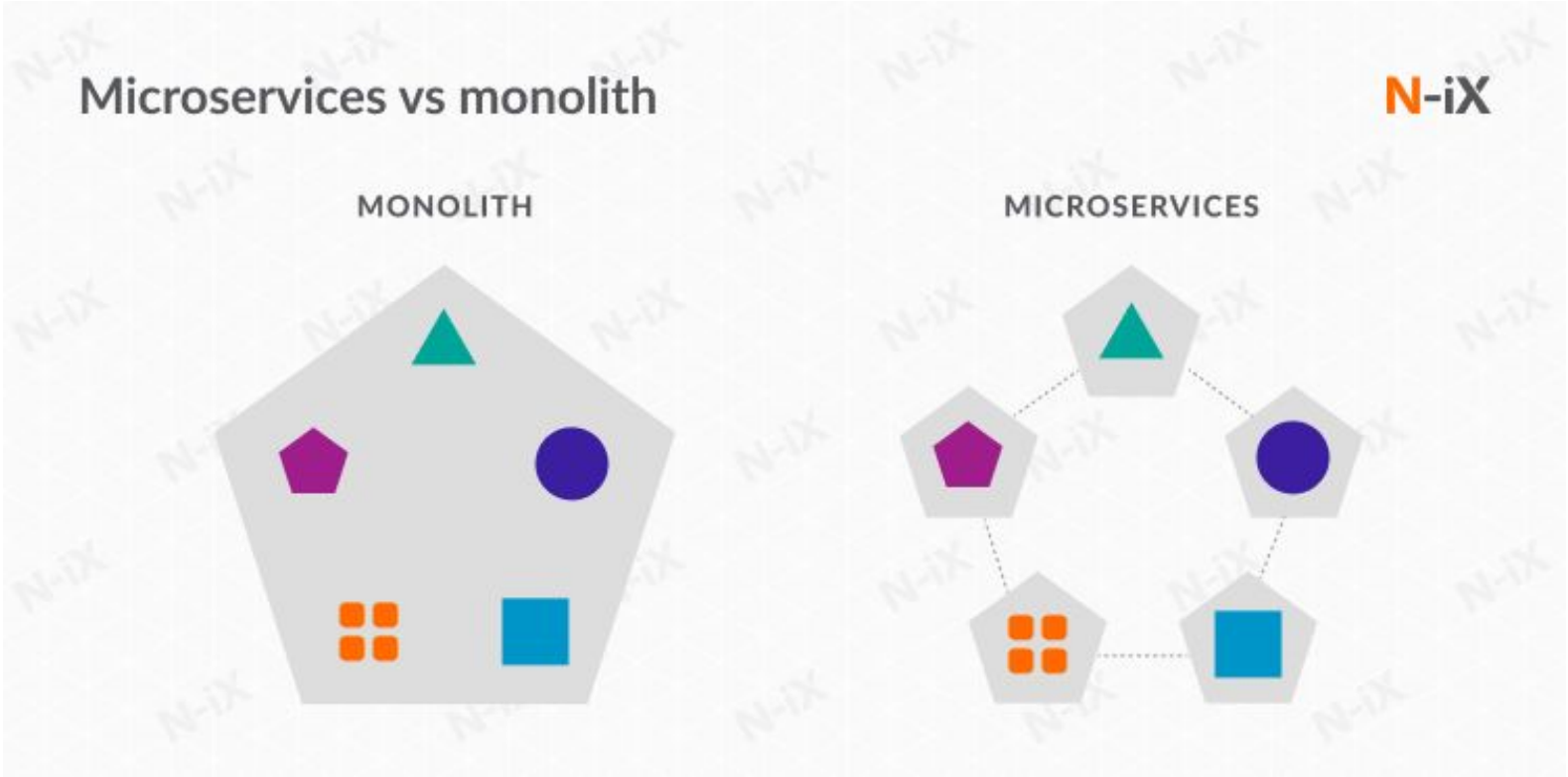
Monolith and Microservice

...

Intro: Monolith vs Microservice



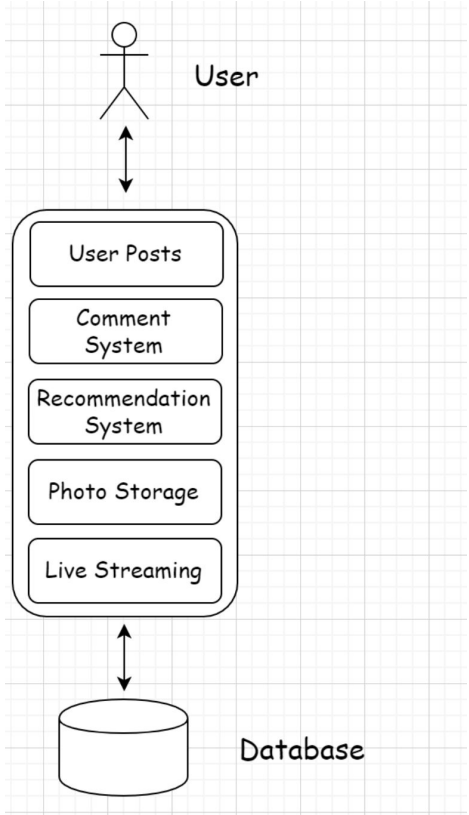
What is Monolith

- A monolithic application is a self-contained, tightly-coupled software application.
- Example: Facebook contains various features such as:
  - User posts
  - Comment system
  - Groups
  - Marketplace
  - Portal Ads
  - Photo storage
  - Live streaming
  - Recommendation system for recommending the latest and contextual content

What is Monolith

- In a monolithic architecture, all the modules will be:
  - Coded in a single codebase
  - Tightly coupled with each other (as opposed to having one or more dedicated microservice for running respective features.)
  - Deployed together as one unit
- Codebase: whether you have separate repos
- Deployment: how you send code to production

What is Monolith



Upsides of Monolith

- Monolithic apps are simpler to build, test, and deploy
- Less code duplication
- Shared tooling easier
- During the initial stages of a business, teams often choose monolithic architecture
    - intending to branch out into a distributed microservices architecture later.

Downsides of Monolith

- Continuous deployment
  - Continuous deployment is a pain in monolithic applications
  - Minor code change in a certain application layer requires re-deployment of entire app

Downsides of Monolith

- Continuous deployment
  - Continuous deployment is a pain in monolithic applications as even a minor code change in a certain application layer or a feature necessitates a re-deployment of the entire application.
- Regression testing
  - A minor code change in one feature can potentially impact the functionality of other features
  - All the features are tightly coupled with each other.

Downsides of Monolith

- Continuous deployment
  - Continuous deployment is a pain in monolithic applications as even a minor code change in a certain application layer or a feature necessitates a re-deployment of the entire application.
- Regression testing
  - A minor code change in one feature can potentially impact the functionality of other features
  - All the features are tightly coupled with each other.
- Single points of failure
  - Monolithic applications have a single point of failure. A bug in any of the application features can bring down the entire application.

Downsides of Monolith

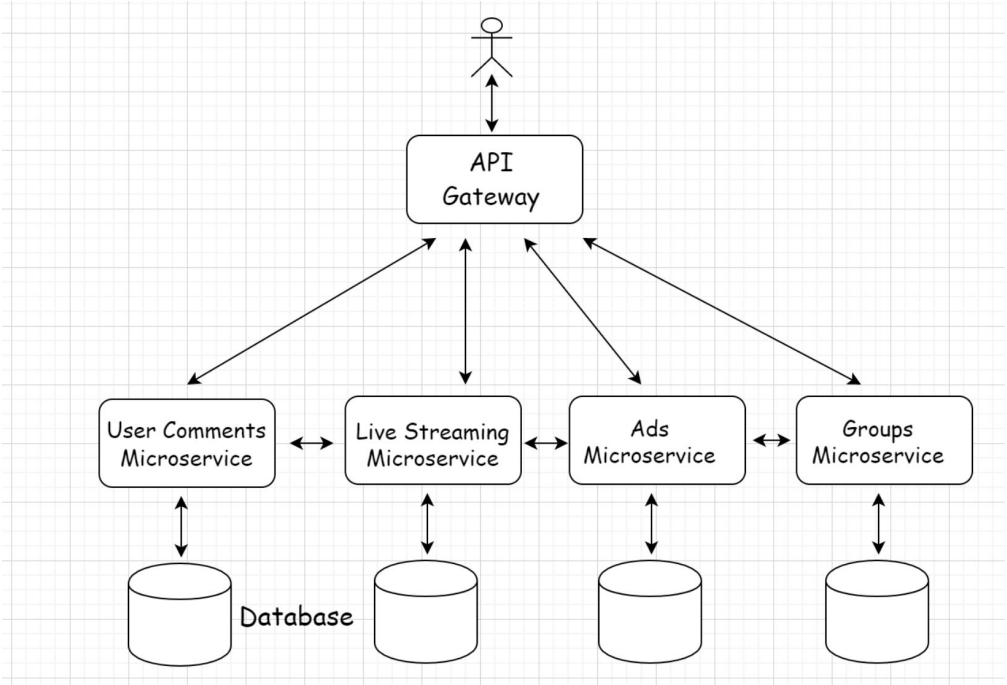
- Continuous deployment
  - Continuous deployment is a pain in monolithic applications as even a minor code change in a certain application layer or a feature necessitates a re-deployment of the entire application.
- Regression testing
  - A minor code change in one feature can potentially impact the functionality of other features
  - All the features are tightly coupled with each other.
- Single points of failure
  - Monolithic applications have a single point of failure. A bug in any of the application features can bring down the entire application.
- Scalability issues
  - Maintenance and scalability are a challenge in monolith apps as all the components are so tightly coupled with each other.
  - As the code size increases, things get trickier to manage.

Downsides of Monolith

- Continuous deployment
  - Continuous deployment is a pain in monolithic applications as even a minor code change in a certain application layer or a feature necessitates a re-deployment of the entire application.
- Regression testing
  - A minor code change in one feature can potentially impact the functionality of other features
  - All the features are tightly coupled with each other.
- Single points of failure
  - Monolithic applications have a single point of failure. A bug in any of the application features can bring down the entire application.
- Scalability issues
  - Maintenance and scalability are a challenge in monolith apps as all the components are so tightly coupled with each other.
  - As the code size increases, things get trickier to manage.
- Harder to leverage heterogeneous technologies
  - Different versioning of packages or conflicting requirements

What is Microservice Architecture?

- Different features deployed separately as services
- They work in conjunction to form a large distributed online service as a whole.
- Every service has a single responsibility of running a specific feature and is separated from other services



Pros of Microservice

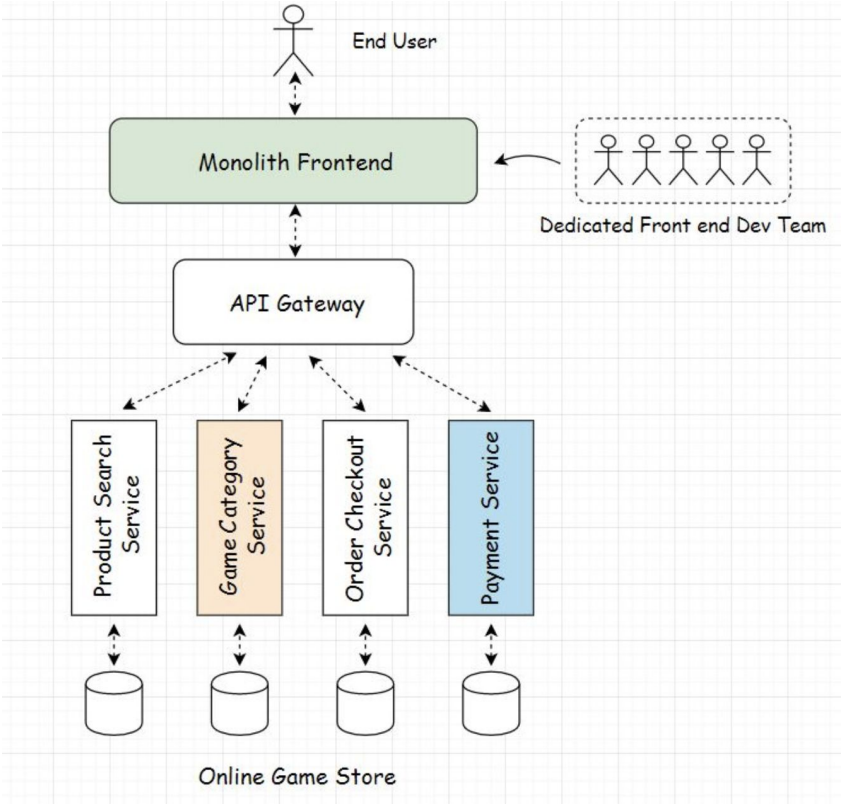
- Scalability more granular
  - Services that need scaling can be scaled independently without affecting other services.
- Fewer single points of failure
  - This eliminates single points of failure and system bottlenecks.
- Independent and continuous deployments
  - We can have dedicated teams for every microservice, and they can be deployed independently
- Dev Team autonomy
- Tailored Scalability
- Faster release of new features

Cons of Microservice

- Management complexity
  - Microservices is a distributed environment with several services powered by clusters of servers. This makes system management and monitoring complex.
- We need to set up additional components to manage microservices
  - Requires more DevOps work
- Code sharing can be harder
  - Depending if it’s separate repos or just separate services

Trend: Micro Frontends

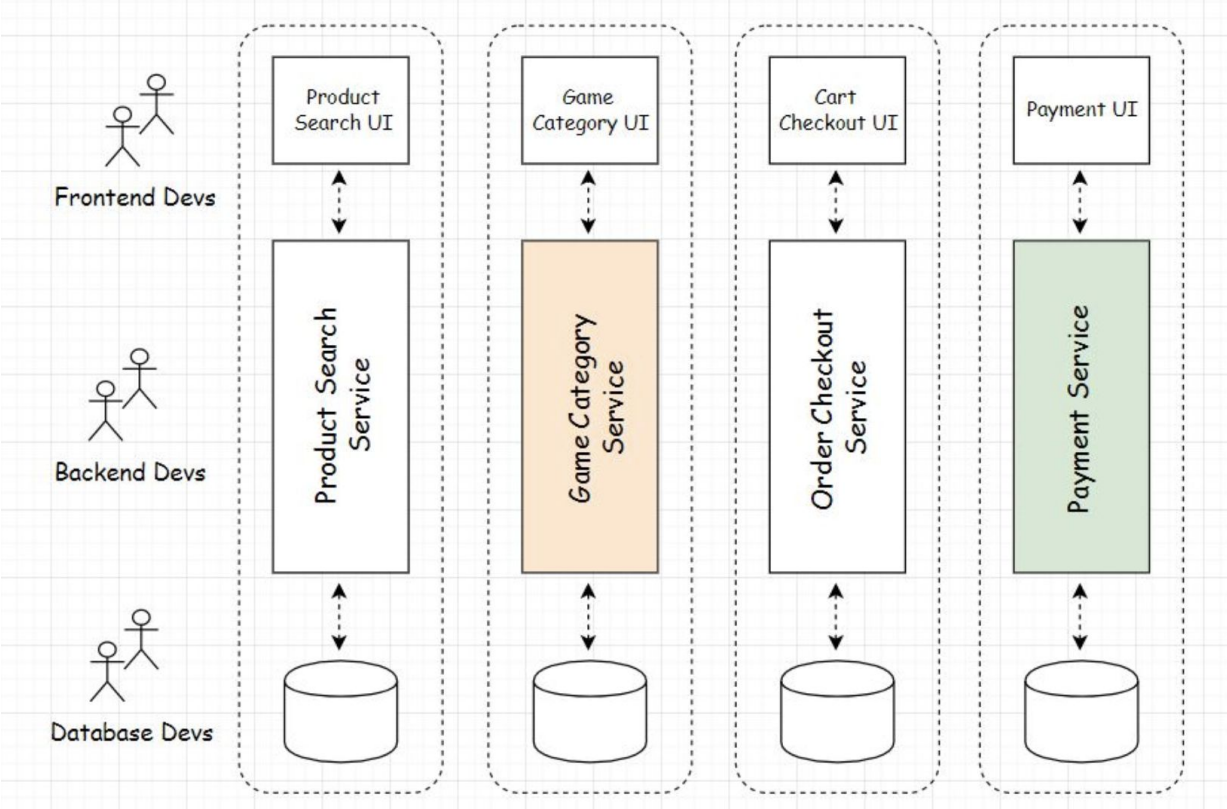
- Typically, in applications:
  - our frontend is a monolith that is developed by a dedicated frontend development team.
  - Backend trend is microservices but until recently, not reached frontend



Trend: Micro Frontends

- With the micro frontends approach, we split our application into vertical slices
  - A single slice goes end to end from the user interface to the database.
  - Every slice is owned by a dedicated team.
- Every team builds their user interface component choosing their desired technology, and later all these components are integrated, forming the complete user interface of the application.
  - This micro frontend approach averts the need for a dedicated centralized user interface team.
  - Every micro frontend team becomes more of a full-stack team.

Trend: Micro Frontends



Trend: Micro Frontends

- A micro frontend team may own:
  - a more extensive UI component like the checkout page.
  - a minor component that fits in a particular component
    - like the game category component on the home page
- The smaller components that integrate into other pages/components of the application are known as **fragments**.

Pros: Micro Frontends

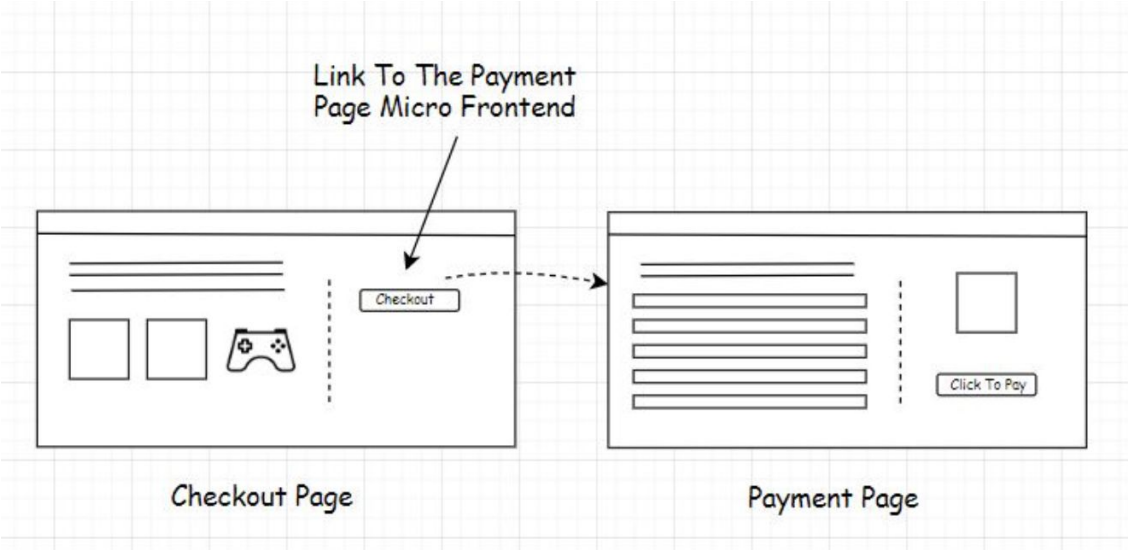
- Easier coordination between the frontend and the backend devs
- Flexibility re: which technology and tools
    - Since the micro frontends are loosely coupled, we can develop them leveraging different technologies, just like microservices.
    - One can use React, other Vue

Cons: Micro Frontends

- Added complexity
  - Only fit for medium to large websites.
  - Overkill for simple use cases
- Need to write additional code to combine all the components built with heterogeneous tech.
- Have to deal with compatibility and performance issues when using multiple technologies together

Micro Frontends: Client Side Integration

- Rendering micro frontends with unique links.
  - basic naïve way of integrating components on the client
  - We just place the links on the website to enable the user to navigate to a certain micro frontend



Micro Frontends: Client Side Integration

- Use iframes: mostly obsolete
  - Aren’t good from an SEO standpoint
  - Have stability and performance issues and so on
- SingleSPA or competitors
  - JavaScript framework for frontend microservices
  - Enables developers to build their frontend while leveraging different JavaScript frameworks.

Micro Frontends: Server Side Integration

- Upon user request, the complete pre-built page of the website is delivered to the client from the server
  - Not sending individual micro frontends to the client and combining at client.
- Can cut down the website's loading time on the client significantly
  - browser does not have to do any sort of heavy lifting.
- More-complex server logic
- Frameworks that facilitate server-side integration of micro frontends:
  - Project Mosaic, Open Components and Podium

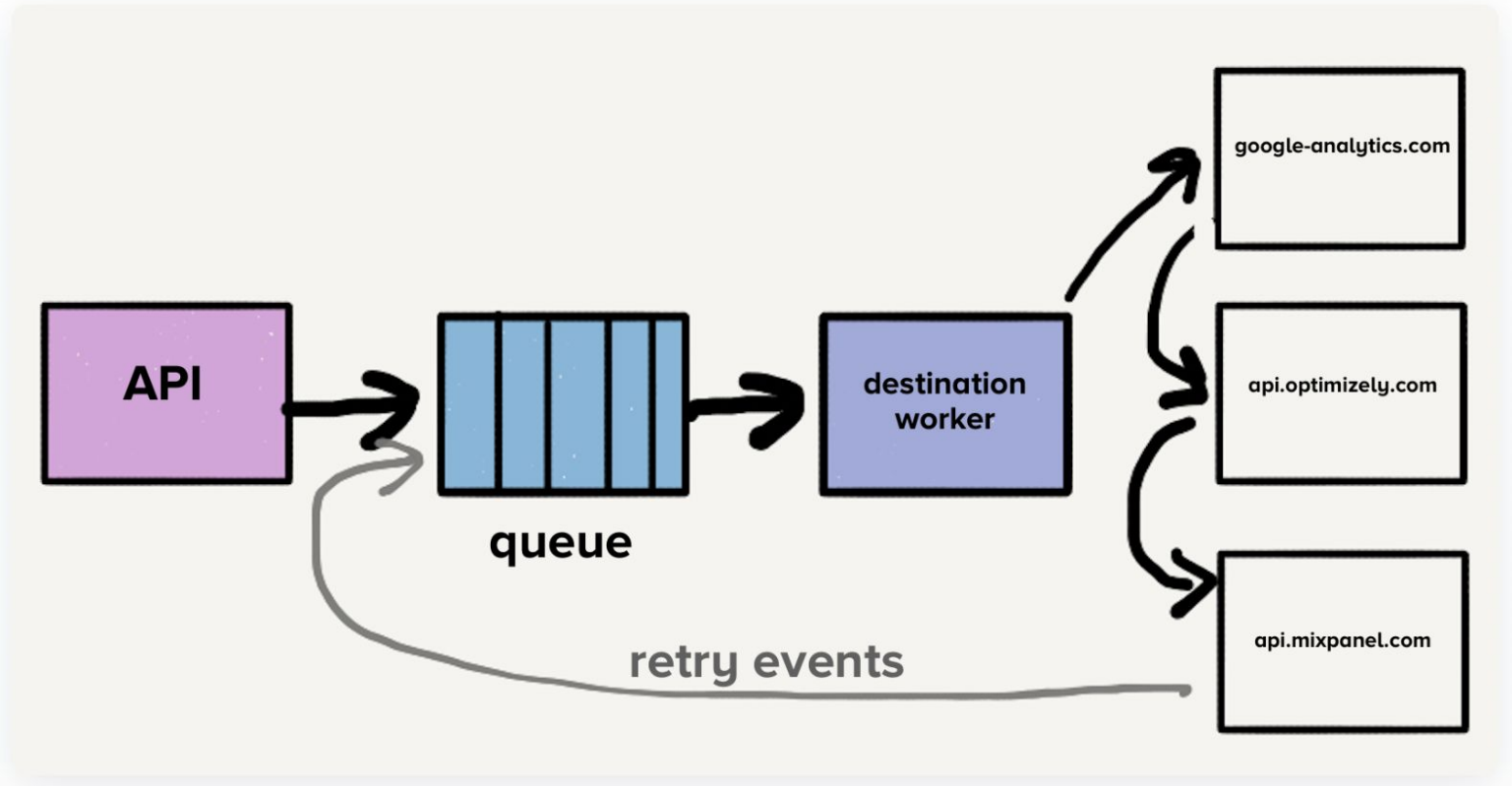
Questions?

- We'll look at an example next

An Engineering Story: Monolith to Microservices and Back

- https://segment.com/blog/goodbye-microservices/
- Segment's customer data infrastructure ingests hundreds of thousands of events per second and forwards them to partner APIs
    - "destinations."
- There are over one hundred types of these destinations, such as Google Analytics, Optimizely, or a custom webhook.
- Years back, when the product initially launched, the architecture was simple.
- There was an API that ingested events and forwarded them to a distributed message queue.
    - An event, in this case, is a JSON object generated by a web or mobile app containing information about users and their actions.

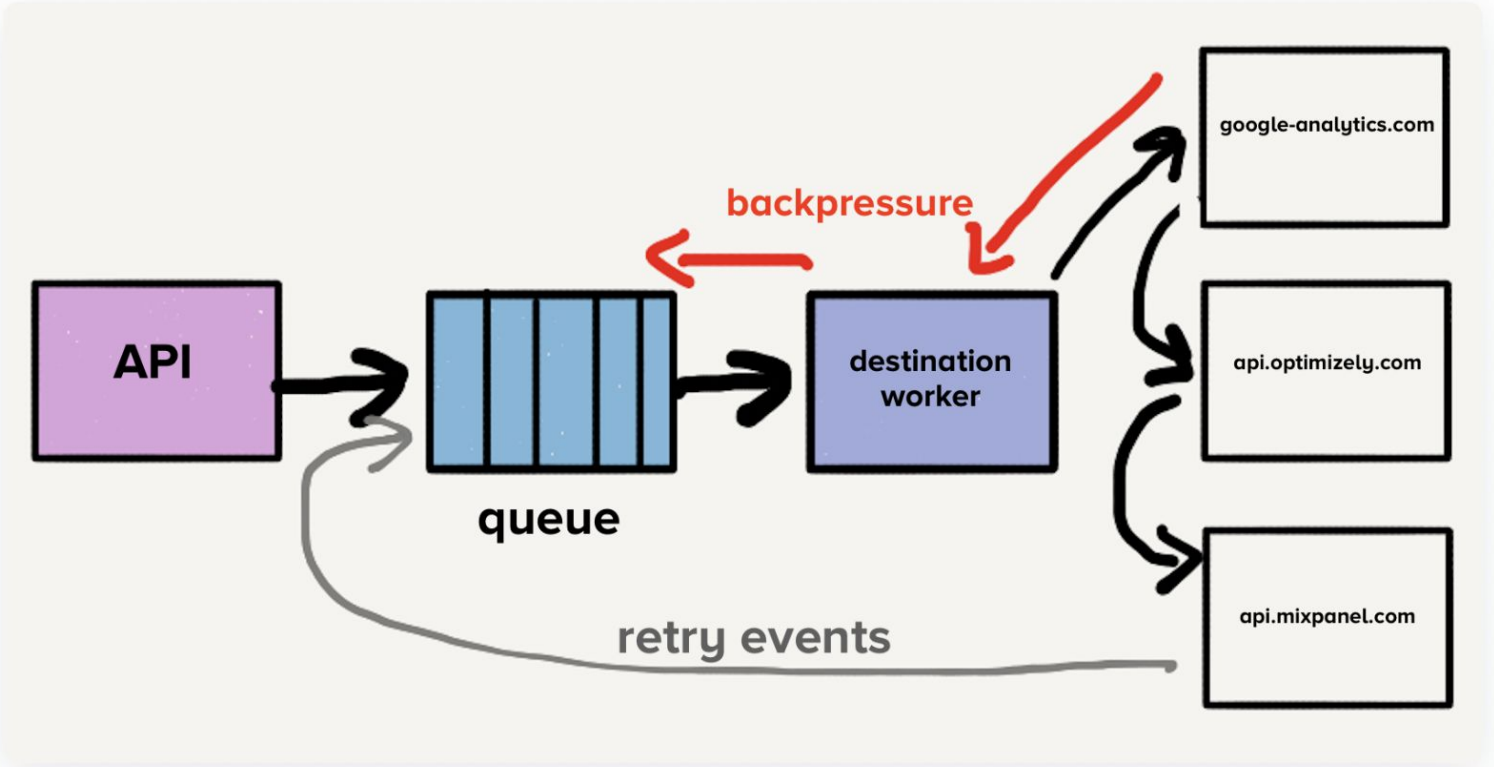
An Engineering Story: Monolith to Microservices and Back



An Engineering Story: Monolith to Microservices and Back

- Single queue contained both the newest events as well as those which may have had several retry attempts, across all destinations, which resulted in head-of-line blocking.
- If one destination slowed or went down, retries would flood the queue, resulting in delays across all destinations.
- Imagine destination X is experiencing a temporary issue and every request errors with a timeout
    - Every failed event is put back to retry in the queue.
    - Delivery times for all destinations would increase because destination X had a momentary outage.
    - Customers rely on the timeliness of this delivery, so can’t afford increases in wait times anywhere in pipeline.

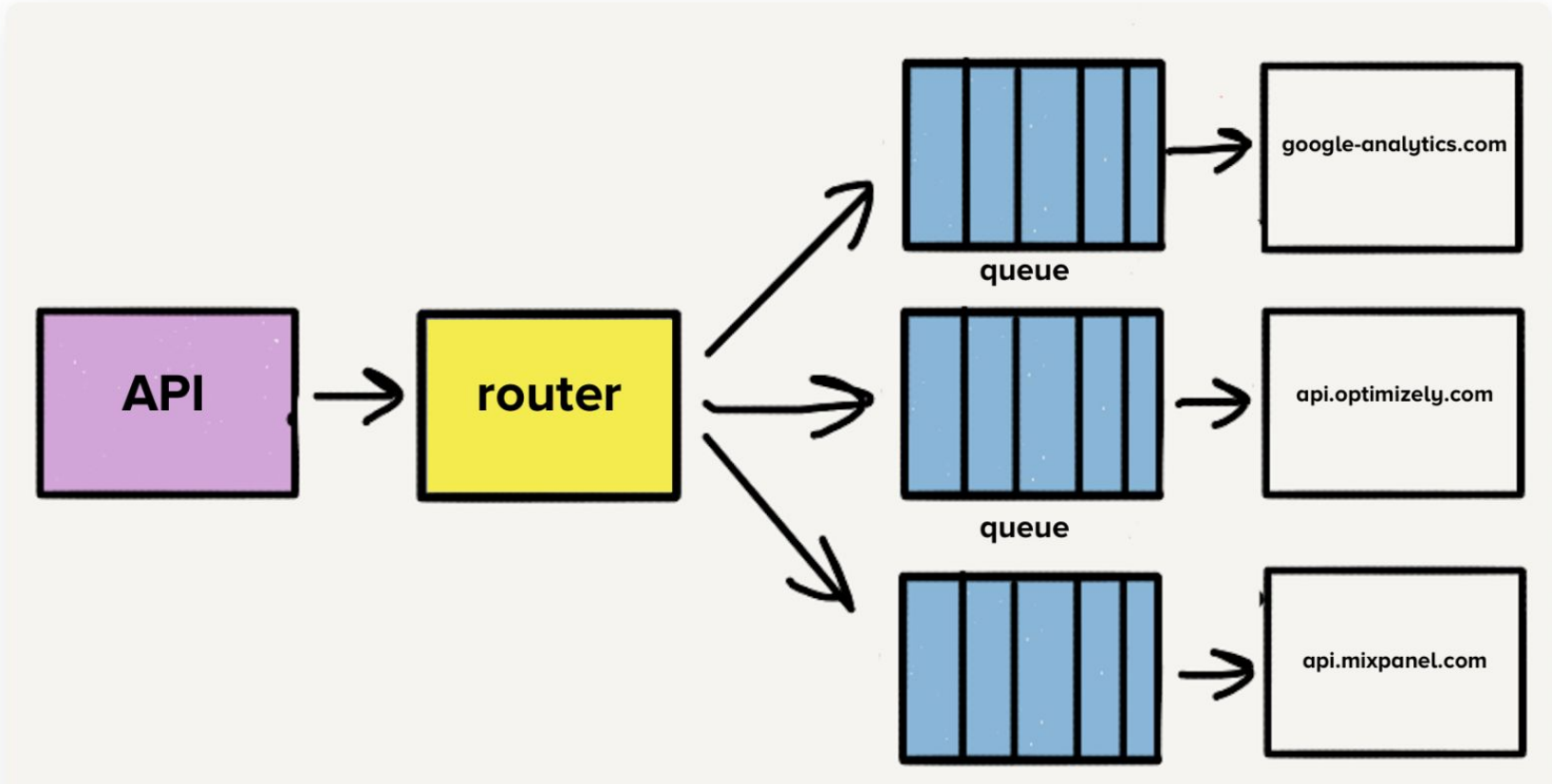
An Engineering Story: Monolith to Microservices and Back



An Engineering Story: Monolith to Microservices and Back

- Initially, when the destinations were divided into separate services, all of the code lived in one repo.
- A huge point of frustration was that a single broken test caused tests to fail across all destinations.
    - to deploy a change, we had to spend time fixing the broken test even if the changes had nothing to do with the initial change.
- In response to this problem, it was decided to break out the code for each destination into their own repos.
- All the destinations were already broken out into their own service, so the transition was natural.
- This isolation allowed the development team to move quickly when maintaining destinations.

An Engineering Story: Monolith to Microservices and Back



An Engineering Story: Monolith to Microservices and Back

- Over time, added over 50 new destinations, and that meant 50 new repos.
- Created shared libraries to make common transforms and functionality, such as HTTP request handling, across our destinations easier and more uniform.
- The shared libraries made building new destinations quick.
    - The familiarity brought by a uniform set of shared functionality made maintenance less of a headache.

An Engineering Story: Monolith to Microservices and Back

- However, a new problem began to arise.
- Testing and deploying changes to these shared libraries impacted all of our destinations.
- 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.
- Over time, the versions of these shared libraries began to diverge across the different destination codebases.
- Eventually, all of them were using different versions of these shared libraries.

An Engineering Story: Monolith to Microservices and Back

- Different services have different traffic and thus, scaling needs
- Need to tune each individually and autoscale each individually

An Engineering Story: Monolith to Microservices and Back

- Back to Monolith (repo and service)
- Consolidate the now over 140 services into a single service.
- For each of the 120 unique dependencies, commit to one version for all our destinations.
  - While moving destinations over, check the dependencies it was using and update them to the latest versions.
  - Fix anything in the destinations that broke with the newer versions.
- No longer needed to keep track of the differences between dependency versions.
  - All destinations were using the same version, which significantly reduced the complexity across the codebase.
  - Maintaining destinations now became less time consuming and less risky.

An Engineering Story: Monolith to Microservices and Back