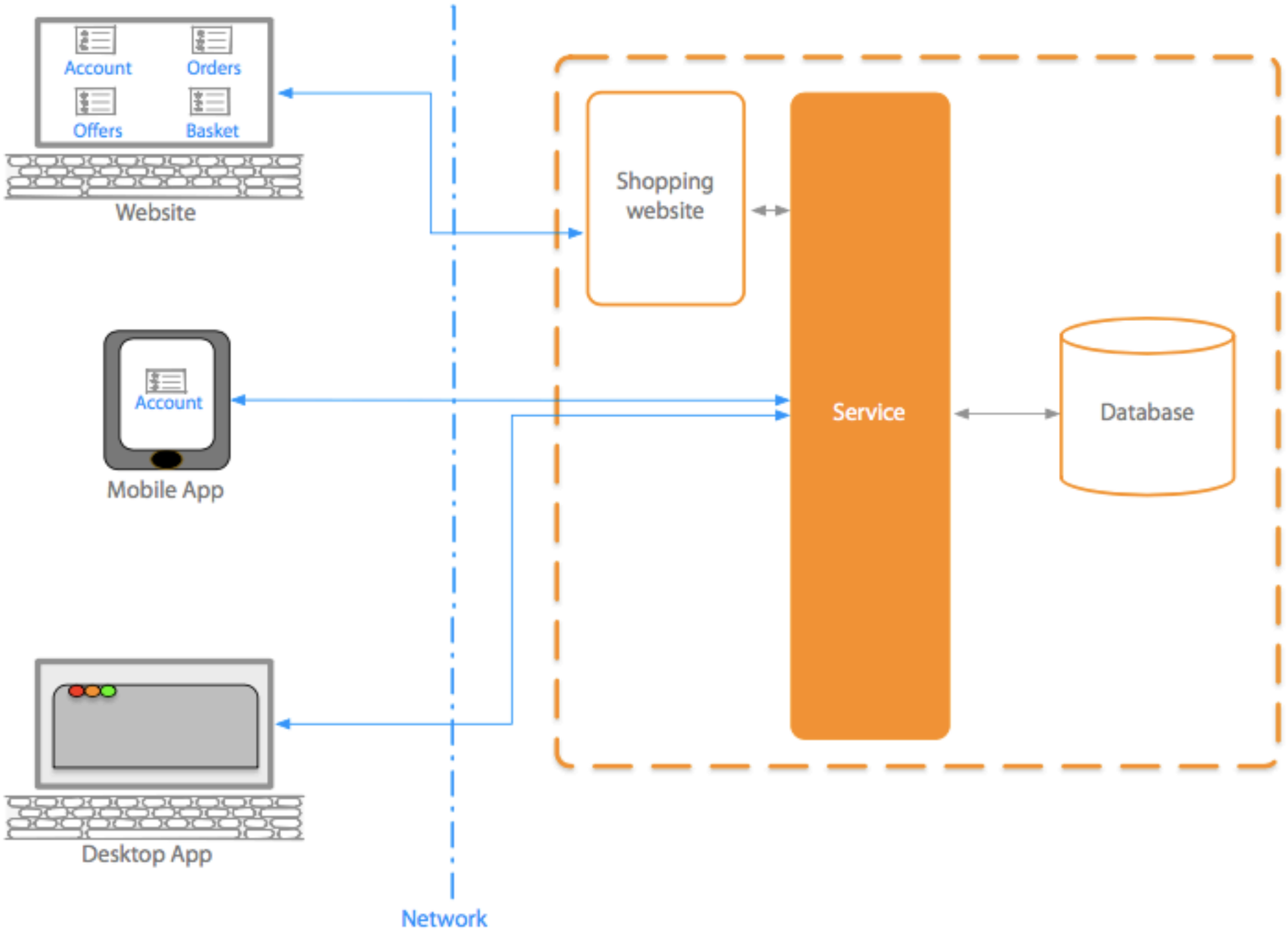
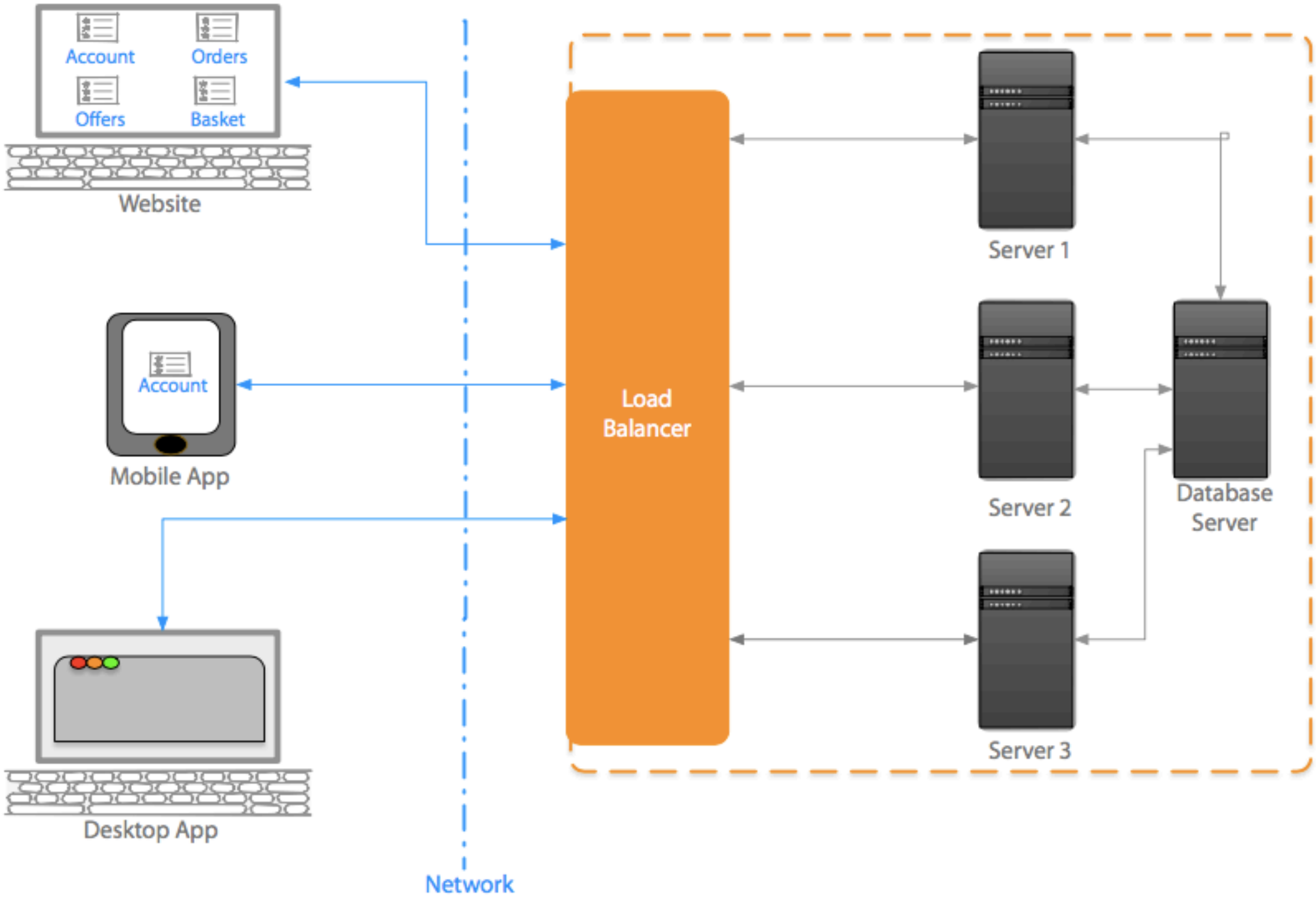
Microservices Architecture

What is a Service?

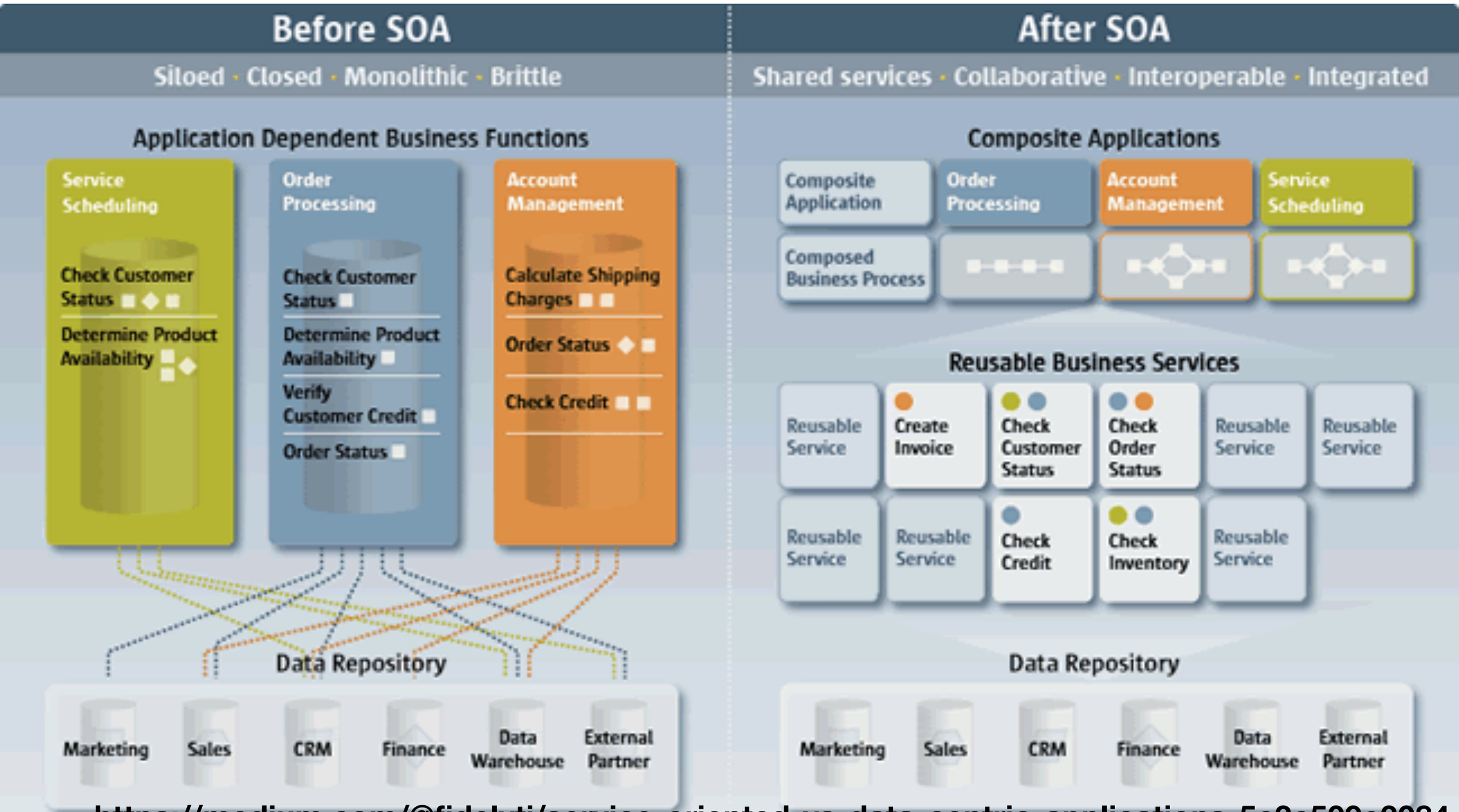
Service


Account
Orders
Offers
Basket
Website
Shopping website
Service
Database
Account
Mobile App
Desktop App
Network

Service


Account
Orders
Offers
Basket
Website
Account
Mobile App
Desktop App
Network
Load Balancer
Server 1
Server 2
Server 3
Database Server

SOA



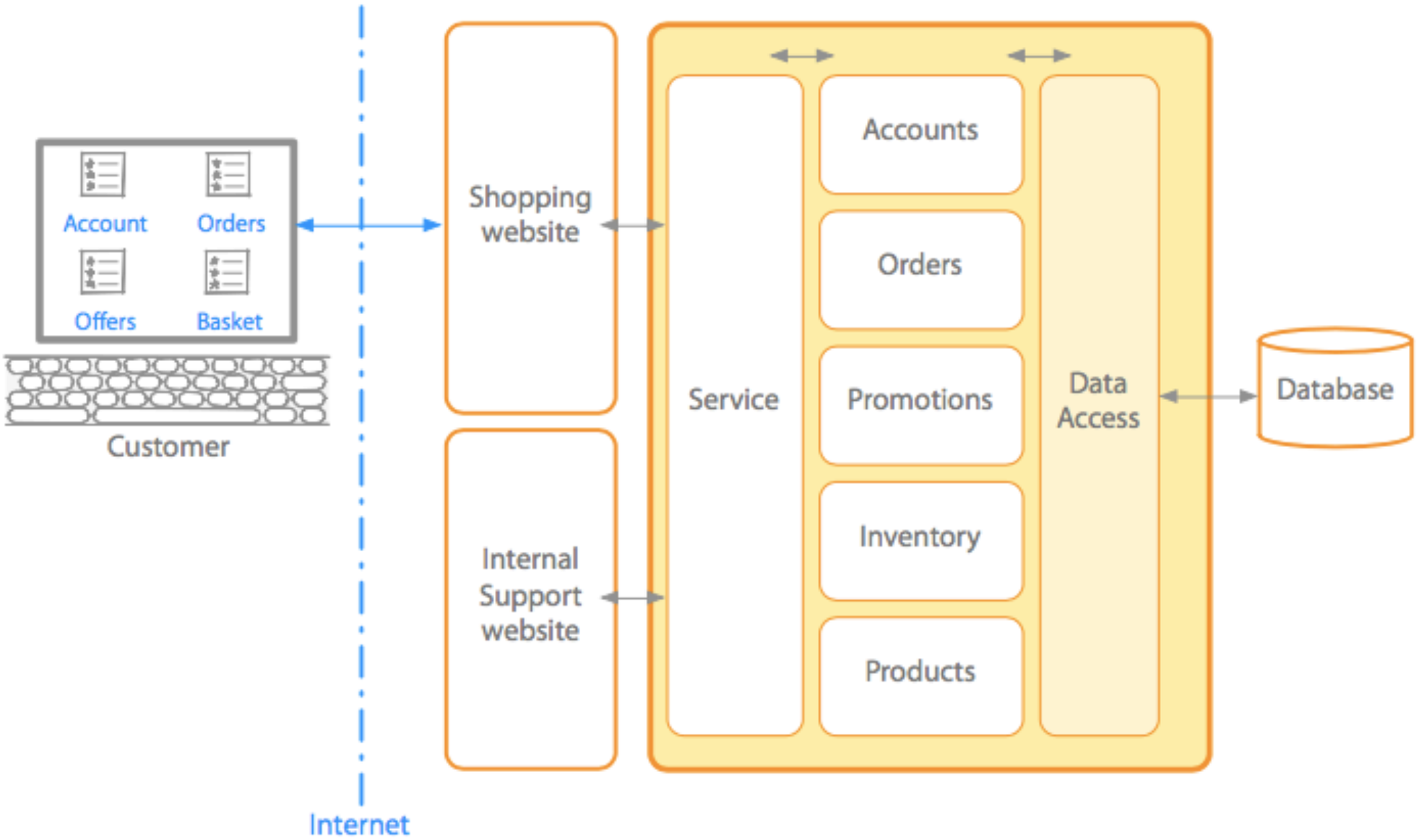
https://medium.com/@fidelvti/service-oriented-vs-data-centric-applications-5c3c509e3084

SOA

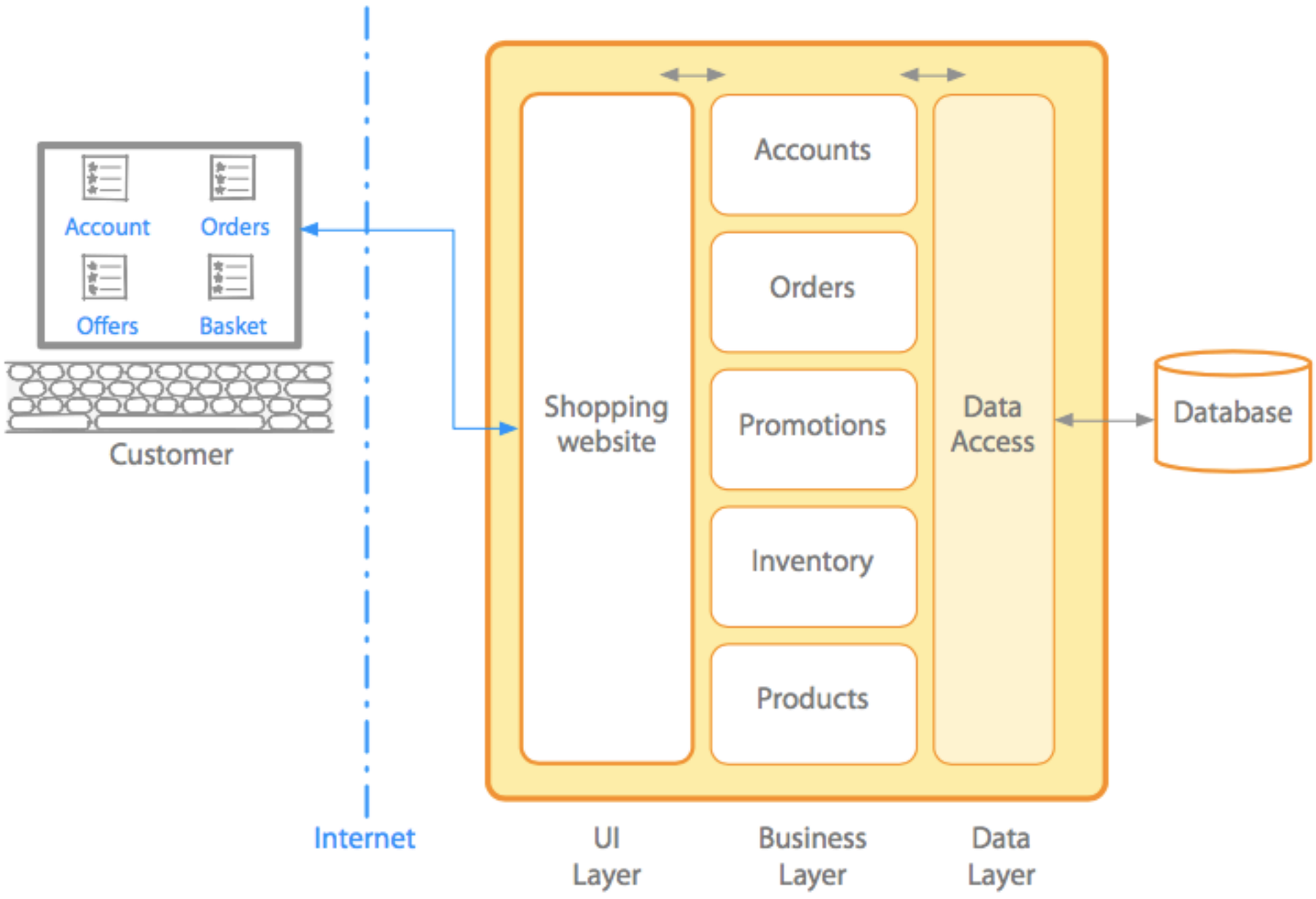
SOA done well

- Knowing how to size a service
- Traditional SOA resulted is a monolithic services

Monolithic


Account
Orders
Offers
Basket
Customer
Internet
Shopping website
Internal Support website
Accounts
Orders
Promotions
Inventory
Products
Service
Data Access
Database

Monolithic


Account
Orders
Offers
Basket
Customer
Internet
Accounts
Orders
Promotions
Inventory
Products
Shopping website
Data Access
Database
UI Layer
Business Layer
Data Layer

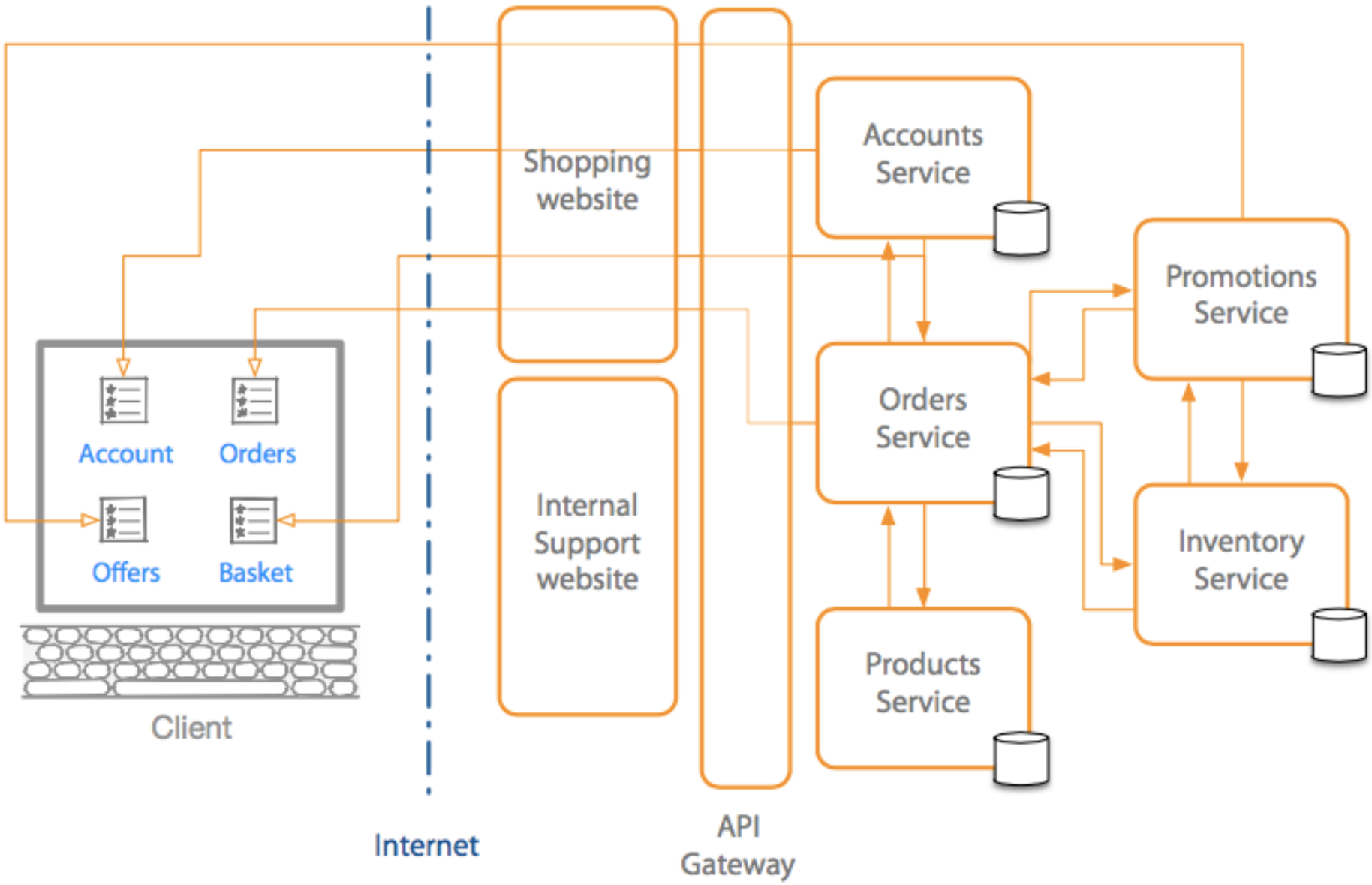
Monolithic

- Typical enterprise application (No restriction on size)
- Large codebase
- Longer development times
- Challenging deployment
- Inaccessible features
- Fixed technology stack

Monolithic

- High levels of coupling
  - Between modules
  - Between services
- Failure could affect whole system
- Scaling requires duplication of the whole
- Single service on server
- Minor change could result in complete rebuild
- Easy to replicate environment

Microservices



Resources

https://microservices.io

Microservice Architecture

Supported by Kong

Patterns Articles Presentations Resources Assessment Platform new Other Languages About

Posts

22 Mar 2018 » microXchg 2018 - Managing data consistency in a microservice architecture using Sagas
20 Feb 2018 » CodeFreeze 2018 - There is no such thing as a microservice!
04 Dec 2017 » QCONSF 2017 presentation - ACID Is So Yesterday, Maintaining Data Consistency with Sagas
04 Dec 2017 » Upcoming public training - Microservices at UMN, Minneapolis in January
01 Aug 2017 » Presentation - Solving distributed data management problems in a microservice architecture
24 Jul 2017 » Revised data patterns
26 Mar 2017 » There is no such thing as a microservice!
24 Feb 2017 » New book - Microservice patterns
12 Feb 2017 » How to apply the pattern language
06 Jun 2016 » Fantastic presentation by Eric Evans on DDD and microservices
23 Feb 2016 » One day microservices class in Oakland, CA
21 Feb 2016 » Microservice chassis pattern
28 Jul 2015 » What's new with #microservices - July 28, 2015: integration platforms, new article on microservices and IPC
23 Jun 2015 » What's new with #microservices - June 23, 2015: @crichardson and microservices training
06 Jun 2015 » What's new with #microservices - June 6, 2015: @martinfowler, @crichardson, @adrianco

What are microservices?

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack.

Microservices are not a silver bullet

The microservice architecture is not a silver bullet. It has several drawbacks. Moreover, when using this architecture there are numerous issues that you must address. The microservice architecture pattern language is a collection of patterns for applying the microservice architecture. It has two goals:

1. The pattern language enables you to decide whether microservices are a good fit for your application.
2. The pattern language enables you to use the microservice architecture successfully.

Where to start?

A good starting point is the Monolithic Architecture pattern, which is the traditional architectural style that is still a good choice for many applications. It does, however, have numerous limitations and issues and so a better choice for large/complex applications is the Microservice architecture pattern.

Example microservices applications

Want to see an example? Check out Chris Richardson's Money Transfer and Kanban board examples.

See code

How to apply the pattern language

An article that describes how develop a microservice architecture by applying the patterns

Learn more

Microservices adoption: Who is using microservices?

Many companies are either using microservices or considering using them. Read the case studies...

About Microservices.io

Microservices.io is brought to you by Chris Richardson. Experienced software architect, author of POJOs in Action and the creator of the original CloudFoundry.com. His latest startup is eventuate.io, a microservices application platform.

Learn more about microservices

Chris offers a comprehensive set of resources for learning about microservices including articles, an O'Reilly training video, and example code.

Learn more

Resources

BLOG TECH Chris Richardson of Eventuate, Inc. May 19, 2015

Introduction to Microservices

microservices, monolithic application, Docker, cloud

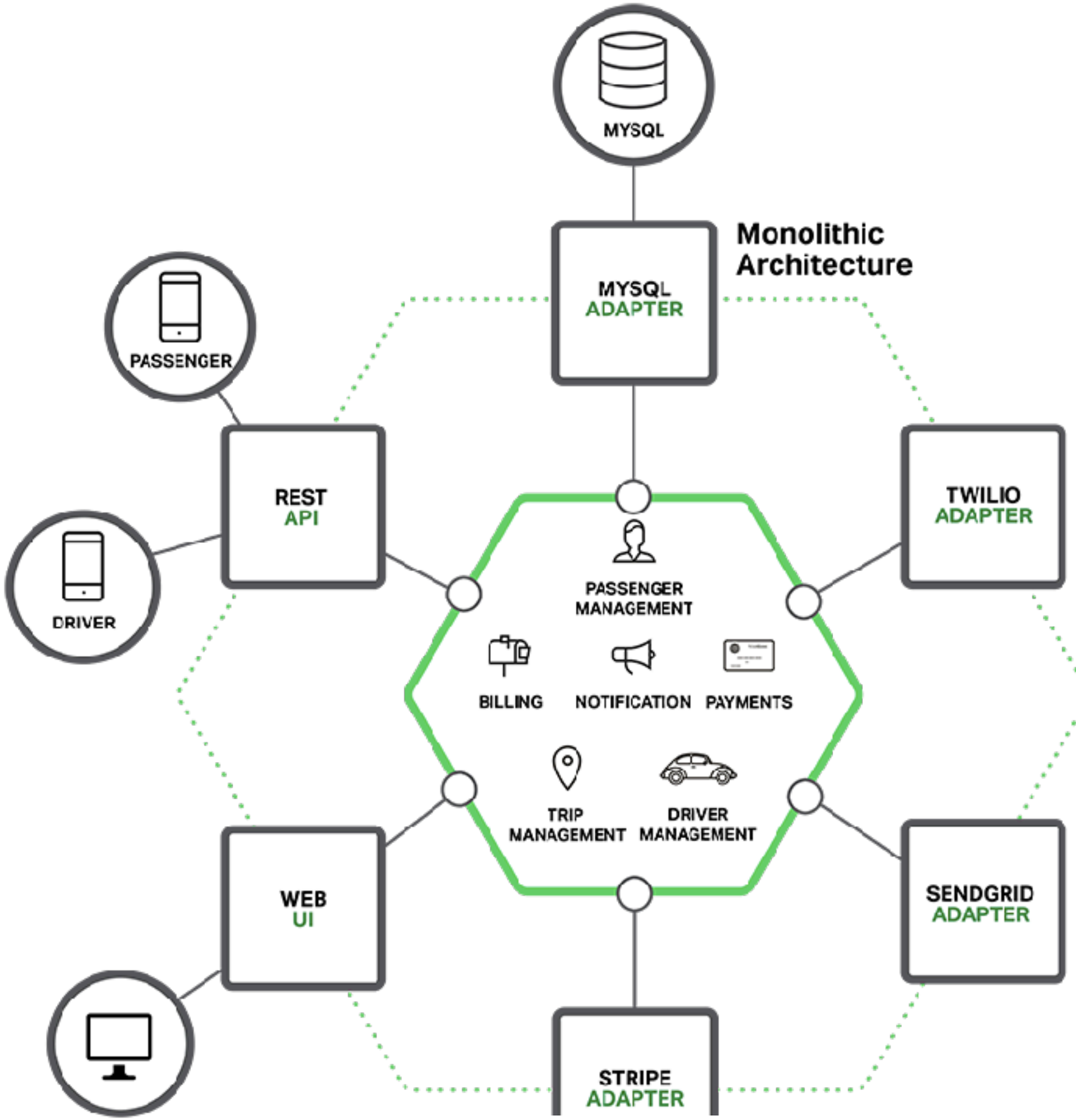
TWITTER LINKEDIN

Editor *– This seven-part series of articles is now complete:*

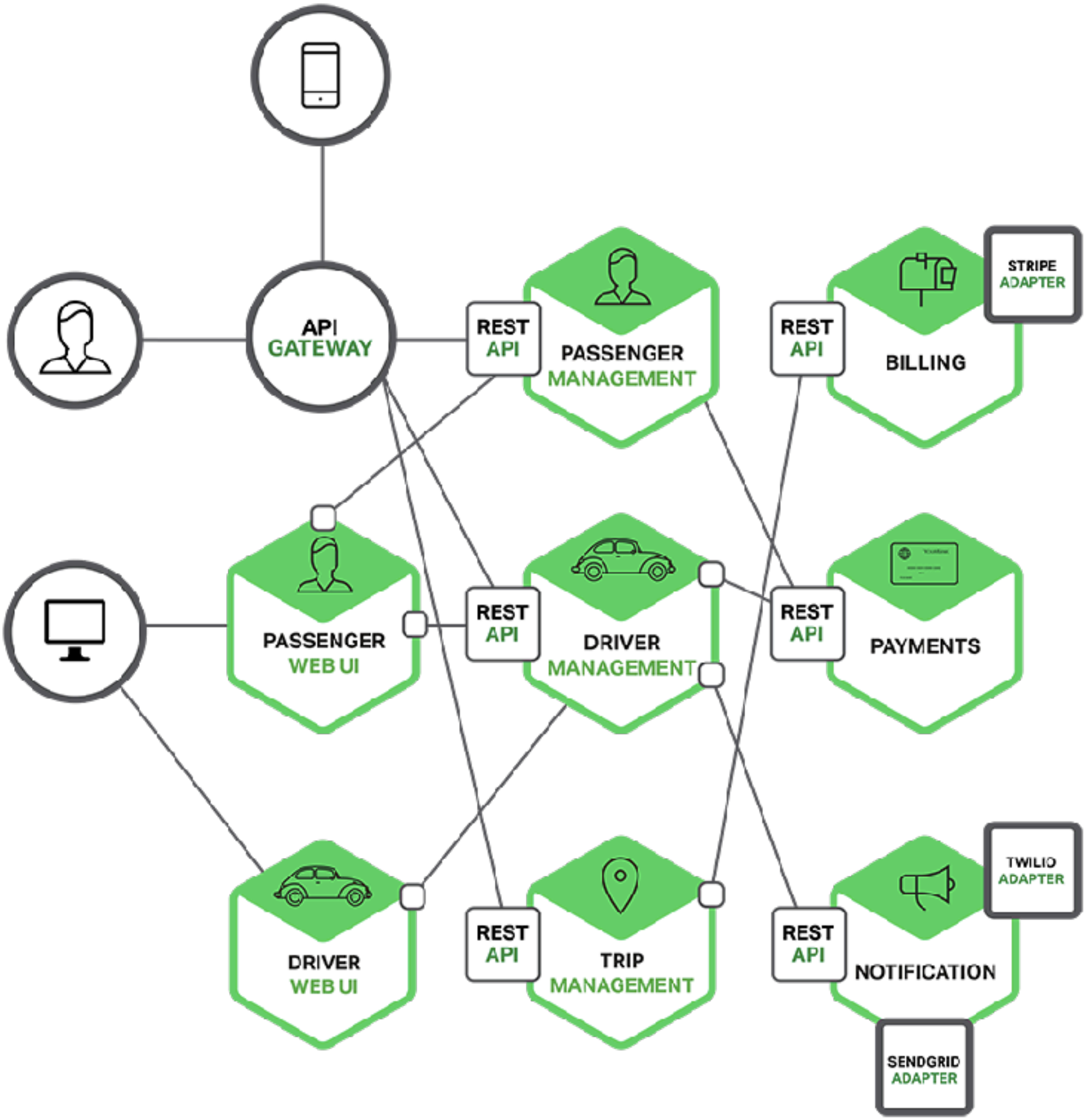
1. *Introduction to Microservices (this article)*
2. *Building Microservices: Using an API Gateway*
3. *Building Microservices: Inter-Process Communication in a Microservices Architecture*
4. *Service Discovery in a Microservices Architecture*
5. *Event-Driven Data Management for Microservices*
6. *Choosing a Microservices Deployment Strategy*
7. *Refactoring a Monolith into Microservices*

You can also download the complete set of

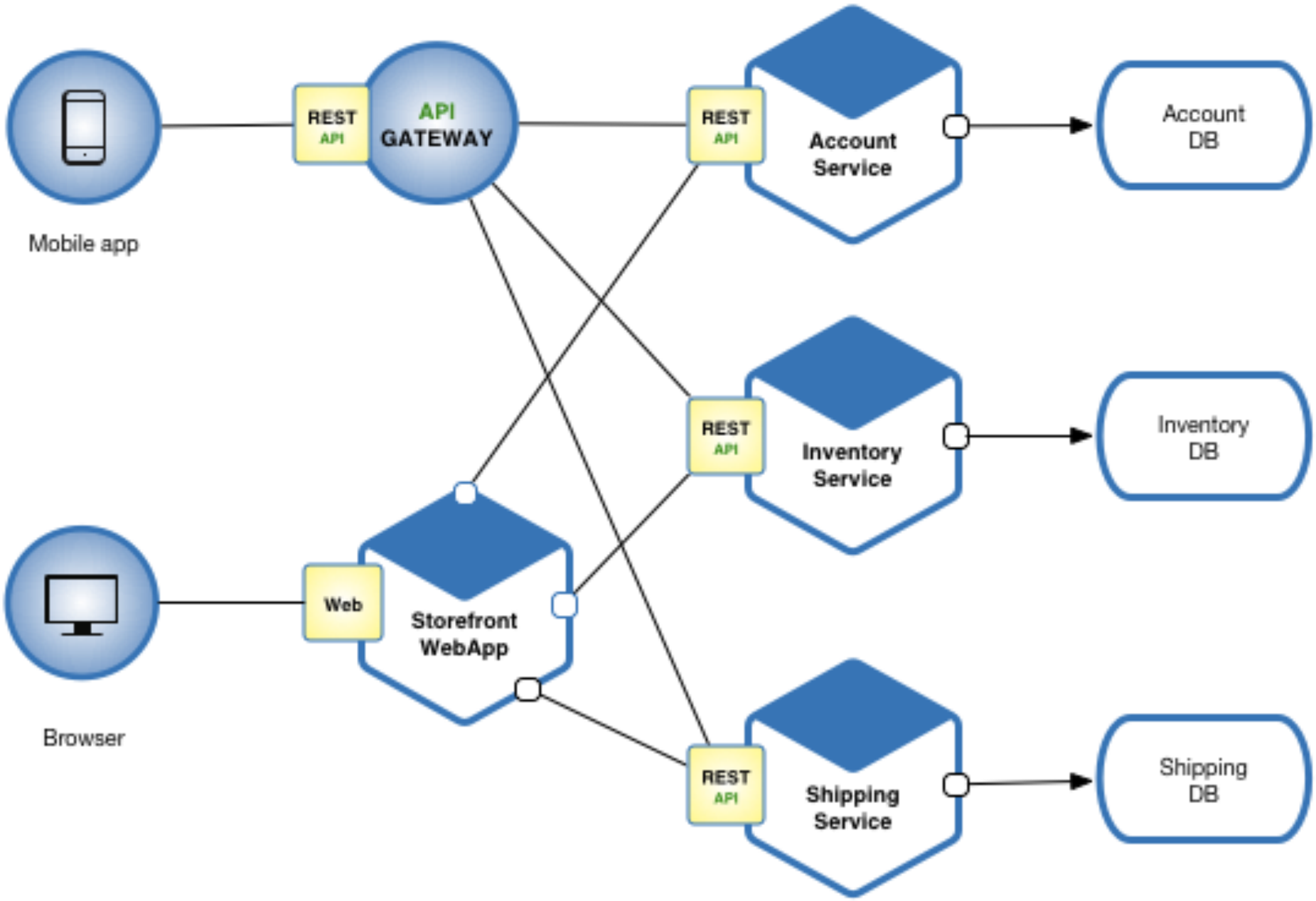
https://www.nginx.com/blog/introduction-to-microservices



https://www.nginx.com/blog/introduction-to-microservices


API
GATEWAY
REST
API
PASSENGER
MANAGEMENT
REST
API
BILLING
STRIPE
ADAPTER
PASSENGER
WEB UI
REST
API
DRIVER
MANAGEMENT
REST
API
PAYMENTS
DRIVER
WEB UI
REST
API
TRIP
MANAGEMENT
REST
API
NOTIFICATION
TWILIO
ADAPTER
SENDGRID
ADAPTER

E-commerce Application



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

Netflix OSS

NETFLIX | OSS

POWERED BY

Getting Started

How can you get started quickly?

For the simple approach, try out our ZeroToDocker container images. After downloading the images, you can be up and running NetflixOSS in just a few minutes.

After you've tackled that, check out the IBM ACME Air and Flux Capacitor apps, and the Zero-to-Cloud workshop.

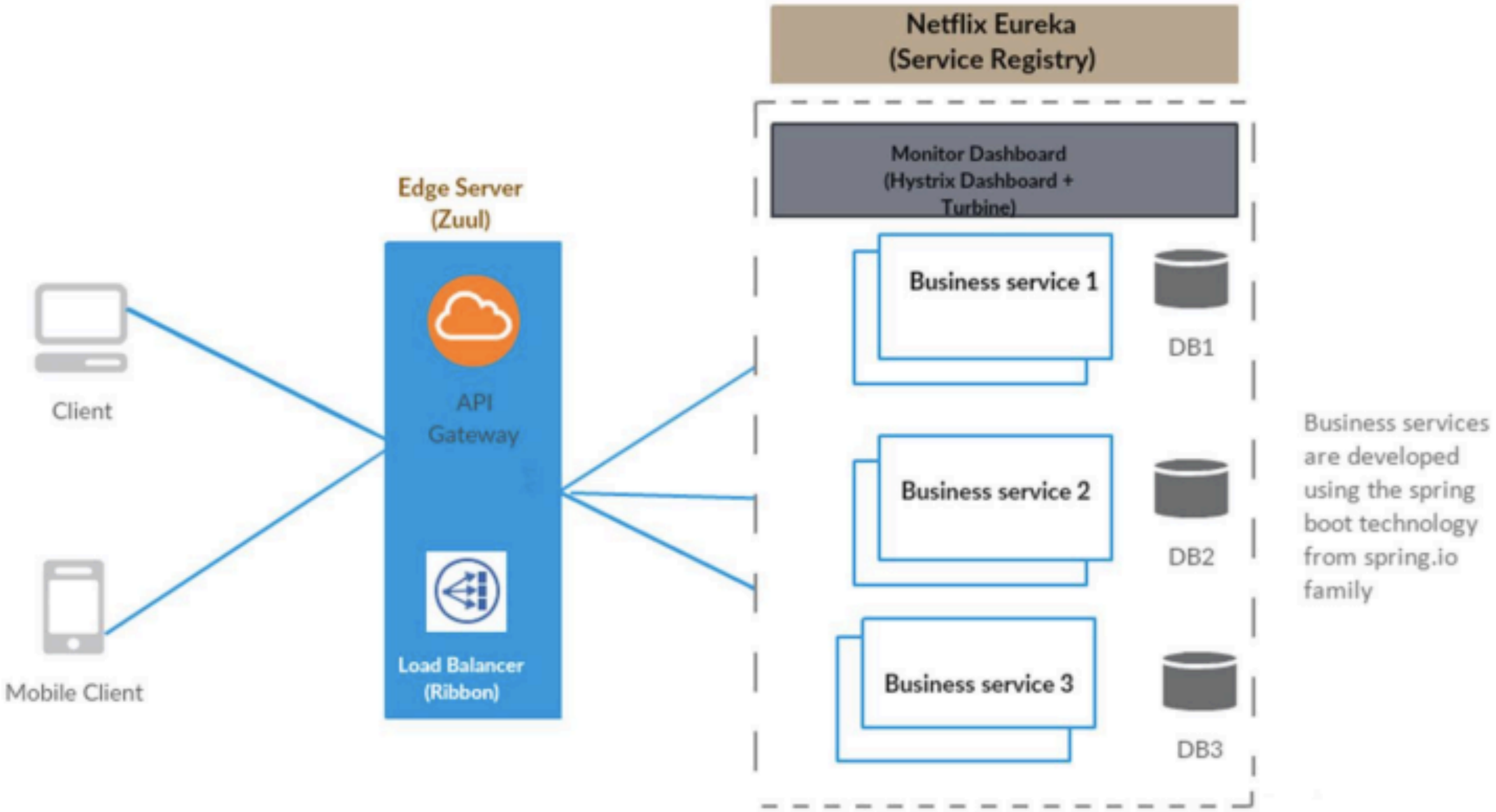
See these CloudFormation templates on Answers For AWS for use of NetflixOSS through CloudFormation.

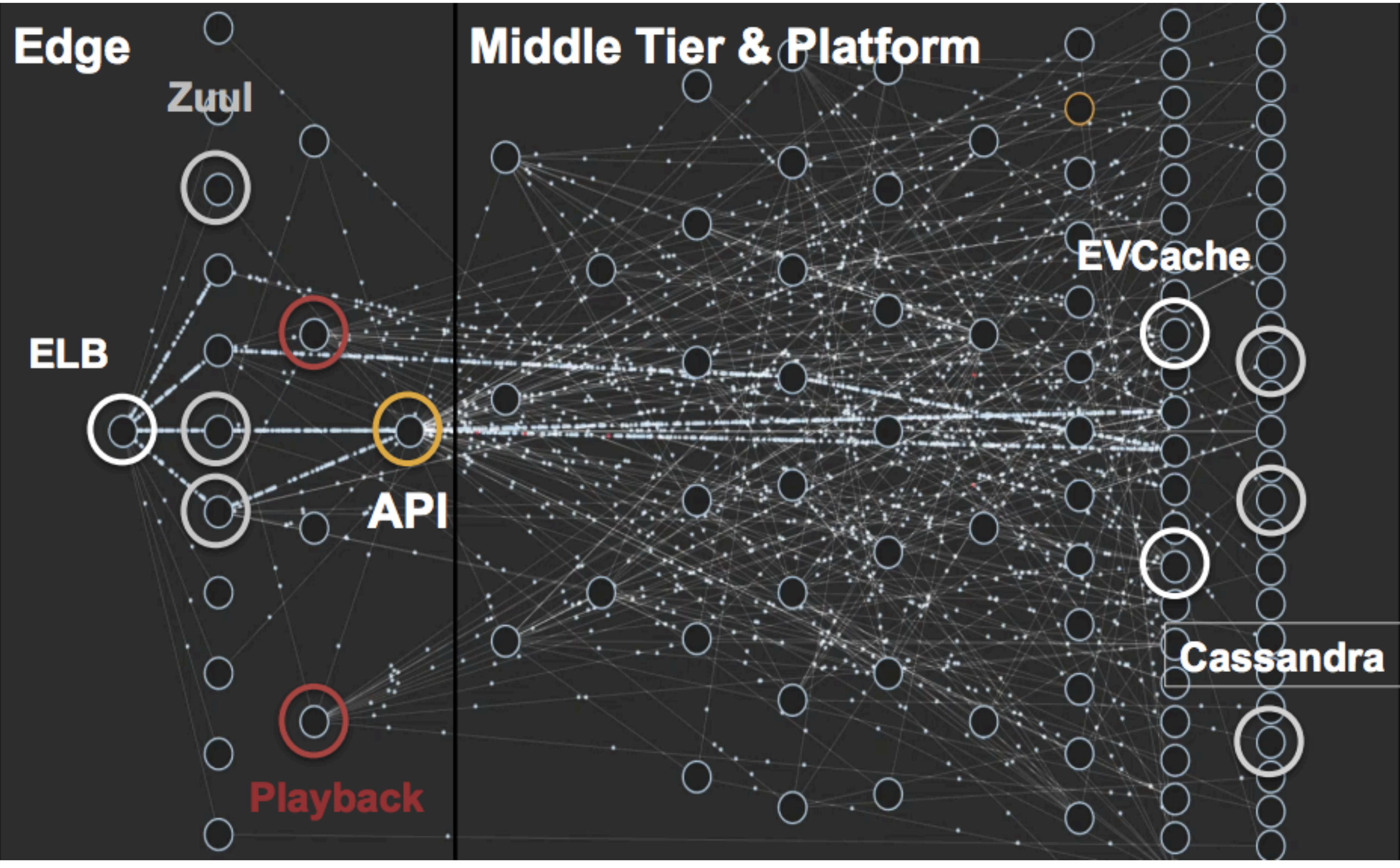
Netflix Open Source Software Center

Netflix is committed to open source. Netflix both leverages and provides open source technology focused on providing the leading internet television network. Our technology focuses on providing immersive experiences across all internet-connected screens. Netflix's deployment technology allows for continuous build and integration into our worldwide deployments serving members in over 50 countries. Our focus on reliability defined the bar for cloud based elastic deployments with several layers of failover. Netflix also provides the technology to operate services responsibly with operational insight, peak performance, and security. We provide technologies for data (persistent & semi-persistent) that serve the real-time load to our 62 million members, as well as power the big data analytics that allow us to make informed decisions on how to improve our service. If you want to learn more, jump into any of the functional areas below to learn more.

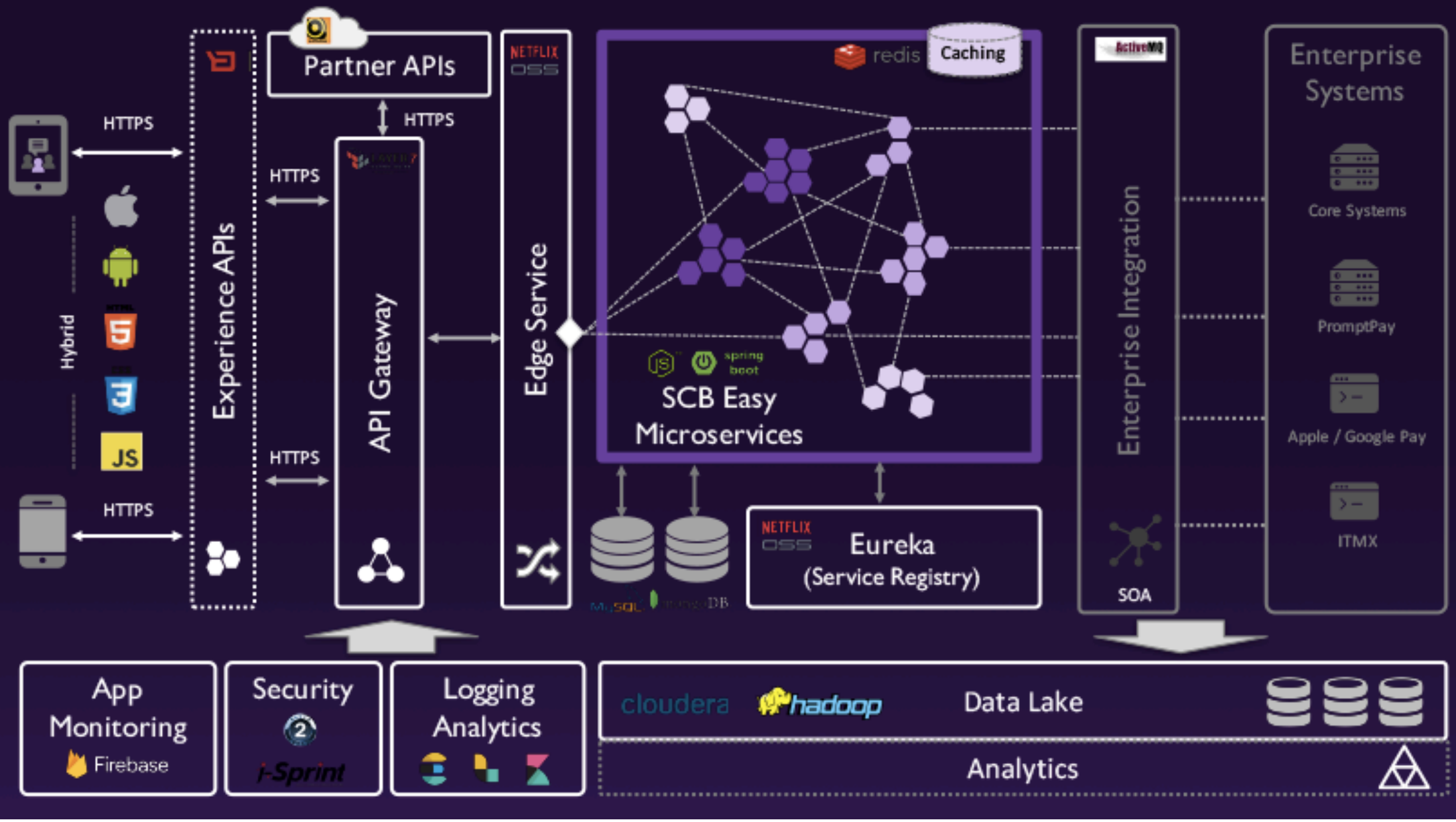
https://netflix.github.io

NetFlix Architecture




Edge
Middle Tier & Platform
Zuul
ELB
API
Playback
EVCache
Cassandra

SCB EASY MICROSERVICES ARCHITECTURE



https://www.blognone.com/node/95873

SCB EASY TECHNOLOGY

Applications

BACKBASE · JS · Swift · spring boot · 2 · Java · mongoDB · j-Sprint · HTML5 · MySQL · redis

Service Integration

LAYER 7 TECHNOLOGIES · ActiveMQ · NETFLIX OSS Eureka · NETFLIX OSS Zuul

Utilities

logstash · elasticsearch · kibana

DevOps & Automation

GitLab · Se · Jenkins · sonarqube · Confluence · JUnit 5 · appium

Big Data & Analytics

cloudera · hadoop · Firebase · splunk>

Delivery Partners

accenture · FJORD · KPMG · Deloitte. · GeniusSoft · Aware · Stream it consulting · g-able · MFEC

Hitachi Consulting

Microservices

- Application(s) powered by multiple services
- Small service with a single focus
- Lightweight communication mechanism
  Both client to service and service to service
- Technology agnostic API
- Independent data storage

Microservices

Micro sized services provide

- Efficiently scalable applications
- Flexible applications
- High performance applications

Microservices

- Independently changeable
- Independently deployable
- Distributed transactions
- Centralized tooling for management

Why Now?

- Need to respond to change quickly
- Need for reliability
- Business domain-driven design
- Automated test tools
- Release and deployment tools
- On-demand hosting technology
- On-line cloud services
- Need to embrace new technology
- Asynchronous communication technology
- Simpler server side and client side technology

Microservices Pros

- Shorter development times
- Reliable and faster deployment
- Enables frequent updates
- Decouple the changeable parts
- Security
- Increased uptime
- Fast issue resolution
- Highly scalable and better performance
- Better ownership and knowledge
- Right technology
- Enables distributed teams

PolyglotPersistence

Martin Fowler
16 November 2011

In 2006, my colleague Neal Ford coined the term Polyglot Programming, to express the idea that applications should be written in a mix of languages to take advantage of the fact that different languages are suitable for tackling different problems. Complex applications combine different types of problems, so picking the right language for the job may be more productive than trying to fit all aspects into a single language.

Over the last few years there's been an explosion of interest in new languages, particularly functional languages, and I'm often tempted to spend some time delving into Clojure, Scala, Erlang, or the like. But my time is limited and I'm giving a higher priority to another, more significant shift, that of the DatabaseThaw. The first drips have been coming through from clients and other contacts and the prospects are enticing. I'm confident to say that if you starting a new strategic enterprise application you should no longer be assuming that your persistence should be relational. The relational option might be the right one - but you should seriously look at other alternatives.

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

Polyglot Persistence

Twelve Factor

Twelve Factor App

- Source: http://12factor.net
- Initial proposed to build SaaS apps for Heroku
- Principles translate well to cloud and container native application
- Translate to microservices

Principle 1: Codebase

One codebase tracked in revision control, many deploys

Principle 2: Dependencies

Explicitly declare and isolate dependencies

Principle 3: Config

Store config in the environment

Principle 4: Backing Services

Treat backing services as attached resources

Principle 5: Build Release Run

Strictly separate build and run stages

Principle 6: Processes

Execute the app as one or more stateless processes

Principle 7: Port Binding

Export services via port binding

Principle 8: Concurrency

Scale out via the process model

Principle 9: Disposability

Maximize robustness with fast startup and graceful shutdown

Principle 10: Dev/Prod Parity

Keep development, staging, and production as similar as possible

Principle 11: Logs

Treat logs as event streams

Principle 12: Admin Processes

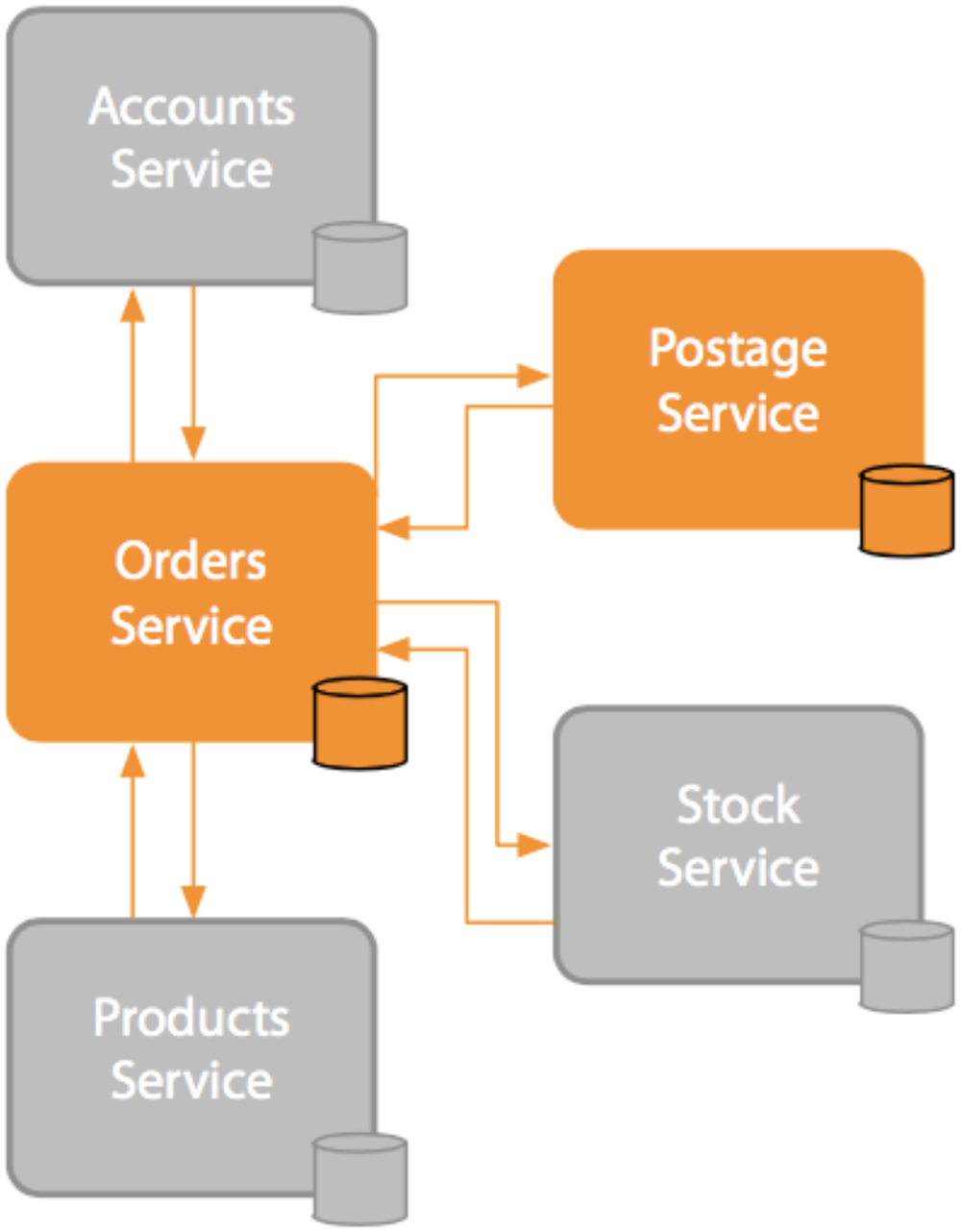
Run admin/management tasks as one-off processes

Design Principles

Design Principles

High Cohesion	Autonomous	Business Domain Centric
Resillience	Observable	Automation

High Cohesion

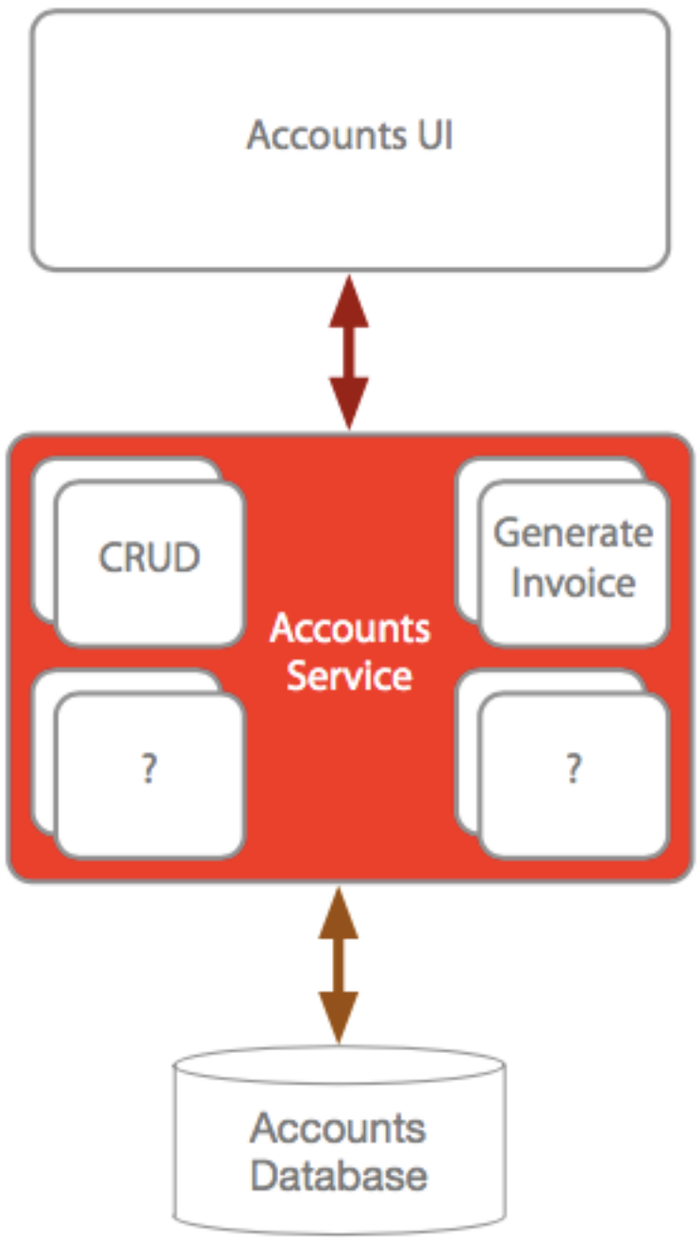


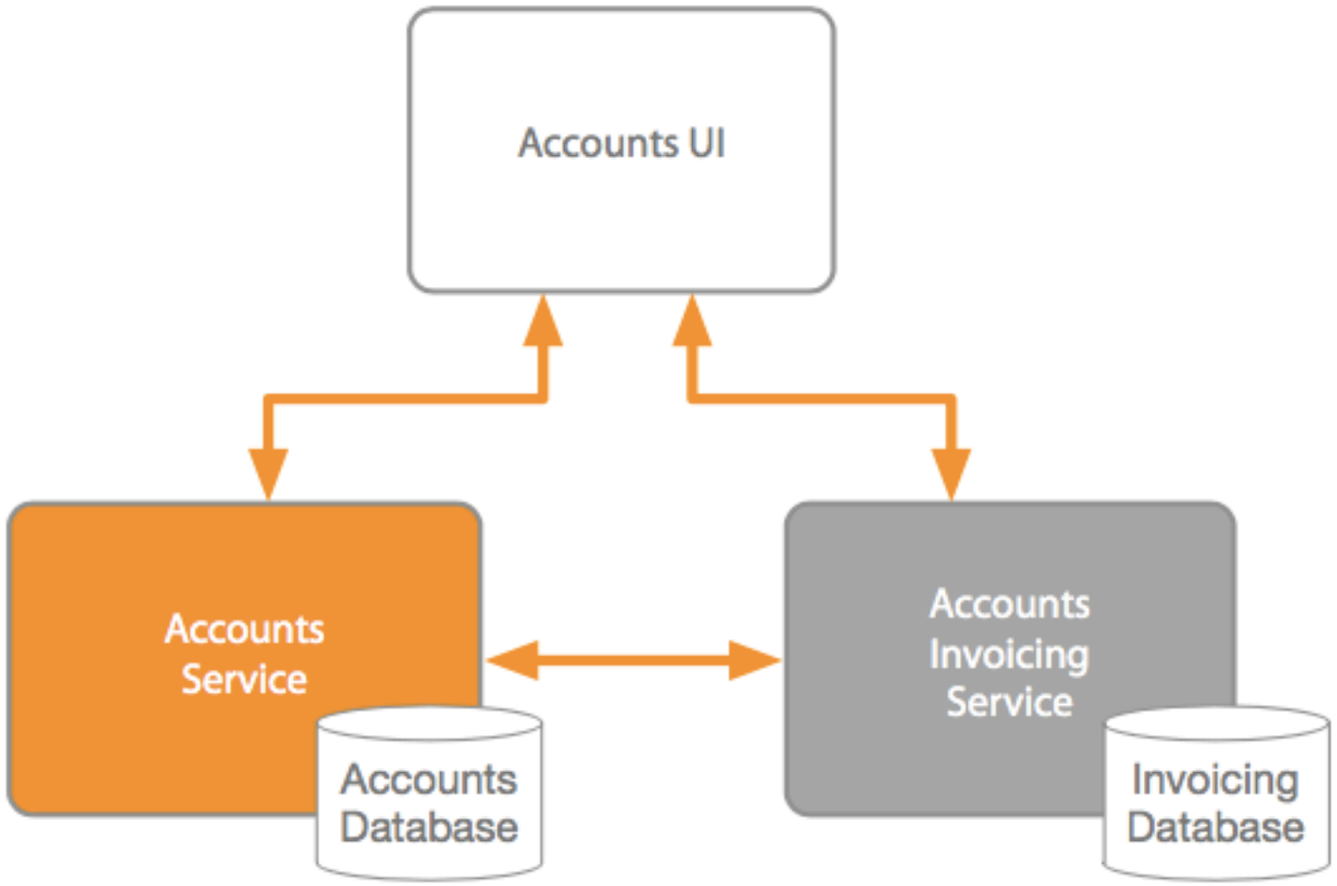
- Single focus
- Single responsibility
  - SOLID principle
  - Only change for one reason
- Reason represents
  - A business function
  - A business domain
- Encapsulation principle
- OOP principle
- Easily rewritable code

High Cohesion

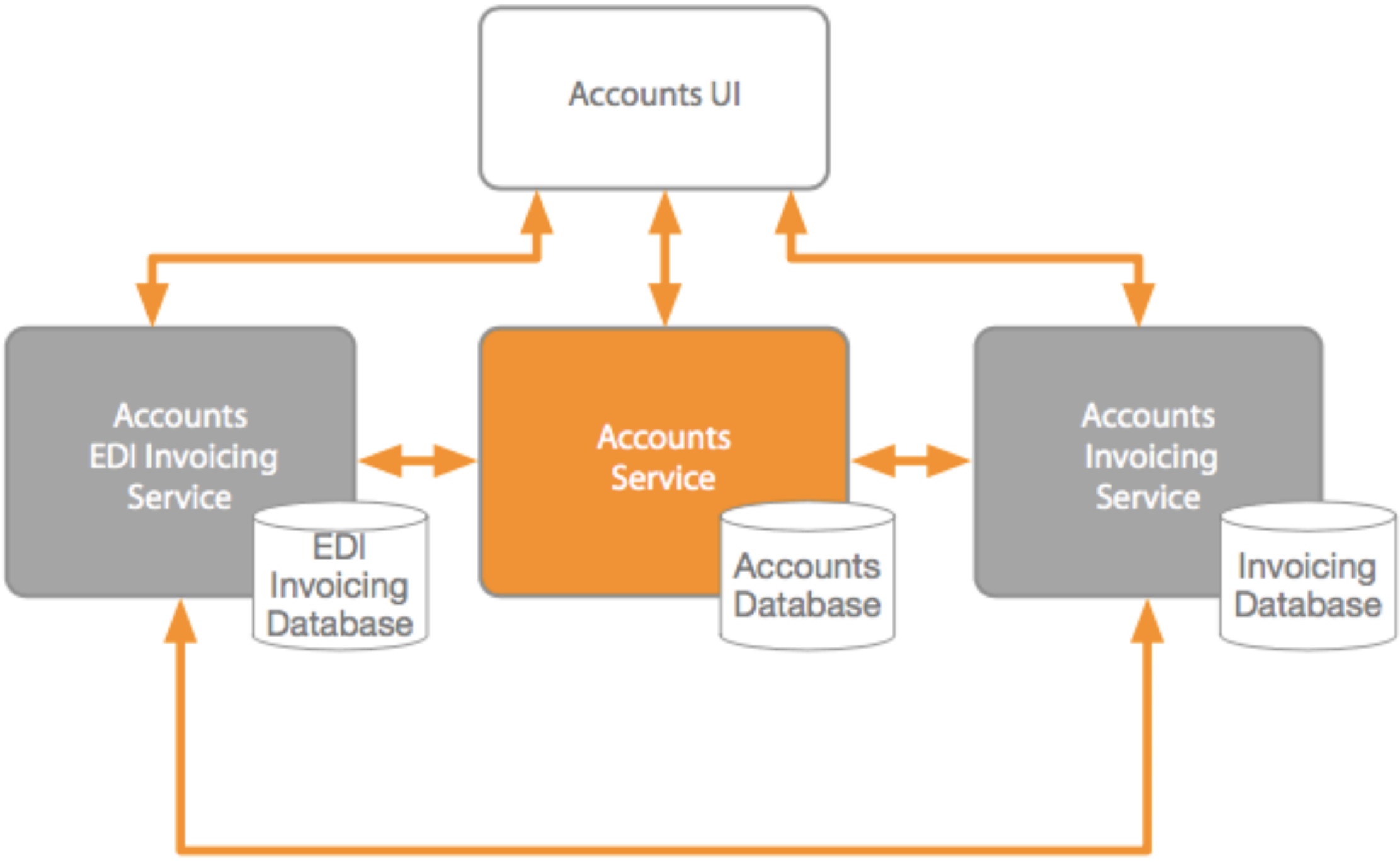
- Identify a single focus
  - Business function
  - Business domain
- Split into finer grained services
- Avoid “Is kind of the same”
- Don't get lazy!
- Don’t be afraid to create many services

High Cohesion

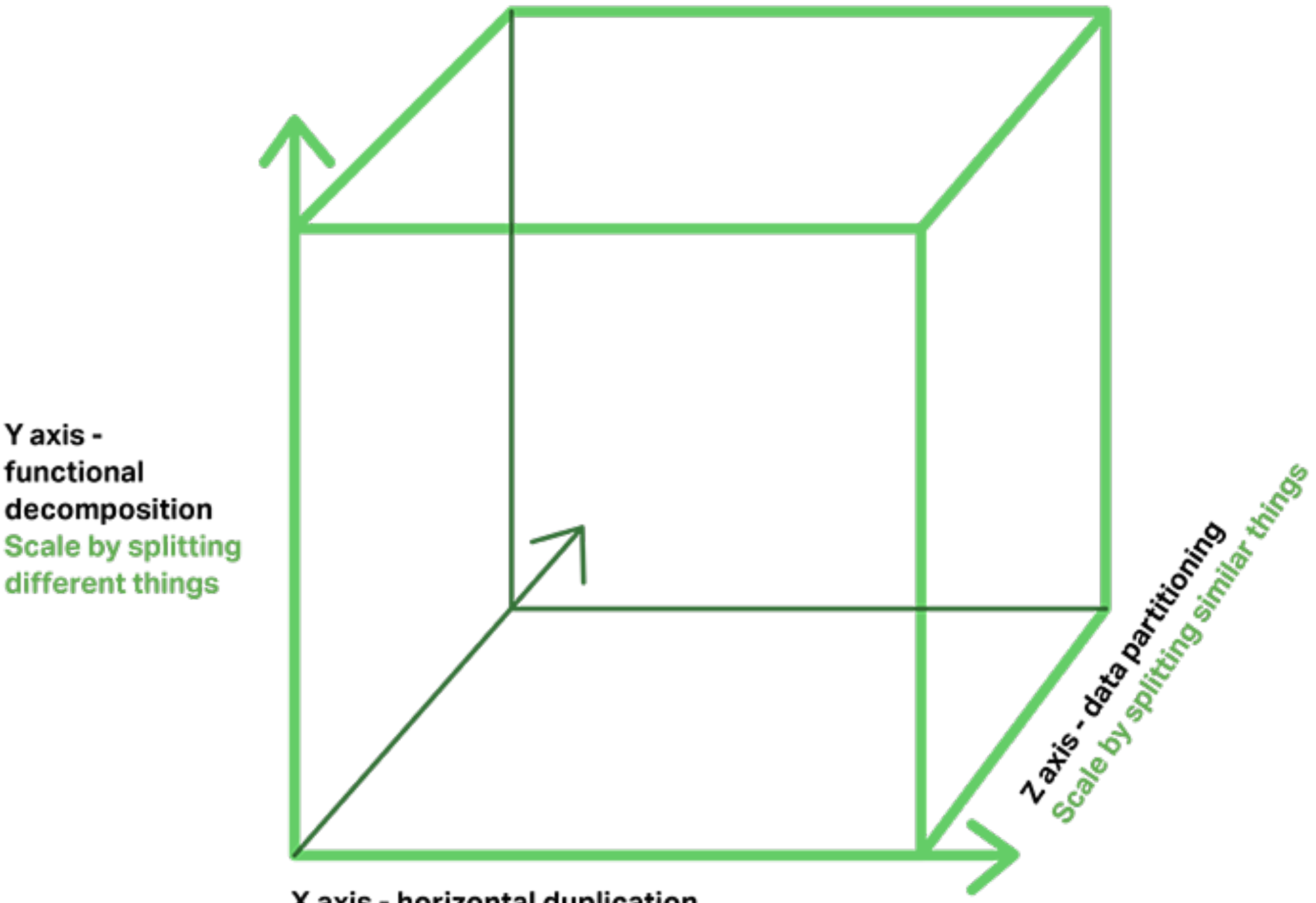




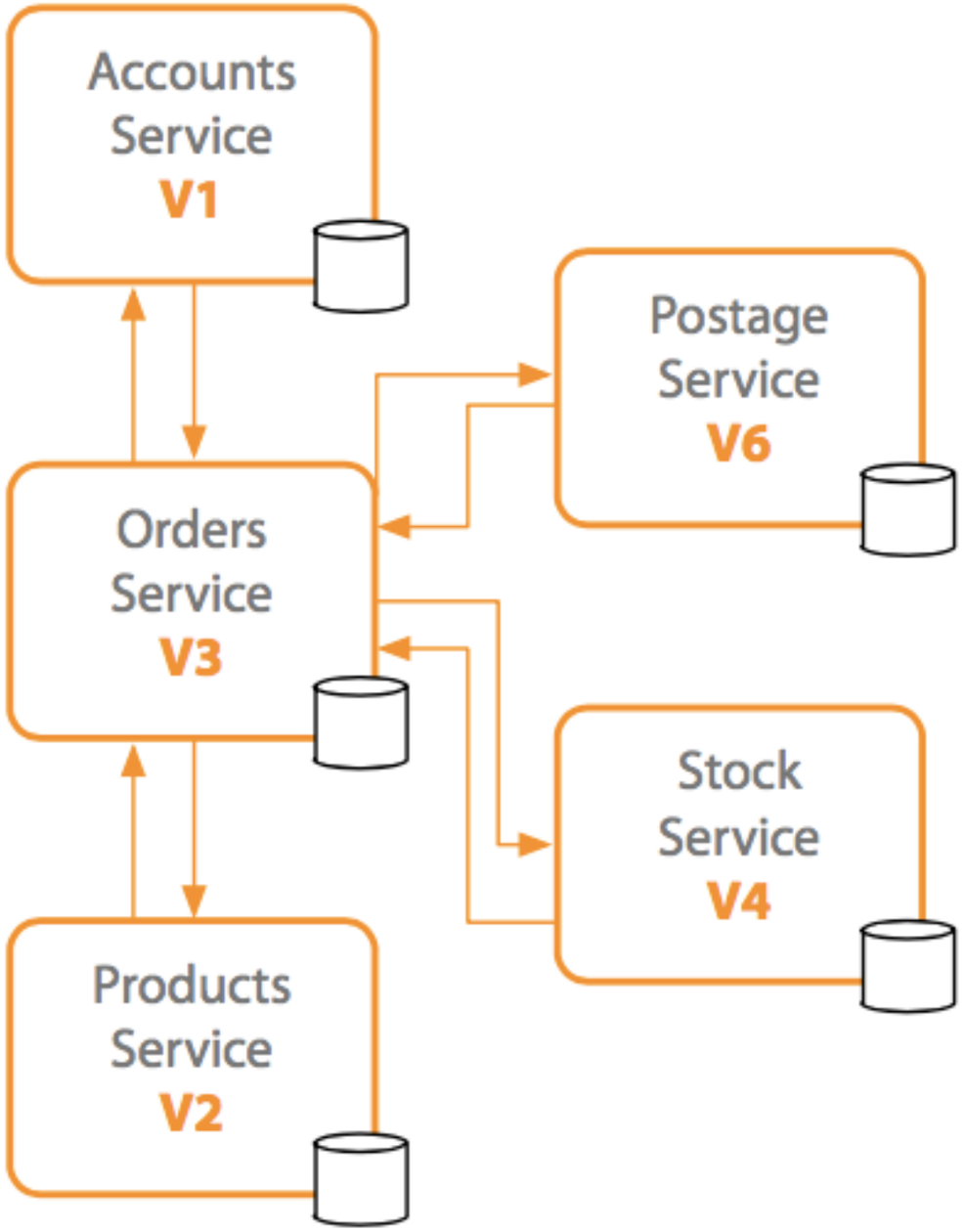
High Cohesion



The Art of Scalability

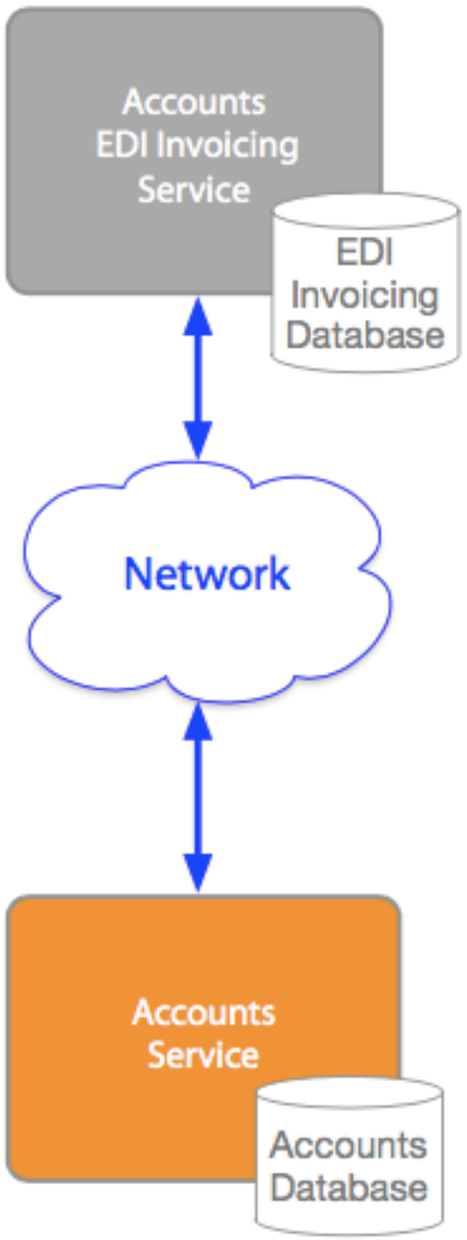


Autonomous



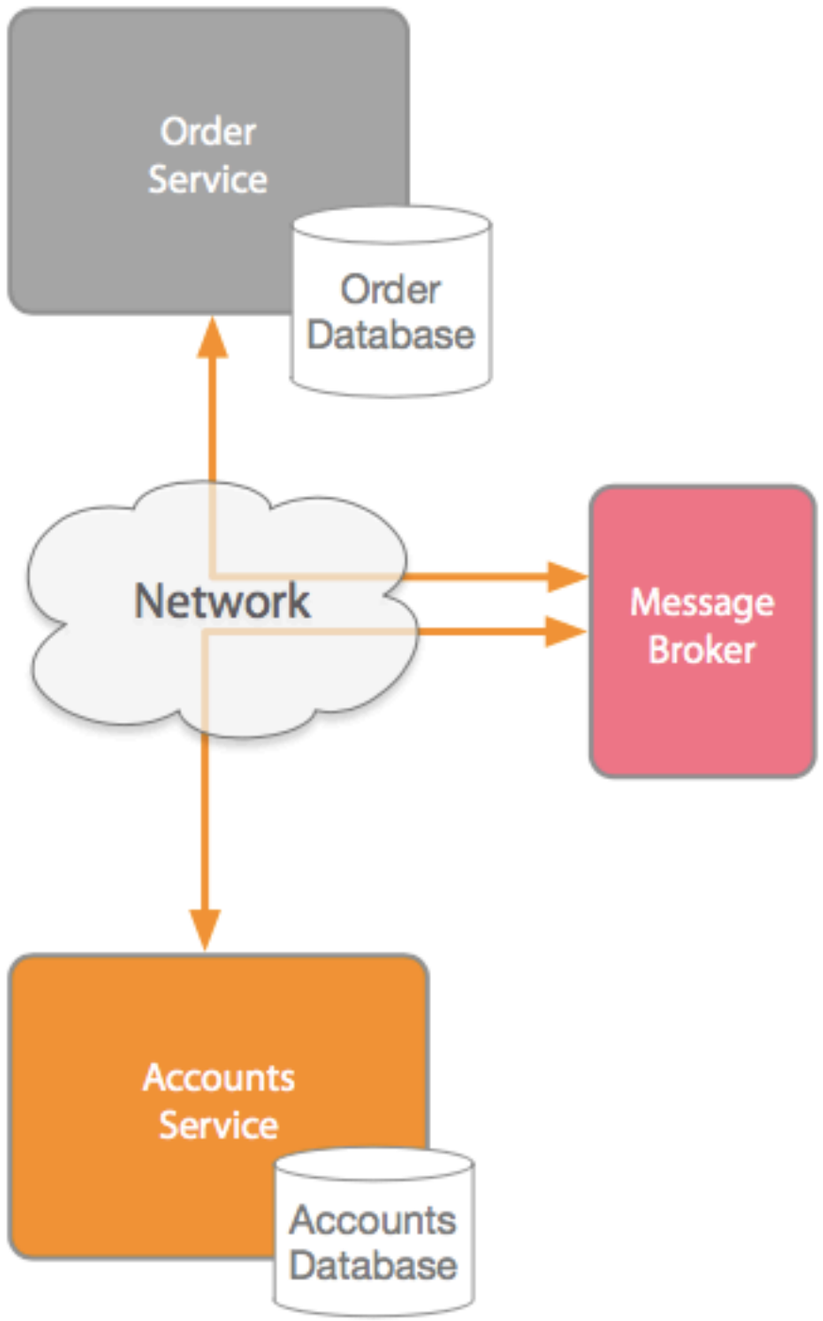
- Loose coupling
- Honor contracts and interfaces
- Stateless
- Independently changeable
- Independently deployable
- Backwards compatible
- Concurrent development

Autonomous



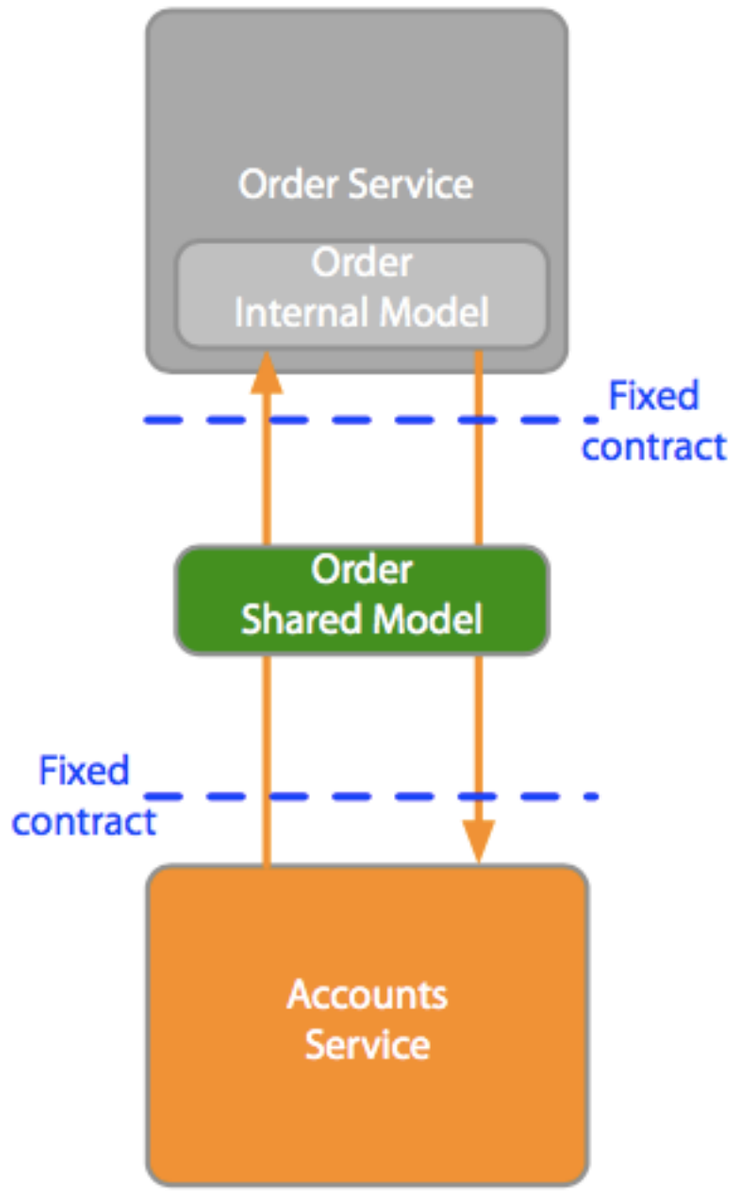
- Communication by network
- Technology agnostic API
- Avoid client libraries
- Contracts between services
- Avoid chatty exchanges between services
- Avoid sharing between services

Autonomous



- Communication by network
  - Synchronous
  - Asynchronous
    - Publish events
    - Subscribe to events

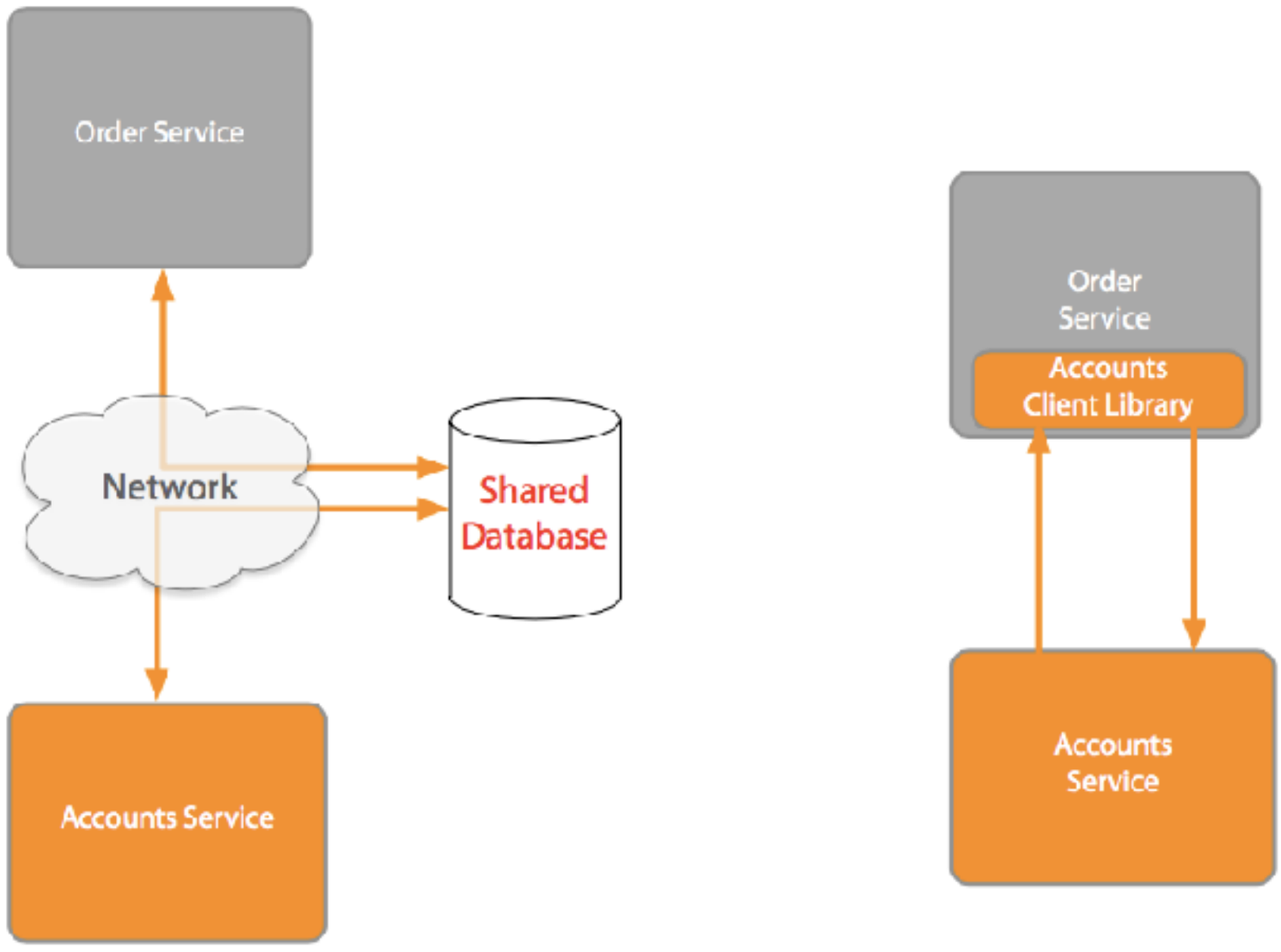
Autonomous



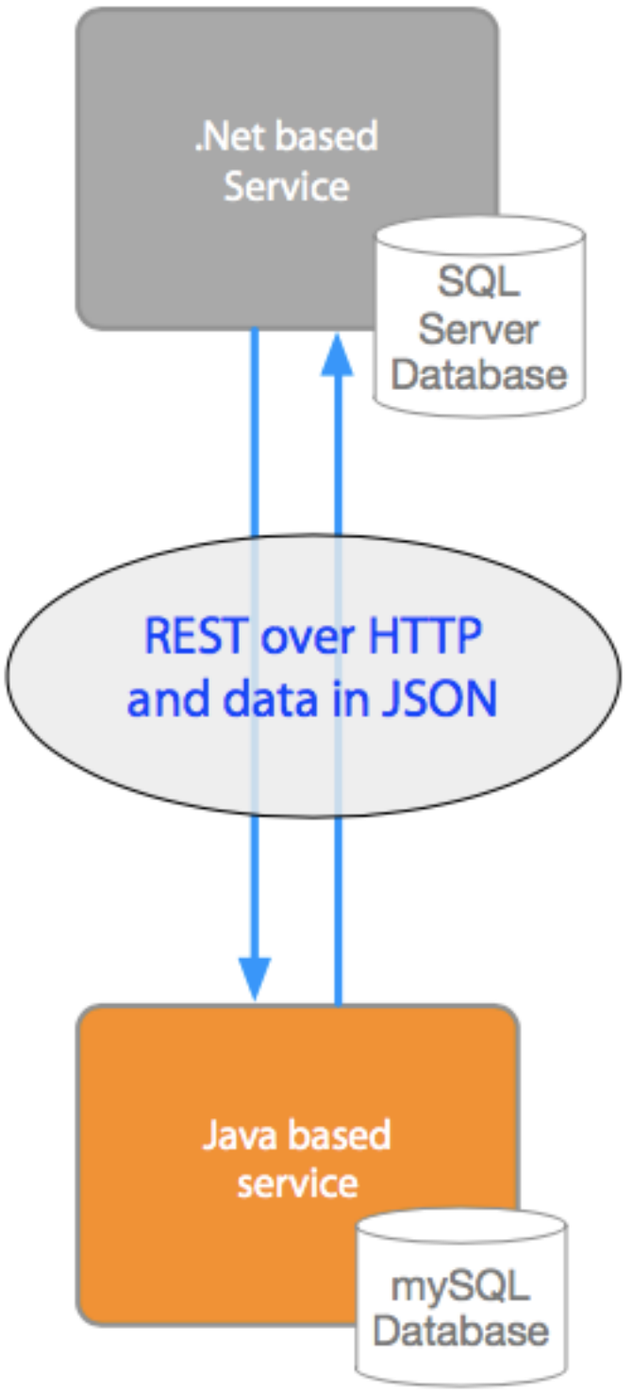
- Contracts between services
  - Fixed and agreed interfaces
  - Shared models
  - Clear input and output

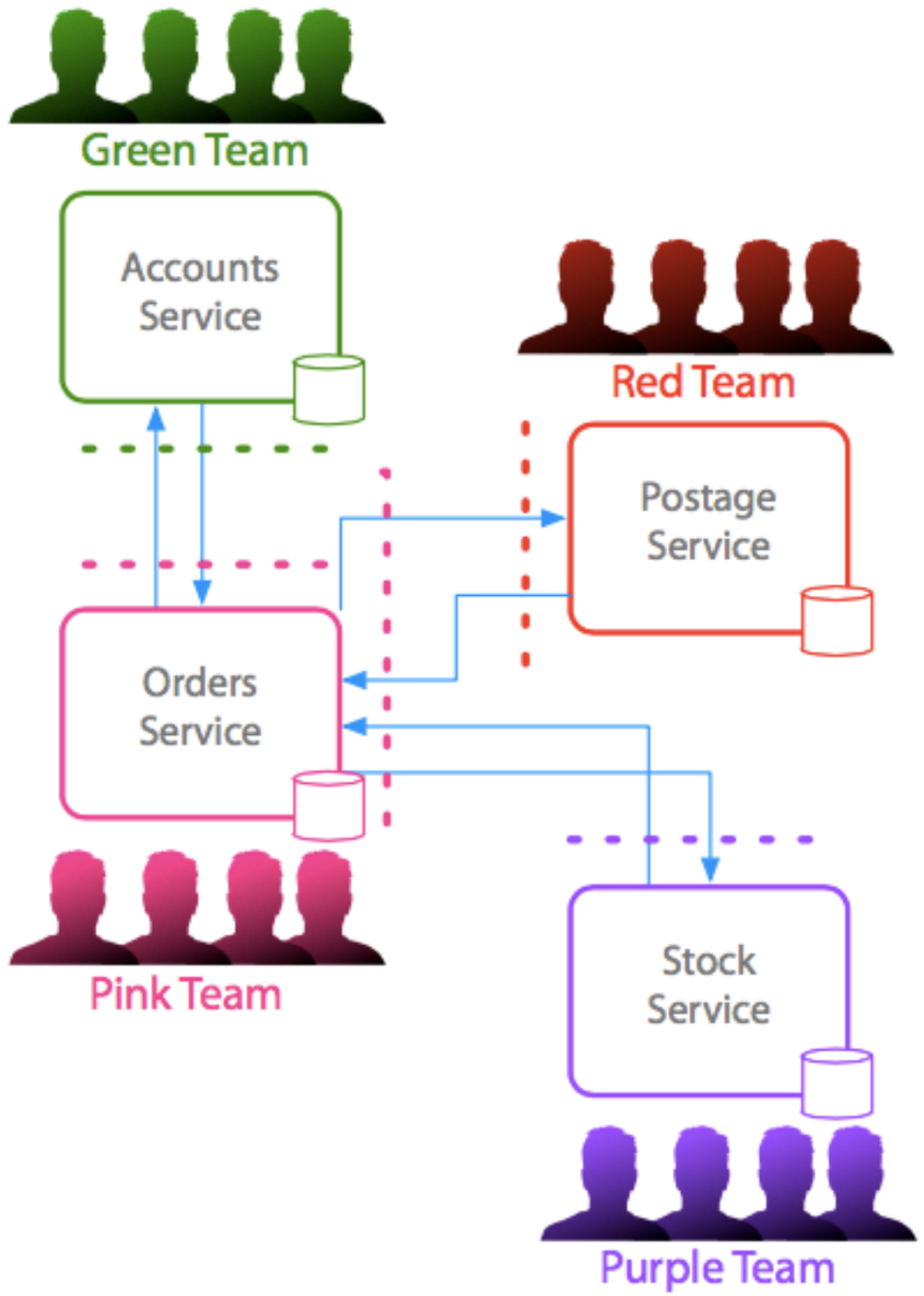
Avoid

Avoid sharing between services



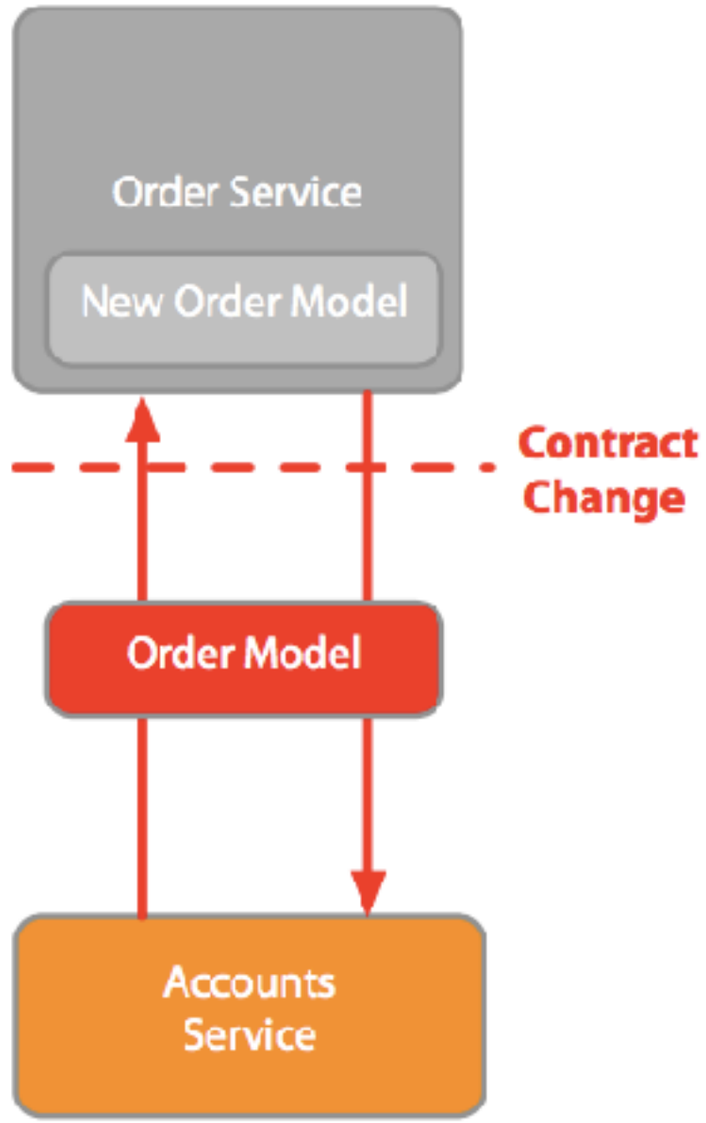
Autonomous


.Net based
Service
SQL
Server
Database
REST over HTTP
and data in JSON
Java based
service
mySQL
Database


Green Team
Accounts
Service
Red Team
Postage
Service
Orders
Service
Pink Team
Stock
Service
Purple Team

Ownership by team

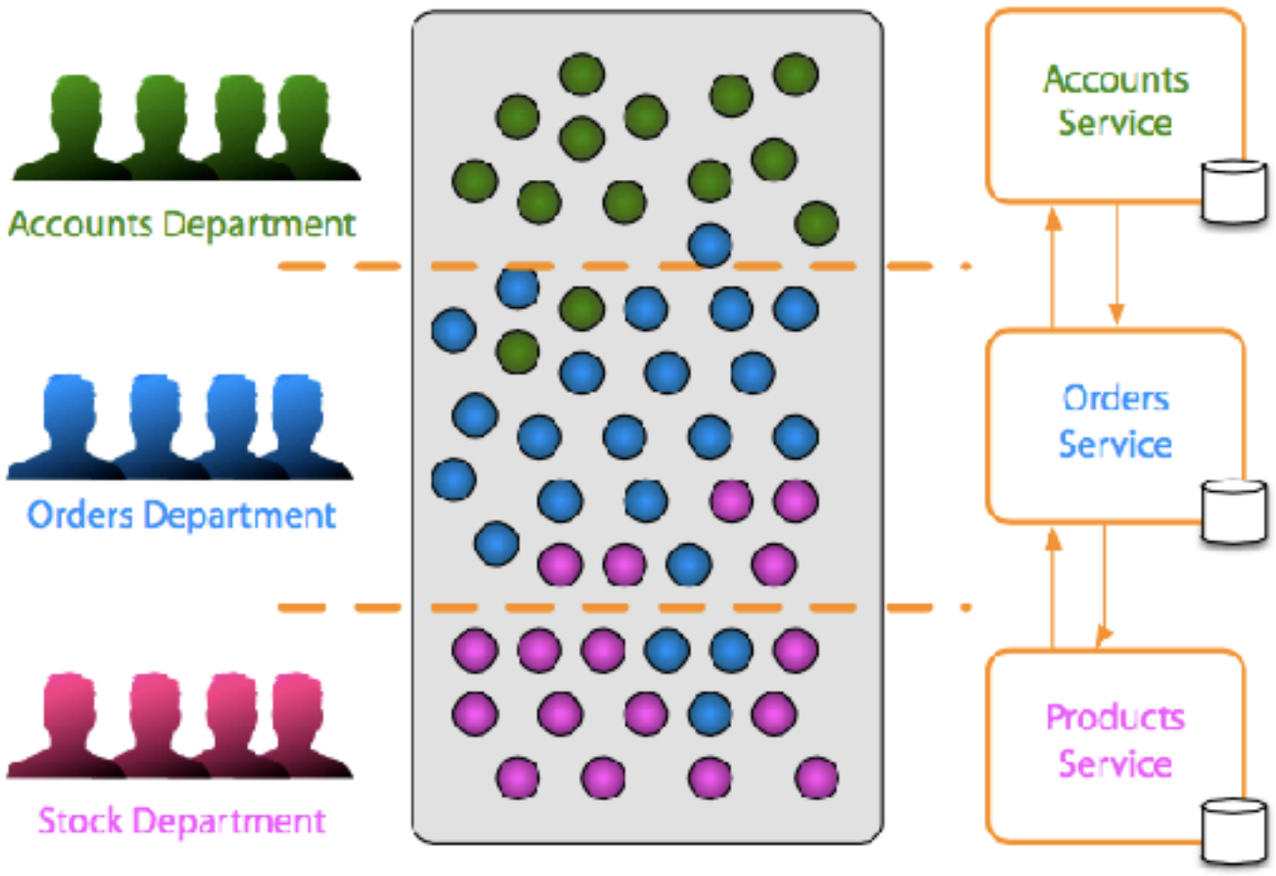
- Responsibility to make autonomous
- Agreeing contracts between teams
- Responsible for long-term maintenance
- Collaborative development
  - Communicate contract requirements
  - Communicate data requirements
- Concurrent development



Versioning

- Avoid breaking changes
- Backwards compatibility
- Integration tests
- Have a versioning strategy
  - Concurrent versions (Old and new)
  - Semantic versioning (Major.Minor.Patch , e.g. 15.1.2)
  - Coexisting endpoints (/V2/customer/)

Business Domain Centric



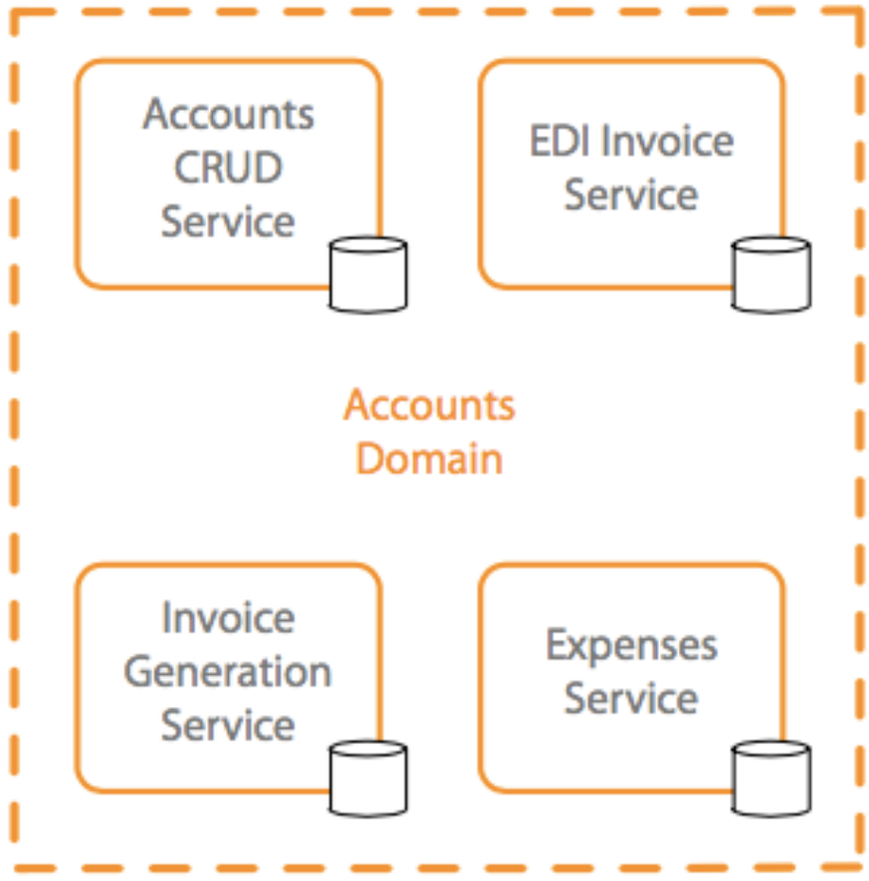
- Service represents business function
- Scope of service
- Bounded context from DDD
- Identify boundary
- Shuffle code if required

Business Domain Centric

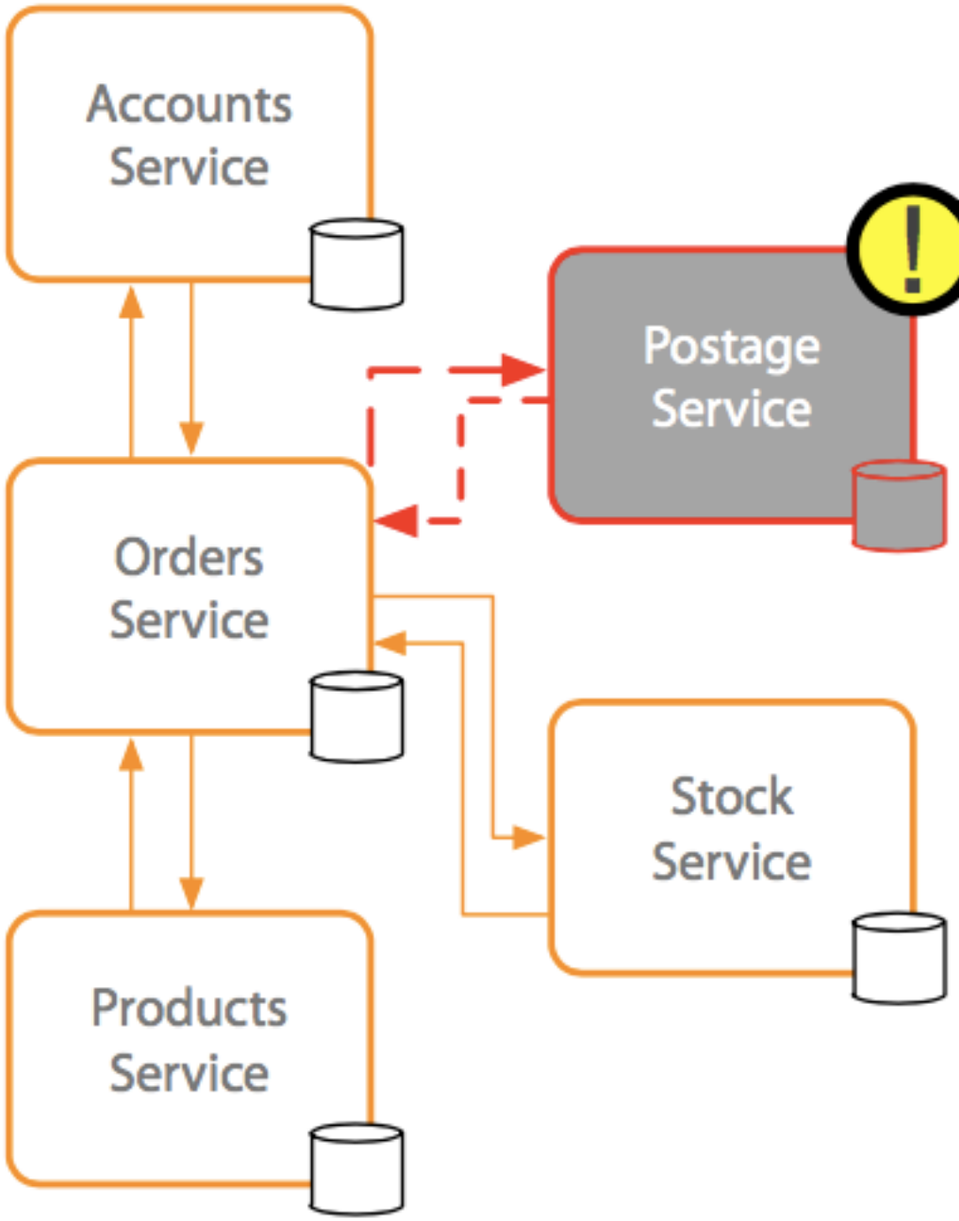
- Identify business domains in a coarse manner
- Review sub groups of business functions or areas
- Review benefits of splitting further
- Agree a common language

Business Domain Centric

- Fix incorrect boundaries
  - Merge or split
- Explicit interfaces for outside world
- Splitting using technical boundaries
  - Service to access archive data
  - For performance tuning

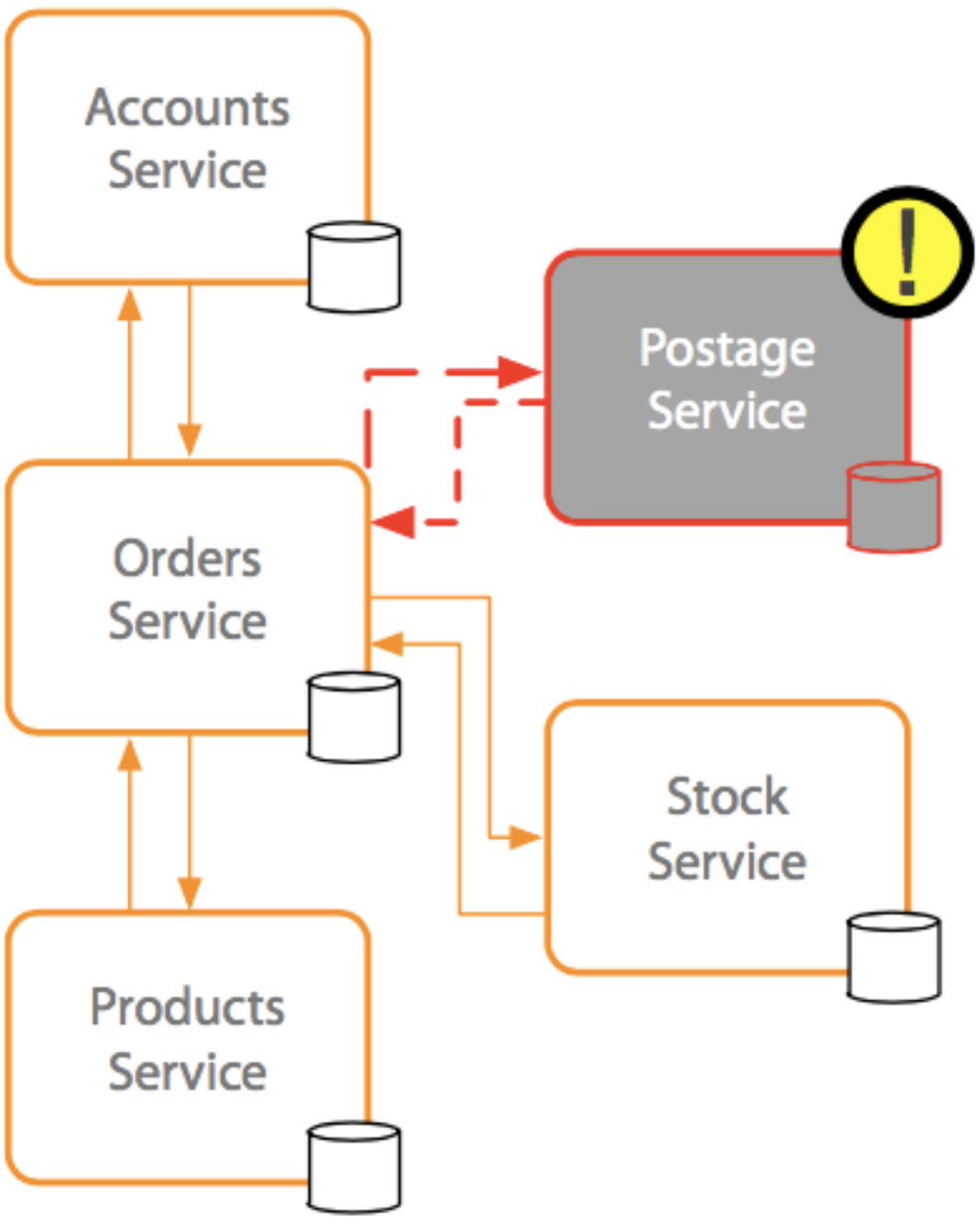


Resilience



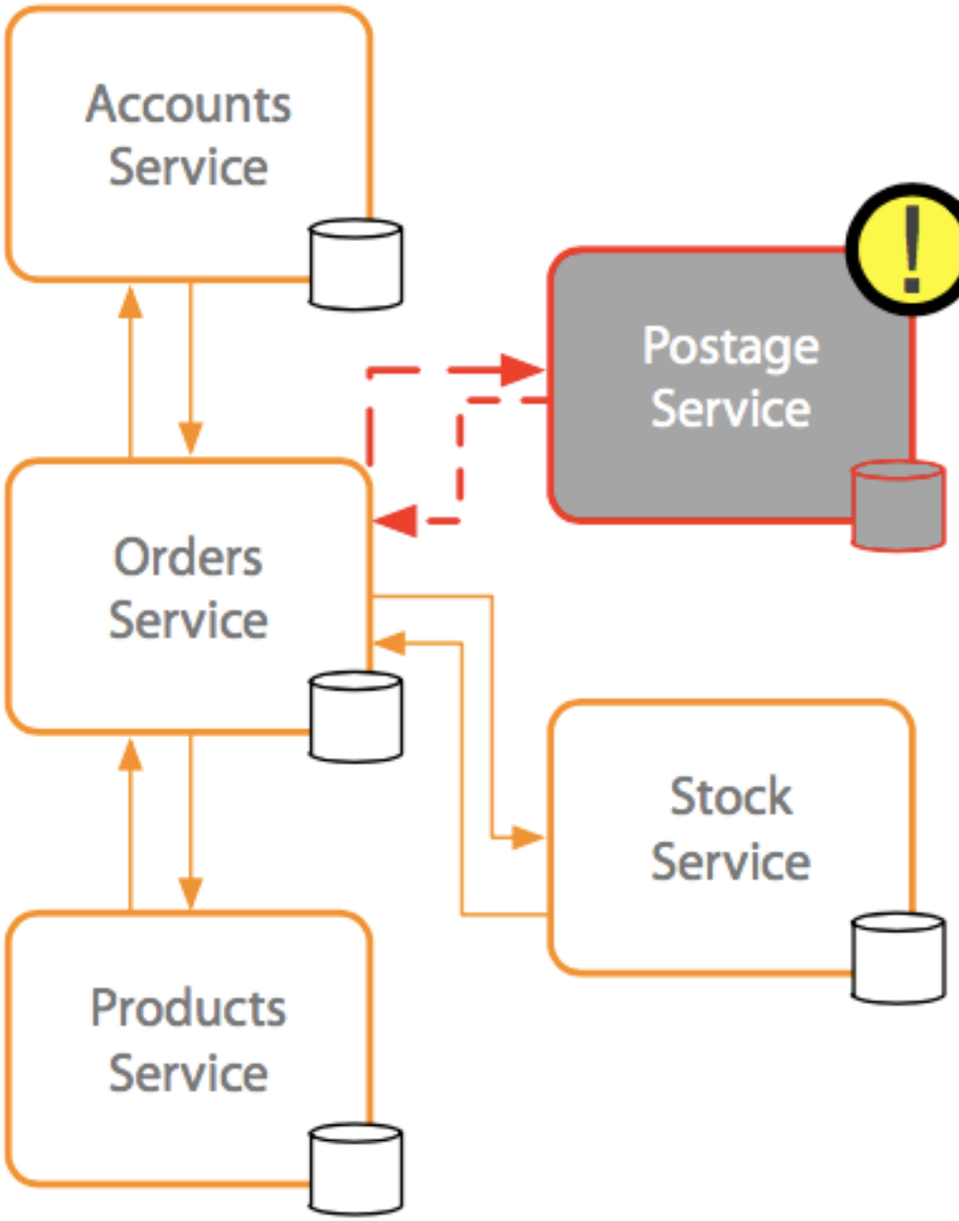
- Embrace failure
  - Another service
  - Specific connection
  - Third-party system
- Degrade functionality
- Default functionality
- Multiple instances
  - Register on startup
  - Deregister on failure

Resilience



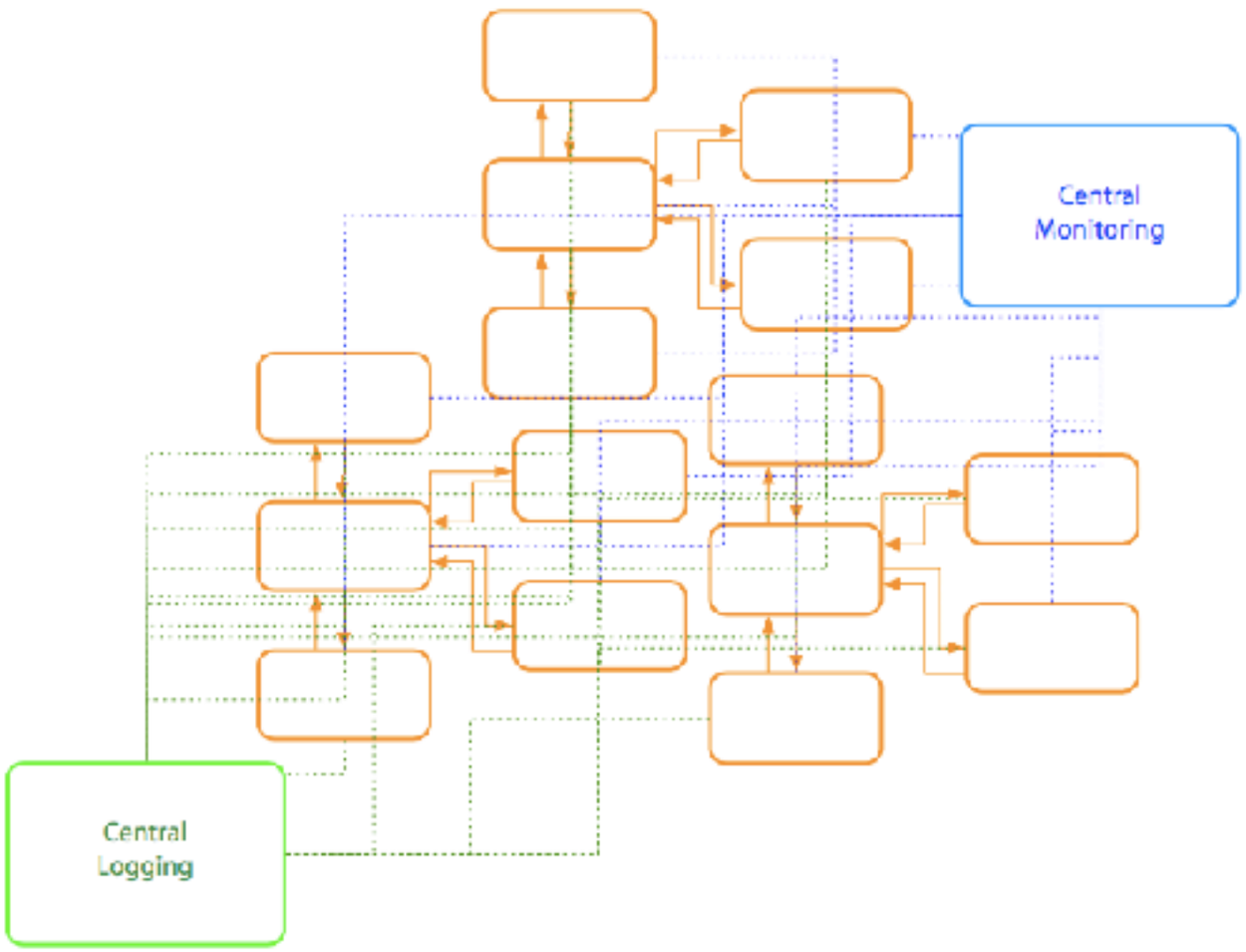
- Types of failure
  - Exceptions\Errors
  - Delays
  - Unavailability
- Network issues
  - Delay
  - Unavailability
- Validate input
  - Service to service
  - Client to service

Resilience



- Design system to fail fast
- Design System for known failures
- Use timeouts
  - Use for connected systems
  - Timeout our requests after a threshold
  - Service to service
  - Service to other systems
  - Standard timeout length
  - Adjust length on a case by case basis
- Network outages and latency
- Monitor timeout
- Log timeouts

Observable



- System Health
  - Status
  - Logs
  - Errors
- Centralized monitoring
- Centralized logging

Centralize Monitoring

- Real-time monitoring
- Monitor the host (CPU, memory, disk usage, etc.)
- Expose metrics within the services
  - Response times
  - Timeouts
  - Exceptions and errors
- Business data related metrics
  - Number of orders
  - Average time from basket to checkout

Centralize Monitoring

- Collect and aggregate monitoring data
  - Monitoring tools that provide aggregation
  - Monitoring tools that provide drill down options
- Monitoring tool that can help visualize trends
- Monitoring tool that can compare data across servers
- Monitoring tool that can trigger alerts

Centralize Logging

- When to log
  - Startup or shutdown
  - Code path milestones
  - Requests, responses and decisions
  - Timeouts, exceptions and errors
- Traceable distributed transactions
  - Correlation ID (Passed service to service)

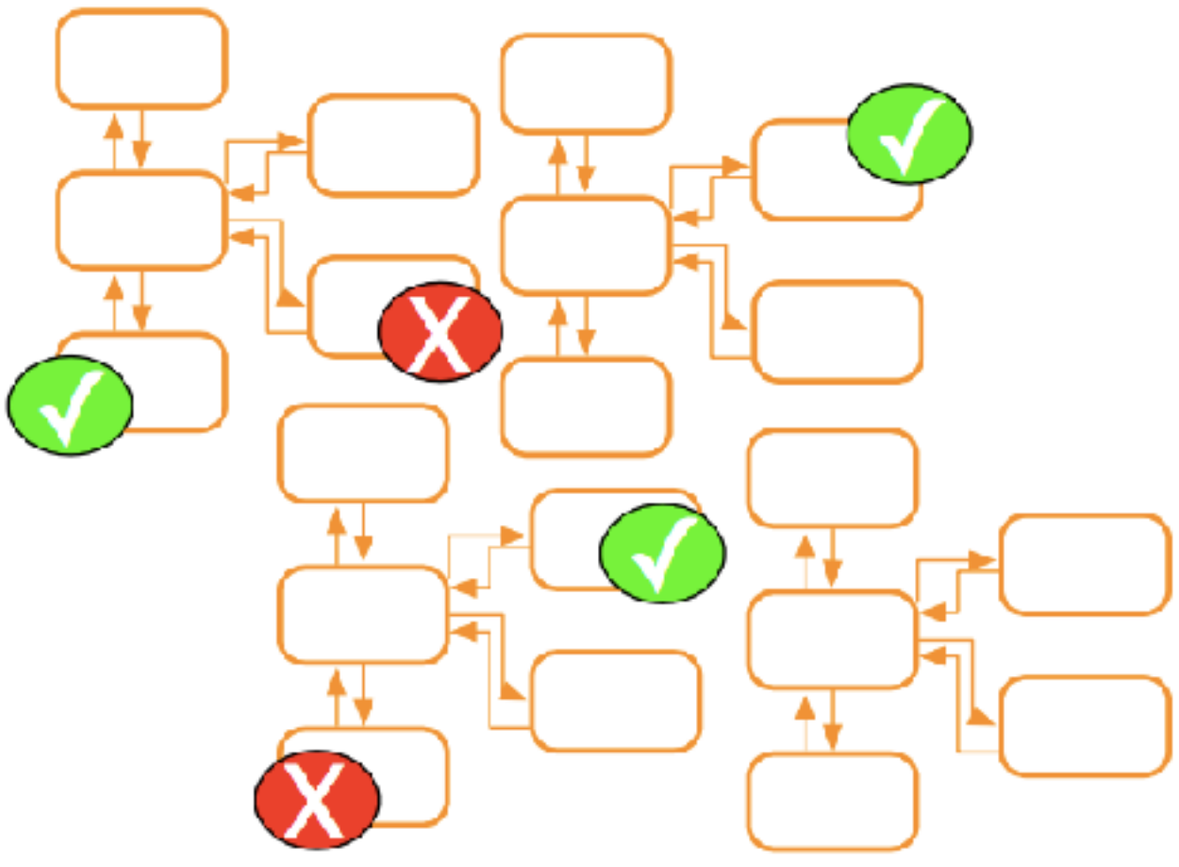
Structure Logging

- Level
  - Information
  - Error
  - Debug
  - Statistic
- Date and time
- Correlation ID
- Host name
- Service name and service instance
- Message

Why Observable

- Distributed transactions
- Quick problem solving
- Quick deployment requires feedback
- Data used for capacity planning
- Data used for scaling
- Whats actually used
- Monitor business data

Automation



- Tools to reduce testing
  - Manual regression testing
  - Time taken on testing integration
  - Environment setup for testing
- Tools to provide quick feedback
  - Integration feedback on check in
  - Continuous Integration
- Tools to provide quick deployment
  - Pipeline to deployment
  - Deployment ready status
  - Automated deployment
  - Reliable deployment
  - Continuous Deployment

Why Automation

- Distributed system
- Multiple instances of services
- Manual integration testing too time consuming
- Manual deployment time consuming and unreliable

Automation

Continuous Intregration

Continuous Deployment

Continuous Integration

- Work with source control system
- Automatic after check-in
- Unit tests and Integration tests required
- Ensure to quality of check-in
  - Code compiles
  - Test pass
  - Changes integrate
  - Quick feedback

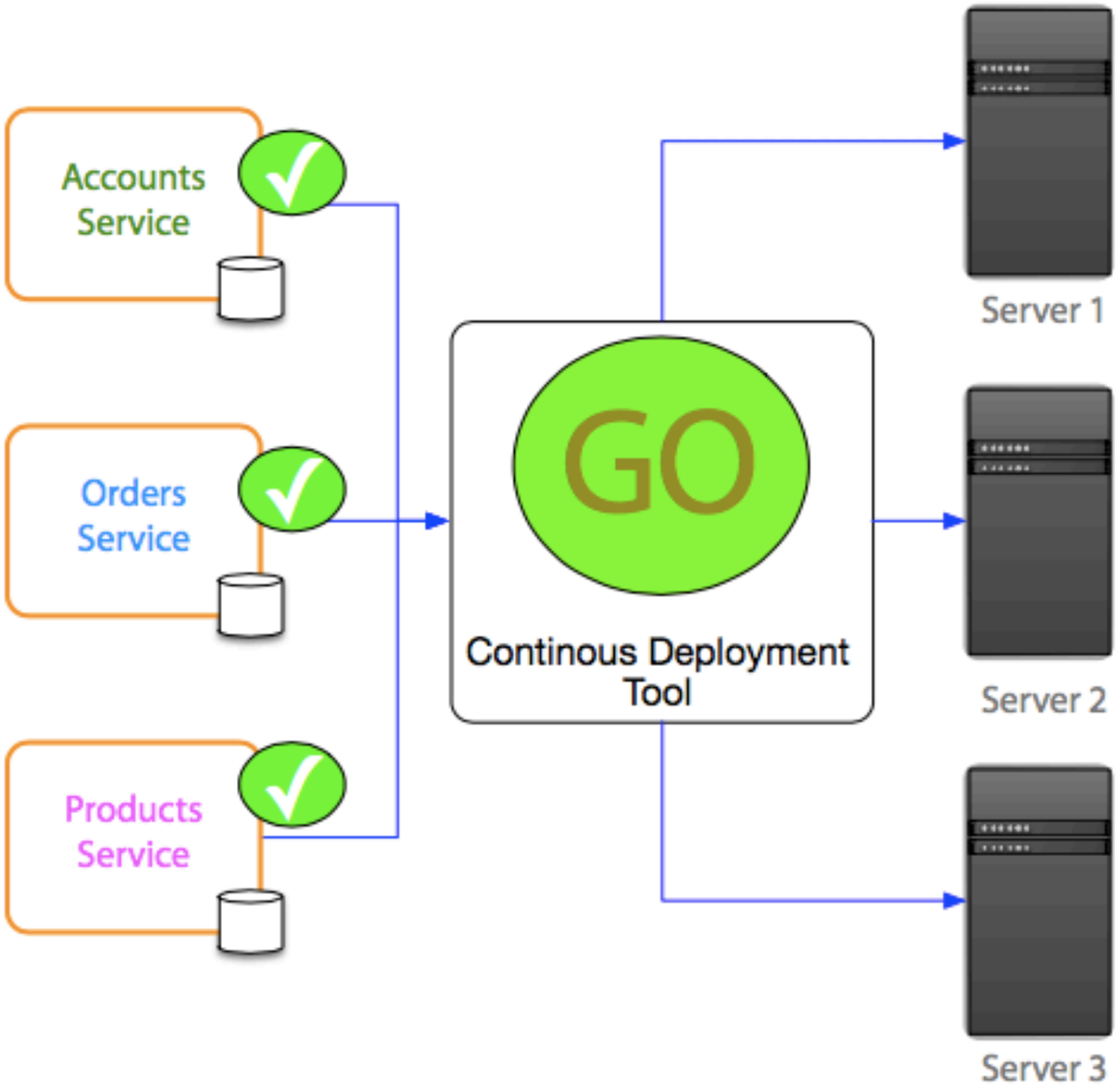
Continuous Integration

- Urgency to fix quickly
- Creation of build
- Build ready for test team
- Build ready for deployment

Continuous Deployment

- Automate software deployment
  - Configure once
  - Works with CI tools
  - Deployable after check in
  - Reliably released at anytime
- Benefits
  - Quick to market
  - Reliable deployment
  - Better customer experience

Continuous Deployment



Design Principle

High Cohesion
Single thing done well
Single focus

Autonomous
Independently changeable
Independently deployable

Business Domain Centric
Represent business function or represent a business domain

Resillience
Embrace failure
Default or Degrade functionality

Observable
See System Health
Centralize logging and monitoring

Automation
Tools for testing and feedback
Tools for deployment

Approach

High Cohesion
Keeps splitting service until it only has one reason to change

Autonomous
Loosely Coupled system
Versioning strategy
Microservice Ownership by team

Business Domain Centric
Course grain business domain
Subgroups into functions and areas

Resillience
Design for known failures
Design for fail fast (recover fast)

Observable
Real-time centralized monitoring
Centralized logging

Automation
Continuous Integration
Cntinuous Deployment

Technology for MicroServices

Communication

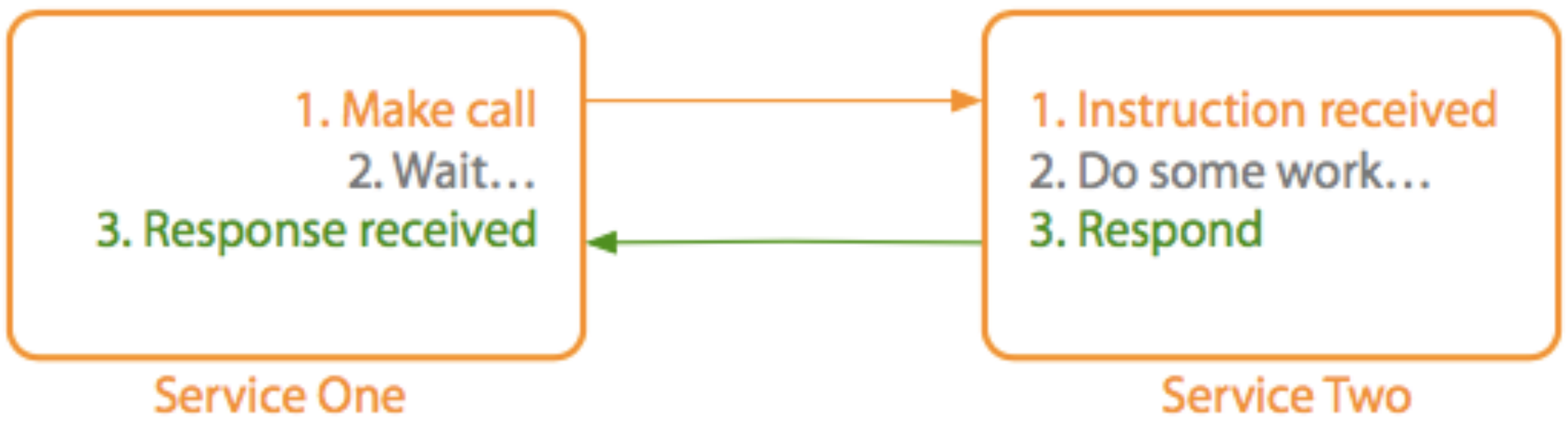
Inter-Process Communications

Synchronous
One-To-One

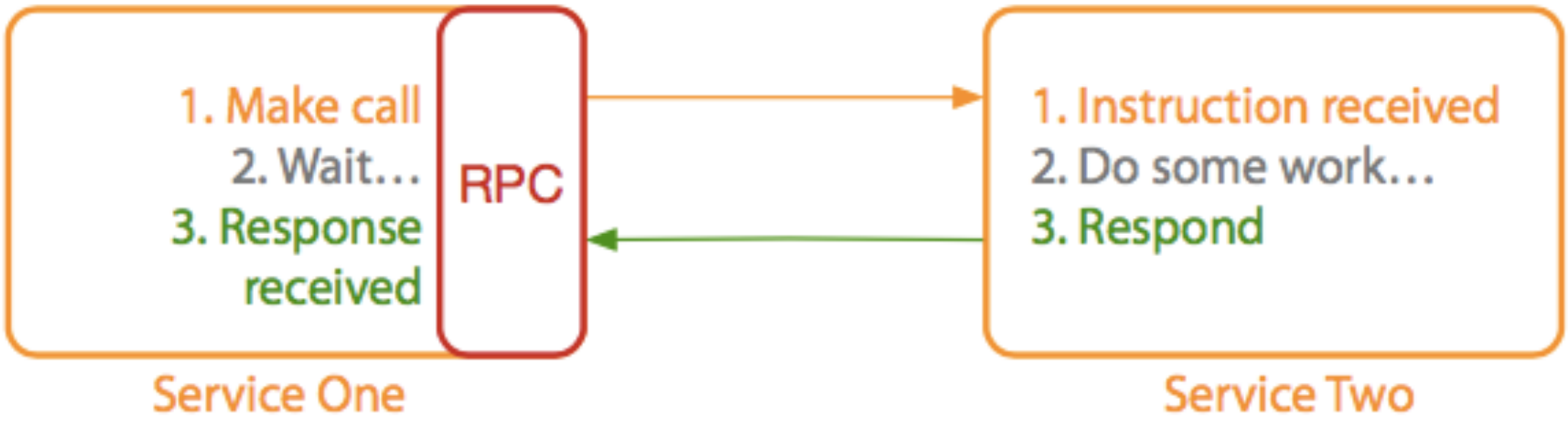
Asynchronous
One-To-Many

Synchronous

- Request response communication
  - Client to service
  - Service to service
  - Service to external



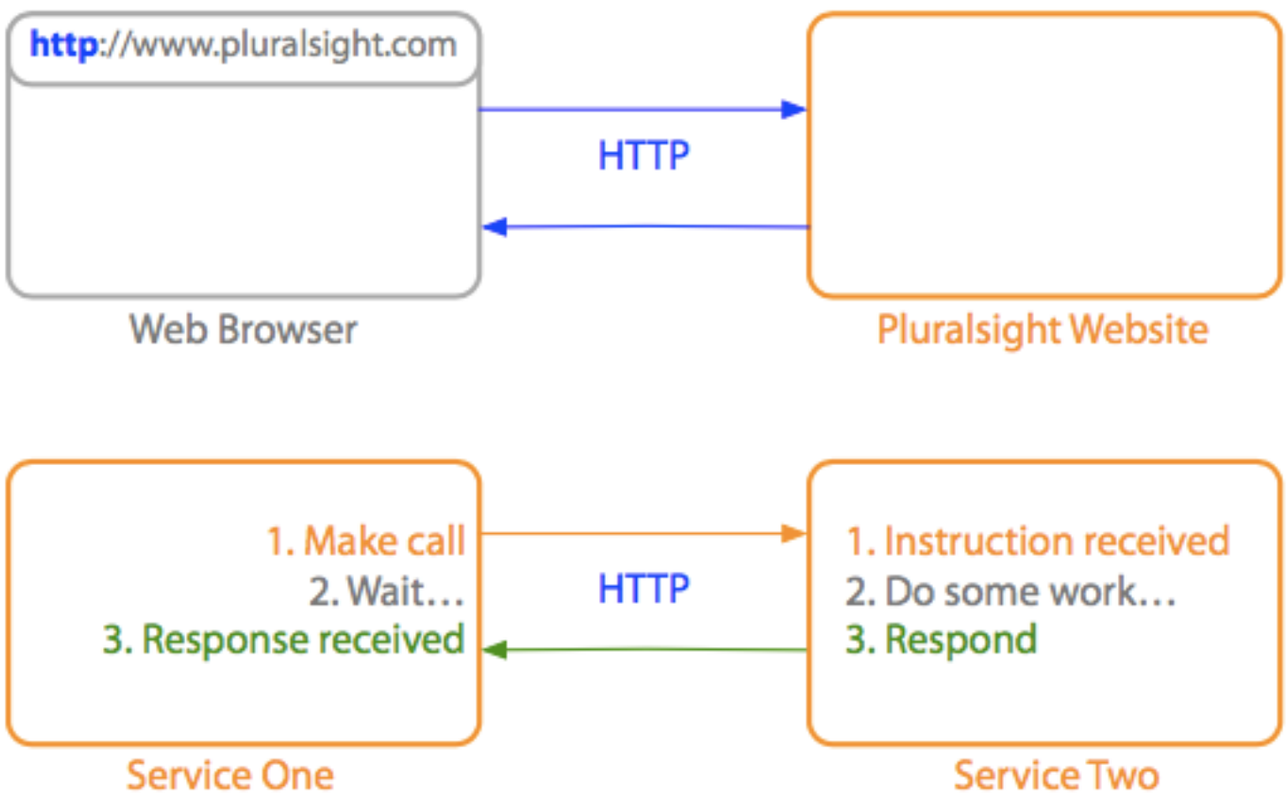
Remote Procedure Call



Sensitive to Change

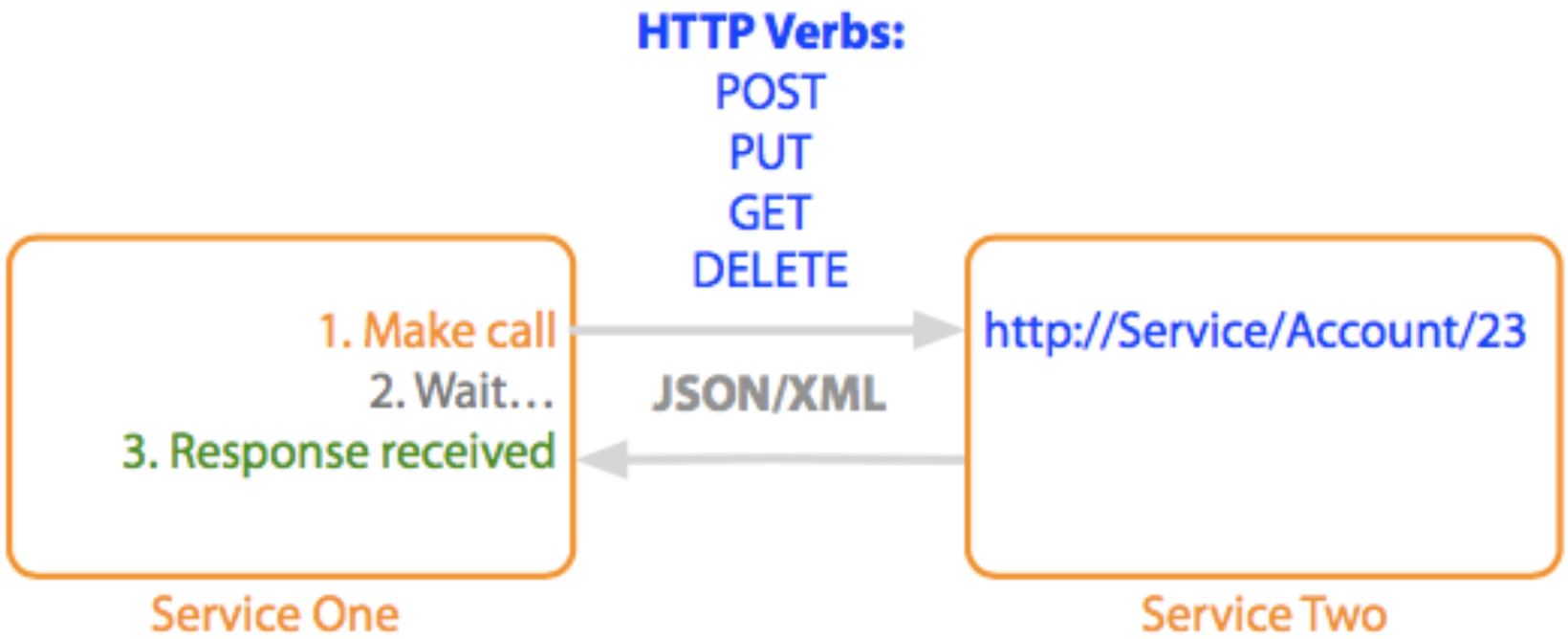
HTTP

- Work across the internet
- Firewall friendly

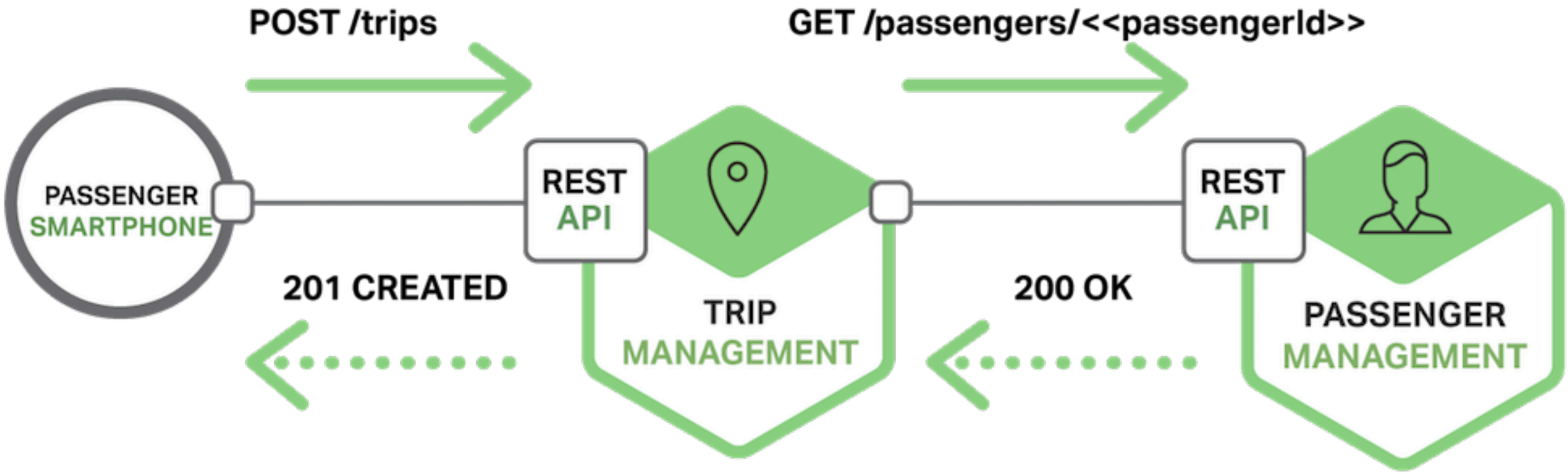


RESTful Service

- CRUD using HTTP verbs
- Natural decoupling
- Open communication protocol



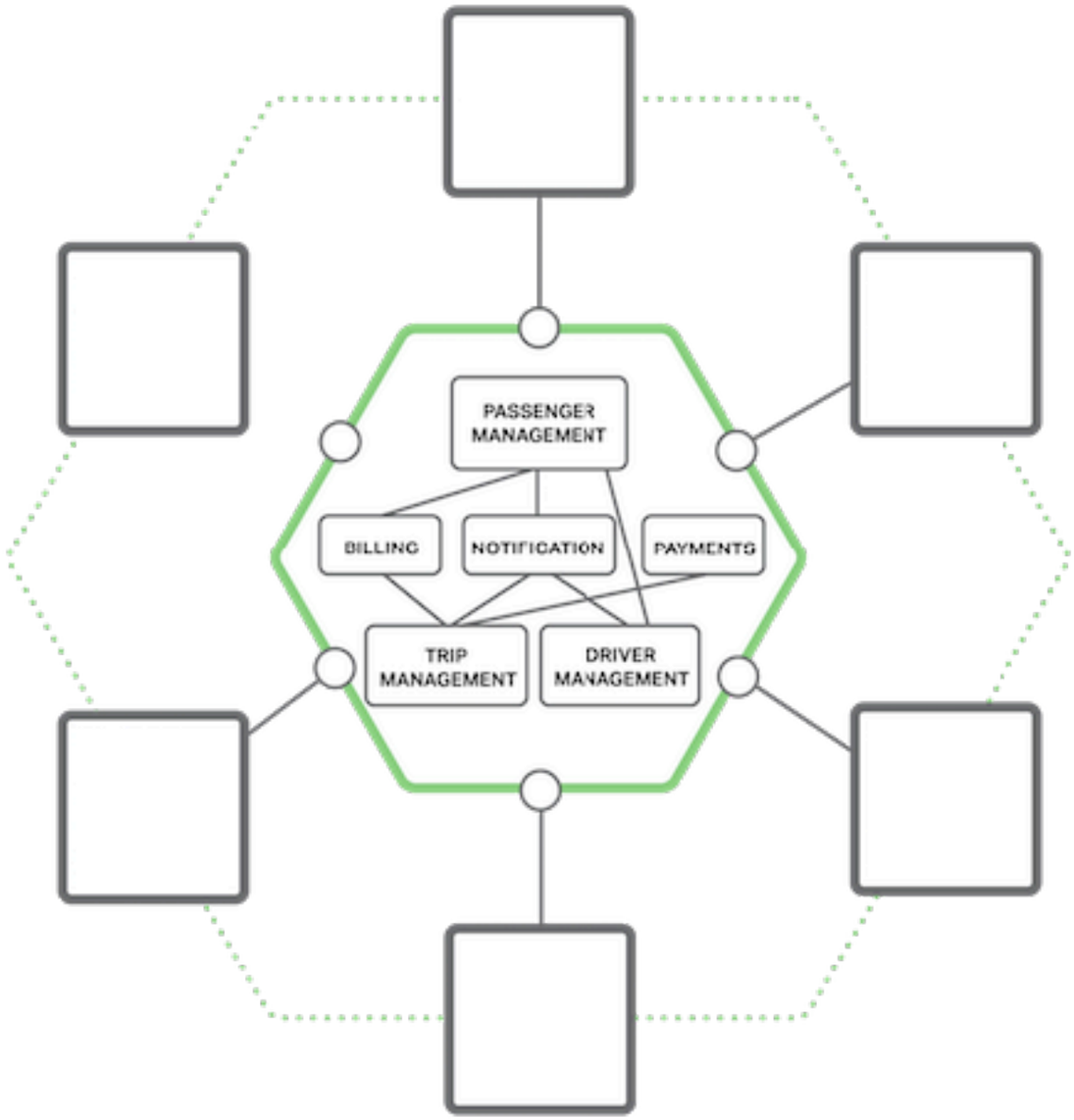
Taxi-Haling REST

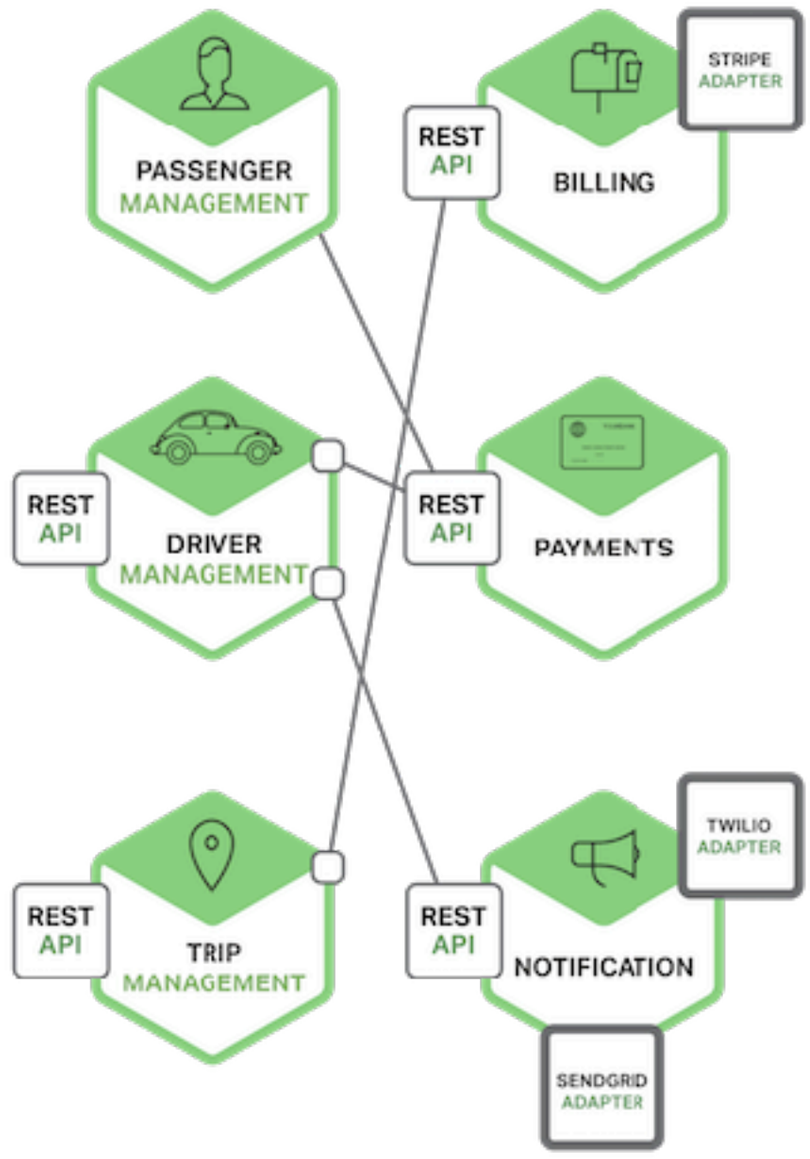


Synchronous Issues

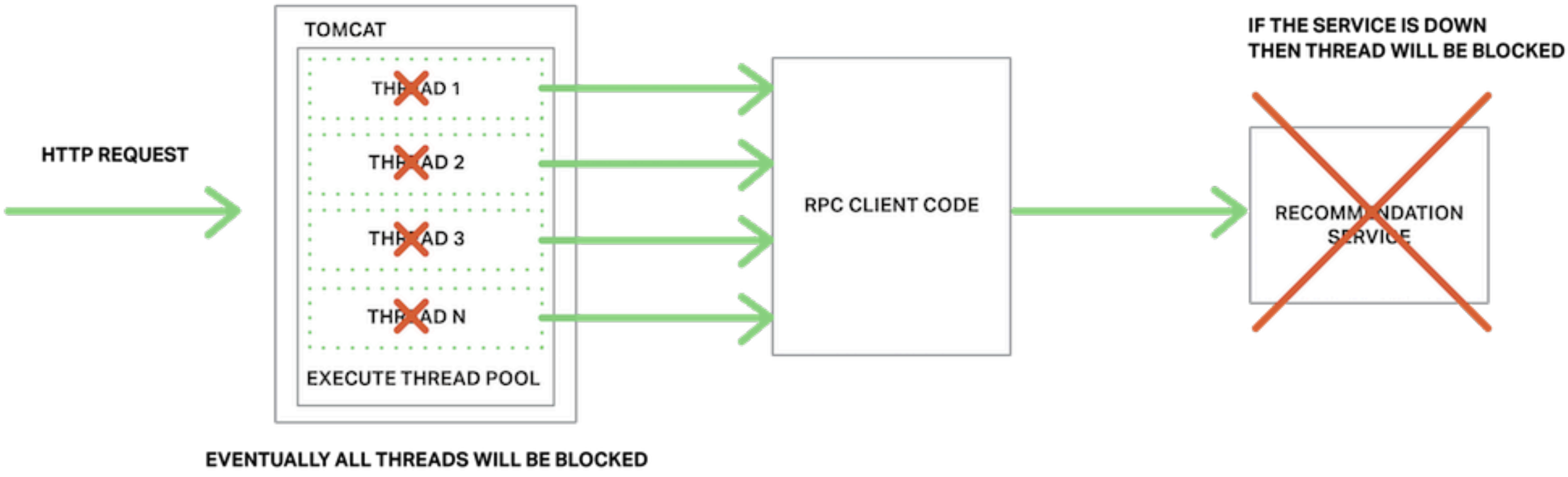
- Both parties have to be available
- Performance subject to network quality
- Clients must know location of service (host\port)

One-To-One

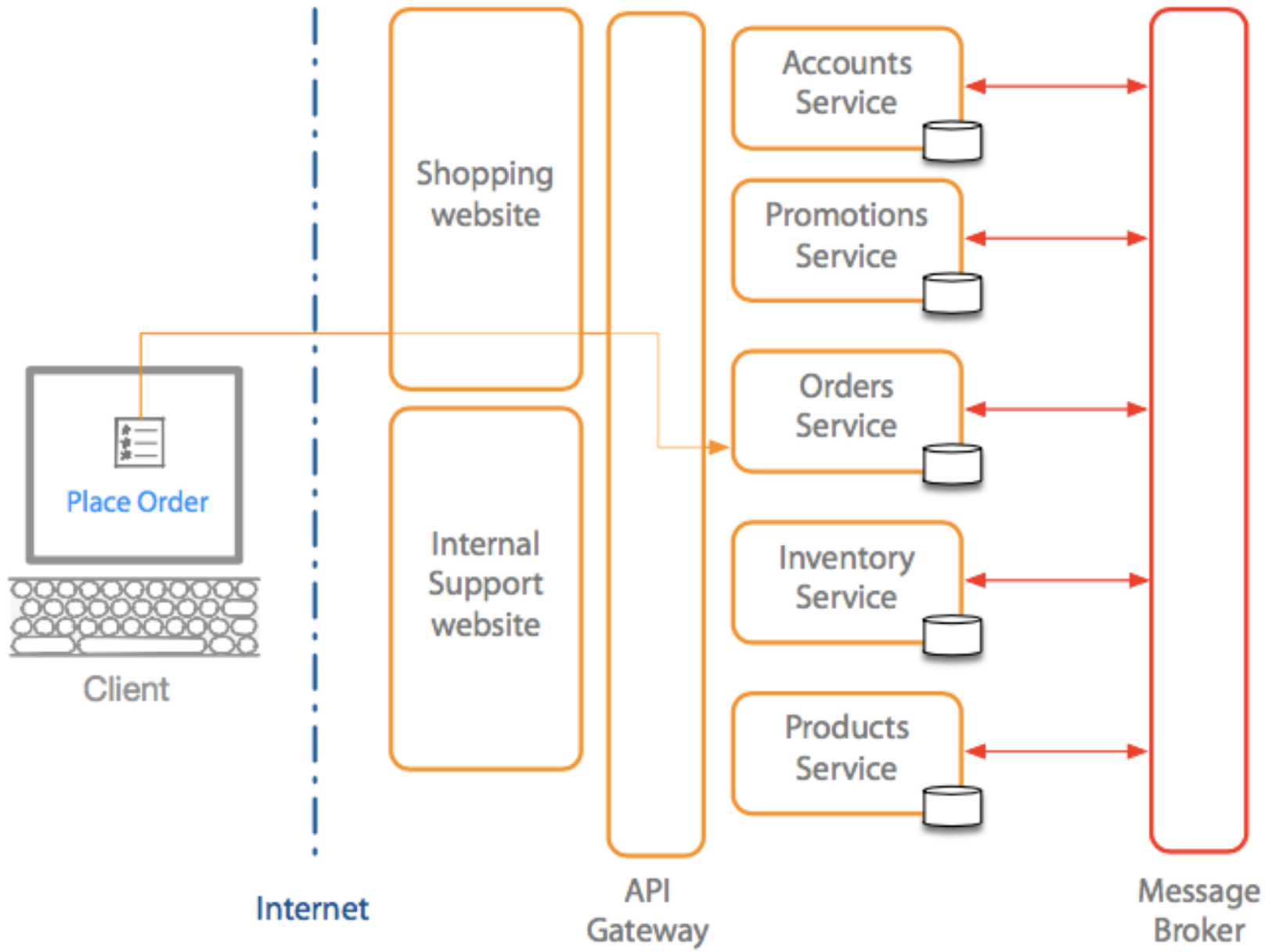

PASSENGER MANAGEMENT
BILLING
NOTIFICATION
PAYMENTS
TRIP MANAGEMENT
DRIVER MANAGEMENT


PASSENGER MANAGEMENT
REST API
BILLING
STRIPE ADAPTER
REST API
DRIVER MANAGEMENT
REST API
PAYMENTS
REST API
TRIP MANAGEMENT
REST API
NOTIFICATION
TWILIO ADAPTER
SENDGRID ADAPTER

Problem


TOMCAT
THREAD 1
THREAD 2
THREAD 3
THREAD N
EXECUTE THREAD POOL
HTTP REQUEST
RPC CLIENT CODE
IF THE SERVICE IS DOWN
THEN THREAD WILL BE BLOCKED
RECOMMENDATION
SERVICE
EVENTUALLY ALL THREADS WILL BE BLOCKED

Asynchronous

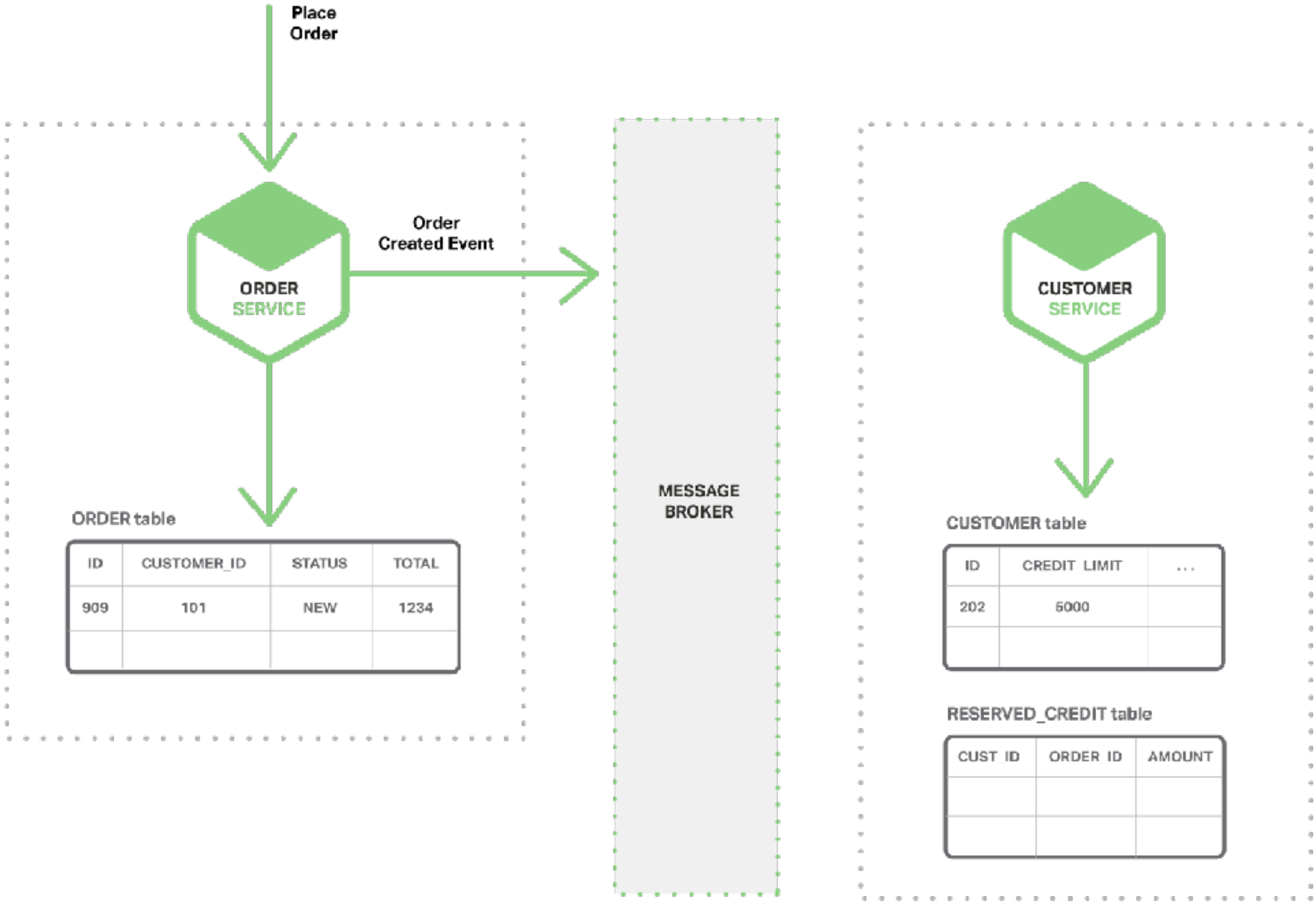


Asynchronous

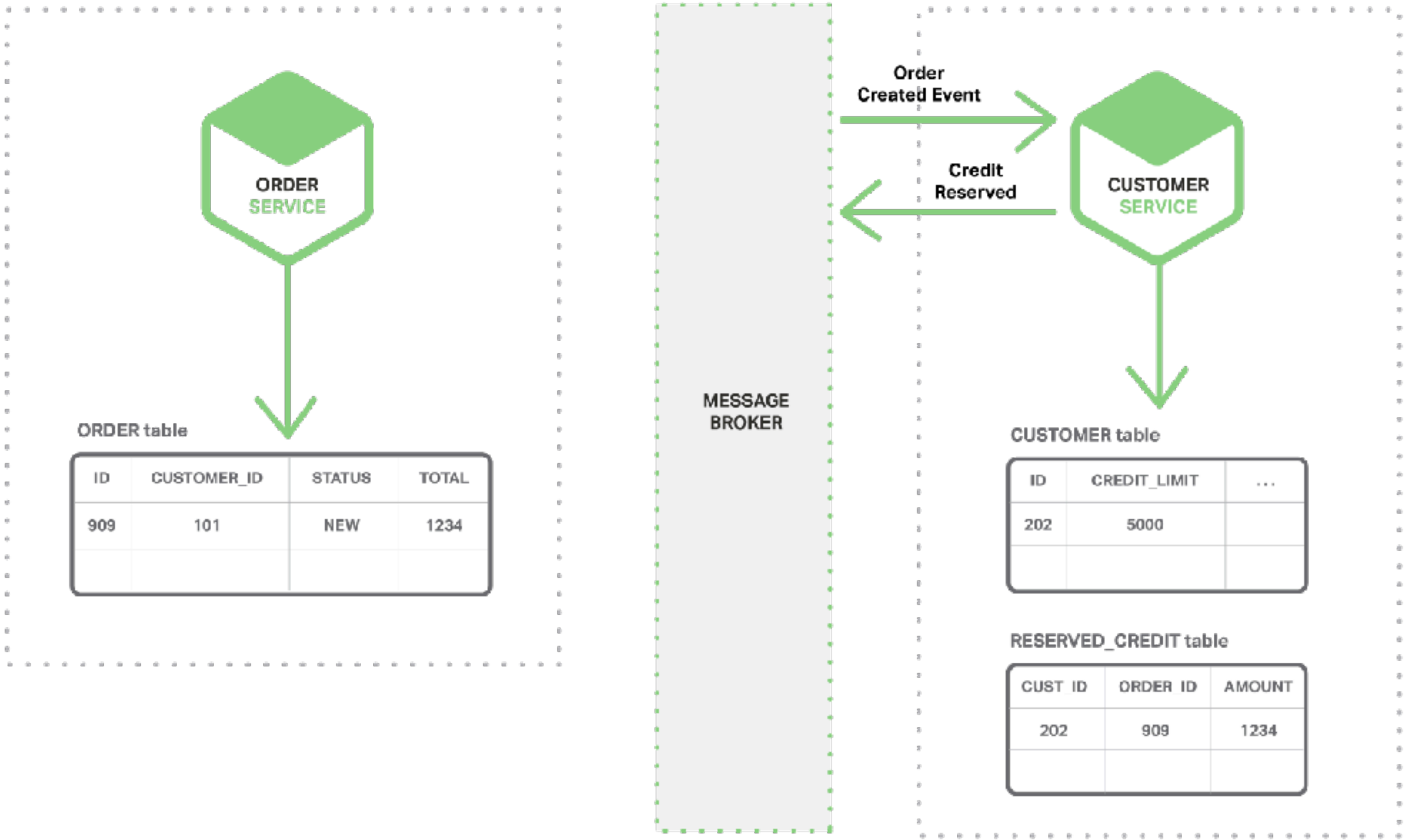
- Event based
  - Mitigates the need of client and service availability
  - Decouples client and service
- Message queueing protocol
  - Message Brokers
  - Subscriber and publisher are decoupled
  - Microsoft message queuing (MSMQ)
  - RabbitMQ
  - ATOM (HTTP to propagate events)

Asynchronous

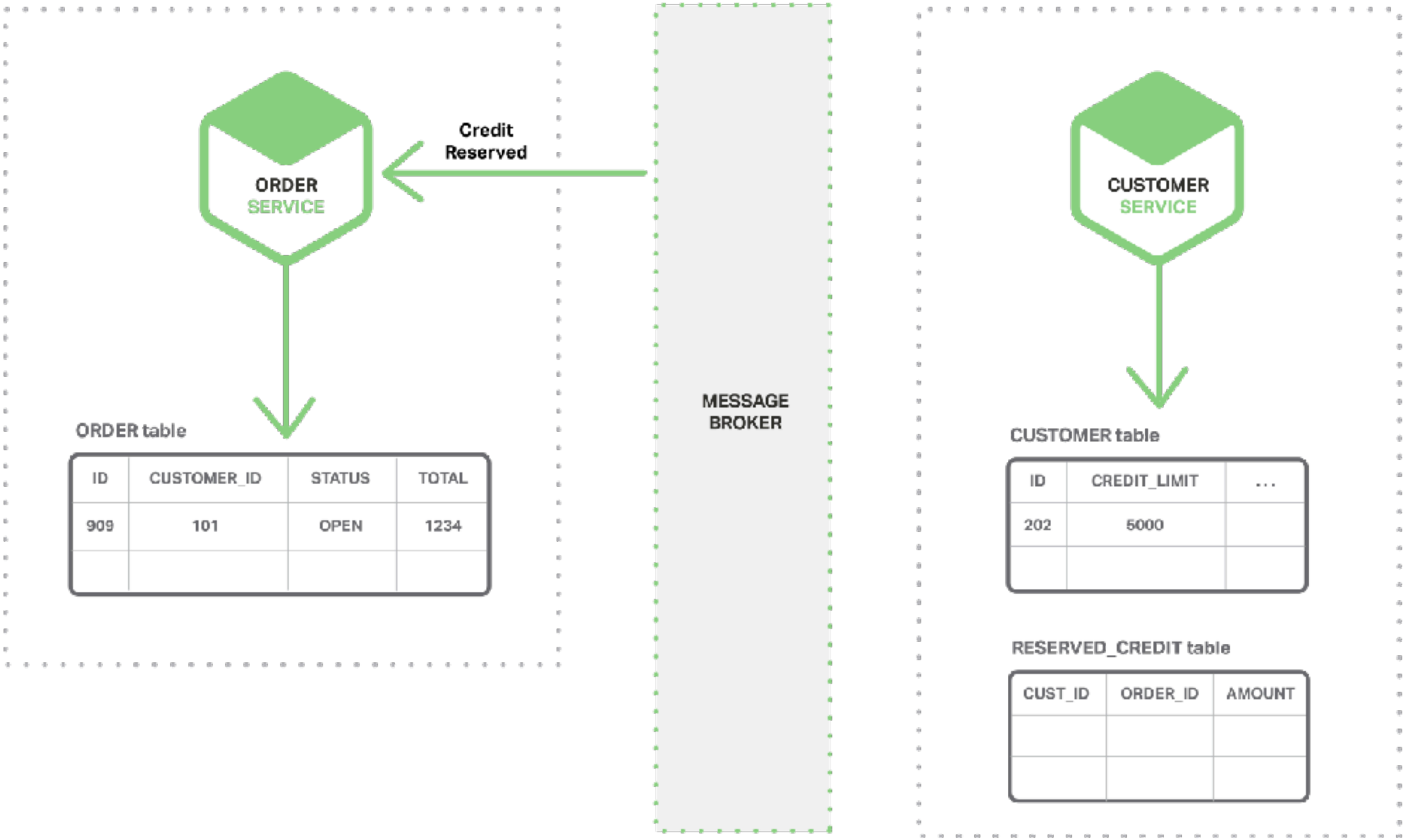
- Asynchronous challenge
  - Complicated
  - Reliance on message broker
  - Visibility of the transaction
  - Managing the messaging queue
- Real world systems
  - Would use both synchronous and asynchronous



Event-Driven Architecture

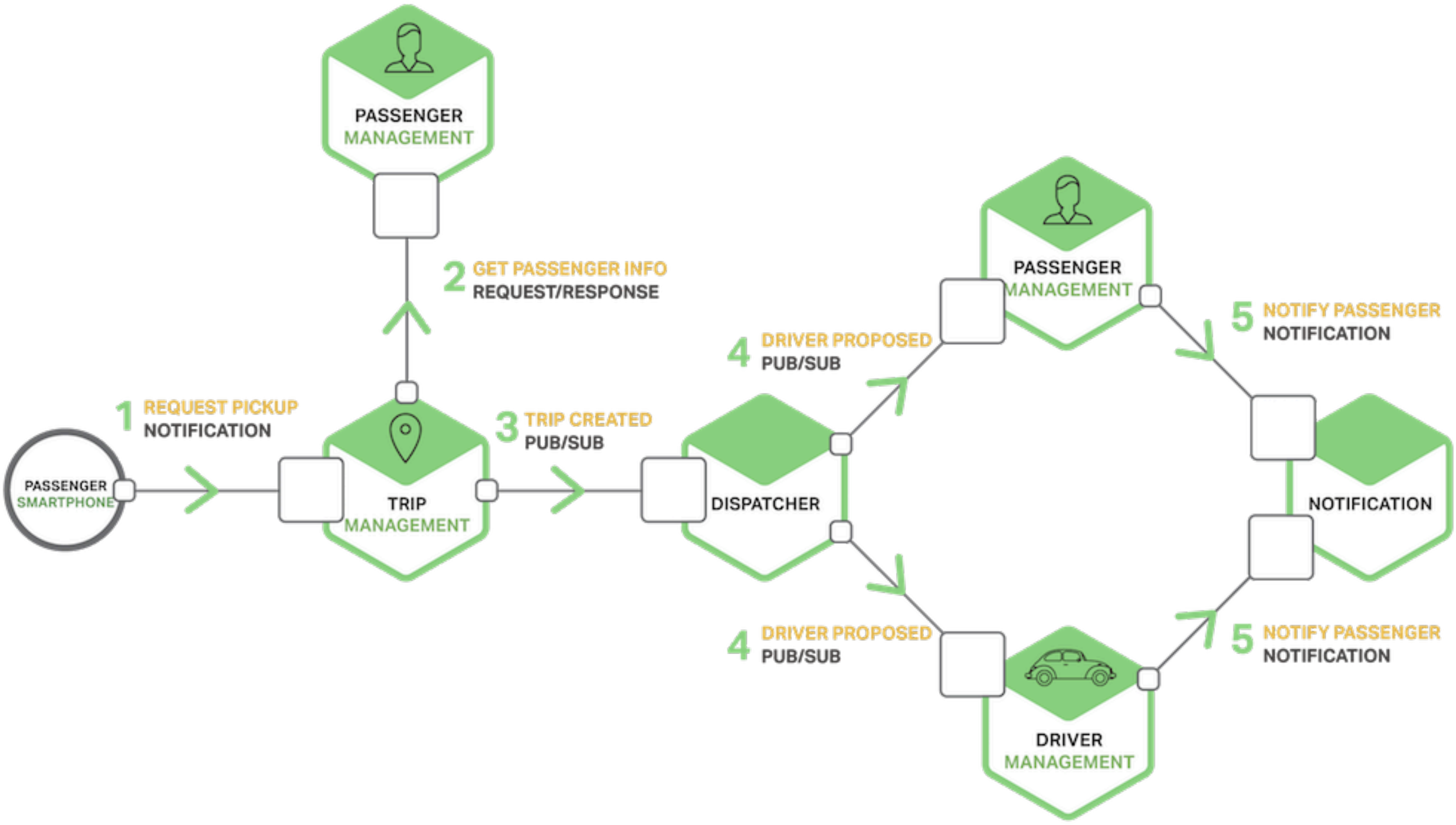


Event-Driven Architecture



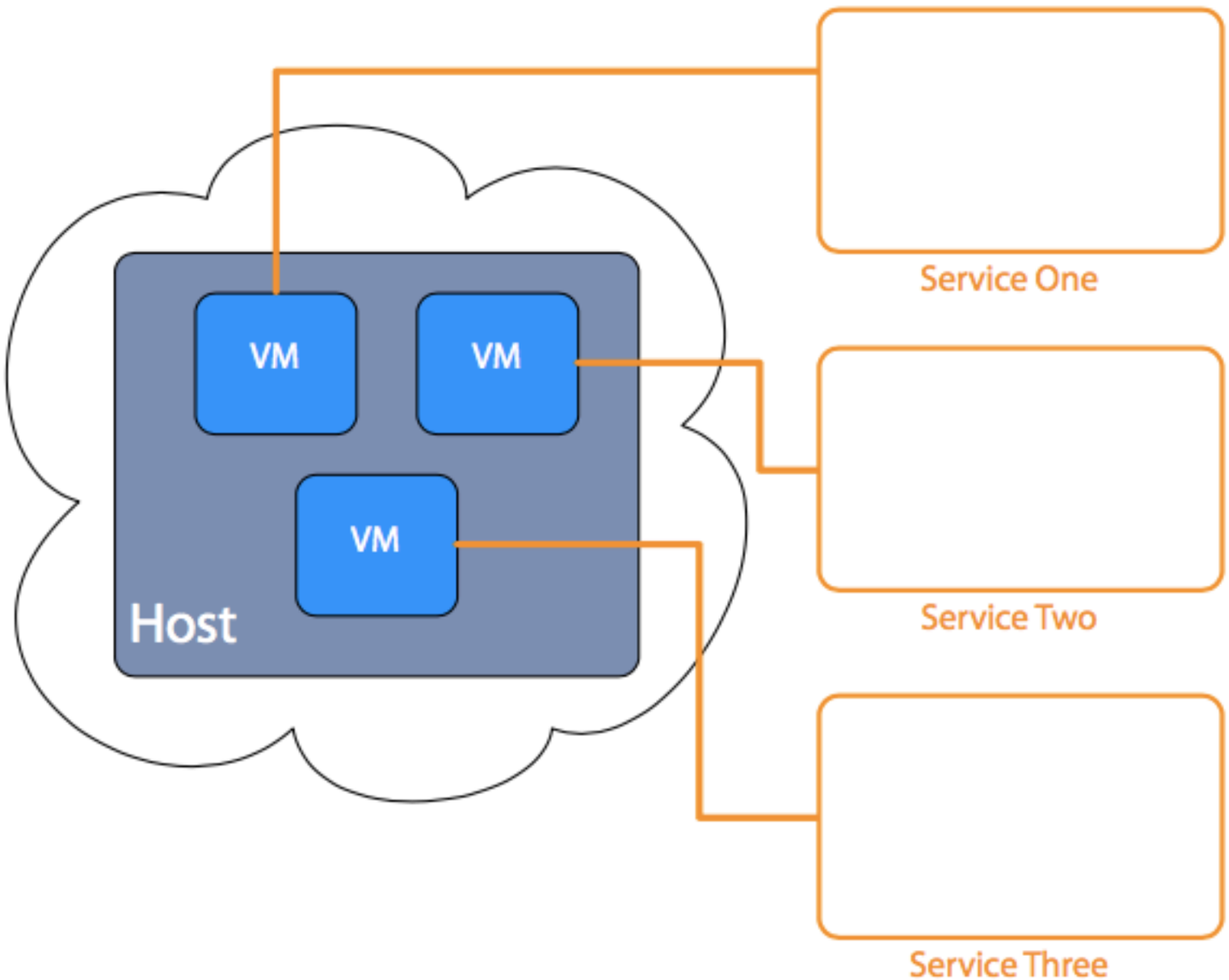
Event-Driven Architecture

One-To Many


PASSENGER
MANAGEMENT
2 GET PASSENGER INFO
REQUEST/RESPONSE
1 REQUEST PICKUP
NOTIFICATION
PASSENGER
SMARTPHONE
TRIP
MANAGEMENT
3 TRIP CREATED
PUB/SUB
DISPATCHER
4 DRIVER PROPOSED
PUB/SUB
PASSENGER
MANAGEMENT
5 NOTIFY PASSENGER
NOTIFICATION
NOTIFICATION
4 DRIVER PROPOSED
PUB/SUB
DRIVER
MANAGEMENT
5 NOTIFY PASSENGER
NOTIFICATION

Hosting Platforms

Virtualization



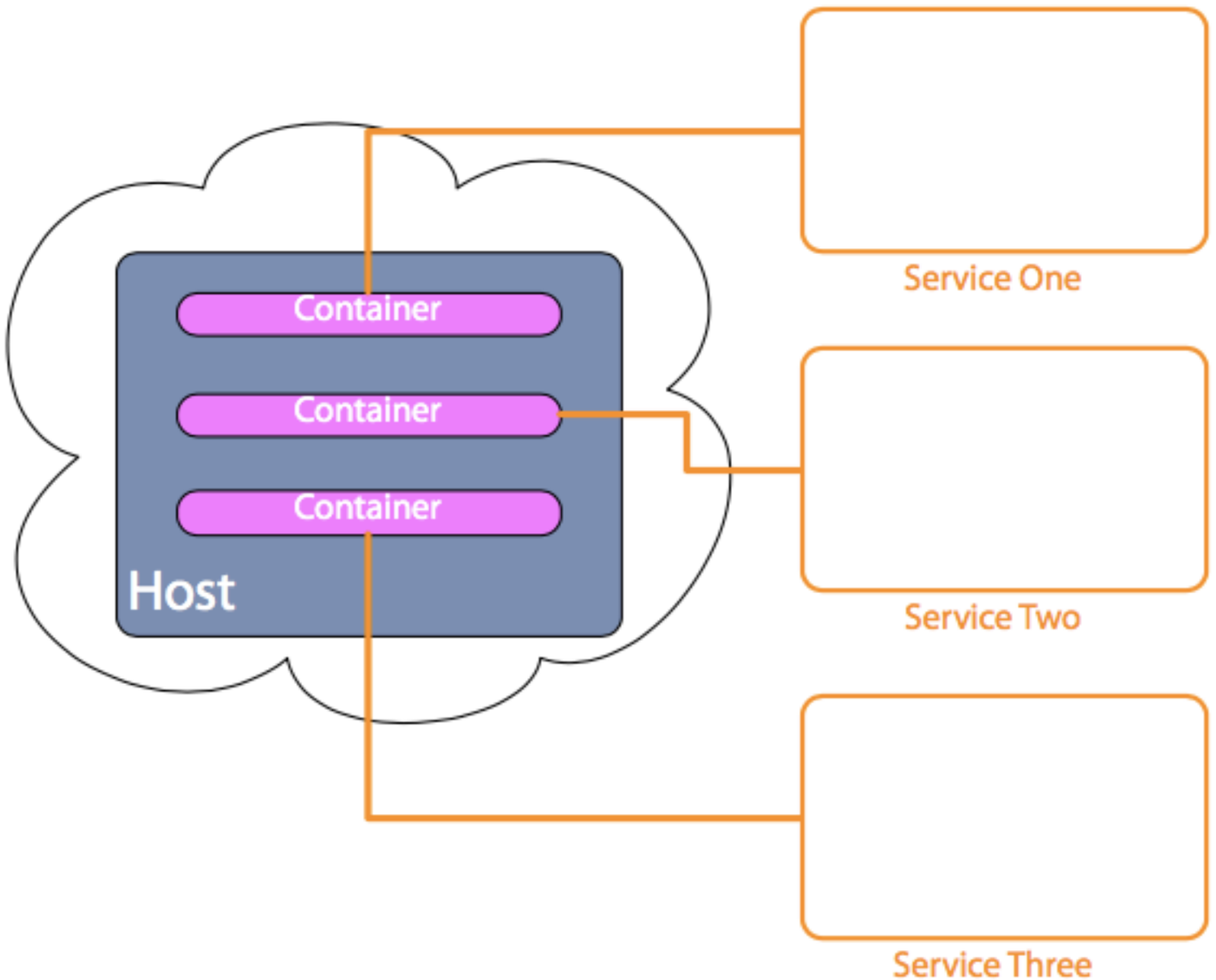
Virtualization

- A virtual machine as a host
- Foundation of cloud platforms
  - Platform as a service (PAAS)
  - Microsoft Azure
  - Amazon web services
  - Your own cloud (for example vSphere)

Virtualization

- Could be more efficient
  - Takes time to setup
  - Takes time to load
  - Take quite a bit of resource
- Unique features
  - Take snapshot
  - Clone instances

Container


Container
Container
Container
Host
Service One
Service Two
Service Three

Container

- Type of virtualization
- Isolate services from each other
- Single service per container
- Different to a virtual machine
  - Use less resource than VM
  - Faster than VM
  - Quicker to create new instances

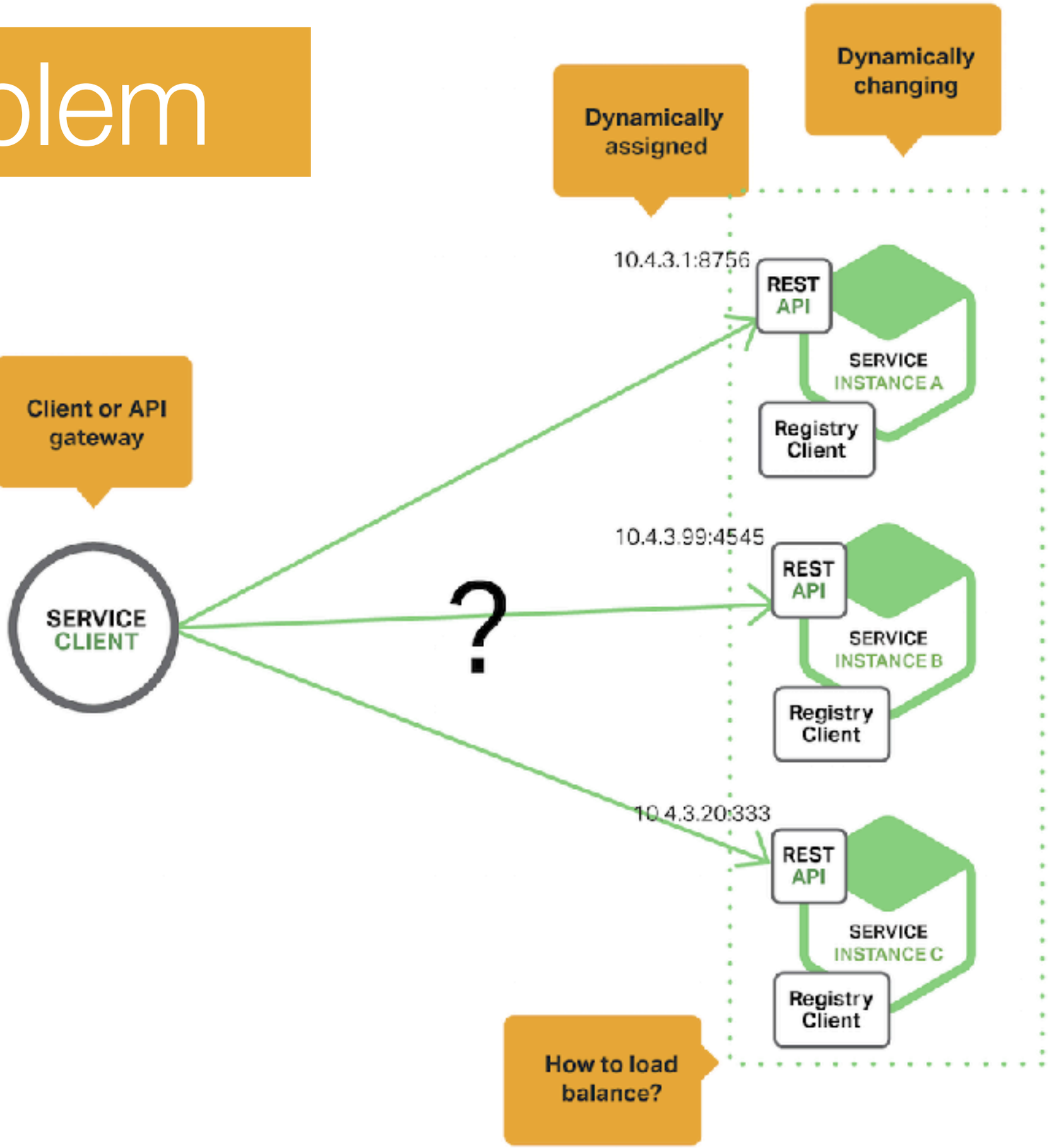
Container

- Future of hosted apps
- Cloud platform support growing
- Mainly Linux based
- Not as established as virtual machines
  - Not standardised
  - Limited features and tooling
  - Infrastructure support in its infancy
  - Complex to setup

Self-Hosting

- Implement your own cloud
  - Virtualization platform
  - Implement containers
- Use of physical machines
  - Single service on a server
  - Multiple services on a server
- Challenges
  - Long-term maintenance
  - Need for technicians
  - Training
  - Need for space
  - Scaling is not as immediate

Problem

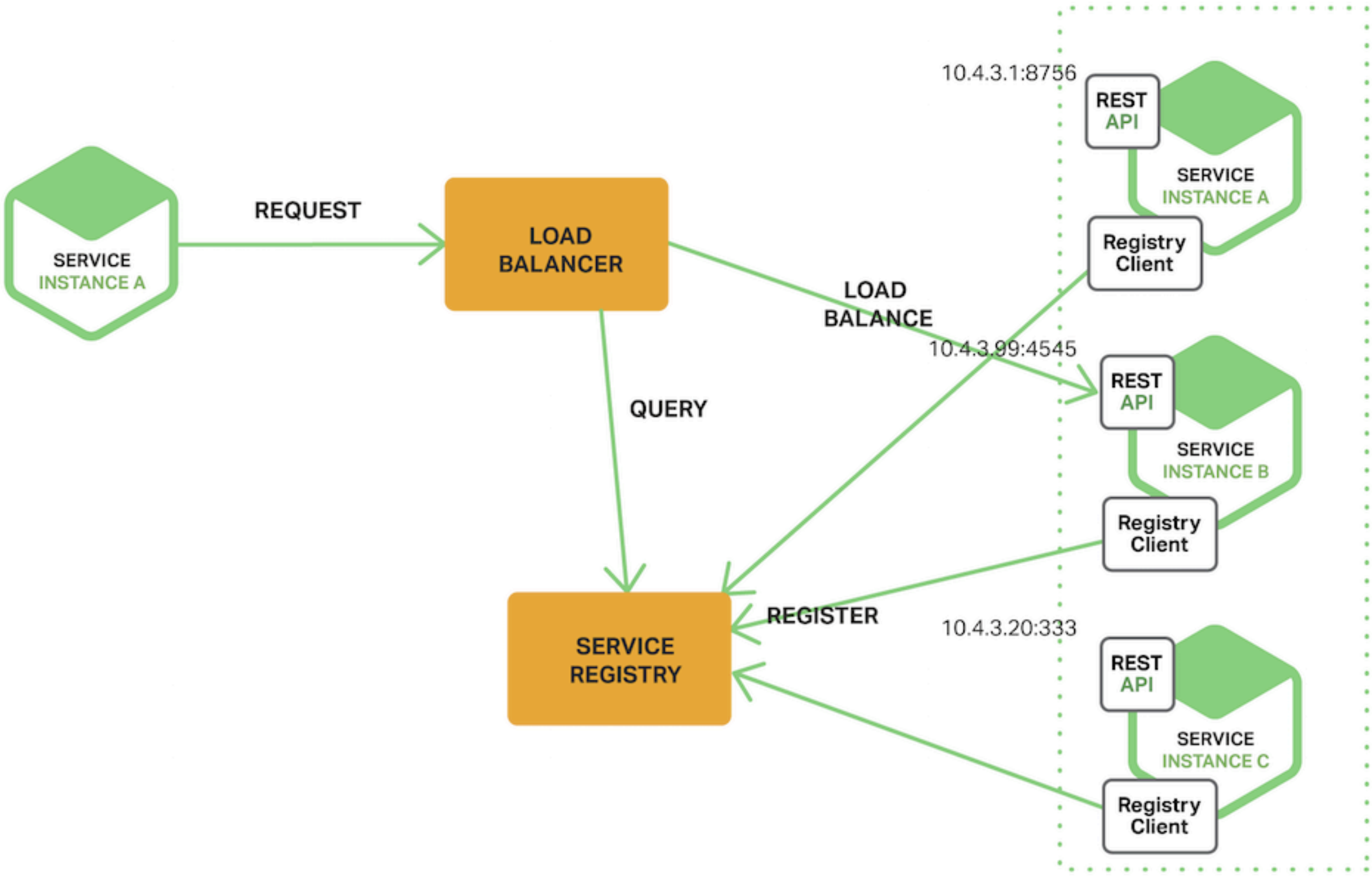

Dynamically changing
Dynamically assigned
Client or API gateway
10.4.3.1:8756
REST API
SERVICE INSTANCE A
Registry Client
10.4.3.99:4545
REST API
SERVICE INSTANCE B
Registry Client
SERVICE CLIENT
?
10.4.3.20:333
REST API
SERVICE INSTANCE C
Registry Client
How to load balance?

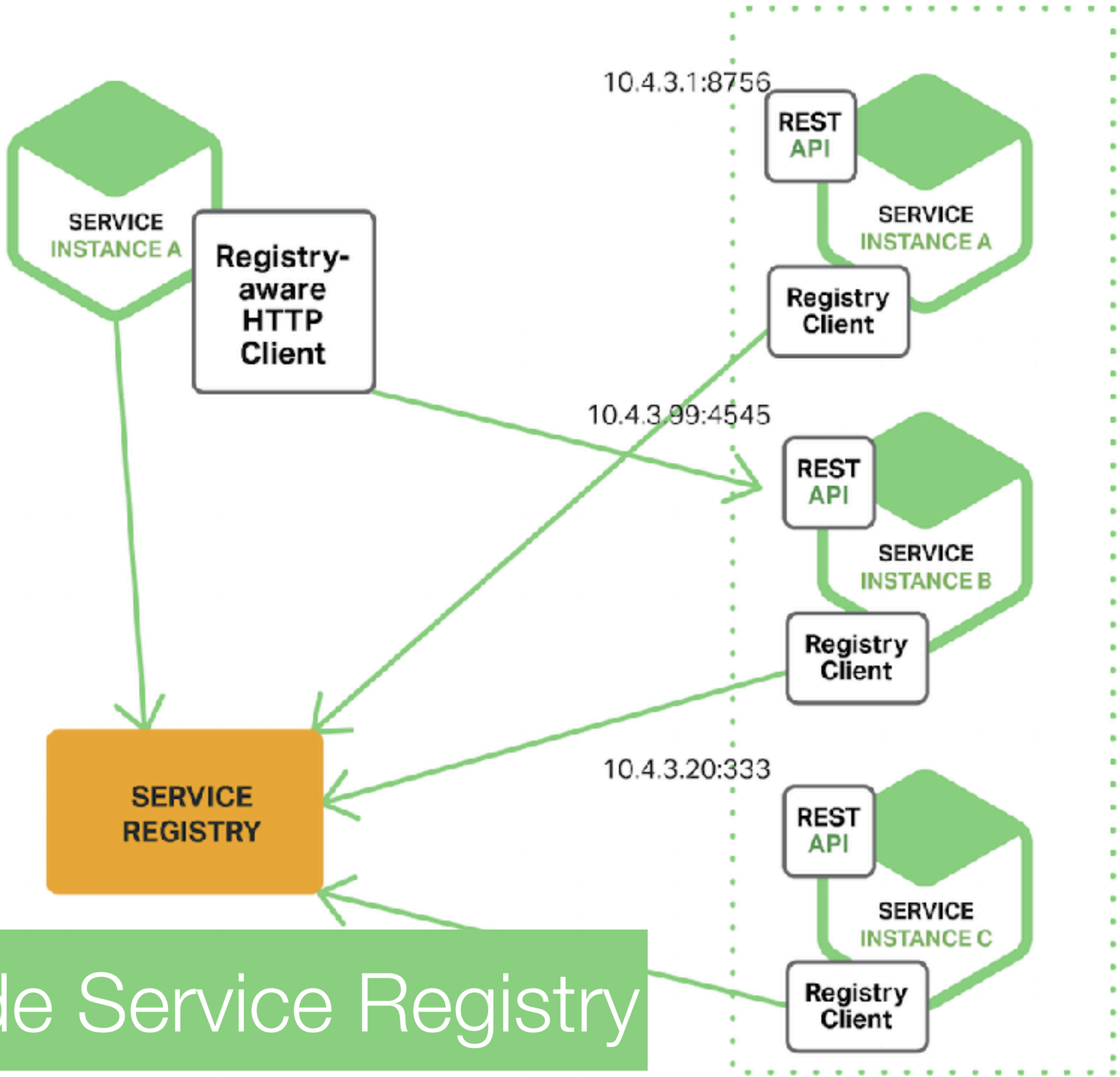
Registration and Discovery

- Service registry database
- Register on startup
- Deregister service on failure
- Cloud platforms make it easy
- Local platform registration options
  - Self registration
  - Third-party registration
- Local platform discovery options
  - Client-side discovery
  - Server-side discovery

Host, Port, Version

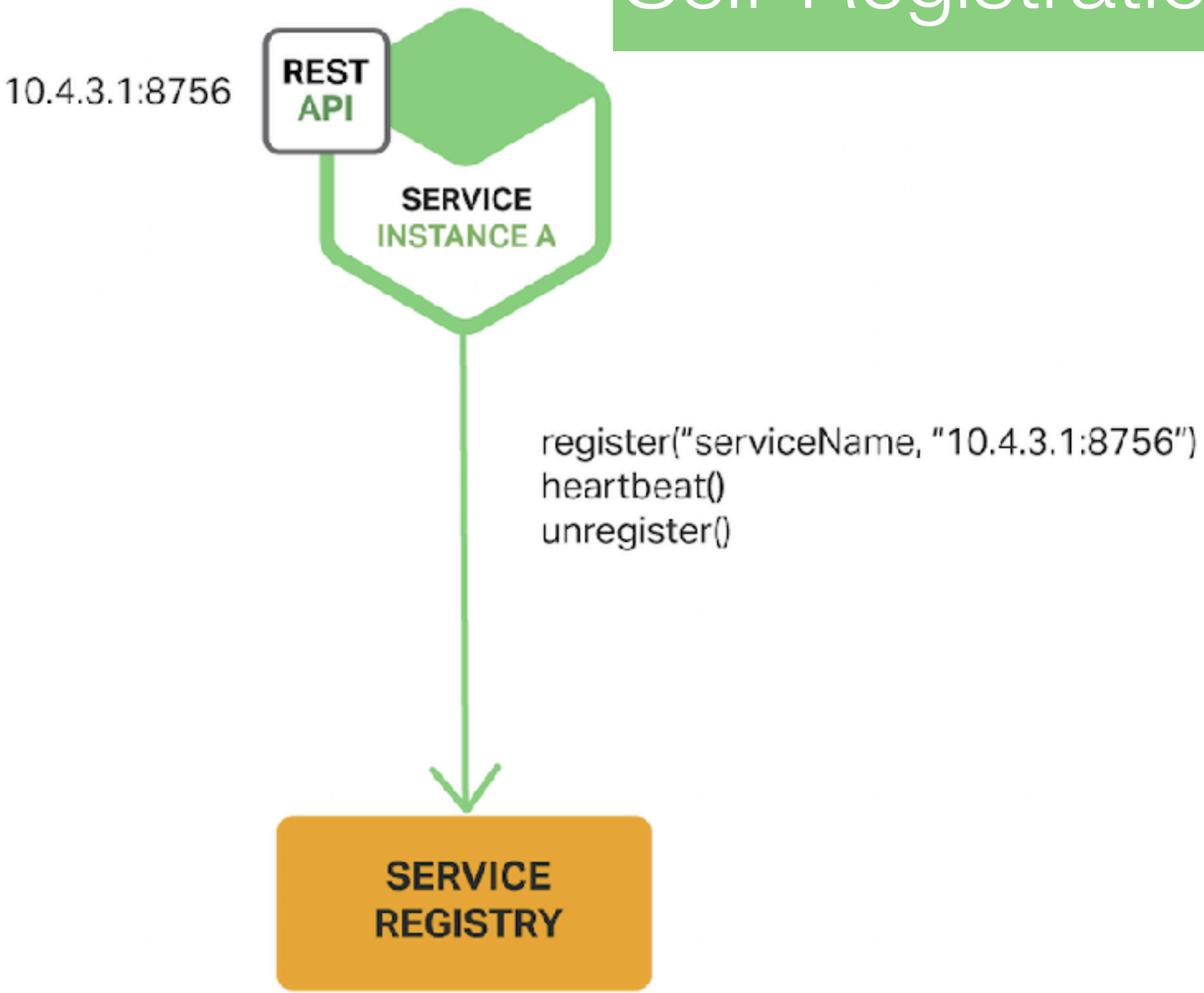
Server-Side Service Registry



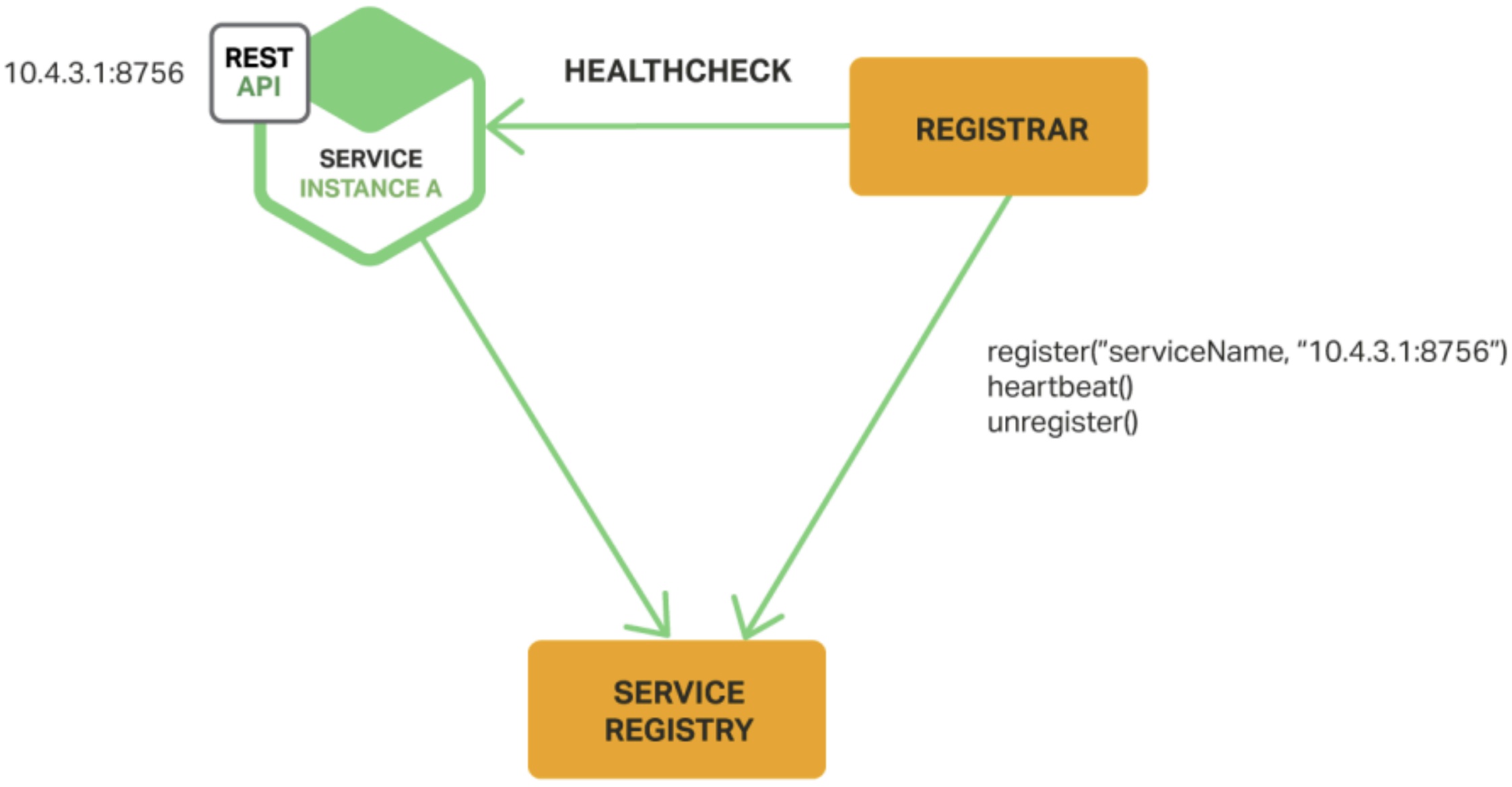


Client-Side Service Registry

Self-Registration Pattern



Third-Party Registration Pattern



Observable

Observable Microservices

Logging Tech

Monitoring Tech

Monitoring Tools

- cAdvisor
  This container-monitoring tool, developed by Google, natively supports Docker containers. The cAdvisor daemon, which collects, aggregates, processes, and exports information about containers, is available in a graphical version

- Heapster
  Use Heapster to monitor Kubernetes container clusters. It collects and interprets various signals such as compute resource usage and lifecycle events, supporting multiple sources of data

- Prometheus
  Prometheus is an open-source service-monitoring system and time-series database

- Grafana
  A metrics dashboard, Grafana integrates with many data sources, including Graphite, CloudWatch, Prometheus, Elsaticsearch, and InfluxDB

Features

- Metrics across servers
- Automatic or minimal configuration
- Client libraries to send metrics
- Test transactions support
- Alerting

Microservice Monitoring

- Why the need to monitor
  - Microservices complexity
  - High availability is a requirement

- Causes of microservices outages
  - Deployment issues
  - Delayed and long repair
  - Downstream dependency issues

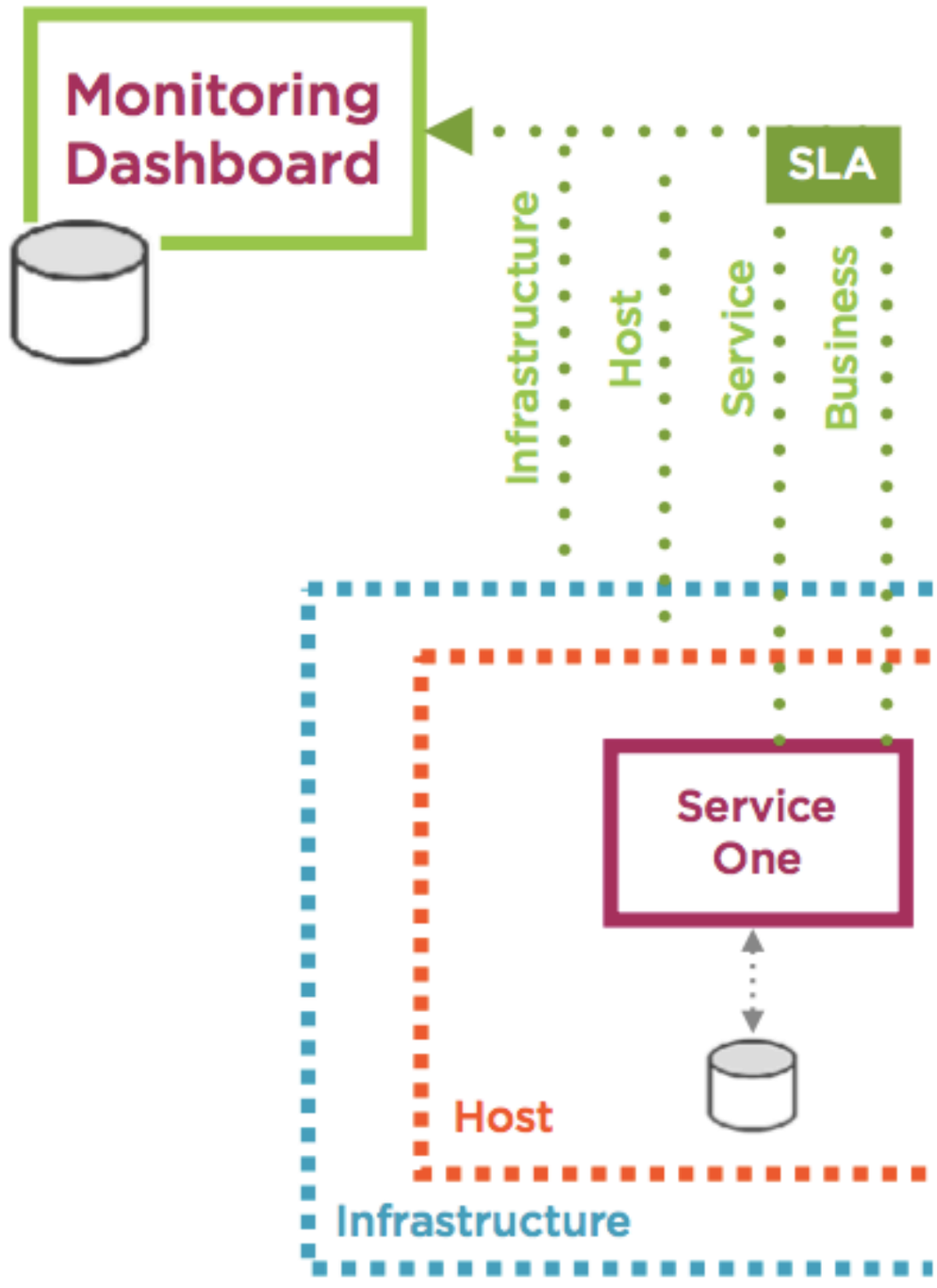
- Monitoring is the proactive solution
  - Know the state of all components
  - Know the state at every stage
  - Collect all statistics using metrics

Key Metrics

- Host and infrastructure metrics
  - State of hosts and infrastructure
  - State affects microservices
- Monitor microservice metrics
  - Availability
  - Response rate
  - Success or failures of end Business metrics
  - Hidden issues and failures
  - Sudden changes in behavior points
  - Errors, exceptions and timeouts
  - Health of dependencies
  - Metrics at different stages
- Business metrics
  - Hidden issues and failures
  - Sudden changes in behavior

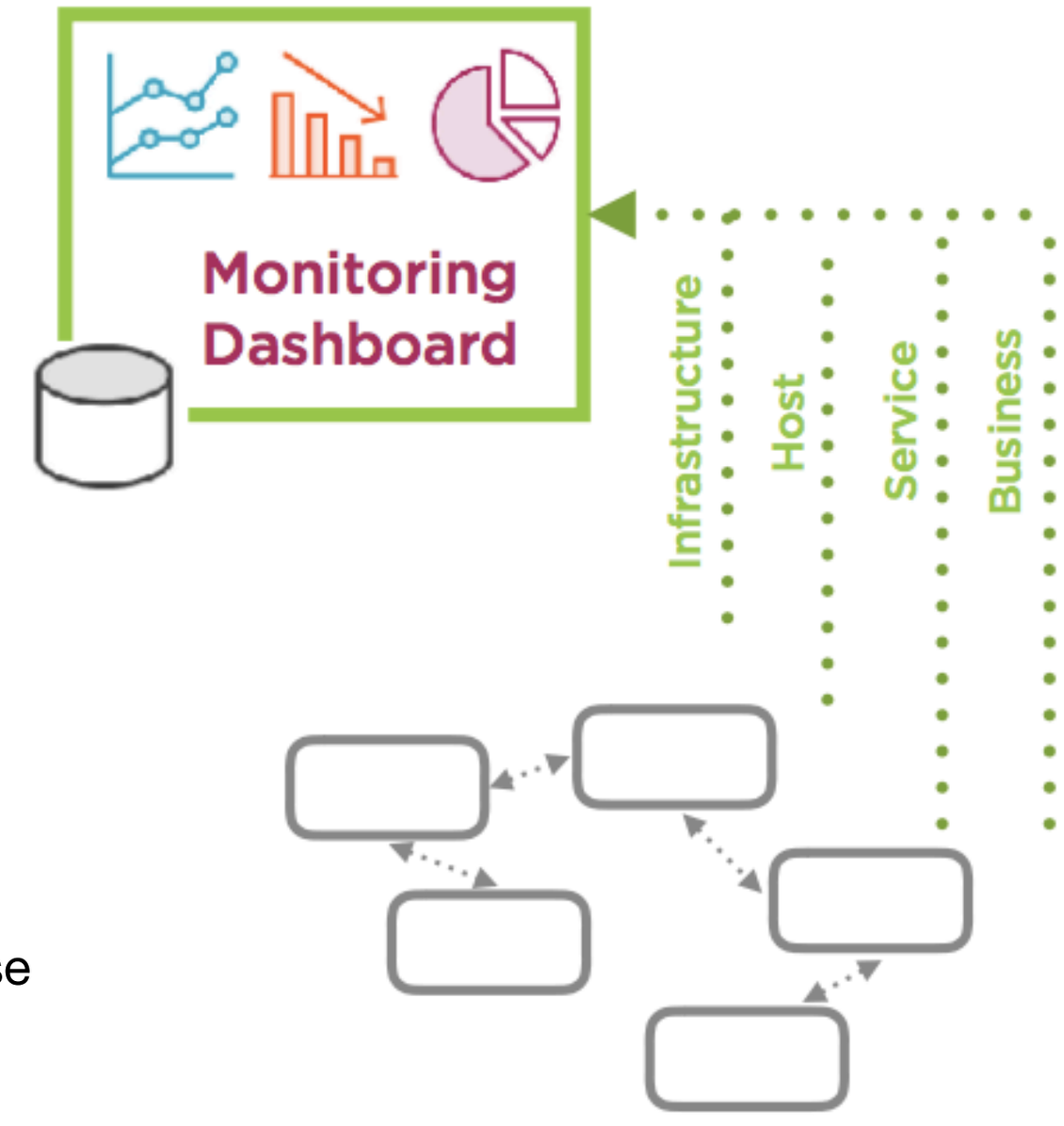
Monitor SLA Metrics

- Monitor SLAs at all levels
  - Service level
  - Service endpoint level
- Examples of SLA metrics
  - Throughput and uptime
  - Concurrent clients
  - Average response times
- Impact of dependencies on SLAs
  - Monitor dependencies
- Use SLA monitoring to alert
  - Actionable alerts to fix SLA issues
- Communicate SLA issues early



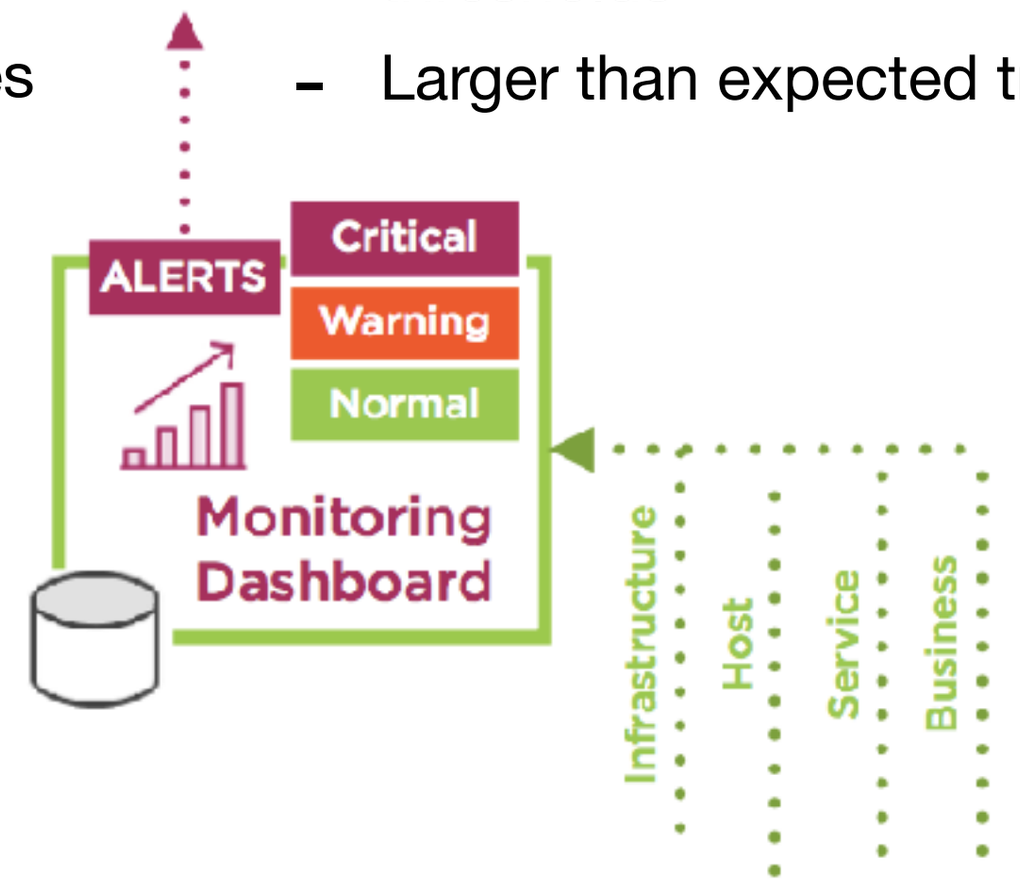
Monitoring Dashboard

- Display and collect metrics
- Real-time
- Centralized, accessible and standardized
- Simple, visual and minimal information
  Graphs showing metrics over time
- Highlights effectiveness of monitoring
- What to display
  - Metrics and their alerts
  - Metrics for each environment and phase
  - Correlate metrics to events
- Use to help define thresholds for alerts

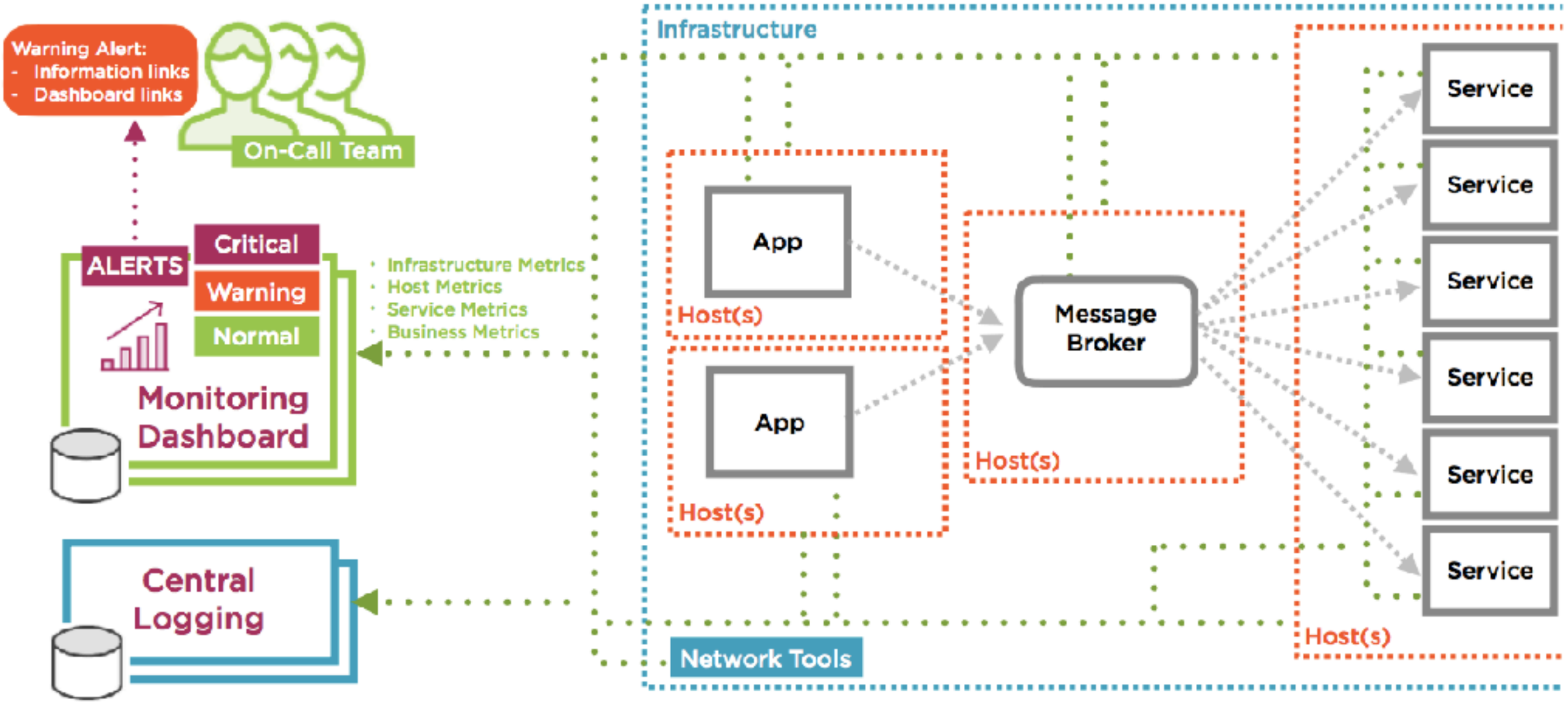


Defining Thresholds for Alerts

- Ranges required for effective alerting
- Types of thresholds
  - Normal state
  - Warning state
  - Critical state
- Warning alerts prevent outages
- Effective levels for thresholds
- Defining thresholds
  - Use historical data
  - Performance testing and load testing data
  - Expected traffic for normal thresholds
  - Larger than expected traffic



MicroServices Monitoring Patterns



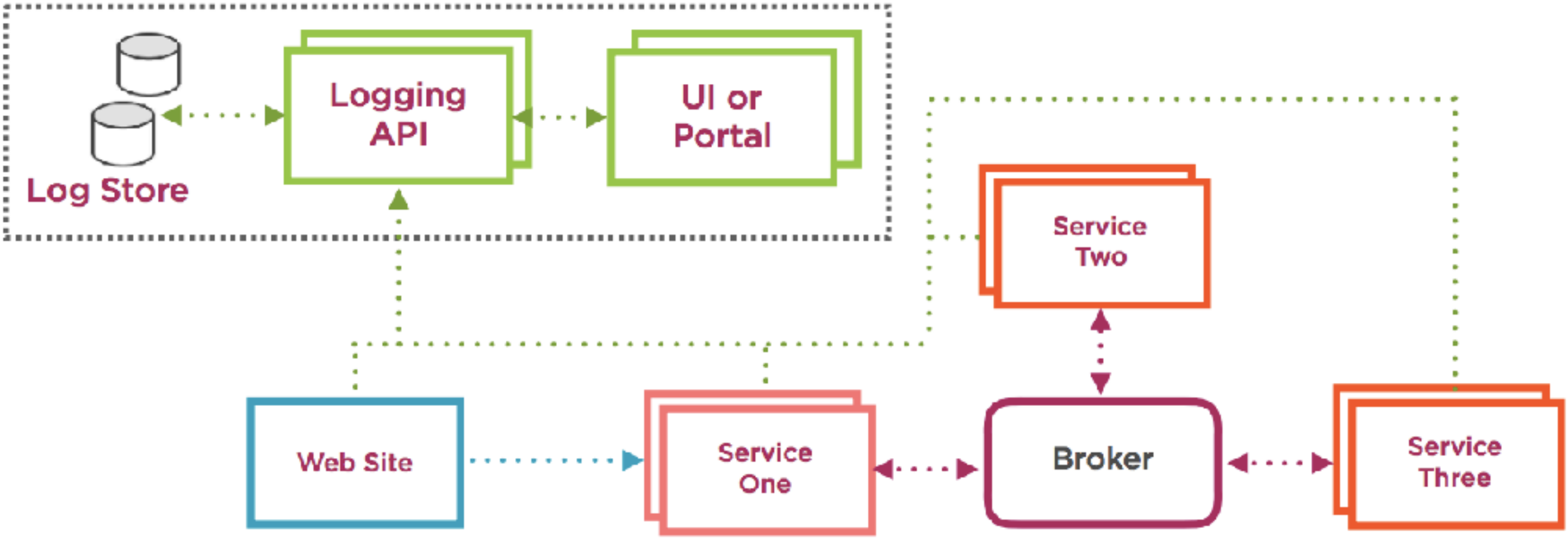
Logging Tools

- Logstash
- Kibana
- Splunk
- Graphite

Features

- Structured logging
- Logging across servers
- Automatic or minimal configuration
- Correlation\Context ID for transactions

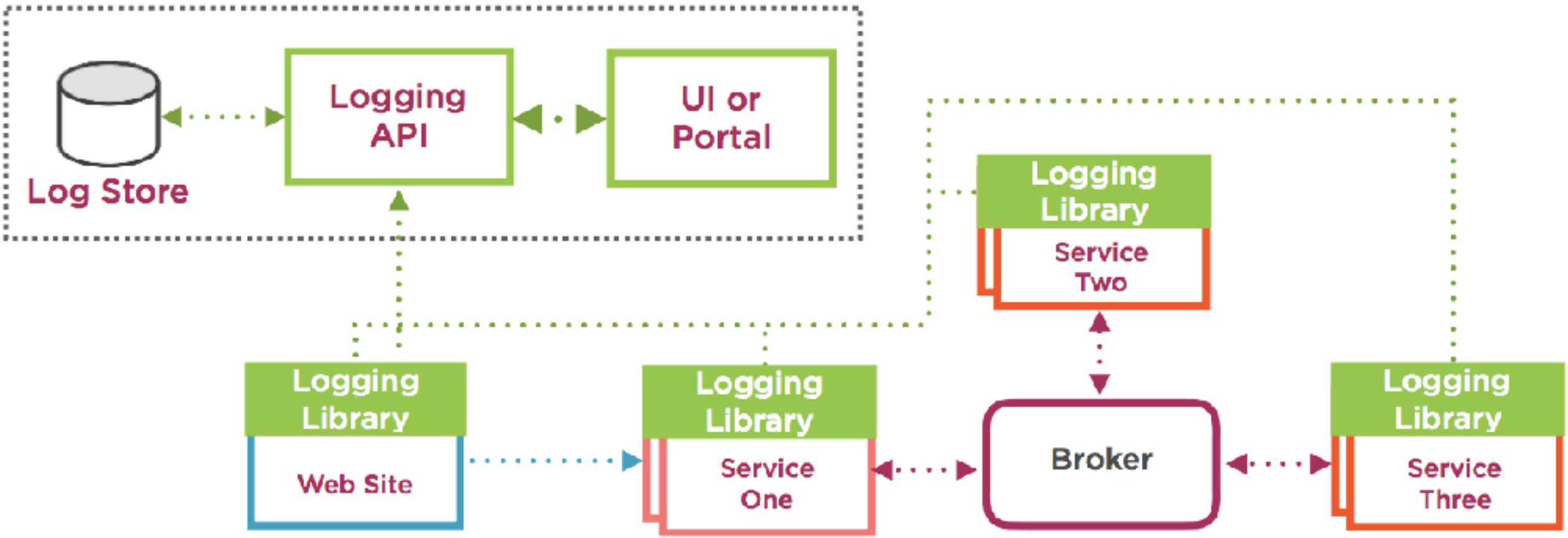
Centralize Logging



Consistence Logging Format

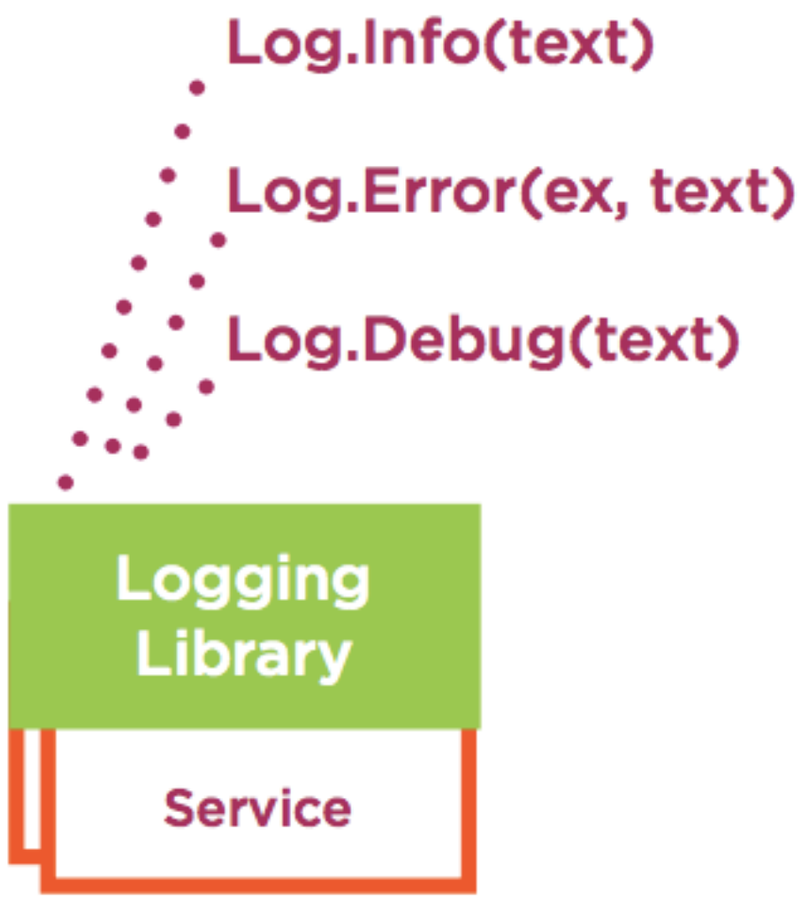
- Across the architecture
- Structured format
  - Levels
  - Date and time
  - Correlation ID
  - Host and app information - Message
- Shared library
- Decoupled from implementation

Consistence Logging Format



Logging Level

- Information level
  - Non-technical descriptive message
  - Simple enough for entire business
  - Key transaction milestones
- Debug level
  - Technical milestones
  - Calls and parameters
  - Response times and timeout stats
- Error level
  - Information on exceptions raised
  - Error information and number
  - Call stack



Performance

Scaling

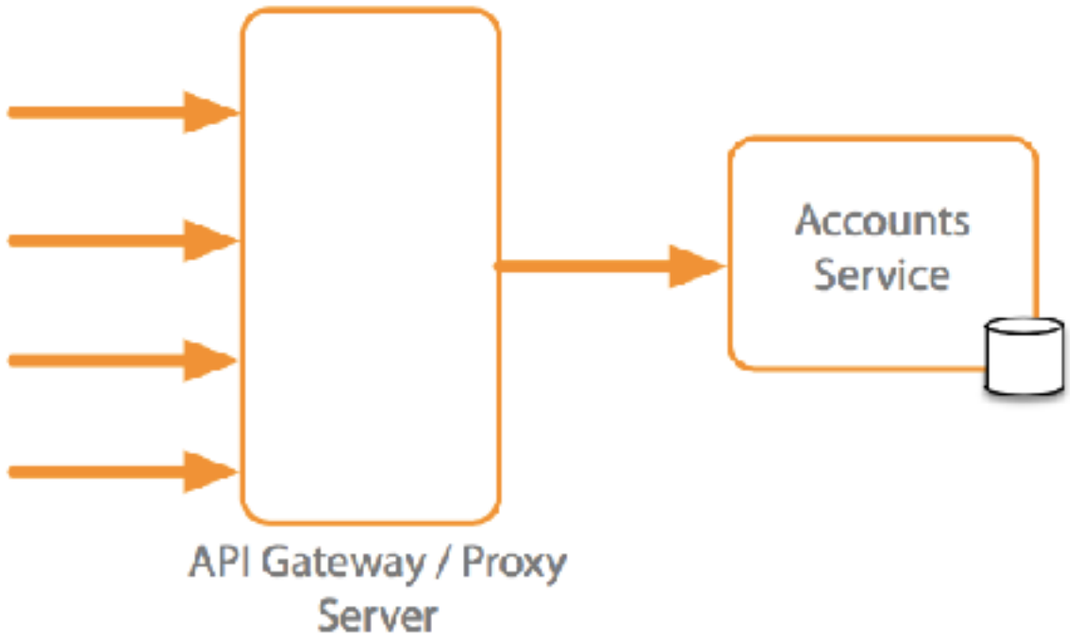
- How
  - Creating multiple instances of service
  - Adding resource to existing service
- Automated or on-demand
- PAAS auto scaling options
- Virtualization and containers

Scaling

- Physical host servers
- Load balancers (API Gateway)
- When to scale up
  - Performance issues
  - Monitoring data
  - Capacity planning

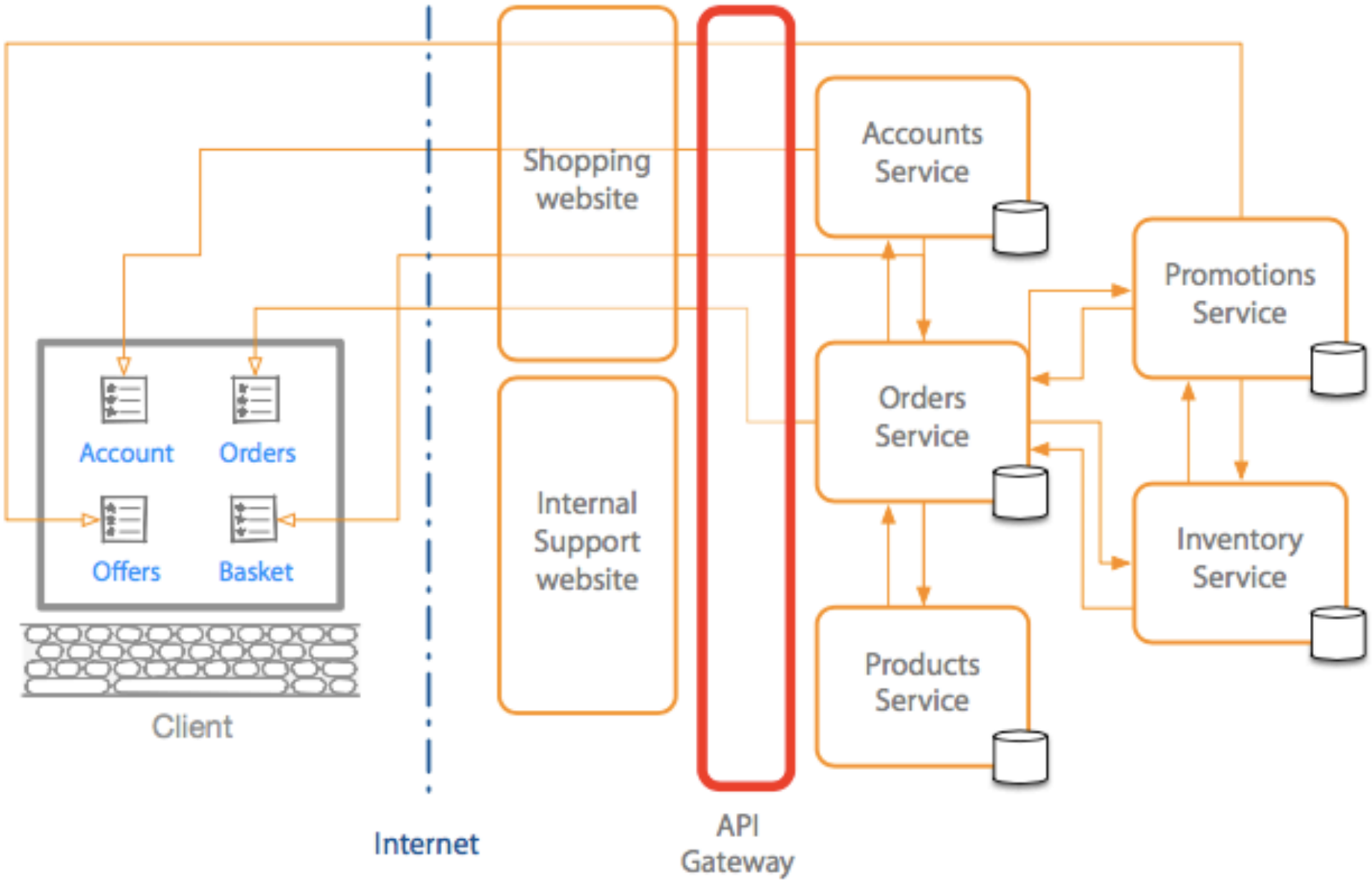
Caching

- Caching to reduce
  - Client calls to services
  - Service calls to databases
  - Service to service calls
- API Gateway\Proxy level
- Client side
- Service level
- Considerations
  - Simple to setup and manage
  - Data leaks

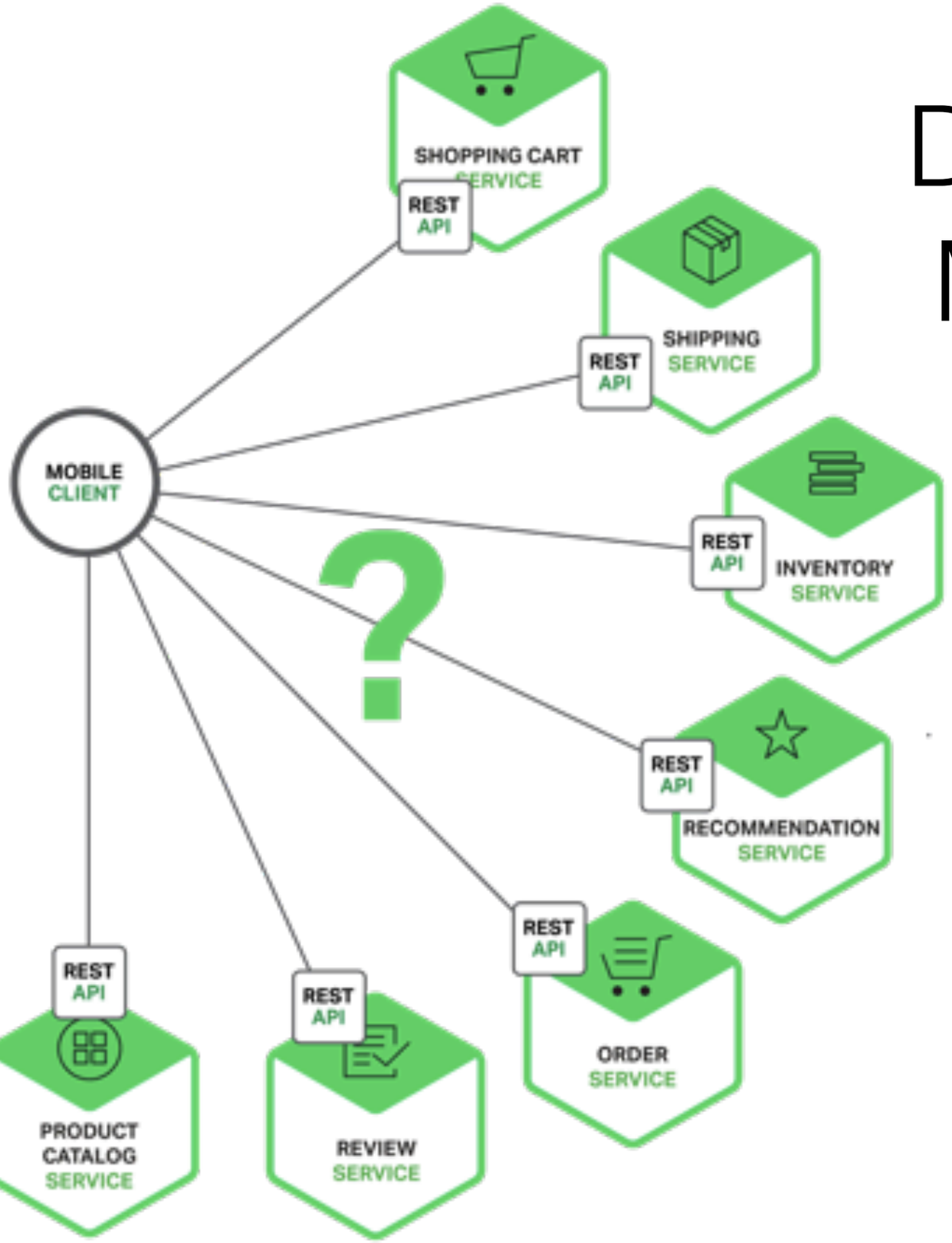


API Gateway

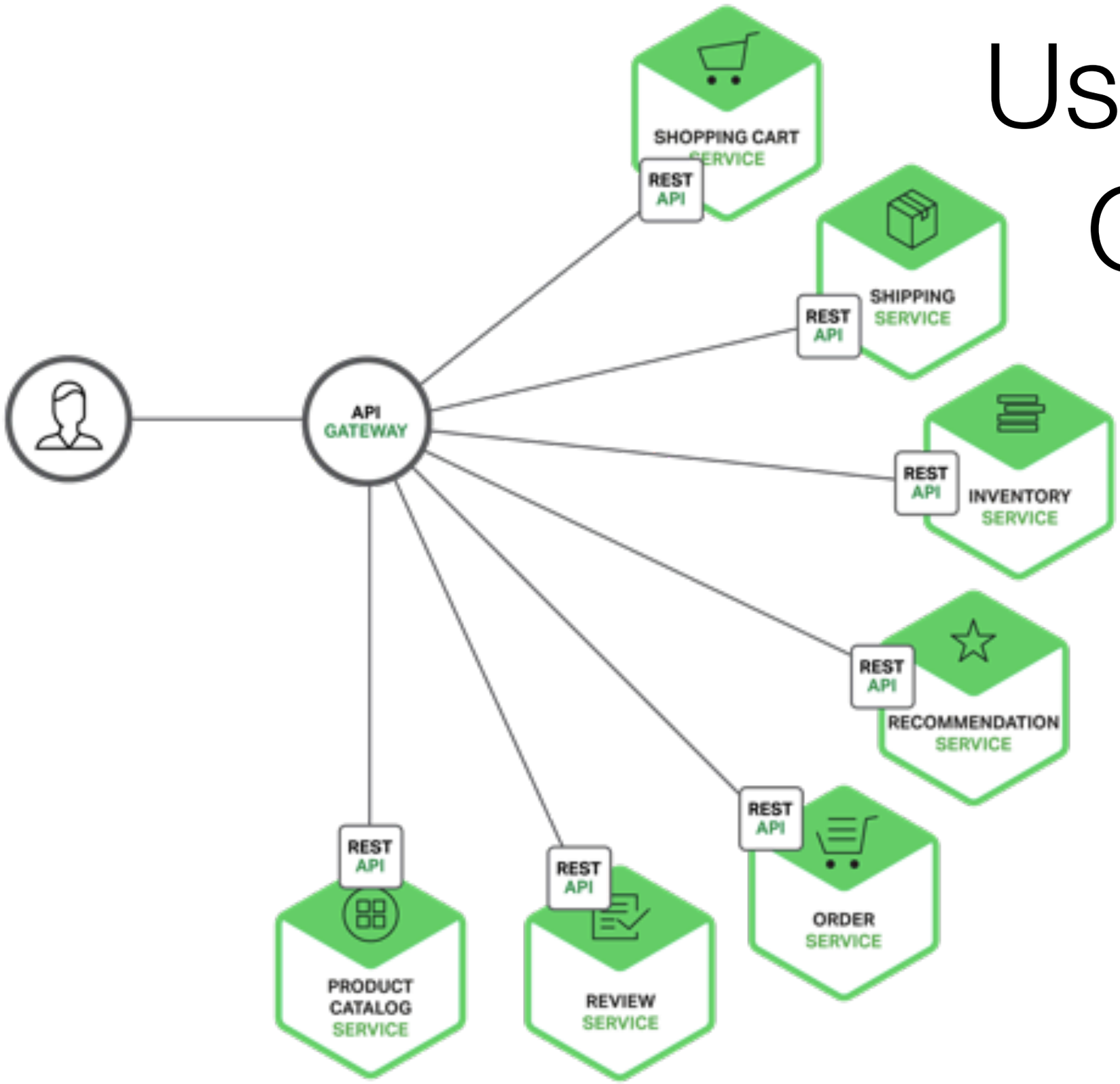
API Gateway


Shopping website
Accounts Service
Promotions Service
Orders Service
Inventory Service
Products Service
Internal Support website
Account
Orders
Offers
Basket
Client
Internet
API Gateway

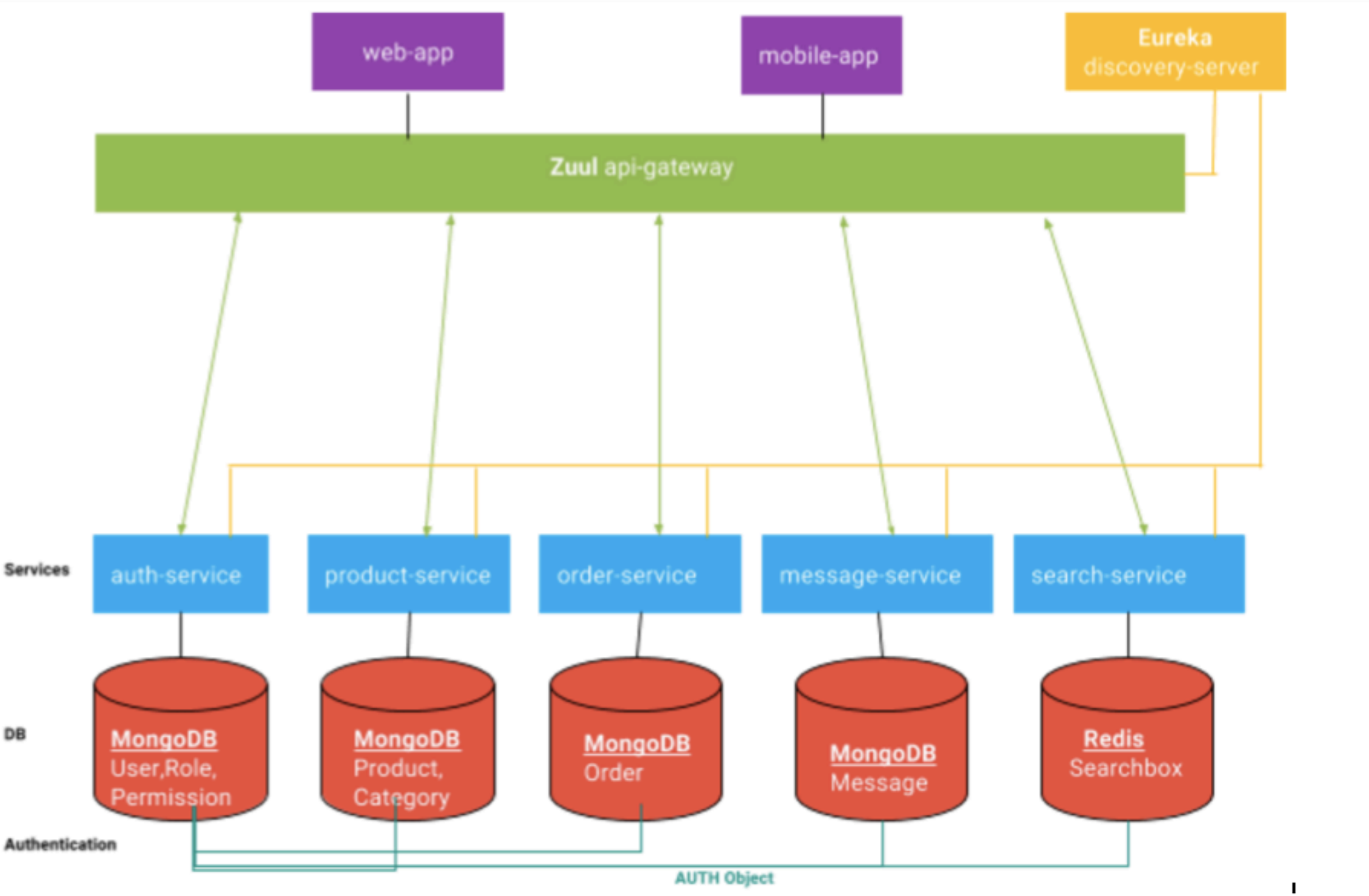
Direct Client to Microservices



Using an API Gateway



Netflix


web-app
mobile-app
Eureka
discovery-server
Zuul api-gateway
Services
auth-service
product-service
order-service
message-service
search-service
DB
MongoDB
User,Role,
Permission
MongoDB
Product,
Category
MongoDB
Order
MongoDB
Message
Redis
Searchbox
Authentication
AUTH Object

Help with

- Creating central entry point
- Exposing services to clients
- One interface to many services
- Dynamic location of services
- Routing to specific instance of service
- Service registry database

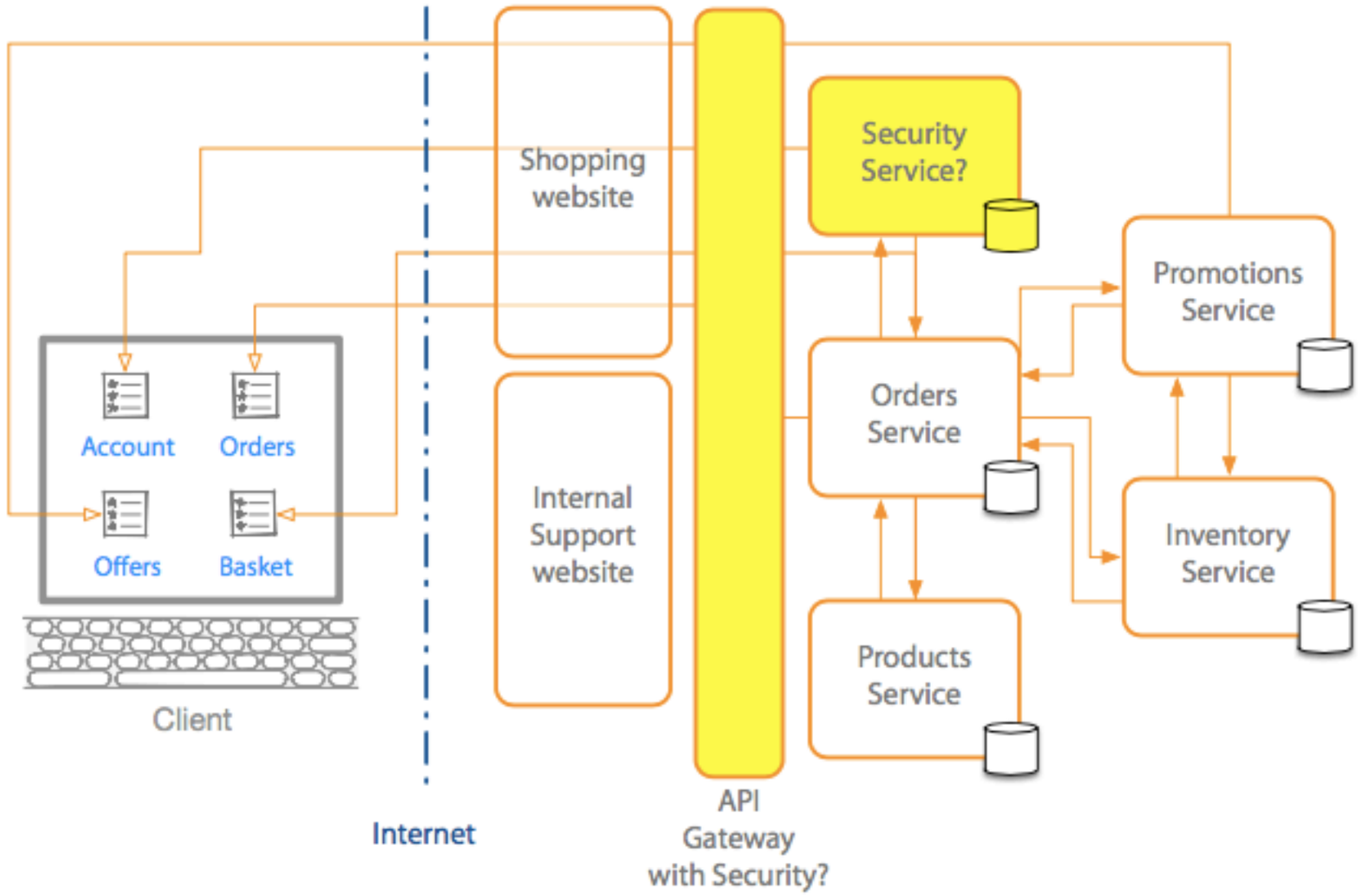
Help with Performance

Load Balancing

Caching

Help with Security

- Dedicated security service
- Central security vs service level



Automation Tools

Automation Tools

Continuous Integration

Continuous Deployment

Continuous Integration Tools

- Team Foundation Server
- TeamCity
- Jenkins

Features

- Cross platform
  Windows builders, Java builders and others

- Source control integration

- Notifications

- IDE Integration (optional)

Map to CI Build

- Code change triggers build of specific service
- Feedback just received on that service
- Builds and tests run quicker
- Separate code repository for service
- End product is in one place
- CI builds to test database changes
- Both microservice build and database upgrade are ready

Avoid one CI build for all services

Continuous Deployment

- Jenkins
- Ansible
- Chef & Puppet

Features

- Central control panel
- Simple to add deployment targets
- Support for scripting
- Support for build statuses
- Integration with CI tool
- Support for multiple environments
- Support for PAAS

Moving Forward

Environment

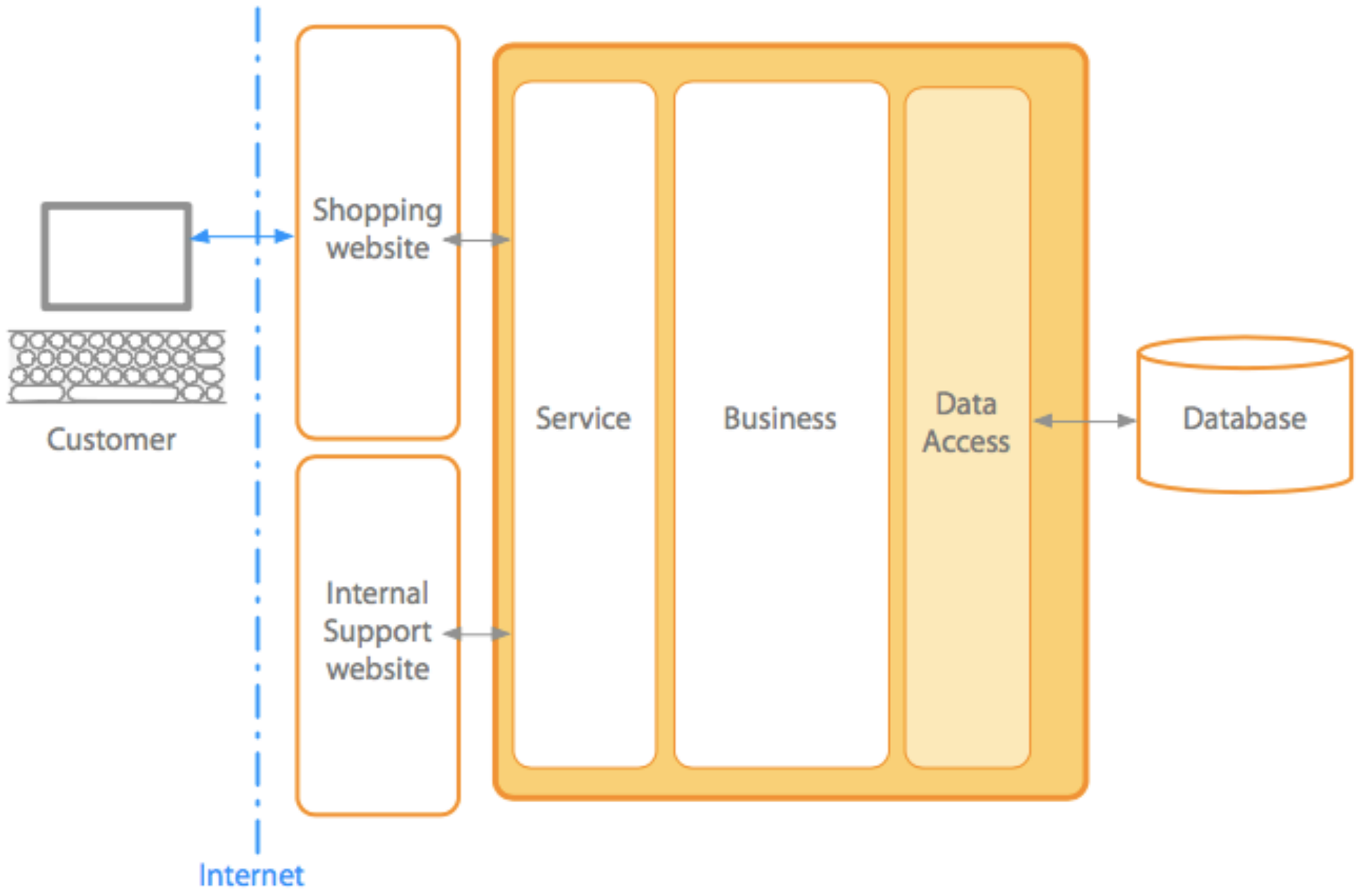
Brownfield
Microservices

Greenfield
Microservices

Brownfield MicroServices

- Existing system
  - Monolithic system
  - Organically grown
  - Seems to large to split
- Lacks microservices design principles

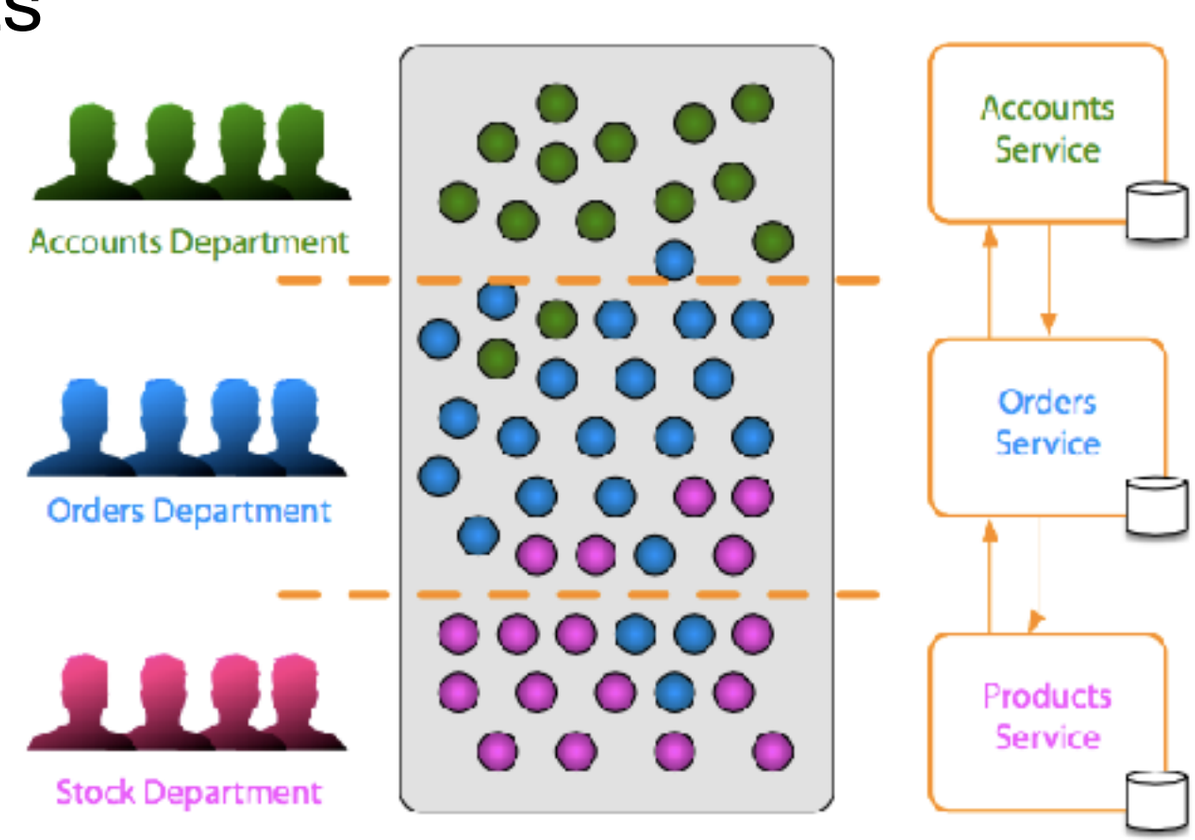
Brownfield MicroServices



Brownfield MicroServices

Identify seams

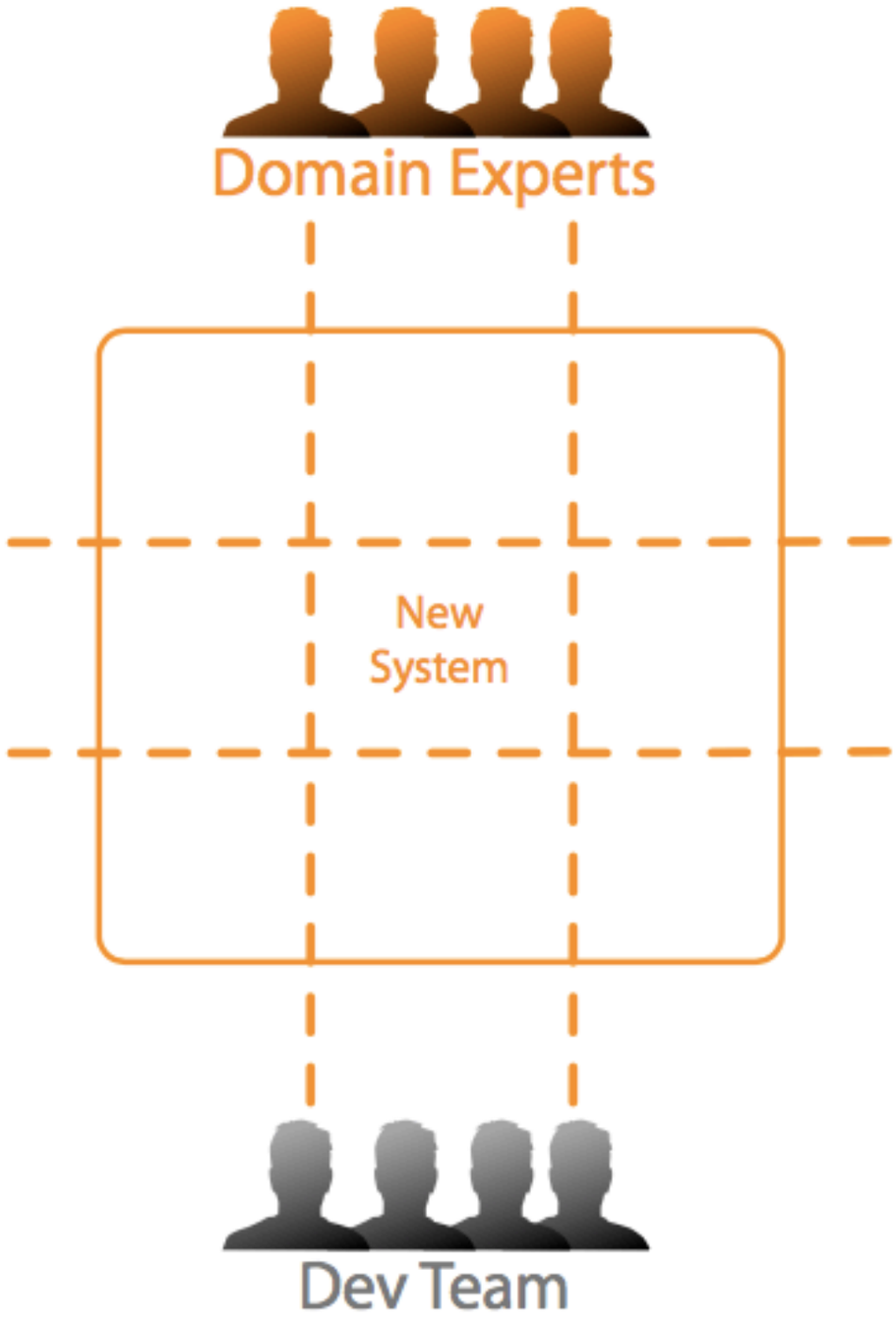
- Separation that reflects domains
- Identify bounded contexts



Seams are future microservice boundaries

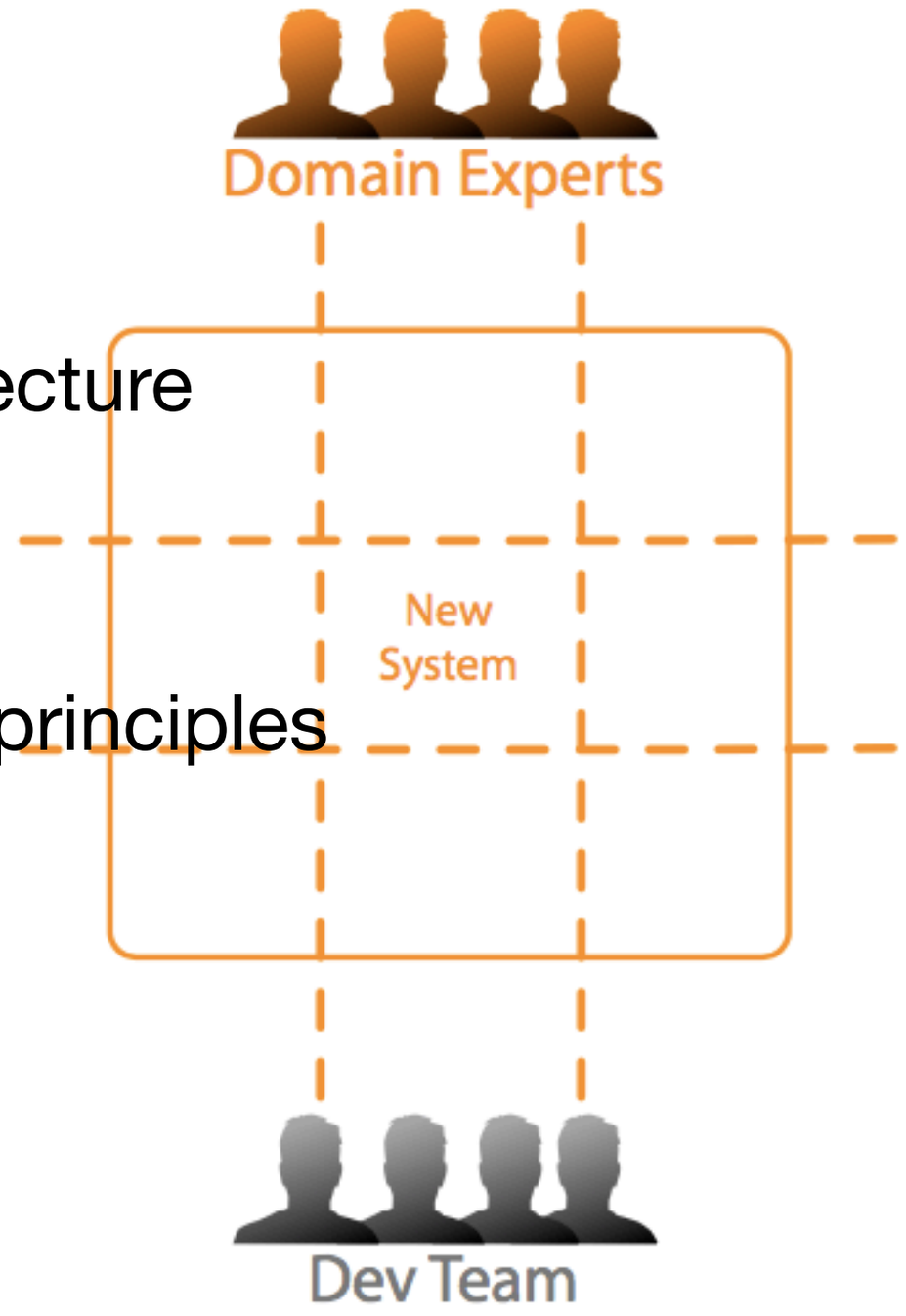
Greenfield MicroServices

- New project
- Evolving requirements
- Business domain
  - Not fully understood
  - Getting domain experts involved
  - System boundaries will evolve
- Teams experience
  - First microservices
  - Experienced with microservices



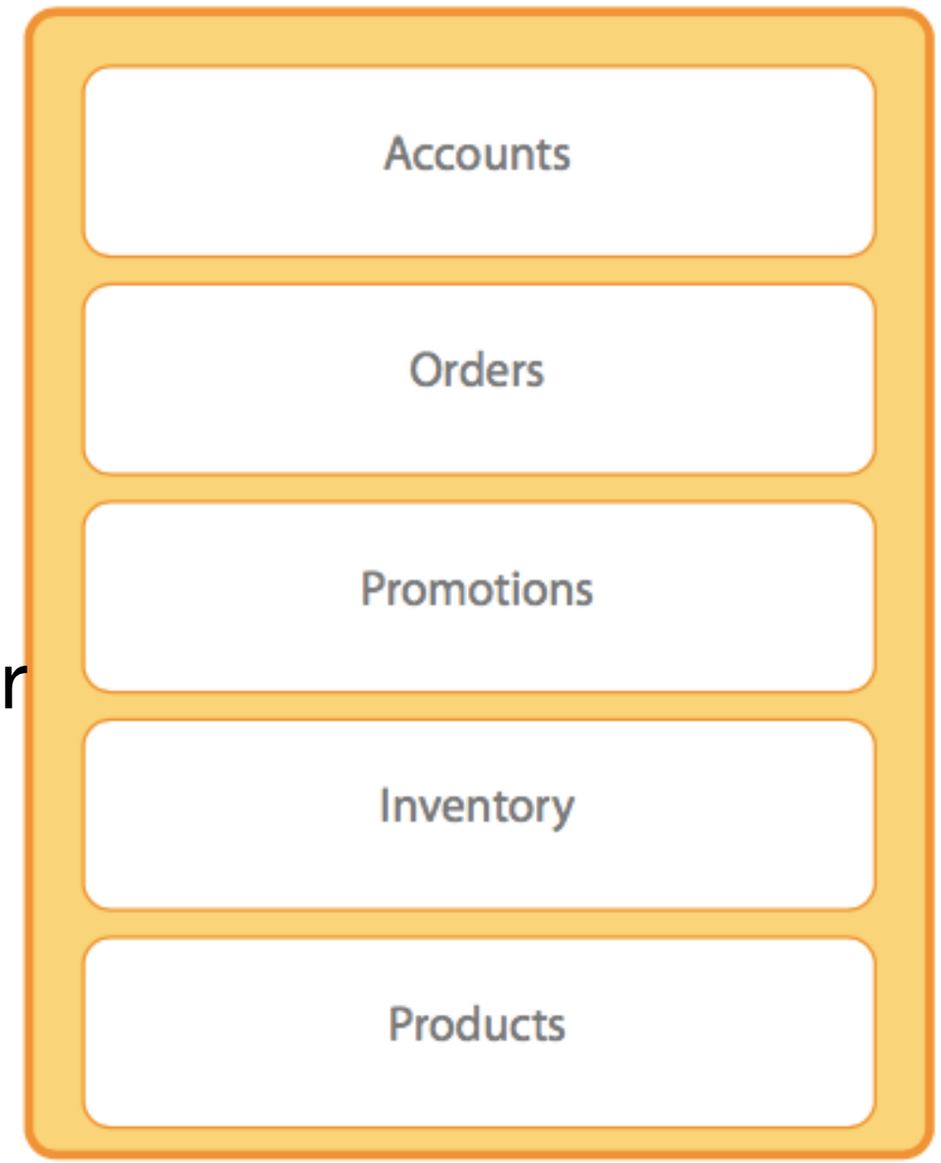
Greenfield MicroServices

- Existing system integration
  - Monolithic system
  - Established microservices architecture
- Push for change
  - Changes to apply microservices principles



Start with Monolithic Design

- High level
- Evolving seams
- Develop areas into modules
- Boundaries start to become clearer
- Refine and refactor design
- Split further when required



Bounded Context

A specific responsibility enforced by an explicit boundary

Core Aspects of Your Domain

Driver

Transport

Shifts

Annual leave

Customer Orders

Orders

Returns

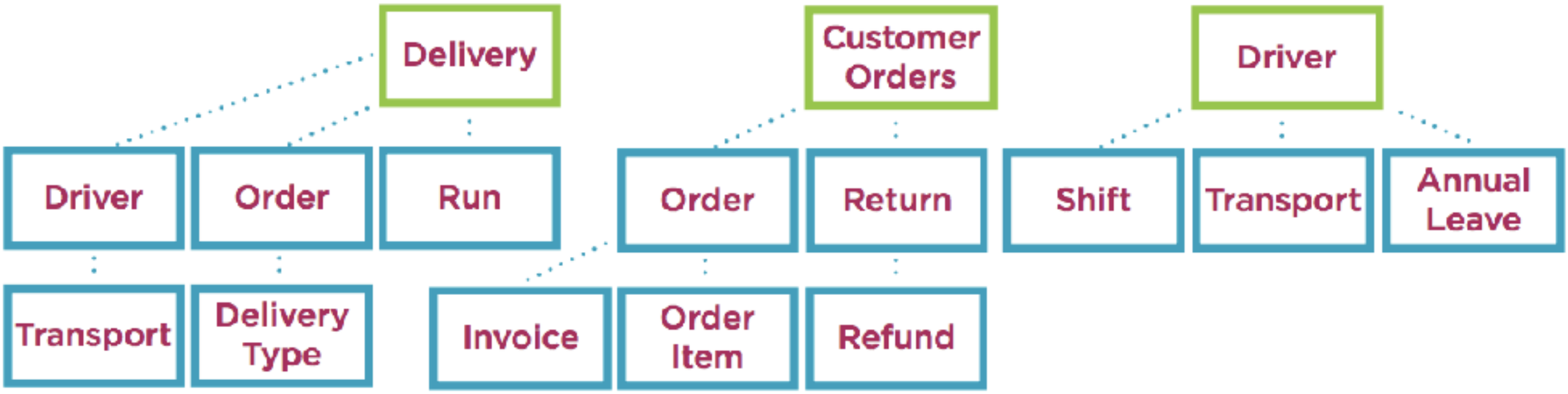
Delivery

Drivers

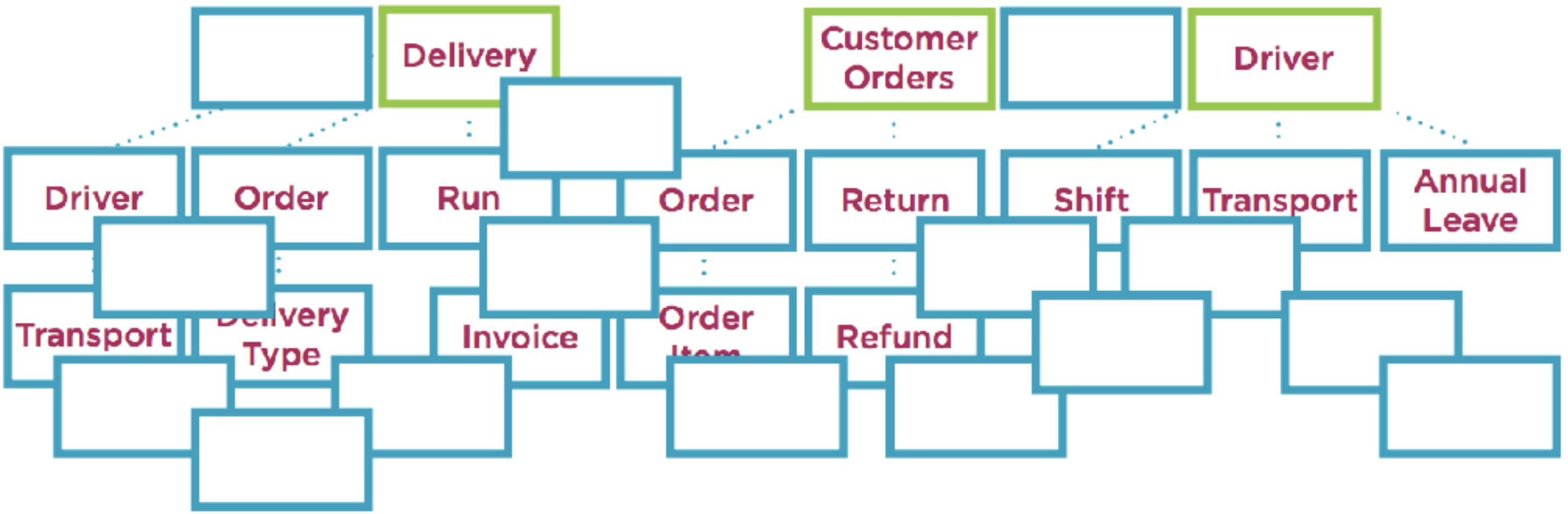
Orders

Runs

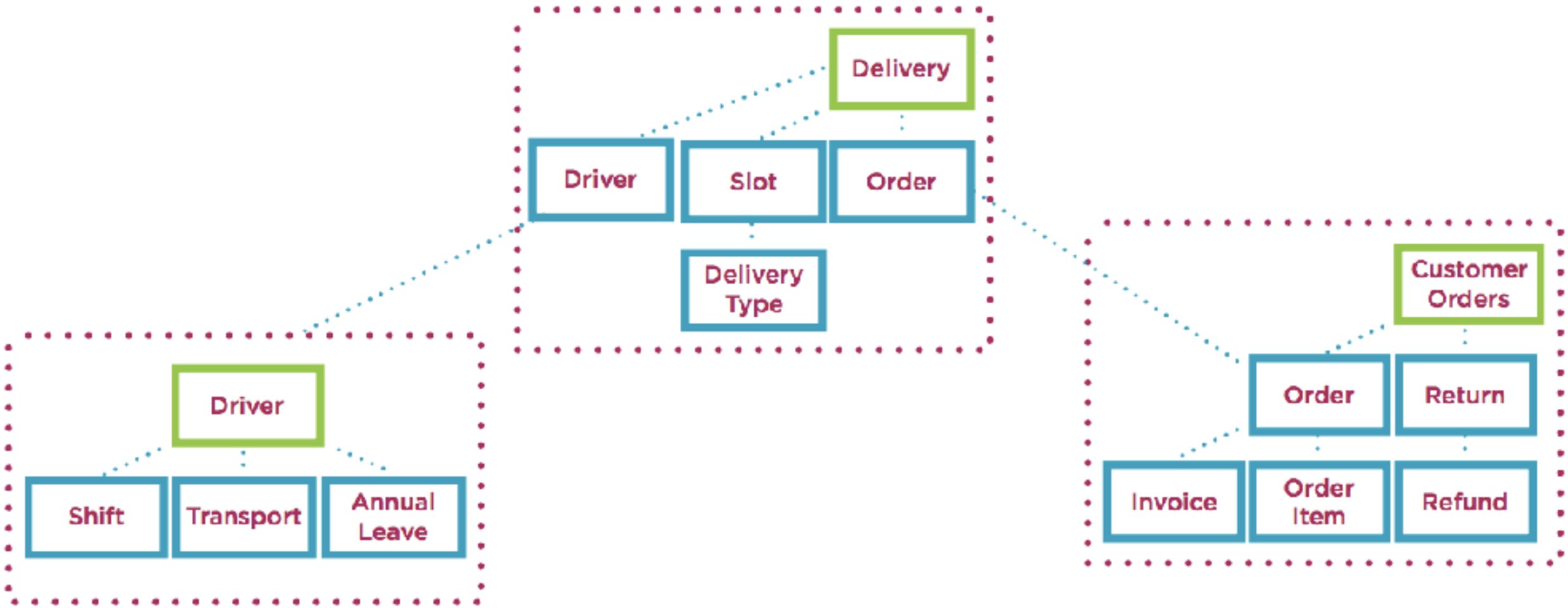
Core Concepts with Supporting Concepts



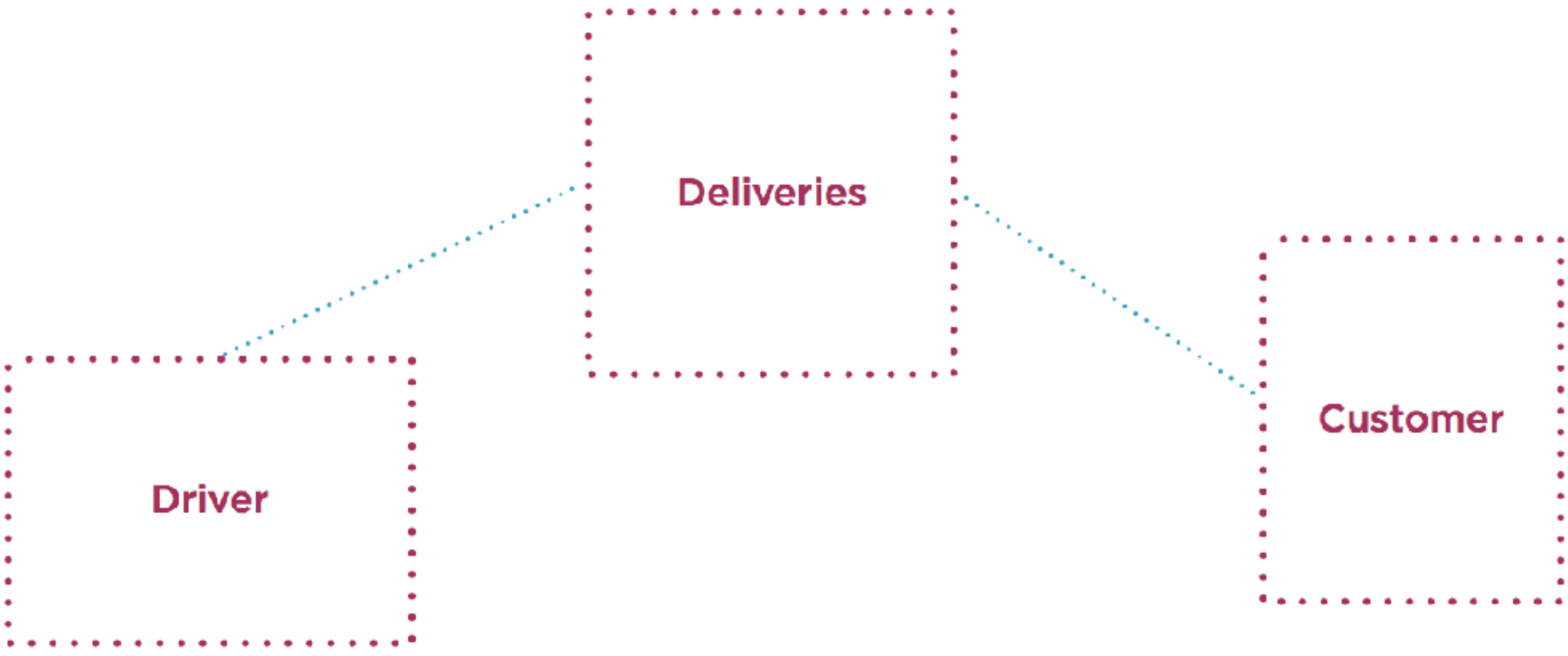
Unbounded Context



Indicate Integration



Become MicroServices



Start Modularizing

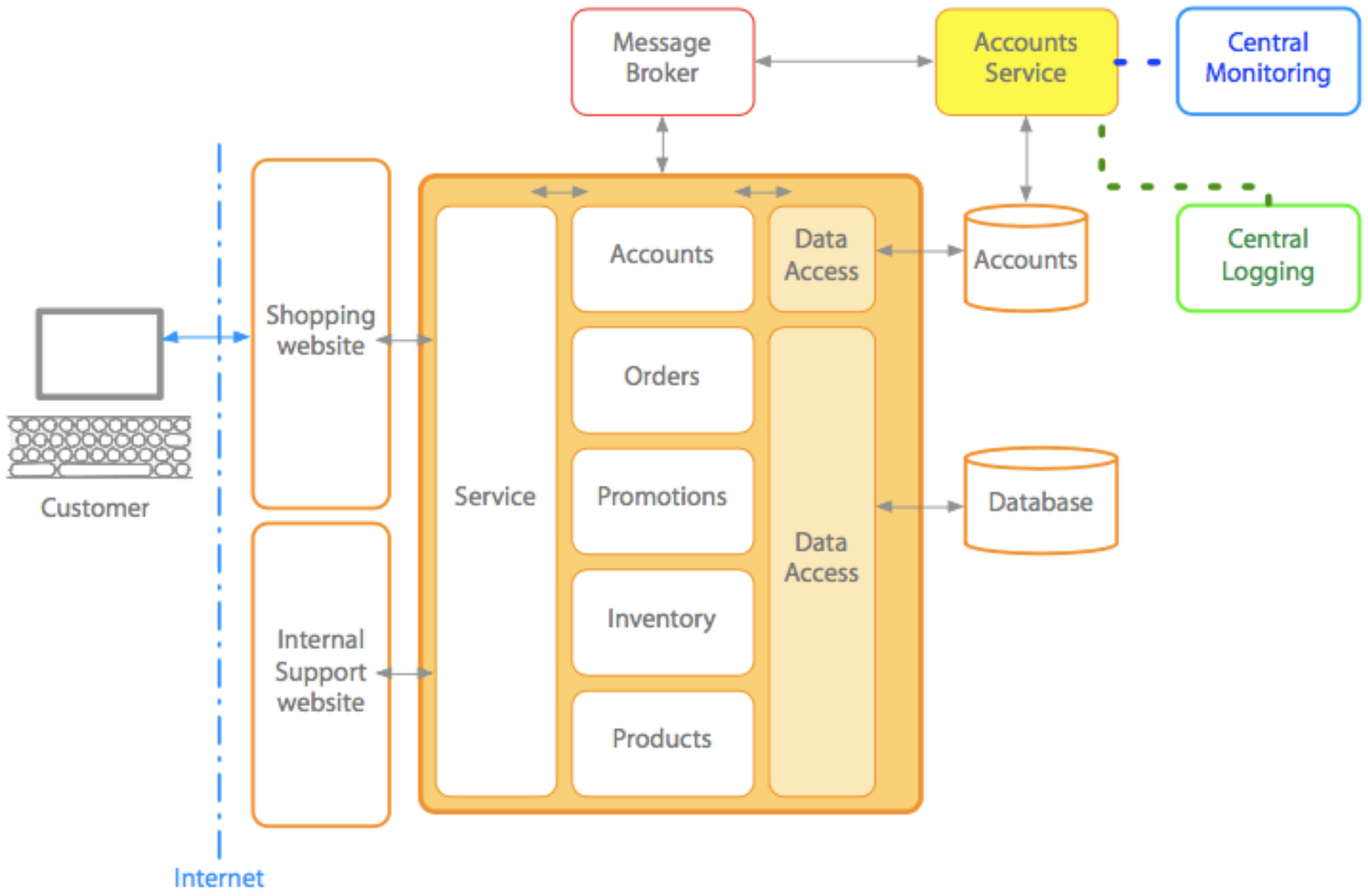
- Move code incrementally
- Tidy up a section per release
  - Take your time
  - Existing functionality needs to remain intact
- Run unit tests and integration tests to validate change
- Keep reviewing

Migration

Migration

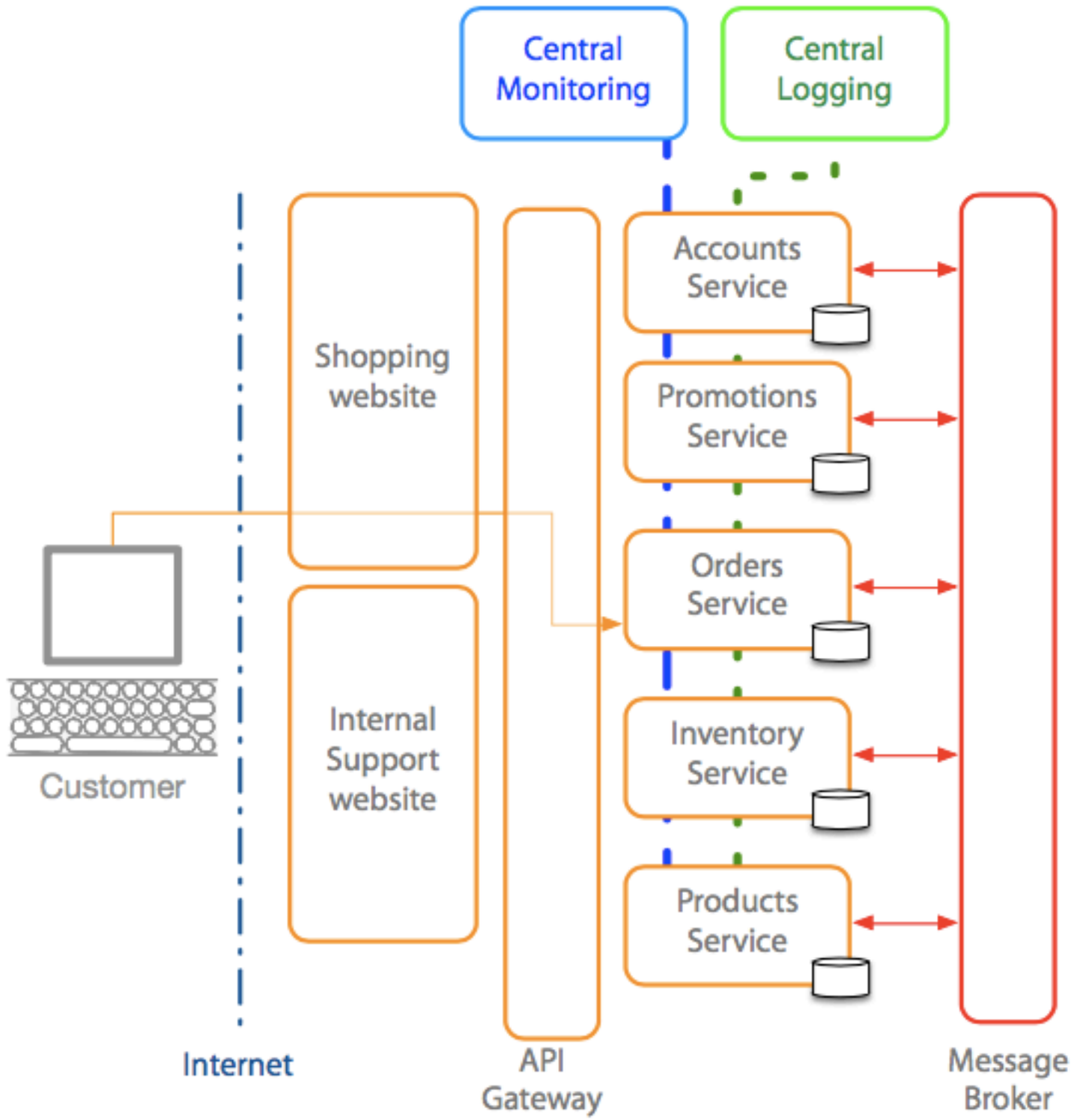
- Code is organized into bounded contexts
  - Code related to a business domain or function is in one place
  - Clear boundaries with clear interfaces between each
- Convert bounded contexts into microservices
  - Start off with one
    Use to get comfortable
  - Make it switchable
    Maintain two versions of the code

Migration

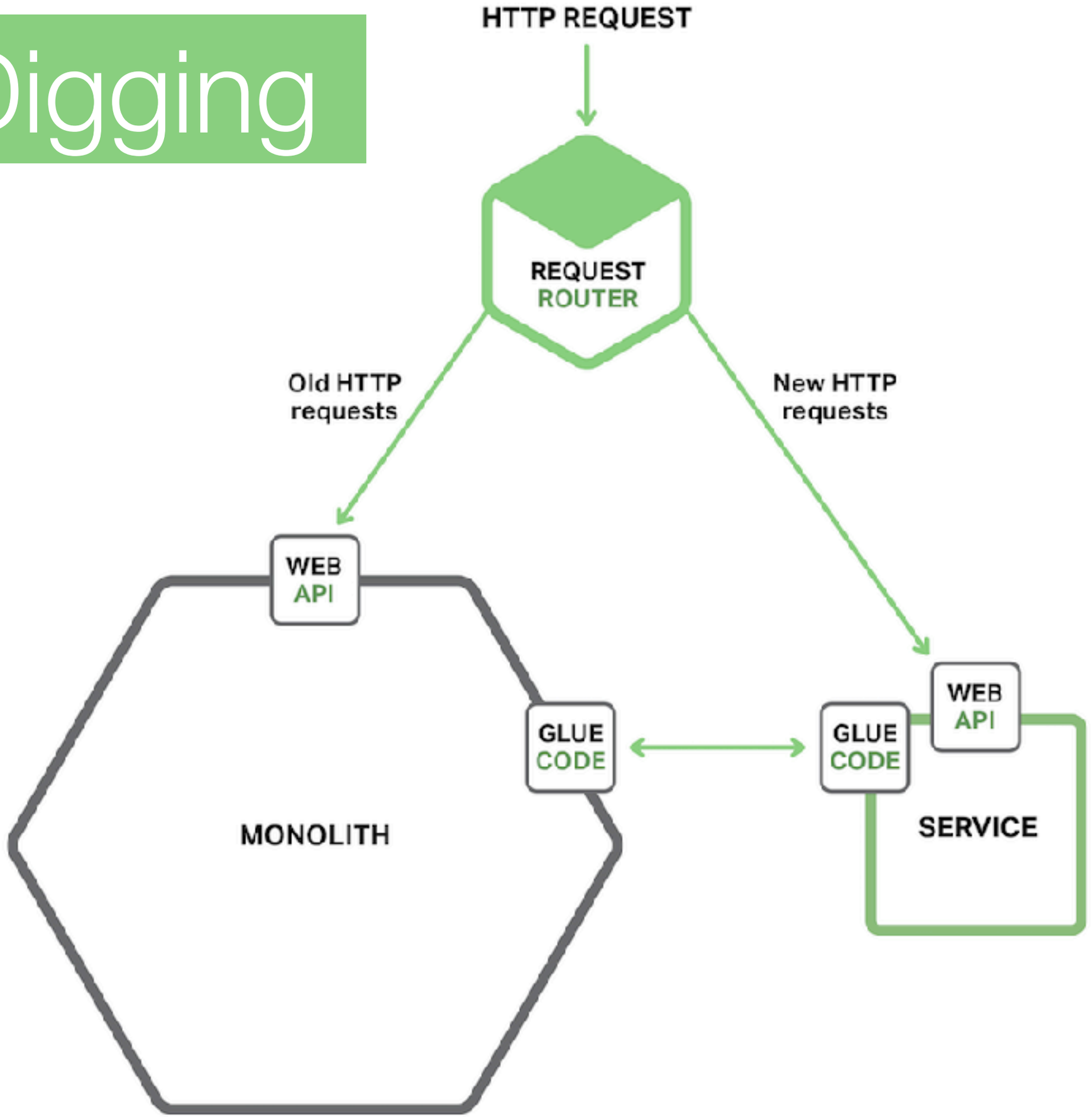


Incremental approach

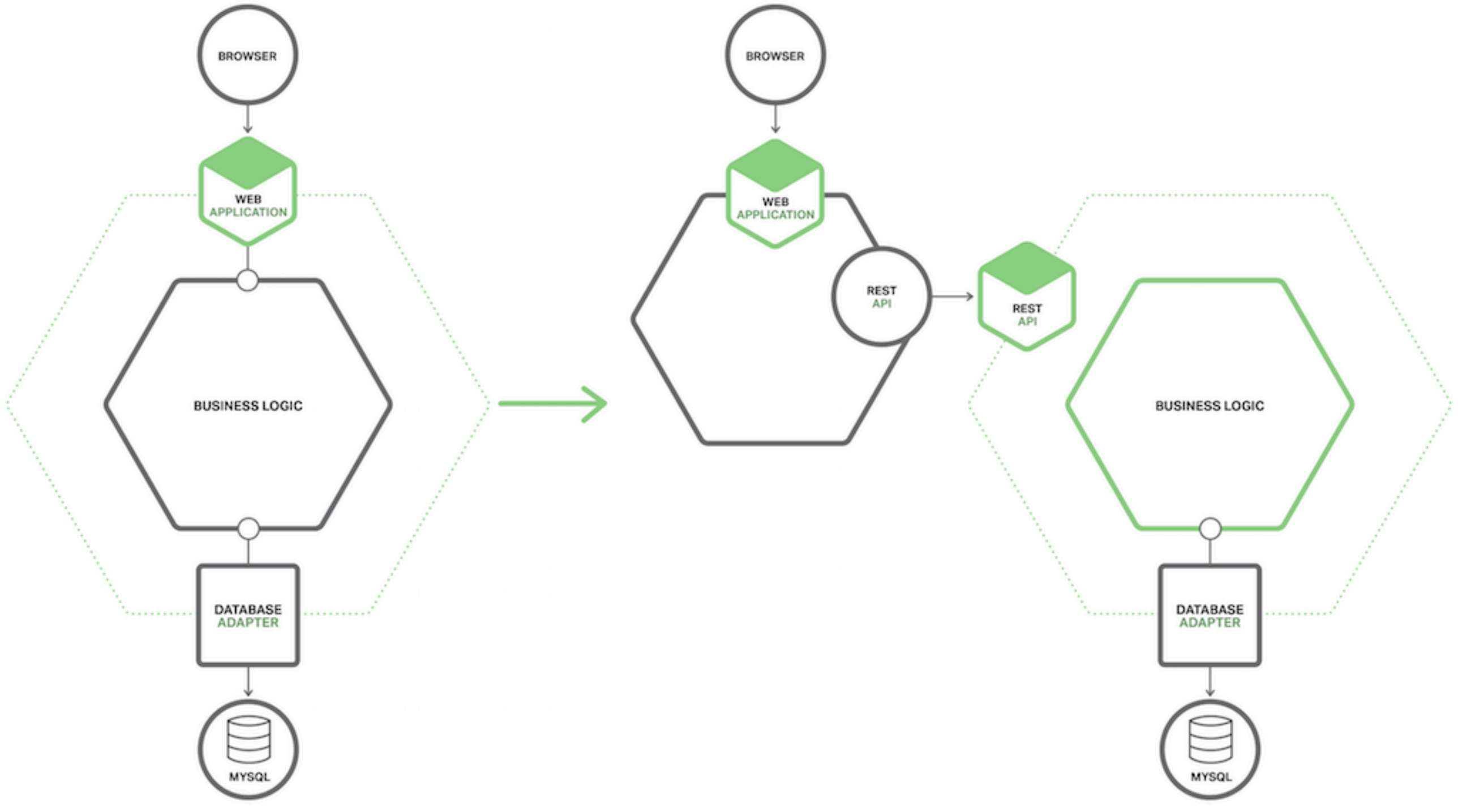
Migration


Central Monitoring
Central Logging
Shopping website
Accounts Service
Promotions Service
Orders Service
Inventory Service
Products Service
Internal Support website
Customer
Internet
API Gateway
Message Broker

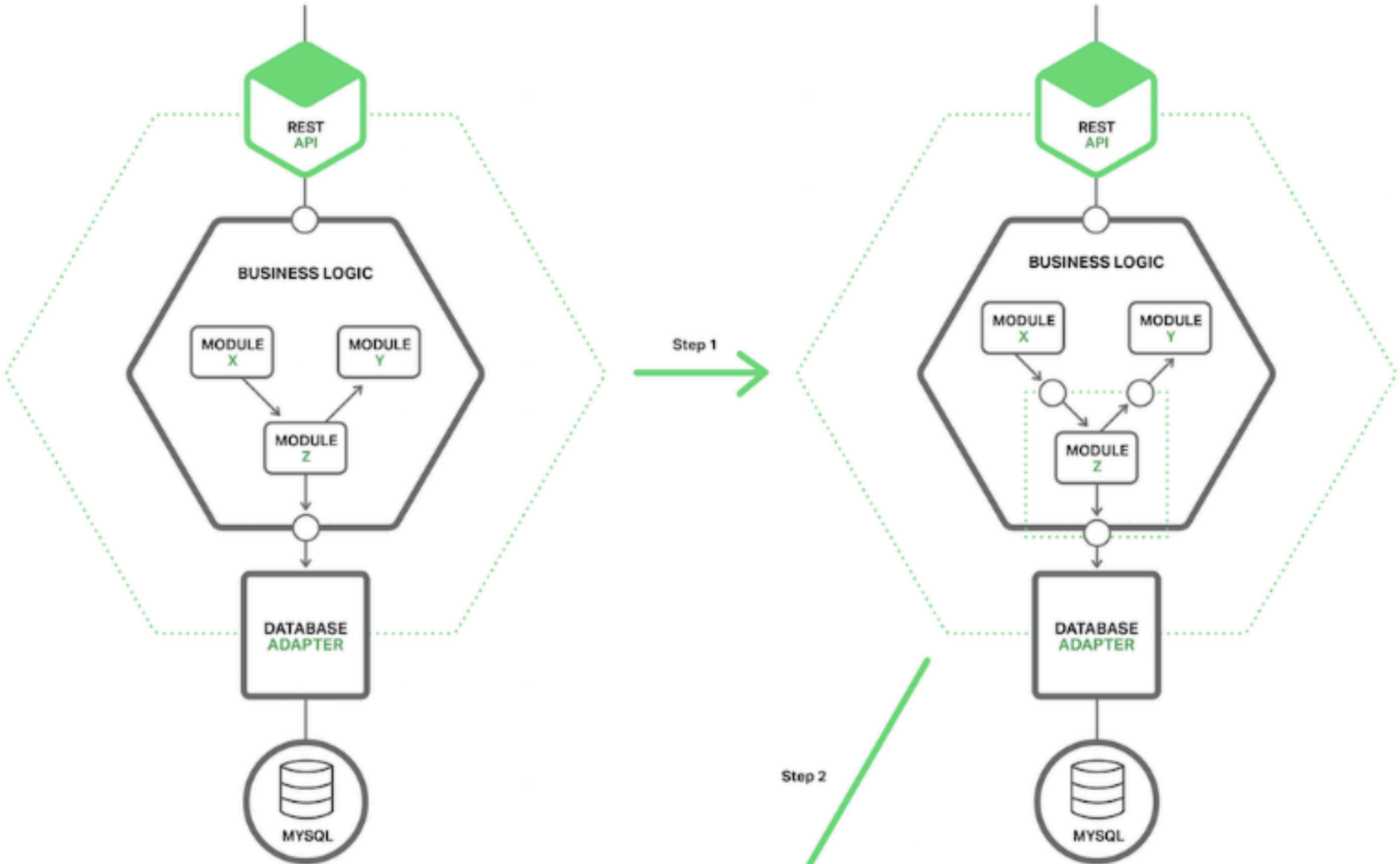
Stop Digging



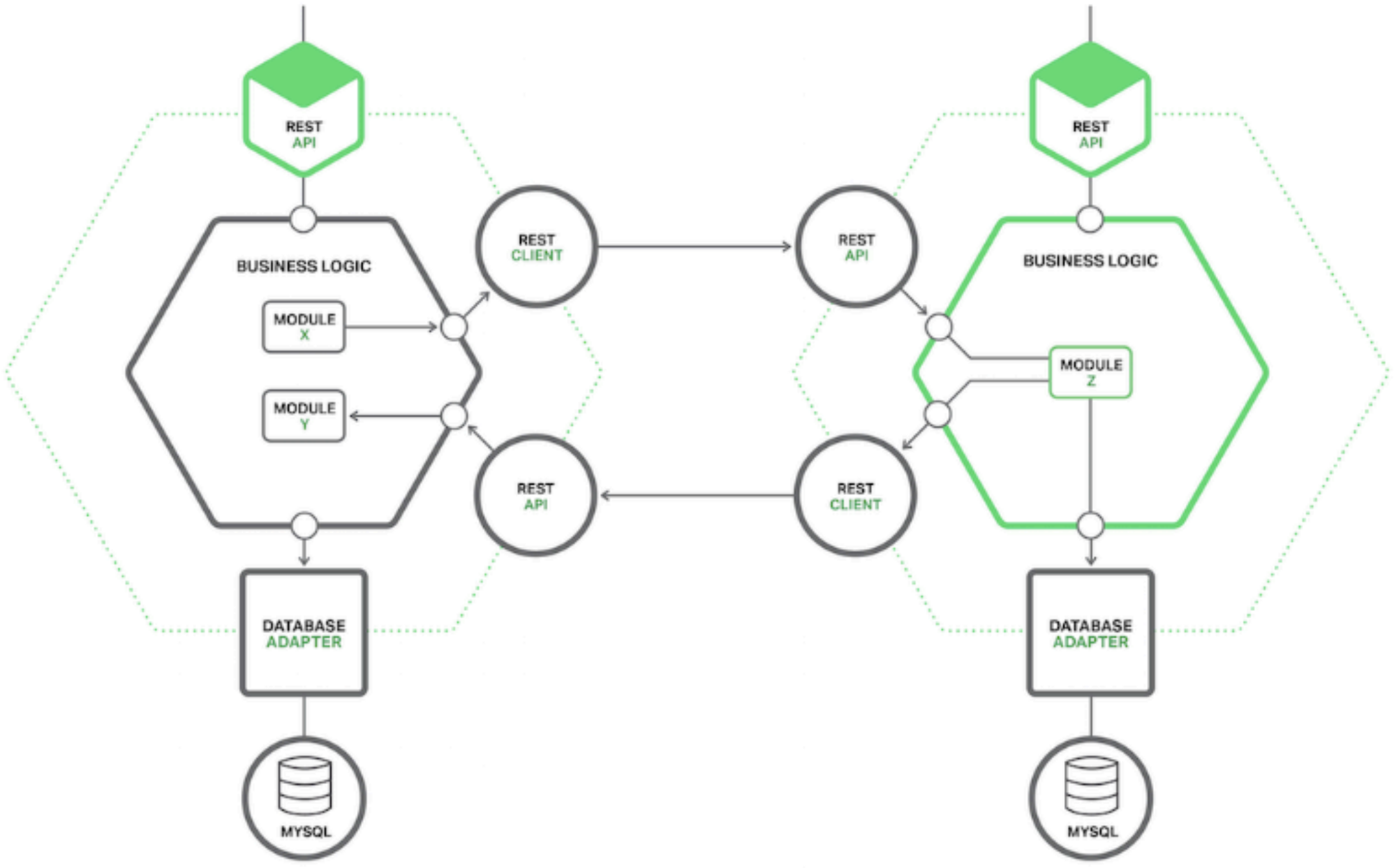
Split Frontend and Backend



Extract Service



Extract Service



How to Prioritize

Risk

Technology

Dependencies

Integrate with Monolith

- Monitor both for impact
- Monitor operations that talk to microservices
- Review and improve infrastructure
- Incrementally the monolithic will be converted

API Architecture Pattern

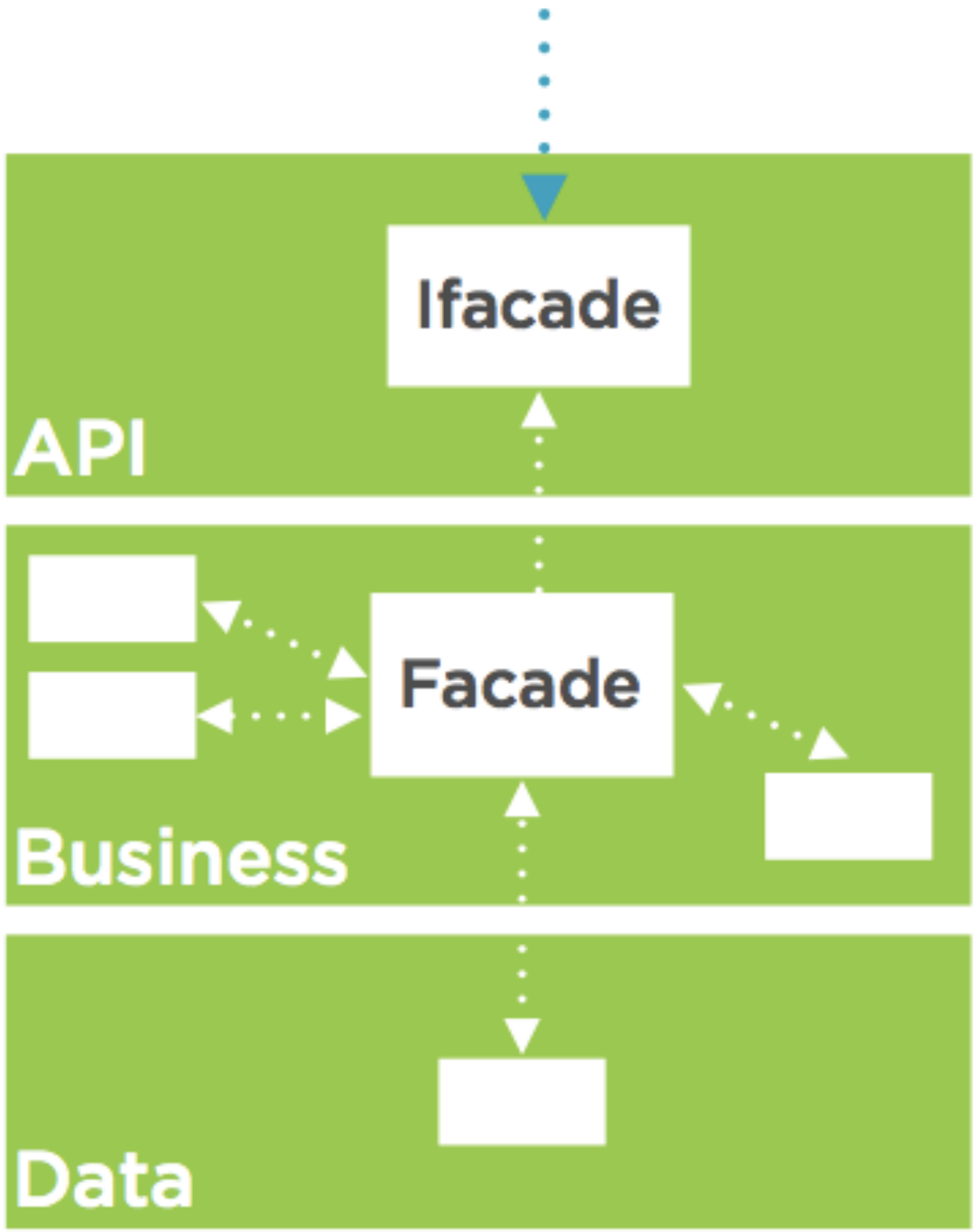
Design Pattern

Facade
design pattern

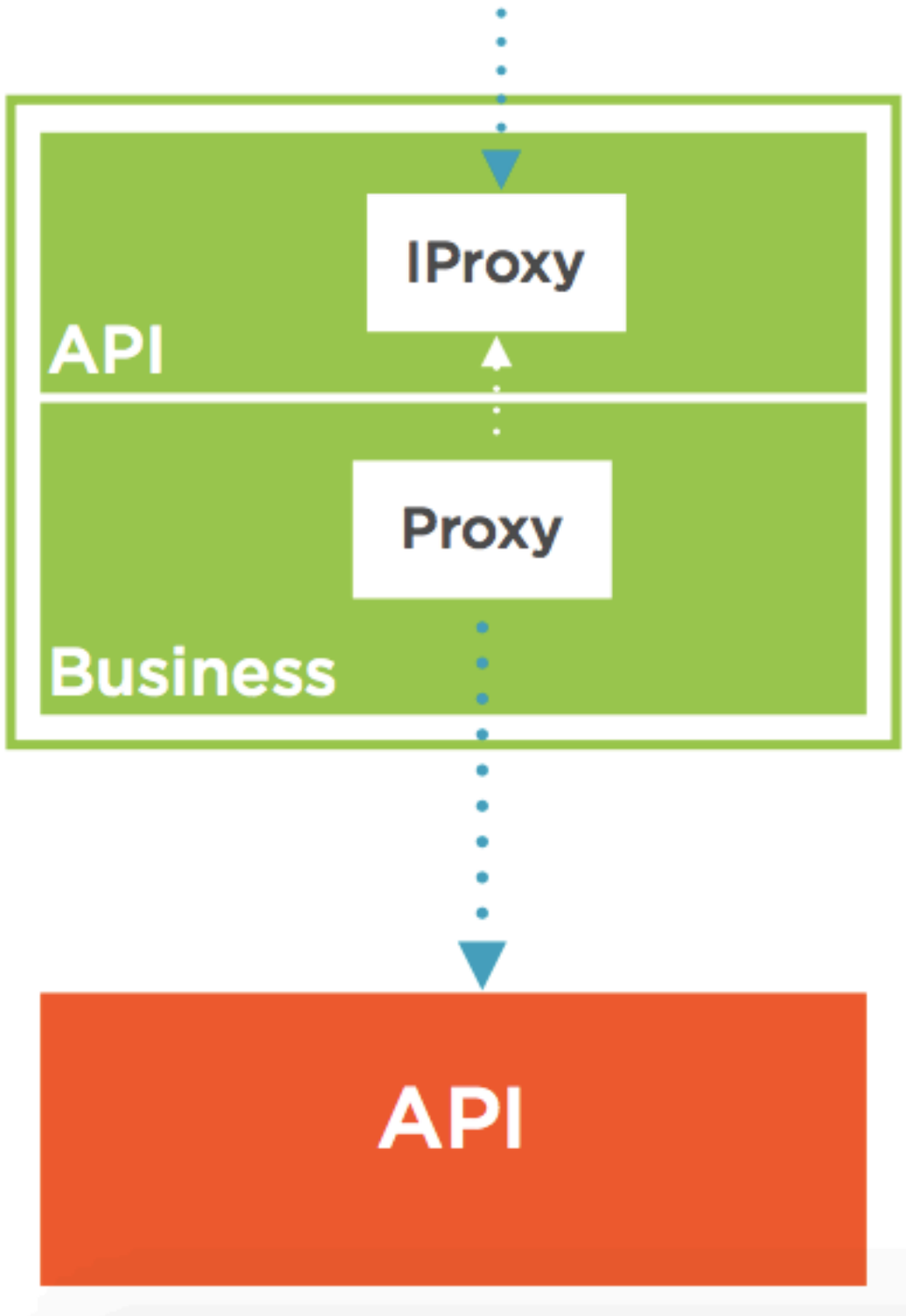
Proxy
design pattern

Stateless
design pattern

Facade Design Pattern



Proxy Design Pattern



- Simplification
- Transformation
- Security
- Validation

Stateless Design Pattern

- What are stateful services
  Avoid for microservices

- Stateless Service Pattern
  - No state information maintained
  - Clients maintain state
  - State sent as part of request

- Advantages for microservice architecture
  Scalability, performance and availability

Increase Network Traffic

Database Migration

Database Migration

- Avoid shared databases

- Split databases using seams
  Relate tables to code seams

- Supporting the existing application
  Data layer that connects to multiple database

- Tables that link across seams
  API calls that can fetch that data for a relationship

Database Migration

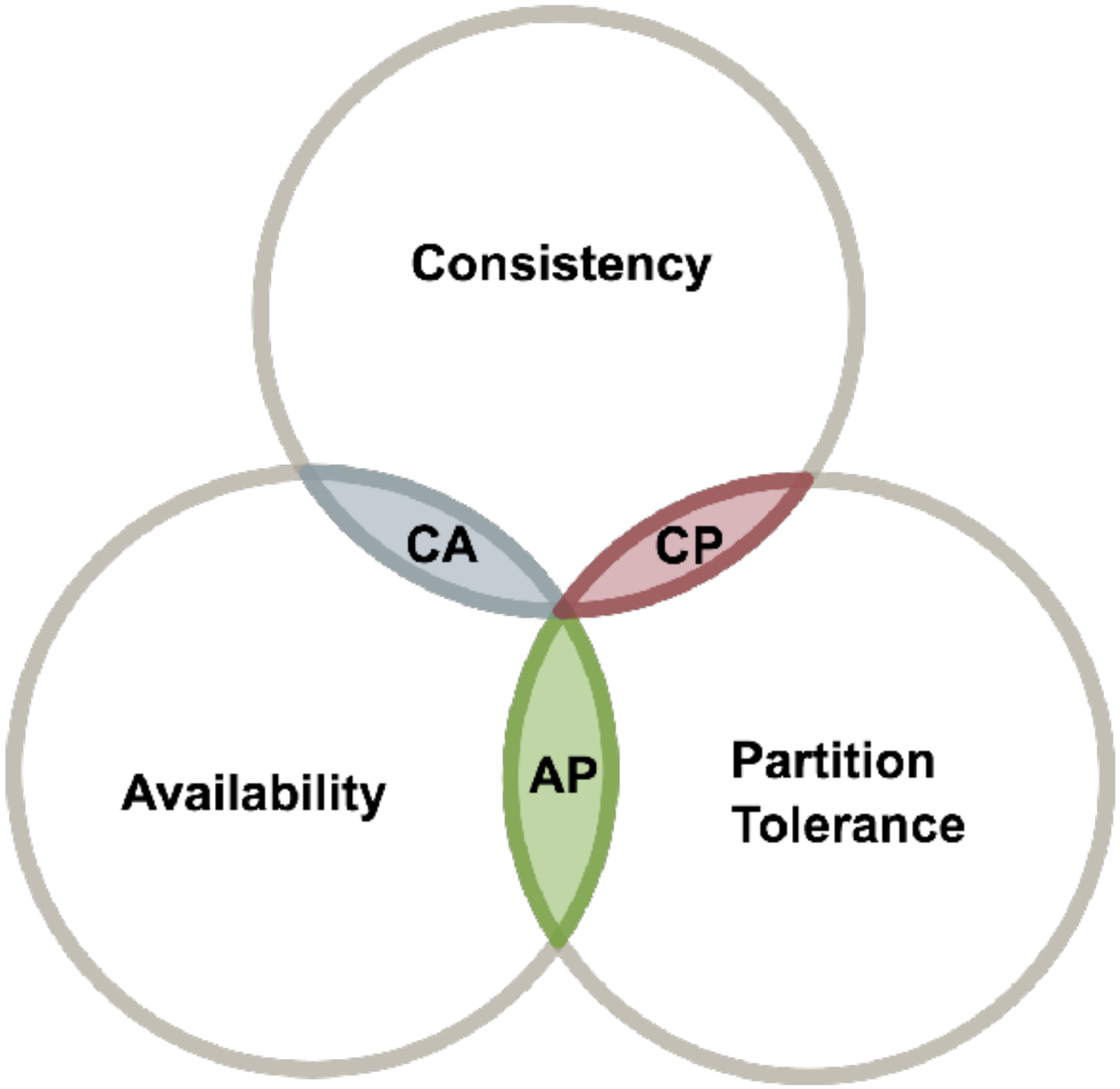
- Refactor database into multiple databases
- Data referential integrity
- Static data tables
- Shared data

Data Consistency

Data Consistency

- Transactions are the traditional approach
- Monolithic system transactions
  Single database which is the single truth
- Microservice transactions
  - Distributed architecture
  - Distributed data
  - Distributed transactions
- CAP theorem
  - Network failure will happen
  - Data availability or data consistency?

CAP Theorem



Options

- Traditional ACID transactions
  Atomicity, consistency, isolation and durability

- Two phase commit pattern
  - ACID is mandatory
  - CAP Theorem: Choosing consistency

- Saga Pattern
  Trading atomicity for availability and consistency

- Eventual consistency pattern
  - Compromise ACID
  - CAP Theorem: Choosing availability

Transactions

ACID

- Atomic
  Either succeeds completely or fails completely

- Consistency
  Constraints of underly datastore are enforced

- Isolate
  Can’t read by other transactions until in a specific state base on isolation rules

- Durability
  Once saved, guaranteed to be in the datastore until modified

Transactions

- Transactions ensure data integrity
- Transactions are simple in monolithic applications
- Transactions spanning microservices are complex
  - Complex to observe
  - Complex to problem solve
  - Complex to rollback

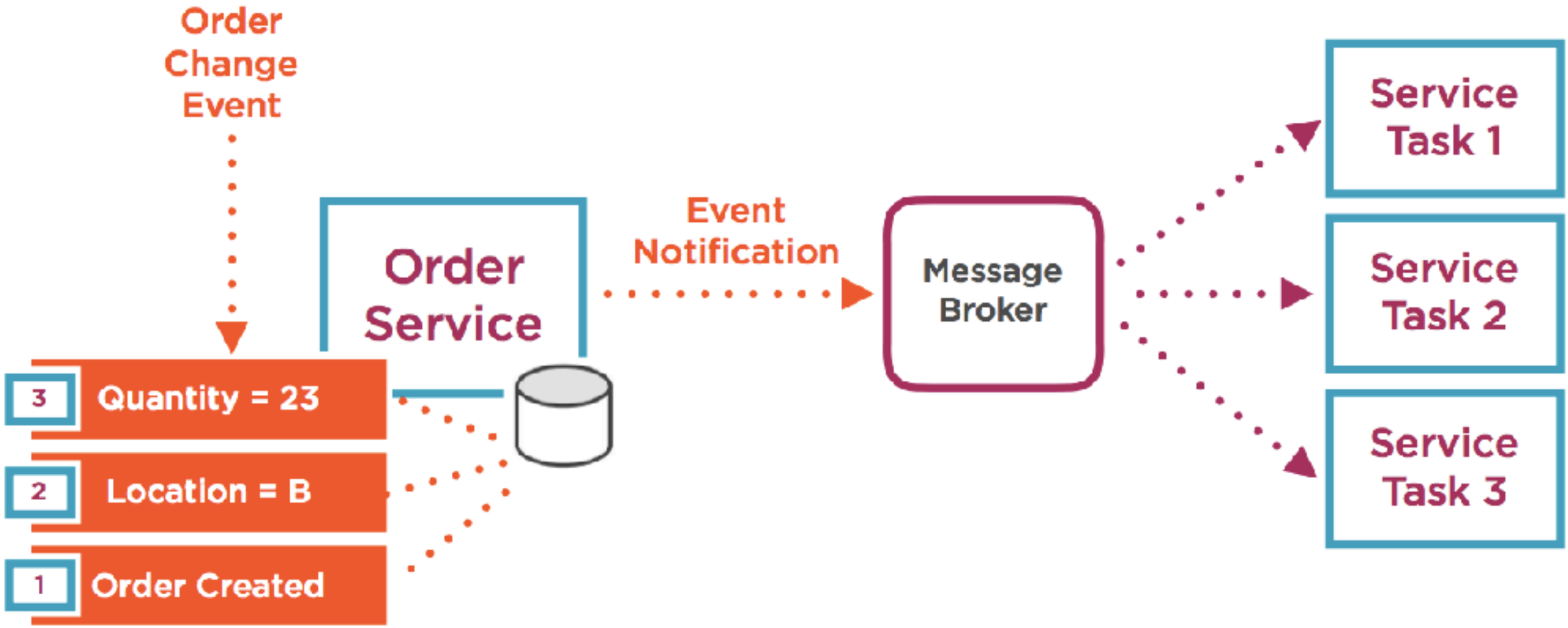
No ACID

Options for Failed

- Try again later
- Abort entire transaction
- Use a transaction manager (Two phase commit)
- Disadvantage of transaction manager
  - Reliance on transaction manager
  - Delay in processing
  - Potential bottleneck
  - Complex to implement

Patterns for Database Design

Event Sourcing



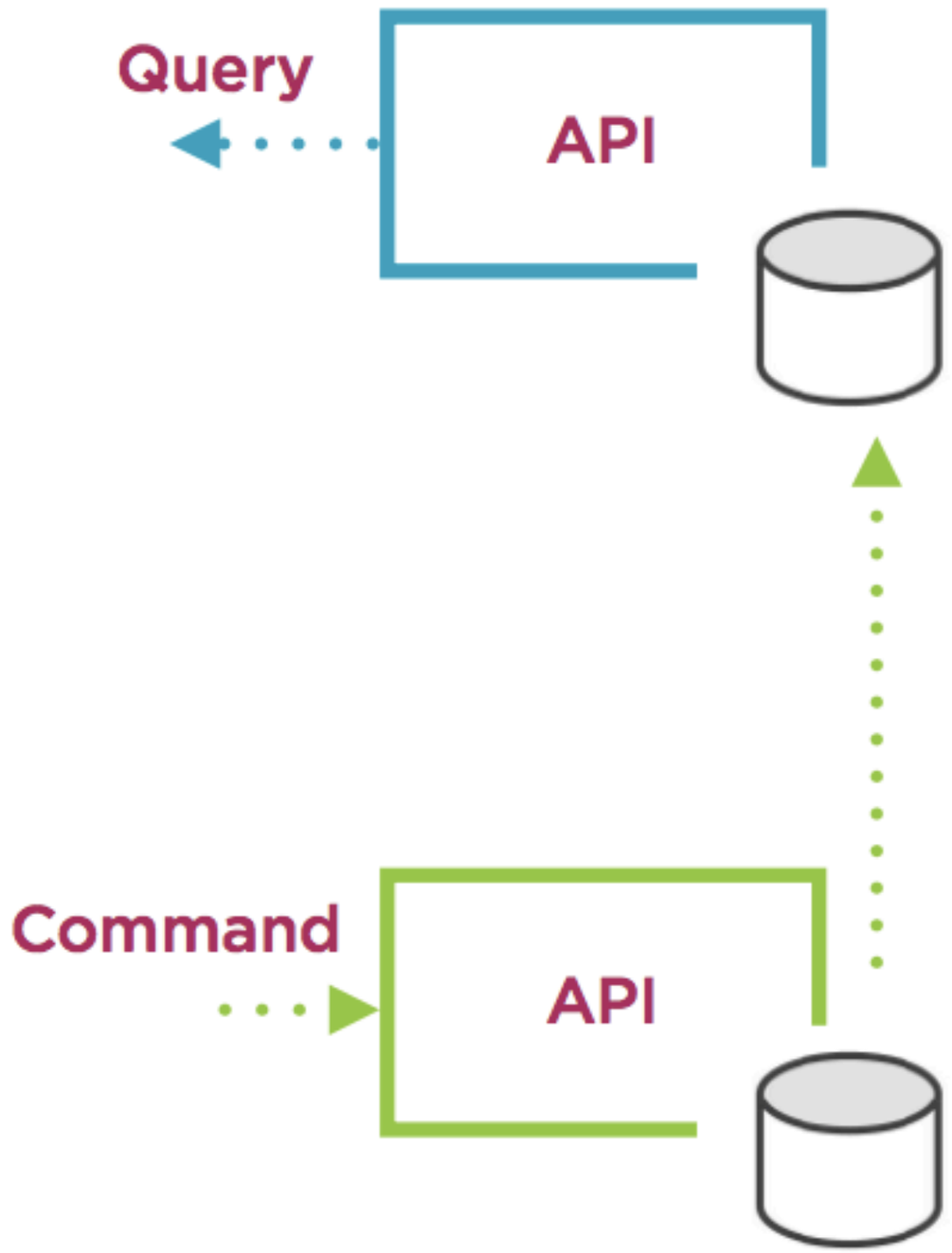
CQRS

- CQRS
  - Command models and or services
  - Query models and or services

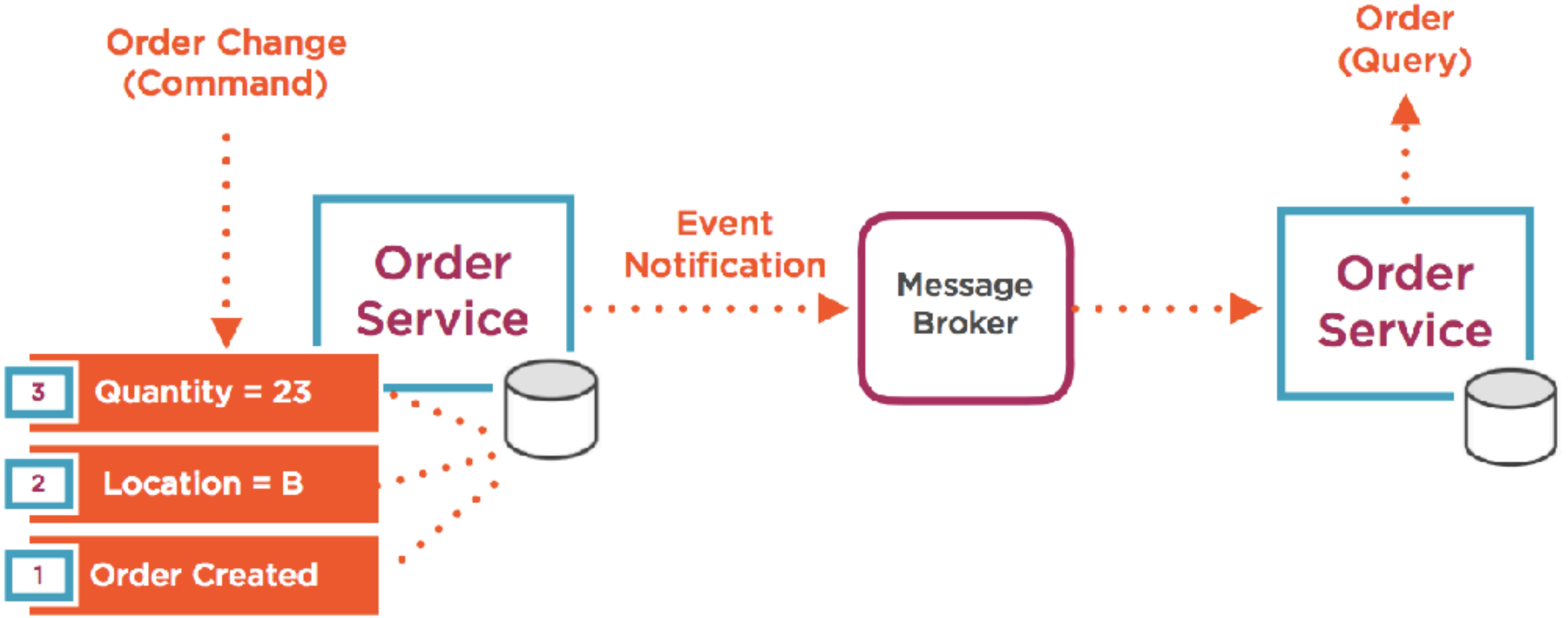
- Why
  - Separation of responsibilities
  - Event notifications handled by command - Reporting/ functions handled by query
  - Separation of technologies - Service and storage

- Challenges
  Command and query database syncing

CQRS

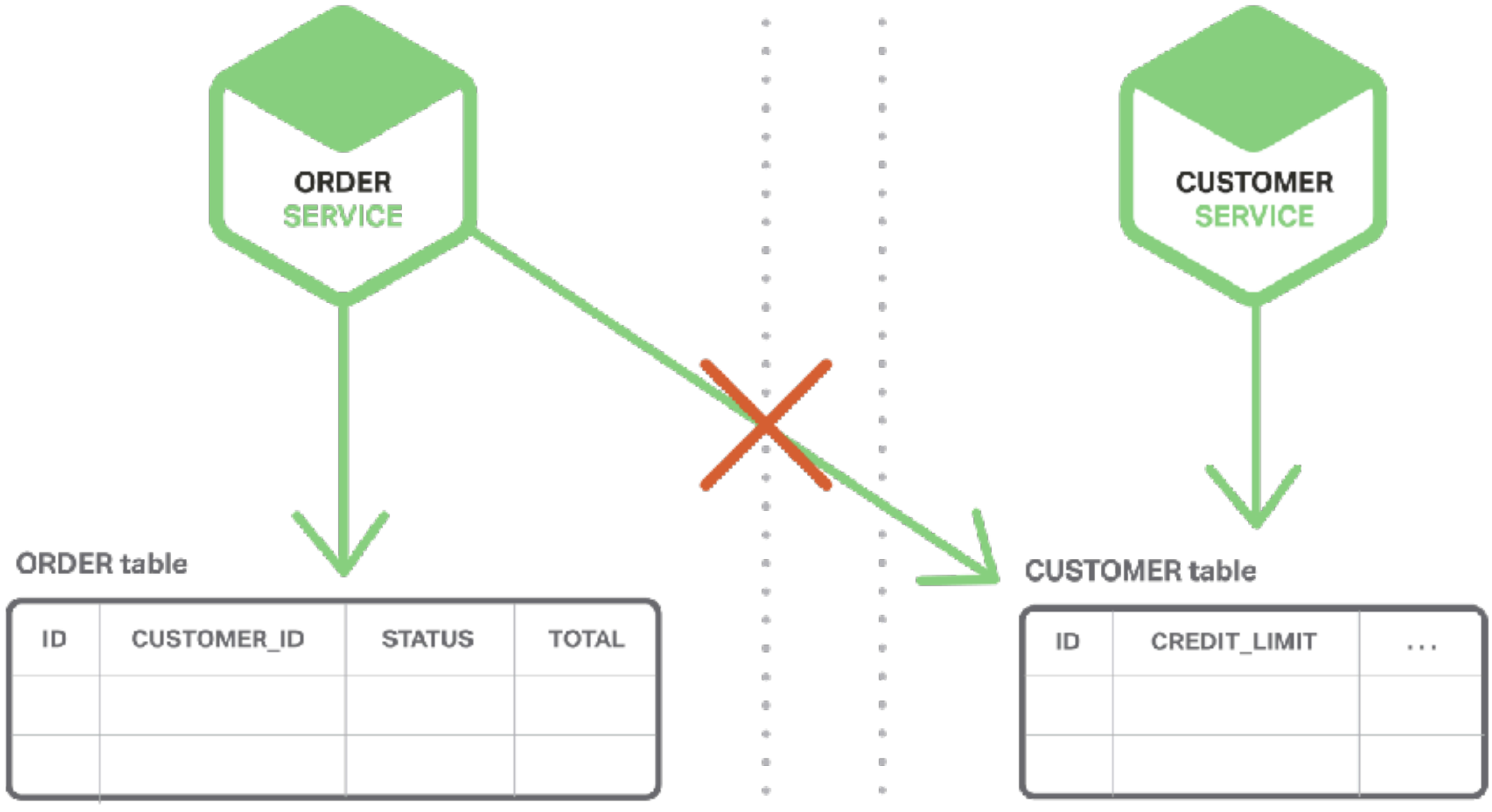


Event Sourcing and CQRS



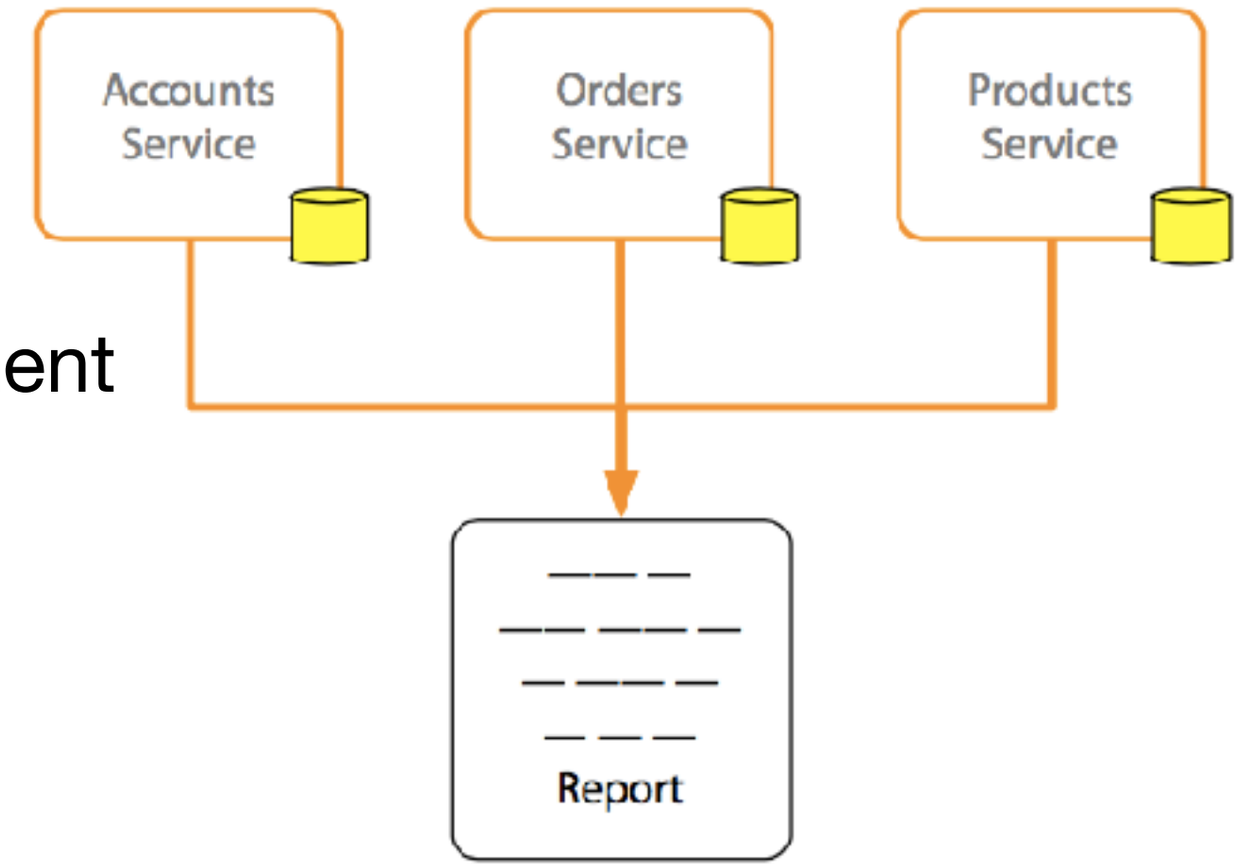
Reporting

No Shared Data



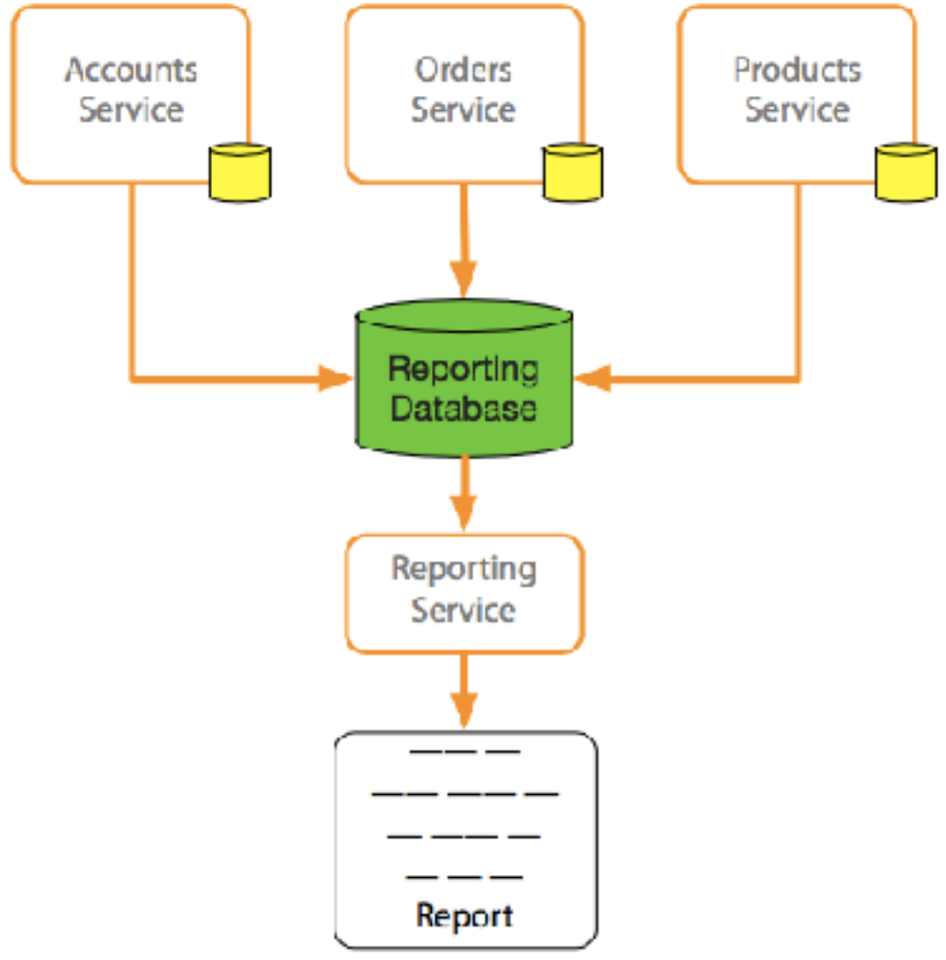
Complicating Report

- Data split across microservices
- No central database
- Joining data across databases
- Slower reporting
- Complicate report development



Possible Solution

- Service calls for report data
- Data dumps (Not good for real-time application)
- Consolidation environment



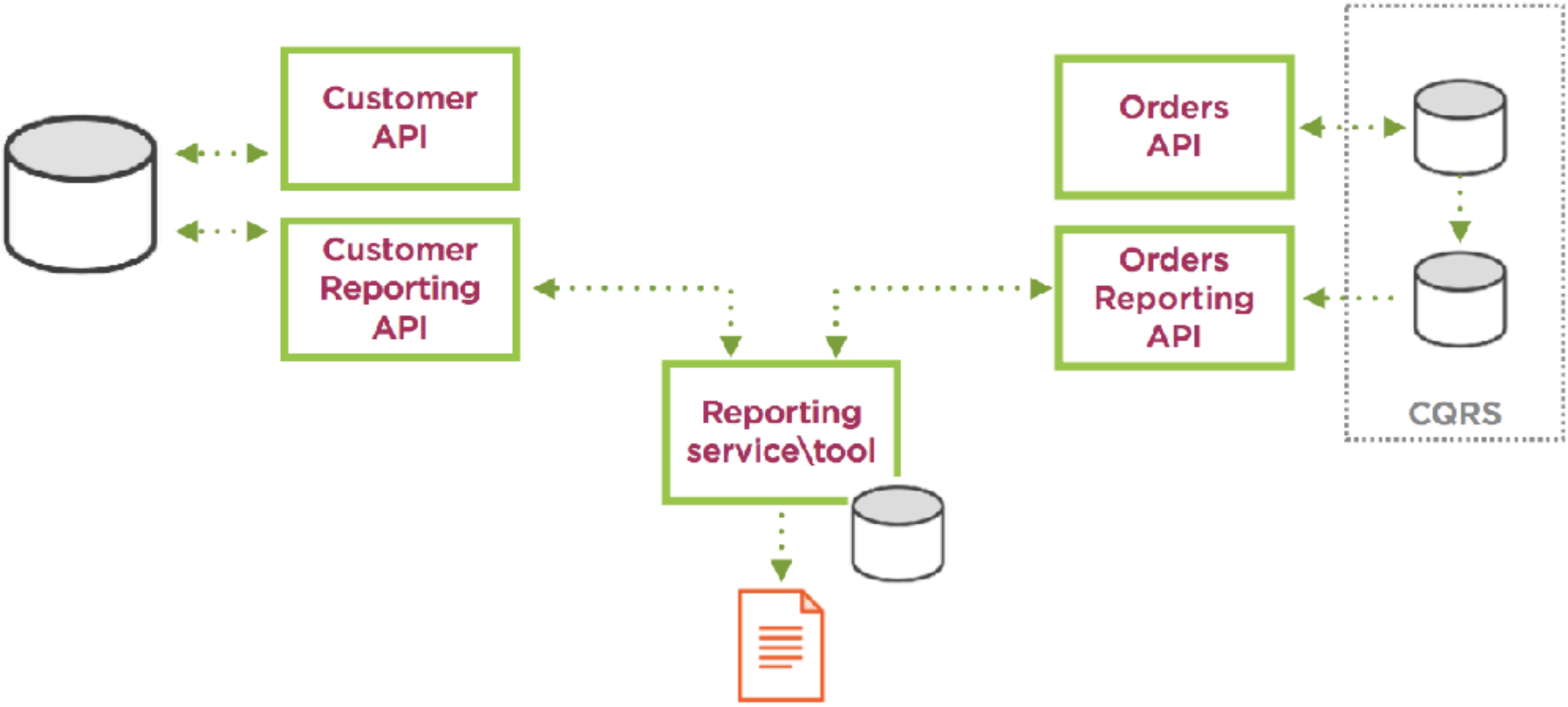
Approach

- Modules become services
- Shareable code libraries promote to service
- Review micro-services principles at each stage
- Prioritize by
  - Minimal viable product (MVP)
  - Customer needs and demand

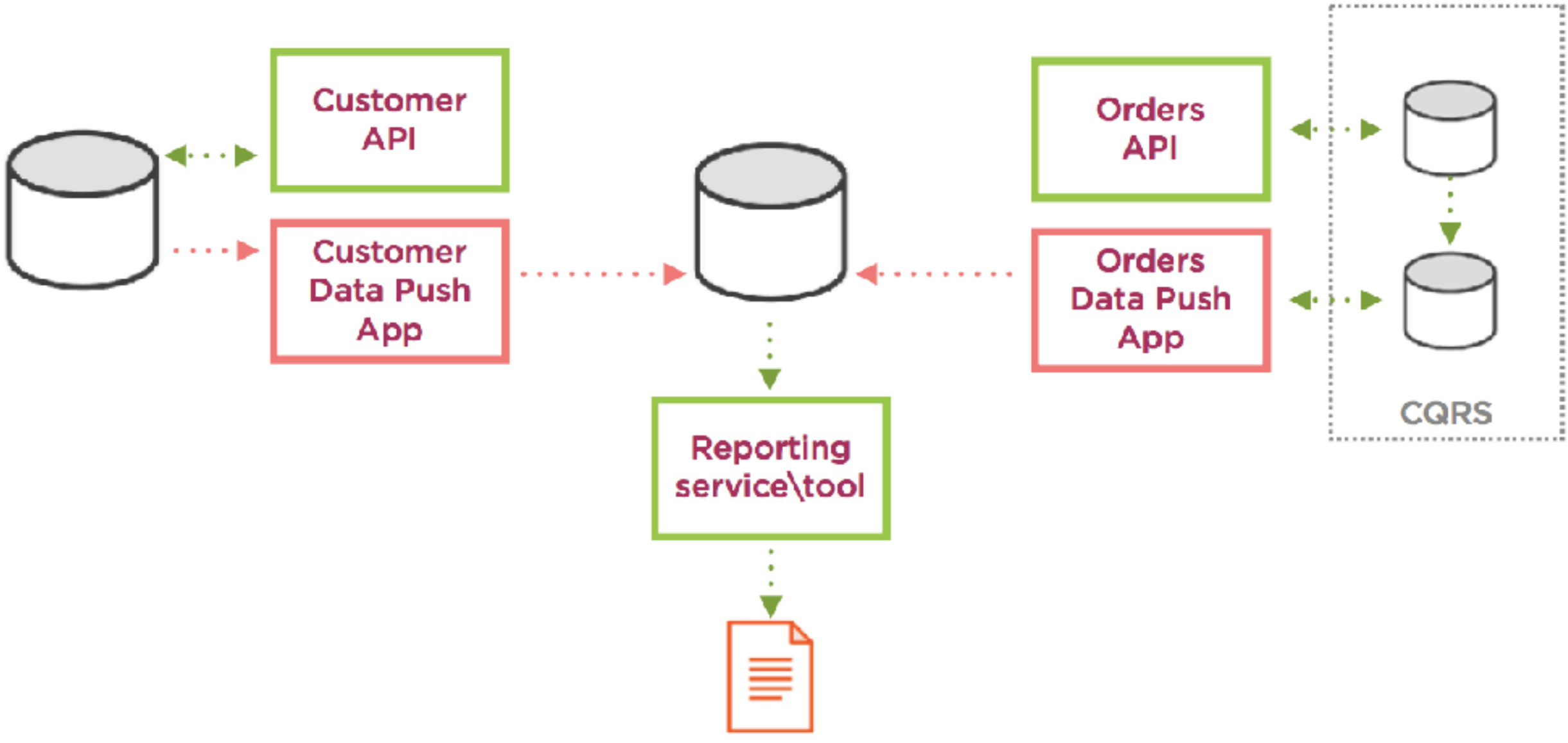
Solutions for Centralize Reporting

- Reporting service calls
- Data push application
- Reporting event subscribers
- Reporting events via gateway
- Using backup imports for reporting
- ETL and data-warehouses

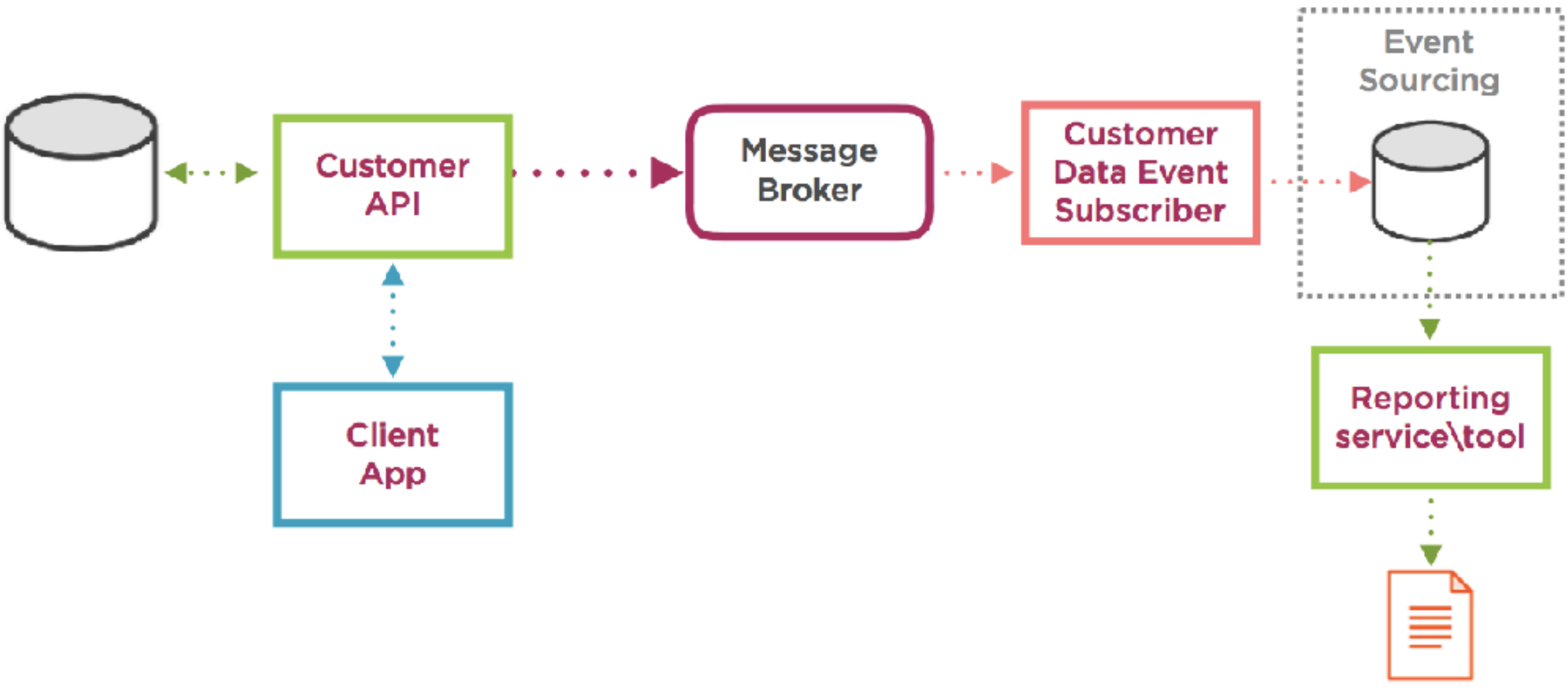
Reporting Service Calls



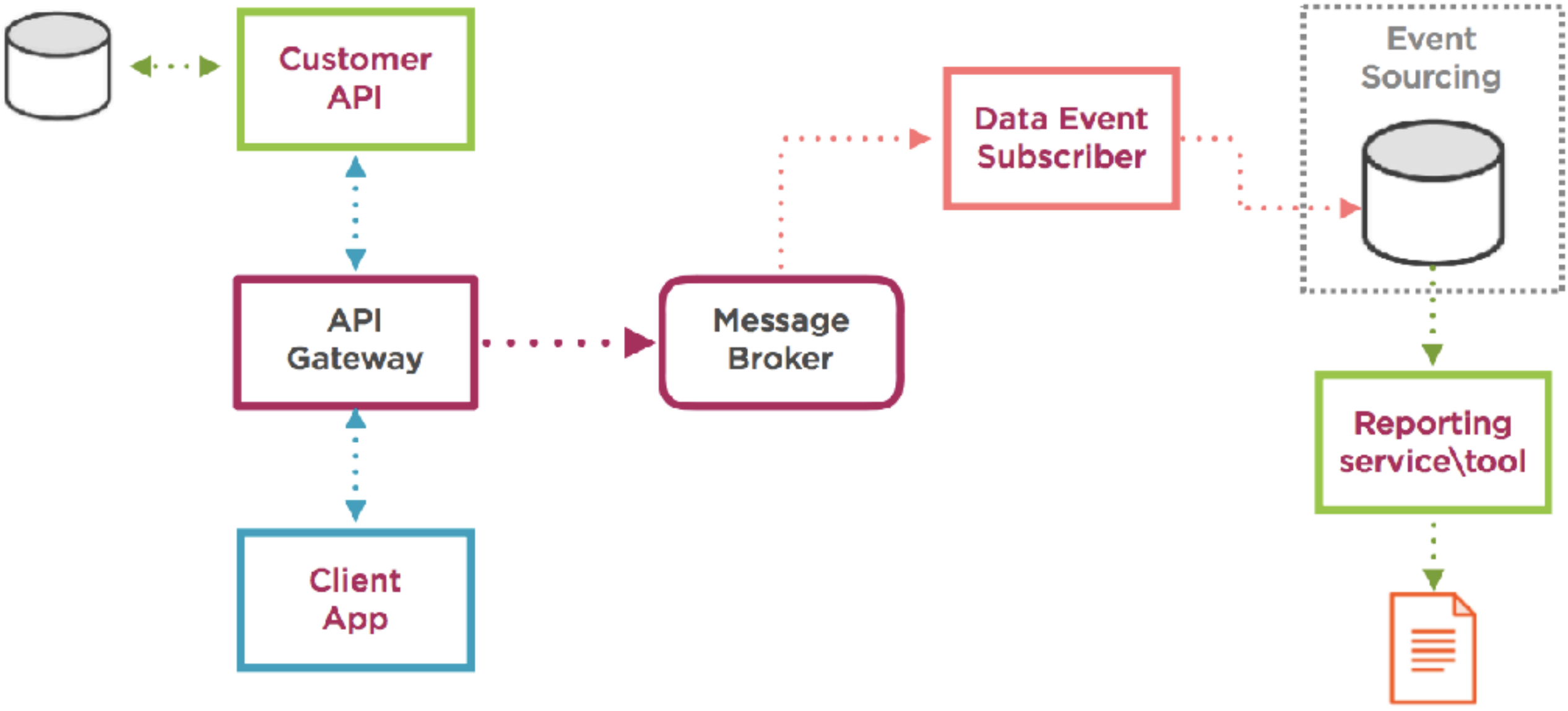
Data Push Application



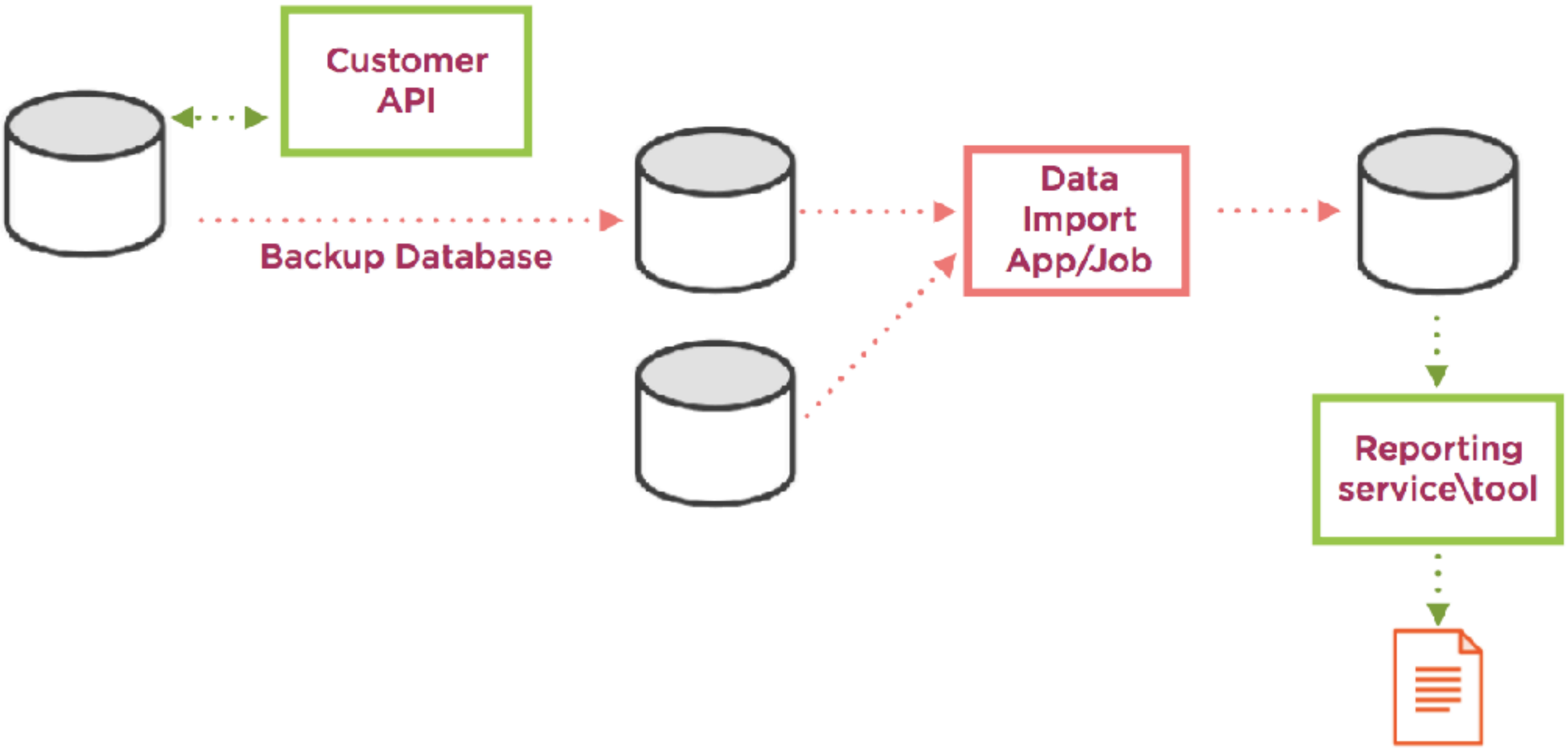
Reporting Event Subscribers



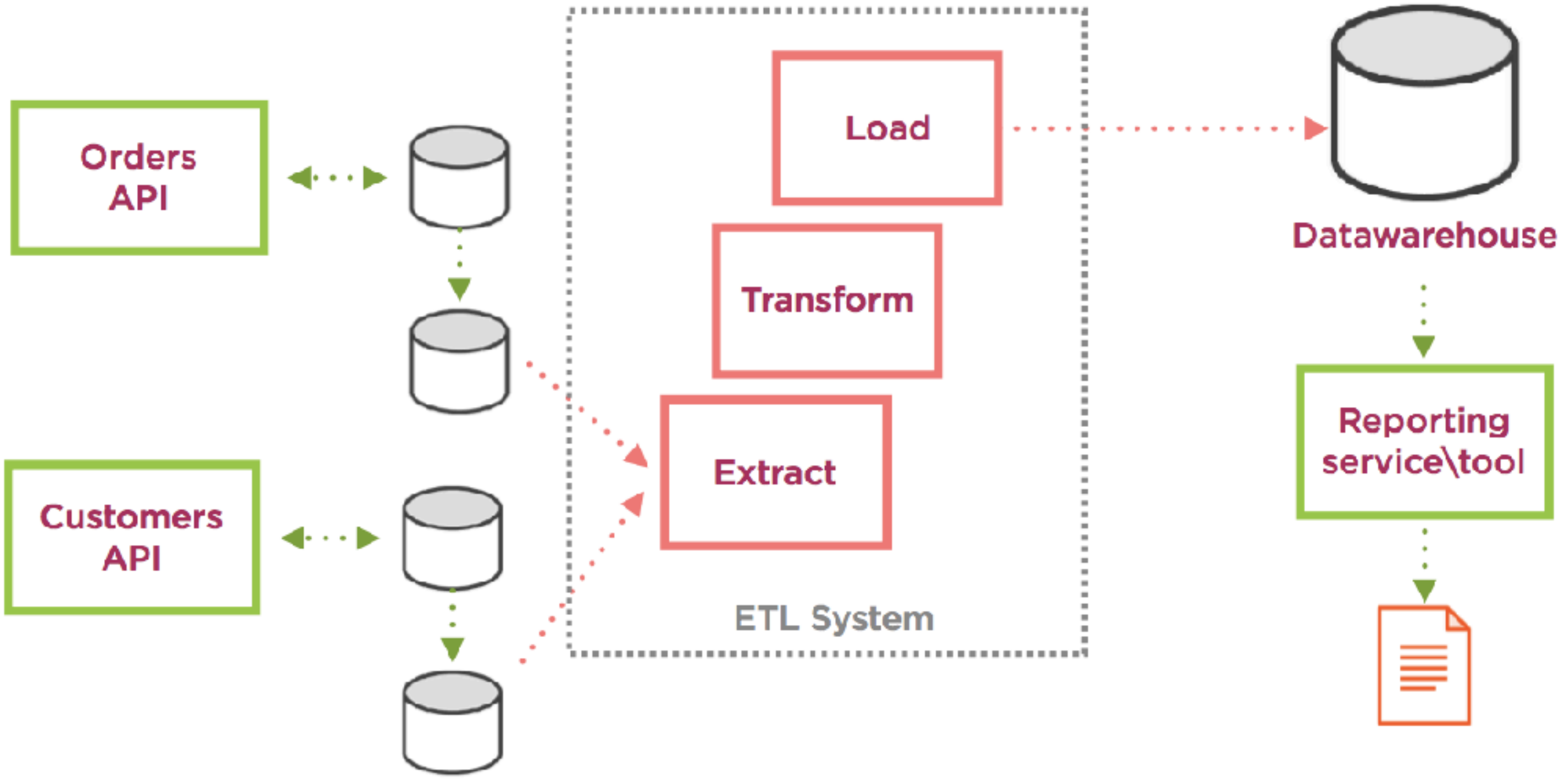
Reporting Event via Gateway



Using Backup Imports



ETL and Data-warehouse



MicroServices Provisos

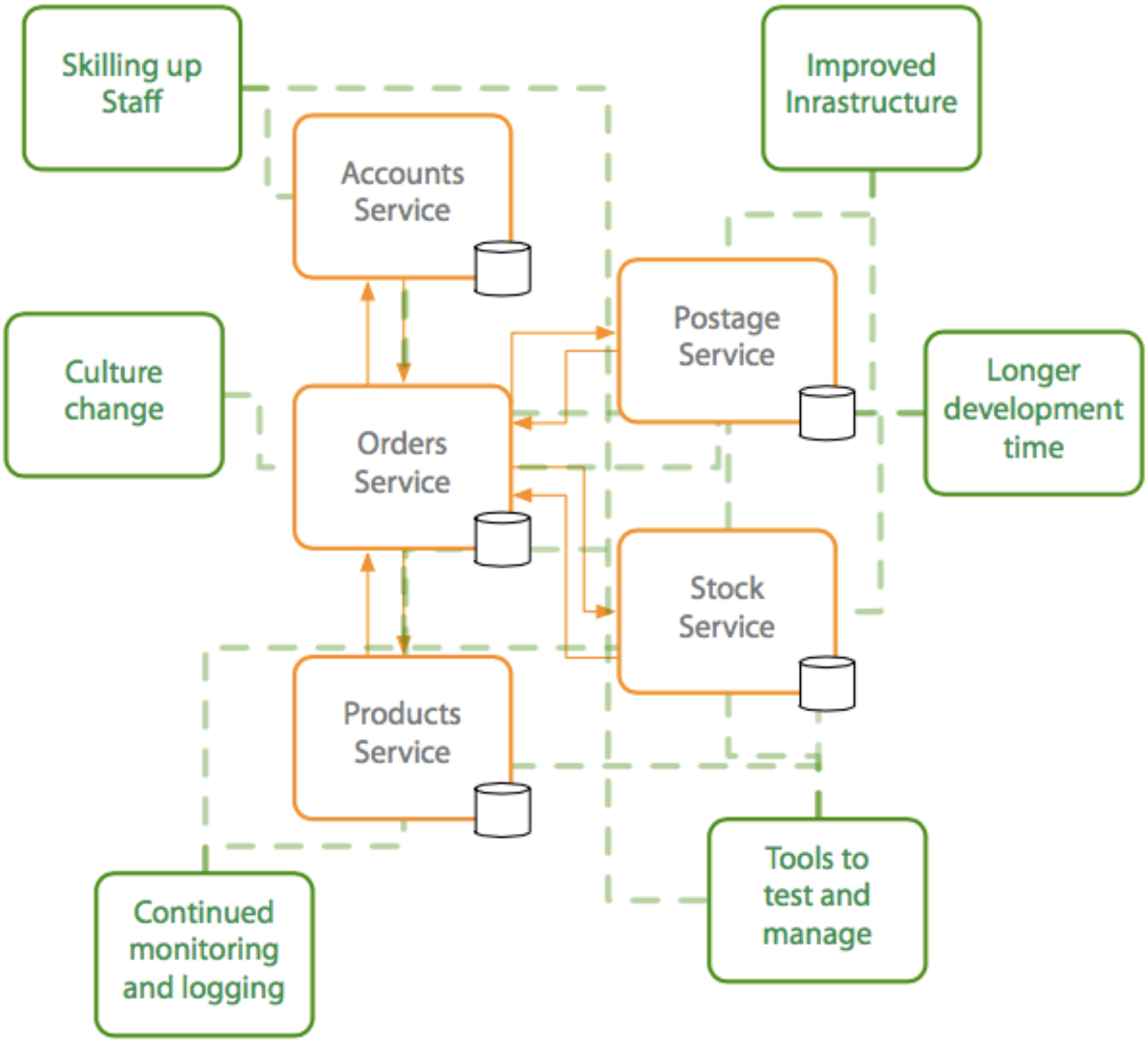
Provisos

- Accepting initial expense
  - Longer development times
  - Cost and training for tools and new skills
- Skilling up for distributed systems
  - Handling distributed transactions
  - Handling reporting
- Additional testing resource
  - Latency and performance testing
  - Testing for resilience

Provisos

- Improving infrastructure
  - Security
  - Performance
  - Reliance
- Overhead to mange microservices
- Cloud technologies
- Culture change

Provisos


Skilling up Staff
Improved Inrastructure
Accounts Service
Postage Service
Culture change
Longer development time
Orders Service
Stock Service
Products Service
Tools to test and manage
Continued monitoring and logging

Provisos

Business

Technical

Production

Business	Technical	Production
Agile and DevOps	Technical diversity	Infrastructure
Cost	Performance	Portability
Time to market	Maintainability	Scalability
Open source	Extensibility	Availability
Single support	Testing	Continuous Integration
Standard	Design	Monitoring
	Data store	Environments
	Distribution	

How to Scope Microservices using Bounded Contexts

How to Architect API based Microservices

How to Achieve Data Consistency Across Microservices

How to Make Microservices Backwards Compatible

How to Split Monolithic Databases Across Microservices

How to Implement Microservices Centralized Logging

How to Automate the Deployment of Your Microservices Architecture

How to Manage Microservices Configuration

How to Monitor Microservices

How to Architect Asynchronous Microservices

How to Compose Microservices Together

How to Centralize Access to Microservices using an API Gateway

How to Make Microservices More Resilient

How to Define and Document Microservice Contracts

How to Provide Reporting From Distributed Microservices Data

How to Approach Cloud Based Microservices Infrastructure

How to Manage Microservices Registration and Discovery

Q&A