Core argument for Apex: although existing api gateways are good for outward facing traffic, they are not ideal for the kind of scenario where a business is transitioning from a monolithic architecture to microservices architecture, because they are optimized for traffic at the edge of the data center, they are inefficient for the large volume of traffic in distributed microservices environments. It cannot be containerized and the constant communication with the database and a configuration server adds latency.

Connecting Microservices: Broad requirements

Authentication: a client needs to be authenticated in all the services
Security (encryption): with many microservices, the attack surface is greater
Logging (broadly observability): logs of all the services are distributed in various places
Service discovery (needs sth like dns): services running on ephimeral containers need a way to track which service is running on which container; where to forward a request
Traffic control: Related to load-balancing and routing (not sure yet)

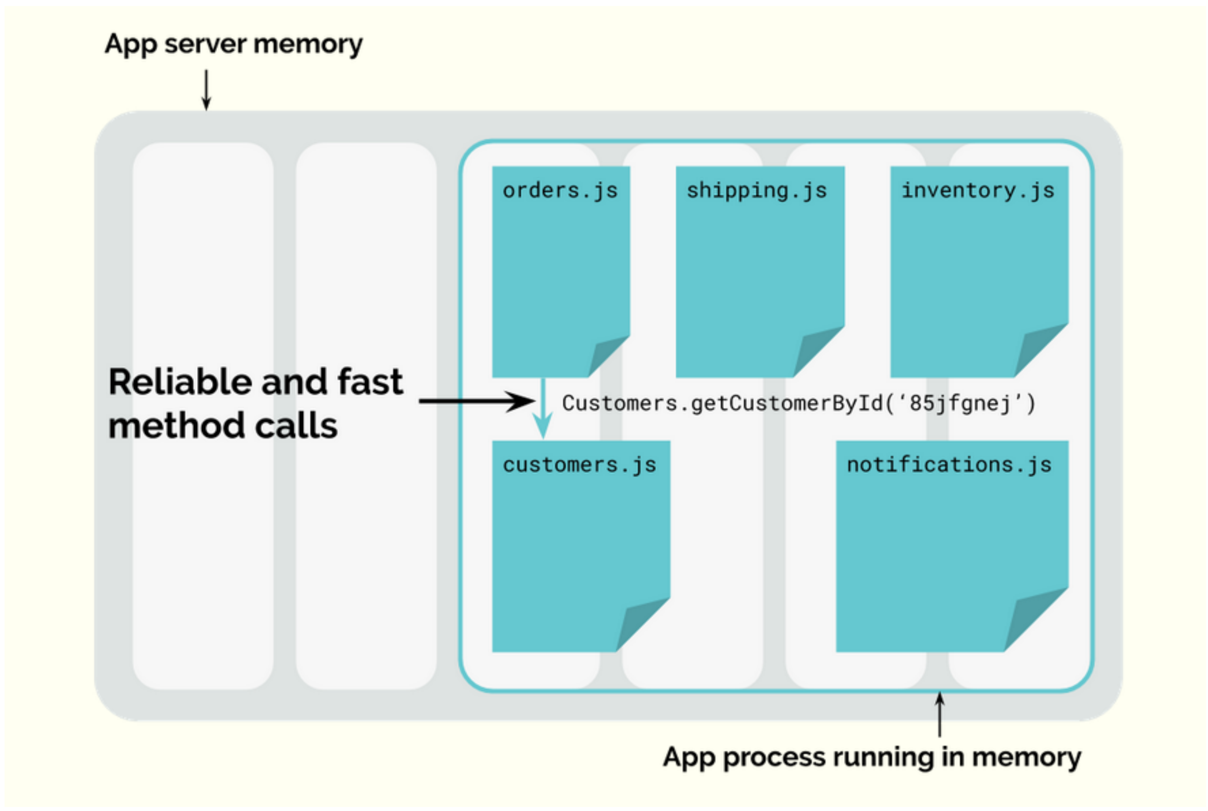
cross-cutting concerns: authentication, security, service-discovery

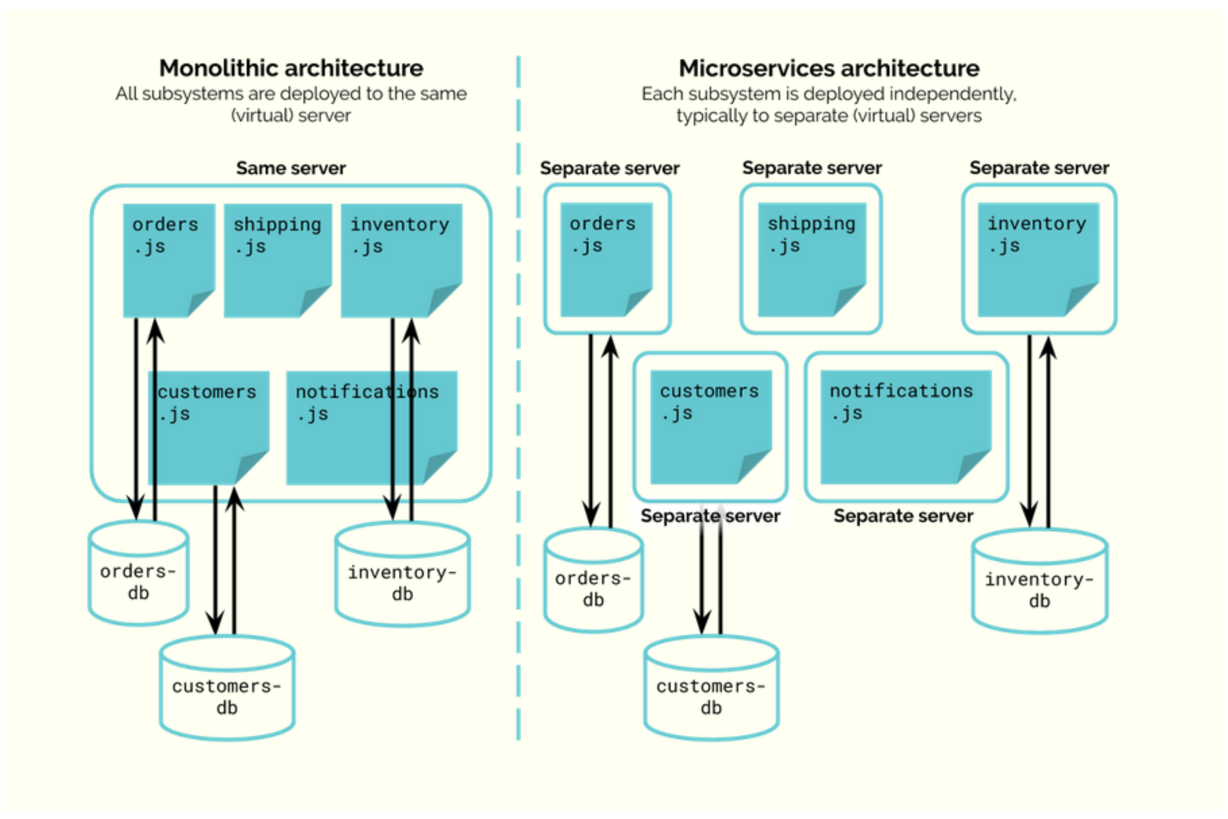
Two Main Solutions:

```
[[#API gateway]]  [kong, tyk, mulesoft, boomi, amazon api gateway, azure api management, cilium, apache APISIX]

[[Service Mesh]]: [istio, hashicorp consul, aws app mesh, apache serviceComb]
```

BackStory: Monolith to Microservices


App server memory
orders.js
shipping.js
inventory.js
Reliable and fast method calls
Customers.getCustomerById('85jfgnej')
customers.js
notifications.js
App process running in memory



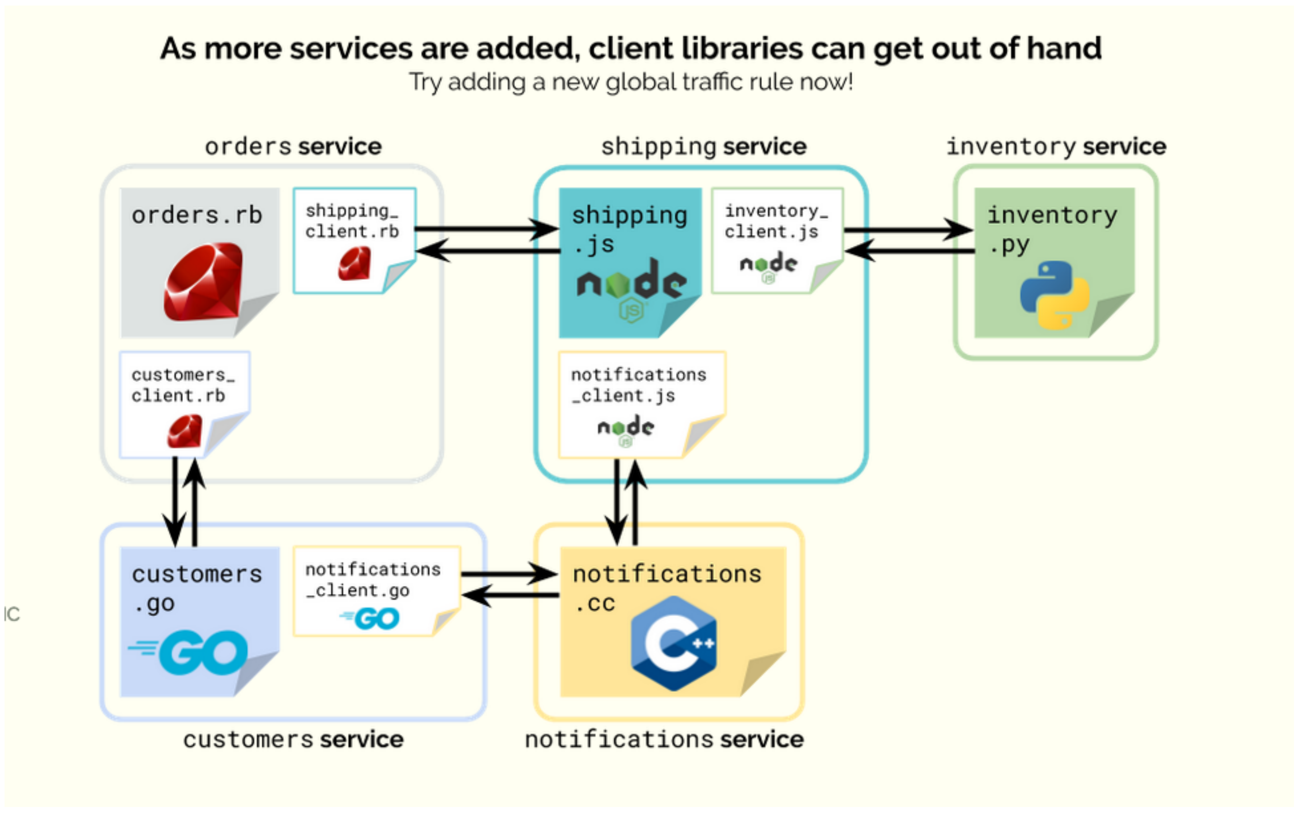
In a microservice architecture, services talk to each other through api calls. For instance, `orders` service talks to the `shipping` service by making a `POST` request to `/shipments` endpoint.

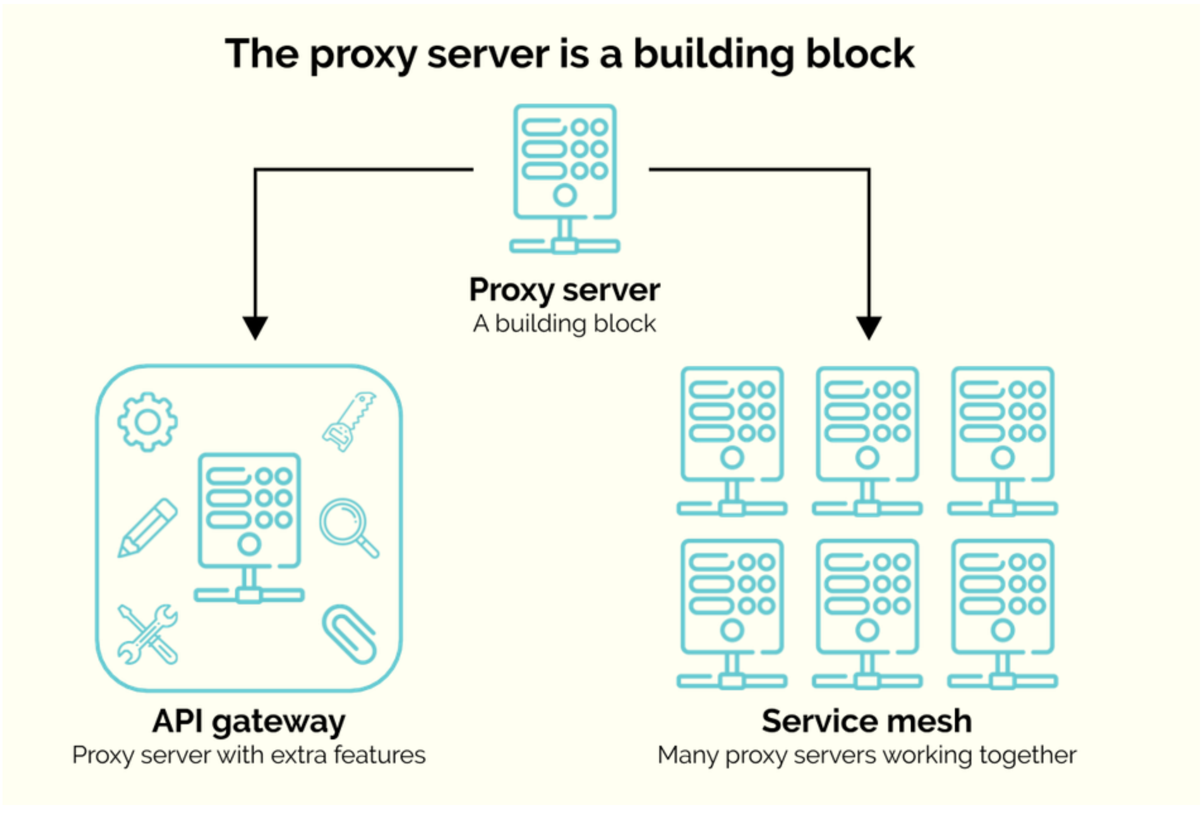
Path towards microservices: dividing up different services that are independently deployable.

Challenges of Microservices architecture

- Network is unreliable: hence, requires retry logic and circuit-breakers. Calls might be unsuccessful for various reasons. Other service might be down.
- Latency: hence, services making a call to others require hopping over networks, a complete cycle of a business transaction would require for the request to cyle through various services. Thus, adding latency.
- Debugging or diagnosing network faults is cumbersome: tracing the request through the services. The error logs are distributed across the system.
- Managing fault-handling logic: where should all the rety-logic be defined?
  - is it in the http library that are imported in each service's code?
  - there are packages for handling the rety logic
  - there are also packages for handling logging, caching, rate-limiting, authentication
  - api-producers might provide their own framework that handles these issues

> Needless to say, this solution becomes less and less manageable as the number of services grows. Every time a new service is built in a new language, every other service owner must write a new client library in that language. More critically, updating fault-handling logic now incurs a great deal of repetitive work. Suppose the CTO wishes to update the global defaults for the retry logic; developers would now have to update the code in multiple client libraries in every service, then carefully coordinate each service's redeployment. The greater the number of services, the slower this process becomes. [source]




The proxy server is a building block
Proxy server
A building block
API gateway
Proxy server with extra features
Service mesh
Many proxy servers working together

A proxy is a server that sits on the path of network traffic betwen two communicating machines, and intercepts all their requests and responses. It's the job of the proxy to forward the request to the appropriate servers. At this stopping-point, all the features are implemented. The issues with authentication, traffic control, rate-limiting. When the proxy receives a response, it forward it to the right place.

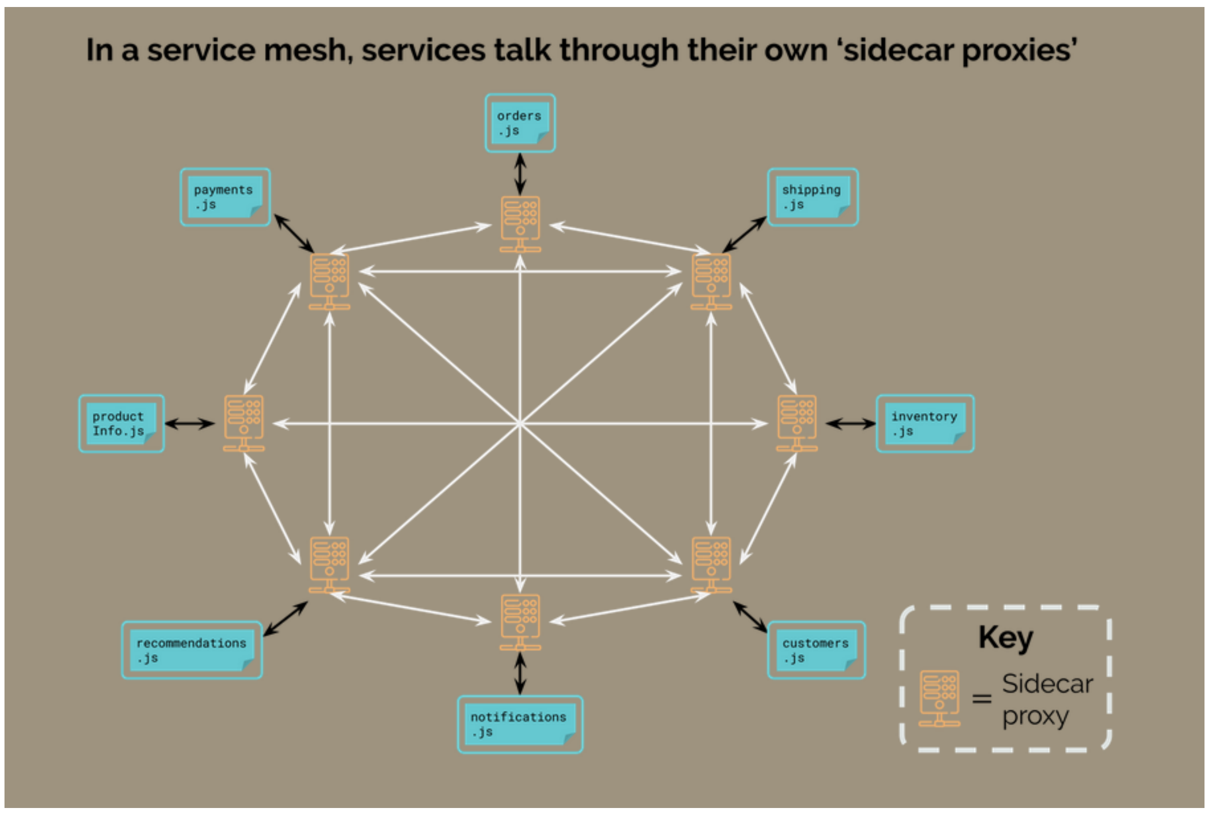
API Gateway

- outward facing reverse proxy
- one of its primary functions is to provide a stable API to clients and route client requests to the appropriate service.

API gateway provides a curtain-like shield such that the client does not have to know how the internal services are undergoing change.

Service Mesh

A complex solution that requires having a sidecar proxy for each node



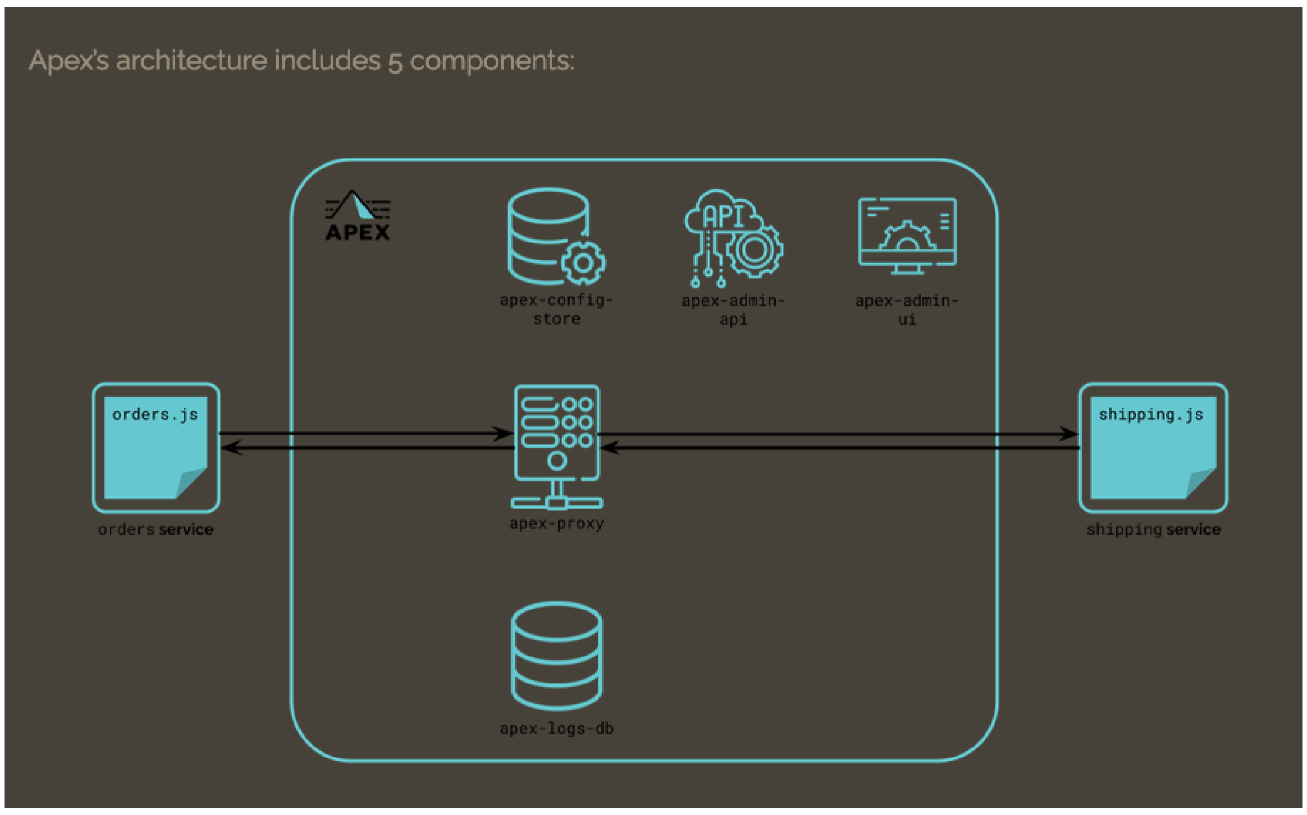
Apex

Solution: We need a different kind of api-gateway that facilitates the communication between services.

> Apex is an api proxy for microservices: to log and control service-to-service traffic

Essential tradeoffs for small companies transitioning to microservices:

- It's an idea between API gateway and service mesh, but more like an api gateway with less features
- trades simplicity and convenience for high availability and scalability
- suited for small companies that are just transitioning from a monolith to a microservices architecture



Features of Apex

- Logging and tracing service-to-service traffic by inserting `correlation-id` to http headers before forwarding the request to another service. Hence, looking up a request cyle is just about quering the database for a particular correlation-id

- Offers fault-handling logic: every request queries the `apex-config-store` for authentication, routing, and retry logic in that order. Only after all three are completed are the requests forwarded to another service.
- Does not rule out a transition to service-mesh architecture
- Processing logs asynchronously
- Persisting logs and config data in containers

Limitations of Apex

- Not designed for high scalability and high availability, bc of the single point of failure
- Single point of failure
- Discounts the case of bursty traffic when the queue for processing logs is overwhelmed
- Requires multiple reads from the `apex-config-store`; this adds significant latency, but is an acceptable tradeoff under the assumption that the ones using apex are going to use it for connecting a small number of microservices