Scaling Your Architecture with Services and Events

Randy Shoup
@randyshoup

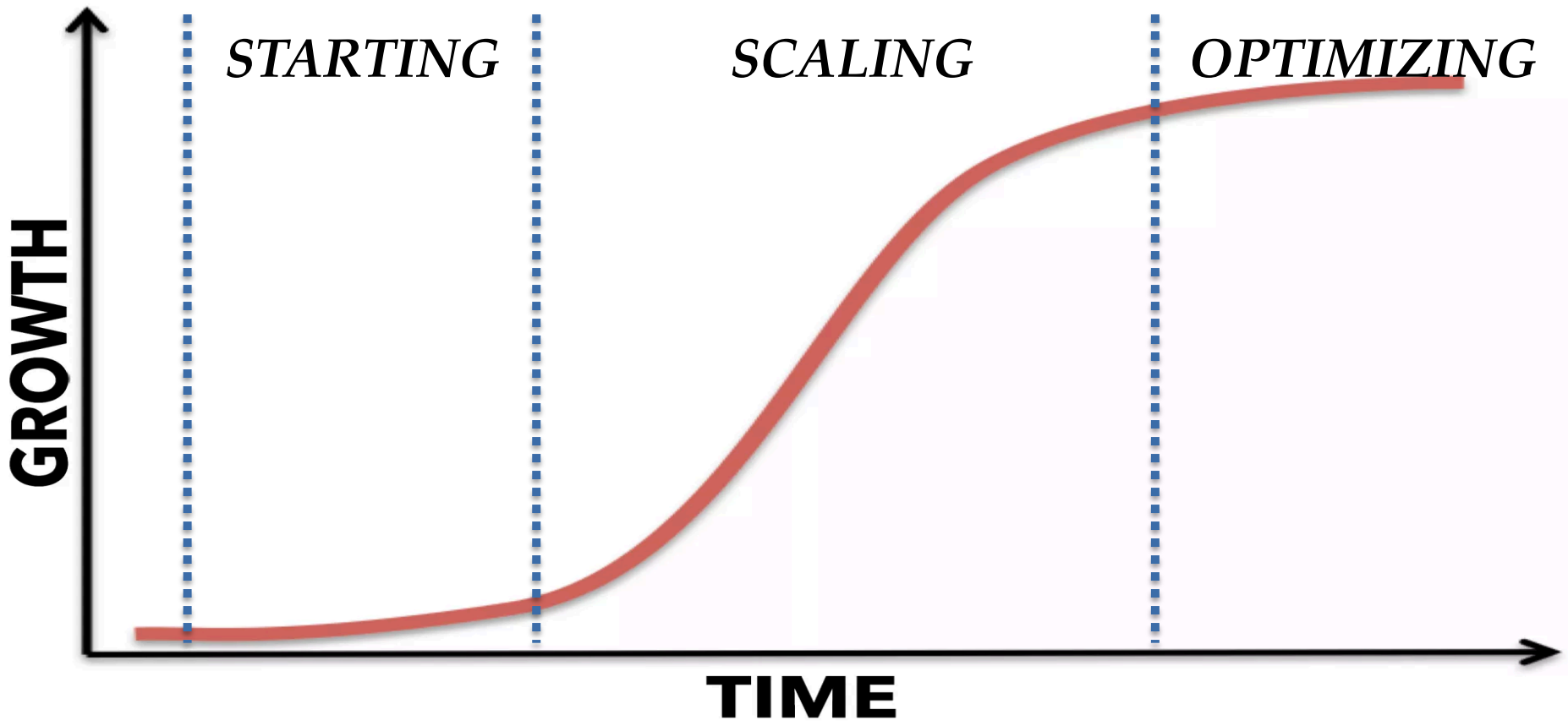
Background

Scaling Architecture

- Architecture Evolution
- Service Architecture
- Event-Driven Communication
- Combining Services + Events

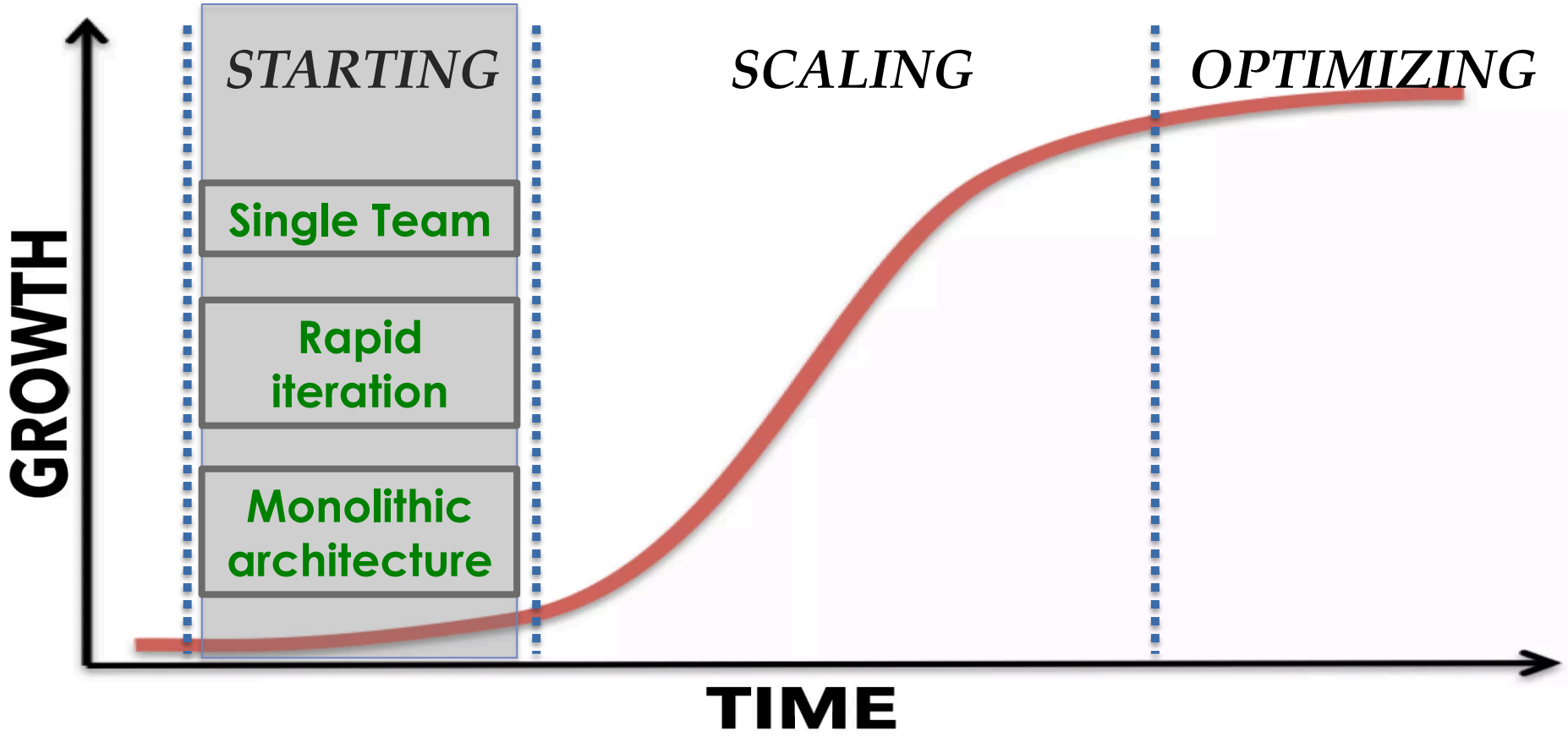
Scaling Architecture

- Architecture Evolution
- Service Architecture
- Event-Driven Communication
- Combining Services + Events



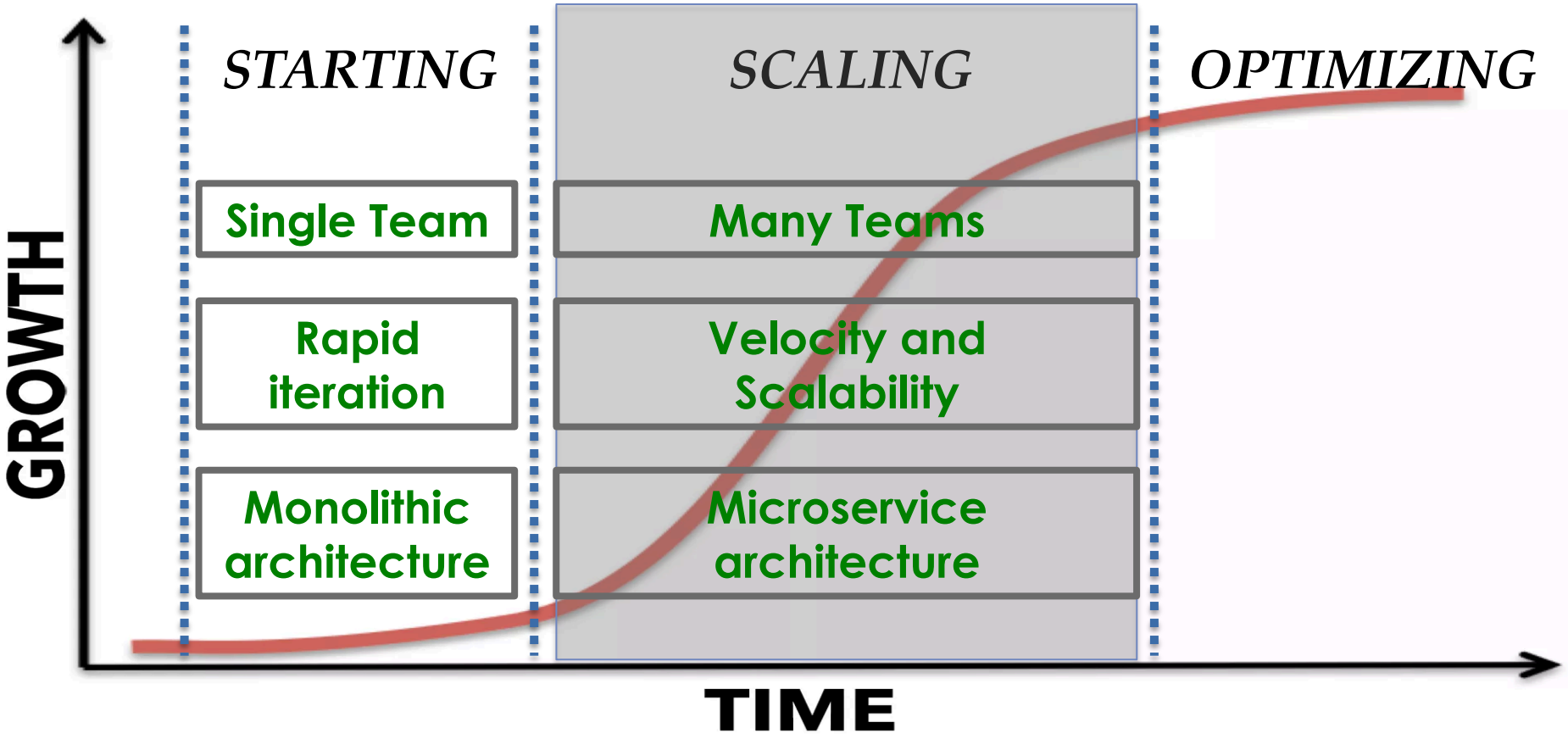
@randyshoup

https://ittybiz.com/s-curve/



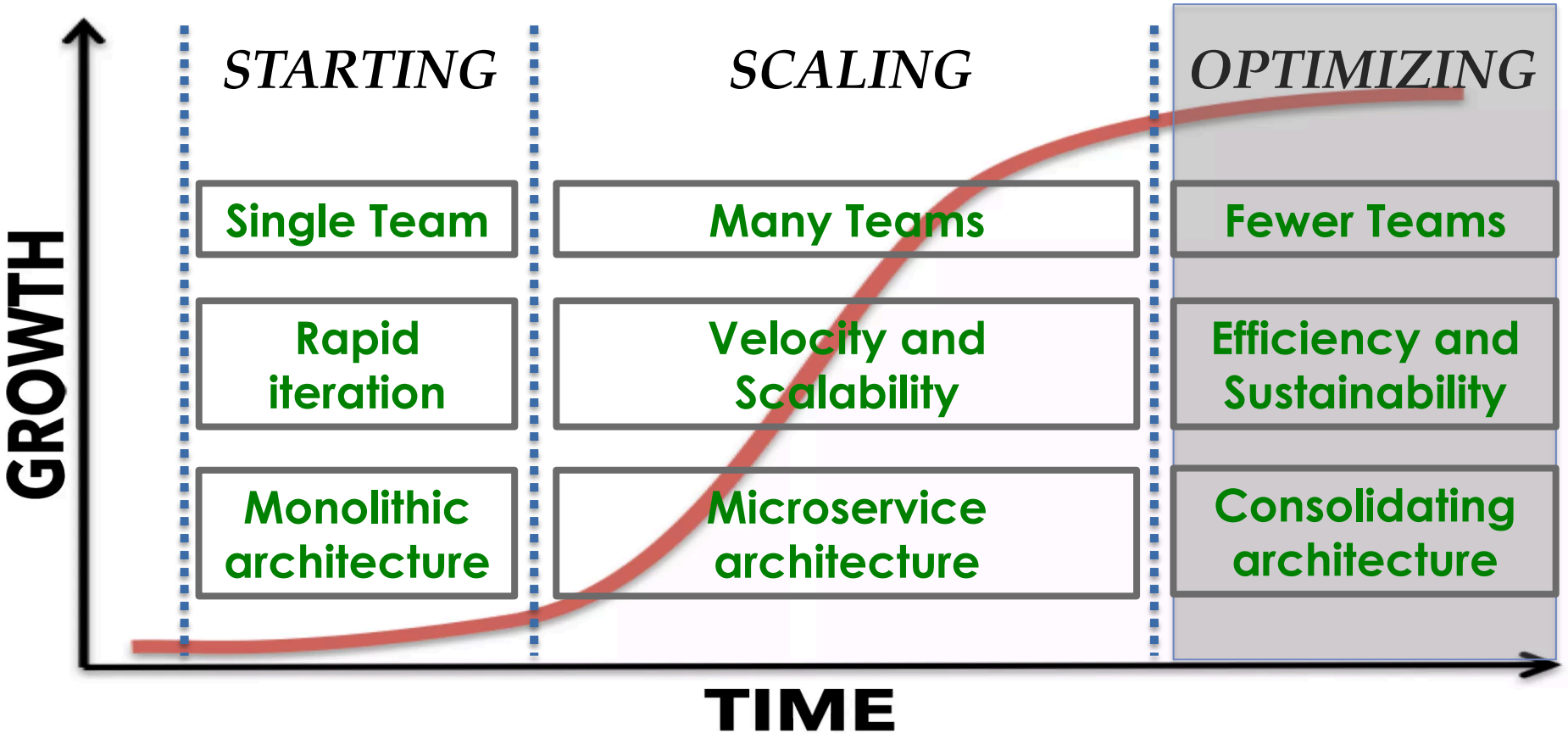
@randyshoup

https://ittybiz.com/s-curve/



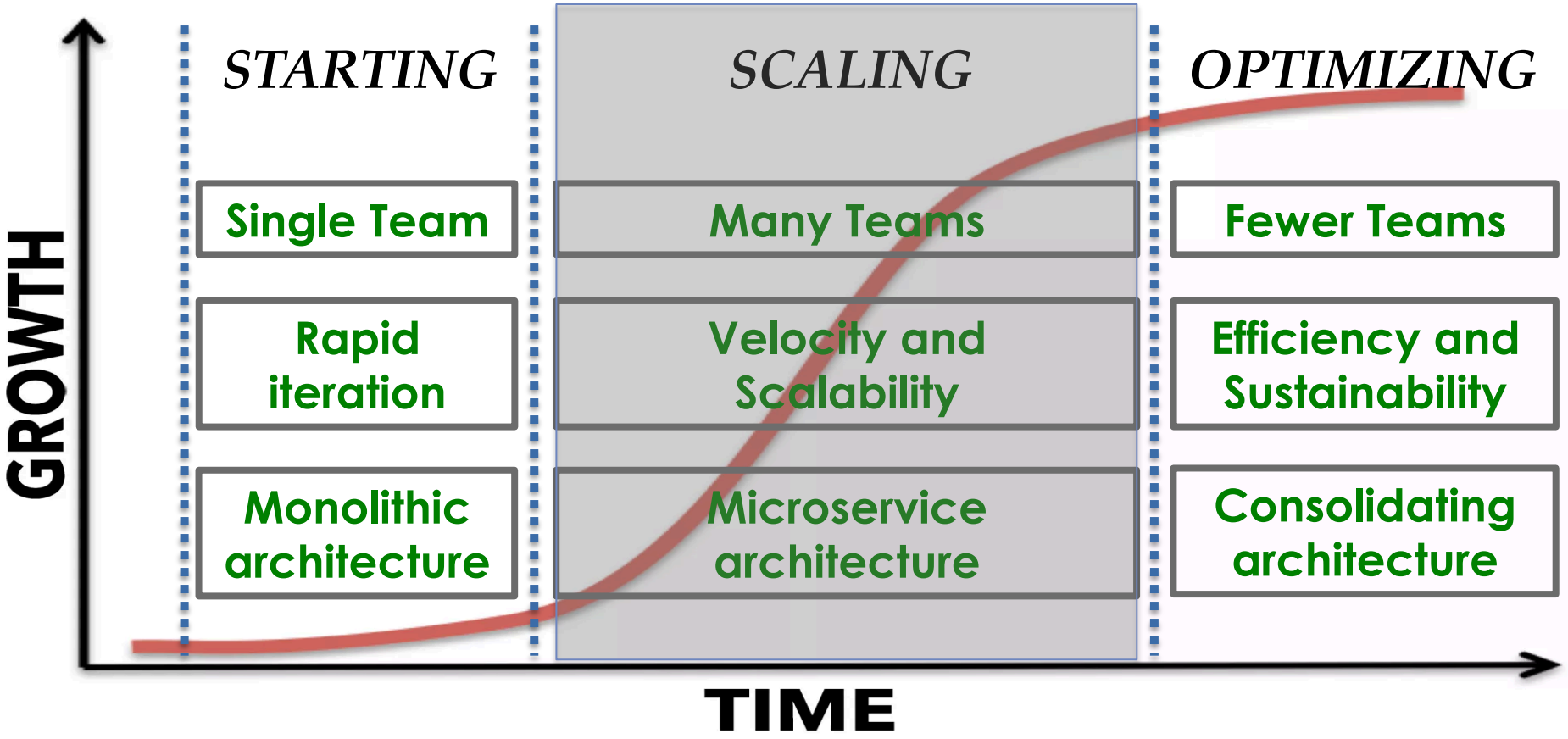
@randyshoup

https://ittybiz.com/s-curve/



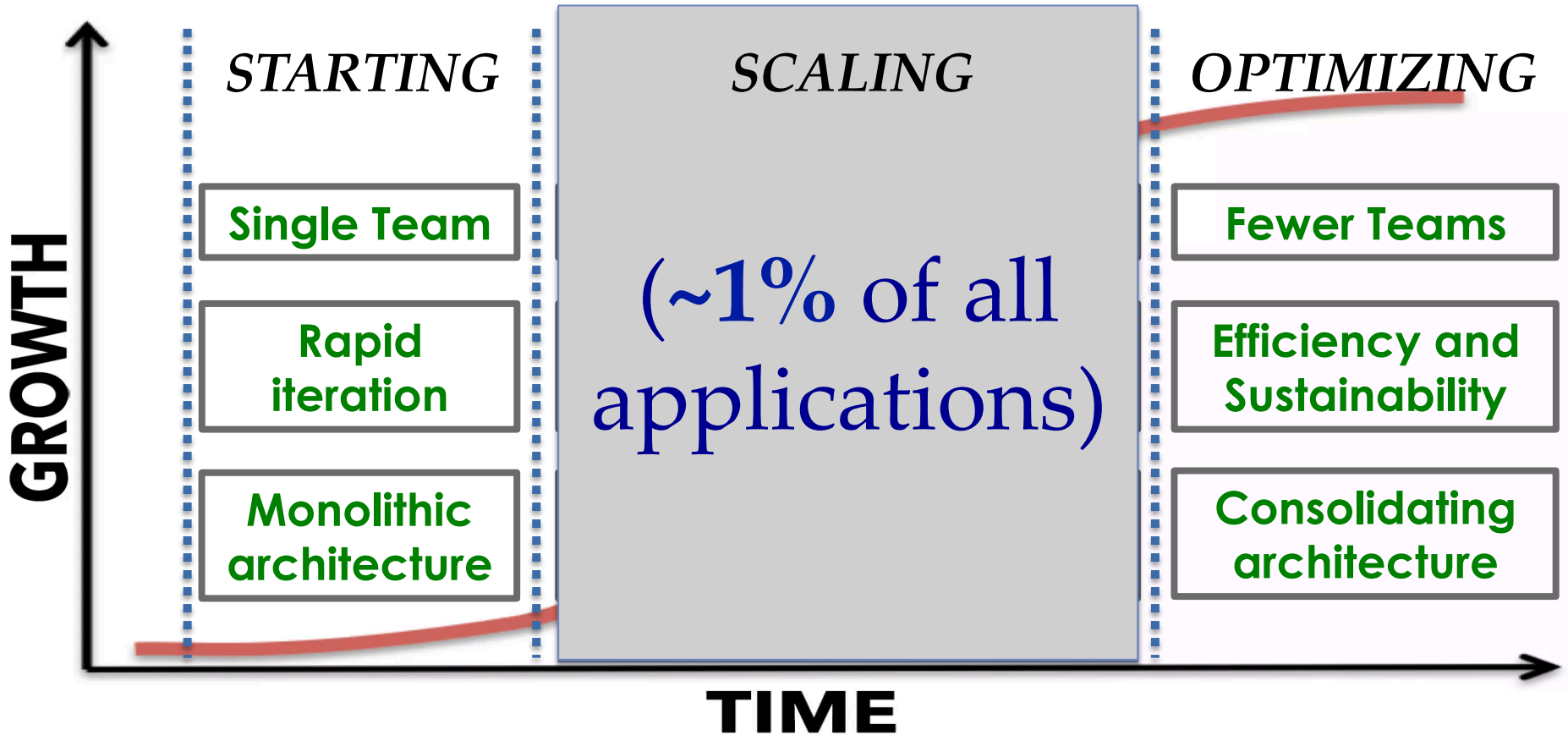
@randyshoup

https://ittybiz.com/s-curve/



@randyshoup

https://ittybiz.com/s-curve/



@randyshoup

https://ittybiz.com/s-curve/

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 / C++ → microservices

No one starts with microservices

…

Past a certain scale, everyone ends up with microservices

…

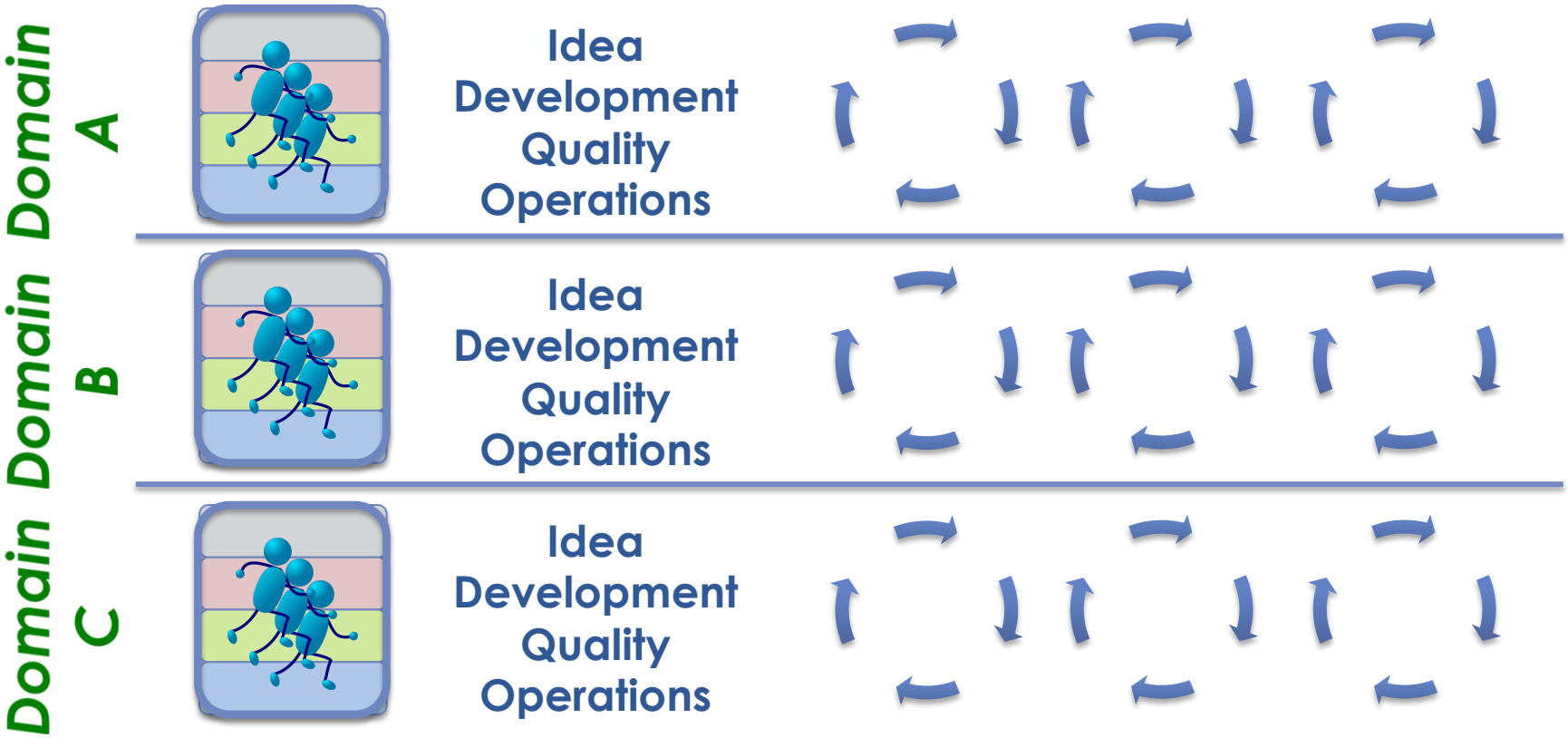
but most never reach that scale

@randyshoup

Scaling Architecture

- Architecture Evolution
- Service Architecture
- Event-Driven Communication
- Combining Services + Events

Many Autonomous Teams



Service Architecture

- One domain: One team: One / few service(s)
  - Organization ← *reflects* → Architecture ("Conway's Law")
- Autonomy and Accountability
  - Team can independently design, develop, deploy, operate its service(s)
  - Team owns its service(s) end to end

Service Architecture

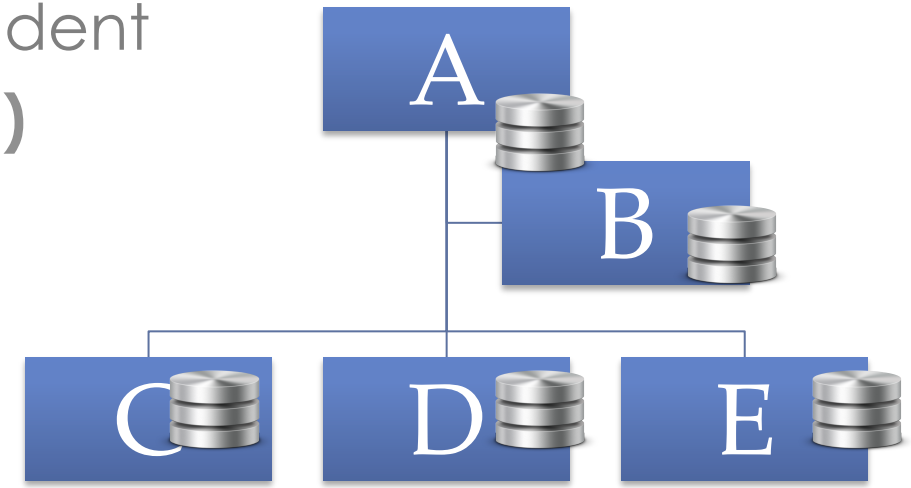
- Abstraction and Encapsulation
  - Fault isolation
  - Performance optimization
  - Security boundary
- Strict interface discipline
  - Well-specified interface contract
  - Testable and mockable

Service Architecture

- All operations through published service interface
  - No backdoor access to database (!)

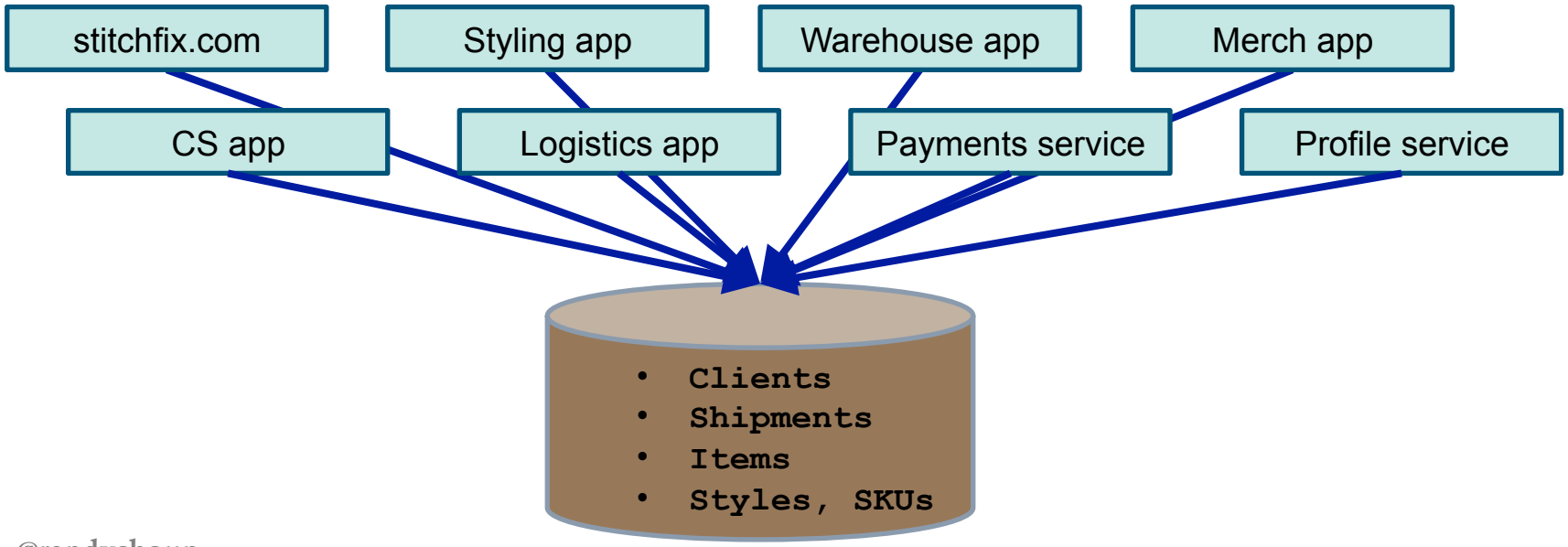
Microservices

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



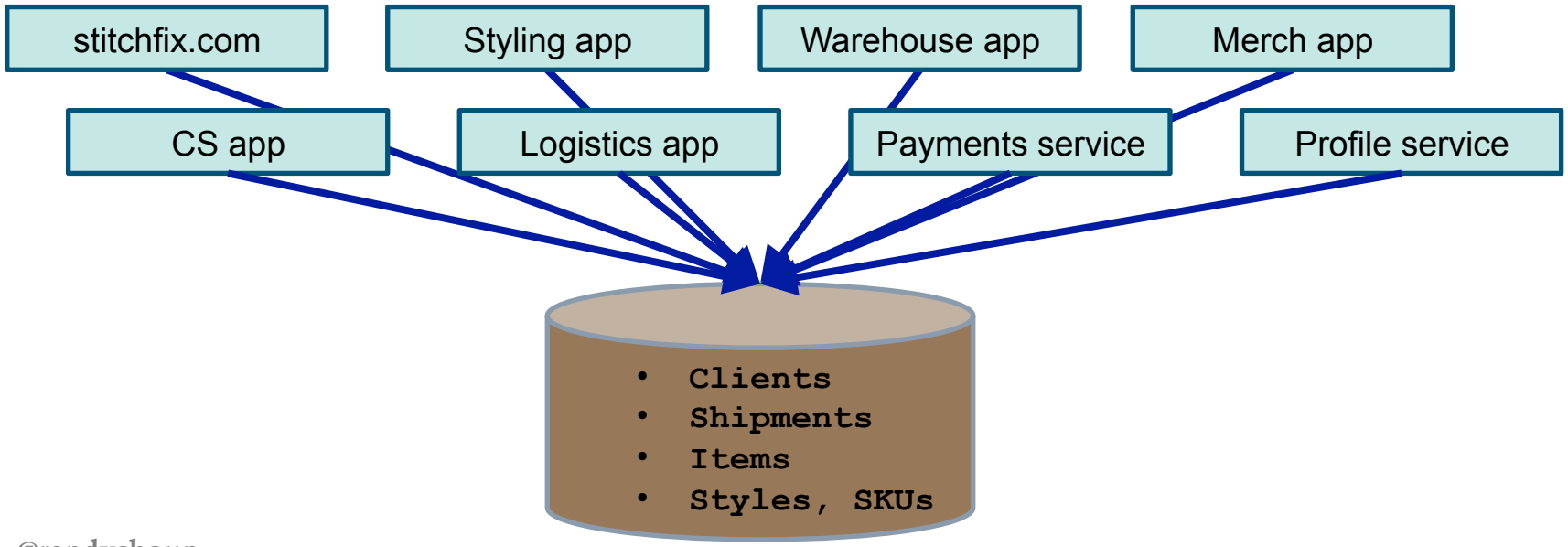
Extracting Microservices

- Problem: Monolithic shared DB



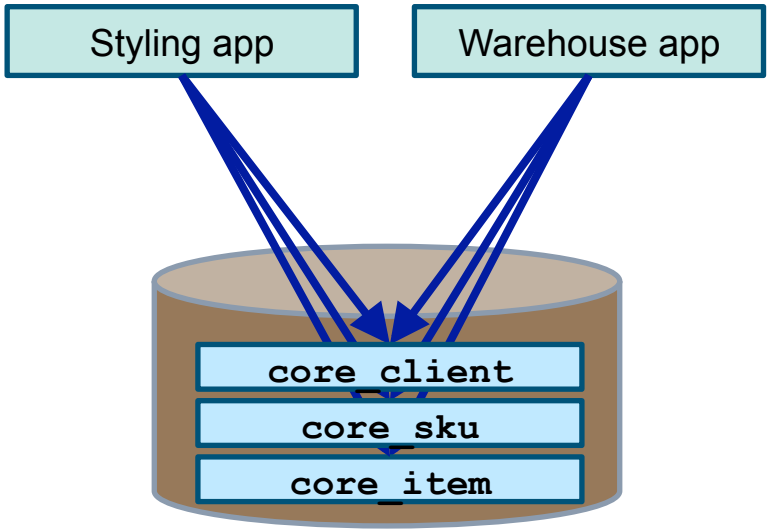
Extracting Microservices

- Decouple applications / services from shared DB



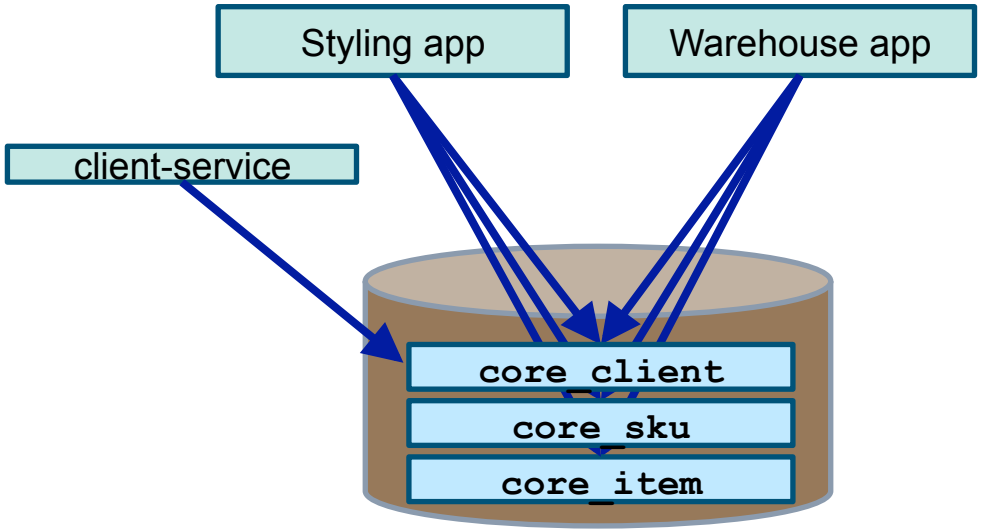
Extracting Microservices

- Decouple applications / services from shared DB



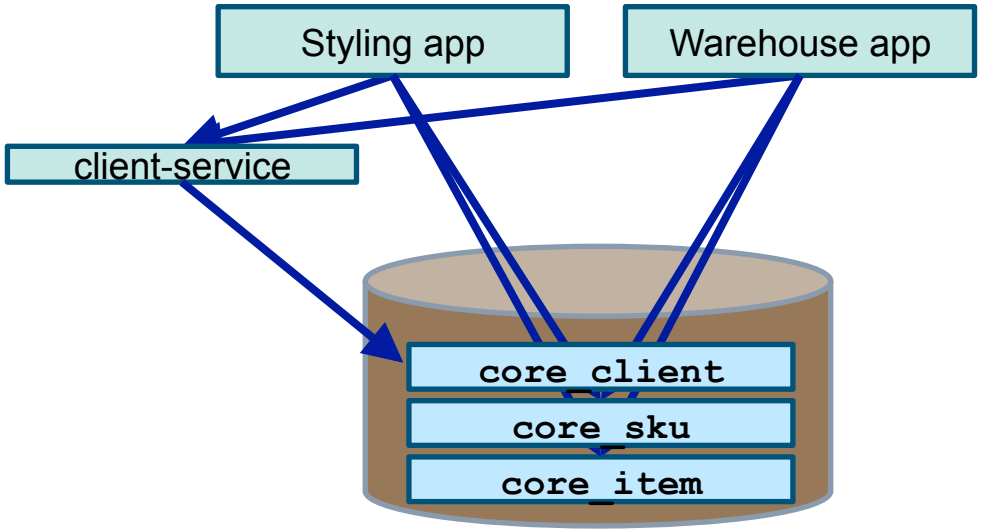
Extracting Microservices

- Step 1: Create a service



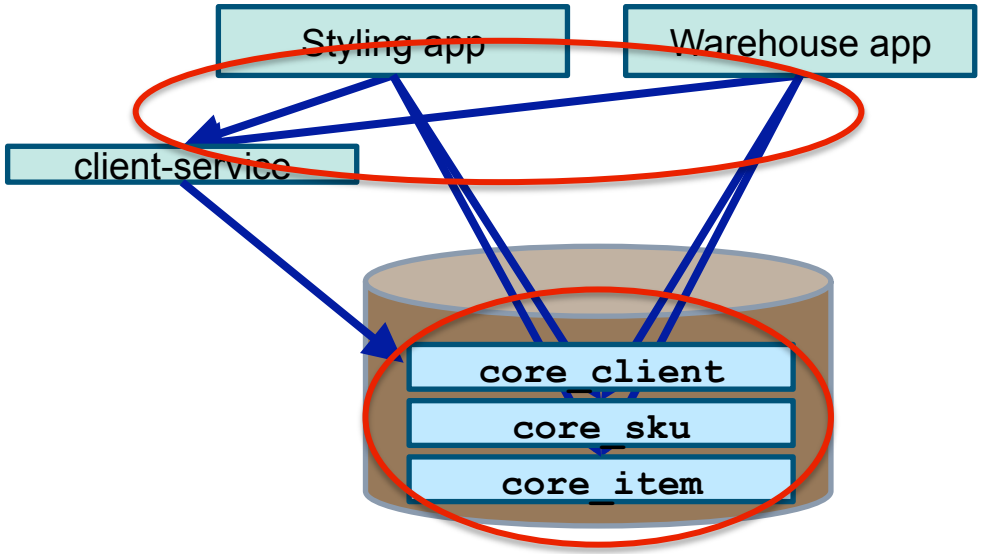
Extracting Microservices

- Step 2: Applications use the service



Extracting Microservices

- Step 2: Applications use the service



Do NOT stop here!

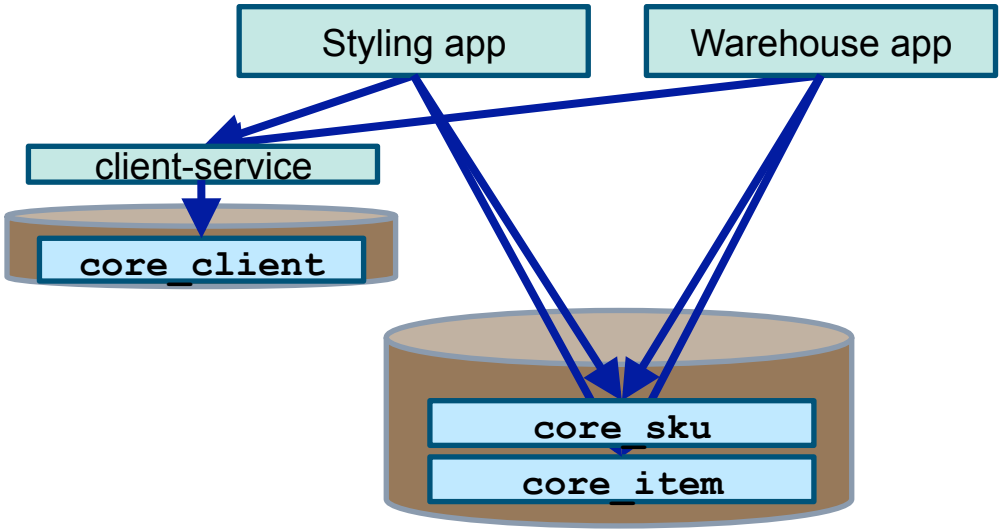
① All the problems of a distributed system

② All the problems of a shared database

③ None of the benefits of microservices ☹

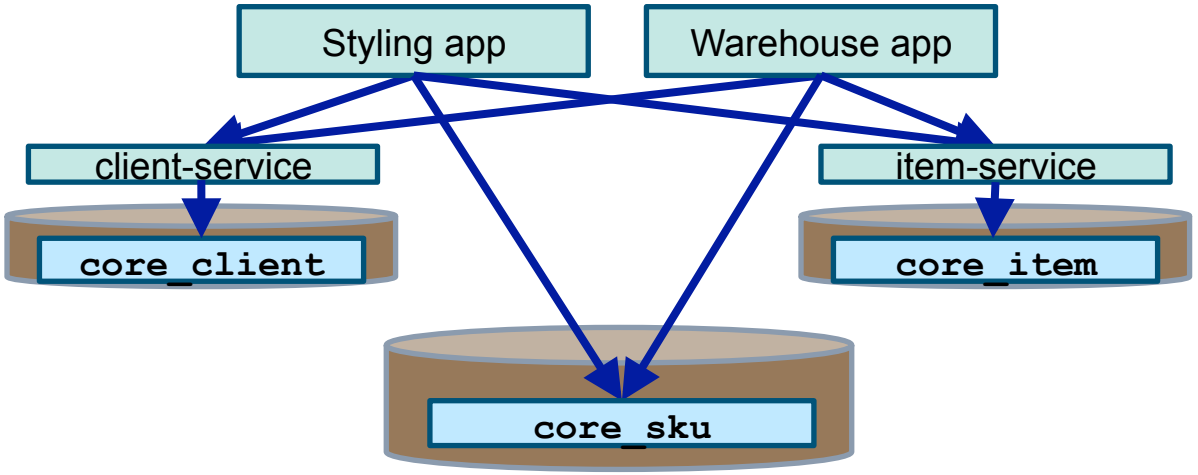
Extracting Microservices

- Step 3: Move data to private database



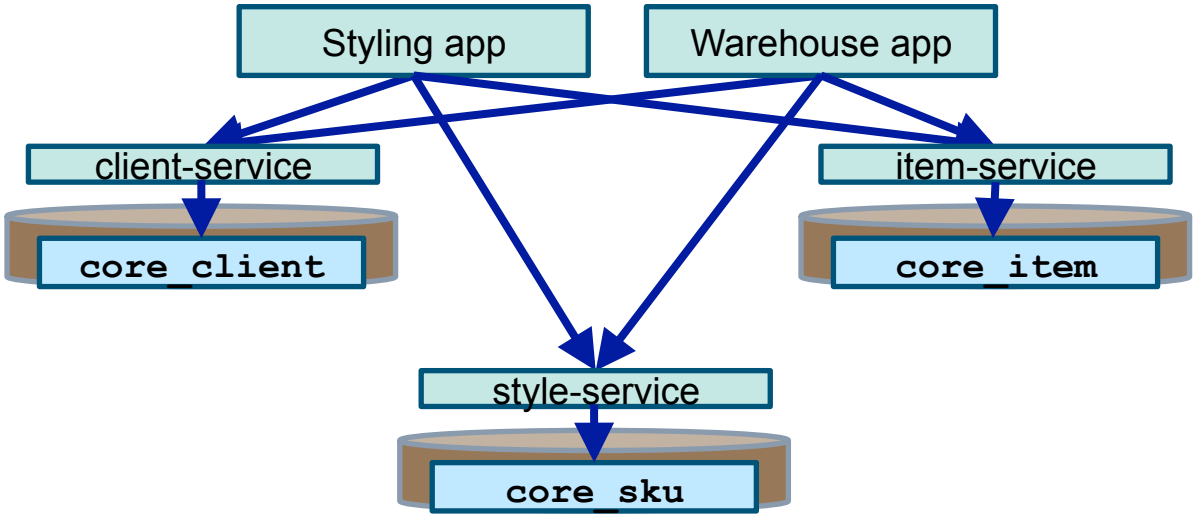
Extracting Microservices

- Step 4: Rinse and Repeat



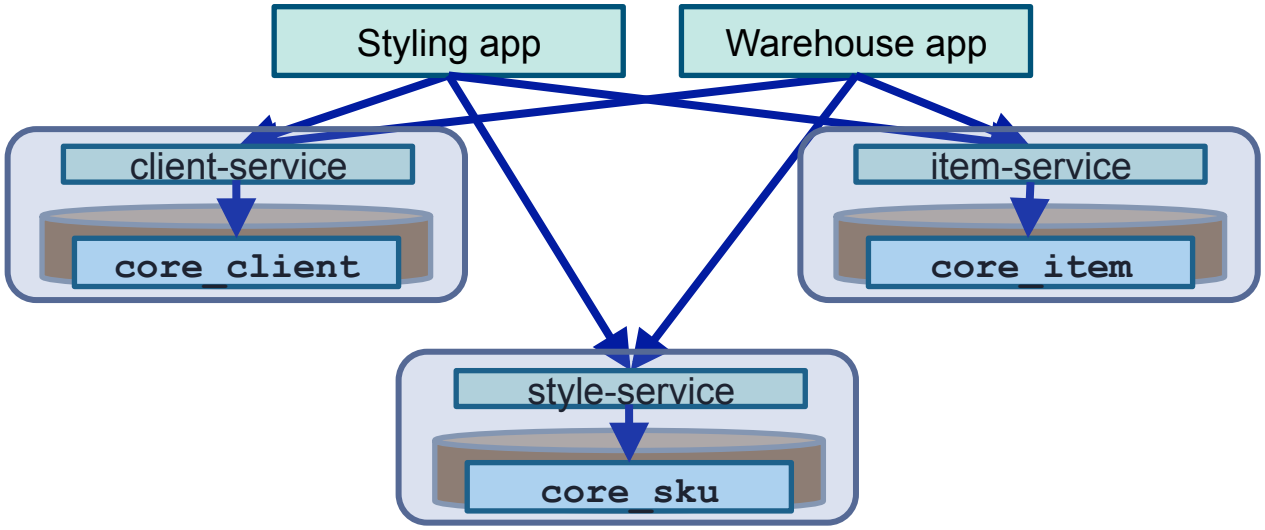
Extracting Microservices

- Step 4: Rinse and Repeat



Extracting Microservices

- Step 4: Rinse and Repeat



Scaling Architecture

- Architecture Evolution
- Service Architecture
- Event-Driven Communication
- Combining Services + Events

Event-Driven Communication

- Service publishes an event when state changes
  - Statement that some interesting thing occurred
- Consumers subscribe to the event
- Events are a first-class part of a service interface

Event-Driven Communication

- Decouple domains and teams
  - Abstracted through a well-defined interface
  - Asynchronous from one another
- Decouple producer and consumer services
  - Decoupled availability
  - Independent scalability

Event-Driven Communication

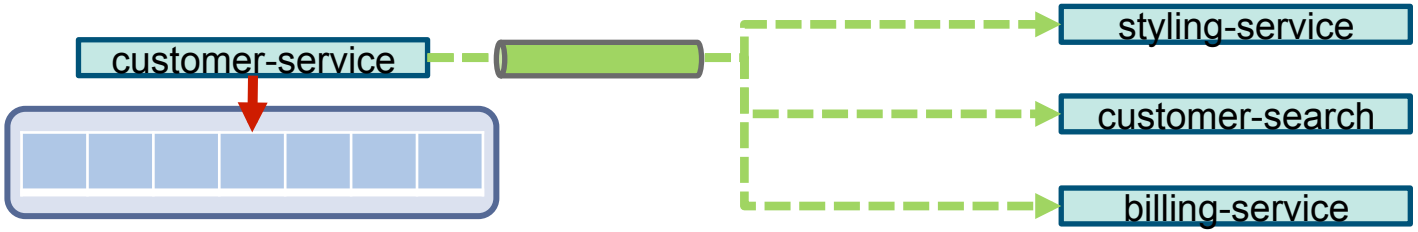
- Strict interface discipline
  - Well-specified event schema
  - Testable and mockable

Scaling Architecture

- Architecture Evolution
- Service Architecture
- Event-Driven Communication
- Combining Services + Events

Combining Services + Events

- Service as System of Record
  - Every piece of data is owned by a single service
  - That service is the **canonical system of record** for that data

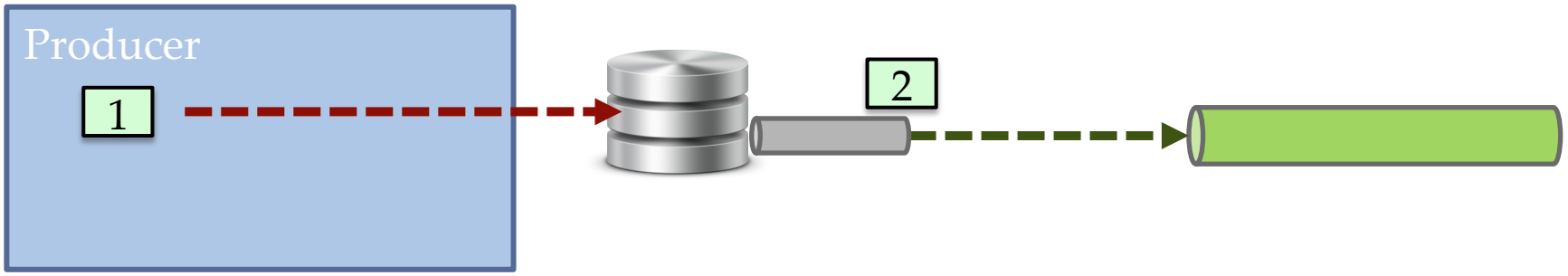


- Events as State Changes
  - Every other copy is a **read-only, non-authoritative cache**

Producer "Correctness"

Option 1: Change Data Capture

- Write state change to database
  - *(Database writes change to its transaction log)*
- "Connector" tails transaction log, sends event

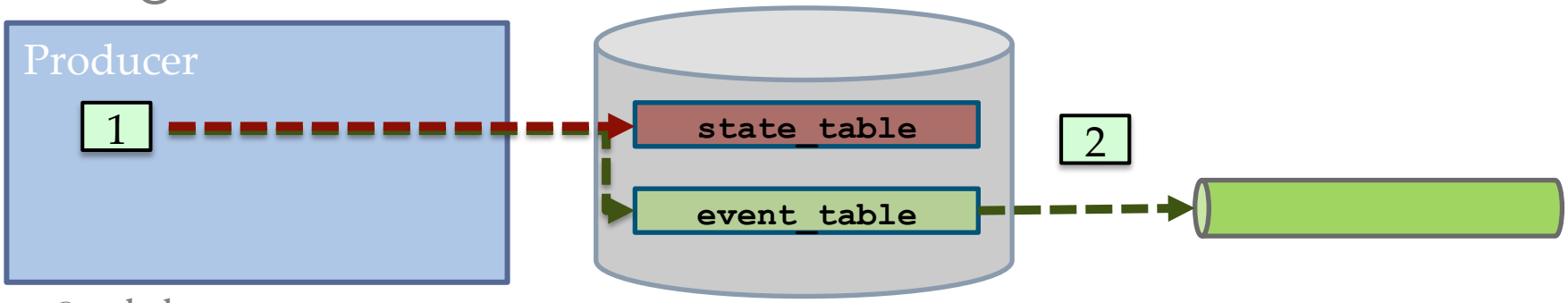


@randyshoup

Producer "Correctness"

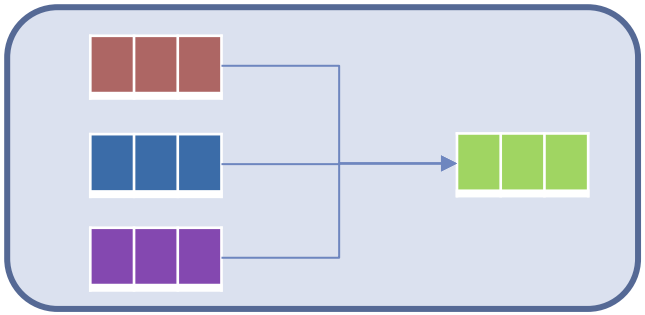
Option 2: Transactional Outbox

- State changes and events are stored in the same system
- E.g., state and events live in database tables



Shared Data

- Monolithic database makes it easy to leverage shared data

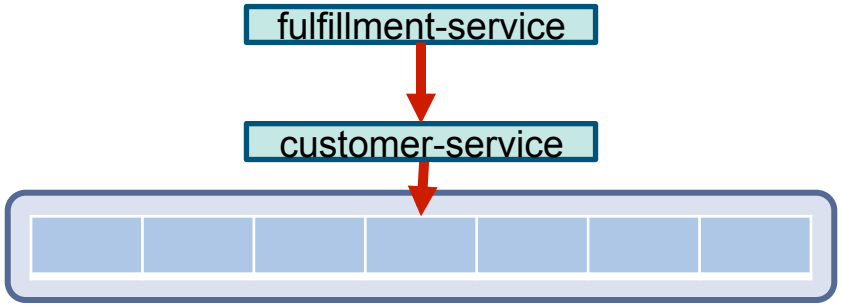


- Where does shared data go in a microservices world?

Shared Data

Option 1: Synchronous Lookup

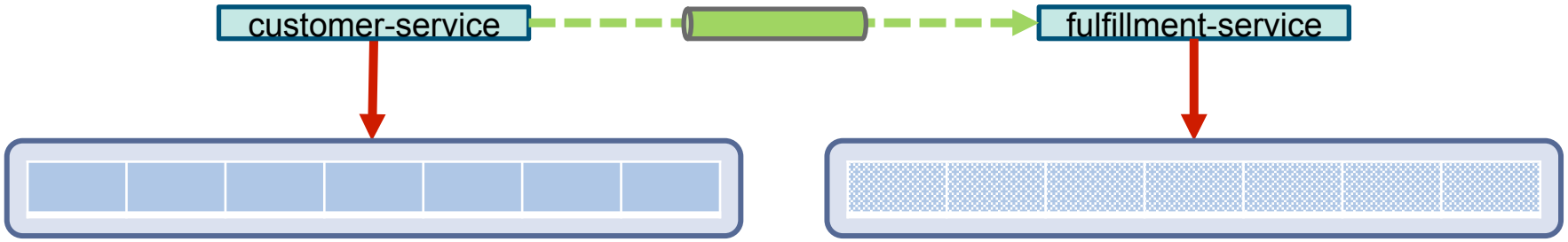
- Customer service owns customer data
- Fulfillment service calls customer service in real time



Shared Data

Option 2: Async event + local cache

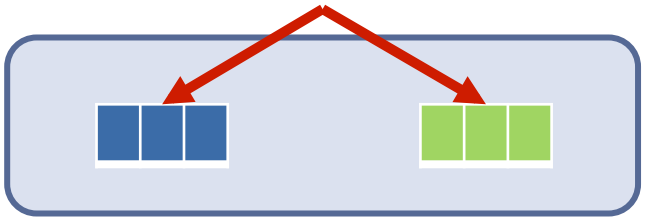
- Customer service owns customer data
- Customer service sends `address-updated` event when customer address changes
- Fulfillment service caches current customer address



Joins

- Monolithic database makes it easy to join tables

```
SELECT FROM A INNER JOIN B ON …
```

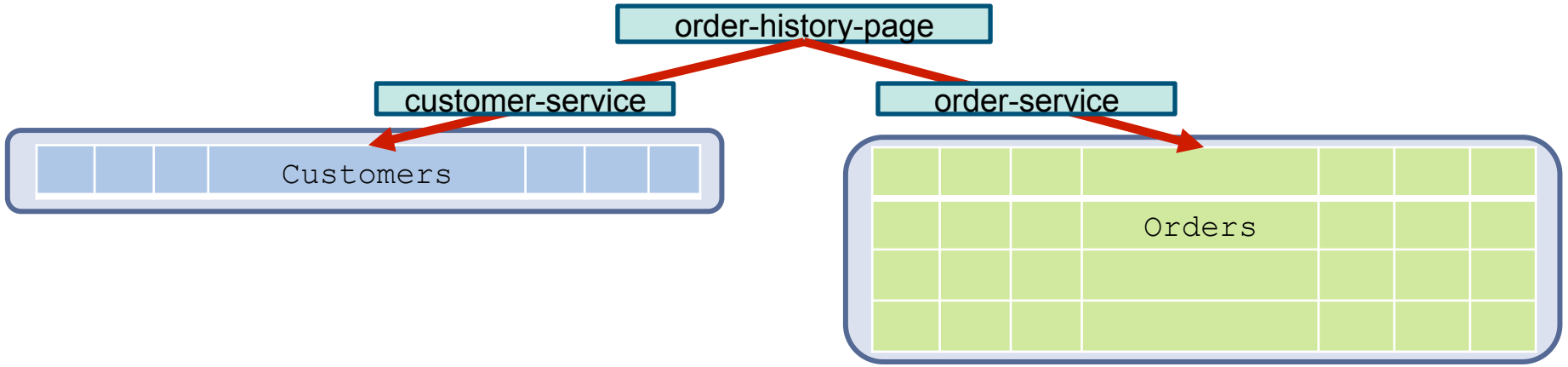


- Splitting the data across microservices makes joins very hard

Joins

Option 1: Join in Client Application

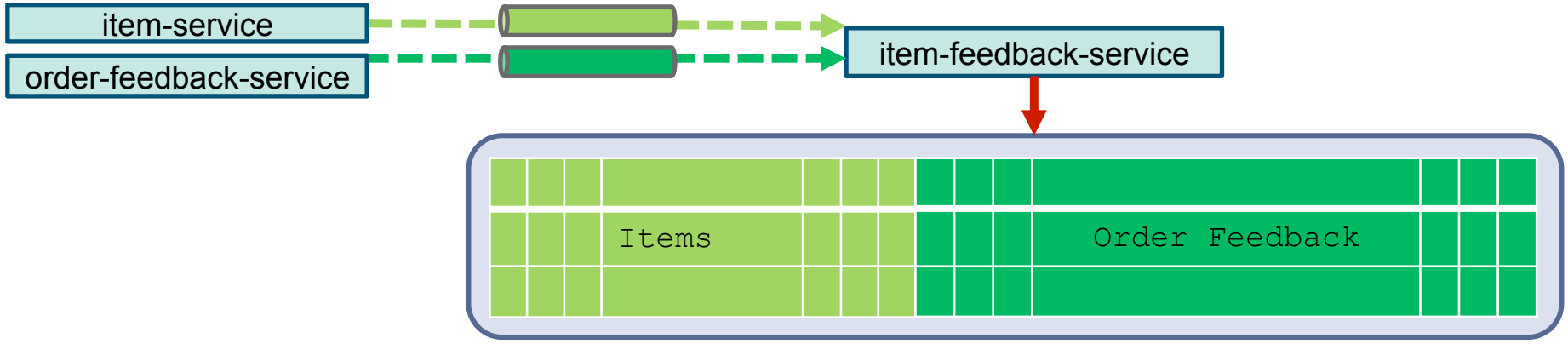
- Get a single customer from `customer-service`
- Query matching orders for that customer from `order-service`



Joins

Option 2: 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



Joins

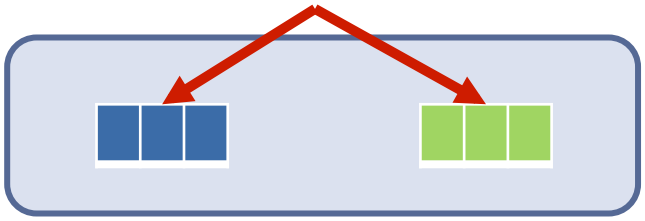
Many common systems do this

- "Materialized view" in database systems
- Most NoSQL systems
- Search engines
- Analytic systems

Transactions

- Monolithic database makes transactions across multiple entities easy

```
BEGIN; INSERT INTO A …; UPDATE B...; COMMIT;
```



- Splitting data across services makes transactions very hard

"In general, application developers simply **do not implement large scalable applications assuming distributed transactions.**"

-- Pat Helland

Life After Distributed Transactions: An Apostate's Opinion, 2007

"Grownups don't use distributed transactions"

-- Pat Helland

Workflows and Sagas

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

- Roll back with compensating operations in reverse

Workflows and Sagas

Many real-world systems work like this

- Payment processing
- Expense approval
- Software development process

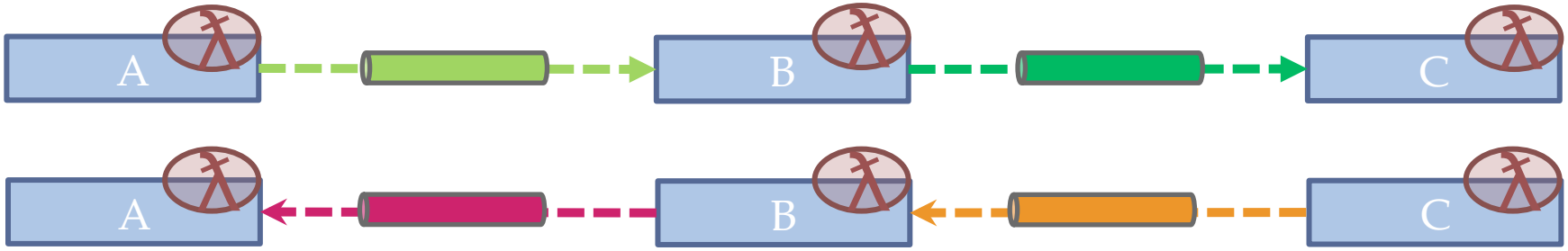
Intermediate States

Model intermediate states explicitly

- Payment *started*, *pending*, *complete*
- Expense *submitted*, *approved*, *paid*
- Feature *developed*, *reviewed*, *deployed*, *released*

Serverless in Action

- Simple event-driven processing
  - Very lightweight logic
  - Stateless
  - Triggered by an event



- ➔ Consider Function-as-a-Service ("*Serverless*")

Scaling Architecture

- Architecture Evolution
- Service Architecture
- Event-Driven Communication
- Combining Services + Events

Thank you!

 @randyshoup

 linkedin.com/in/randyshoup

 medium.com/@randyshoup