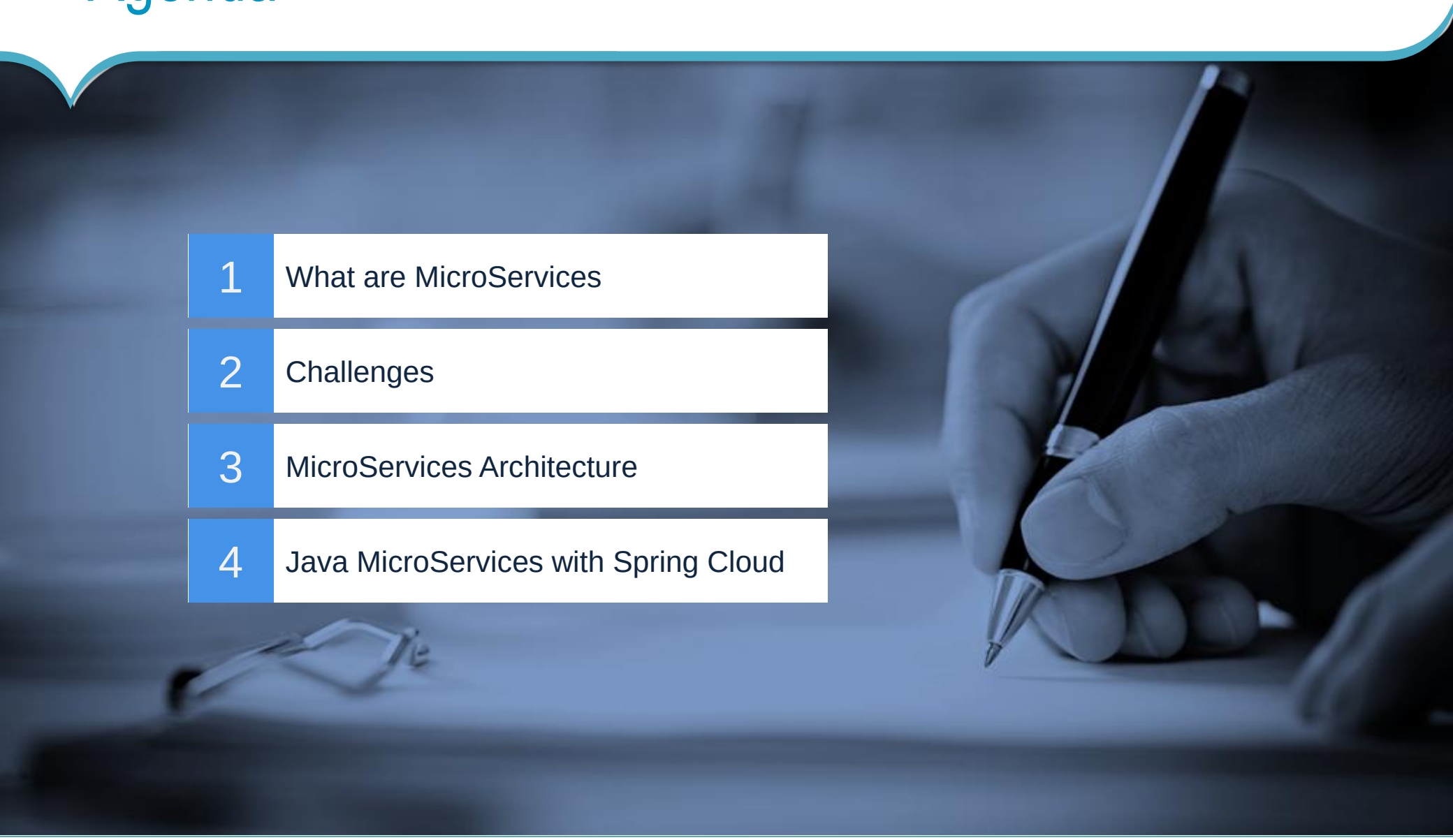
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

MicroServices

2017

People matter, results count.

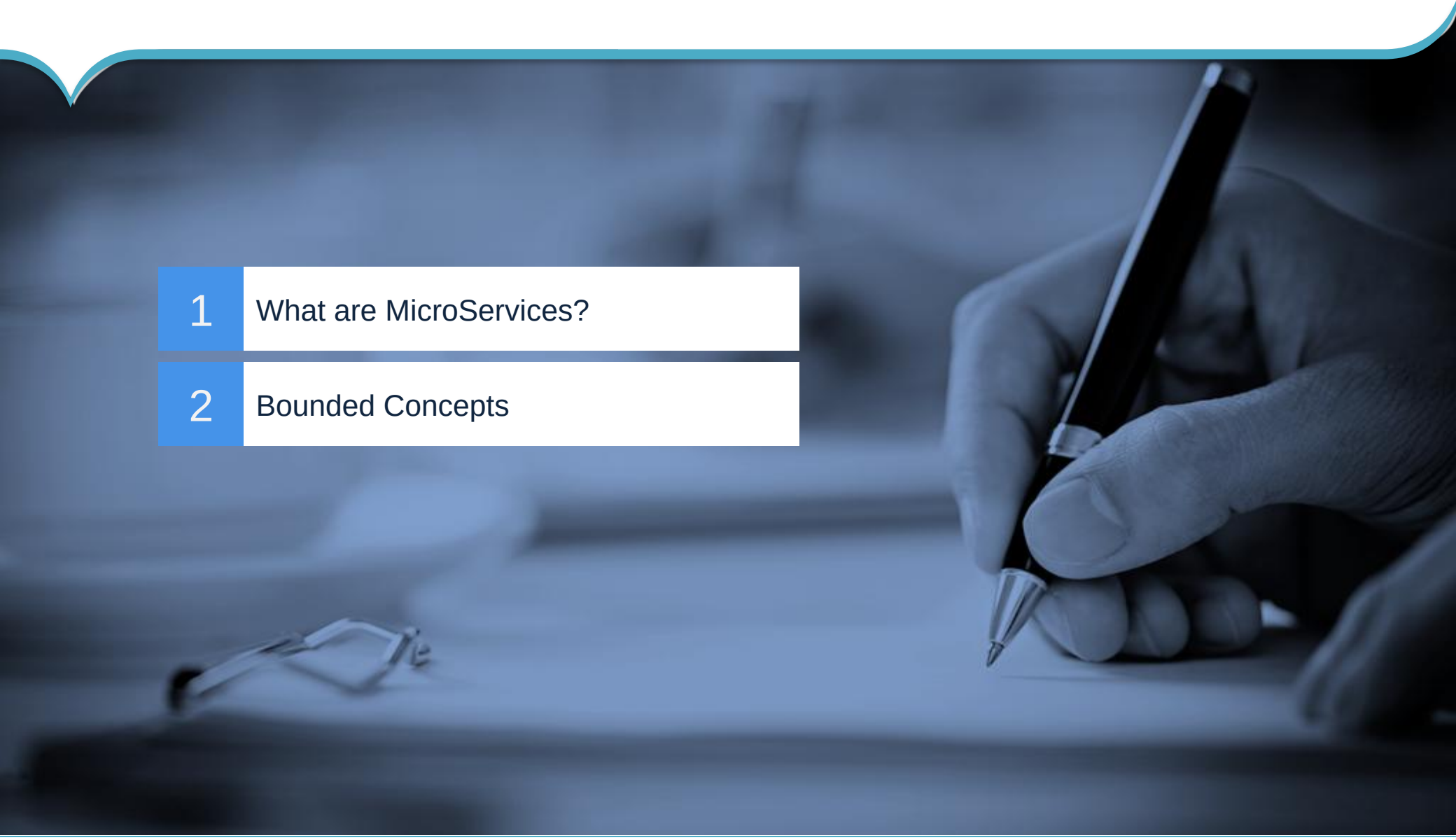
Agenda



1. What are MicroServices
2. Challenges
3. MicroServices Architecture
4. Java MicroServices with Spring Cloud

What are MicroServices

Outline



1. What are MicroServices?
2. Bounded Concepts

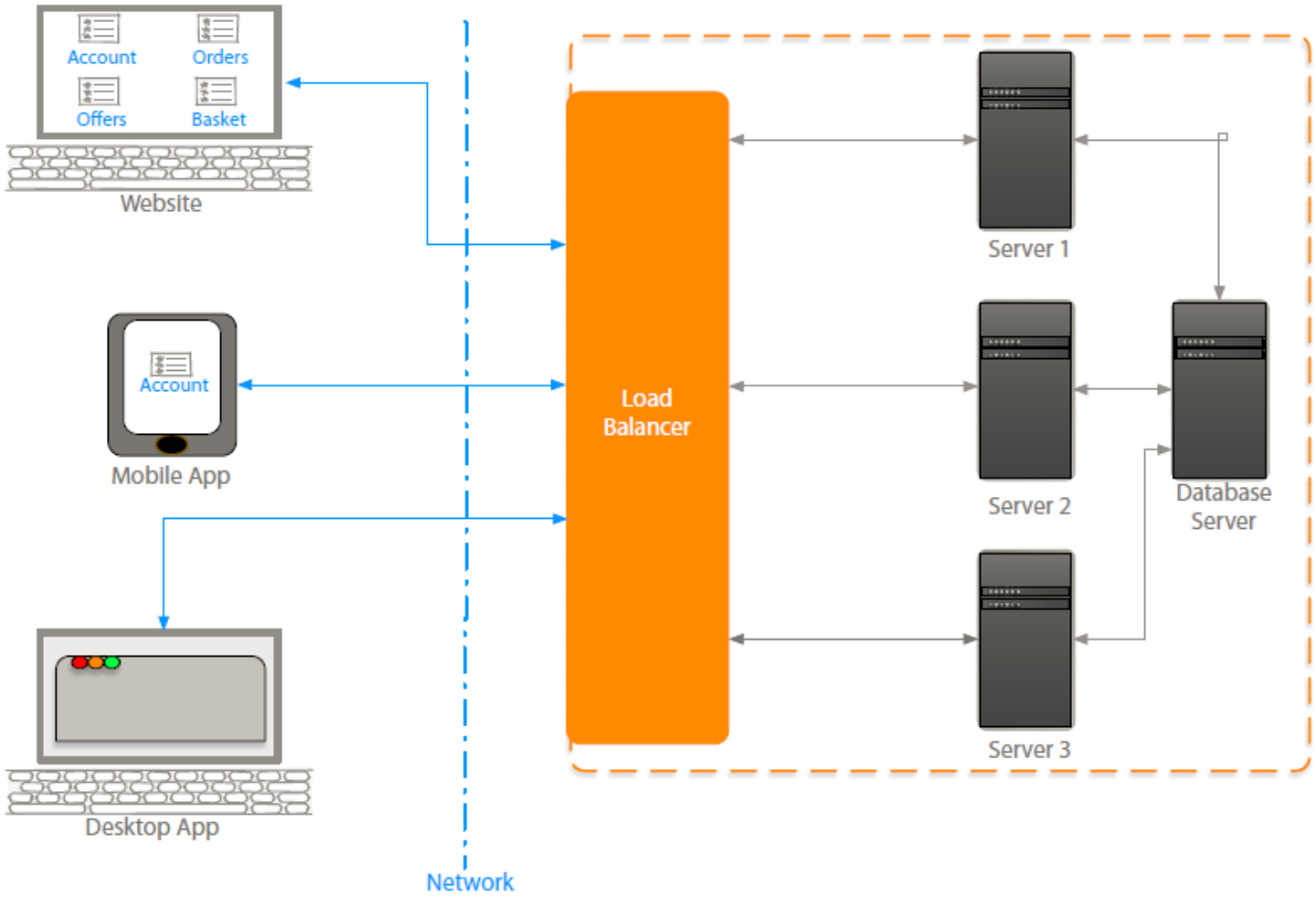
What are MicroServices?

- Loosely Coupled, Service Oriented Architecture (SOA) with Bounded Contexts
  - *Adrian Cockcroft(Netflix)*

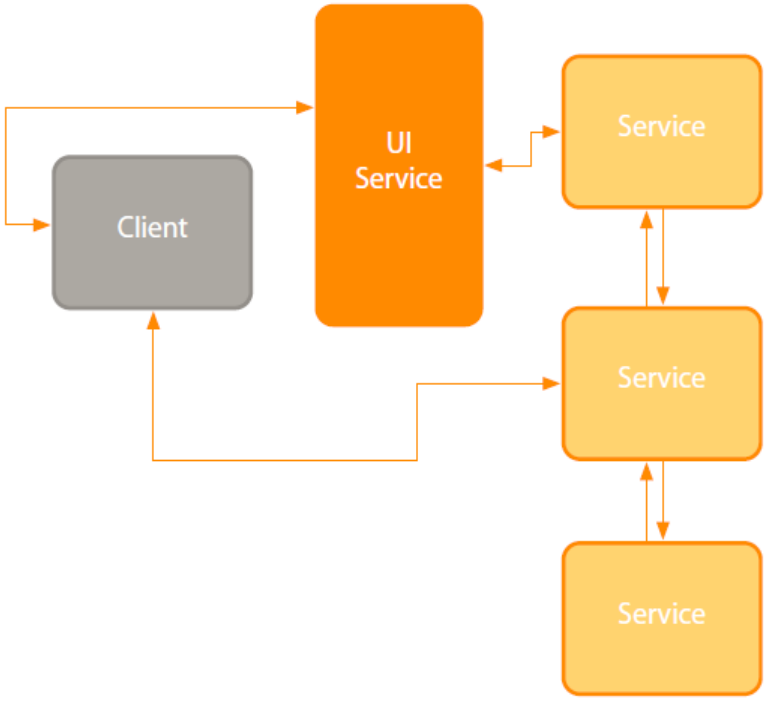
- Familiar concepts, reimagined
  - Loosely coupled
    - Deploy any time, no dependencies on anything else
    - SOA with an ESB is not loosely coupled

- Service Oriented Architecture
  - Inherently distributed system
  - But using simpler components (Services)
  - Designed that way, not a reuse mechanism for Silos

What is a Service?

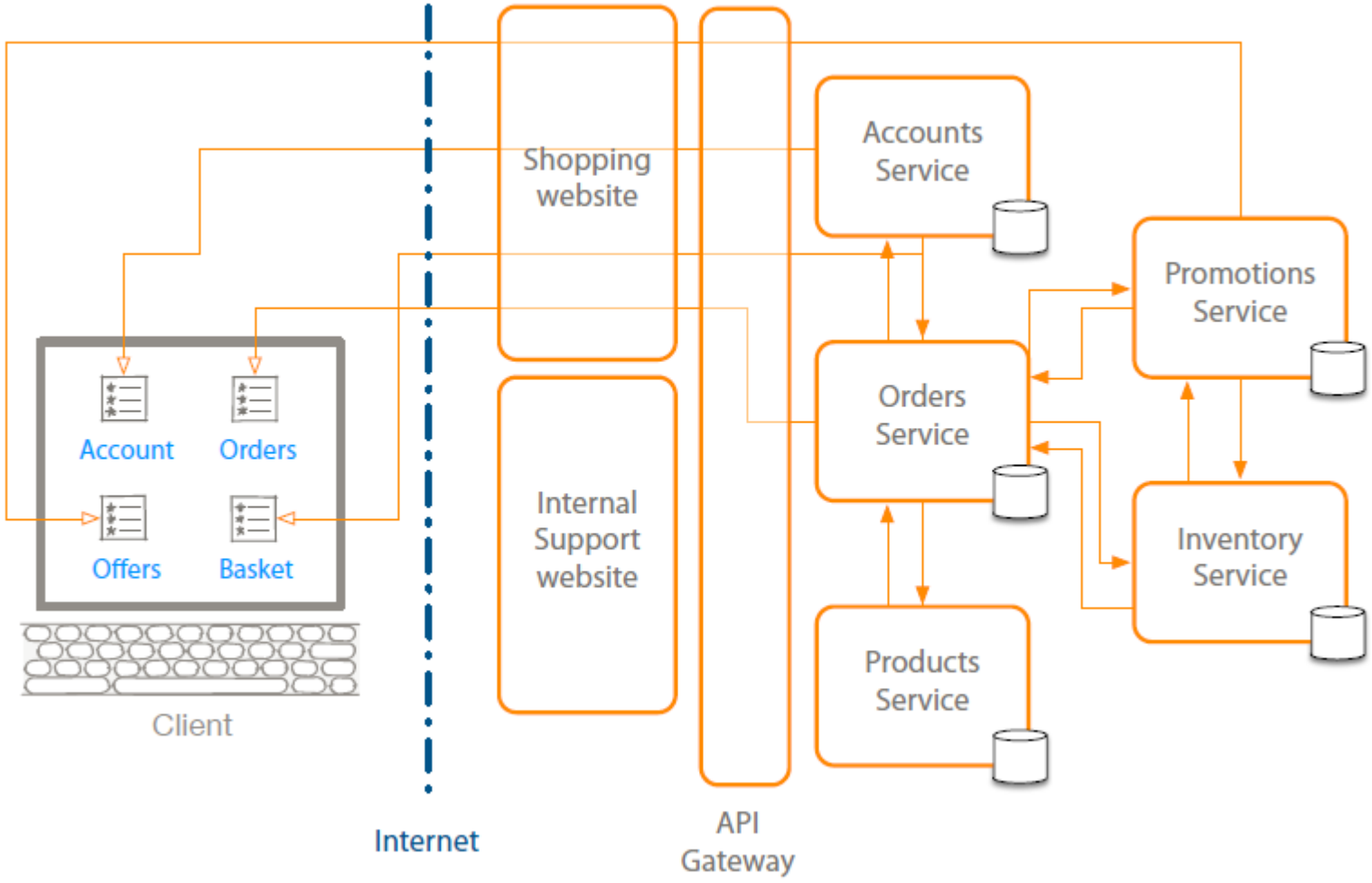


What is a Service?



- SOA done well
  - Knowing how to size a service
  - Traditional SOA resulted in monolithic services
- Micro sized services provide
  - Efficiently scalable applications
  - Flexible applications
  - High performance applications
- 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
- Independently changeable
- Independently deployable
- Distributed transactions
- Centralized tooling for management

MicroServices



Bounded Concepts

- From Eric Evan's "Domain-Driven Design" book
  - The setting in which a word or a statement appears that determines its meaning

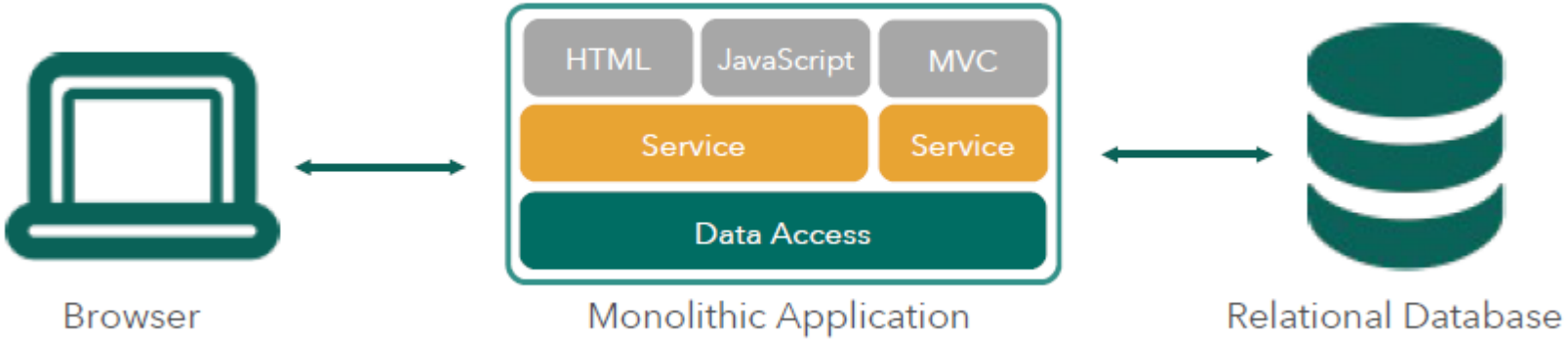
- Given a central concept, each use is a separate context
  - Example: "reservation" in an airline booking system
- Difficult in an single "monolithic" application
  - Easier with microservices
    - Each can implement the same concept to suit their use of it
    - Each is free to have its own independent representation
- A self-consistent subset of the domain used by a micro-service

Three Tenets of MicroServices

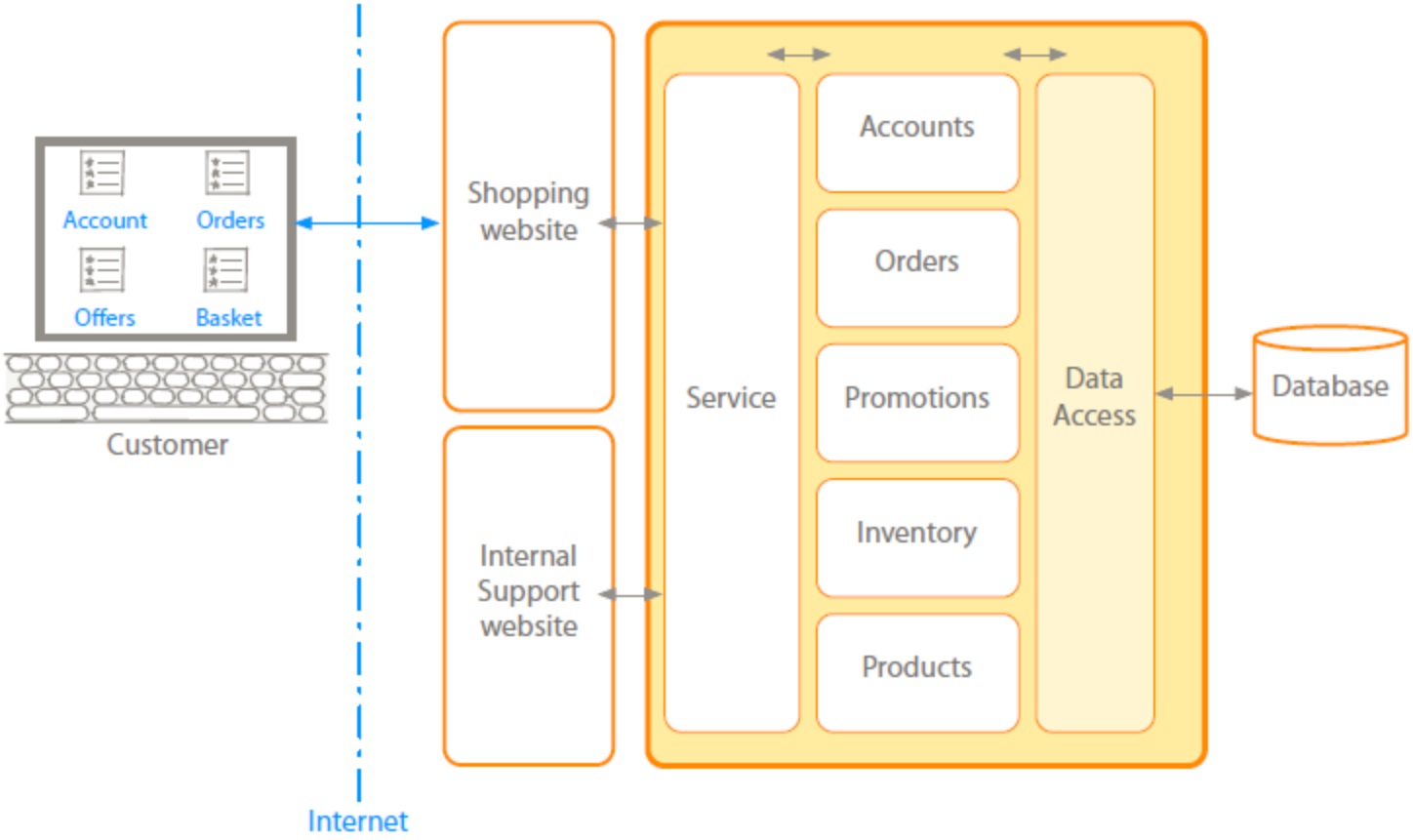
- Gary Ollifee, research director at Gartner
  - Consuming services separate from provisioning services
  - Separating infrastructure management from the delivery of the application capability
    - Using a Paas like Cloud Foundry
  - Separating teams and decoupling services
    - Each can be built, enhanced and deployed separately
    - Embrace Dev Ops to do this successfully

Monolithic Architecture

- How apps have traditionally been developed
  - Large, involved code-base
  - Infrequent updates
  - Risky to make small changes



MicroServices: Monolithic

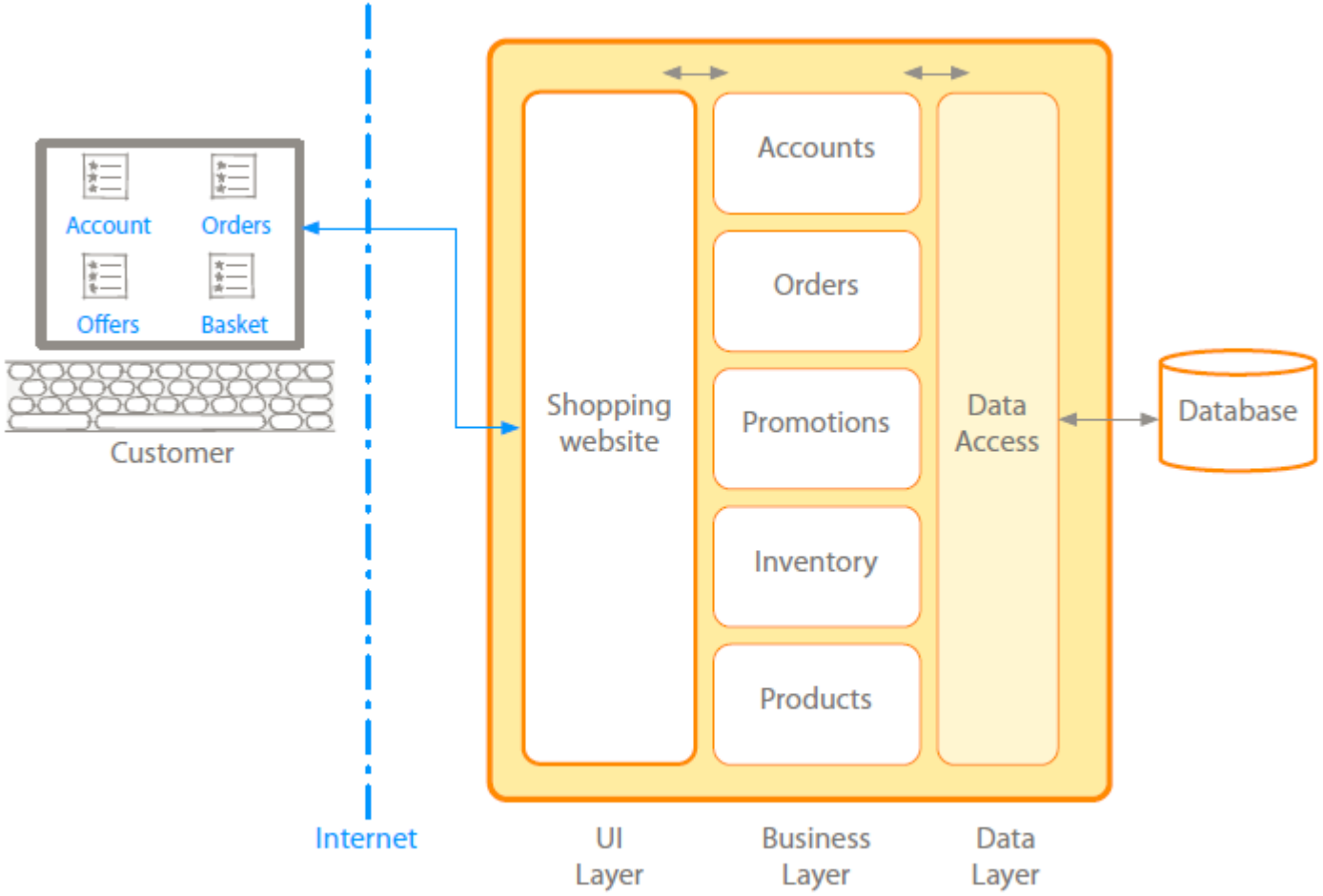


MicroServices: Monolithic

Typical enterprise application

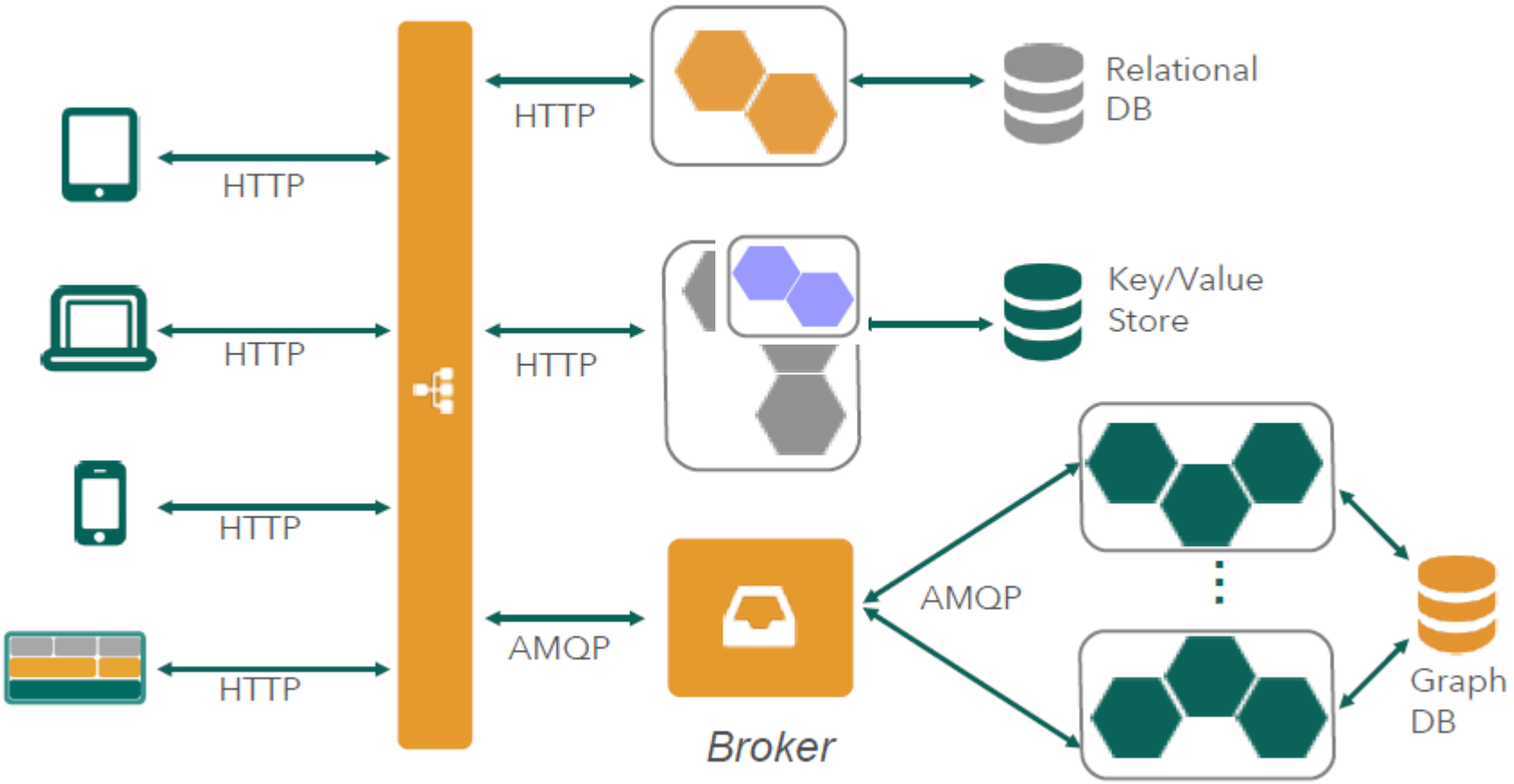
- No restriction on size
- Large codebase
- Longer development times
- Challenging deployment
- Inaccessible features
- Fixed technology stack
- High levels of coupling
  - Between modules
  - Between services
- Failure could affect whole system
- Scaling requires duplication of the whole
- Single service on server
- Minor change could result in complete rebuild
- Easy to replicate environment

MicroServices: Monolithic



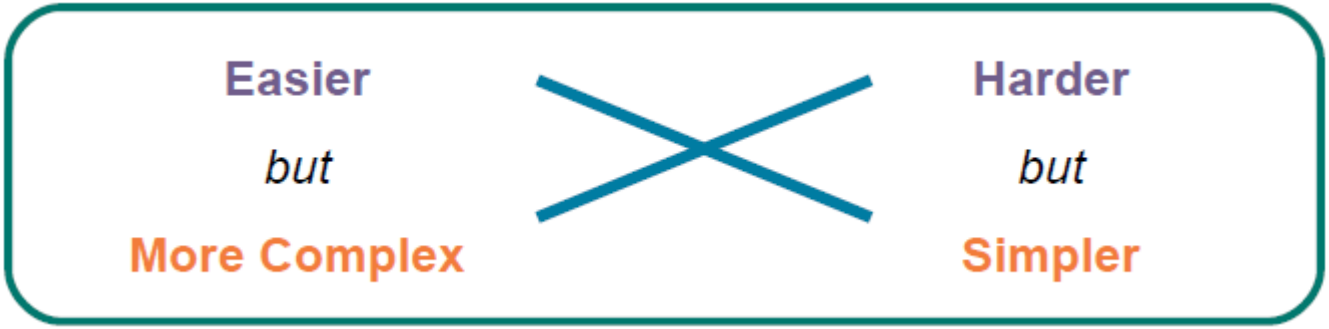
Microservices Architecture

- Decompose into collaborating components



Trade-Off

- **Monolith**
  - Easier to build
  - But ultimately more complex to enhance and maintain
  - Scaling Up (bigger processers) limited
- **Microservices**
  - Harder to build
  - But ultimately simpler to extend, enhance and maintain
  - Scaling Out (more processes) easier



Summary

Bounded Concepts

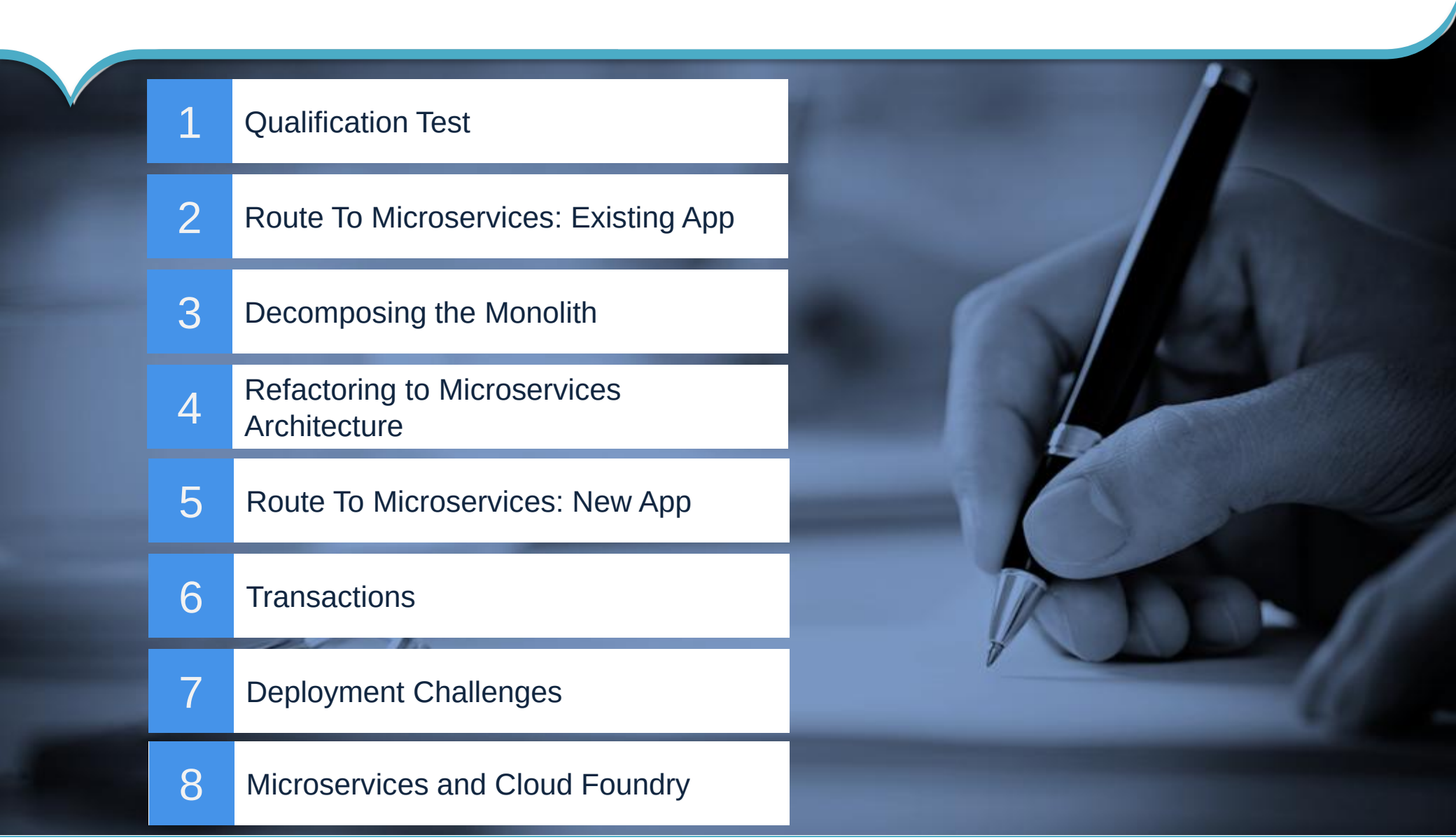
Monolithic Architecture

Microservices Architecture

Trade-Off

Challenges

Outline



1. Qualification Test
2. Route To Microservices: Existing App
3. Decomposing the Monolith
4. Refactoring to Microservices Architecture
5. Route To Microservices: New App
6. Transactions
7. Deployment Challenges
8. Microservices and Cloud Foundry

Qualification Test

- Microservices are not for everyone
  - It's as much how you develop as what you develop

- "You must be this tall" to "ride" Microservices
  - Rapid provisioning
  - Basic monitoring
  - Rapid Application Development
  - Devops culture

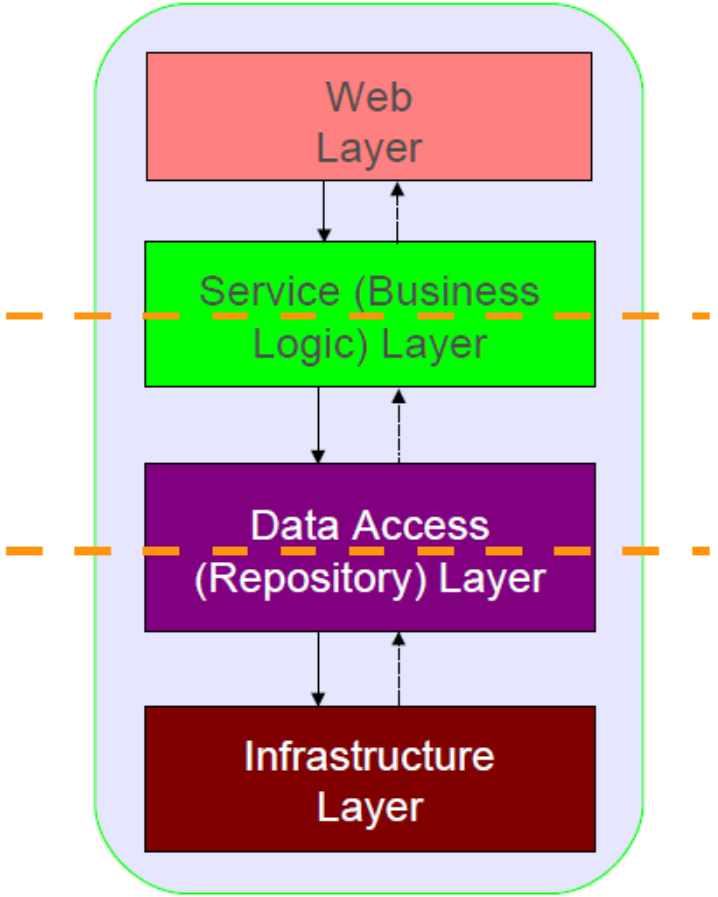
Route To Microservices: Existing App

- Develop new functionality as microservice(S) around existing Monolith
  - Use Facades/Adapters/Translators to integrate them
- Strangle the Monolith
  - Refactor existing monolith functionality into new microservice(s)
  - Long-term evolution:
    - Monolith wither to nothing
    - Or is reduced to a solid, reliable core that is not worth refactoring (becuase we know it works)

Decomposing the Monolith

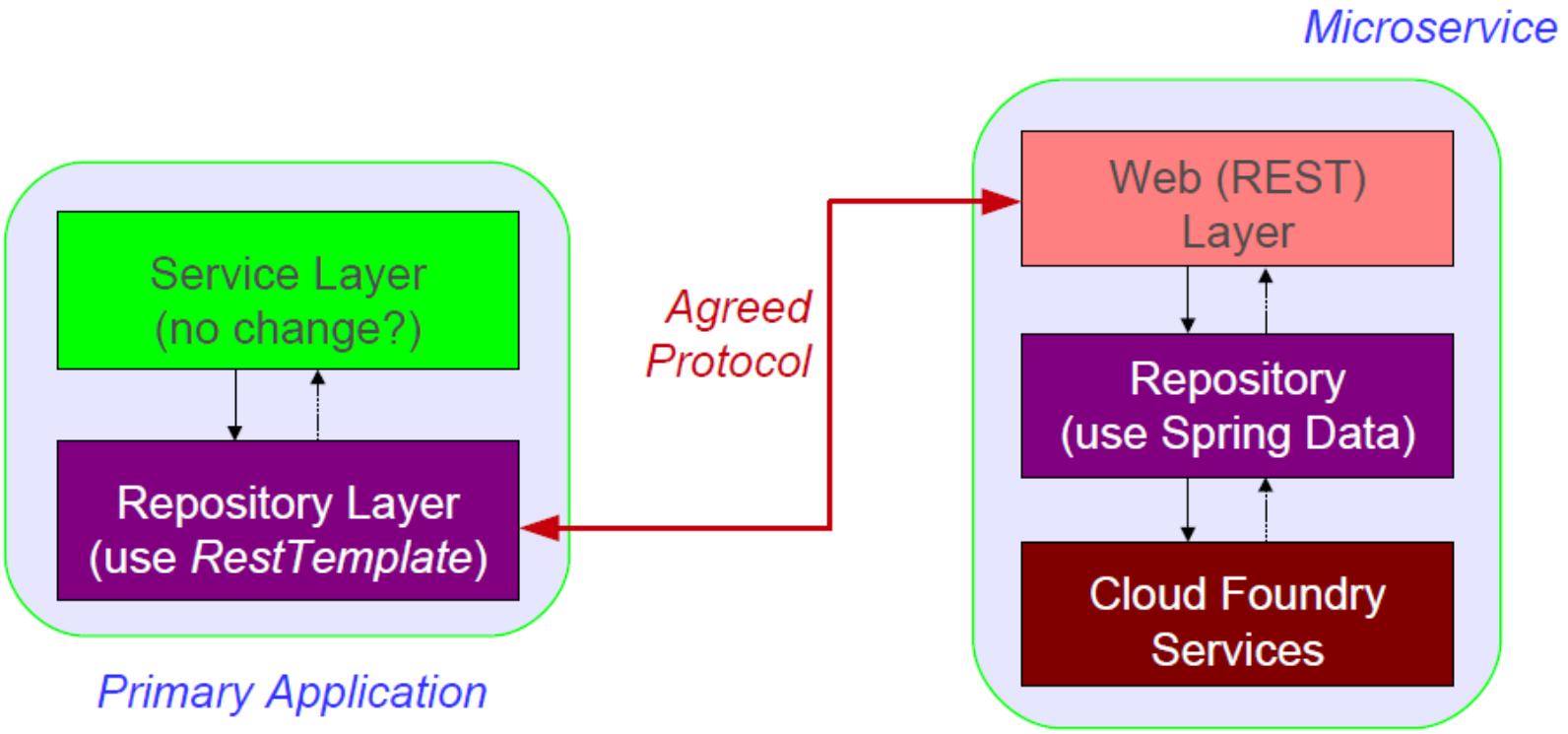
- Many Java applications use the classic three layer architecture
  - Services (business logic)
  - Repositories (data access)
  - Infrastructure (interface to external resources)
  - Web-layer(optional), other interfaces possible

- Refactor into two processes



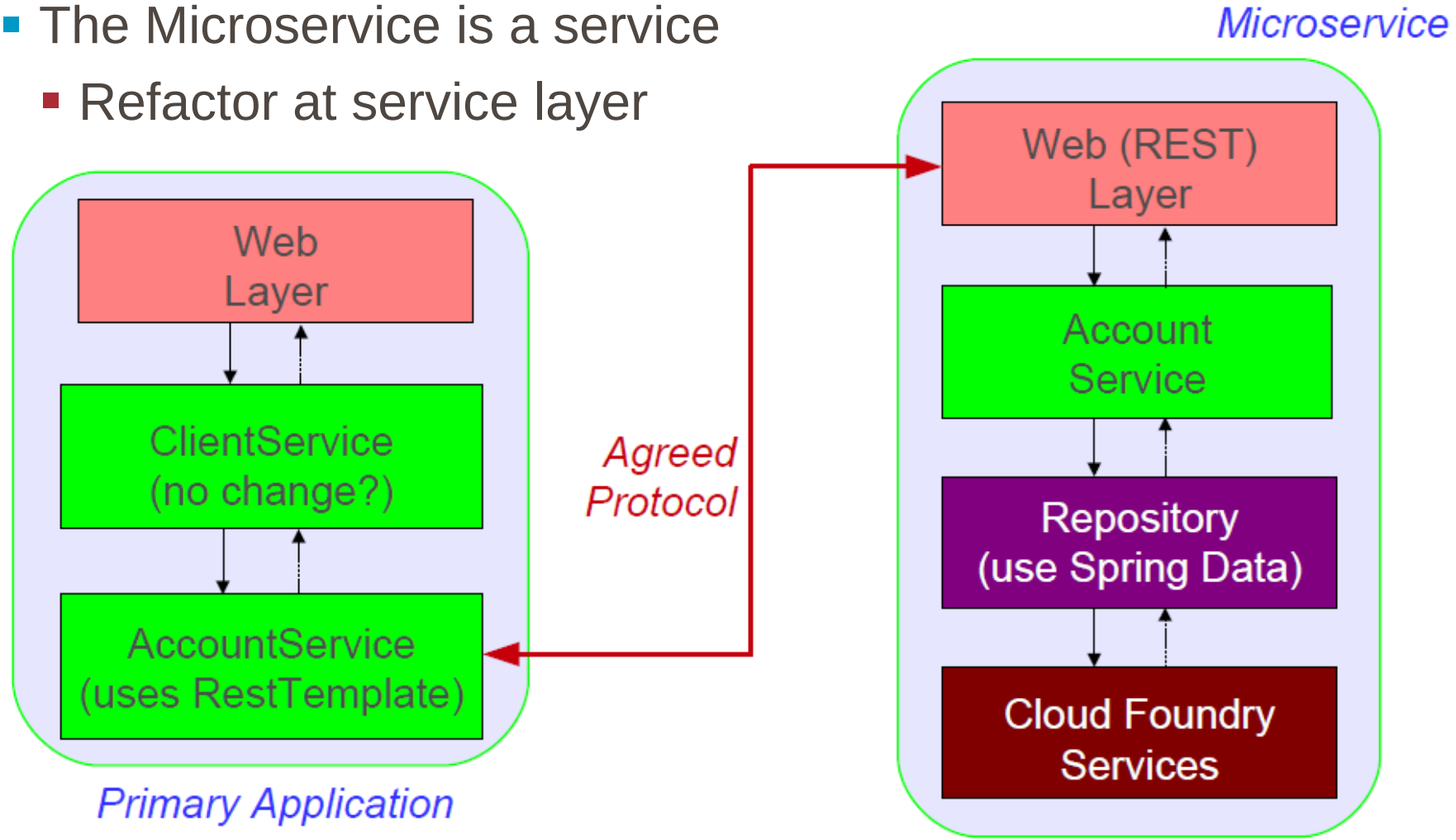
Refactoring to Microservices Architecture

- Refactor the repository to talk to the microservice
  - Any protocol you like, here using REST
  - Microservices talks to CF Services



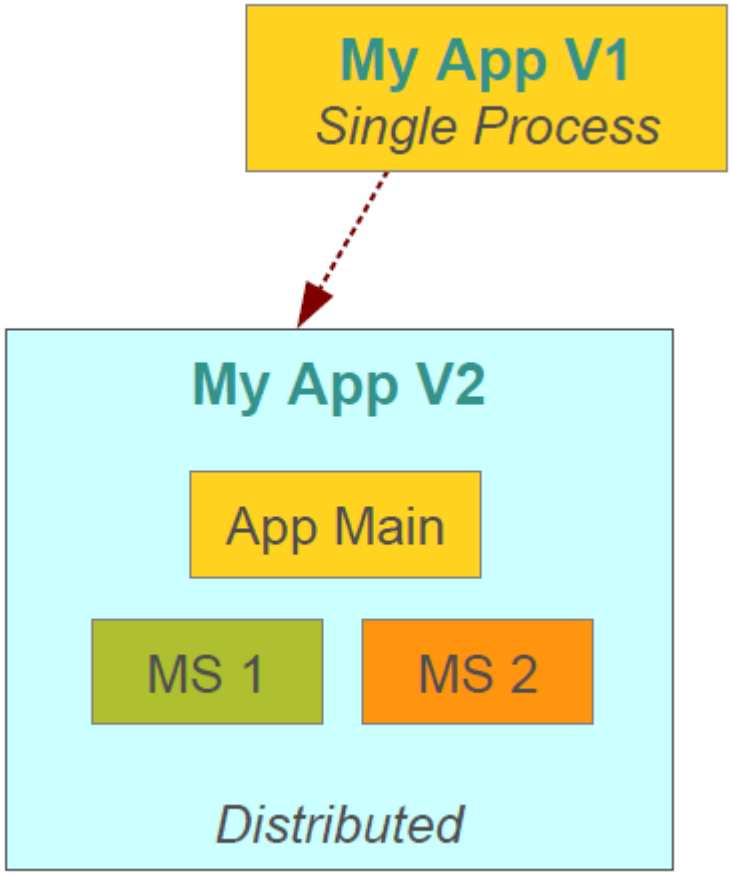
Refactoring to Microservices Architecture

- The Microservice is a service
  - Refactor at service layer



Route To Microservices: New App

- Start with a "Monolith"
  - Keep it simple, at first
  - Single process application
  - Apply 12-factor patterns
  - Cloud-ready even at this stage
- As it grows
  - Decompose into micro-services
  - Enables separately manageable and deployable units
  - Each can use own storage solution (polyglot persistence)



Transactions - I

- What happens if I need to co-ordinate a change to multiple microservices at the same time?
  - Transactions help with consistency, but force significant temporal coupling
  - Distributed transactions are notoriously difficult to implement

- Microservice architecture emphasize transaction less coordination between services,
  - Eventual consistency is all you can guarantee
  - Problems are dealt with by "compensating" operations.
    - A transaction to undo a previous transaction

Transactions - II

- Many business already do this
  - But it's new to developers

- Business may choose to handle a degree of inconsistency in order to respond quickly to demand
  - Define a reversal process to deal with mistakes
  - Trade-Off is worth it if cost of fixing mistakes is less than the cost of lost business

Deployment Challenges - I

- Inherently more complex, distributed architecture
- We need to support
  - Configuration management
  - Service registration and discovery
  - Routing and load balancing
  - Fault tolerance
  - Monitoring the individual components
  - And also need a global/consolidated view

Deployment Challenges - II

- No microservice is an island- Dr Dave Syer (Pivotal)
  - Must be part of an "archipelago"

- How to handle a whole (composite) system?
  - The "Big A" app
  - CF manifest does some of this, but its static
  - Static vs. dynamic - need "BOSH for microservices" = PCF
  - Decentralized, autonomous capability required
    - Different teams can deploy at any time
    - You own it, you write it, you run it!

Microservices and Cloud Foundry

- what does a microservice application require?
  - Environment provisioning
  - On-demand scaling
  - Failover and resilience
  - Routing and Load balancing
  - Data services ops (BOSH)

- Cloud Foundry gives you all these
  - Don't have to deploy to a Paas, but it works well
  - A naturally symbiotic relationship

Summary

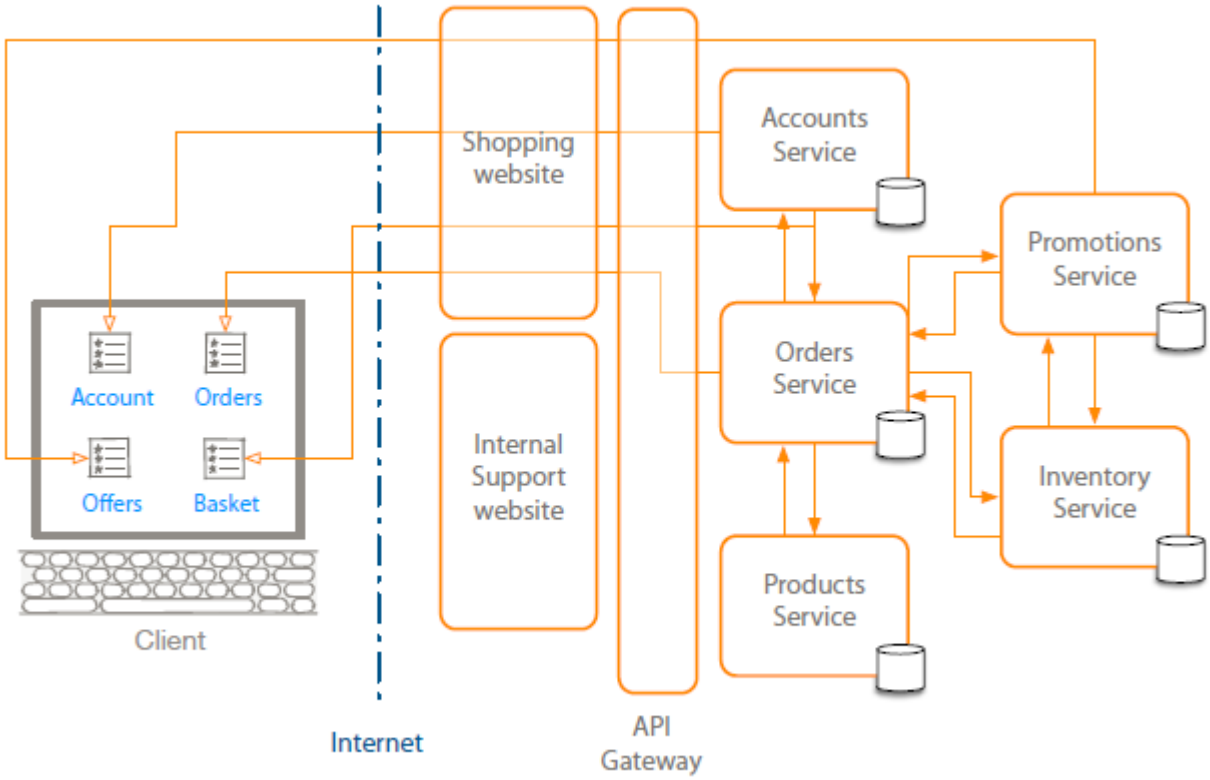
- **Refactoring to Microservices Architecture**
- **Route to Microservices – New App**
- **Transactions.**
- **Deployment Challenges**
- **Microservices and Cloud Foundry.**

MicroServices Architecture

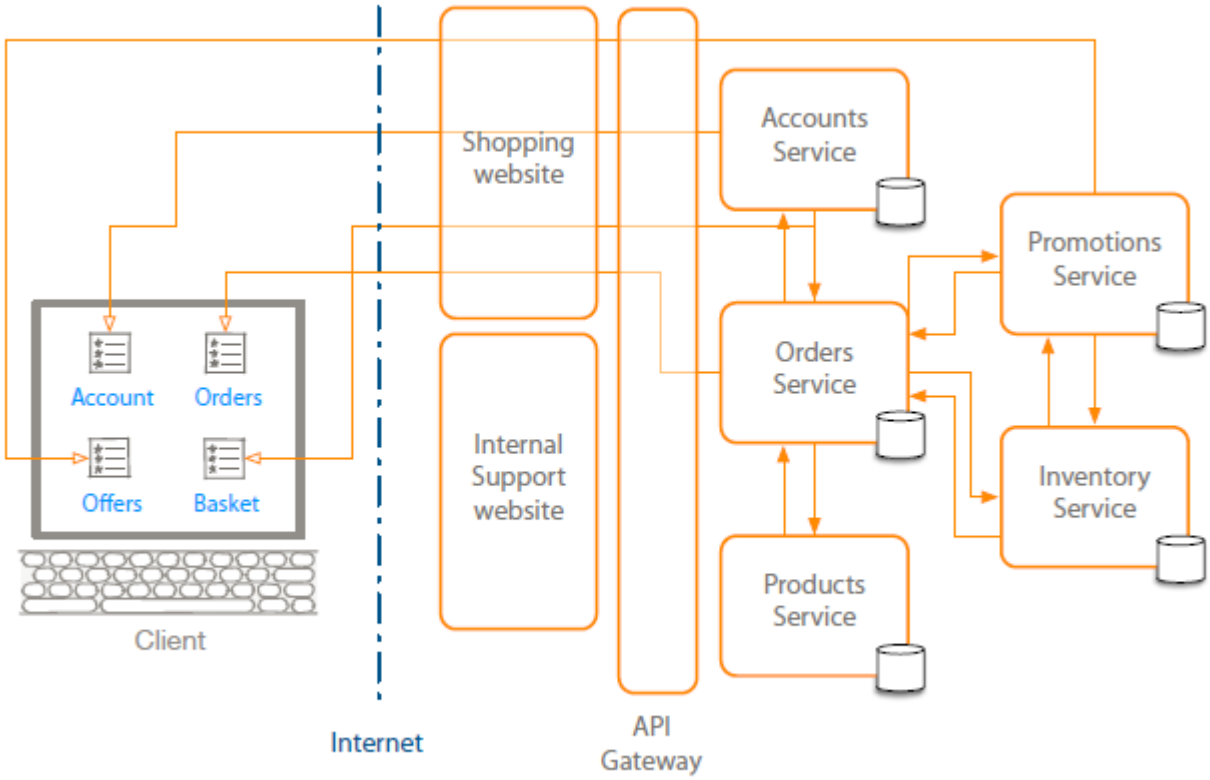
Outline

1. Emergence of MicroServices
2. Design Principles Of MicroServices
3. Approaches of MicroServices
4. Technology for Microservices
5. Moving Forwards with MicroServices

Emergence of MicroServices



Emergence of MicroServices

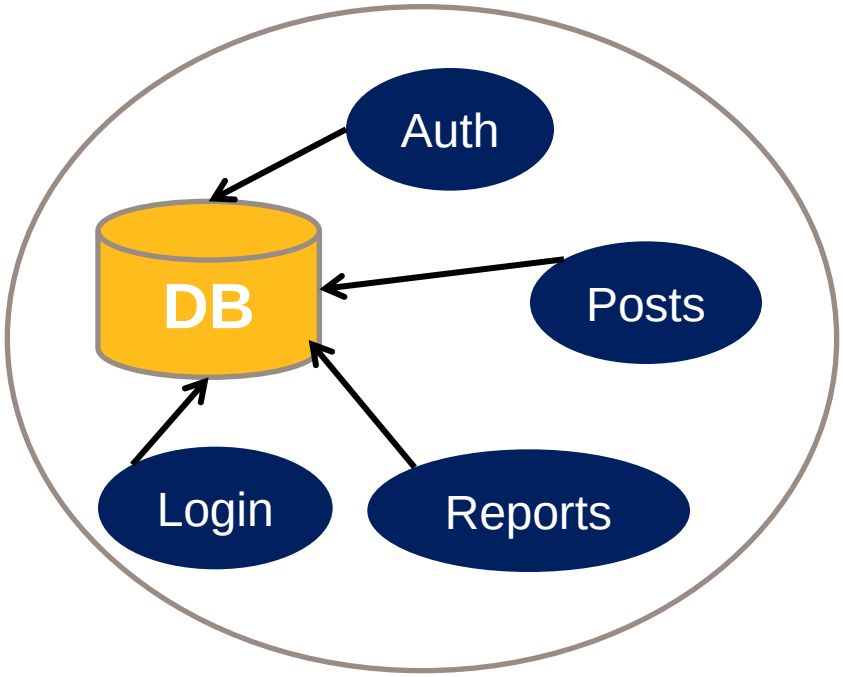


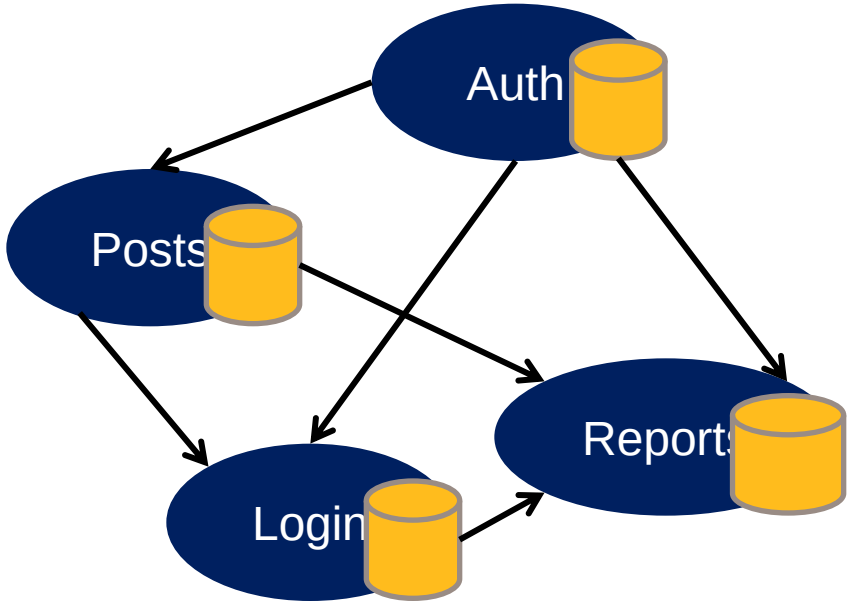
Emergence of MicroServices

Shorter development times

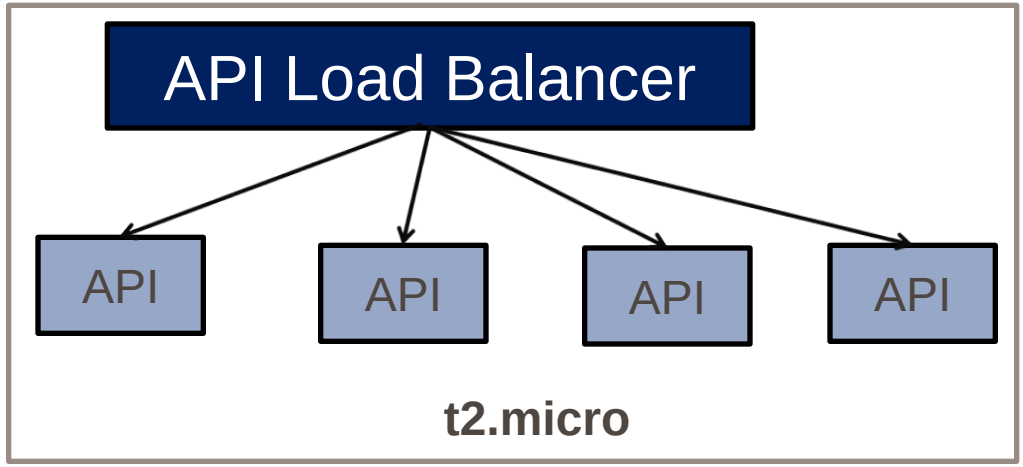
- Reliable and faster deployment
- Enables frequent updates
- Decouple the changeable parts
- Security
- Increased uptime
- Fast issue resolution
- Highly scalable and better performance
- Better ownership and knowledge
- Right technology
- Enables distributed teams

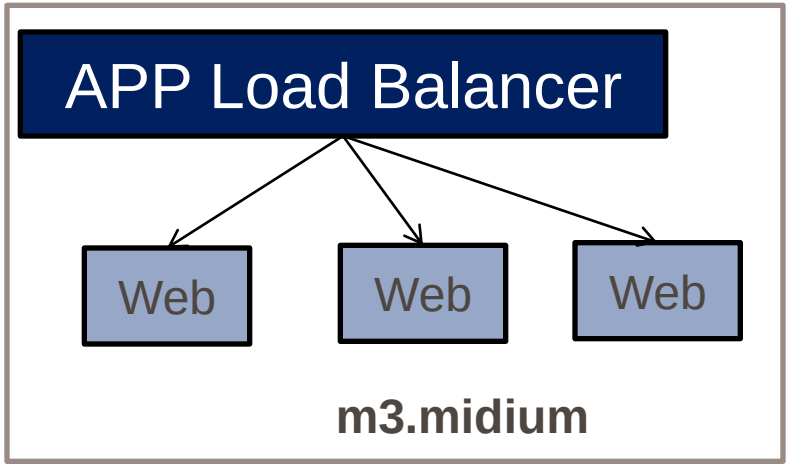
Monolithic Vs MicroServices

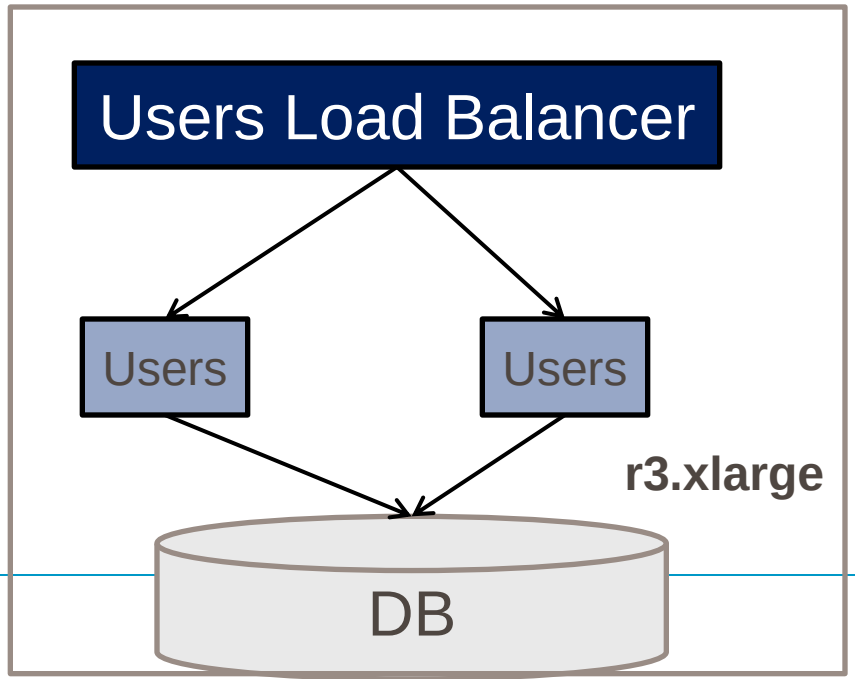




Better Hardware Utilization







MicroServices Vs SOA

- **Microservices**
  - Single distributed System
  - Accelerated Realization of Benefits
  - Specialization of SOA
  - Application Centric
  - Business Ambitious

- **SOA**
  - Large Distributed System
  - Overall Business Transformation
  - Large Scope
  - Spanning Multiple System
  - Enterprise wide changes
  - Business Goals

Design Principles Of MicroServices

High Cohesion

Autonomous

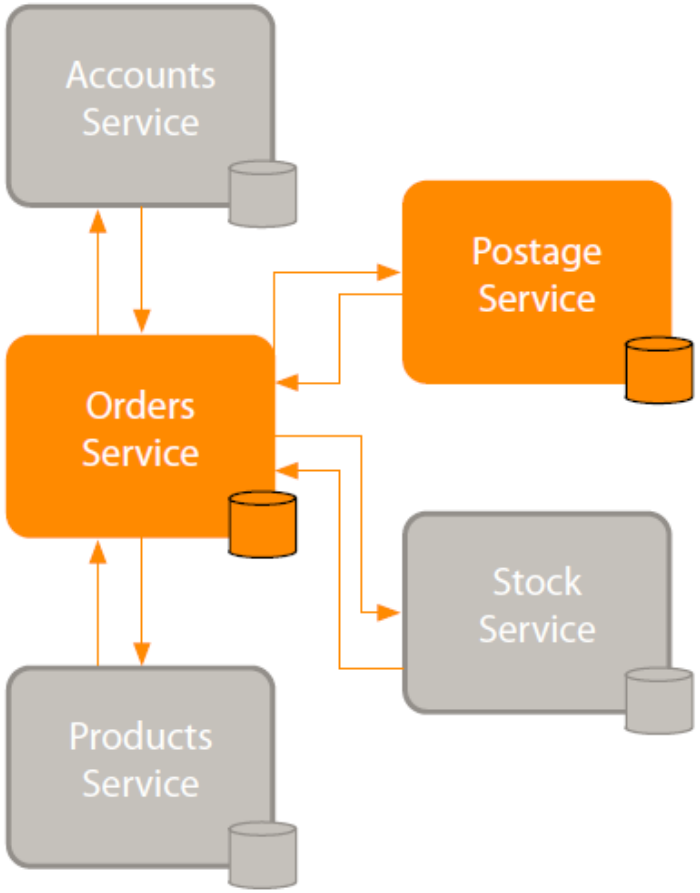
Business Domain Centric

Resilience

Observable

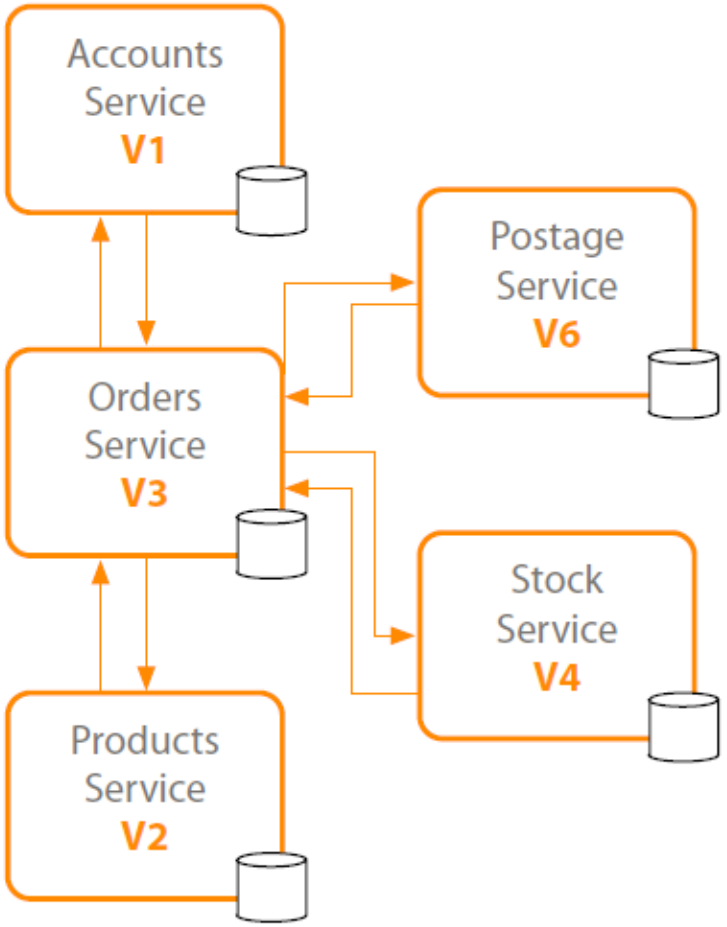
Automation

Design Principles Of MicroServices : High Cohesion



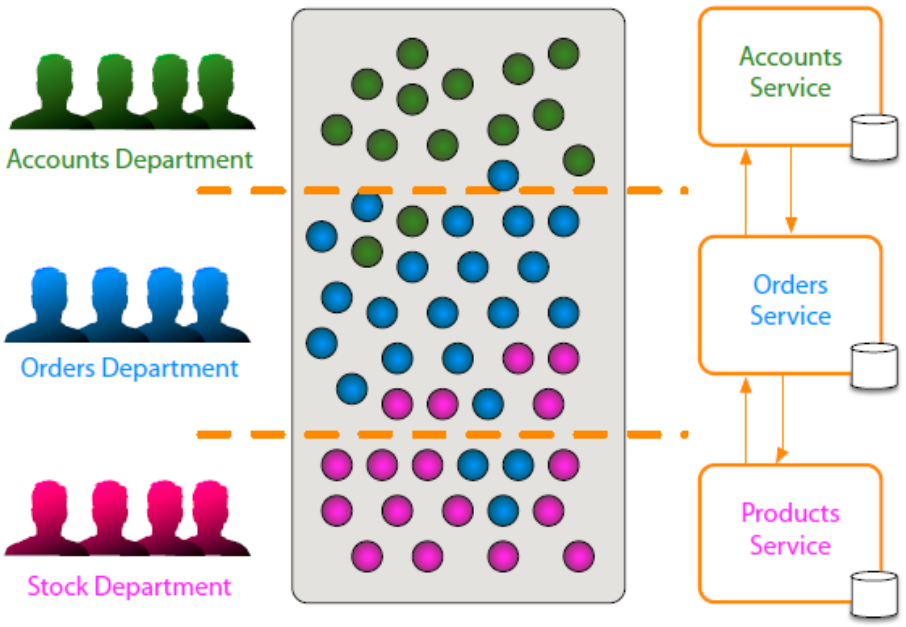
- Single focus
- Single responsibility
  - SOLID principle
  - Only change for one reason
- Reason represents
  - A business function
  - A business domain
- Encapsulation principle
  - OOP principle
- Easily rewritable code
- Why
  - Scalability
  - Flexibility
  - Reliability

Design Principles Of MicroServices : Autonomous



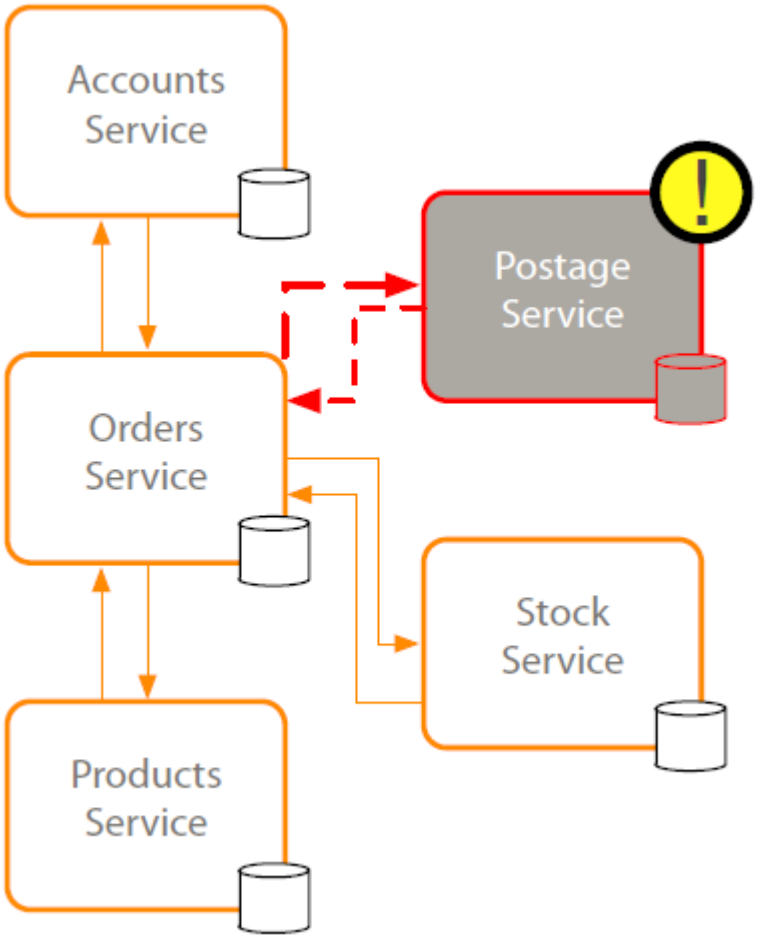
- Loose coupling
- Honor contracts and interfaces
- Stateless
- Independently changeable
- Independently deployable
- Backwards compatible
- Concurrent development

Design Principles Of MicroServices : Business Domain Centric



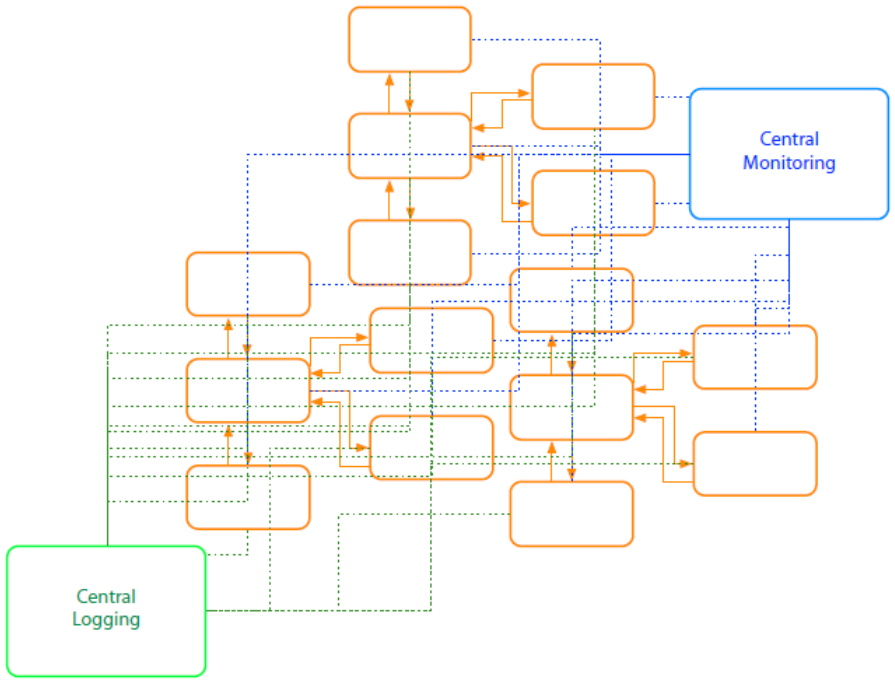
- Service represents business function
  - Accounts Department
  - Postage calculator
- Scope of service
- Bounded context from DDD
- Identify boundaries\seams
- Shuffle code if required
  - Group related code into a service
  - Aim for high cohesion
- Responsive to business change

Design Principles Of MicroServices : Resilience



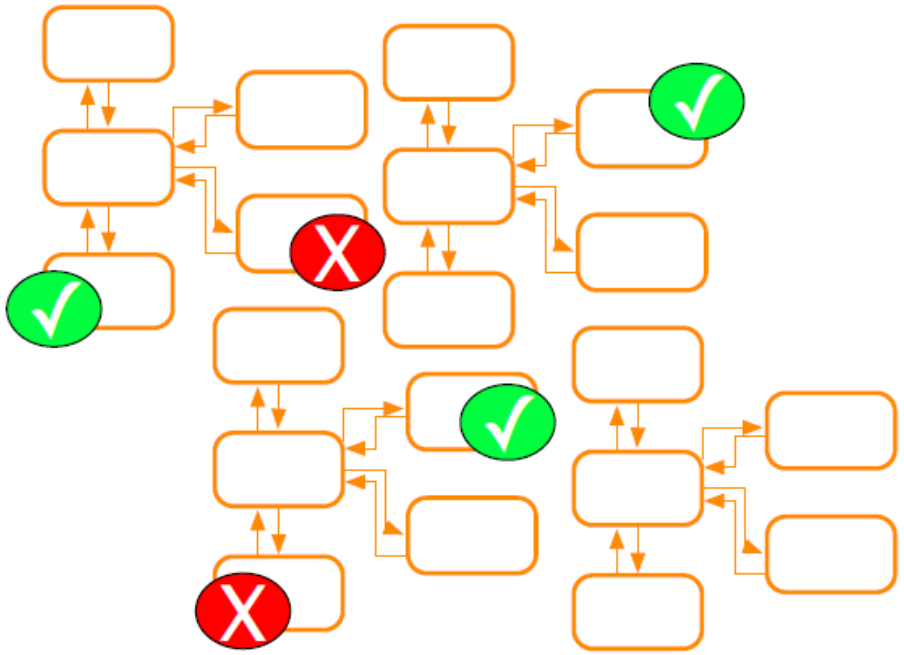
- Embrace failure
  - Another service
  - Specific connection
  - Third-party system
- Degrade functionality
- Default functionality
- Multiple instances
  - Register on startup
  - Deregister on failure
- Types of failure
  - Exceptions\Errors
  - Delays
  - Unavailability
- Network issues
  - Delay
  - Unavailability
- Validate input
  - Service to service
  - Client to service

Design Principles Of MicroServices : Observable



- System Health
  - Status
  - Logs
  - Errors
- Centralized monitoring
- Centralized logging
- Why
  - Distributed transactions
  - Quick problem solving
  - Quick deployment requires feedback
  - Data used for capacity planning
  - Data used for scaling
  - Whats actually used
  - Monitor business data

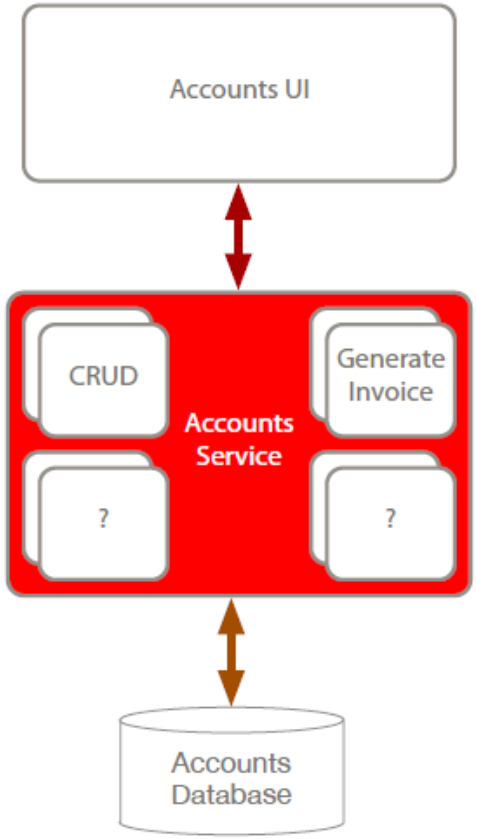
Design Principles Of MicroServices : Automation



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

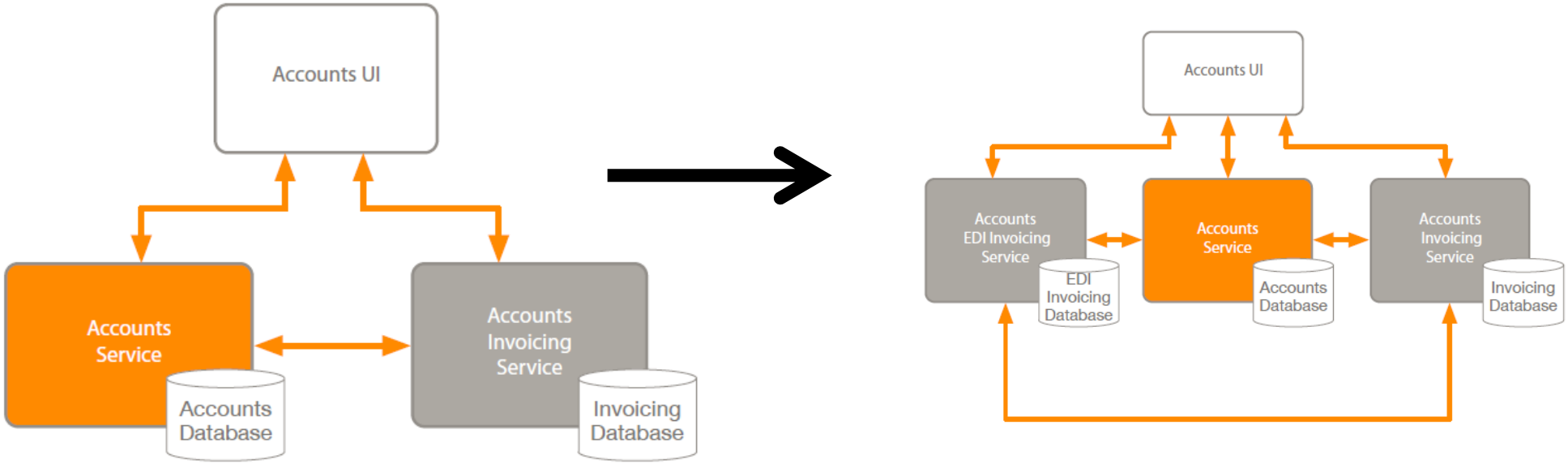
Approaches of MicroServices

Approach: High Cohesion

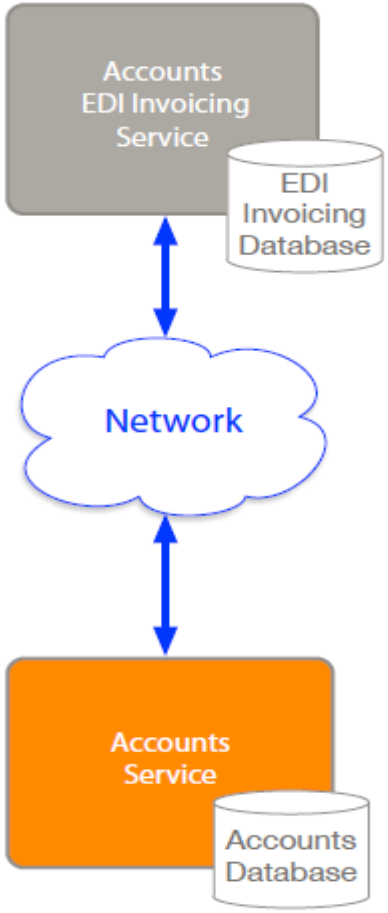


- Identify a single focus
  - Business function
  - Business domain
- Split into finer grained services
- Avoid “Is kind of the same”
- Open to create many Services
- Question in code\peer reviews
  - Can this change for more than one reason

Approach: High Cohesion

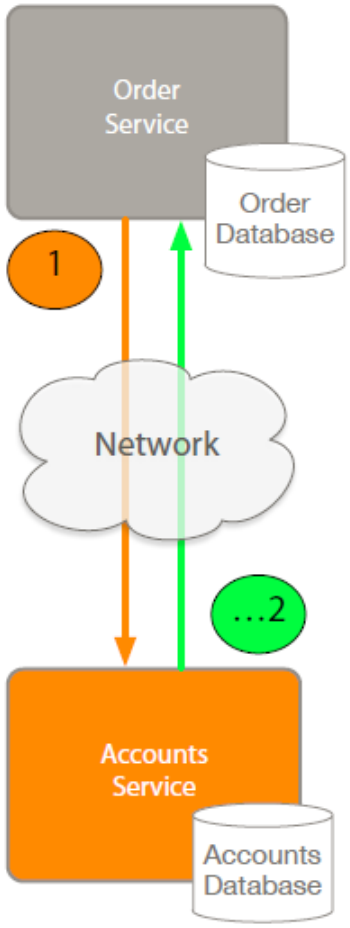


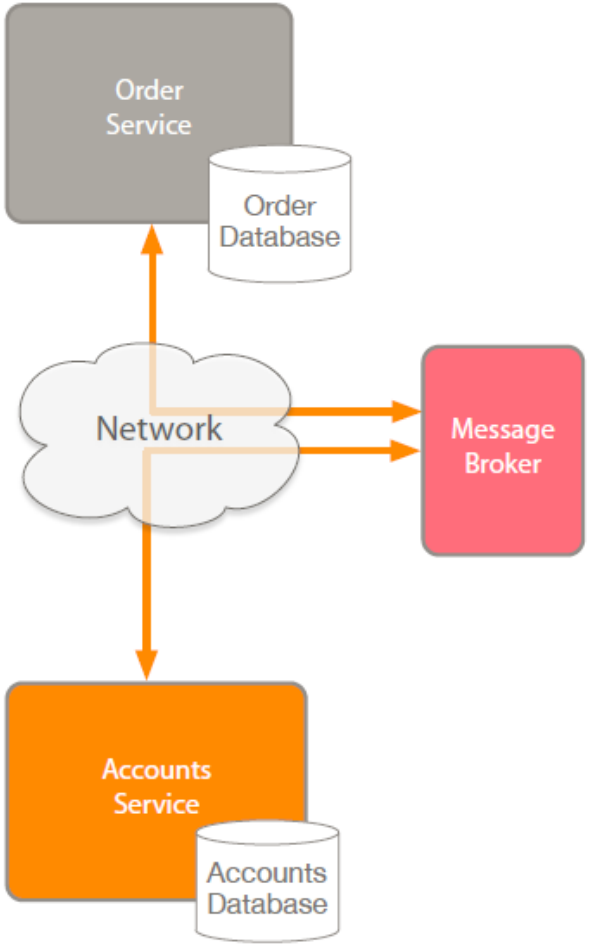
Approach: Autonomous

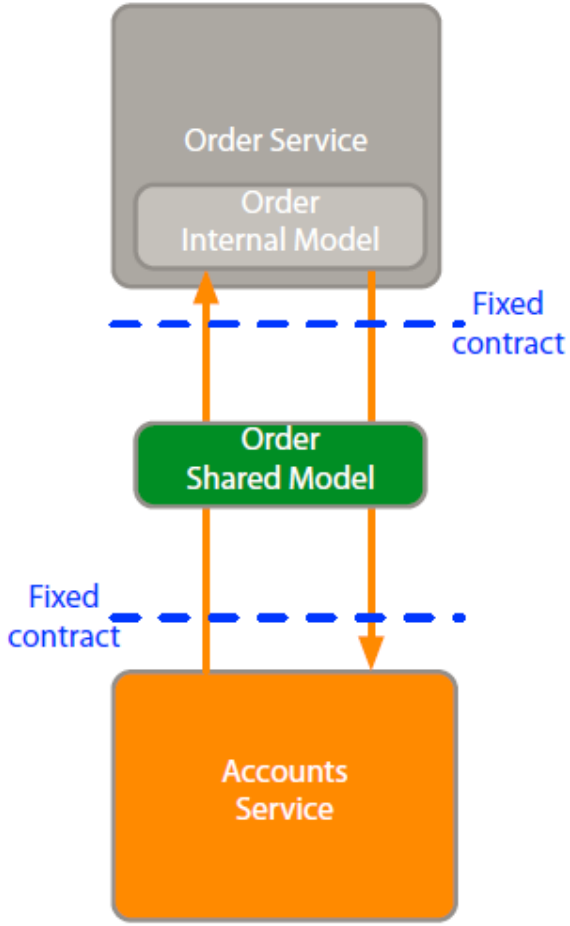


- Loosely coupled
  - Communication by network
    - Synchronous
    - Asynchronous
      - Publish events
      - Subscribe to events
- Technology agnostic API
- Avoid client libraries
- Contracts between services
  - Fixed and agreed interfaces
  - Shared models
  - Clear input and output
- Avoid chatty exchanges between services
- Avoid sharing between services
  - Databases
  - Shared libraries

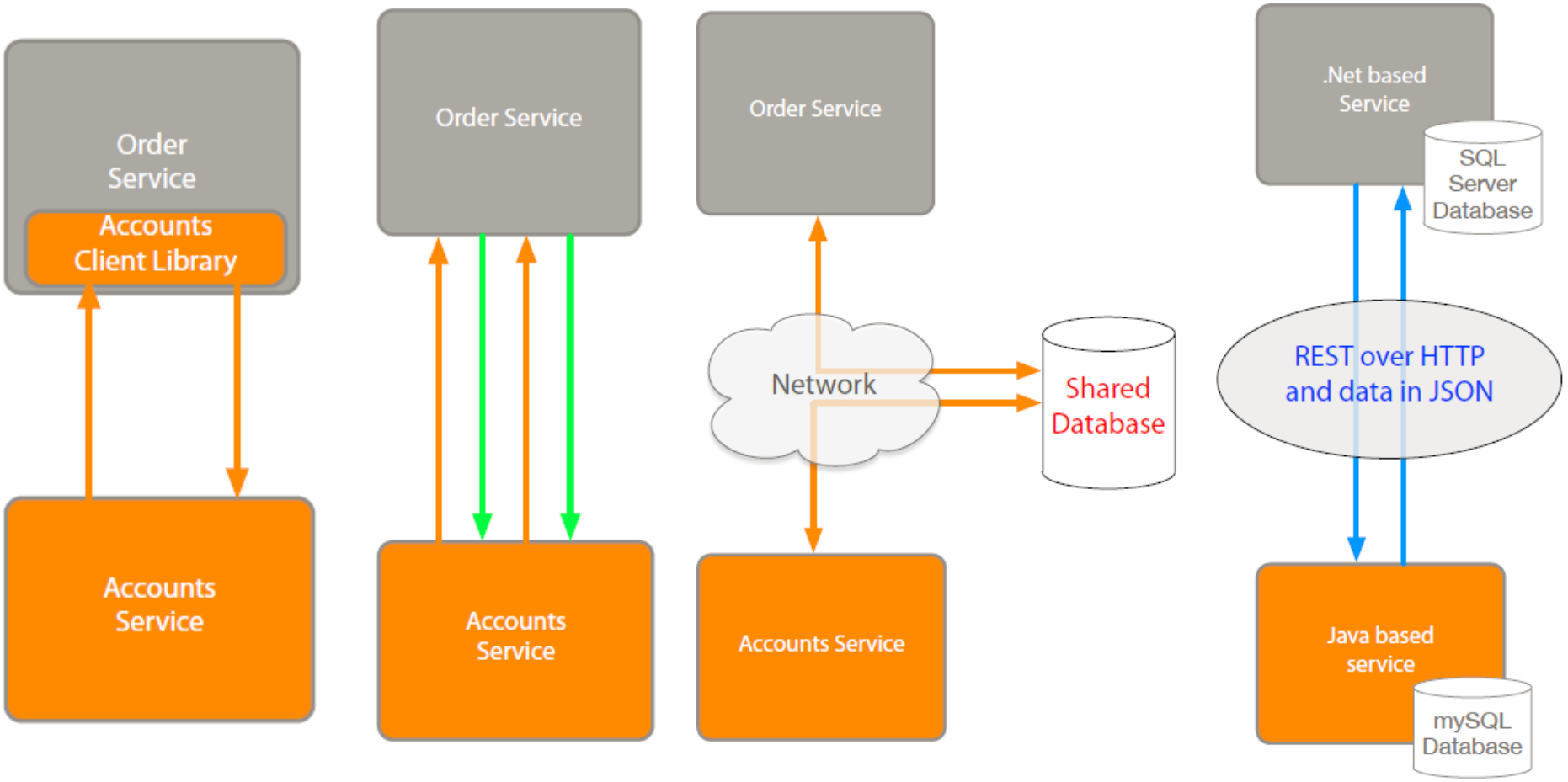
Approach: Autonomous



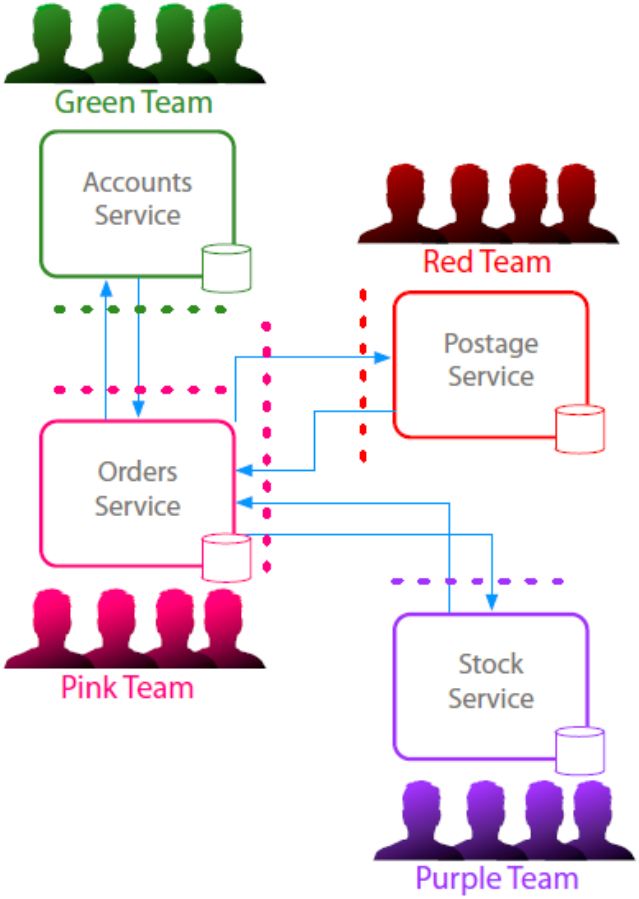




Approach: Autonomous

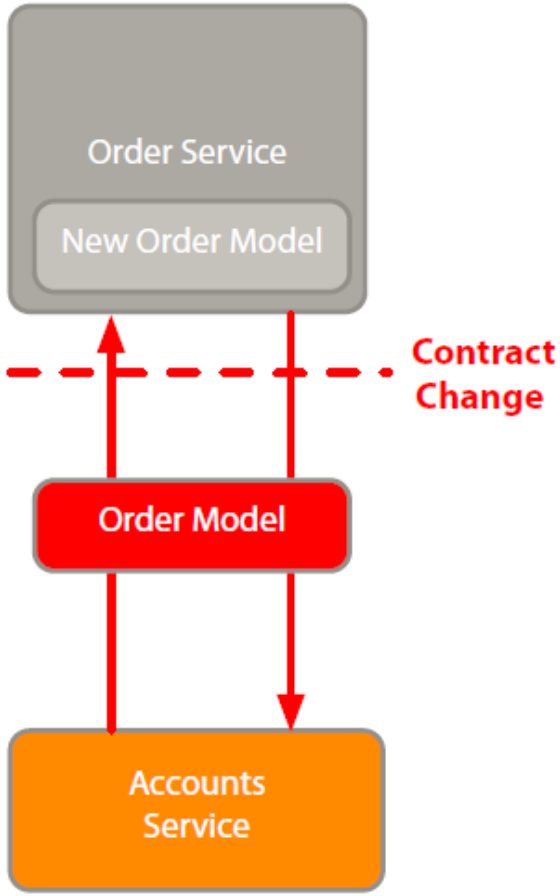


Approach: Autonomous



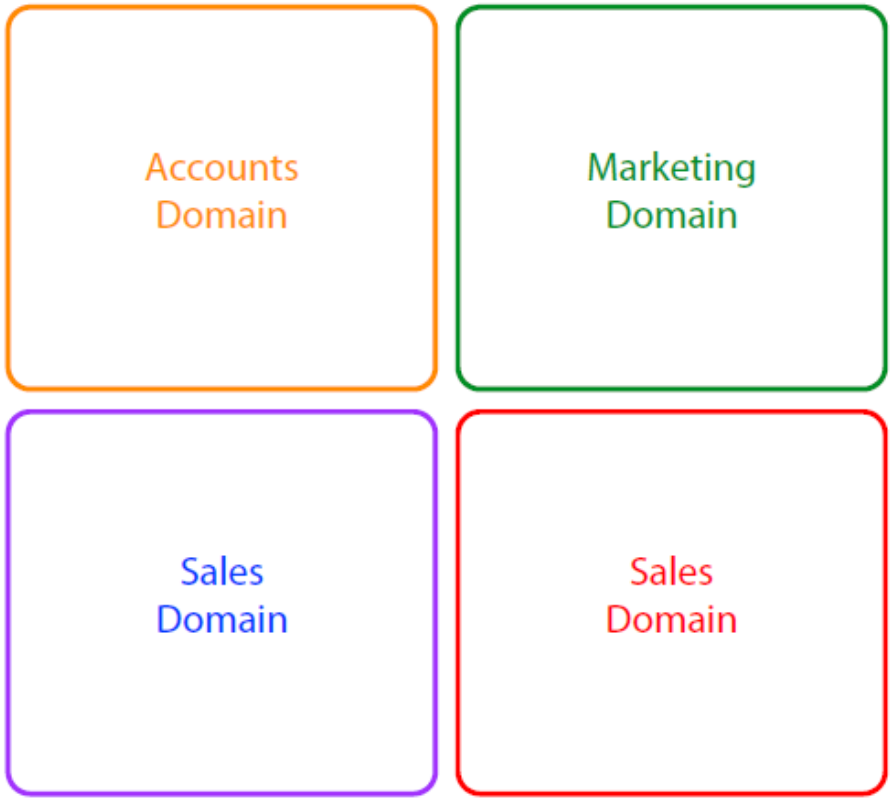
- •Microservice ownership by team
  - Responsibility to make autonomous
  - Agreeing contracts between teams
  - Responsible for long-term maintenance
  - Collaborative development
    - Communicate contract requirements
    - Communicate data requirements
  - Concurrent development

Approach: Autonomous



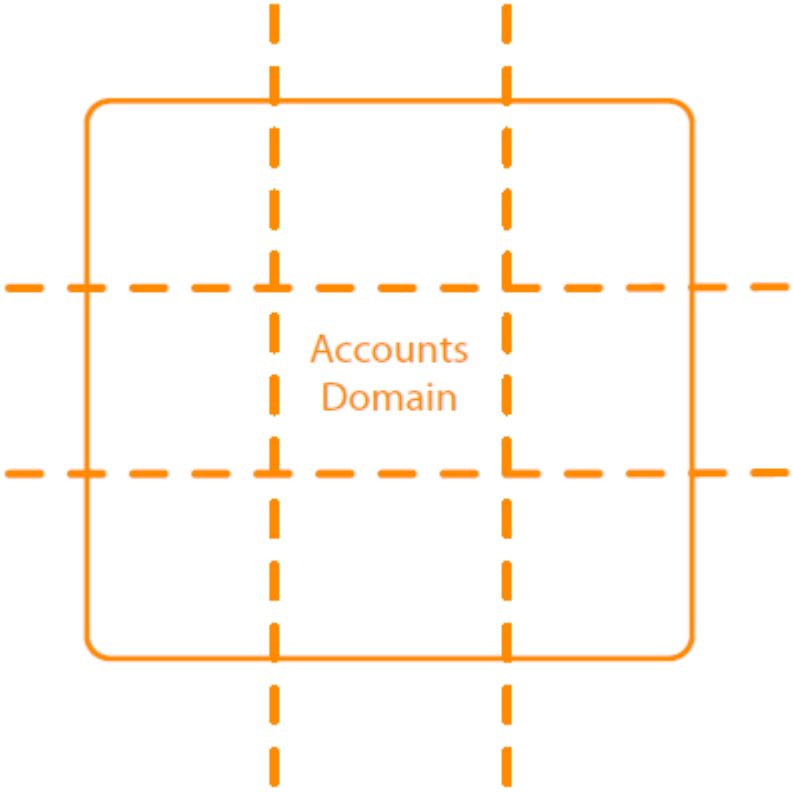
- Versioning
  - Avoid breaking changes
  - Backwards compatibility
  - Integration tests
  - Have a versioning strategy
    - Concurrent versions
    - Old and new
    - Semantic versioning
    - Major.Minor.Patch (e.g. 15.1.2)
    - Coexisting endpoints
    - /V2/customer/

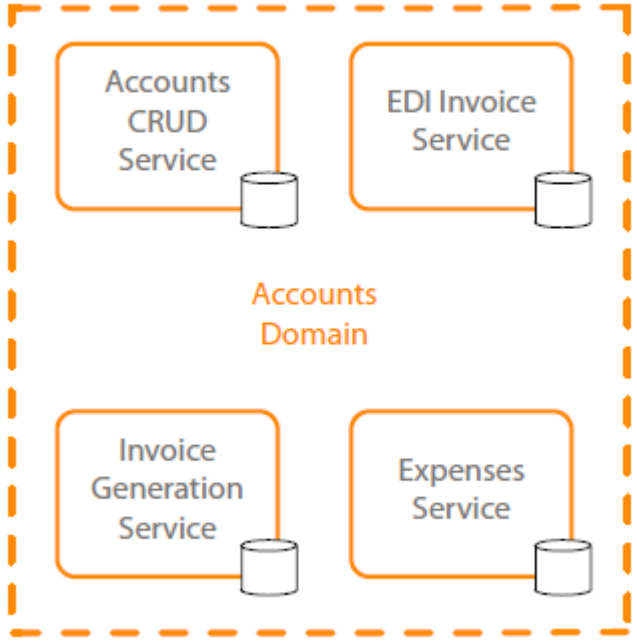
Approach: Business Domain Centric



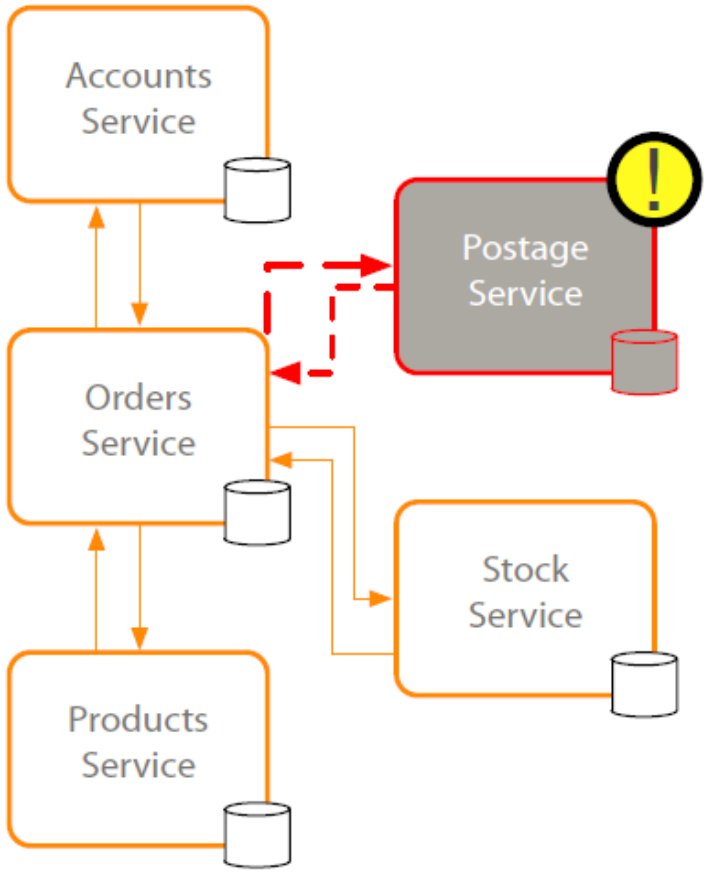
- Business function or business domain approach
  - Identify business domains in a coarse manner
  - Review sub groups of business functions or areas
  - Review benefits of splitting further
  - Agree a common language
- Microservices for data (CRUD) or functions
- Fix incorrect boundaries
  - Merge or split
- •Explicit interfaces for outside world
- •Splitting using technical boundaries
  - Service to access archive data
  - For performance tuning

Approach: Business Domain Centric



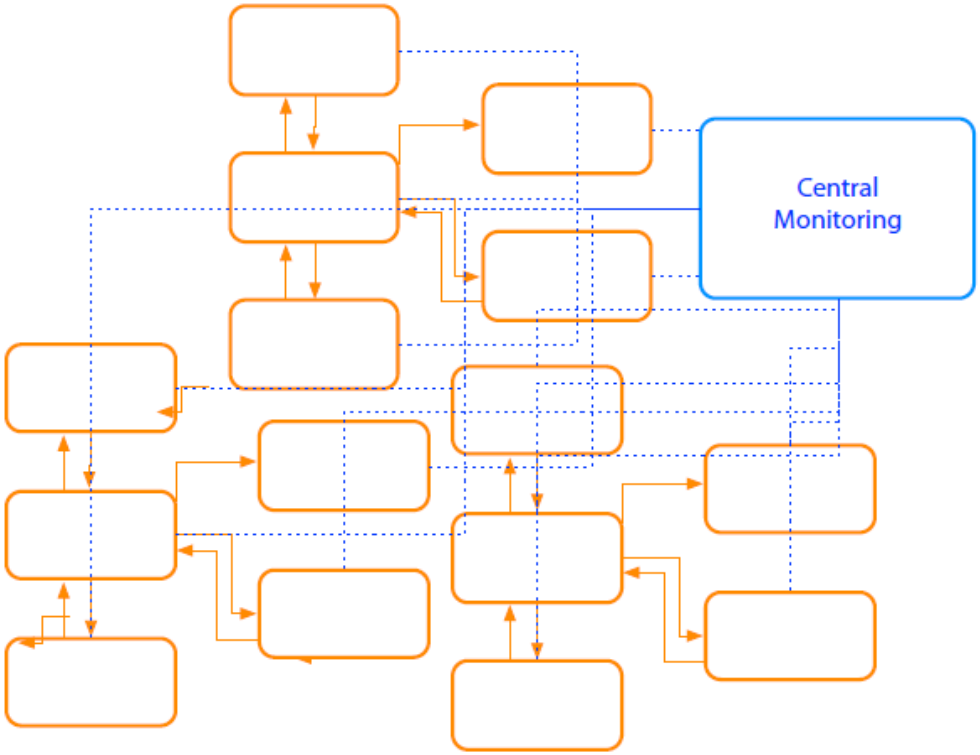


Approach: Resilience



- Design for known failures
- Failure of downstream systems
  - Other services internal or external
- Degrade functionality on failure detection
- Default functionality on failure detection
- Design system to fail fast
- 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 timeouts
- Log timeouts

Approach: Observable



- Centralized monitoring
  - Real-time monitoring

    Monitor the host
    - CPU, memory, disk usage, etc.

    Expose metrics within the services
    - Response times
    - Timeouts
    - Exceptions and errors

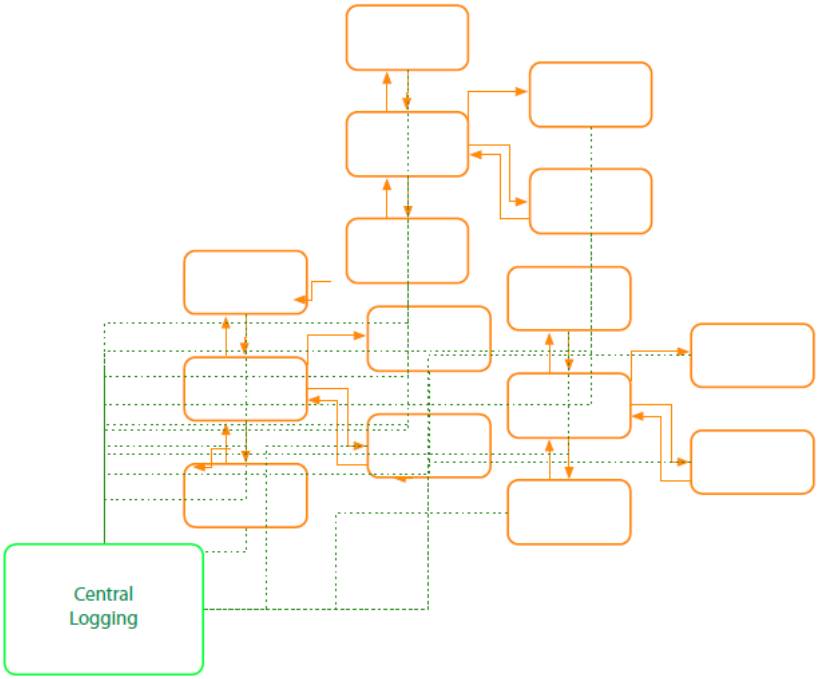
    Business data related metrics
    - Number of orders
    - Average time from basket to checkout

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

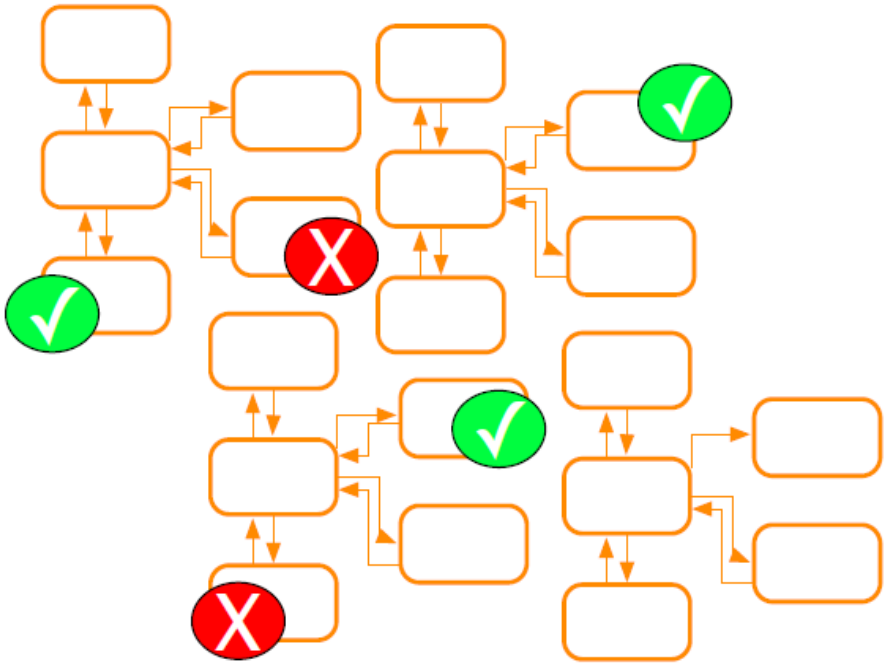
Approach: Observable

- **Centralized Logging**
  - When to log
  - Startup or shutdown
  - Code path milestones
    - Requests, responses and decisions
  - Timeouts, exceptions and errors
- **Structured logging**
  - Level
    - Information
    - Error
    - Debug
    - Statistic
  - Date and time
  - Correlation ID
  - Host name
  - Service name and service instance
  - Message

- **Traceable distributed transactions**
  - Correlation ID
  - Passed service to service

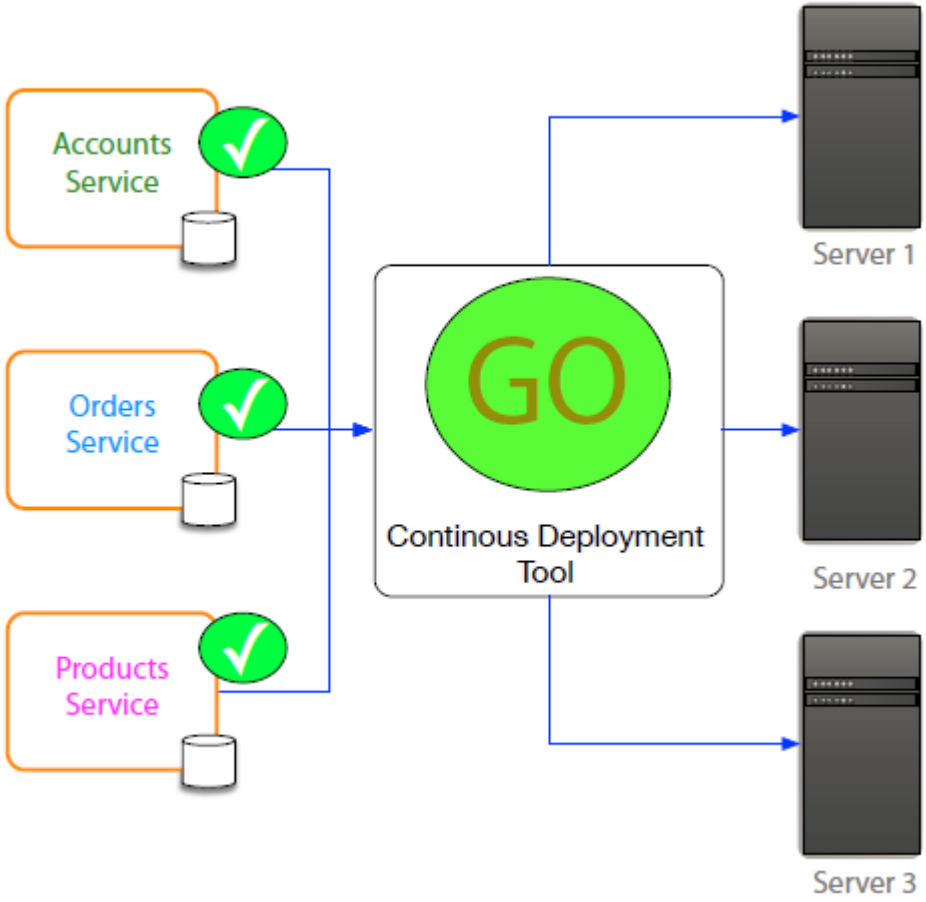


Approach: Automation



- **Continuous Integration Tools**
  - Work with source control systems
  - Automatic after check-in
  - Unit tests and integration tests required
  - Ensure quality of check-in
    - Code compiles
    - Tests pass
    - Changes integrate
    - Quick feedback
  - Urgency to fix quickly
  - Creation of build
  - Build ready for test team
  - Build ready for deployment

Approach: Automation



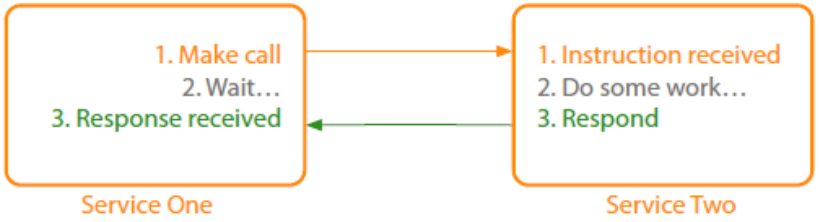
- **Continuous Deployment Tools**
  - Automate software deployment
    - Configure once
    - Works with CI tools
    - Deployable after check in
    - Reliably released at anytime
  - Benefits
    - Quick to market
    - Reliable deployment
    - Better customer experience

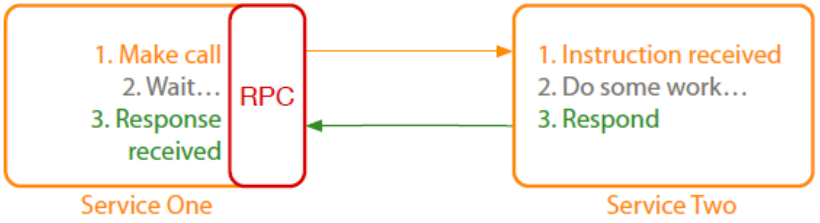
Technology for Microservices

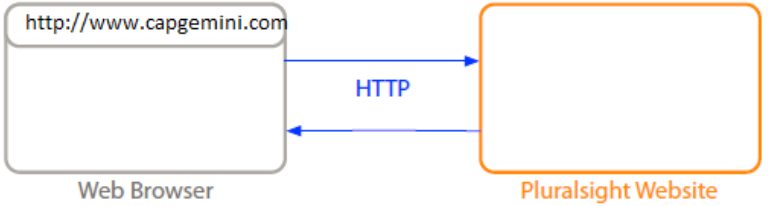
Technology for Microservices

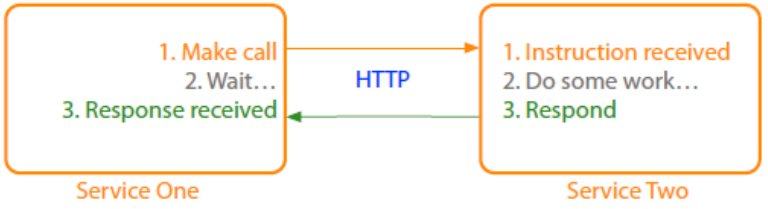
- Communication
  - Synchronous | Asynchronous
- Hosting Platforms
- Observable Microservices
- Performance
- Automation Tools

Communication: Synchronous



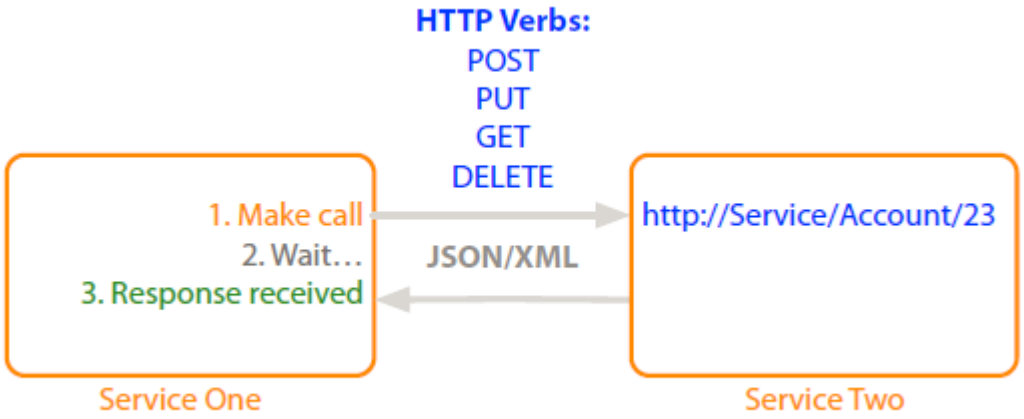




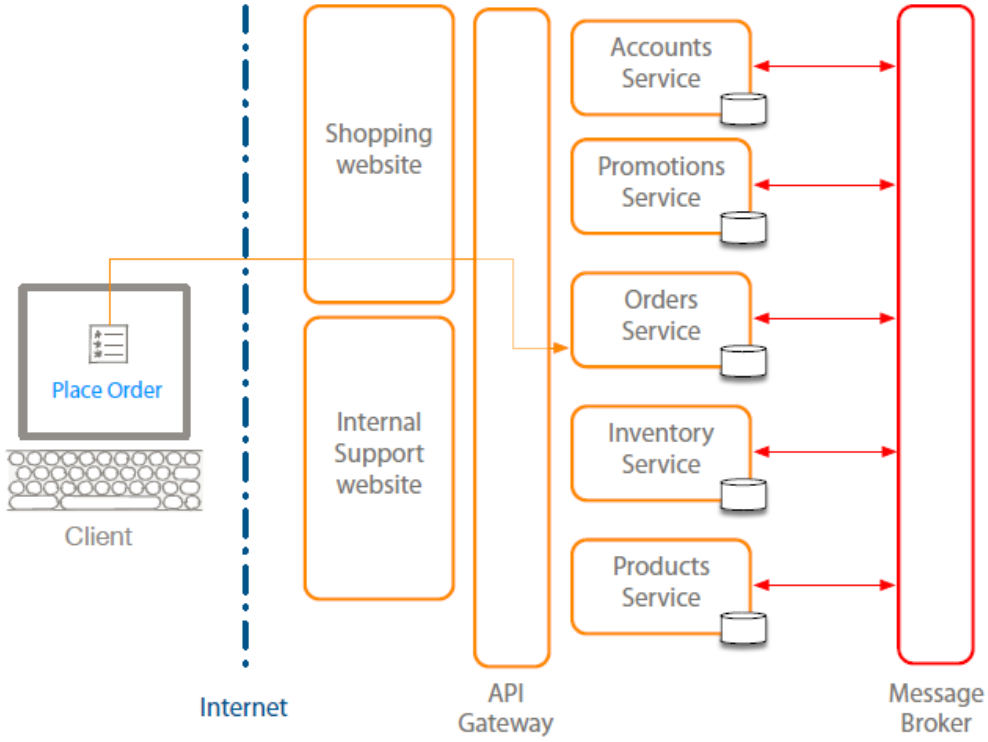


- **Request response communication**
  - Client to service
  - Service to service
  - Service to external
- **Remote procedure call**
  - Sensitive to change
- **HTTP**
  - Work across the internet
  - Firewall friendly
- **REST**
  - CRUD using HTTP verbs
  - Natural decoupling
  - Open communication protocol
  - REST with HATEOS
- **Synchronous issues**
  - Both parties have to be available
  - Performance subject to network quality
  - Clients must know location of service (host\port)

Communication: Synchronous

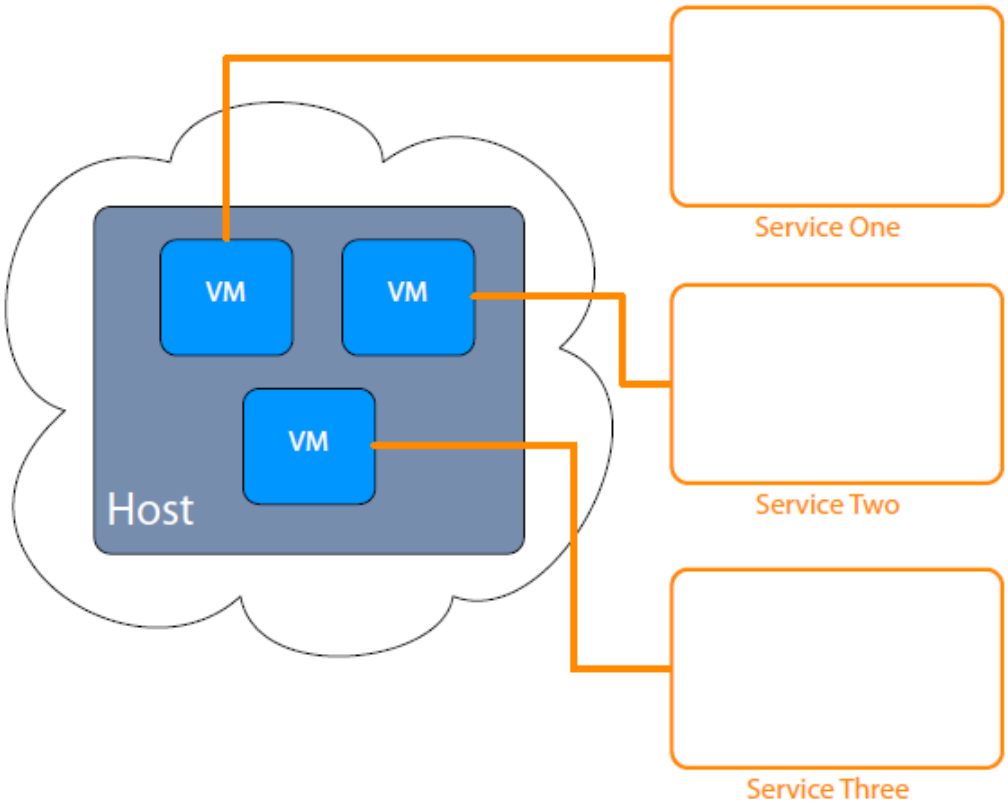


Communication: Asynchronous



- **Event based**
  - Mitigates the need of client and service availability
  - Decouples client and service
- **Message queuing protocol**
  - Message Brokers
  - Subscriber and publisher are decoupled
  - Microsoft message queuing (MSMQ)
  - **RabbitMQ**
  - ATOM (HTTP to propagate events)
- **Asynchronous challenge**
  - Complicated
  - Reliance on message broker
  - Visibility of the transaction
  - Managing the messaging queue
- **Real world systems**
  - Would use both synchronous and asynchronous

Hosting Platforms - Virtualization



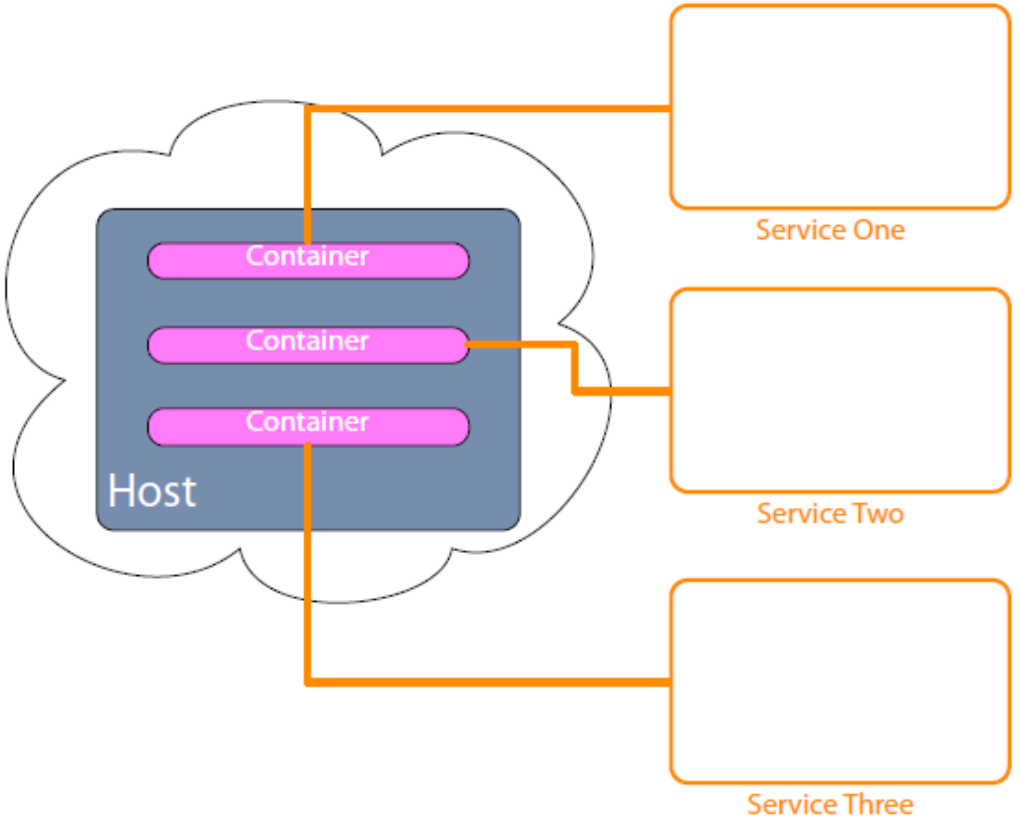
- **A virtual machine as a host**
- **Foundation of cloud platforms**
  - Platform as a service (PAAS)
    - Microsoft Azure
    - Amazon web services
    - Your own cloud (for example vSphere)
- **Could be more efficient**
  - Takes time to setup
  - Takes time to load
  - Take quite a bit of resource
- **Unique features**
  - Take snapshot
  - Clone instances
- **Standardized and mature**

Hosting Platforms: Containers

- **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
- **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

Examples

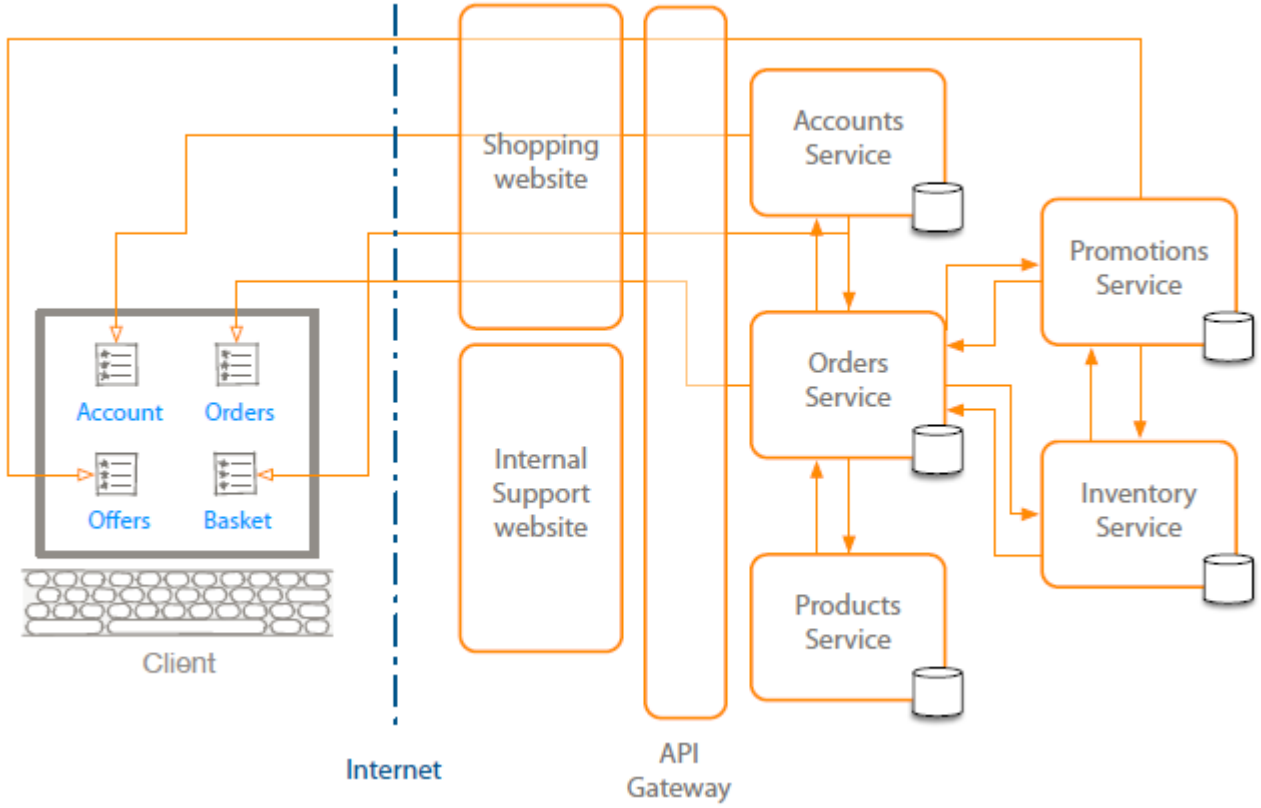
Docker
Rocker
Glassware



Hosting Platforms: Self Hosting

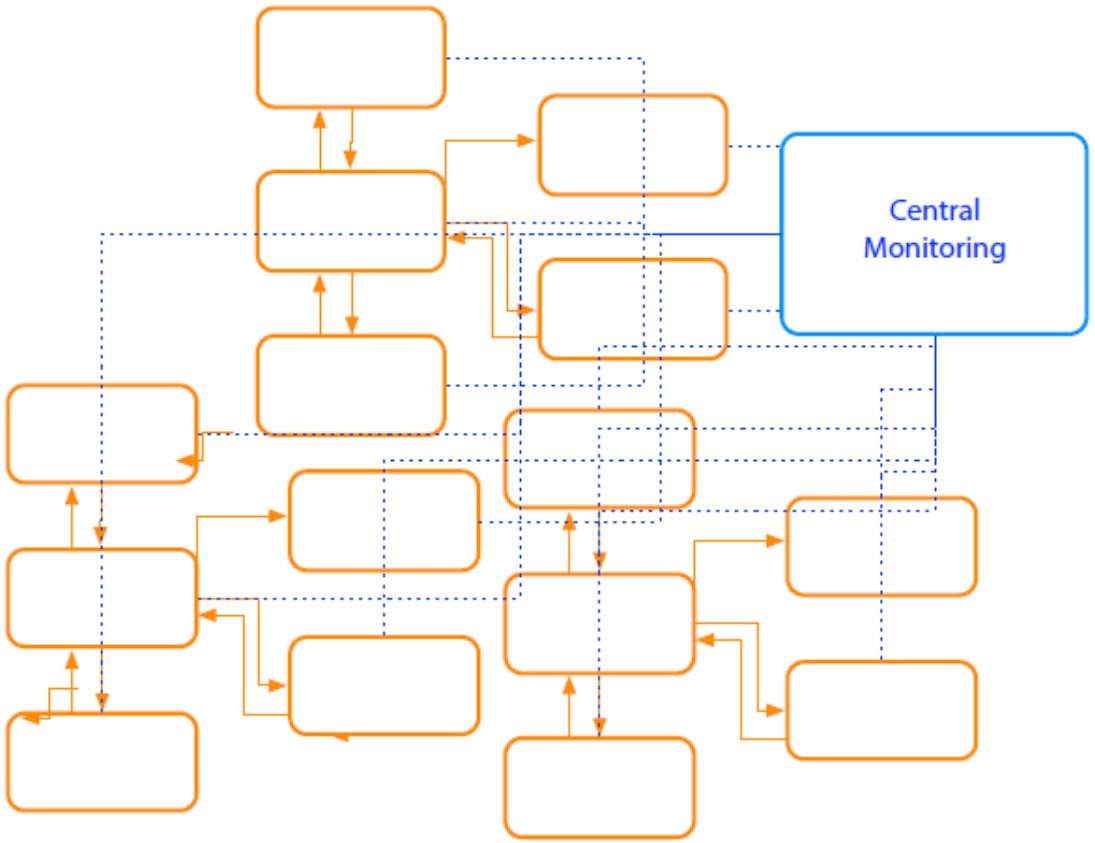
- **Implement your own cloud**
  - Virtualization platform
  - Implement containers
- **Use of physical machines**
  - Single service on a server
  - Multiple services on a server
- **Challenges**
  - Long-term maintenance
  - Need for technicians
  - Training
  - Need for space
  - Scaling is not as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

Hosting Platforms: Registration and Discovery



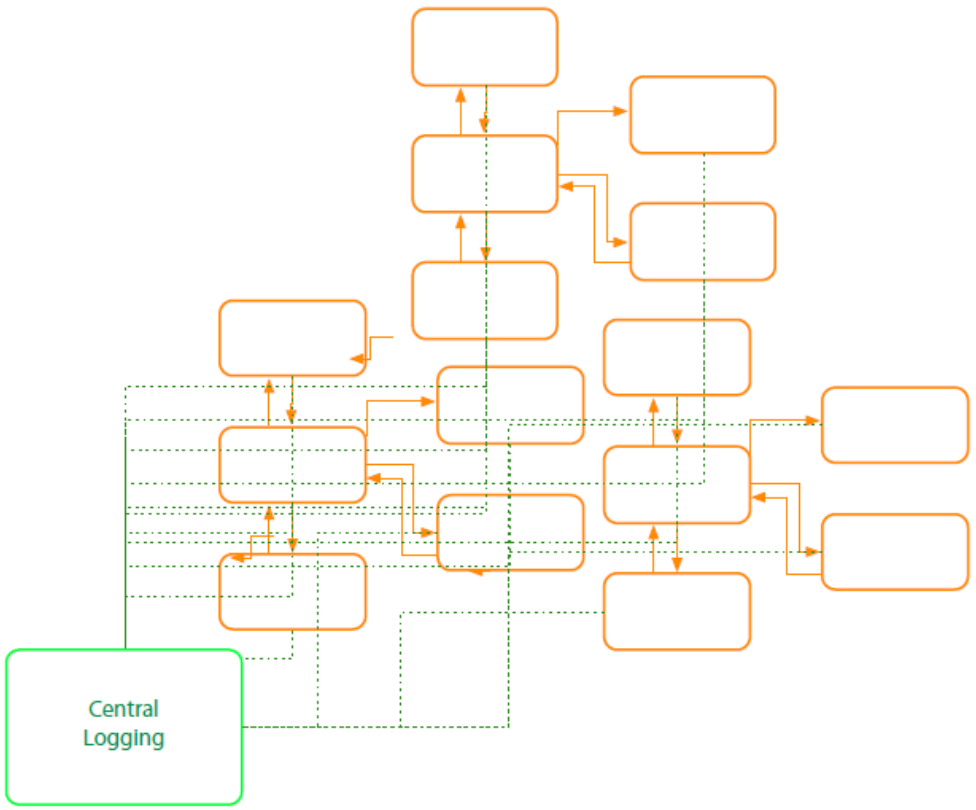
- Where?
  - Host, port and version
- Service registry database
- Register on startup
- Deregister service on failure
- Cloud platforms make it easy
- Local platform registration options
  - Self registration
  - Third-party registration
- Local platform discovery options
  - Client-side discovery
  - Server-side discovery

Observable Microservices: Monitoring Tech



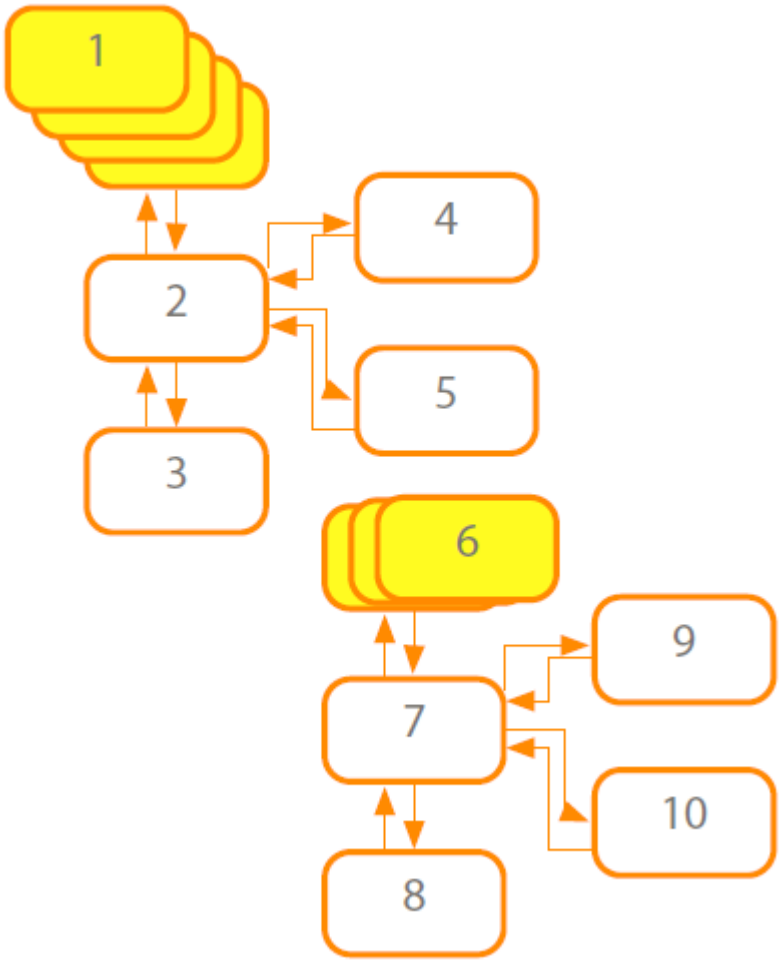
- Centralized tools
  - Nagios
  - PRTG
  - Load balancers
  - New Relic
- Desired features
  - Metrics across servers
  - Automatic or minimal configuration
  - Client libraries to send metrics
  - Test transactions support
  - Alerting
- Network monitoring
- Standardize monitoring
  - Central tool
  - Preconfigured virtual machines or containers
- Real-time monitoring

Observable Microservices: Logging Tech



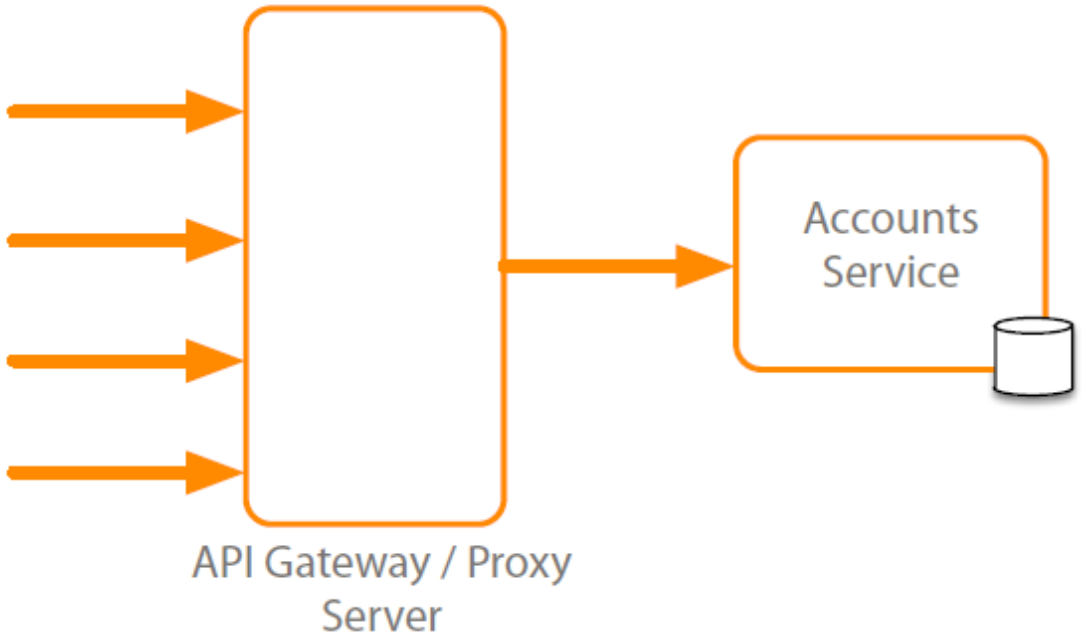
- **Portal for centralized logging data**
  - Elastic log
  - Log stash
  - Splunk
  - Kibana
  - Graphite
- **Client logging libraries**
  - Serilog and many more…
- **Desired features**
  - Structured logging
  - Logging across servers
  - Automatic or minimal configuration
  - Correlation\Context ID for transactions
- **Standardize logging**
  - Central tool
  - Template for client library

Microservices Performance: Scaling



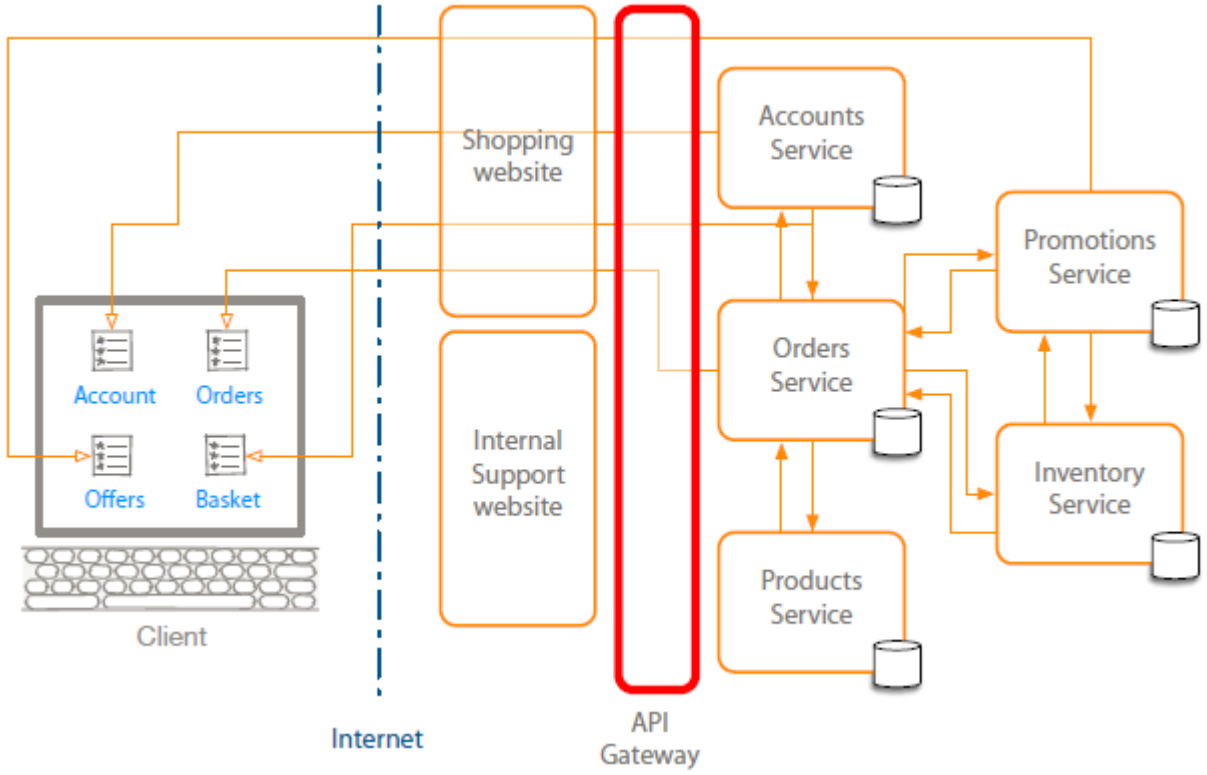
- **How**
  - Creating multiple instances of service
  - Adding resource to existing service
- **Automated or on-demand**
- **PAAS auto scaling options**
- **Virtualization and containers**
- **Physical host servers**
- **Load balancers**
  - API Gateway
- **When to scale up**
  - Performance issues
  - Monitoring data
  - Capacity planning

Microservices Performance: 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

Microservices Performance: API Gateway



- **Help with performance**
  - Load balancing
  - Caching

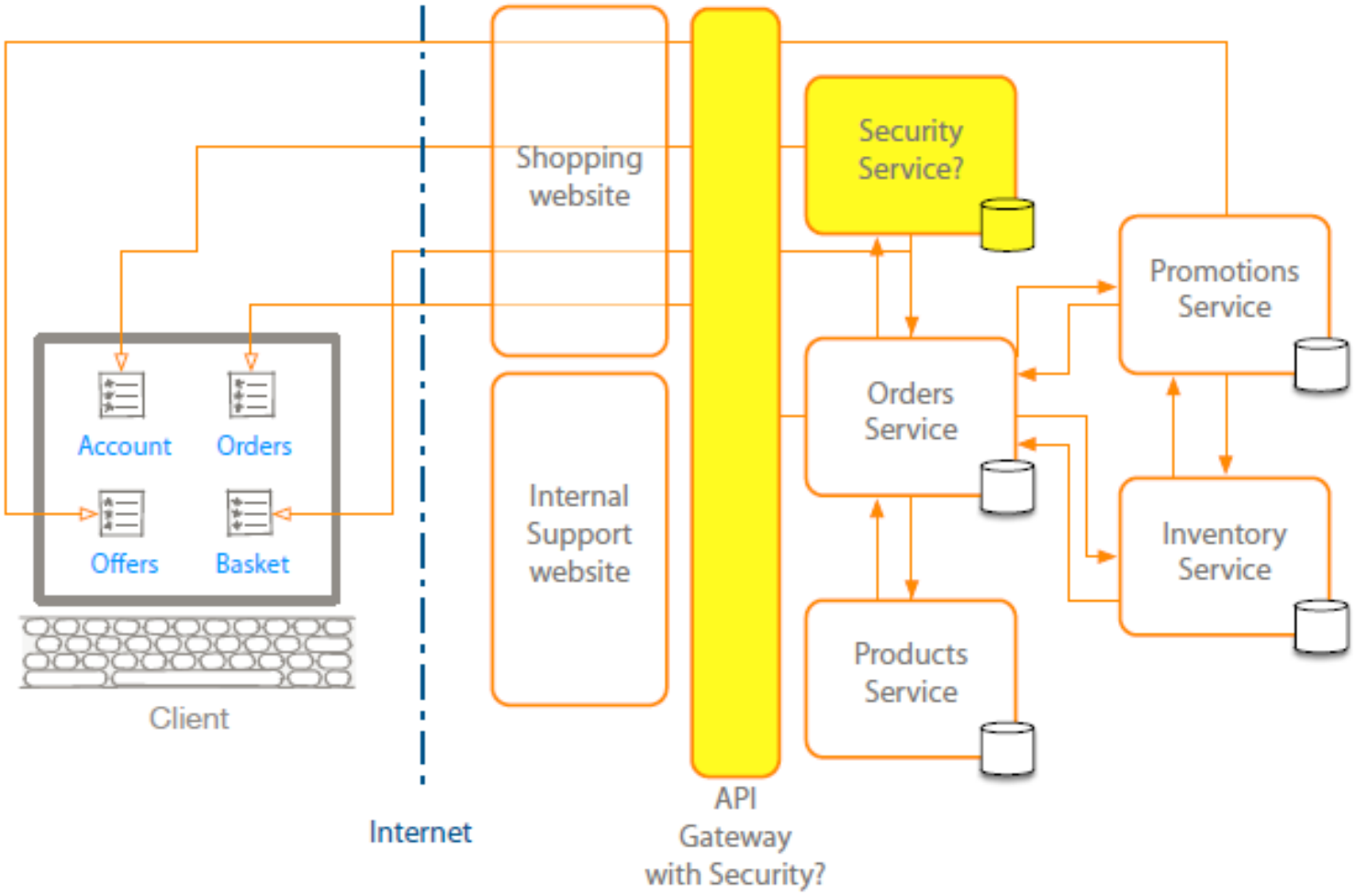
Help with

- 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

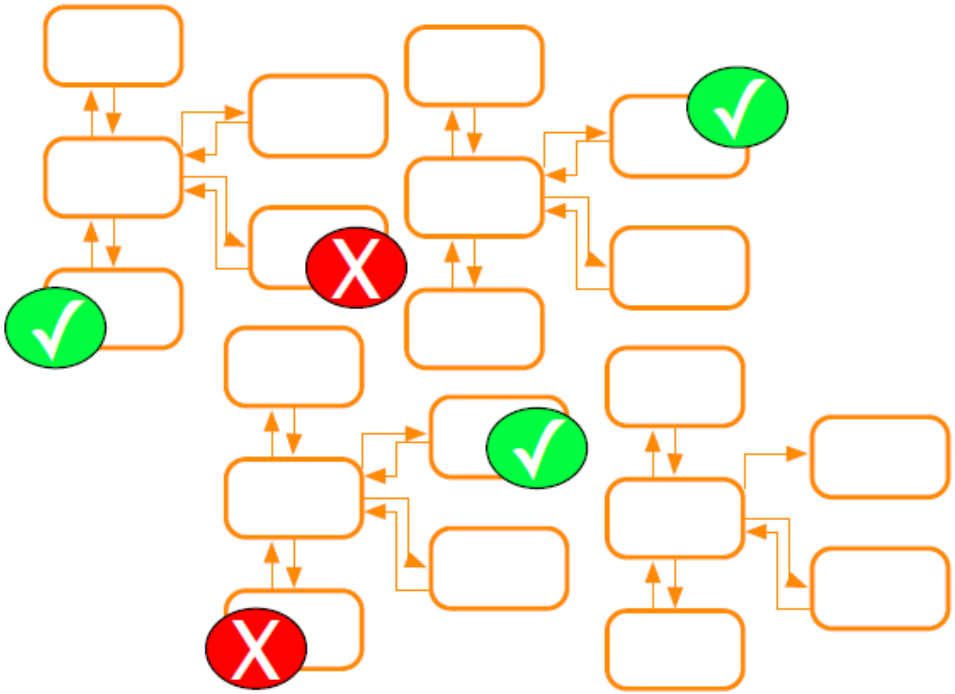
Security

- API Gateway
- Dedicated security service
- Central security vs service level

Microservices Performance: API Gateway

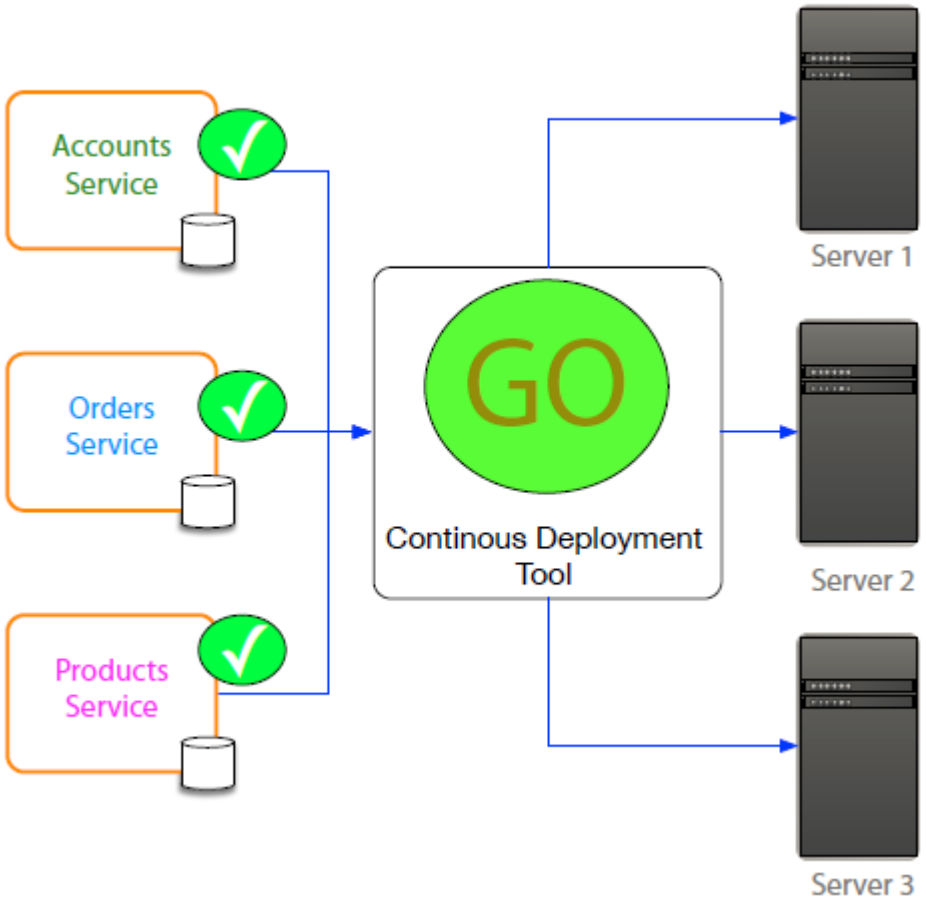


Automation Tools: Continuous Integration



- **Many CI tools**
  - Team Foundation Server
  - TeamCity and Many more!
- **Desired features**
  - Cross platform
    - Windows builders, Java builders and others
  - Source control integration
  - Notifications
  - IDE Integration (optional)
- **Map a microservice to a 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**

Automation Tools: Continuous Deployment



- **Many CD tools**
  - Aim for cross platform tools
- **Desired 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

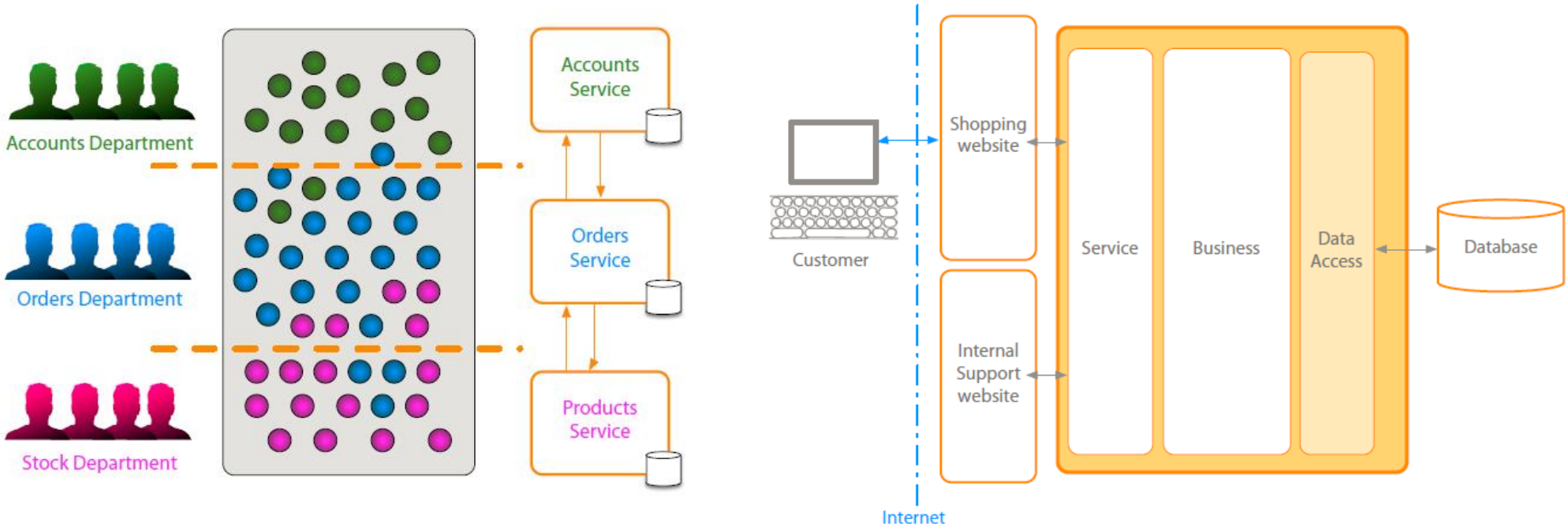
Moving Forwards with MicroServices

- Brownfield Microservices
  - Approach
  - Migration
  - Database Migration
  - Transactions
  - Reporting
- Greenfield Microservices
- Microservices Provisos

Brownfield Microservices: Approach

- **Existing system**
  - Monolithic system
  - Organically grown
  - Seems to large to split
- **Lacks microservices design principles**
- **Identify seams**
  - Separation that reflects domains
  - Identify bounded contexts
- **Start modularising the bounded contexts**
  - 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
- **Seams are future microservice boundaries**

Brownfield Microservices: Approach

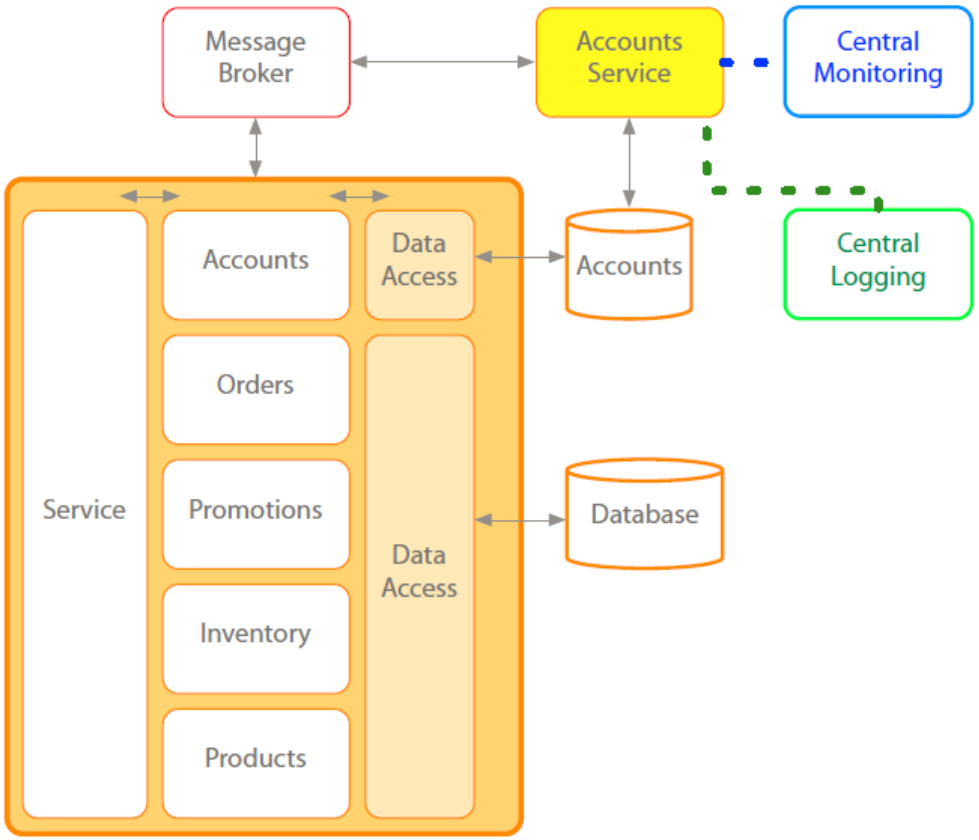


Brownfield Microservices: Migration

- 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
- How to prioritise what to split?
  - By risk
  - By technology
  - By dependencies

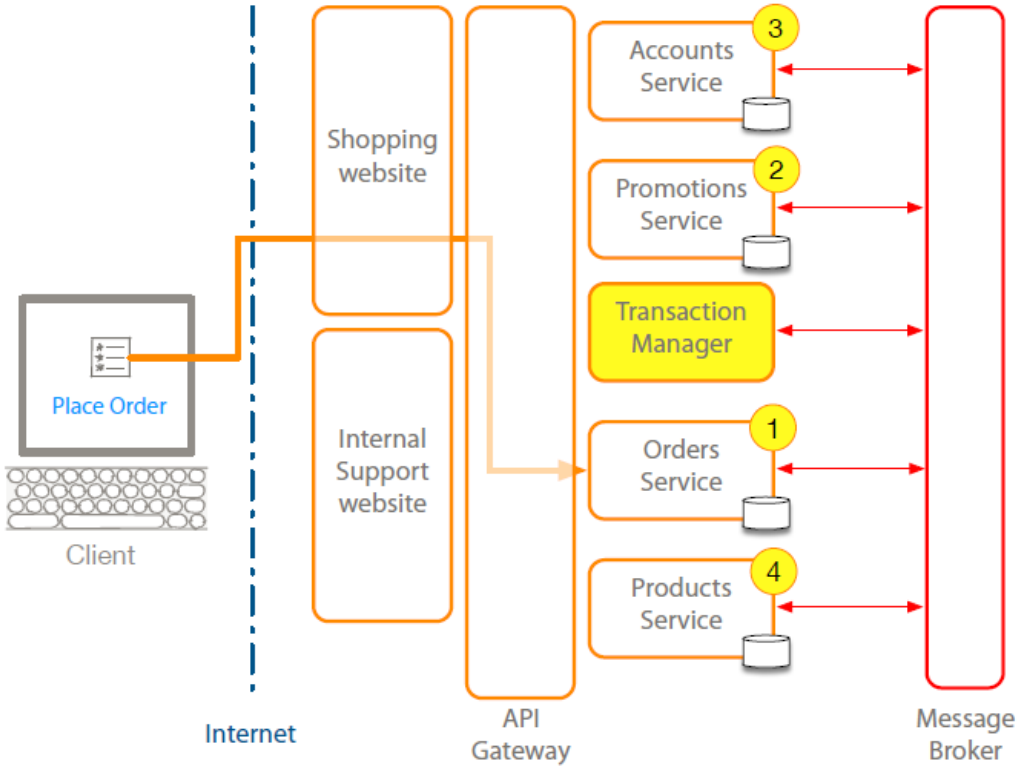
- Incremental approach
- Integrating with the monolithic
  - Monitor both for impact
  - Monitor operations that talk to microservices
  - Review and improve infrastructure
  - Incrementally the monolithic will be converted

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

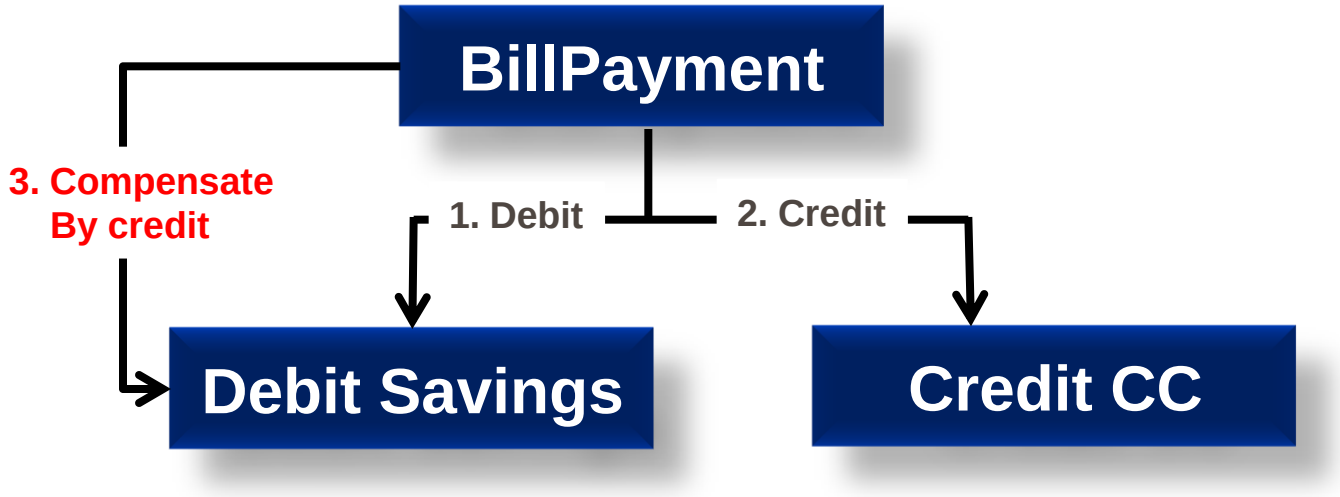
Brownfield Microservices: Transactions



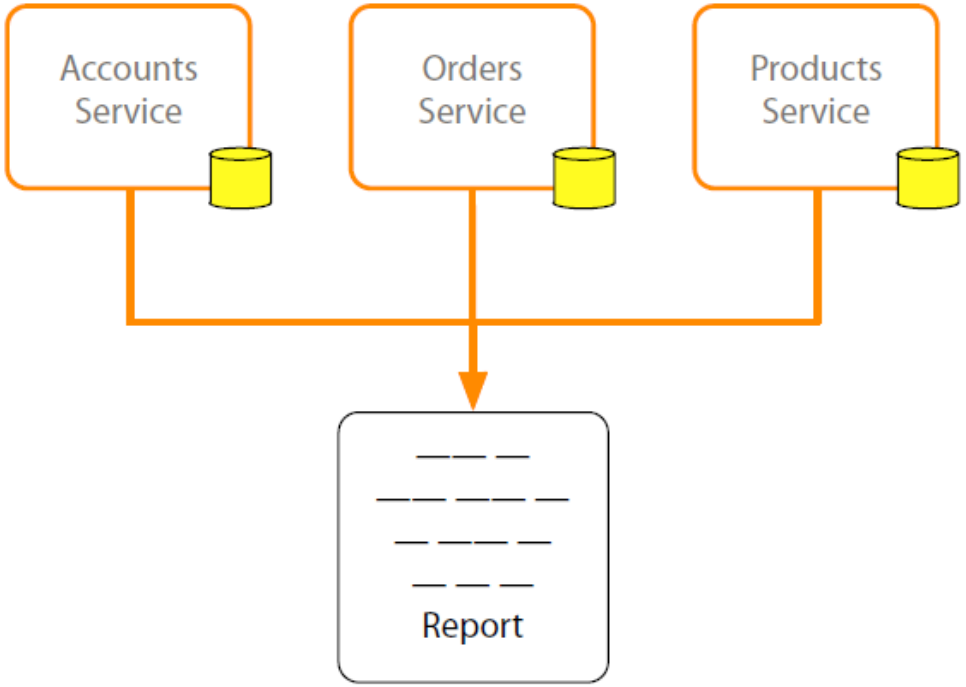
- Transactions ensure data integrity
- Transactions are simple in monolithic applications
- Transactions spanning microservices are complex
  - Complex to observe
  - Complex to problem solve
  - Complex to rollback
- Options for failed transactions
  - Try again later
  - Abort entire transaction
  - Use a transaction manager
    - Two phase commit
  - Disadvantage of transaction manager
    - Reliance on transaction manager
    - Delay in processing
    - Potential bottleneck
    - Complex to implement
- Distributed transaction compatibility
  - Completed message for the monolith

Brownfield Microservices: Transactions

- **Compensating Transactions**
  - State Store.
  - Routing Slip.
  - Process Manager.
- **Two Phase Commit**
  - Prepare Phase
  - Commit Phase
  - Forget Phase

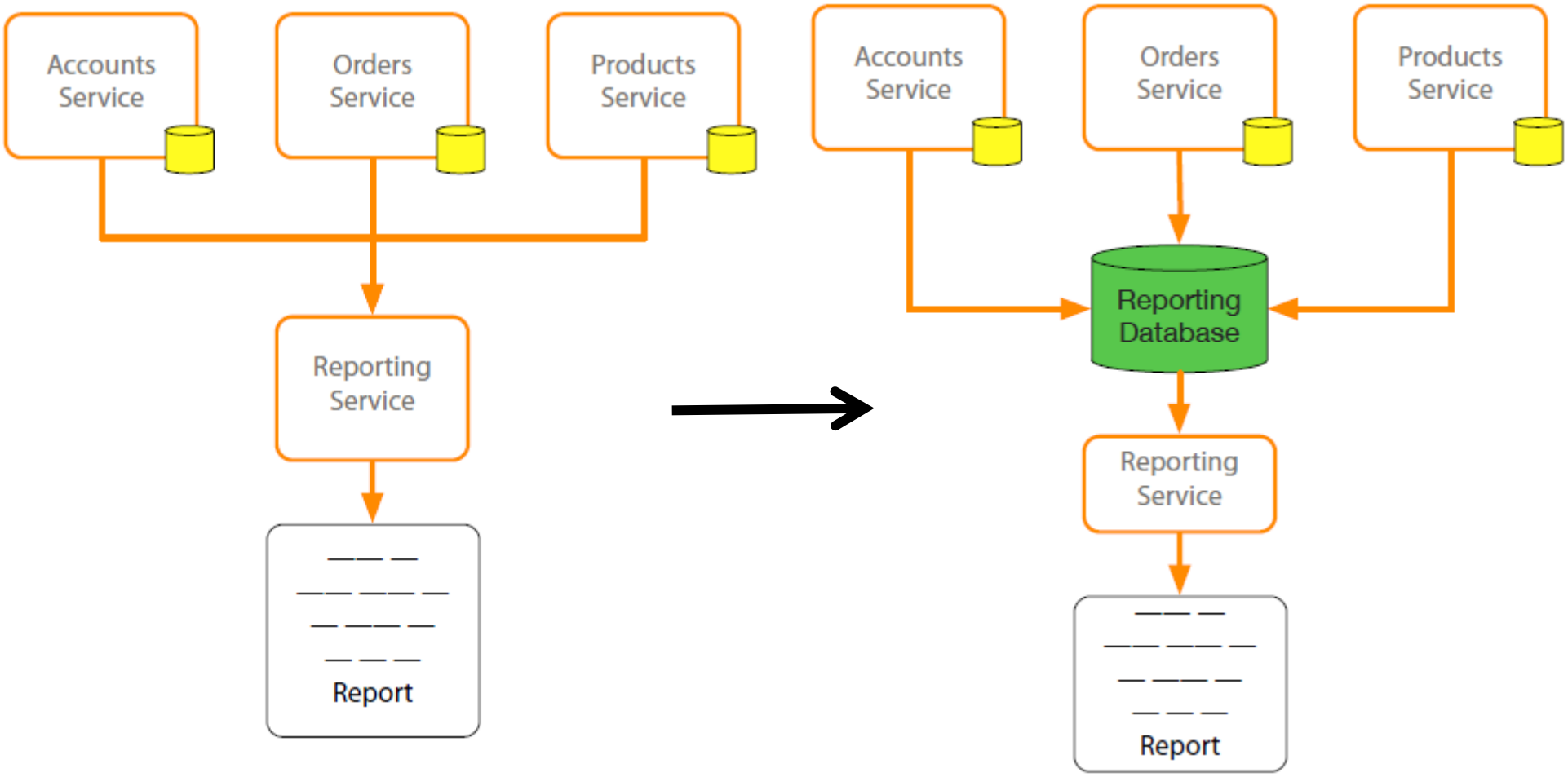


Brownfield Microservices: Reporting

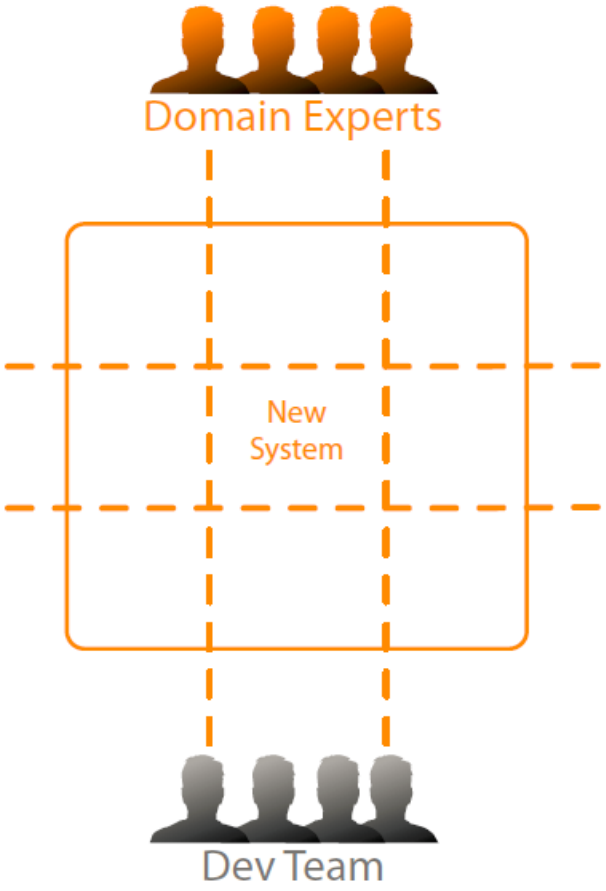


- Microservices complicate reporting
  - Data split across microservices
  - No central database
  - Joining data across databases
  - Slower reporting
  - Complicate report development
- Possible solutions
  - Service calls for report data
  - Data dumps
  - Consolidation environment

Brownfield Microservices: Reporting



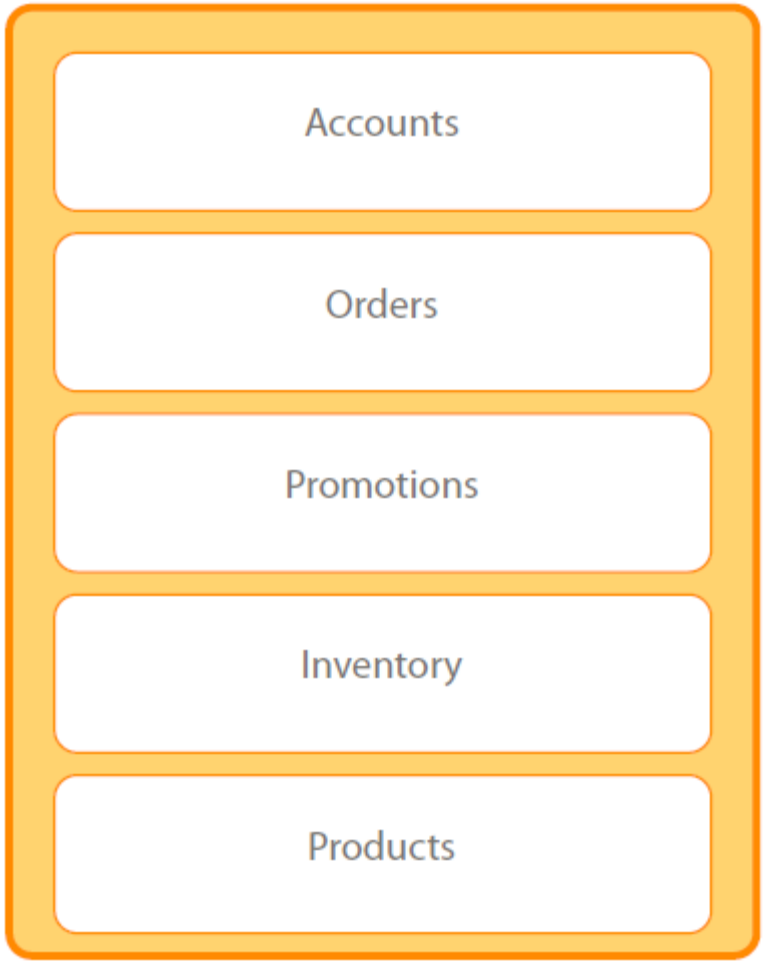
Greenfield Microservices



Greenfield Microservices: Introduction

- New project
- Evolving requirements
- Business domain
  - Not fully understood
  - Getting domain experts involved
  - System boundaries will evolve
- Teams experience
  - First microservice
  - Experienced with microservices
- Existing system integration
  - Monolithic system
  - Established microservices architecture
- Push for change
  - Changes to apply microservice principles

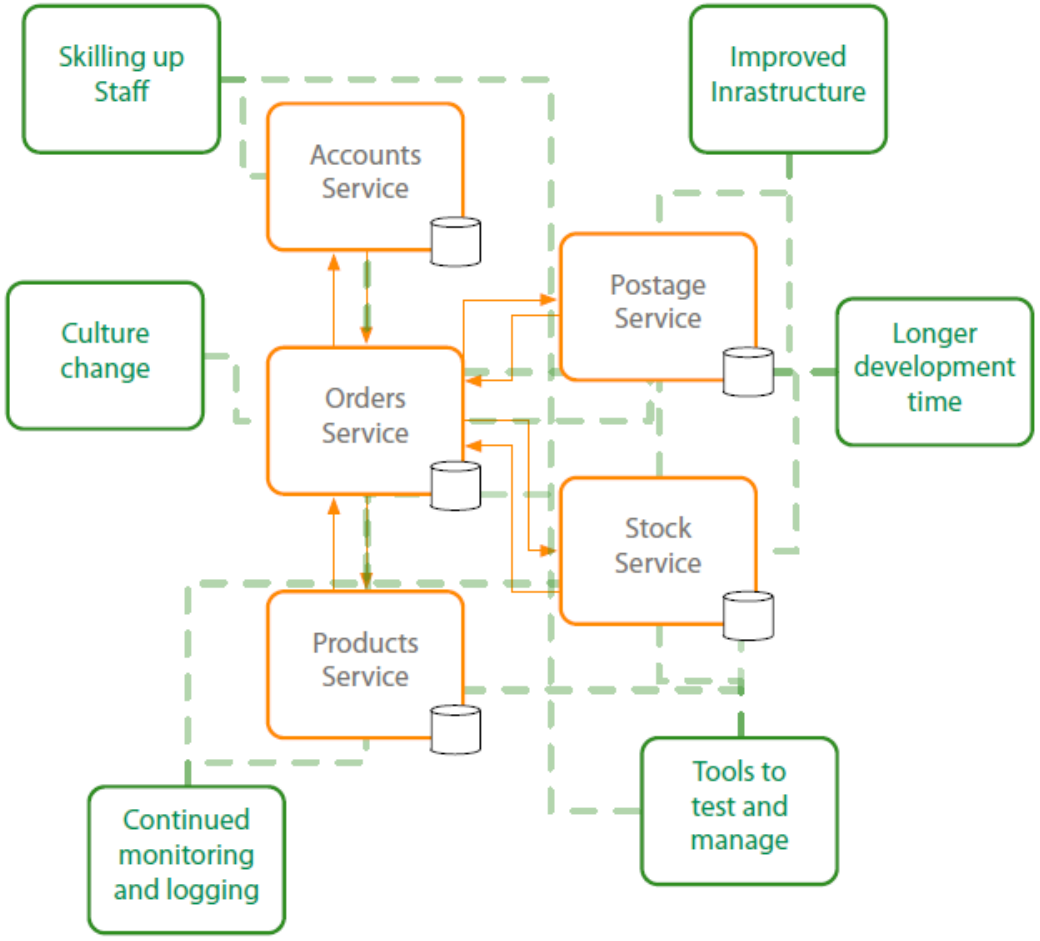
Greenfield Microservices



Greenfield Microservices: Approach

- Start off with monolithic design
  - High level
  - Evolving seams
  - Develop areas into modules
  - Boundaries start to become clearer
  - Refine and refactor design
  - Split further when required
- Modules become services
- Shareable code libraries promote to service
- Review microservice principles at each stage
- Prioritise by
  - Minimal viable product
  - Customer needs and demand

Microservices Provisos



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

THANK YOU

Capgemini

CONSULTING.TECHNOLOGY.OUTSOURCING

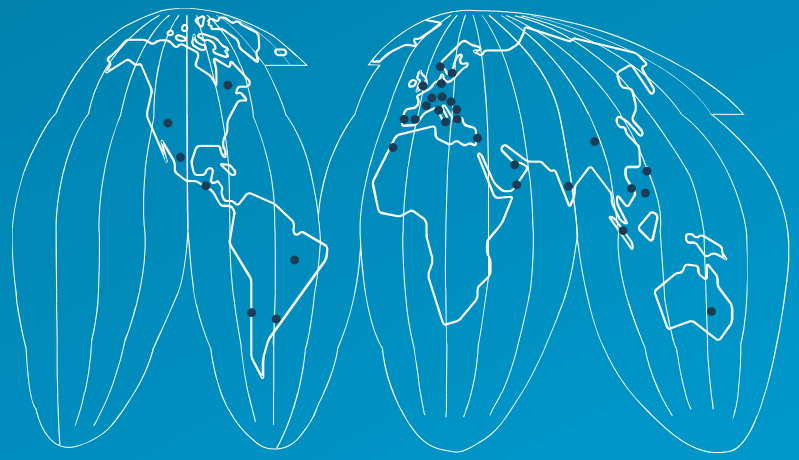
People matter, results count.

About Capgemini

With more than 145,000 people in 40 countries, Capgemini is one of the world's foremost providers of consulting, technology and outsourcing services. The Group reported 2014 global revenues of EUR 10.5 billion.

Together with its clients, Capgemini creates and delivers business and technology solutions that fit their needs and drive the results they want. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Rightshore® is a trademark belonging to Capgemini



www.capgemini.com