a newer dependency version.

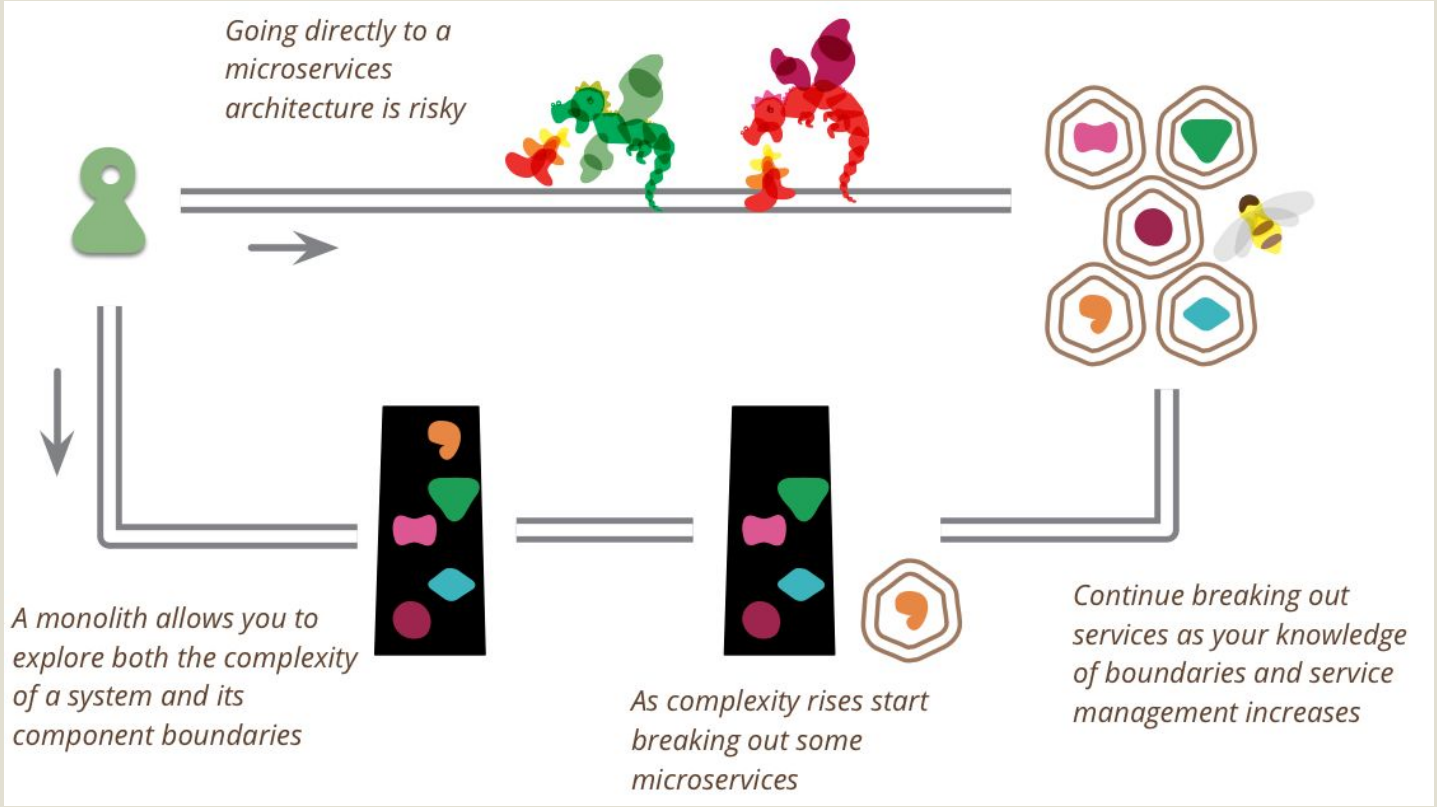
Trade-offs for Monolith:

https://segment.com/blog/goodbye-microservices/



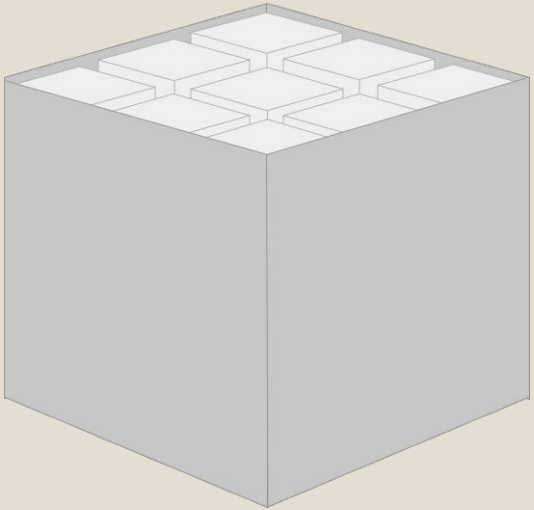
Microservices Design Patterns: Essential Architecture and Design Guide

https://dzone.com/articles/design-patterns-for-microservices



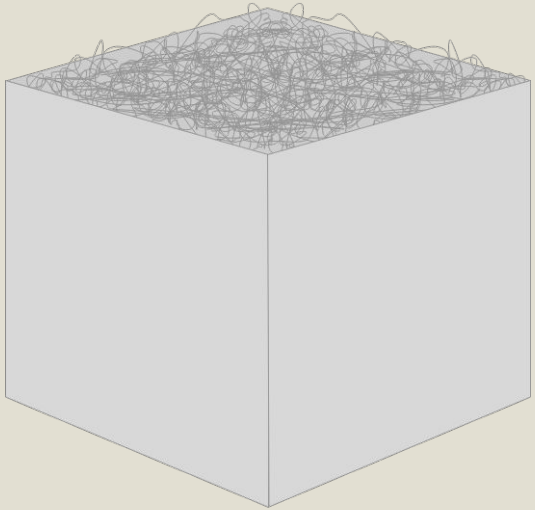
Microservices: Monolith First

https://martinfowler.com/bliki/MonolithFirst.html



Hope

vs.



Reality

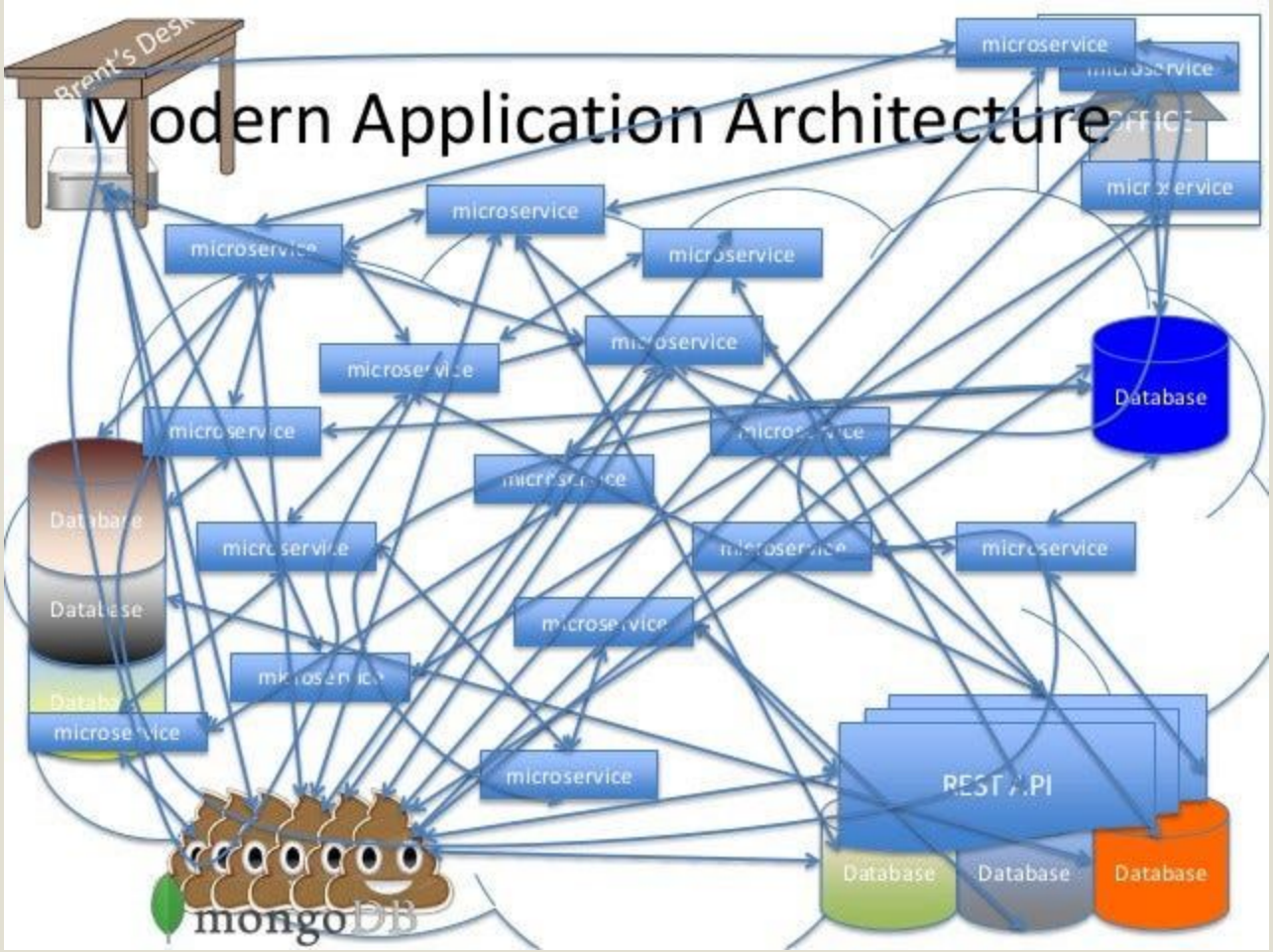
Don’t start with a monolith… when your goal is a microservices architecture

https://martinfowler.com/articles/dont-start-monolith.html

Start with Monolith if...

- You are building a new startup (from ground up)
- Your engineering team is small (< 10)
- Your engineering team has little to no experience with
  - Microservices,
  - Monitoring and Observability, and or
  - Incident Management
- Your team has no time to invest in building shared tooling or infrastructure to support microservices

Starting with Monolith or Microservices



"Modern Application Architecture"

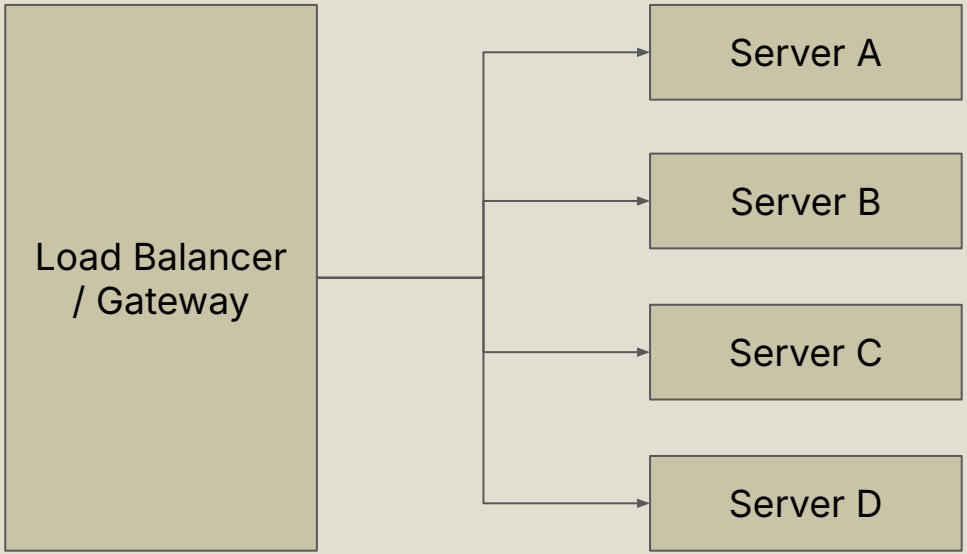
https://medium.com/@damonallison/evolving-microservices-with-event-sourcing-7396e015cfd2

Start with Microservices if...

- Your team has time or has already invested in building shared tooling or infrastructure to support microservices
- There is already sufficient traffic and traction for the service / product to warrant the need
- Teams are sufficient large enough that a single monolith may cause code conflicts or delays
- Services have clear boundaries between them

Starting with Monolith or Microservices

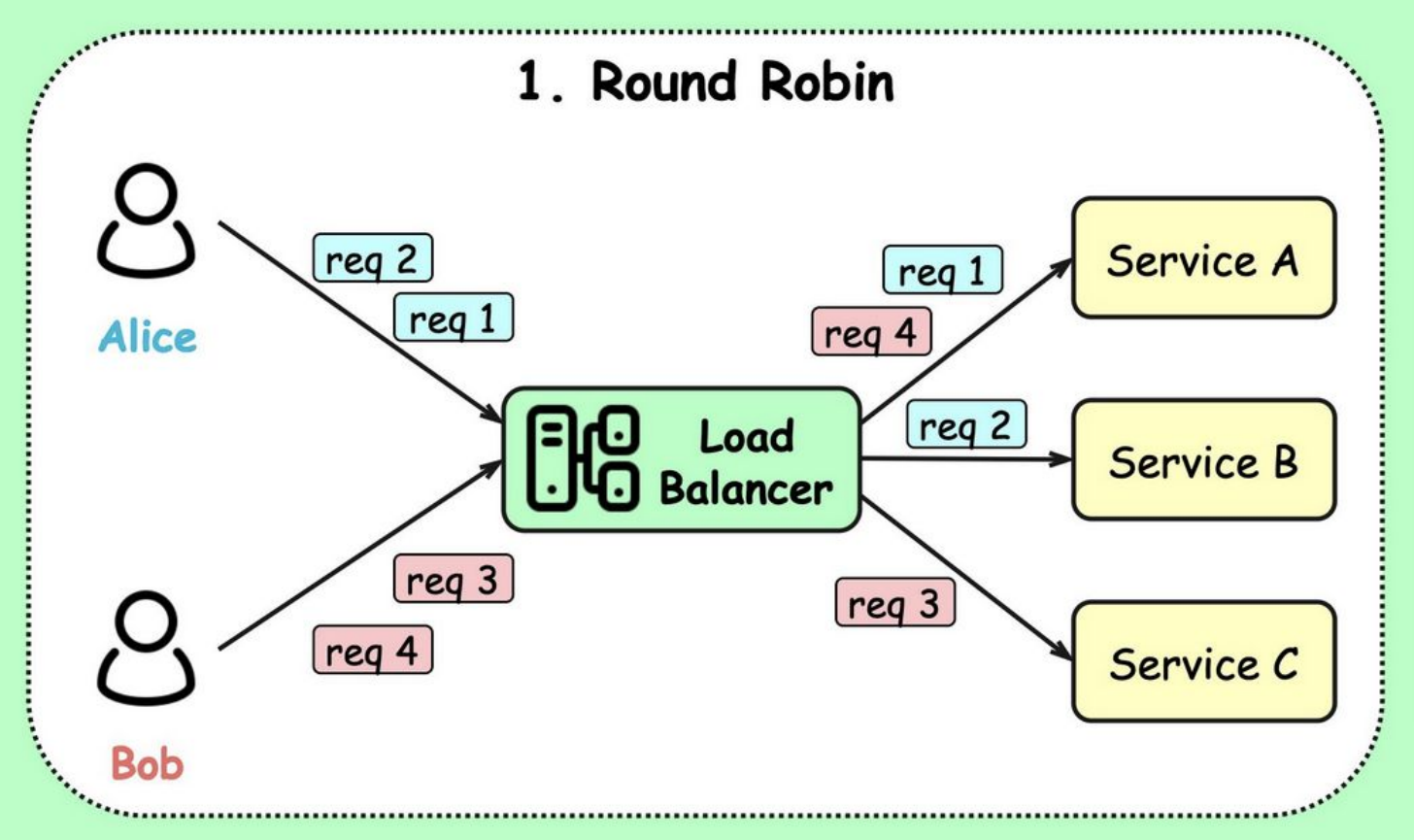
7.2
load balancing case study



Load Balancing Between 4 Servers

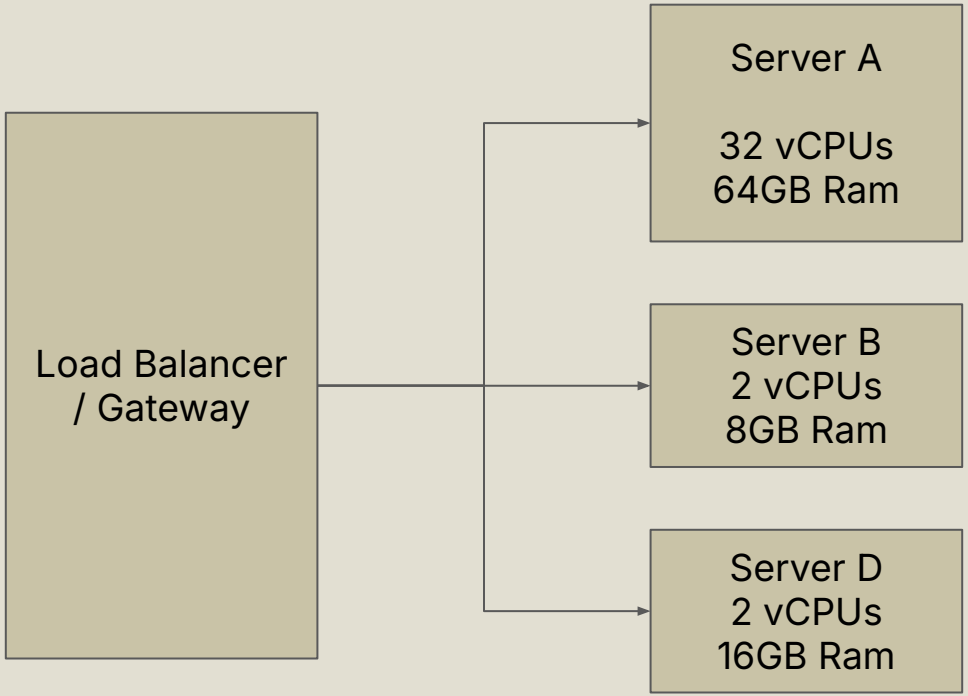
Algorithm	Description
Round Robin	Distributes requests sequentially across all servers in a cyclical order, ideal for uniform workloads.
Weighted Round Robin	Assigns weights to servers based on their capacity, directing more traffic to higher-capacity servers.
Least Connections	Routes traffic to the server with the fewest active connections, useful for dynamic workloads.
Weighted Least Connections	Combines least connections with server weights to account for varying server capacities.

Common Load Balancing Algorithms (i)

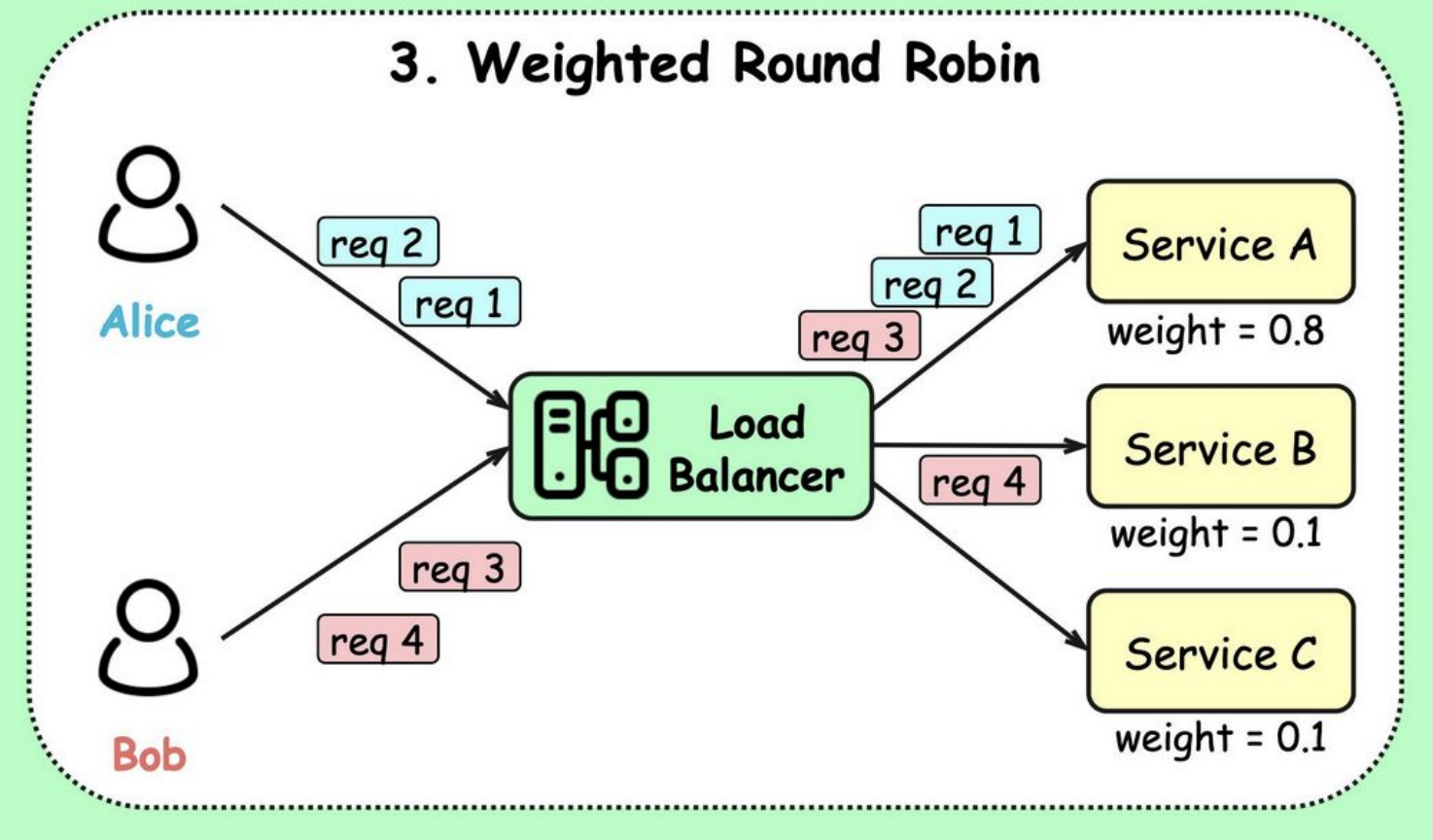


Load Balancing Algorithms Illustrated

https://blog.bytebytego.com/p/ep47-common-load-balancing-algorithms

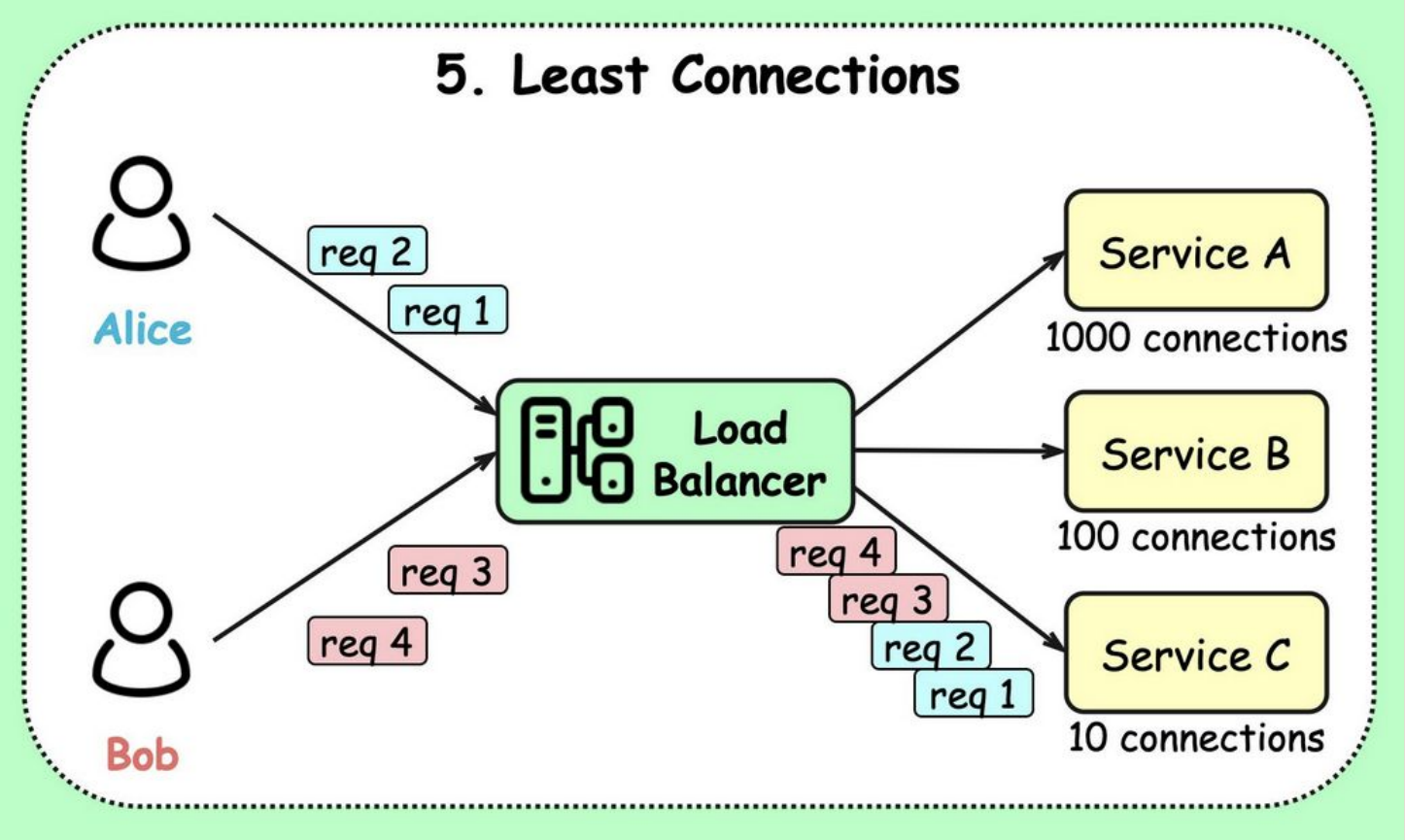


Servers of Different Computing Capacity



Load Balancing Algorithms Illustrated

https://blog.bytebytego.com/p/ep47-common-load-balancing-algorithms

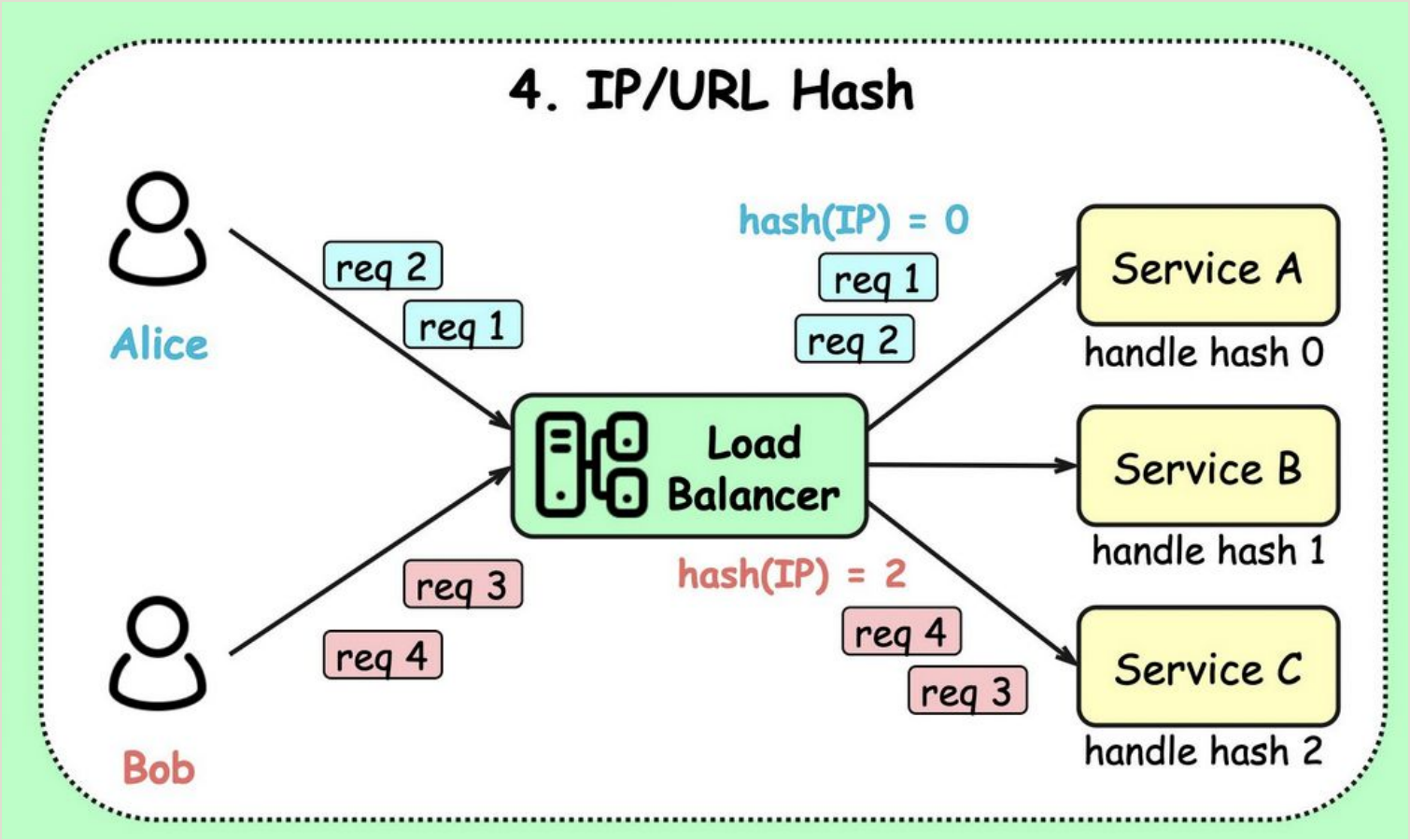


Load Balancing Algorithms Illustrated

https://blog.bytebytego.com/p/ep47-common-load-balancing-algorithms

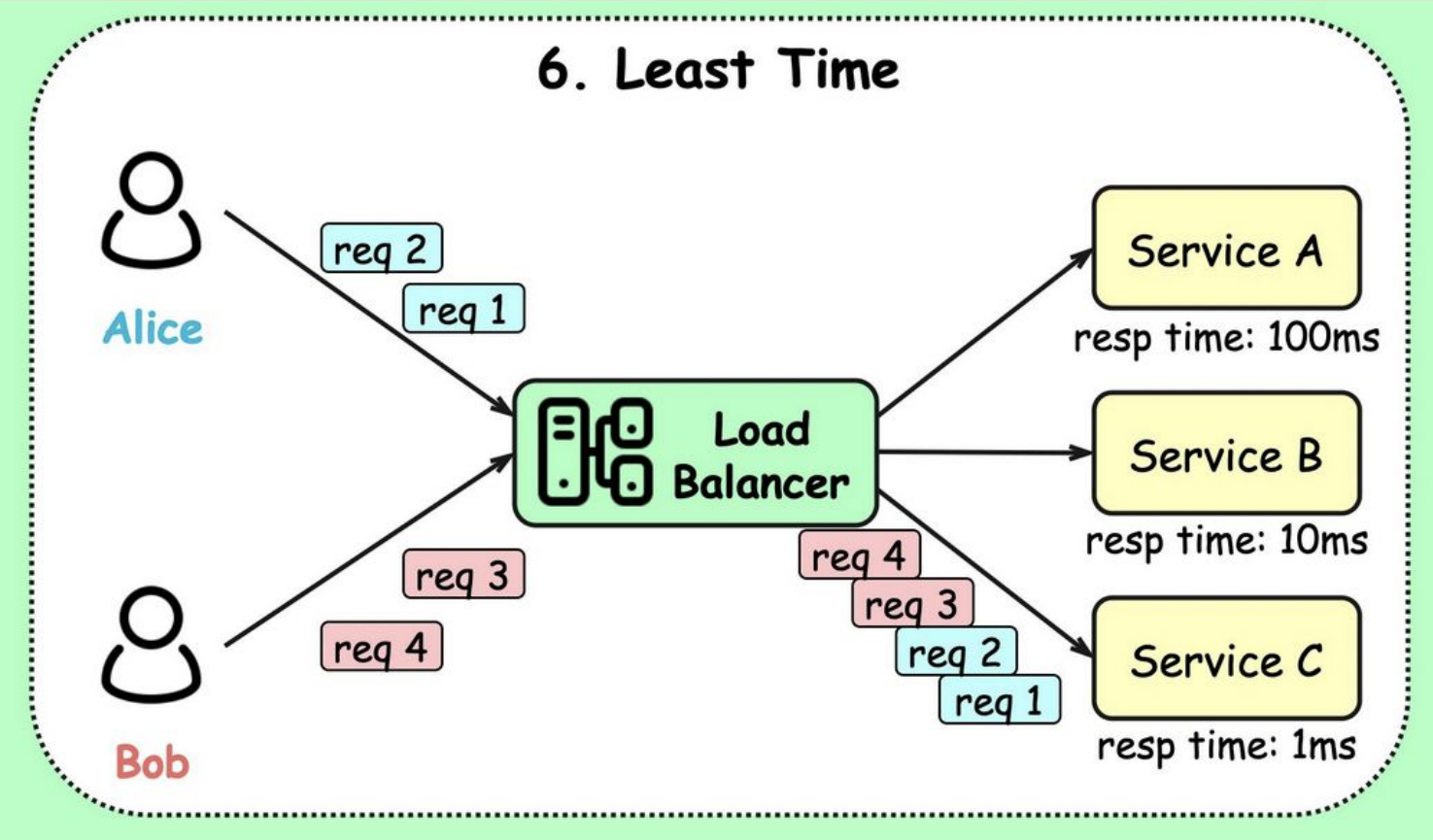
Algorithm	Description
IP / URL Hash	Uses a hash of the client's IP or URL address to consistently route requests to the same server.
Least Response Time	Directs traffic to the server with the fastest response time, optimizing latency-sensitive applications.
Sticky Sessions (Affinity)	Ensures subsequent requests from a client are routed to the same server for session persistence.

Common Load Balancing Algorithms (ii)



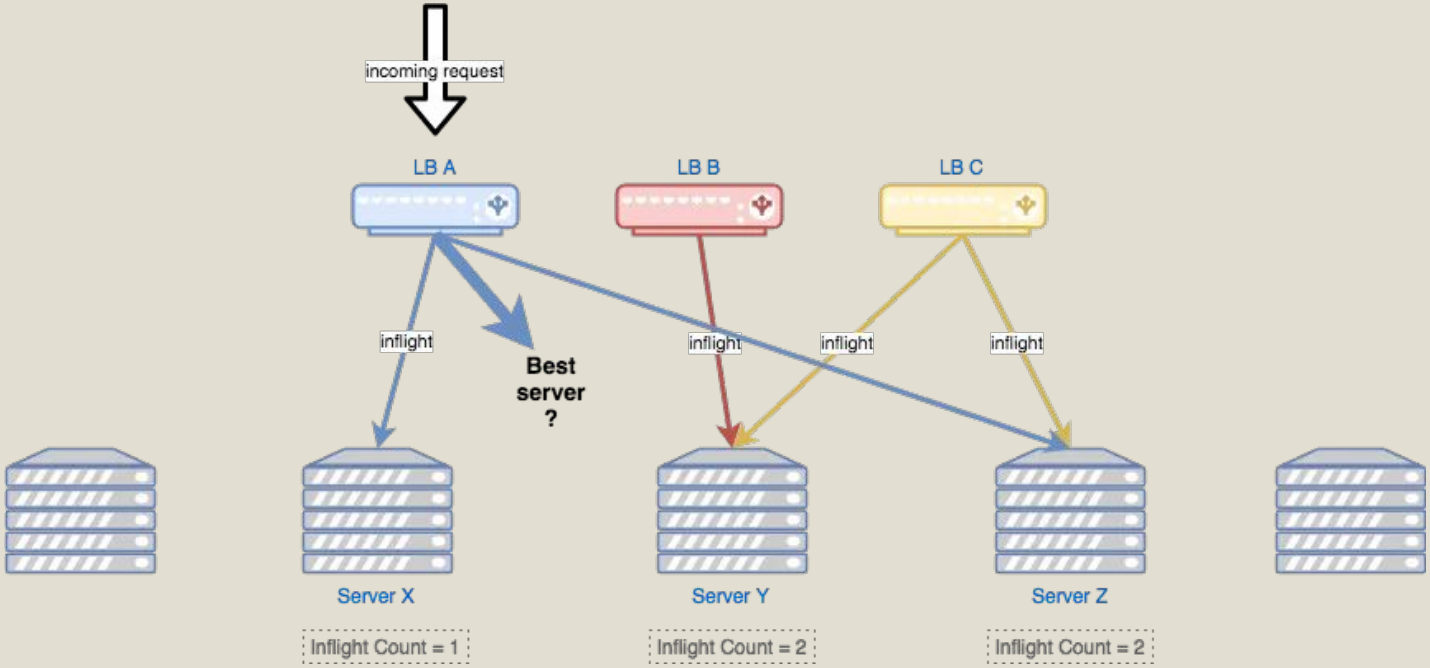
Load Balancing Algorithms Illustrated

https://blog.bytebytego.com/p/ep47-common-load-balancing-algorithms



Load Balancing Algorithms Illustrated

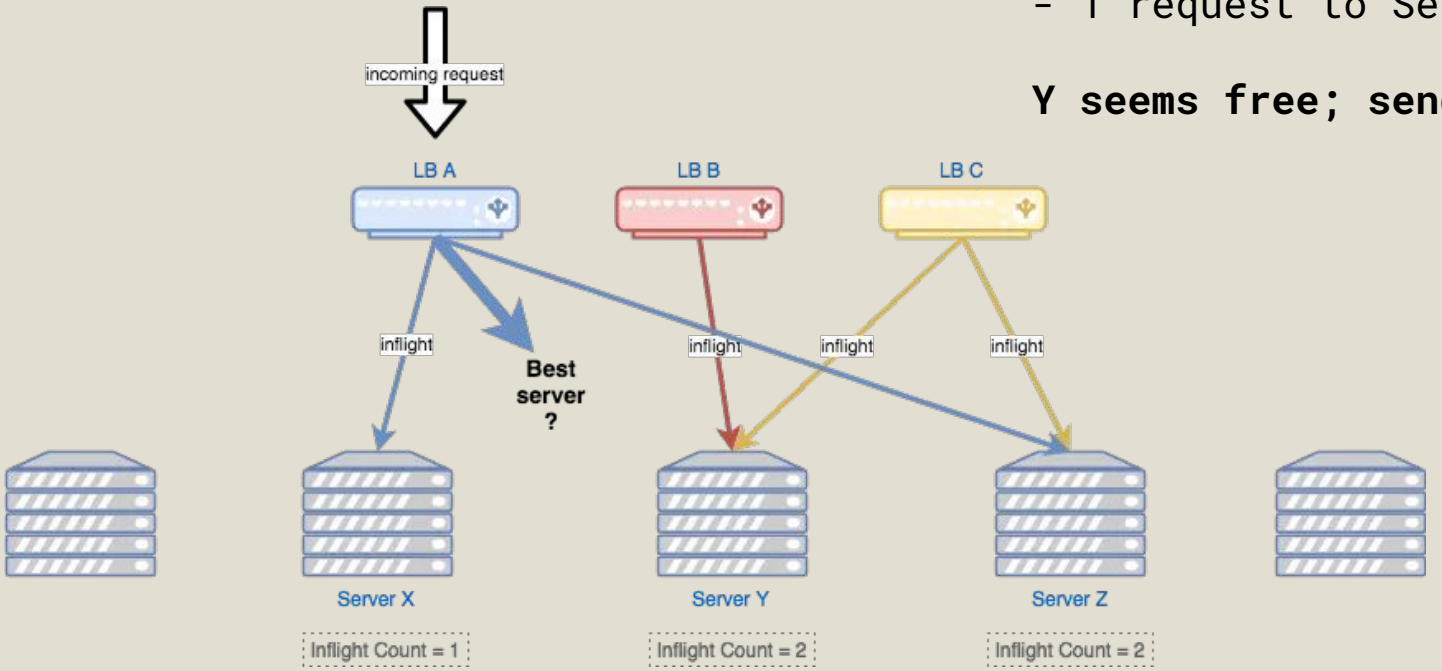
https://blog.bytebytego.com/p/ep47-common-load-balancing-algorithms



Join-the-shortest-queue Algorithm

LB A:
- 1 request to Server X
- 1 request to Server Z

Y seems free; send to Y



JSQ Algorithm at LB A

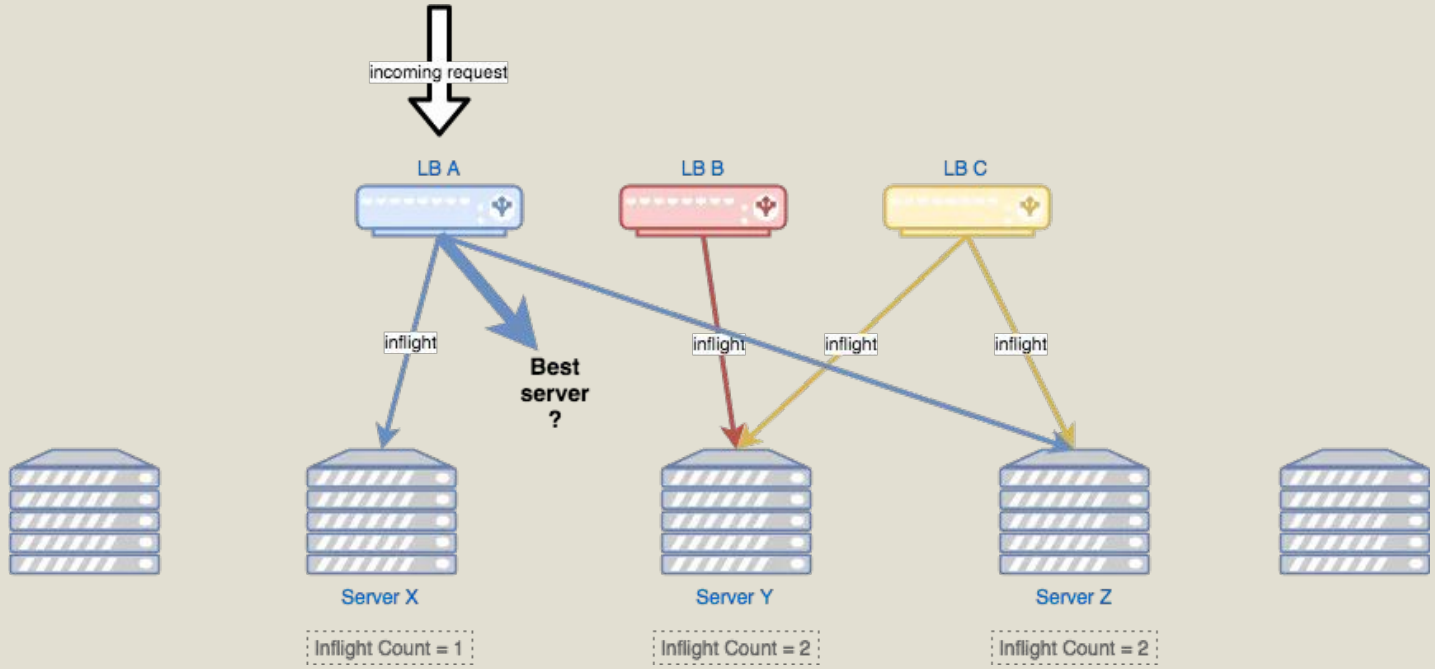
Definition: The Join-the-Shortest-Queue (JSQ) algorithm assigns incoming jobs to the server with the fewest number of jobs in its queue, aiming to balance load across servers

Greedy Nature: JSQ is a greedy algorithm, as it prioritizes minimizing queue length for each incoming job to maximize instantaneous processing speed

Centralized Design: It requires a centralized load balancer that tracks the number of jobs at each server. This eliminates communication overhead since the load balancer already monitors job arrivals and departures

Performance: JSQ achieves excellent load balancing and minimizes response times compared to other algorithms, but it becomes less efficient in distributed systems due to scalability challenges

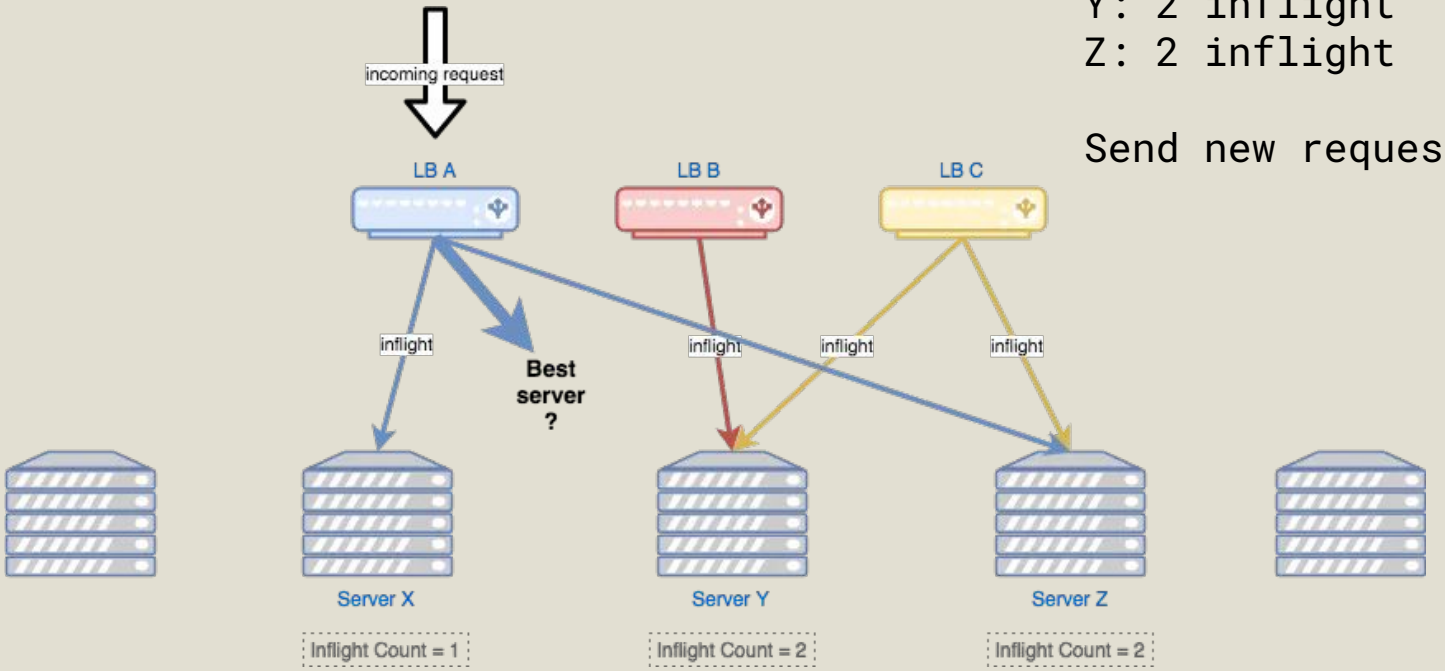
JSQ Algorithm



Company's Challenge: Very Large System

LB A
X: 1 inflight
Y: 2 inflight
Z: 2 inflight

Send new request to X



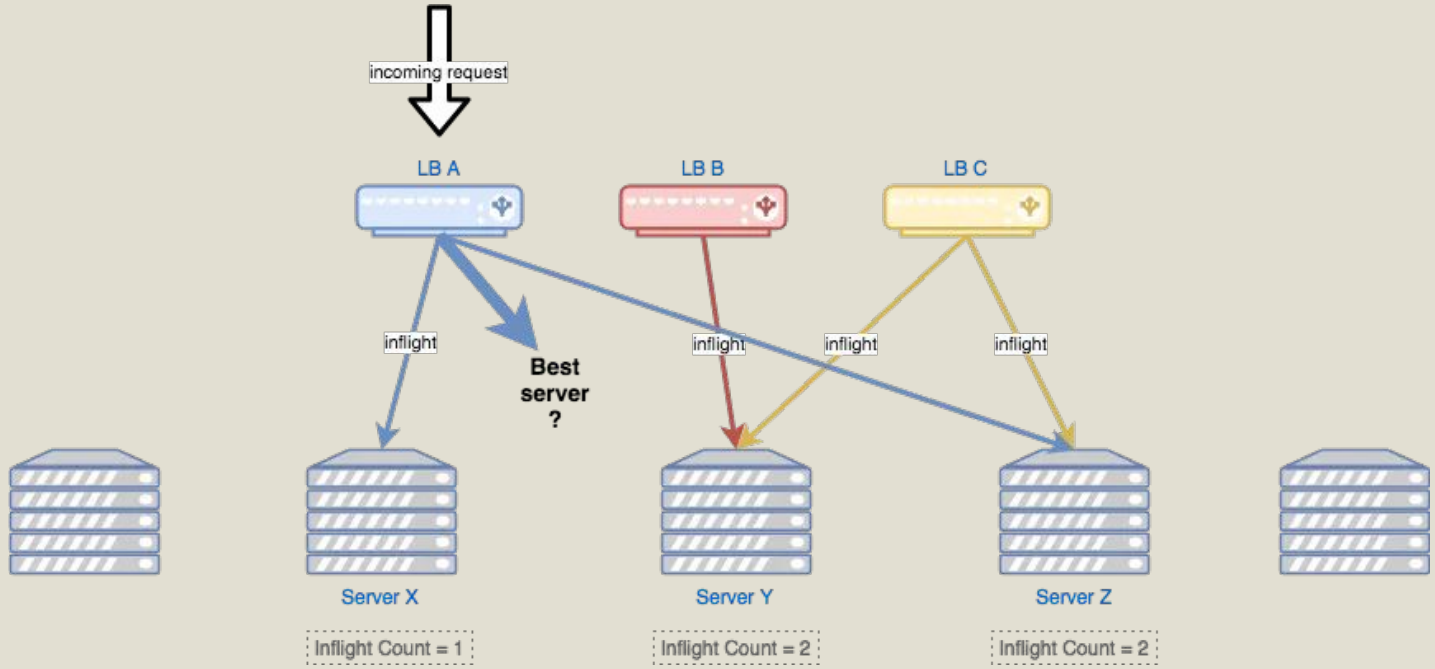
Share State Across Load Balancers?

“One solution to this problem could be to share the state of each load balancers’ inflight counts with all other load balancers … **but then you have a distributed state problem to solve.**

We generally employ distributed mutable state only as a last resort, as the value gained needs to outweigh the substantial costs involved:

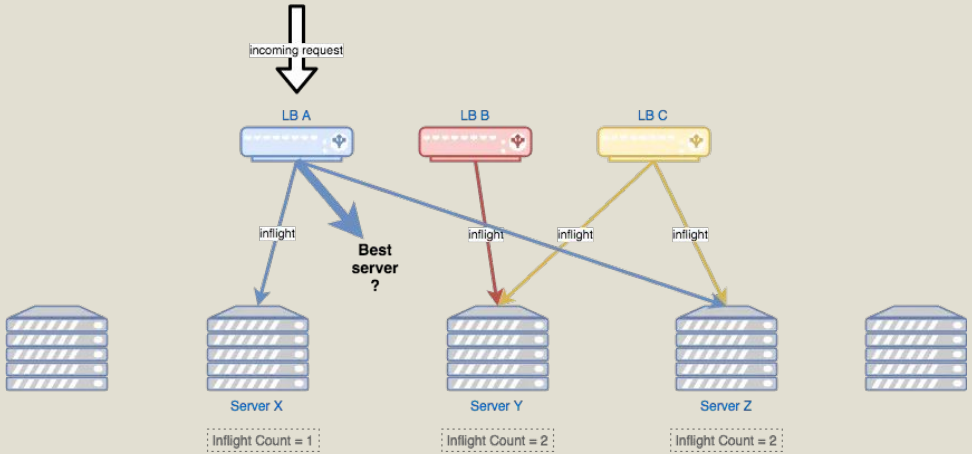
- Operational overhead and complexity it adds to tasks like deployments and canarying.
- Resiliency risks related to the blast radius of data corruptions (ie. bad data on 1% of load balancers is an annoyance, bad data on 100% of them is an outage).
- **The cost of either implementing a P2P distributed state system between the load balancers, or the cost of operating a separate database with the performance and resiliency credentials needed to handle this massive read and write traffic.**”

Load Balancers Between a Distributed Cluster



How would you design a load balancing algorithm?

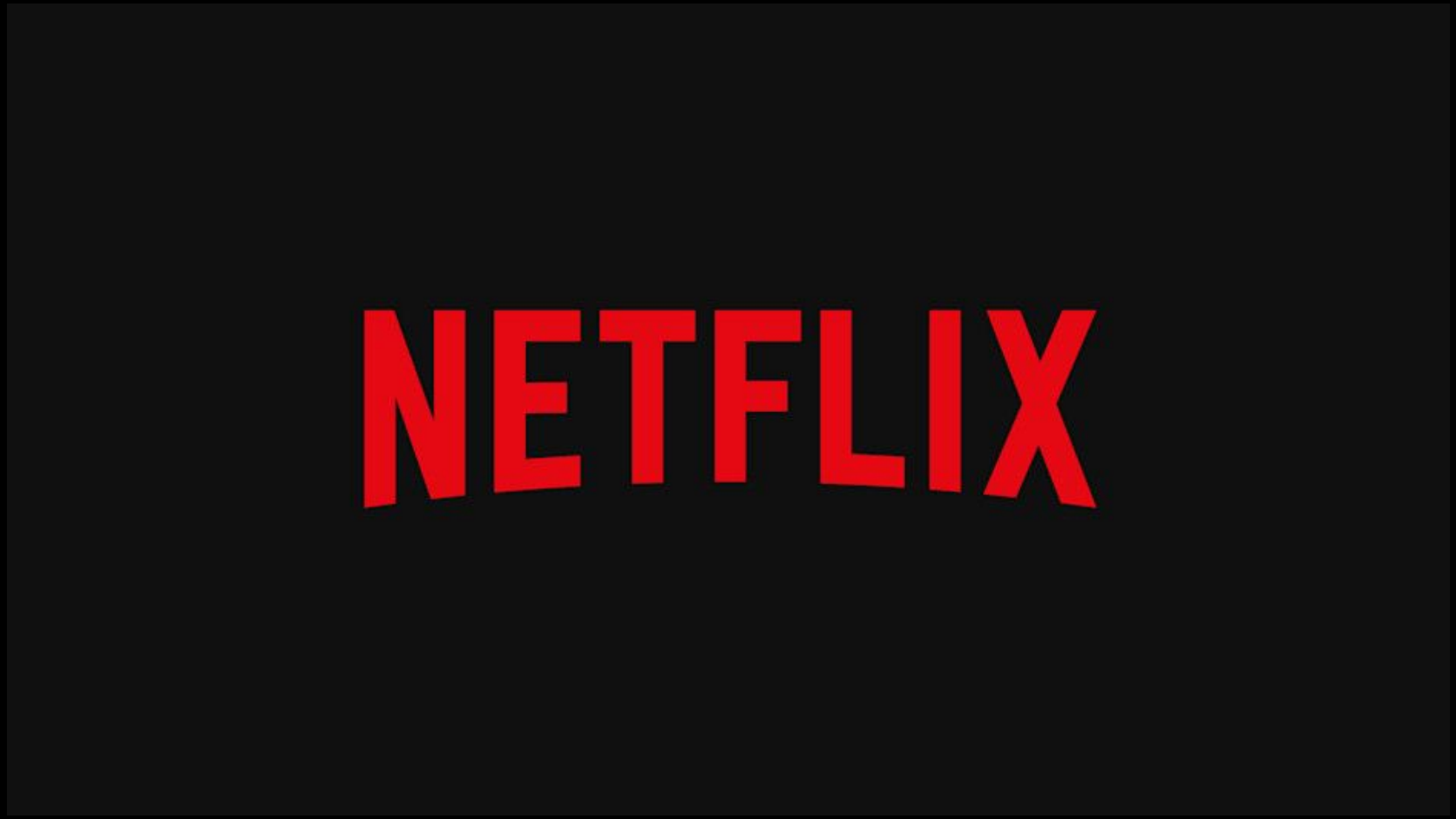
break time



Peer Discussion

Imagine that you are an engineer working with the load balancing problem. Discuss in pairs on how you might design an algorithm that **doesn't use shared state** across all load balancers.

~15 mins, pick 1 person to share after discussing.



https://netflixtechblog.com/netflix-edge-load-balancing-695308b5548c

1. Pick any 2 random servers and compare between them.

2. Choose the “better” option out of them.

Choice-of-2 Algorithm

The solution lies in the “power of two choices” load-balancing algorithm. I**nstead of making the absolute best choice using incomplete data, with “power of two choices” you pick two queues at random and chose the better option of the two, avoiding the worse choice**.

“Power of two choices” is efficient to implement. ... It avoids the undesired herd behavior by the simple approach of avoiding the worst queue and distributing traffic with a degree of randomness.”

Choice-of-2 Algorithm

https://www.f5.com/company/blog/nginx/nginx-power-of-two-choices-load-balancing-algorithm

1. **Client Health**: rolling percentage of connection-related errors for that server.

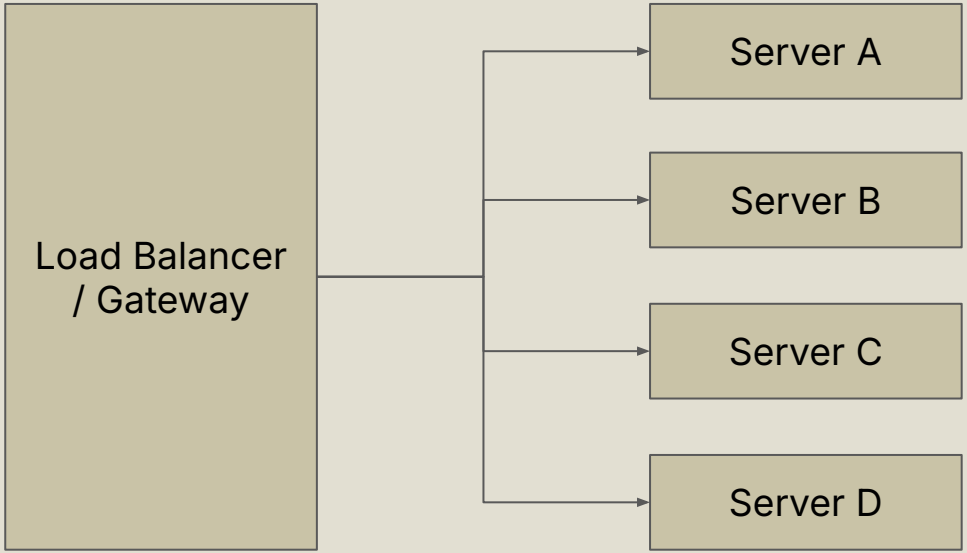
2. **Server Utilization**: most recent score provided by that server.

3. **Client Utilization**: current number of inflight requests to that server from this load balancer.

Choice-of-2 Algorithm: Factors

What about Server Utilization?
Server Utilization: most recent score provided by that server.

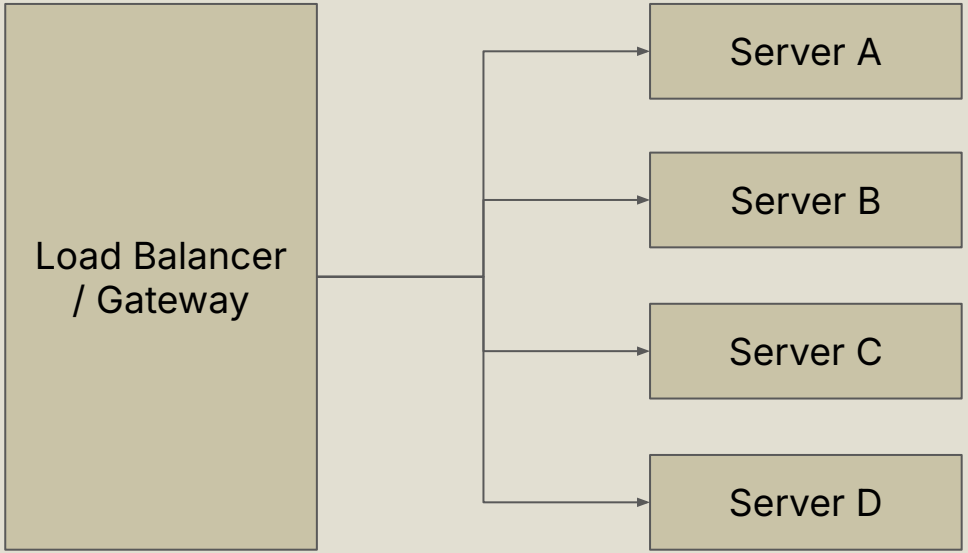
Knows:
- Client Health
- Client Utilization



Source of Information for Comparison

1. Make Request to Server A
2. Server A replies with Response Data + Server Utilization data for Load Balancer

3. Update information for Server A

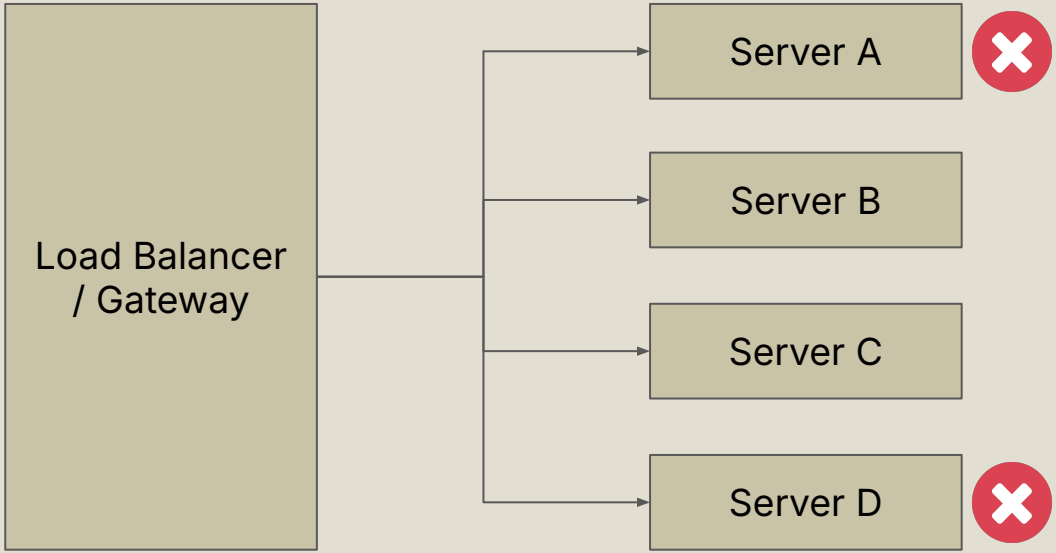


Get Server Utilization With Every Request

Eliminated from comparison if they are too busy or “unhealthy”

Knows:

- Client Health
- Client Utilization
- Server Utilization Info



Filter Servers before Comparison

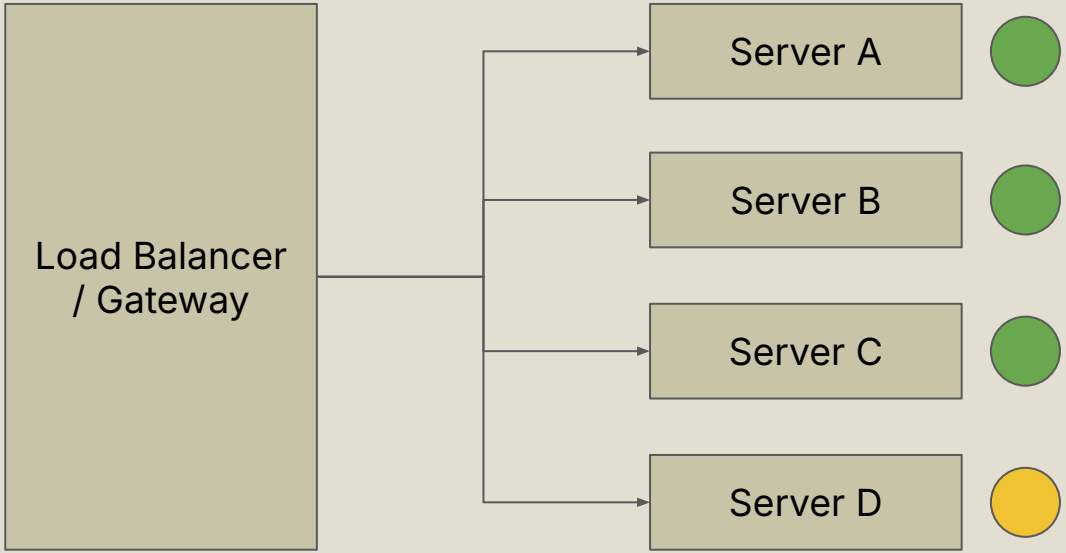
“Filtering like this helps significantly when a large proportion of the pool of servers have persistent problems.

As in that scenario, **randomly choosing 2 servers will frequently result in 2 *bad* servers being chosen for comparison, even though there were many *good* servers available.**”

Filter Servers before Comparison

Server D is new: it will be on Probation.
Only 1 inflight request is allowed until response has been received.

Knows:
- Client Health
- Client Utilization
- Server Utilization Info

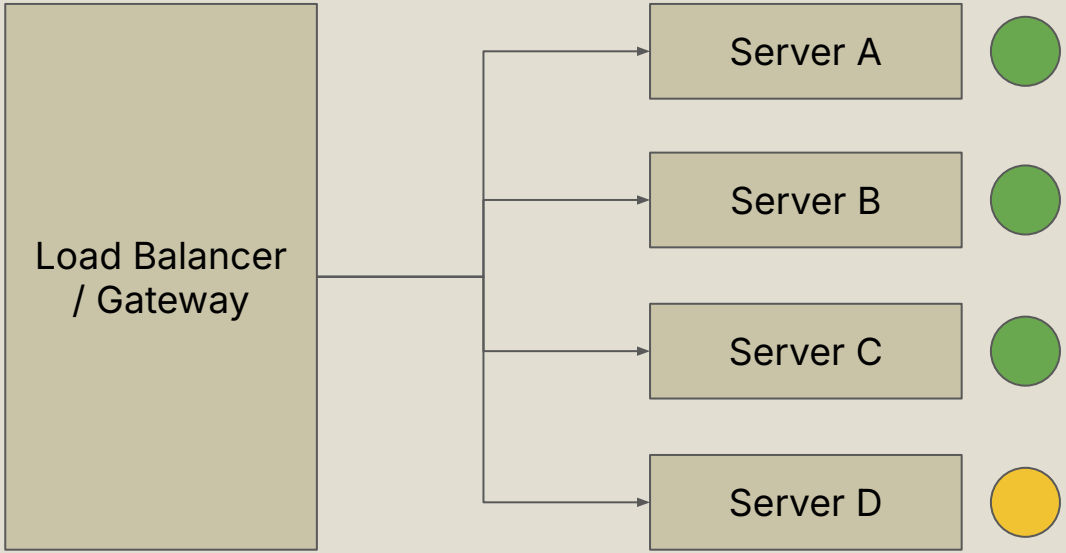


“Probation” for New Servers

Server age: “progressively ramp up traffic to newly launched servers over the course of their first 90 seconds”

Knows:

- Client Health
- Client Utilization
- Server Utilization Info

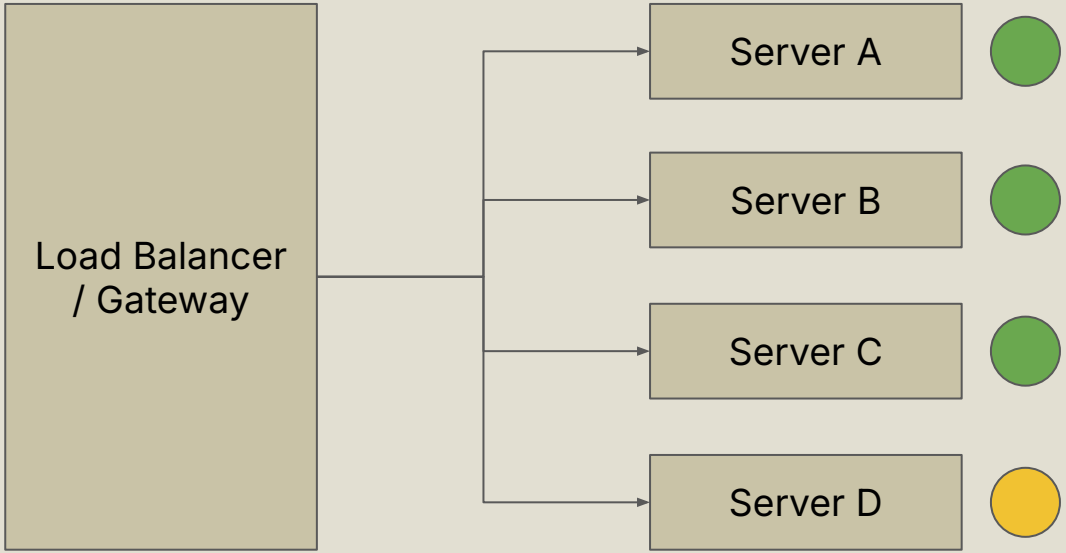


Ramp Up Traffic for New Servers

“To ensure that servers don’t get permanently blacklisted, we apply a decay rate to all stats collected for use in the load-balancing (currently a linear decay over 30 secs)”

Knows:

- Client Health
- Client Utilization
- Server Utilization Info



Stats “Decay”; Servers Don’t Get Permanently Blacklisted

“When clusters are being red-black deployed, if the new server-group has a performance regression, then the proportion of traffic to that group will be less than 50%.

The same effect can be seen with canaries — the baseline may receive a different volume of traffic than the canary cluster. So when looking at metrics, its best to look at the combination of RPS and CPU (as for example RPS may be lower in the canary, while CPU is the same).

Less effective *outlier detection* — we typically have automation that monitors for outlier servers (typically a VM thats slow immediately from launch due to some hardware issue) within a cluster and terminates them. When those outliers receive less traffic due to the load-balancing, this detection is more difficult.”

Tradeoffs

Implementation	Total Failed Request Count	Avg Latency Kept Below (ms)
New w/ all features enabled	5	375
New w/ server-utilization disabled	123	1200
Original implementation	14,700	1400

Metrics Before and After Implementation

Netflix's Zuul

https://github.com/Netflix/zuul

7.3 developer experience case study

Apache HTTP Server

https://httpd.apache.org/

```
# apache.conf or .htaccess
<VirtualHost *:80>
    ServerName opencamp.cc
    Redirect permanent / https://opencamp.cc/
</VirtualHost>

<VirtualHost *:443>
    ServerName opencamp.cc
    DocumentRoot /var/www/html

    SSLEngine on
    SSLCertificateFile /etc/letsencrypt/live/opencamp.cc/cert.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/opencamp.cc/privkey.pem
    SSLCertificateChainFile /etc/letsencrypt/live/opencamp.cc/chain.pem

    # SSL Configuration
    SSLProtocol all -SSLv3 -TLSv1 -TLSv1.1
    SSLHonorCipherOrder on
    SSLCompression off
    SSLSessionTickets off

    <Directory /var/www/html>
        Options -Indexes +FollowSymLinks
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

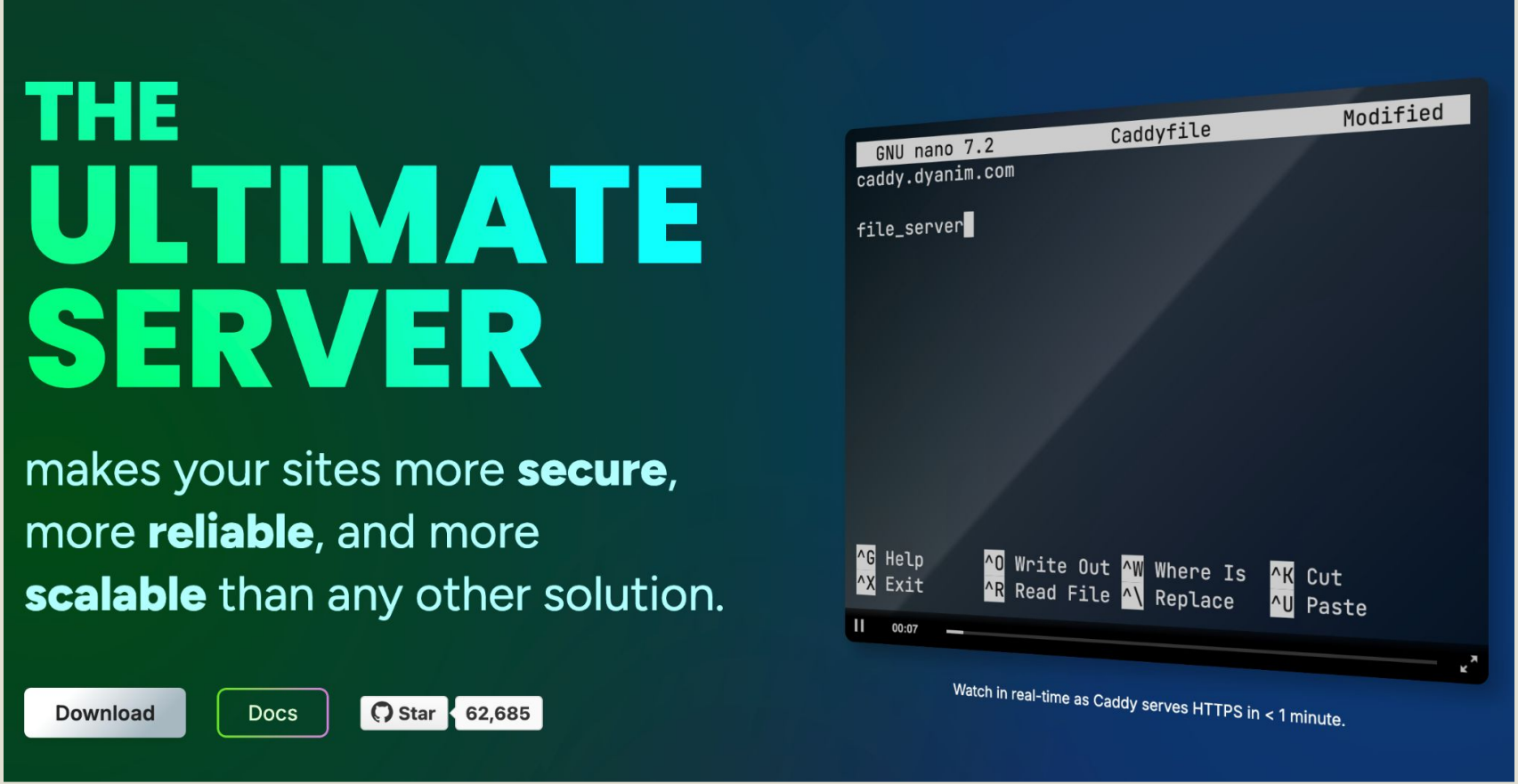
Sample Apache Config for opencamp.cc

nginx

https://nginx.org/

```
# nginx.conf
server {
    listen 80;
    server_name opencamp.cc;
    location / {
        return 301 https://$host$request_uri;
    }
}

server {
    listen 443 ssl;
    server_name opencamp.cc;

    ssl_certificate /etc/letsencrypt/live/opencamp.cc/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/opencamp.cc/privkey.pem;

    # SSL configuration
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers on;
    ssl_ciphers
ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:ECDHE-RSA-AES256-GCM-SHA
384;
    ssl_session_timeout 1d;
    ssl_session_cache shared:SSL:10m;

    location / {
        root /var/www/html;
        index index.html;
    }
}
```

Sample nginx Config for opencamp.cc



Caddy Homepage

https://caddyserver.com/

```
# Caddyfile
opencamp.cc {
    root * /var/www/html
    file_server
}
```

Sample Caddy Config for opencamp.cc

Metric	Description
Configuration Complexity	The burden of setup expressed through lines of code, steps required, and time to implement a solution or tool.
Cognitive Load Analysis	The conceptual frameworks and domain knowledge one must understand before they can effectively use a tool or system.
Mental Models Required	The assessment of mental effort required to learn, use, and troubleshoot a system, including the number of concepts to juggle simultaneously.
"Time to X" Metric	How long it takes to go from 0 to achieve an outcome or a goal.

Some Metrics for Measuring Developer Experience

Server	Configuration Lines	Additional Steps	Time to Deploy
Apache	~25 lines	6 steps	~30-45 minutes
Nginx	~20 lines	5 steps	~20-30 minutes
Caddy	3 lines	3 steps	~5 minutes

Configuration Complexity

Apache & Nginx:

- Understanding of HTTP and HTTPS protocols
- Knowledge of SSL/TLS certificate types and chain of trust
- Familiarity with cipher suites and security best practices
- Understanding of web server directives and syntax
- Knowledge of file permissions and webroot concepts

Caddy:

- Basic understanding of domain names
- Knowledge of file serving concepts

Mental Models Required

Factor	Apache	Nginx	Caddy
Configuration syntax to learn	High	Medium	Very Low
Security decisions required	Many	Many	None
Potential for errors	High	High	Low
Maintenance burden	High	High	Minimal
Documentation needed	Extensive	Extensive	Minimal

Cognitive Load Analysis

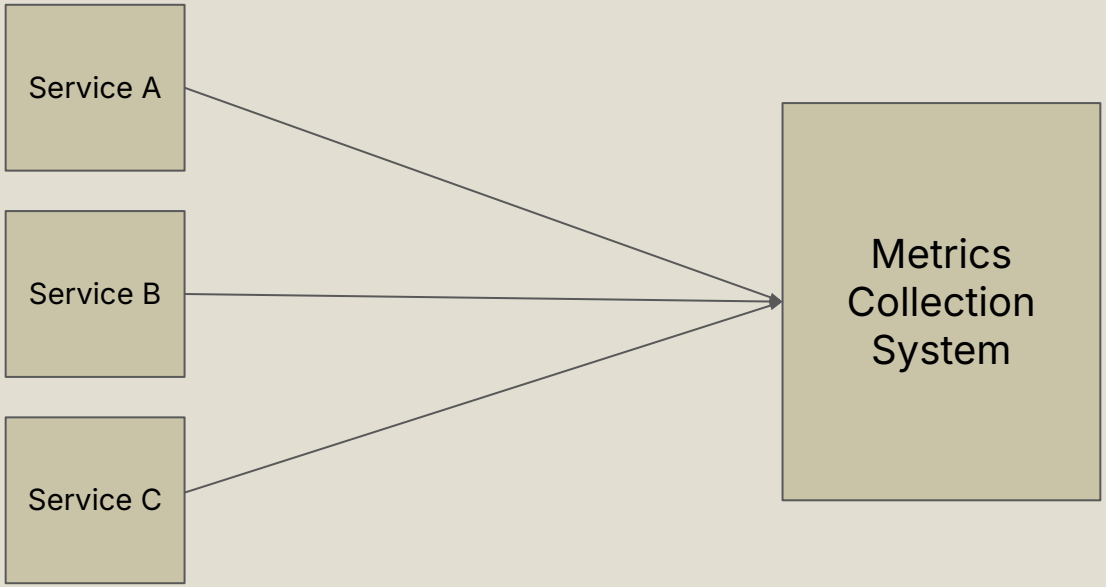
How long it takes a new developer to go from zero to a working HTTPS site:

- Apache: 30-60 minutes (experienced), 2-3 hours (novice)
- Nginx: 20-45 minutes (experienced), 1-2 hours (novice)
- Caddy: 5-10 minutes (experienced), 15-30 minutes (novice)

Time to Display “Index Page” For a Novice

Metric	Description
Configuration Complexity	The burden of setup expressed through lines of code, steps required, and time to implement a solution or tool.
Cognitive Load Analysis	The conceptual frameworks and domain knowledge one must understand before they can effectively use a tool or system.
Mental Models Required	The assessment of mental effort required to learn, use, and troubleshoot a system, including the number of concepts to juggle simultaneously.
"Time to X" Metric	How long it takes to go from 0 to achieve an outcome or a goal.

Some Metrics for Measuring Developer Experience



Example: Metrics Collection System

1. HTTP API:

- Provide OpenAPI specifications
- Provide documentation on how to use, eg. authentication, retries mechanisms, etc.

Good DX First Stab

2. Shared SDK Library:

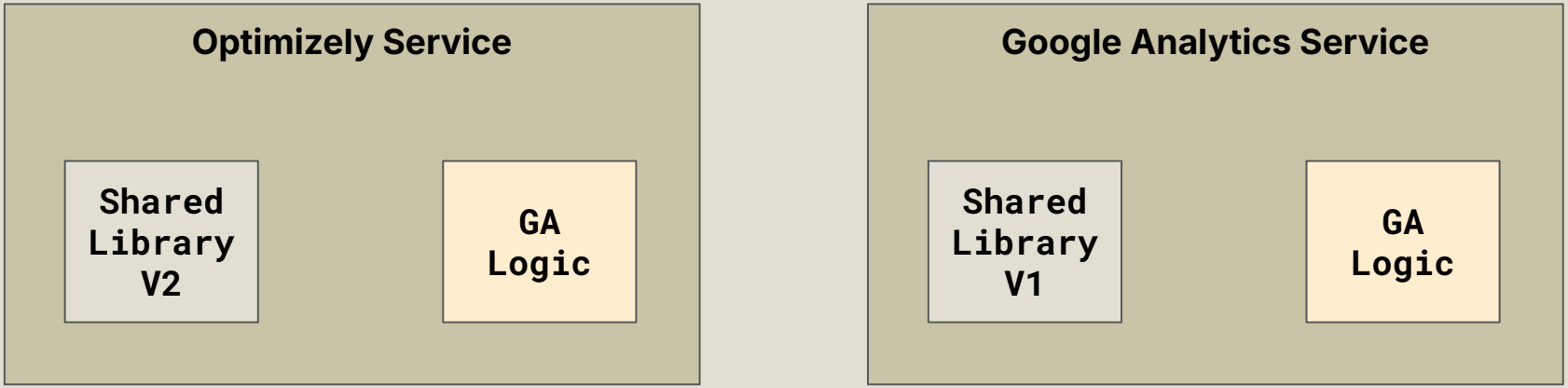
- Provide an SDK that can be easily installed and used, with automatic configuration and instrumentation
- SDK automatically sets up the settings, right retry policies, etc.

Good DX Second Attempt

3. SDK with Test Environments:

- SDK allows metrics to be captured in test environments, allowing tracking behaviour to be verified
- Easy debugging support during development by enabling logs

Good DX Third Attempt



Segment: Updating Shared Libs Was Painful

Good Developer Experience is a constant trade-off between productivity, flow, and centralization.

Too much centralization can result in slower workflows, but a good amount can help accelerate other teams / services so we minimize the amount of duplicate work.

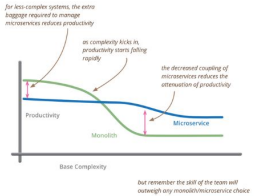
“Shared services focused on application delivery velocity allow software to ship more rapidly with minimal risk.”

Good DX: Trade-offs

https://www.hashicorp.com/en/resources/platform-teams-best-practices

optional readings

Microservice Premium



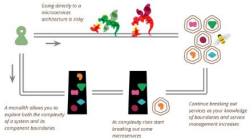
The microservices architectural style has been the hot topic over the last year. At the recent O'Reilly software architecture conference, it seemed like every session talked about microservices. Enough to get everyone's over-hyped-bullshit detector up and flashing. One of the consequences of this is that we've seen teams be too eager to embrace microservices, not realizing that microservices introduce complexity on their own account. This adds a premium to a project's cost and risk - one that often gets projects into serious trouble.

by Martin Fowler BLIKI 13 May 2015

Read more...

◇MICROSERVICES

Monolith First



As I hear stories about teams using a microservices architecture, I've noticed a common pattern.

1. Almost all the successful microservice stories have started with a monolith that got too big and was broken up
2. Almost all the cases where I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble.

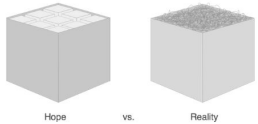
This pattern has led many of my colleagues to argue that **you shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile.** .

by Martin Fowler BLIKI 3 Jun 2015

Read more...

◇EVOLUTIONARY DESIGN ◇MICROSERVICES

Don't start with a monolith



In the last few months, I've heard repeatedly that the only way to get to a successful microservices architecture is by starting with a monolith first. To paraphrase Simon Brown: If you can't build a well-structured monolith, what makes you think you can build a well-structured set of microservices? The most recent – and, as usual, very convincing – rendering of this argument comes from Martin Fowler on this very site. As I had a chance to comment on an earlier draft, I had some time to think about this. And I did, especially because I usually find myself in agreement with him, and some others whose views I typically share seemed to agree with him, too.

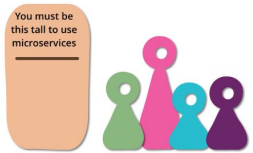
I'm firmly convinced that starting with a monolith is usually exactly the wrong thing to do.

by Stefan Tilkov ARTICLE 9 Jun 2015

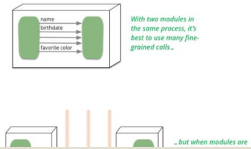
Read more...

◇MICROSERVICES

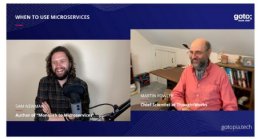
Microservice Prerequisites



Microservices and the First Law of Distributed Objects



Interview with Sam Newman about Microservices



Microservices Guide

https://martinfowler.com/microservices/

Building Infrastructure Platforms

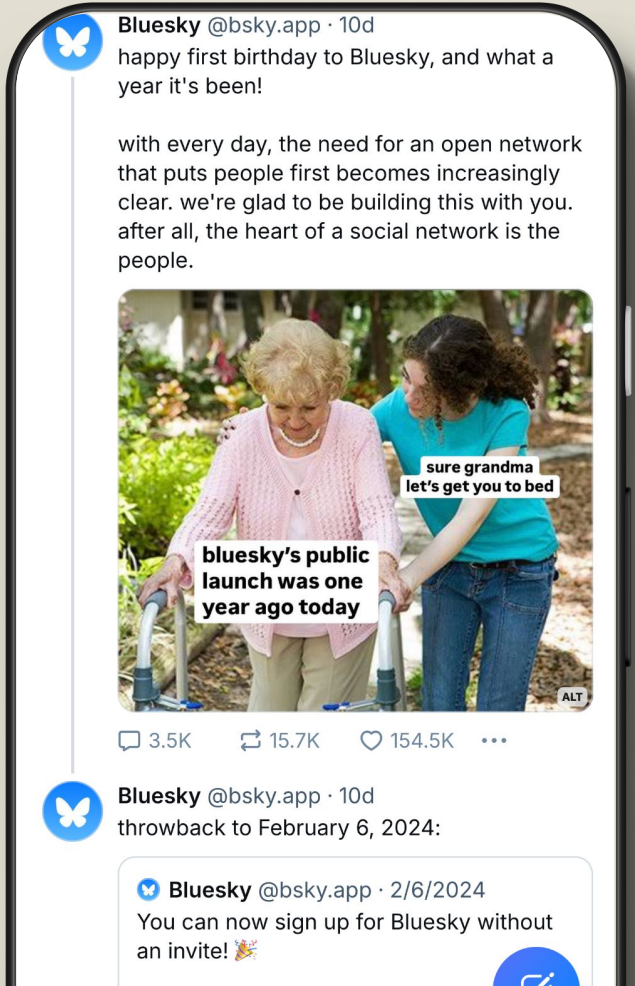
Infrastructure Platform teams enable organisations to scale delivery by solving common product and non-functional requirements with resilient solutions. This allows other teams to focus on building their own things and releasing value for their users. But nobody said building scalable platforms was easy... In this article Poppy and Chris explore 7 key principles which can help you build the right thing right. Spoiler: don't skip strategy, user experience and research.

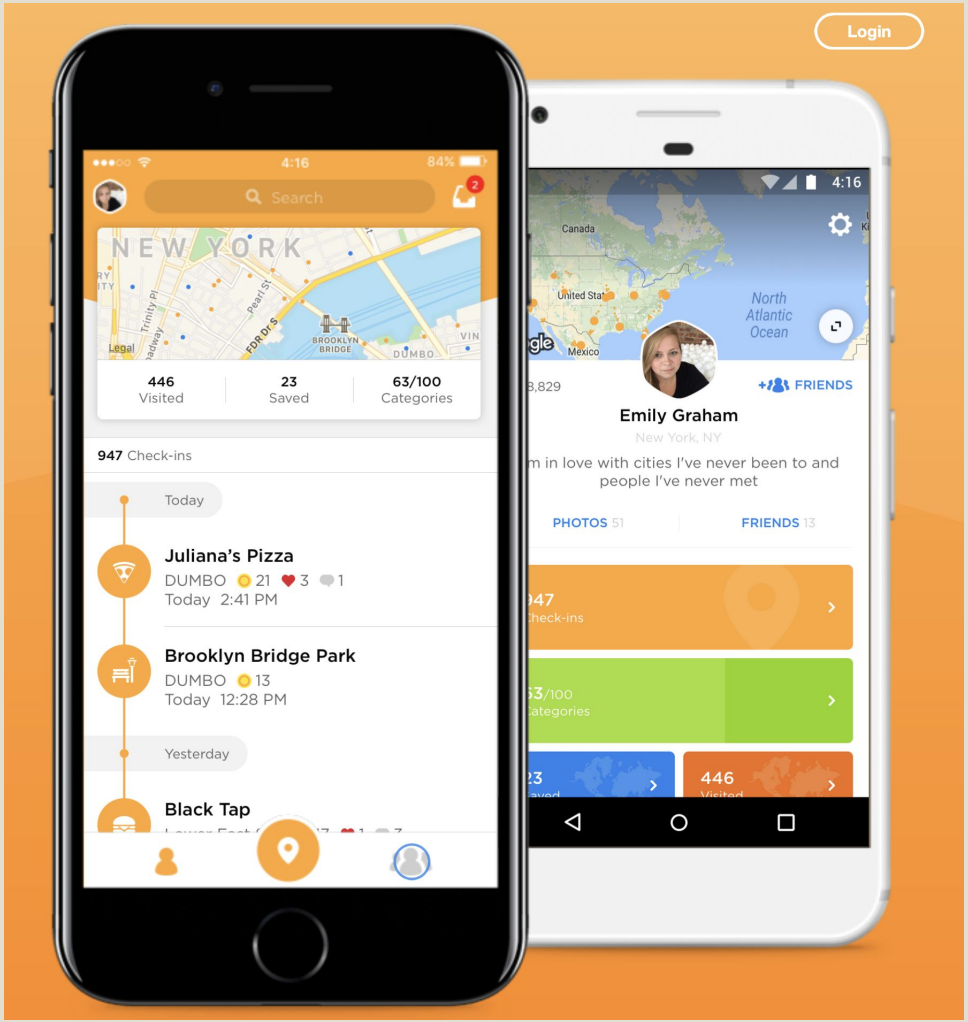
Building Infrastructure Platforms

https://martinfowler.com/articles/building-infrastructure-platform.html

Assignment

Extend a Distributed Social Media Platform




Login
4:16
84%
Search
NEW YORK
446
Visited
23
Saved
63/100
Categories
947 Check-ins
Today
Juliana's Pizza
DUMBO 21 3 1
Today 2:41 PM
Brooklyn Bridge Park
DUMBO 13
Today 12:28 PM
Yesterday
Black Tap
4:16
Canada
North Atlantic Ocean
Mexico
FRIENDS
Emily Graham
New York, NY
PHOTOS 51
FRIENDS 13
446

Assignment

W1 → Distributed Social Media Research
W2 → Review W1 + Build PoC
W3 → Design 1st Draft of System + Sharing
W4 → Implementation
W5 → Implementation (Queues)
W6 → Implementation (Load Testing / Observability)
W7 → Technical Retrospective
W8 → Complete System Implementation

Assignment #6

- Prepare a set of slides to share your progress on the project so far. Be sure to talk about:
  - Your project's progress
  - What you have learned from experimenting with the tools
  - Decisions that you have made, whether they were good or bad in hindsight
  - Things you wish you have done earlier or later
  - If you were to do this project again, what would you have done differently?

- You will be sharing during Week 8, so no submission for this week.