Effective Microservices in a Data-centric World

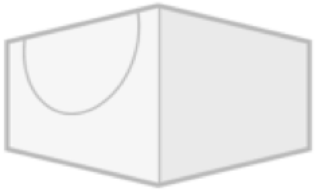
Randy Shoup
@randyshoup
linkedin.com/in/randyshoup

Background

- VP Engineering at Stitch Fix
  - Revolutionizing retail by combining "Art and Science"
- Consulting "CTO as a service"
  - Helping companies scale their organizations and technology
- Director of Engineering for Google App Engine
  - World's largest Platform-as-a-Service
- Chief Engineer at eBay
  - Multiple generations of eBay's infrastructure

Stitch Fix

Create Your Style Profile.

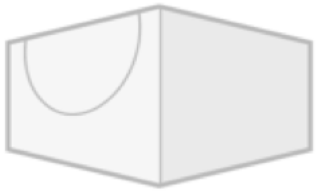


Get Five Hand-picked Items.

Keep What You Like.
Send Back the Rest.

Stitch Fix

Create Your Style Profile.



Get Five Hand-picked Items.

Keep What You Like.
Send Back the Rest.

How do you prefer clothes to fit the top half of your body?

- Mostly Tight / Form Fitting
- Prefer Fitted / Showing my Figure
- Straight
- Mostly Loose
- Oversized

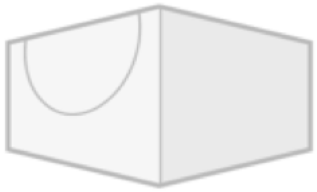
ear for pants, jeans, and skirts?

an waist

Skirts L

Stitch Fix

Create Your Style Profile.

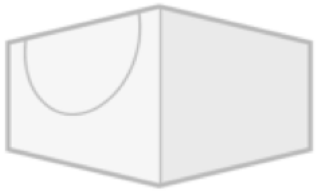


Get Five Hand-picked Items.

Keep What You Like.
Send Back the Rest.

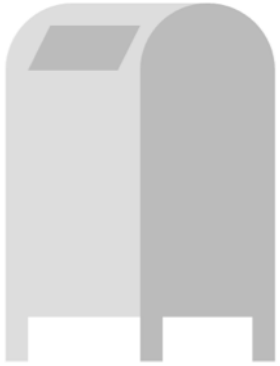
Stitch Fix

Create Your Style Profile.



Get Five Hand-picked Items.

Keep What You Like.
Send Back the Rest.

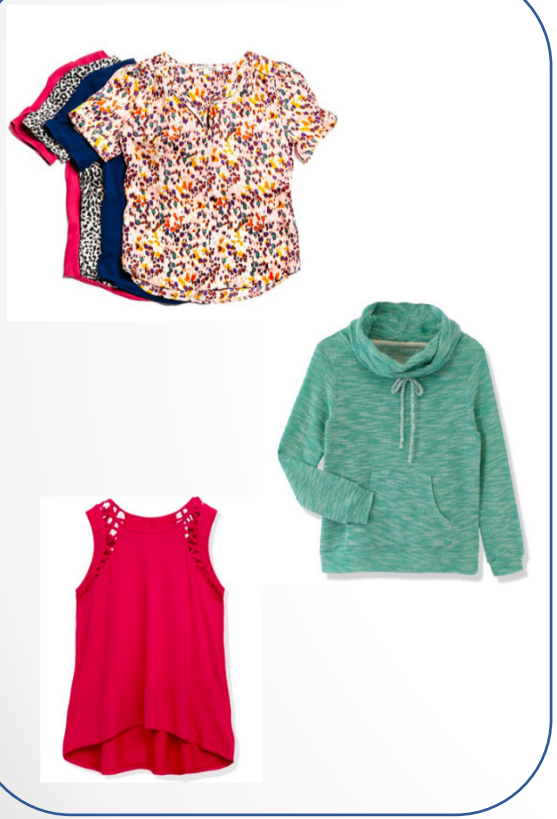


Combining Art and [Data] Science

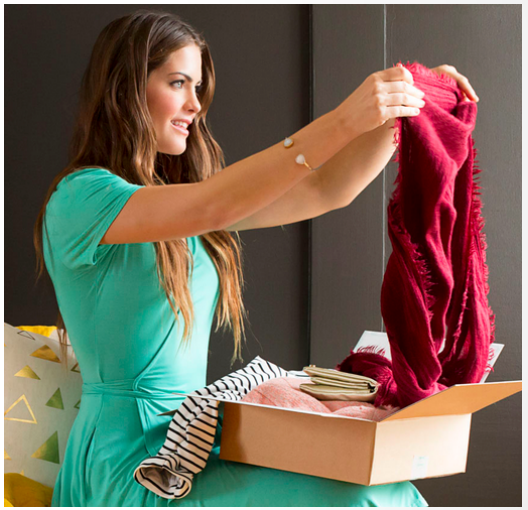
- 1:1 Ratio of Data Science to Engineering
  - >70 software engineers
  - >70 data scientists and algorithm developers
  - Unique in our industry?
- Apply intelligence to *every* part of the business
  - Buying
  - Inventory management
  - Logistics optimization
  - Styling recommendations
  - Demand prediction
- Humans and machines augmenting each other

Styling at Stitch Fix

Inventory

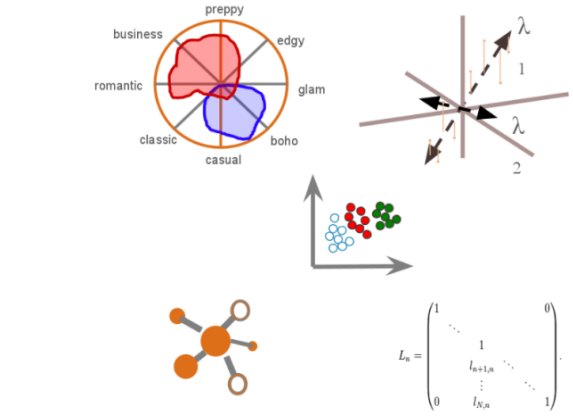


Personal styling



Personalized Recommendations

Inventory

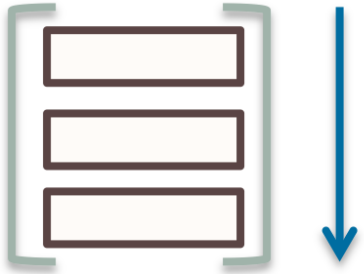


Machine learning

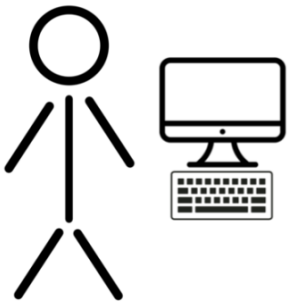
Algorithmic recommendations

Expert Human Curation

Algorithmic recommendations



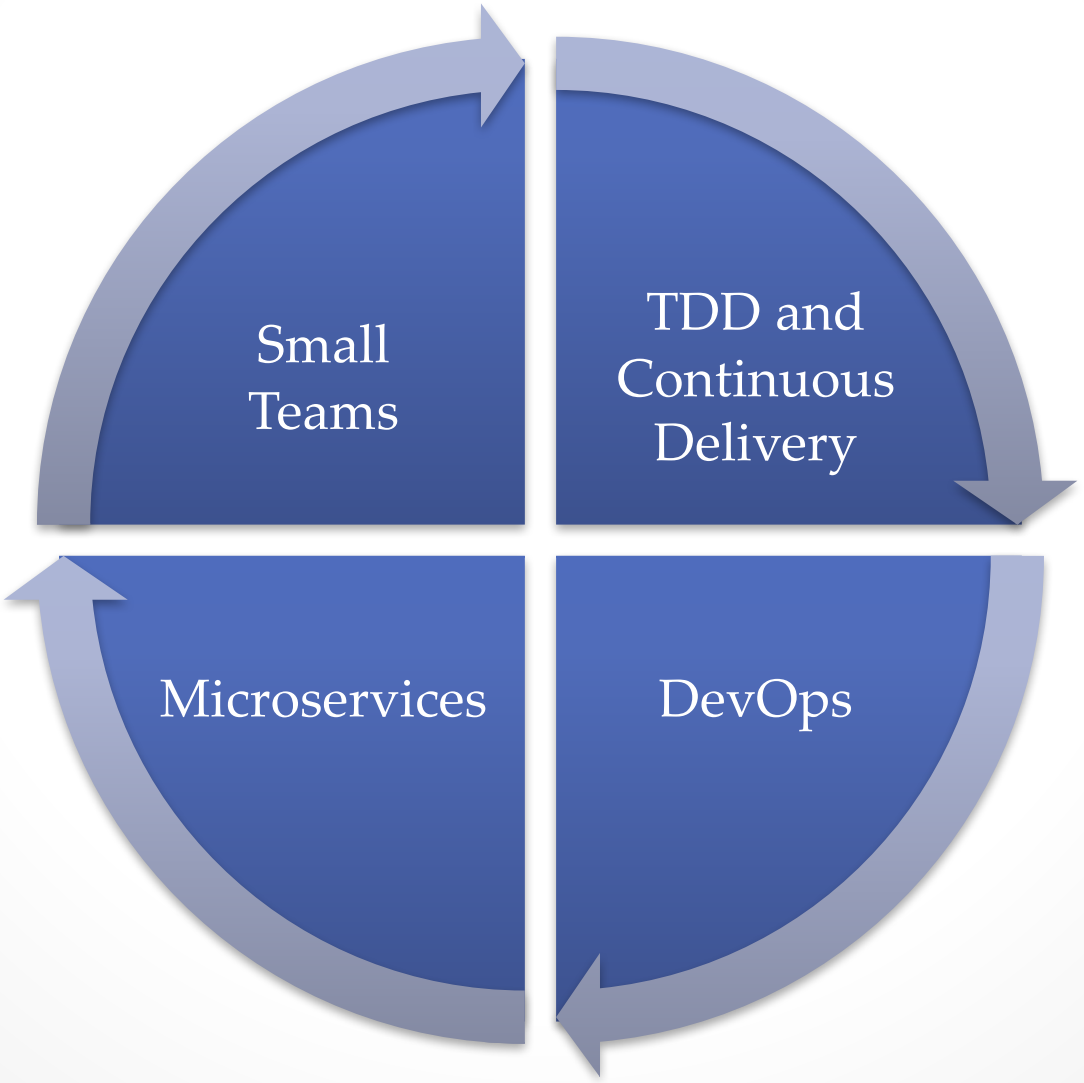
Human curation



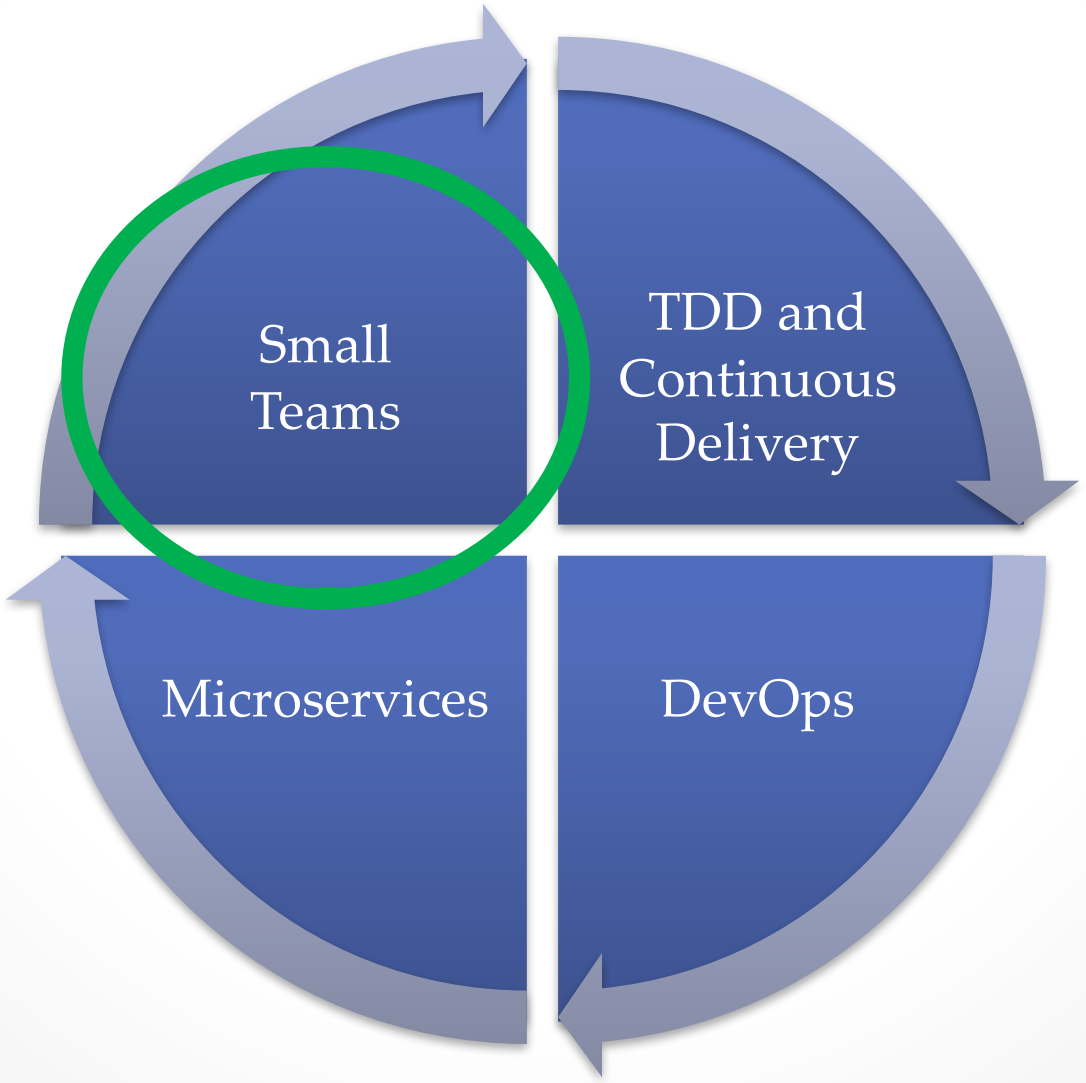
How do we work, and why does it work?

Modern Software Development

Modern Software Development



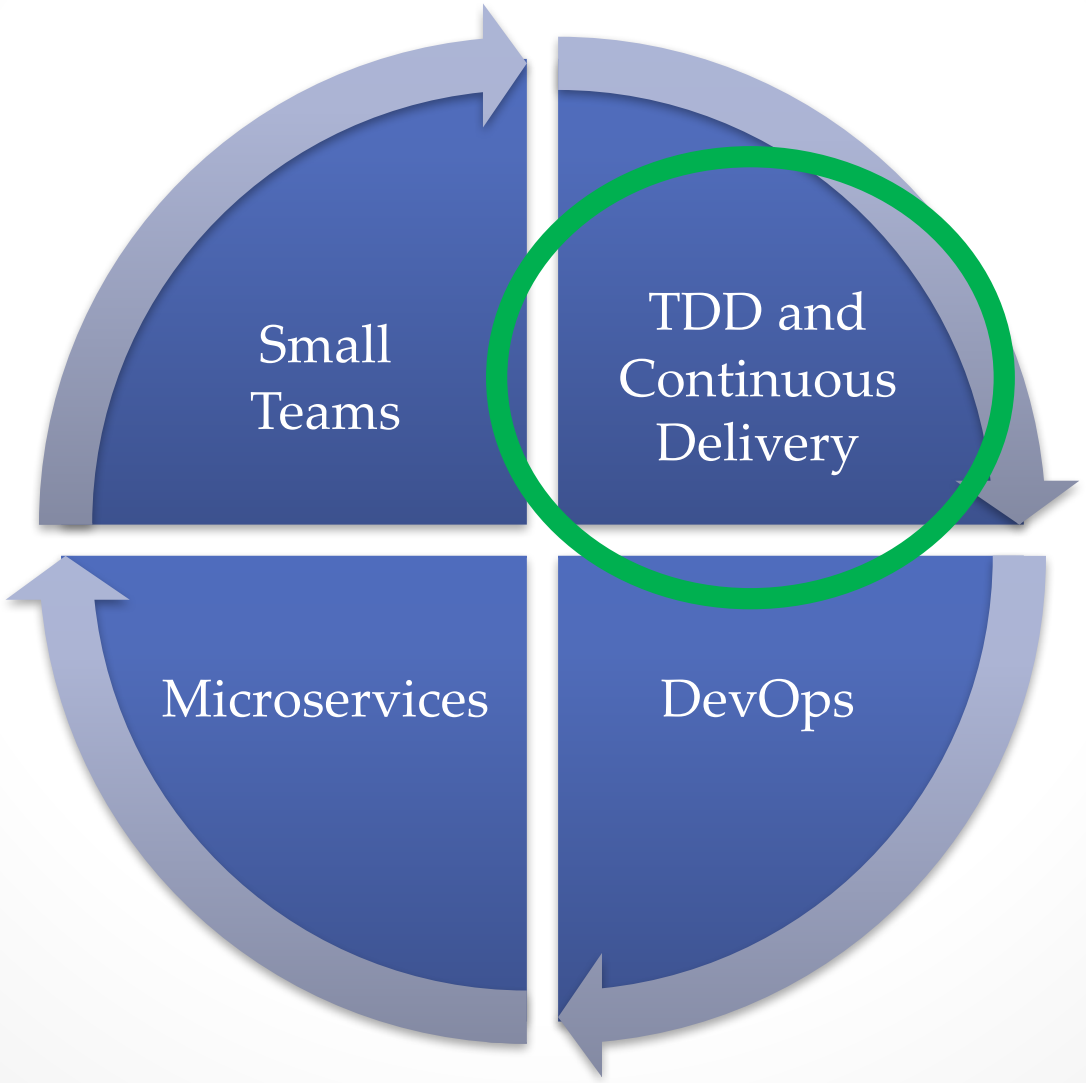
Modern Software Development



Small "Service" Teams

- Teams Aligned to Domains
  - Clear, well-defined area of responsibility
  - Single service or set of related services
- Cross-functional Teams
  - Team has inside it all skill sets needed to do the job
- Depend on other teams for supporting services, libraries, and tools

Modern Software Development



Test-Driven Development

- Tests help you go faster
  - Tests "have your back"
  - Development velocity
- Tests make better code
  - Confidence to break things
  - Confidence to refactor
- Tests make better systems
  - Catch bugs earlier, fail faster

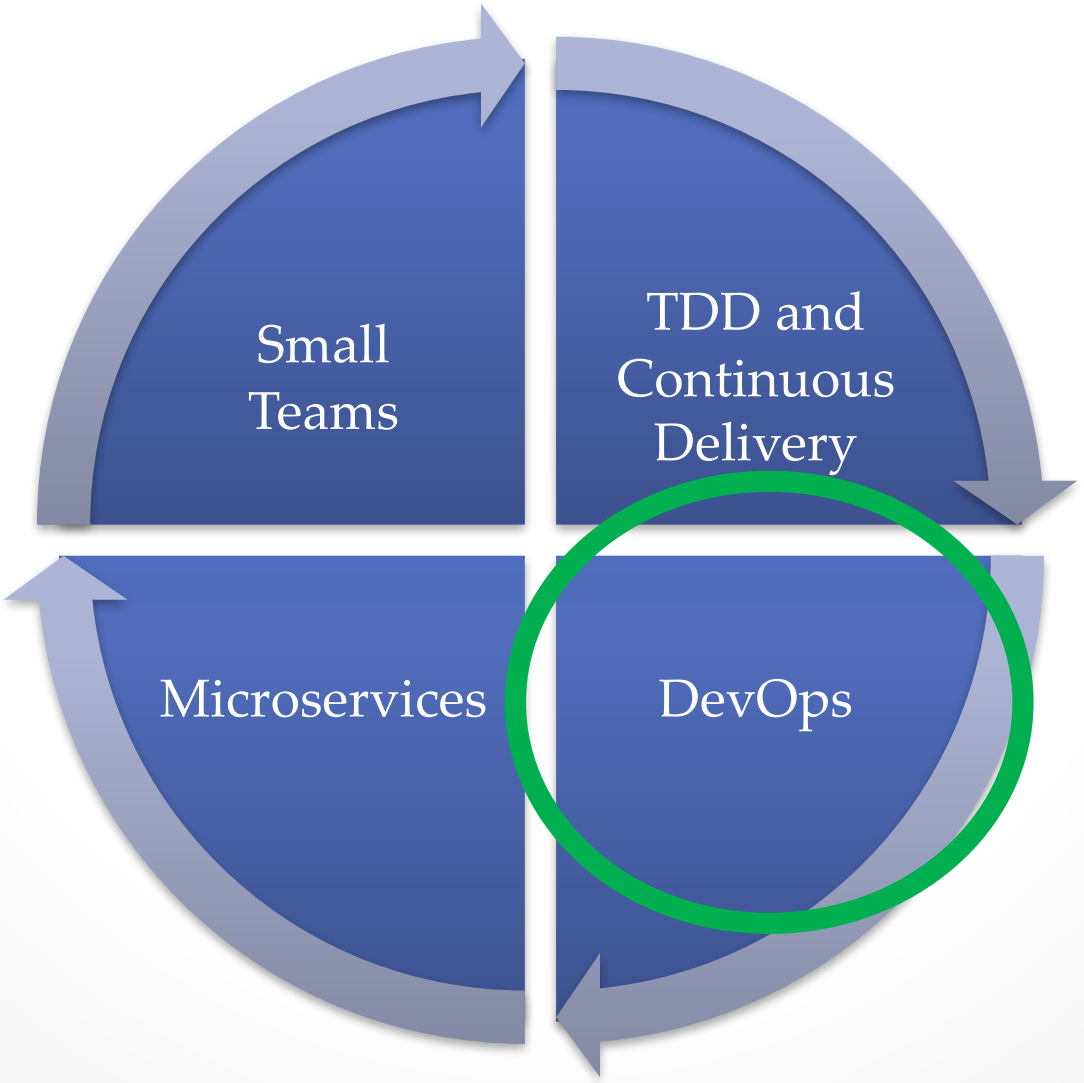
Test-Driven Development

- "Don't have time to do it right" ?
  - WRONG ☺ – Don't have time to do it twice (!)
- Do it right (enough) the first time
  - The more constrained you are on time and resources, the more important it is to build solid features
  - Right != perfect
- ➔ Basically no bug tracking system (!)
  - Bugs are fixed as they come up
  - Backlog contains features we want to build
  - Backlog contains technical debt we want to repay

Continuous Delivery

- Most applications deployed multiple times per day
- More solid systems
  - Release smaller units of work
  - Faster to repair, easier to diagnose
  - Smaller changes to roll back or roll forward
- Enables experimentation
  - Small experiments and rapid iteration are cheap

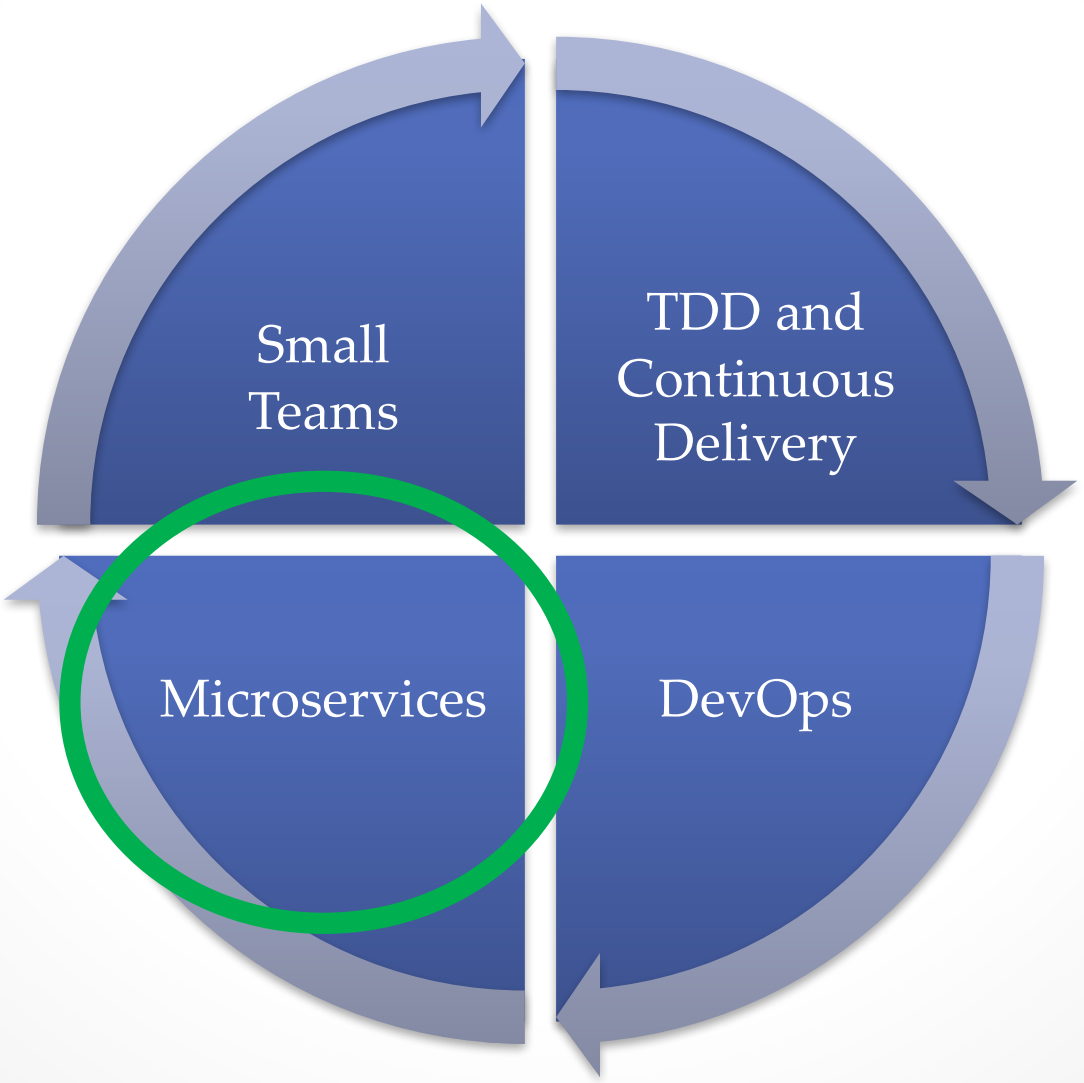
Modern Software Development



DevOps

- End-to-end Ownership
  - Team owns service from design to deployment to retirement
- Responsible for
  - Features
  - Quality
  - Performance
  - Operations
  - Maintenance
- “You build it, you run it!”

Modern Software Development

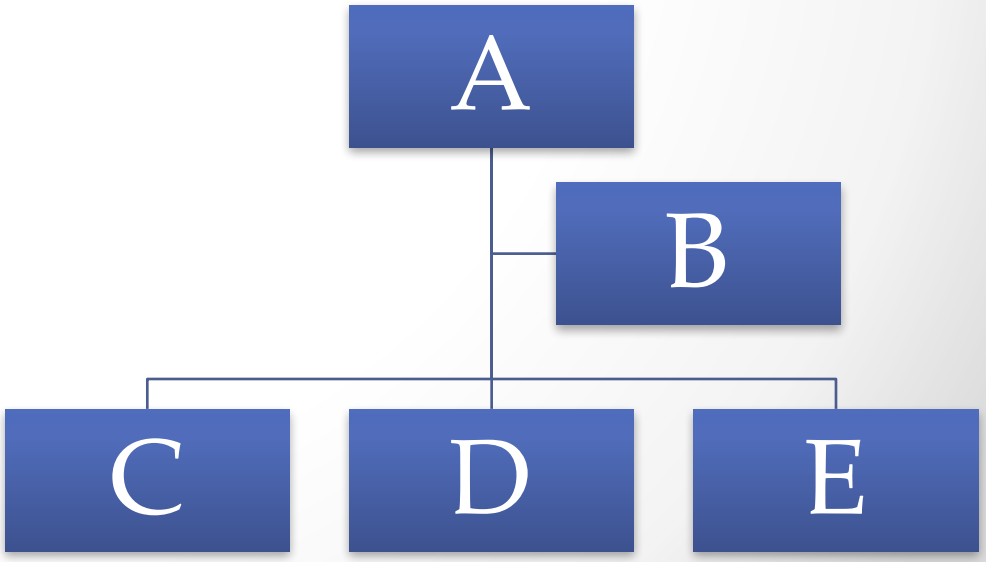


Architecture Evolution

- eBay
  - 5th generation today
  - Monolithic Perl → Monolithic C++ → Java → microservices
- Twitter
  - 3rd generation today
  - Monolithic Rails → JS / Rails / Scala → microservices
- Amazon
  - Nth generation today
  - Monolithic Perl / C++ → Java / Scala → microservices

Microservices

- Single-purpose
- Simple, well-defined interface
- Modular and independent

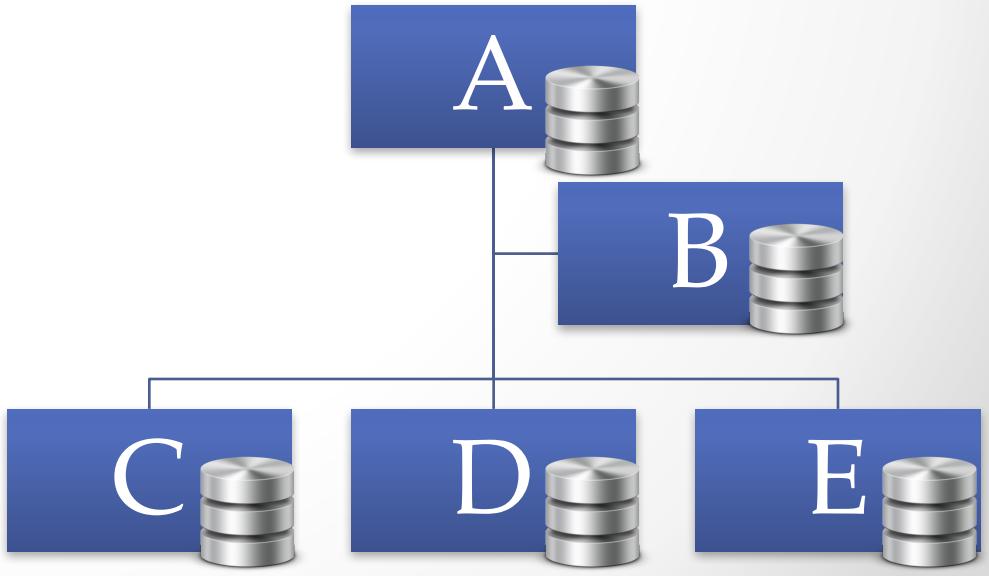


Microservices are nothing more than SOA done properly.

-- me

Microservices

- Single-purpose
- Simple, well-defined interface
- Modular and independent
- **Isolated persistence (!)**



Microservice Persistence

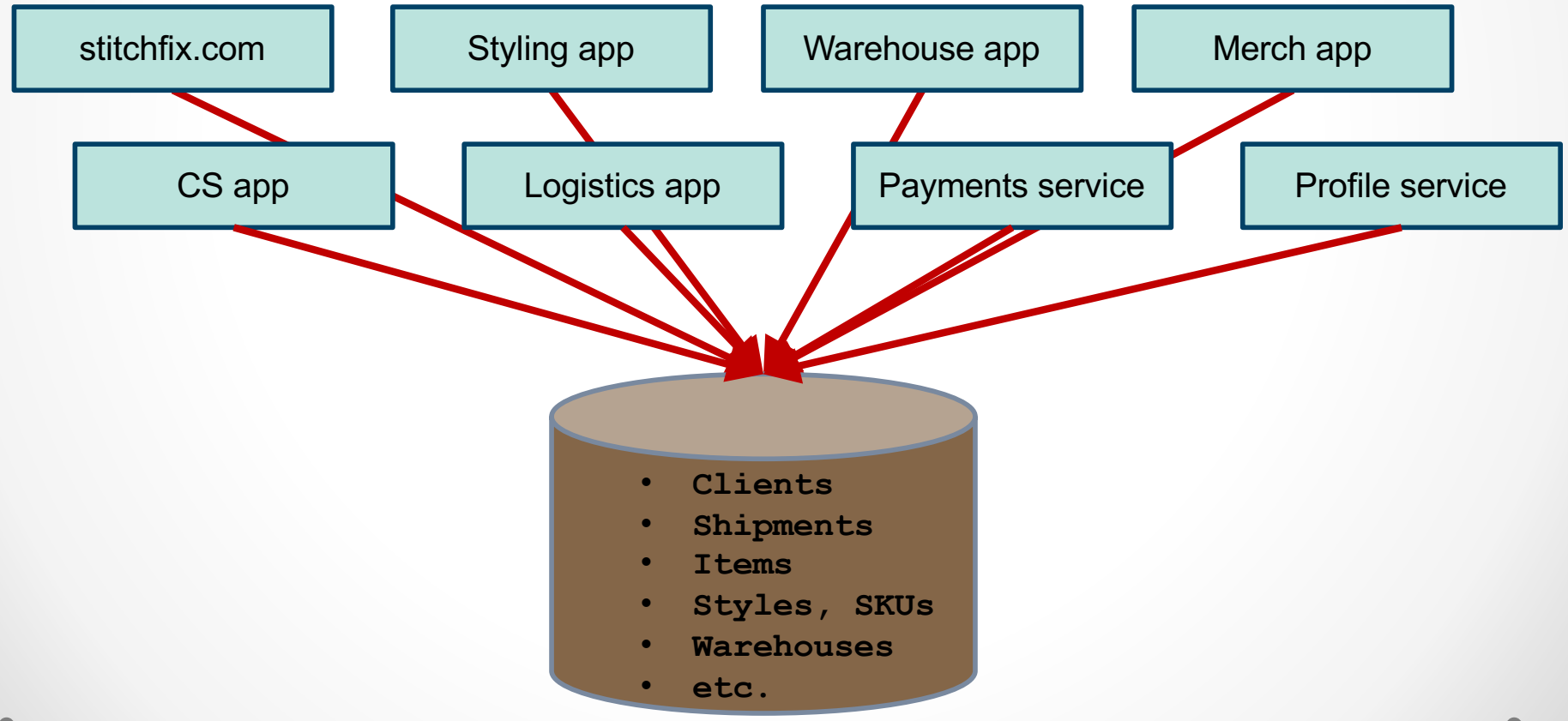
- Approach 1: Operate your own data store
  - Store to your own instance(s) of {Postgres, MySQL, etc.}, owned and operated by the service team
- Approach 2: Use a persistence service
  - Store to your own table(s) in {Dynamo, RDS, Spanner, etc.}, operated as a service by another team or by a third-party provider
  - Isolated from all other users of the service
- ➔ Only external access to data store is through published service interface

Maintaining Interface Stability

- Backward / forward compatibility of interfaces
  - Can *never* break your clients' code
- Semantic versioning (*major.minor.patch*)
  - Often multiple interface versions
  - Sometimes multiple deployments
- Explicit deprecation policy
  - Strong incentive to wean customers off old versions (!)

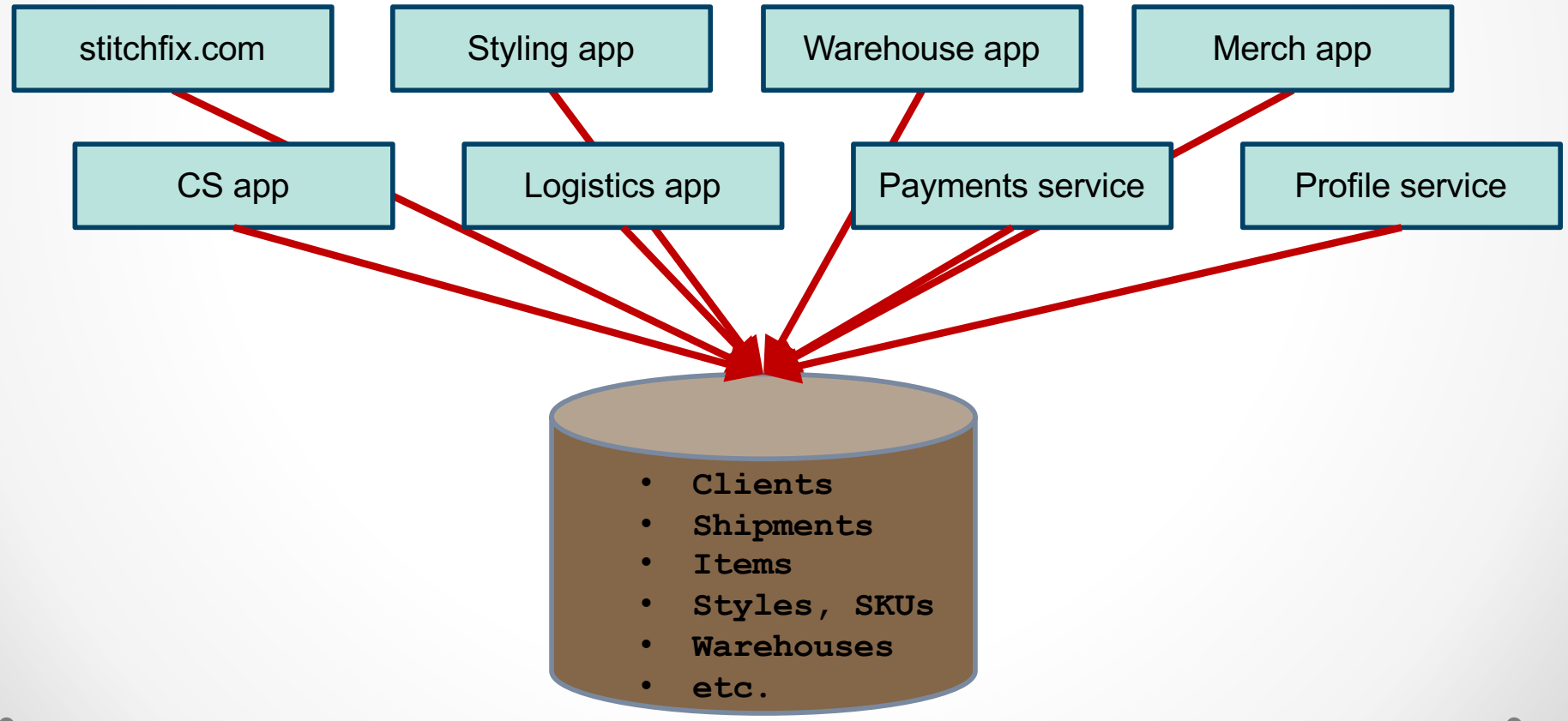
Extracting Microservices

- Problem: Monolithic shared DB



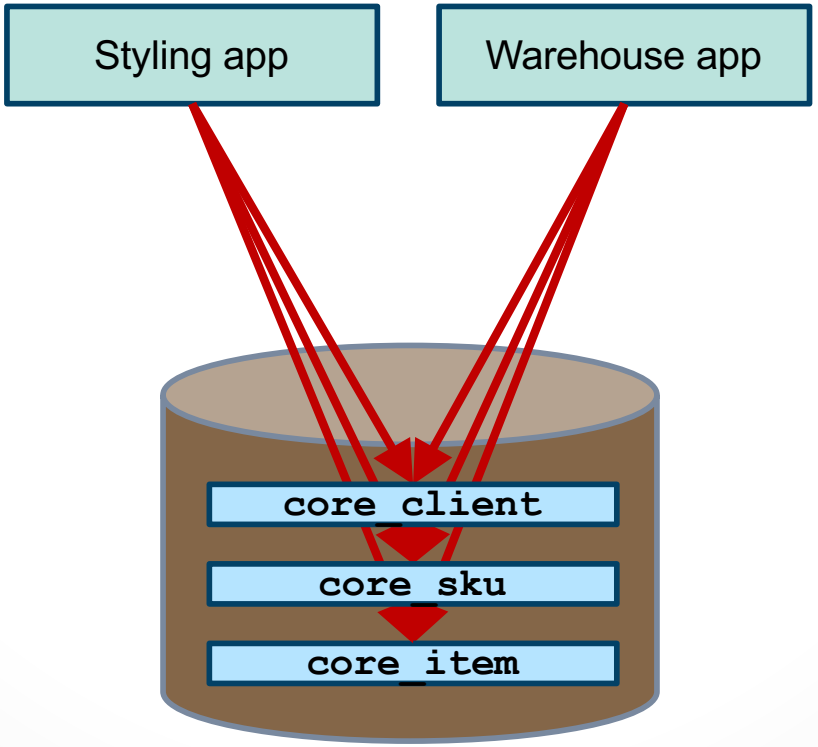
Extracting Microservices

- Decouple applications / services from shared DB



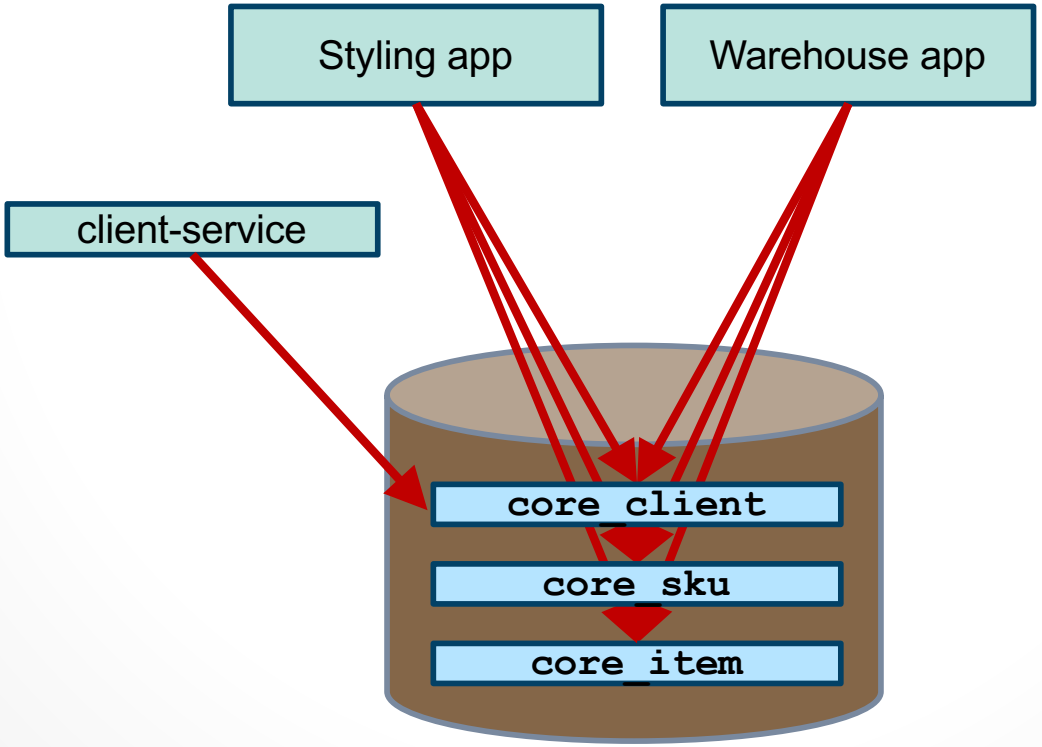
Extracting Microservices

- Decouple applications / services from shared DB



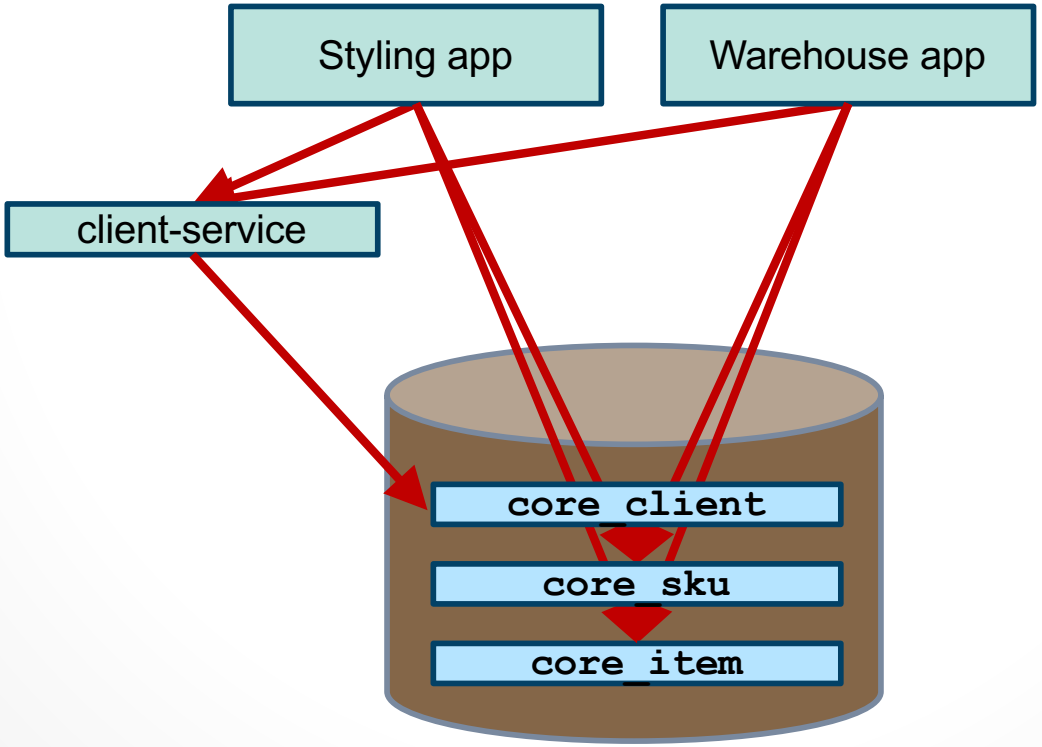
Extracting Microservices

- Step 1: Create a service



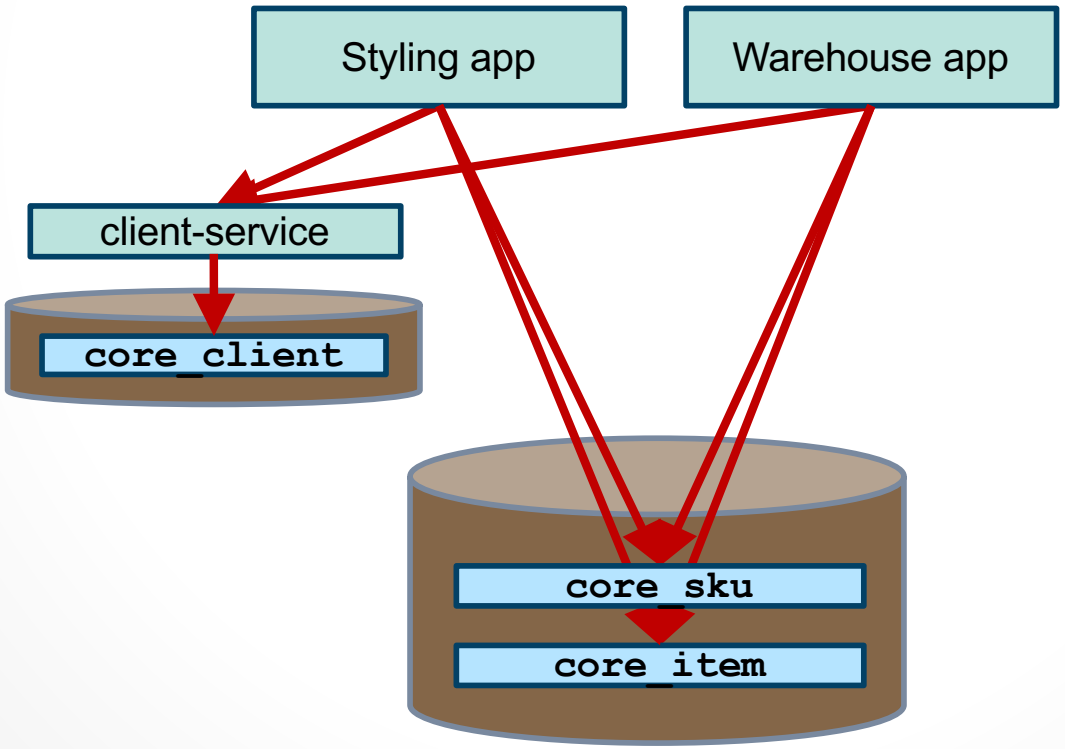
Extracting Microservices

- Step 2: Applications use the service



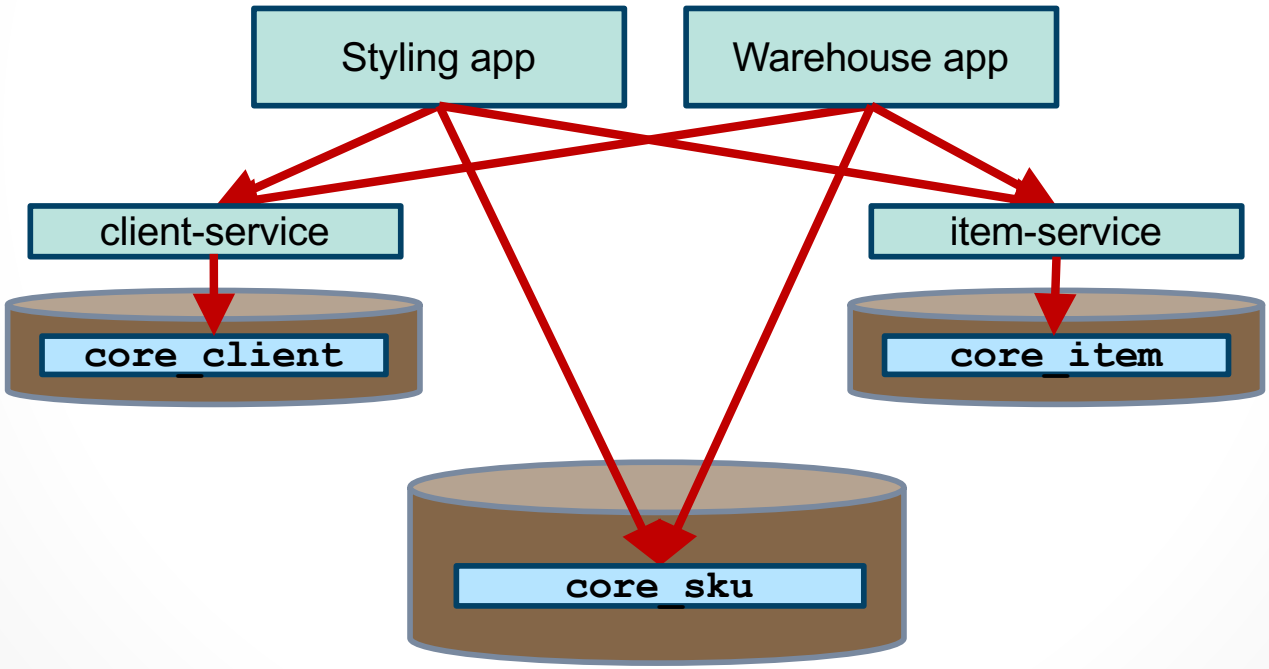
Extracting Microservices

- Step 3: Move data to private database



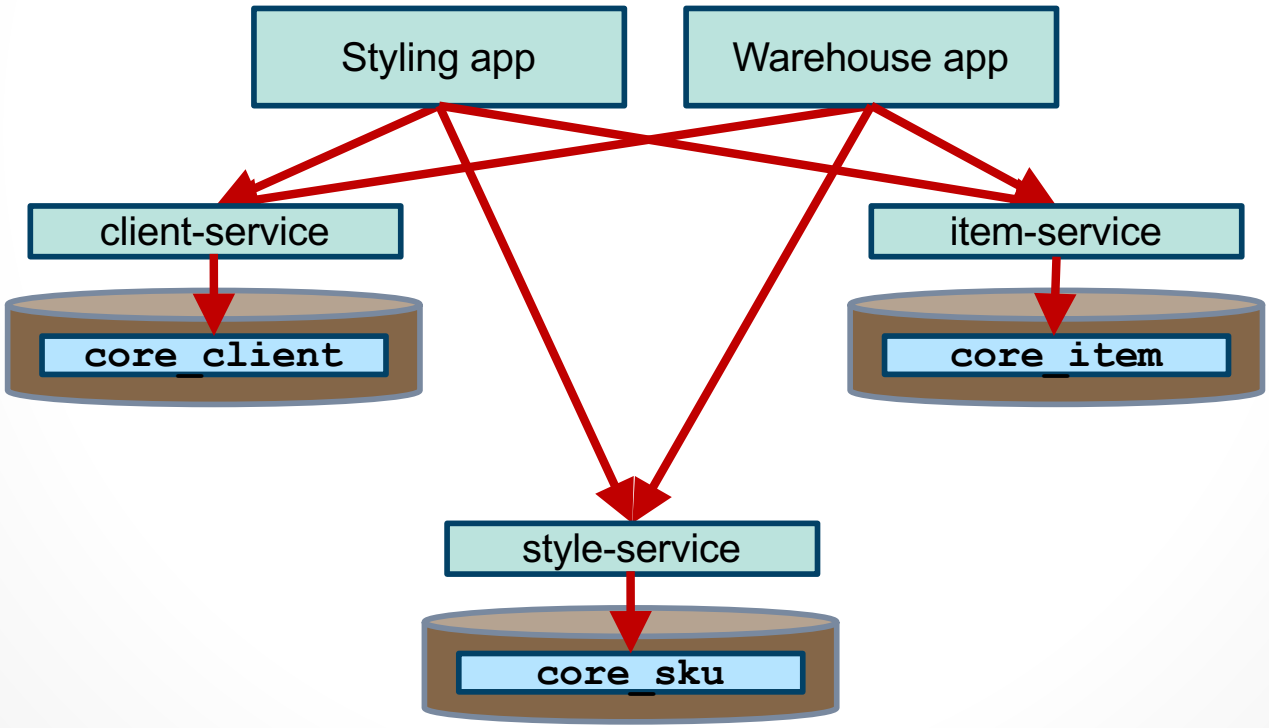
Extracting Microservices

- Step 4: Rinse and Repeat



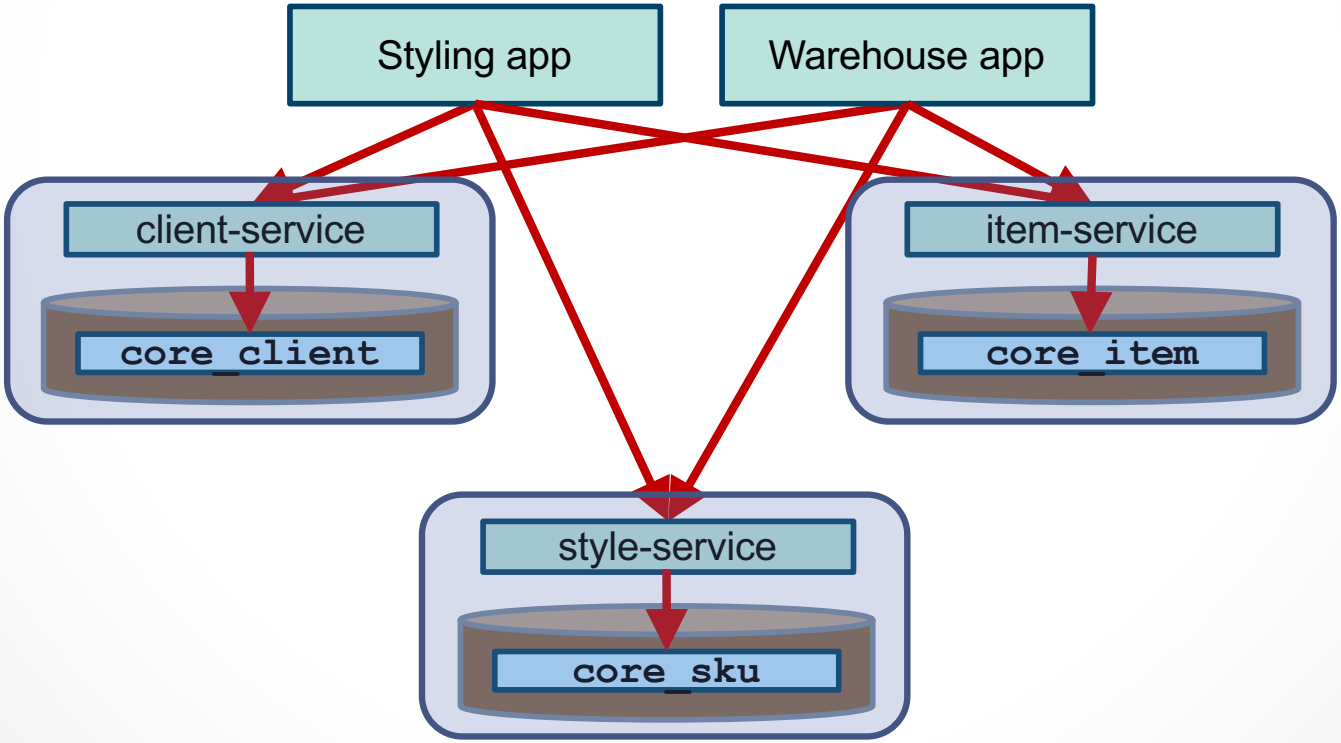
Extracting Microservices

- Step 4: Rinse and Repeat



Extracting Microservices

- Step 4: Rinse and Repeat

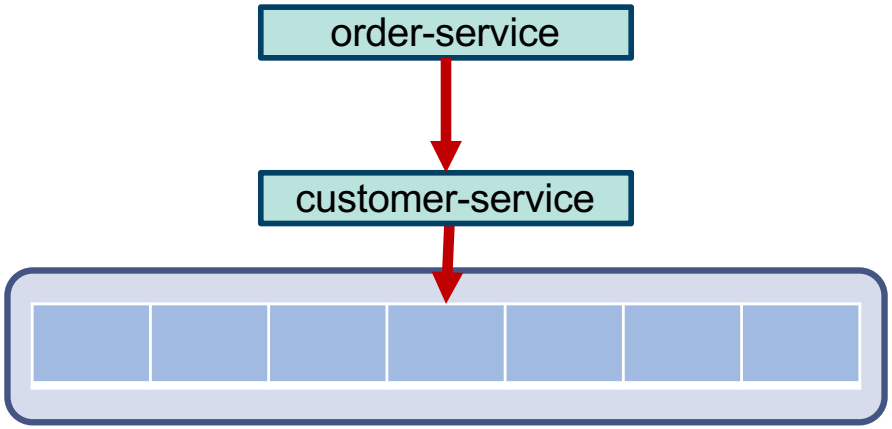


Microservice Techniques: Shared Data

- Problem
  - Monolithic database makes it easy to leverage shared data
  - Where does shared data go in a microservices world?
- Principle: Single System of Record
  - Every piece of data is owned by a single service
  - That service represents its canonical system of record
  - Every other copy of that data is a read-only, non-authoritative cache

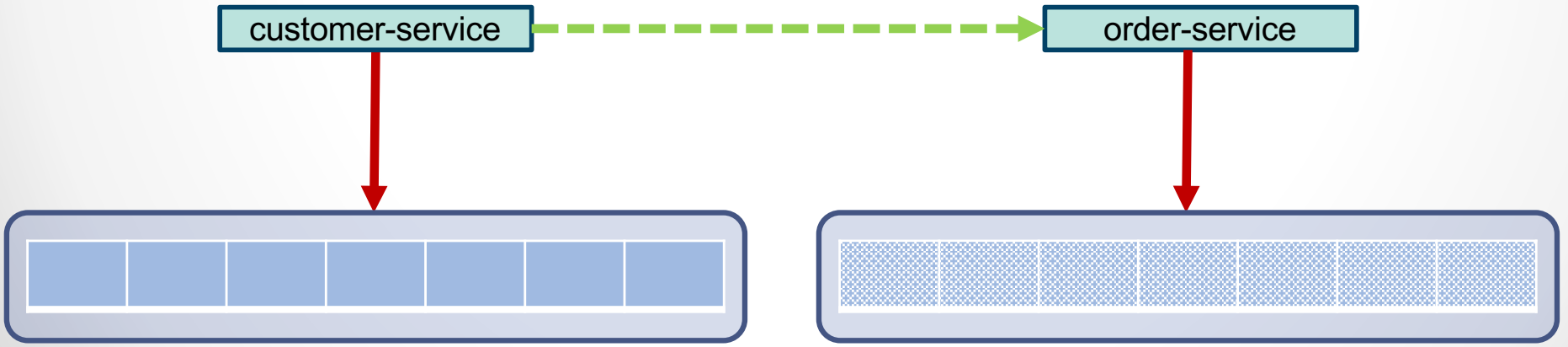
Microservice Techniques: Shared Data

- Approach 1: Synchronous Lookup
  - Customer service owns customer data
  - Order service calls customer service in real time



Microservice Techniques: Shared Data

- Approach 2: Async event + local cache
  - Customer service owns customer data
  - Customer service sends `customer-updated` event when customer changes
  - Order service caches current customer information



Microservice Techniques: Shared Data

- Approach 3: Shared metadata library
  - Read-only metadata, basically immutable
  - E.g., size schemas, colors, fabrics, US States, etc.

Events as First-Class Construct

- "A significant change in state"
  - Statement that some interesting thing occurred
  - 0 | 1 | N consumers subscribe to the event, typically asynchronously
- Fourth fundamental building block
  - Presentation ➔ interface / interaction
  - Application ➔ stateless business logic
  - Persistence ➔ database
  - **State changes ➔ events**
- Events represent how the real world works
  - Finance, software development process, any "workflow"

Microservices and Events

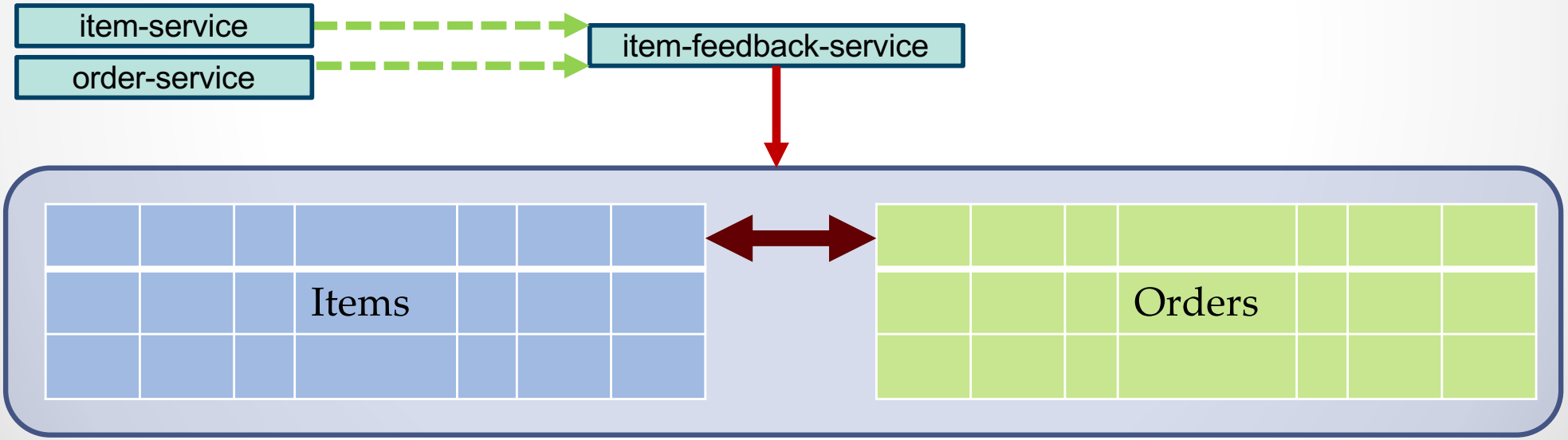
- Events are a first-class part of the interface
- A service interface includes
  - Synchronous request-response (REST, gRPC, etc)
  - Events the service produces
  - Events the service consumes
  - Bulk reads and writes (ETL)
- The interface includes *any mechanism for getting data in or out of the service (!)*

Microservice Techniques: Joins

- Problem
  - Monolithic database makes joins very easy
  - Splitting the data into separate services makes joins very hard

Microservice Techniques: Joins

- Approach 1: Service that "Materializes the View"
  - Listen to events from `item-service`, events from `order-service`
  - Maintain denormalized join of items and orders together in local storage



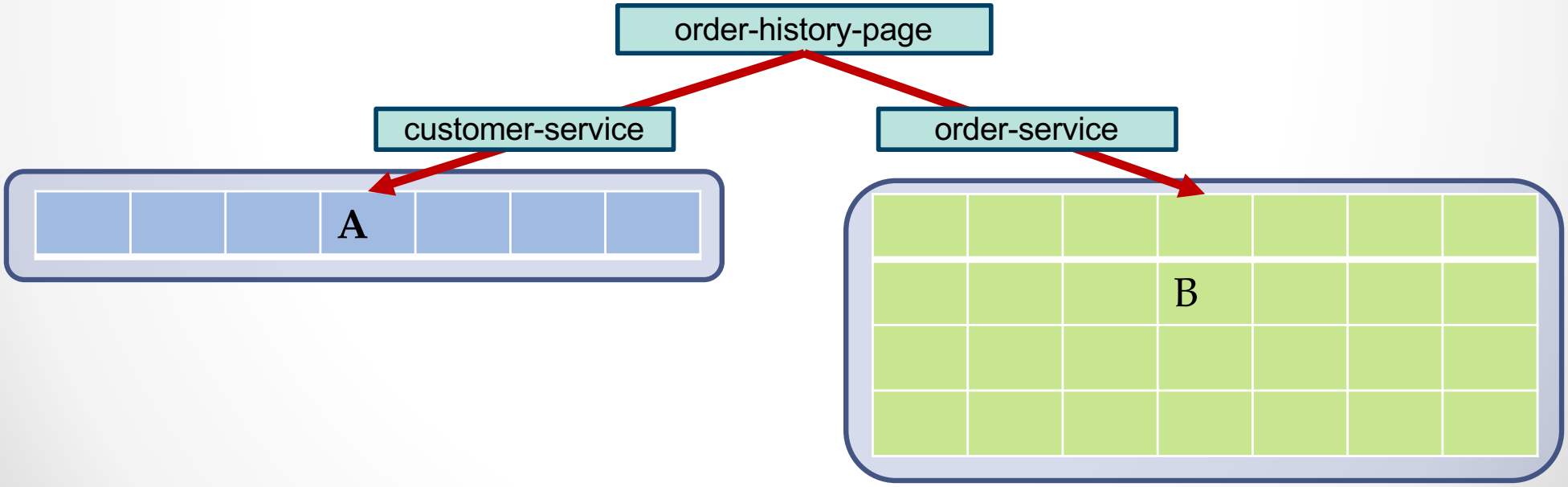
- Best for high cardinality A and high cardinality B (M:N join)

Microservice Techniques: Joins

- Many common systems do this
  - Most NoSQL approaches
  - "Materialized view" in database systems
  - Search engines
  - Analytic systems
  - Log aggregators

Microservice Techniques: Joins

- Approach 2: Join in Application / Client
  - Get a single customer from `customer-service`
  - Query matching orders for that customer from `order-service`



- Best for single A, multiple Bs (1:N join)

Microservice Techniques: Joins

- Many common systems do this
  - Web application "mashup"

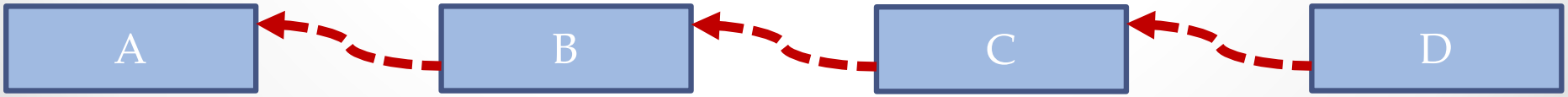
Microservice Techniques: Workflows and Sagas

- Problem
  - Monolithic database makes transactions across multiple entities easy
  - Splitting data across services makes transactions very hard

Microservice Techniques: Workflows and Sagas

- Transaction ➔ Saga
  - Model the transaction as a state machine of atomic events
- Reimplement as a workflow

- Roll back by applying compensating operations in reverse

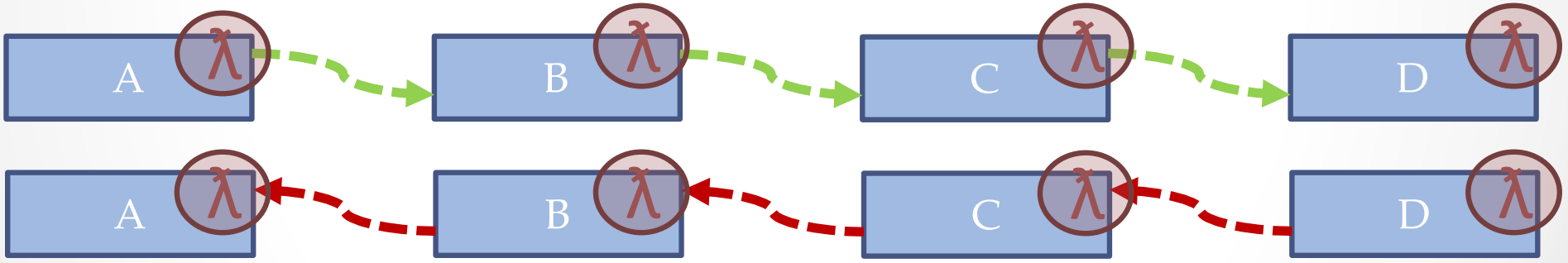


Microservice Techniques: Workflows and Sagas

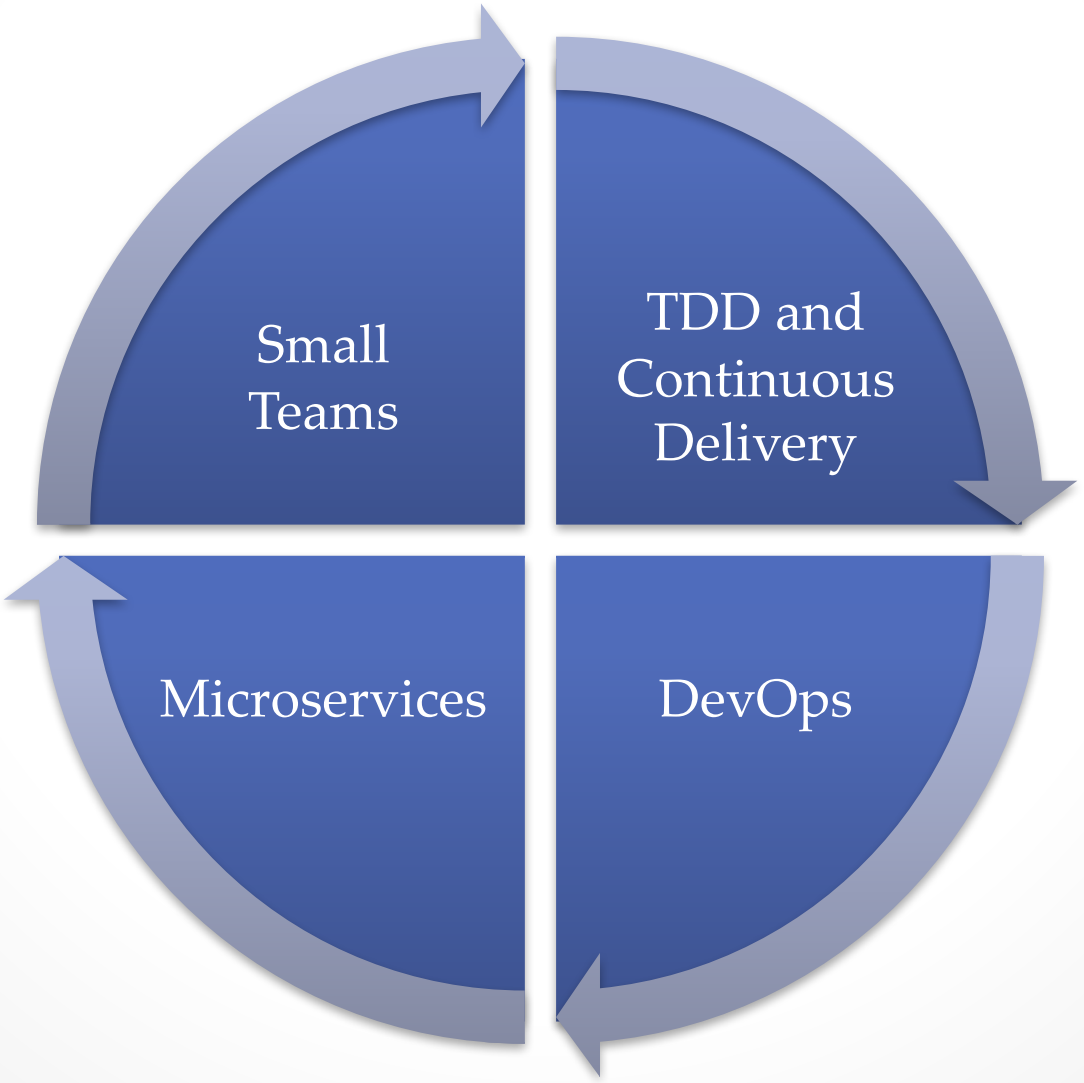
- Many common systems do this
  - Payment processing flow
  - Most approval workflows

Microservice Techniques: Workflows and Sagas

- Ideal use for Functions as a Service ("Serverless")
  - Very lightweight logic
  - Stateless
  - Triggered by an event



Modern Software Development



Thanks!

- Stitch Fix is hiring!
  - www.stitchfix.com/careers
  - Application development, Platform engineering, Data Science
  - Based in San Francisco
  - More than half remote, all across US
- Please contact me
  - @randyshoup
  - linkedin.com/in/randyshoup