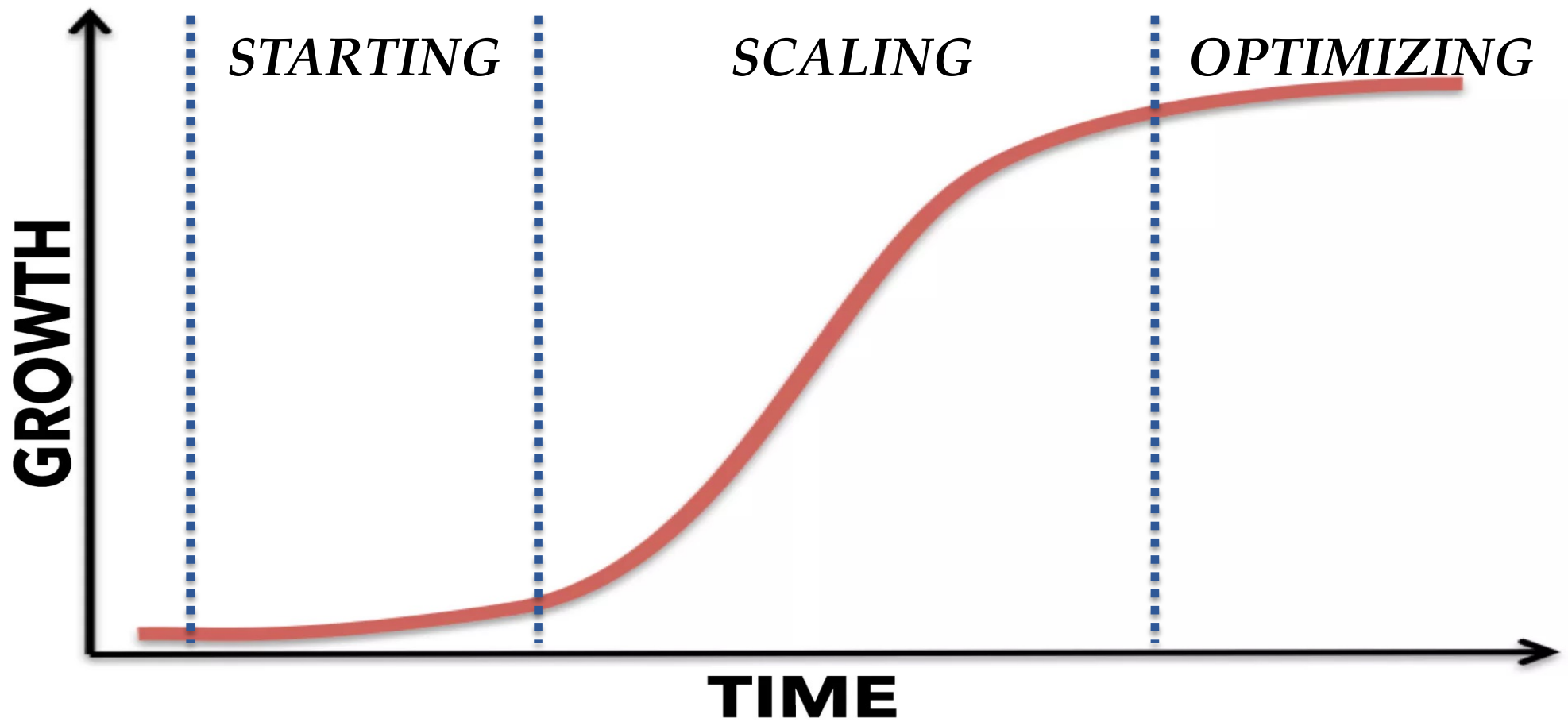


Minimal Viable Architecture

Randy Shoup
@randyshoup

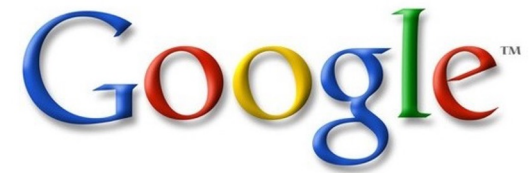
There is no perfect architecture
for all scales
for all phases of evolution
for all problem domains



@randyshoup

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

Background



STITCH FIX™

we work

“Tell us how you did things at Google and eBay.”

“Sure, I’ll tell you, but you have to promise not to do them! [... yet]”

@randyshoup

eBay Architecture



- 1995 Monolithic Perl
- 1996-2002 v2
 - Monolithic C++ ISAPI DLL
 - 3.4M lines of code
 - Compiler limits on number of methods per class
- 2002-2006 v3 Migration
 - Java “mini-applications”
 - Shared databases
- 2012 v4 Microservices
- 2017 v5 Microservices

Amazon Architecture



- 1995-2001 Obidos
 - Monolithic Perl / Mason frontend over C backend
 - ~4GB application in a 4GB address space
 - Regularly breaking Gnu linker
 - Restarting every 100-200 requests for memory leaks
 - Releasing once per quarter
- 2001-2005 Service Migration
 - Services in C++, Java, etc.
 - No shared databases
- 2006 AWS launches

No one starts with microservices

...

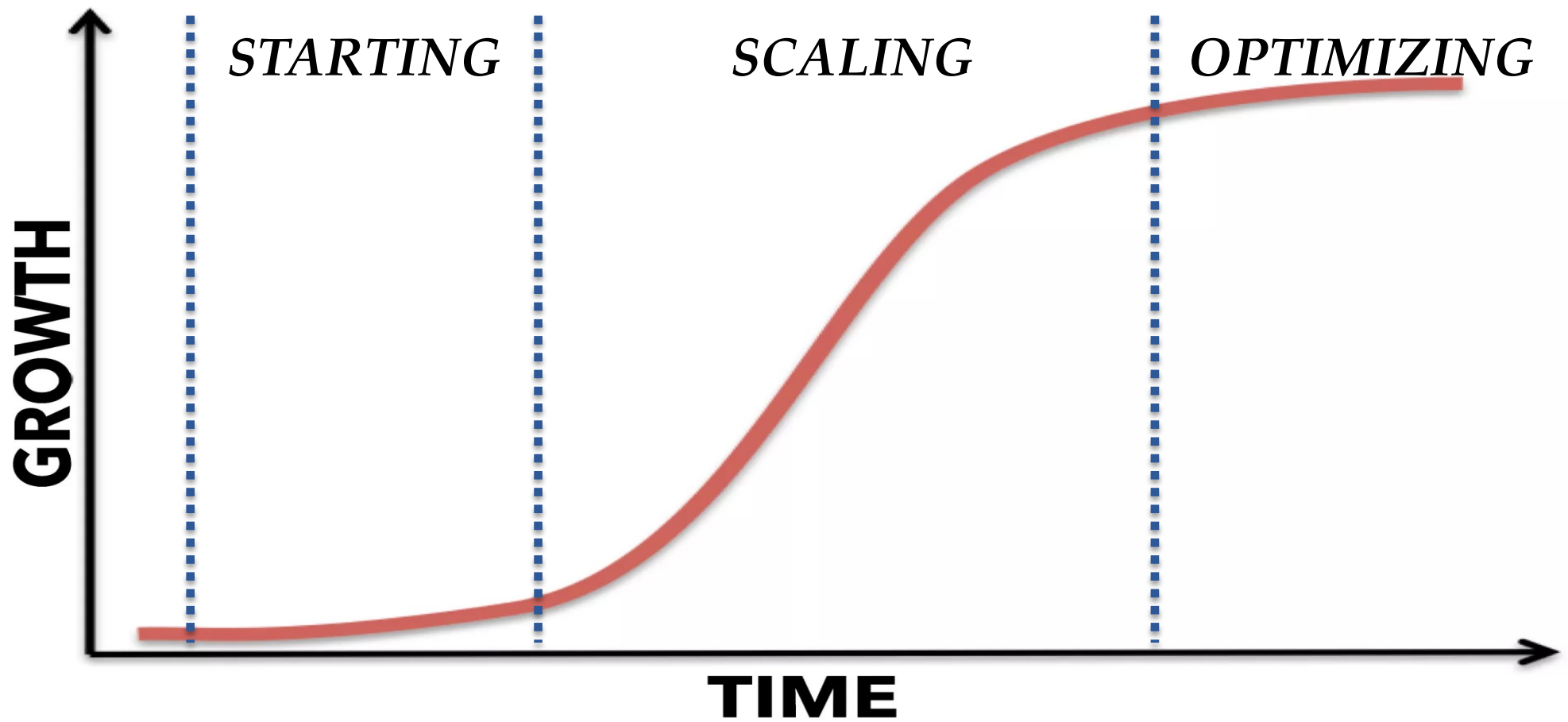
Past a certain scale, everyone ends
up with microservices

...

but most never reach that scale

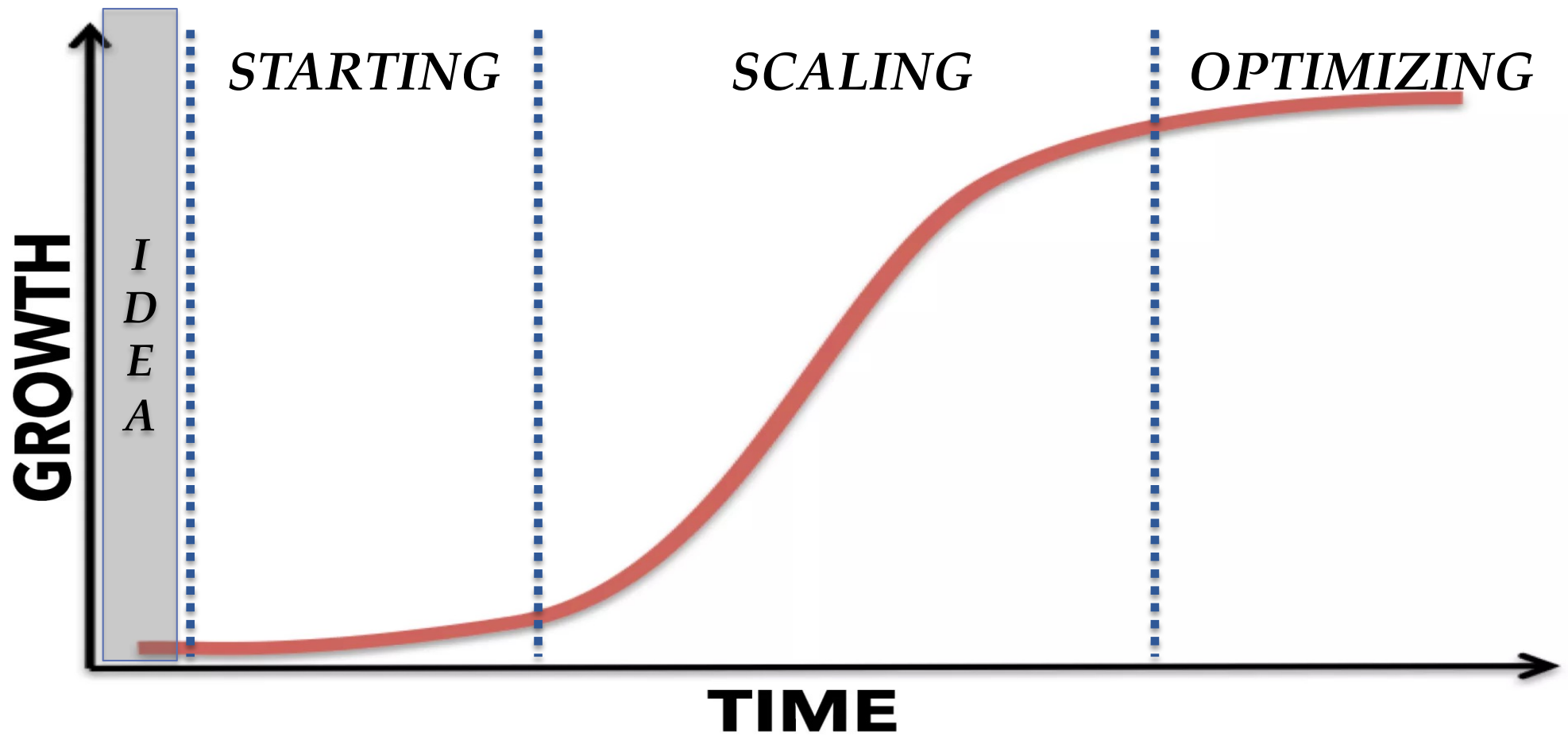
If you don't end up regretting
your early technology
decisions, you probably over-
engineered.

@randyshoup



@randyshoup

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

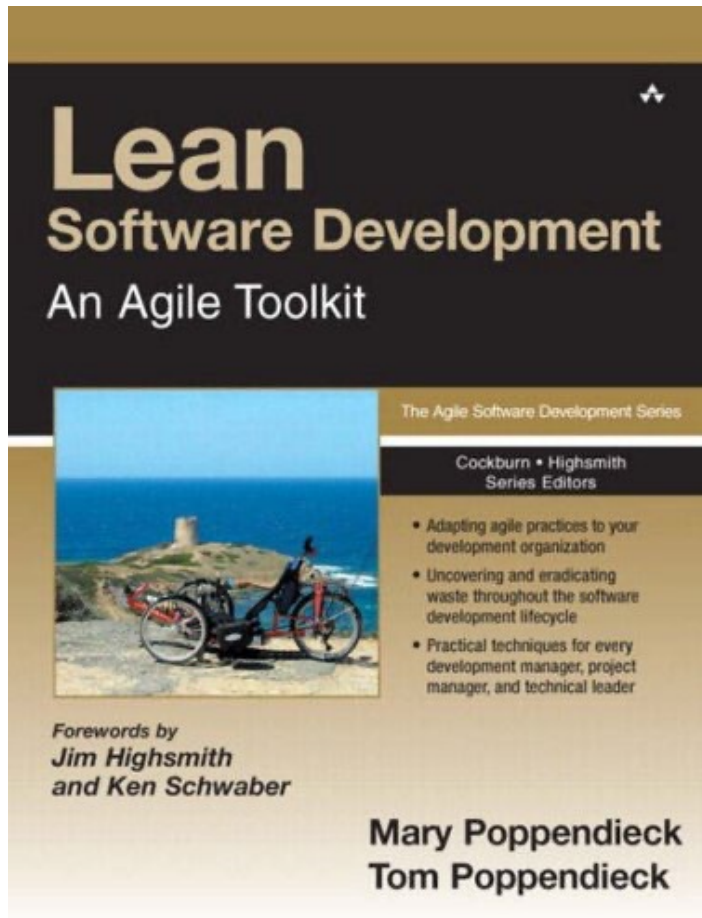


@randyshoup

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

“What Problem Are
We Trying to Solve?”

@randyshoup



“Building the wrong thing is the biggest waste in software development.”

@randyshoup

“A problem well-stated is a problem half-solved.”

-- Charles Kettering,
head of research at GM

@randyshoup



“Prototype” Architecture

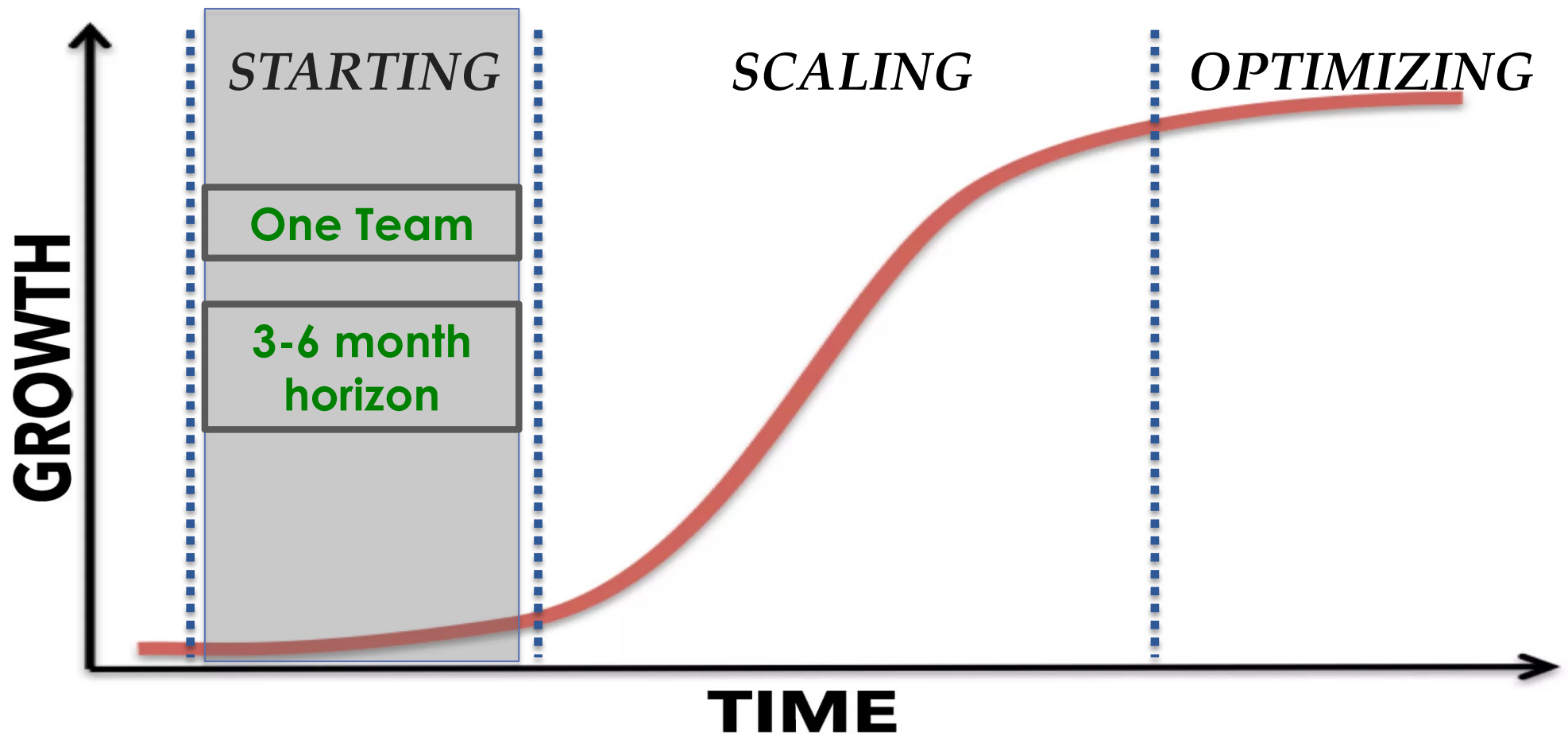
- Goal: Explore our problem space as rapidly and cheaply as possible
 - Find business model
 - Find product market fit
 - Acquire first customers
- ➔ Rapid Iteration
 - *Everything* is a prototype
 - We *will* throw it all away

“Prototype” Architecture

- Ideally No Technology At All
 - Paper prototypes
 - Fake Google ads
 - Excel spreadsheet
 - Wordpress blog

Engineering is about solving problems.

Sometimes we solve those problems by writing code.



“Just Enough” Architecture

- Goal: Meet near-term, evolving customer needs as cheaply as possible
 - Delight first customers
 - Acquire more
- Rapid learning and improvement
- Team productivity
- NOT about scaling

@randyshoup

“The best code you can write now is code you’ll discard in a couple of years time”

-- Martin Fowler

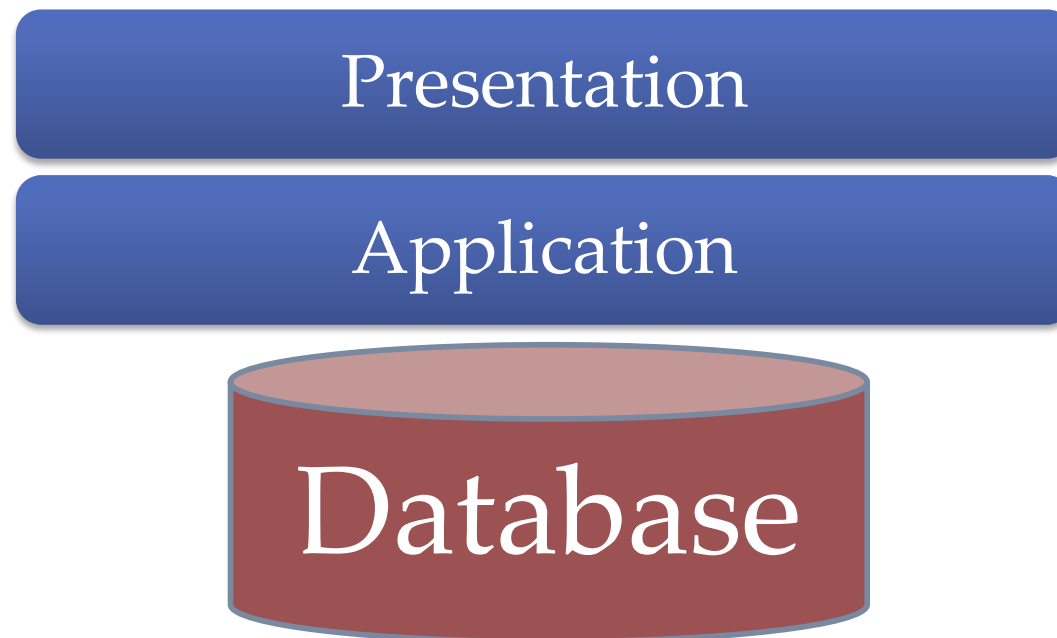
<http://martinfowler.com/bliki/SacrificialArchitecture.html>

“Just Enough” Architecture

- Simple, Familiar Technology
 - Expressive Power and Ease of Use
 - Rapid prototyping frameworks (Rails, PHP, etc.)
- Monolithic Architecture
 - Single application
 - Single database
- No Infrastructure
 - Ideally Platform-as-a-service

@randyshoup

Monolithic Architecture



Monolithic Architecture

Pros

- Simple at first
- In-process latencies
- Single build and deployment unit
- Resource-efficient at small scale

Cons

- Coordination overhead as teams grow
- Poor enforcement of modularity
- No horizontal scaling
- Single point of failure and single performance bottleneck

Buy, Not Build

- Use Cloud Infrastructure
 - Faster, cheaper, better than we can do ourselves
 - Ephemeral, (auto-)scalable, pay-per-use
 - Stitch Fix and WeWork have no physical compute infrastructure anywhere in the world
- Prefer Open Source
 - Usually better than the commercial alternatives (!)

@randyshoup

Buy, Not Build

- Third-Party Services
 - Logging, monitoring, alerting
 - Authentication
 - Payments, billing, fraud detection
 - Etc.
- Focus on our core competency
 - Use third-party services for **everything else** (!)

@randyshoup

Modular Monolith

- Modularity Discipline
 - Use “shared libraries” within the monolith
 - Easy to modify or replace
- Detailed Logging and Instrumentation
 - Understand user behavior
 - Enable diagnosis and recovery

Continuous Delivery

- Repeatable Deployment Pipeline
 - Low-risk, push-button deployment
 - Rapid release cadence
 - Rapid rollback and recovery
- More solid systems
 - Release smaller units of work
 - Smaller changes to roll back or roll forward
 - Faster to repair, easier to understand, simpler to diagnose

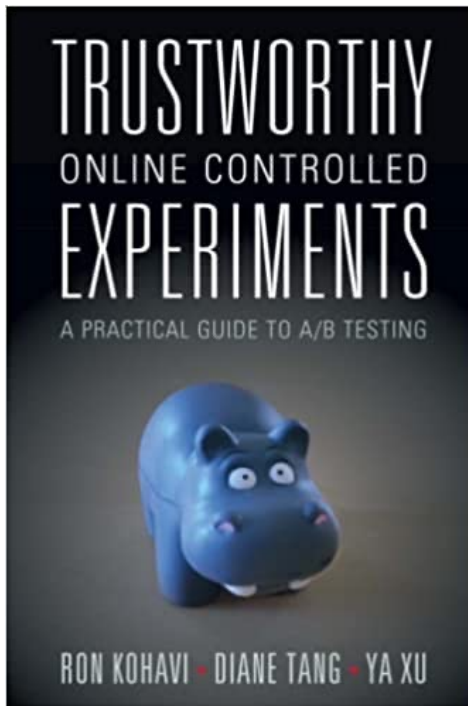
@randyshoup

Feature Flags

- Configuration “flag” to enable / disable a feature for a particular set of users
 - Independently discovered at eBay, Facebook, Google, etc.
- More solid systems
 - Decouple feature delivery from code delivery
 - Rapid on and off
 - Develop / test / verify in production
 - Dark launches
- Enables experimentation
 - A | B testing

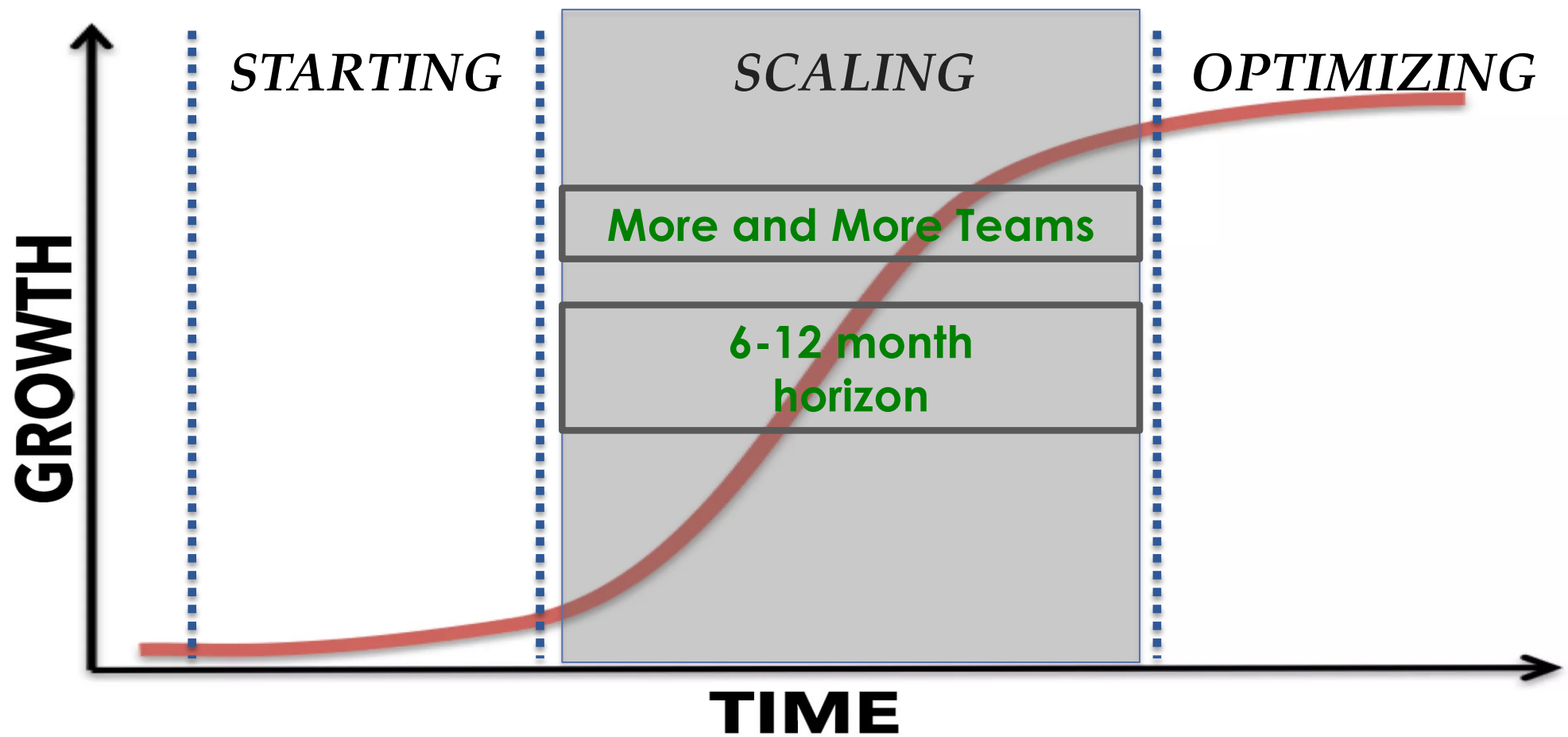
@randyshoup

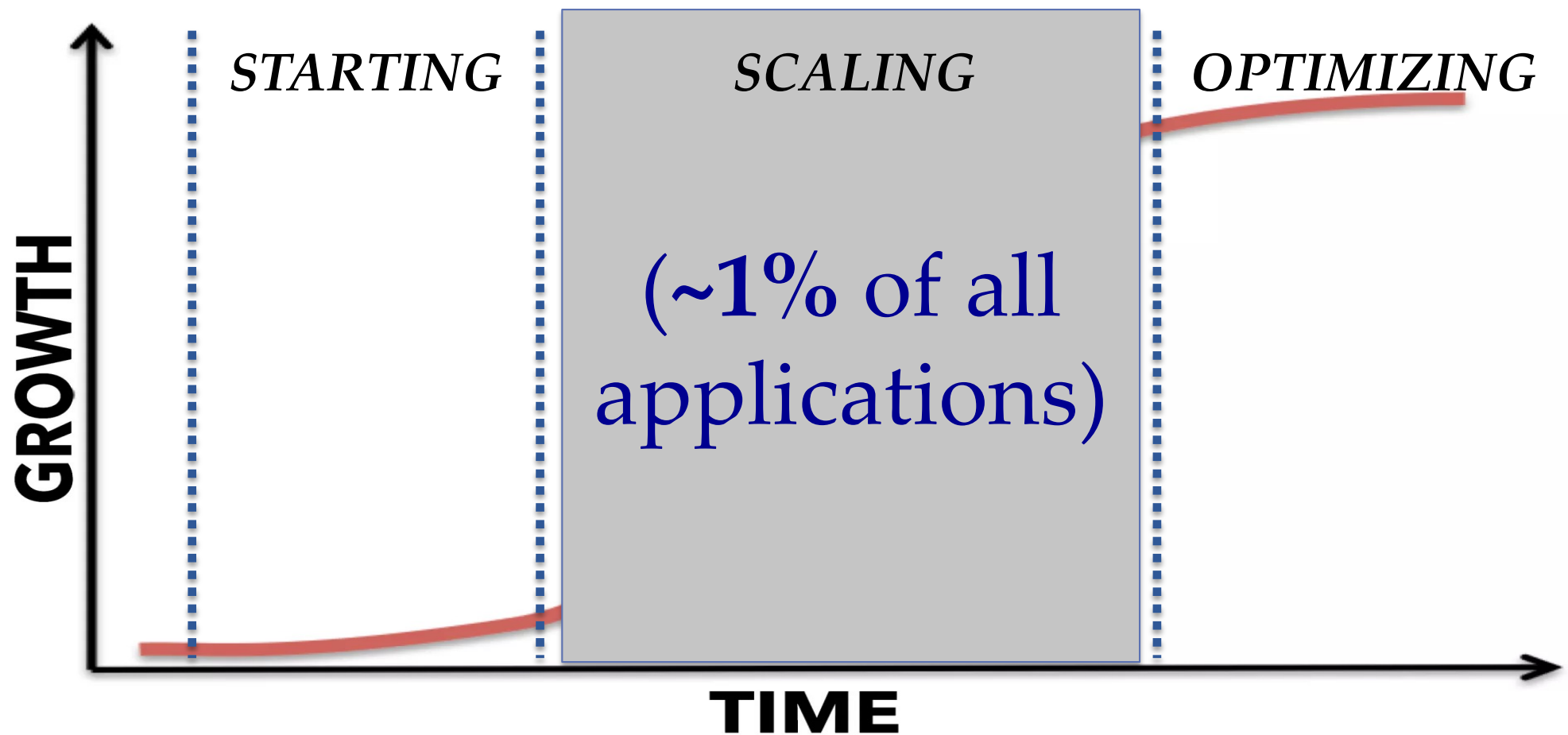
Features and Business Value



- 1/3 of features are positive and statistically significant

- 1/3 of features are flat – no significant difference
- 1/3 of features are negative and statistically significant





When to Rearchitect?

- Velocity
 - Time to market keeps slowing down
 - Teams step on each others' toes, can no longer develop independently
 - Difficult for new engineers to be productive
- Scaling
 - Vertical scaling of the monolith no longer works
 - Parts of the system need to scale faster than others

When to Rearchitect?

- Deployment
 - Parts of the system need to deploy independently
 - Monolithic release is too slow, too complicated, too risky

Getting to rearchitect a system
is a sign of success, not failure.

Scalable Architecture

- Goal: Stay ahead of rapidly growing business. Keep the site up (!)
- Scaling the Teams
- Scaling the Technology
- Repeatable Processes

@randyshoup

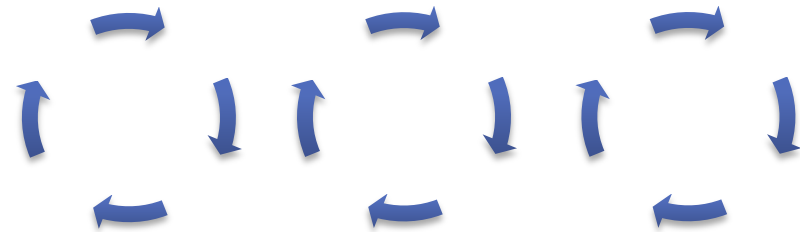
Scalable Organization

Domain Domain Domain

A



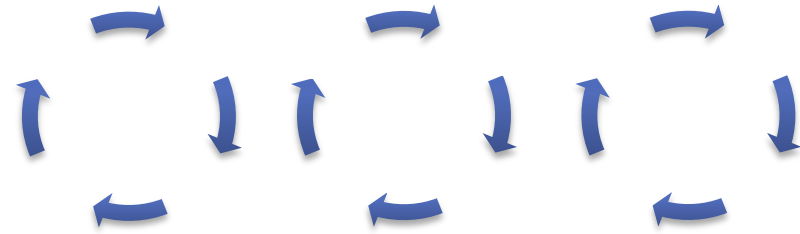
Idea
Development
Quality
Operations



B



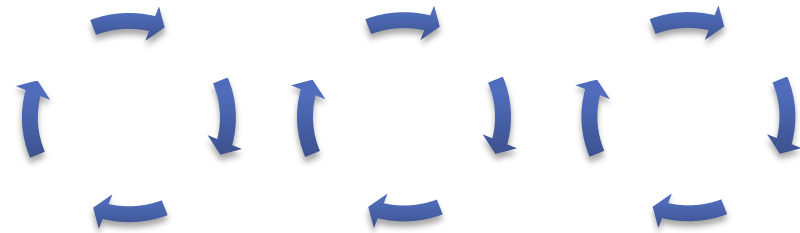
Idea
Development
Quality
Operations



C



Idea
Development
Quality
Operations



“Stream-Aligned” Teams



**<Business
Domain>**

- Single service or set of related services
- Team can design, develop, deploy, and operate its service(s) independently of other teams
- Team owns its service(s) end to end, cradle to grave

Scalable Architecture

- Technology that Scales
 - Common migrations to {Python, Go, JVM languages}
 - Latency and performance
 - Asynchrony
- Independent systems
 - Fit-for-purpose systems: analytics, search
 - Separated services: Payments, etc.
 - Layered services
- Event queue
 - Use events to communicate between applications and services

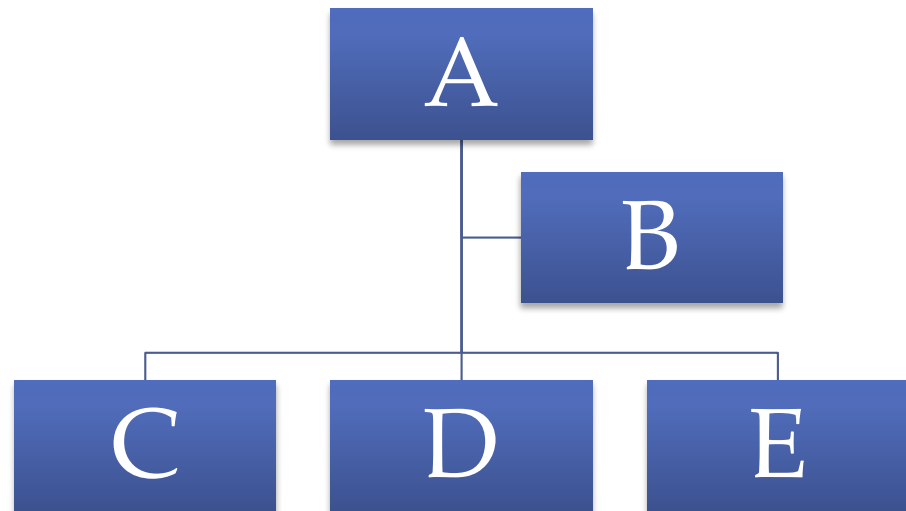
@randyshoup

Scalable Architecture

- Scalable persistence
 - Break up the monolithic database
 - Functional partitioning
 - Sharding

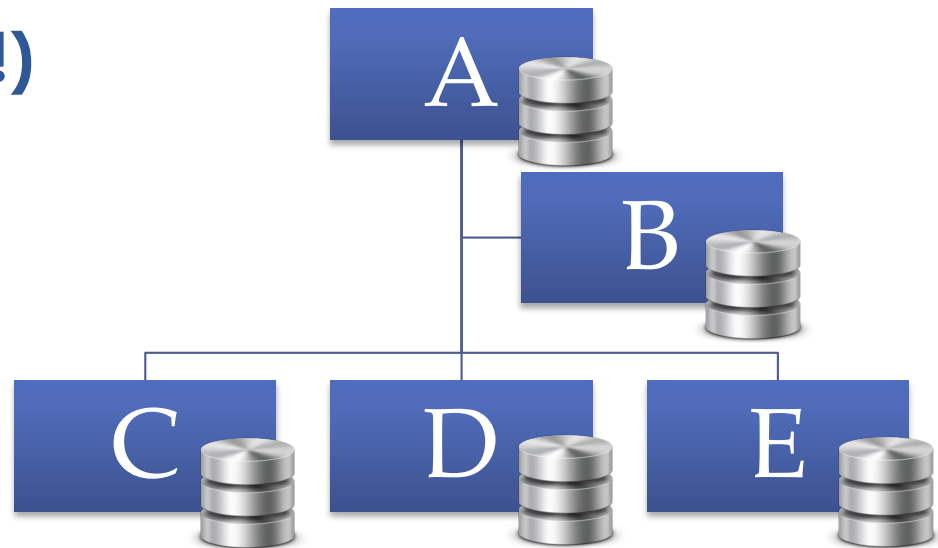
Microservices

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



Microservices

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



Microservice Architecture

Pros

- Each unit is simple
- Independent scaling
- Independent deployment
- “Optimal” technology stack
- Security boundary

Cons

- Multiple cooperating units
- Exchange in-process for network latencies
- More sophisticated deployment and monitoring
- Overall system complexity

“The only thing a Big Bang
migration guarantees is a big
Bang”

-- Martin Fowler

Incremental Migration

- Step 0: Pilot Implementation
 - Choose initial end-to-end vertical experience to migrate / create
 - (+) Opportunity to learn and adjust
 - (+) Demonstrate feasibility and gain confidence
 - (+) Bound investment and risk
 - (+) Provide real customer value
- Initial step is the hardest
 - Learning how to do things in the new way
 - Building out basic supporting capabilities

Incremental Migration

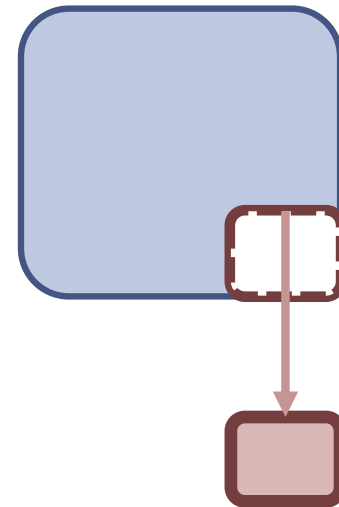
- Steps 1-N: Incremental Migration
 - Prioritize business value -- highest ROI areas first
 - Focus on areas with greatest rate of change
 - (+) Maximize near-term payoff from investment
 - (+) Confront and solve hard problems sooner rather than later
- New feature development in parallel
 - Typically cannot pause all feature work in all areas to migrate 😊
 - Within a particular component, try to separate feature work from migration work in distinct steps

Incremental Migration

- Residual components of original architecture may remain indefinitely
 - Lowest business value
 - Most stable and least changing
 - Can migrate – or not – opportunistically

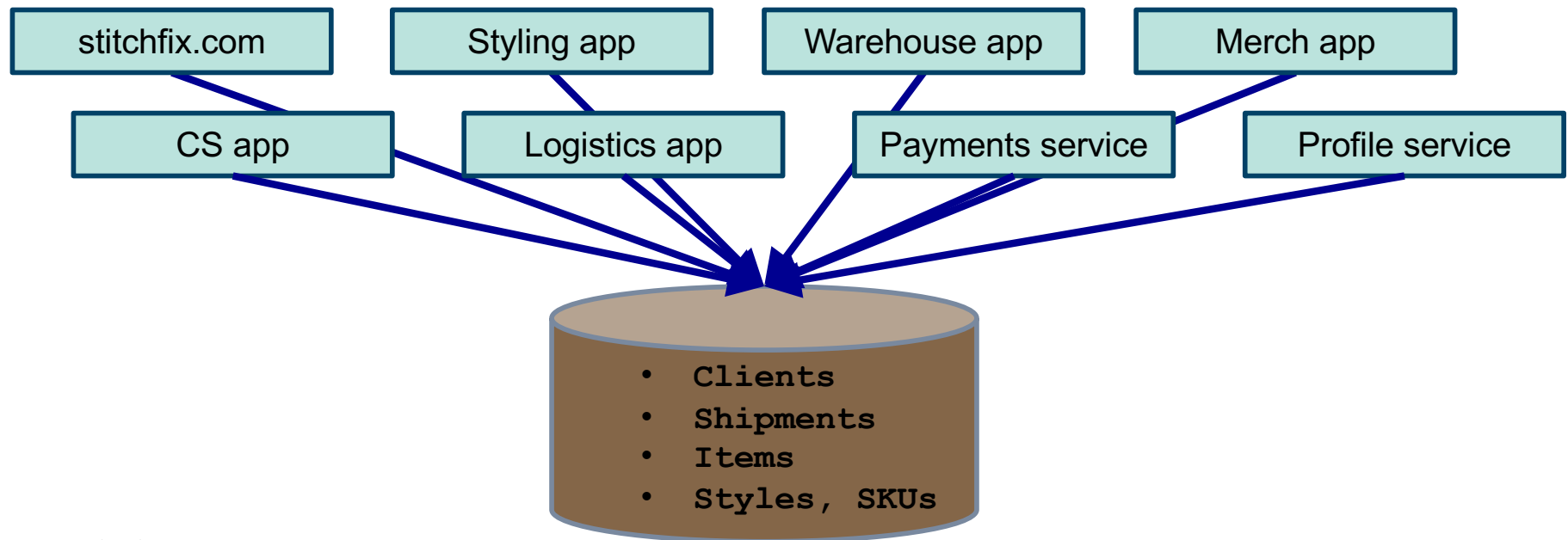
Carving up the Monolith

- Look for (or create) a “seam” in the monolith
 - This is often the hardest part (!)
- Wall it off behind an interface
- Write automated tests around the interface
- Replace implementation with an independent component
- ➔ Rinse and Repeat



Extracting Microservices

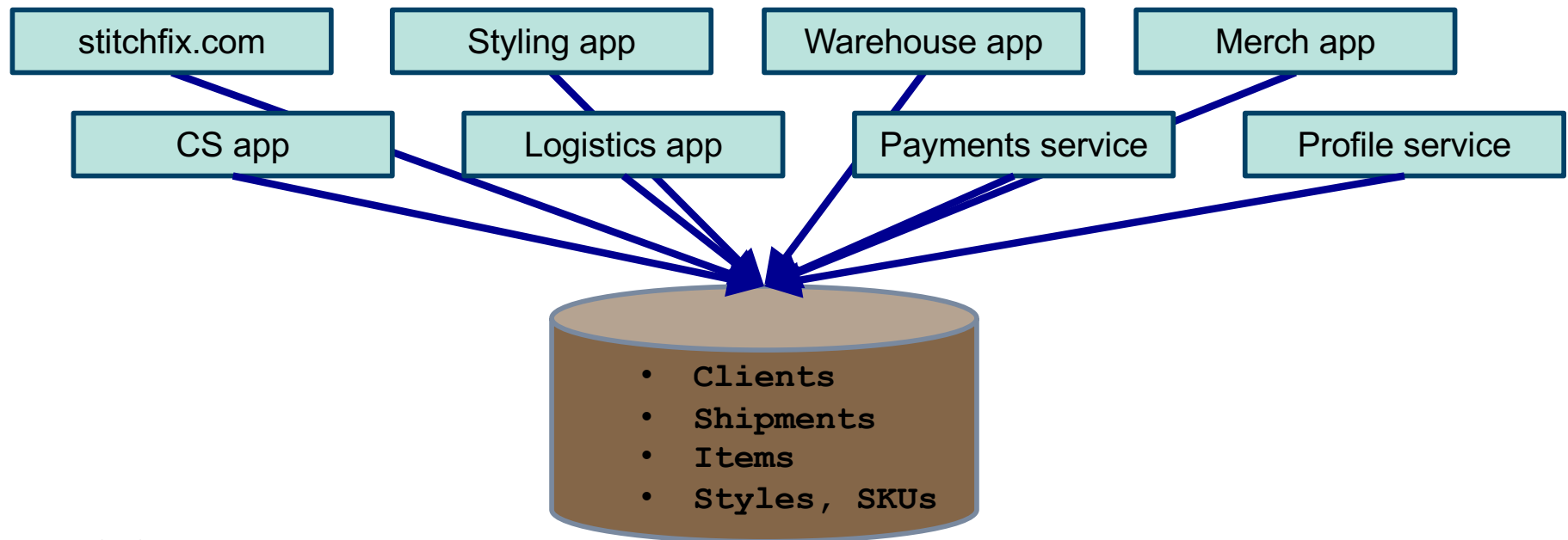
- Problem: Monolithic shared DB



@randyshoup

Extracting Microservices

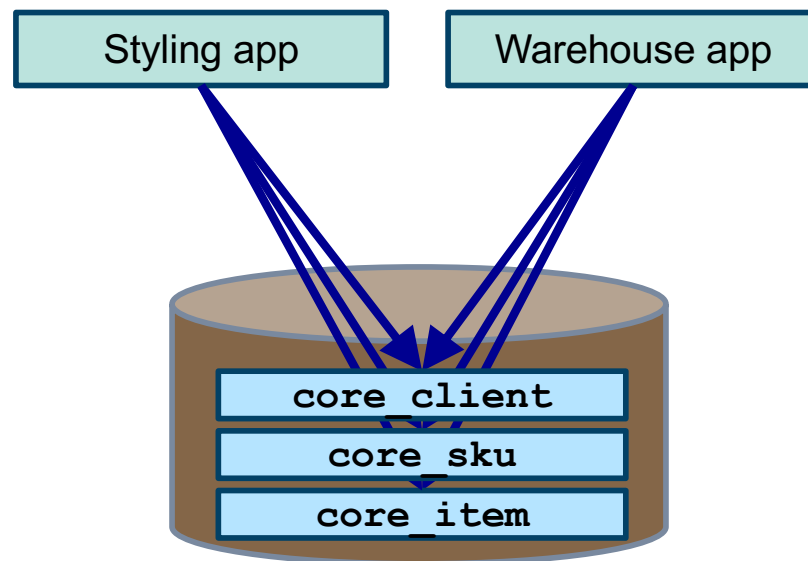
- Decouple applications / services from shared DB



@randyshoup

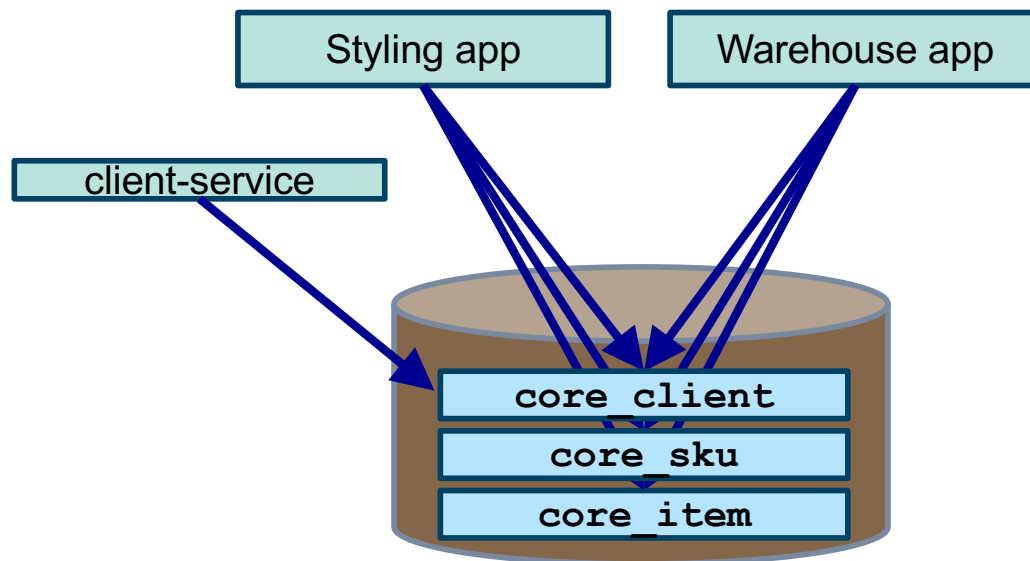
Extracting Microservices

- Decouple applications / services from shared DB



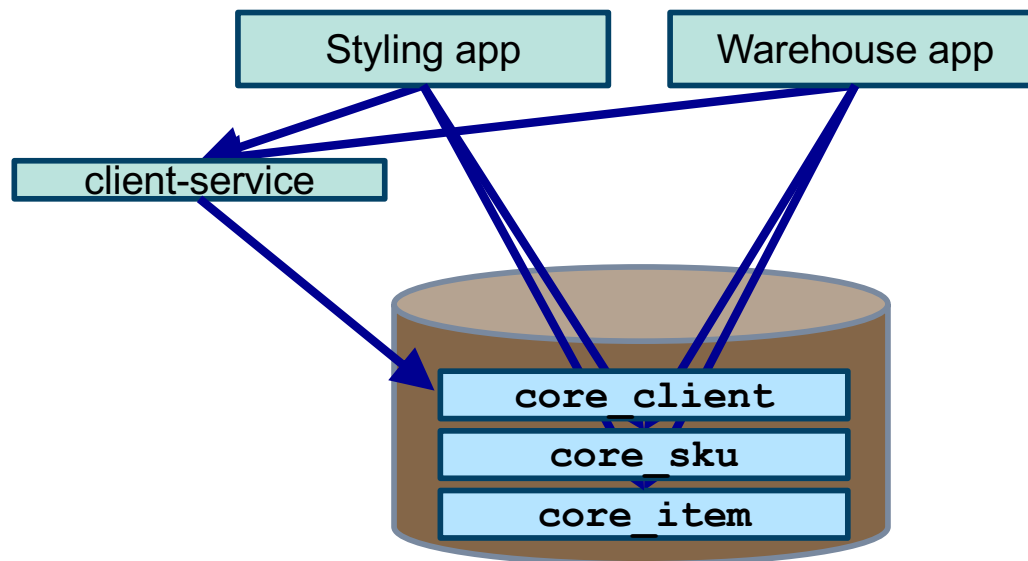
Extracting Microservices

- Step 1: Create a service



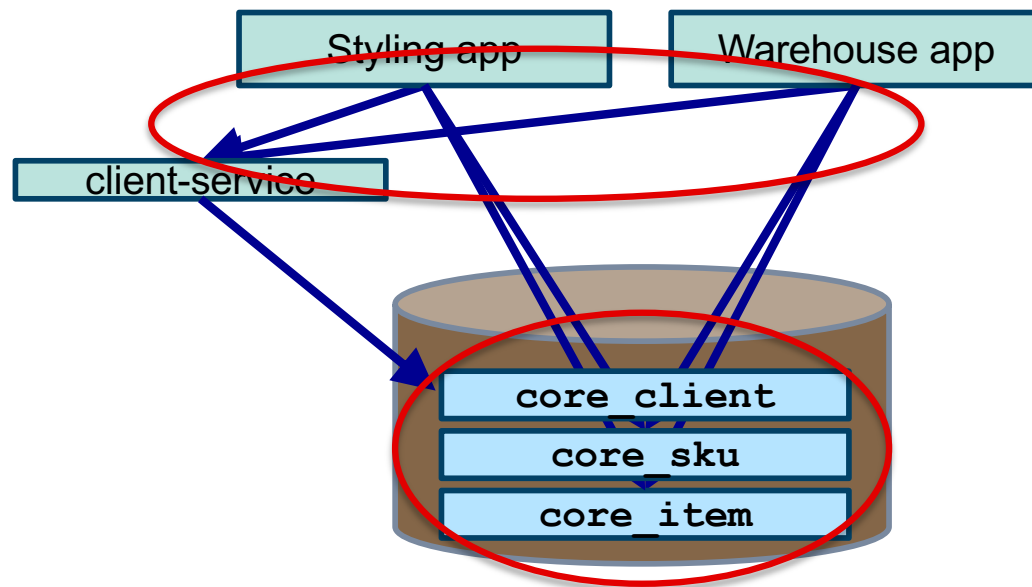
Extracting Microservices

- Step 2: Applications use the service



Extracting Microservices

- Step 2: Applications use the service

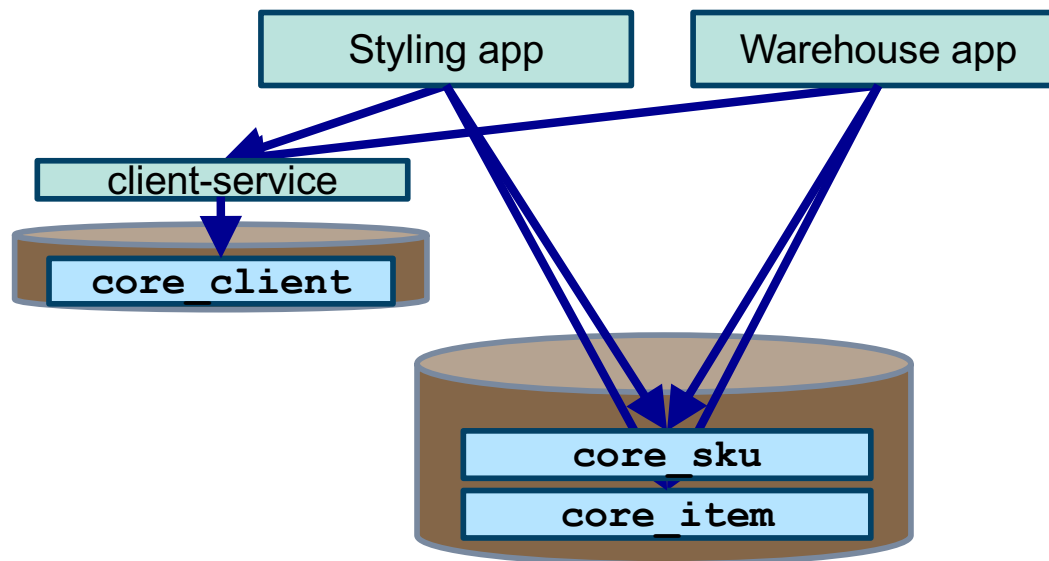


Do NOT stop here!

- ① All the problems of a distributed system
- ② All the problems of a shared database
- ③ None of the benefits of microservices ☹️

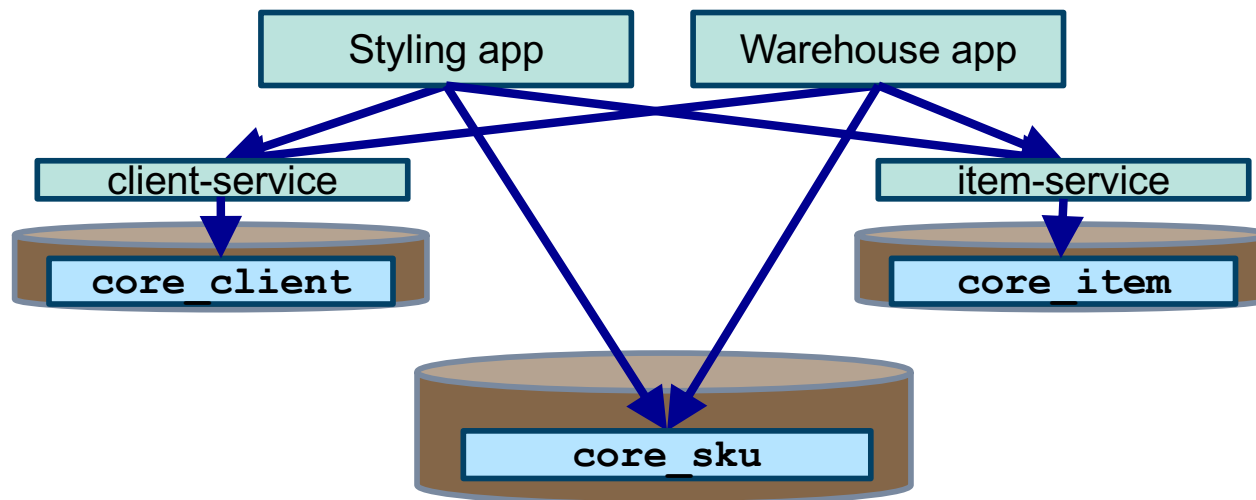
Extracting Microservices

- Step 3: Move data to private database



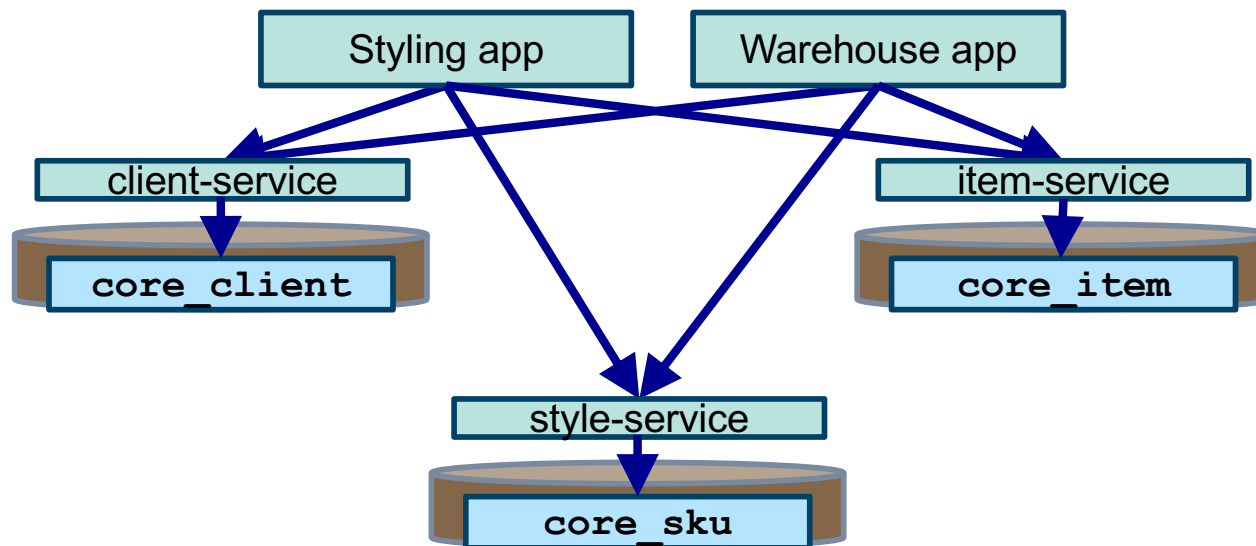
Extracting Microservices

- Step 4: Rinse and Repeat



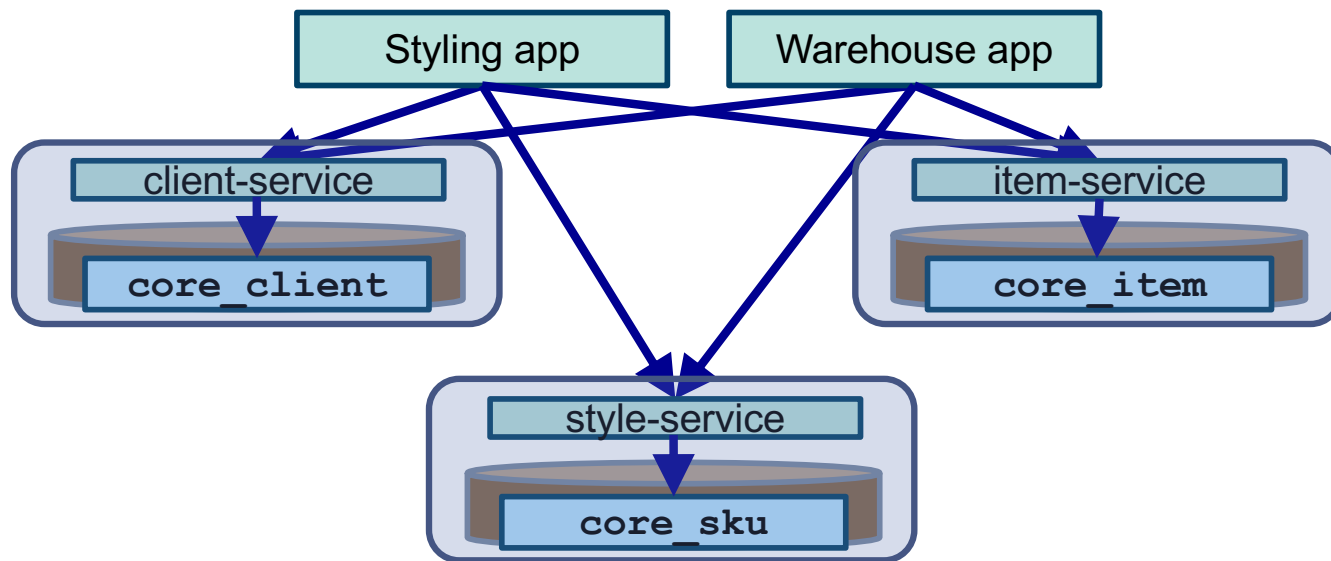
Extracting Microservices

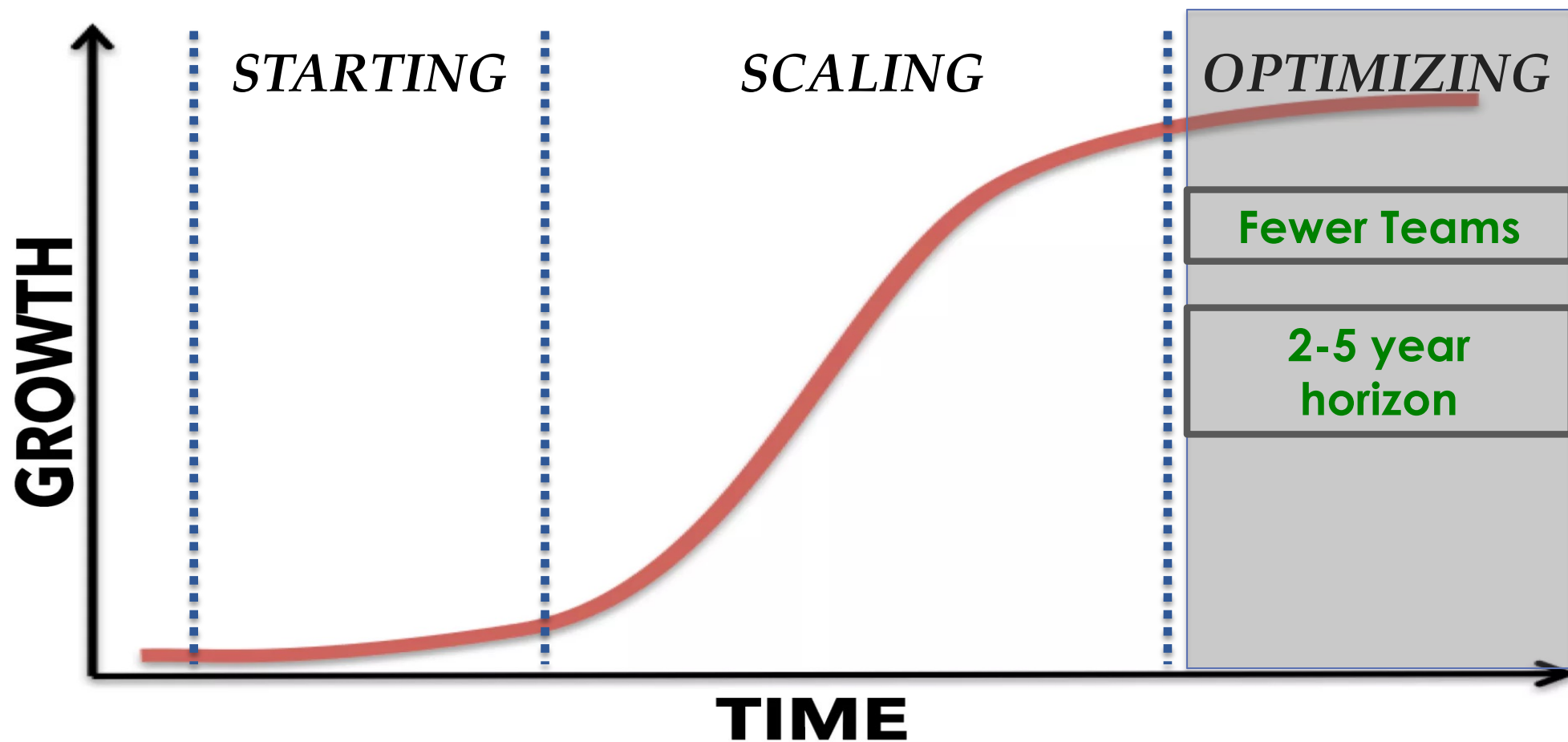
- Step 4: Rinse and Repeat



Extracting Microservices

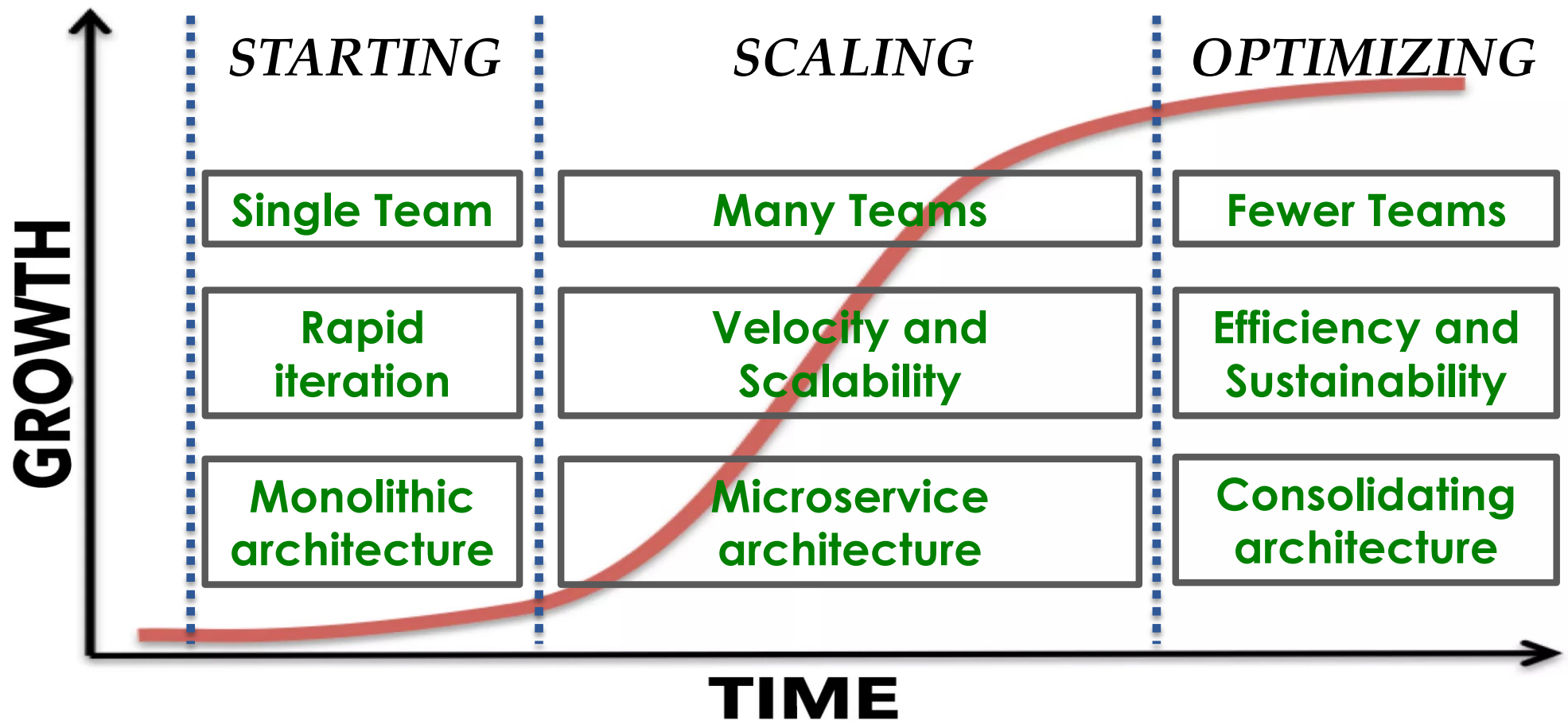
- Step 4: Rinse and Repeat

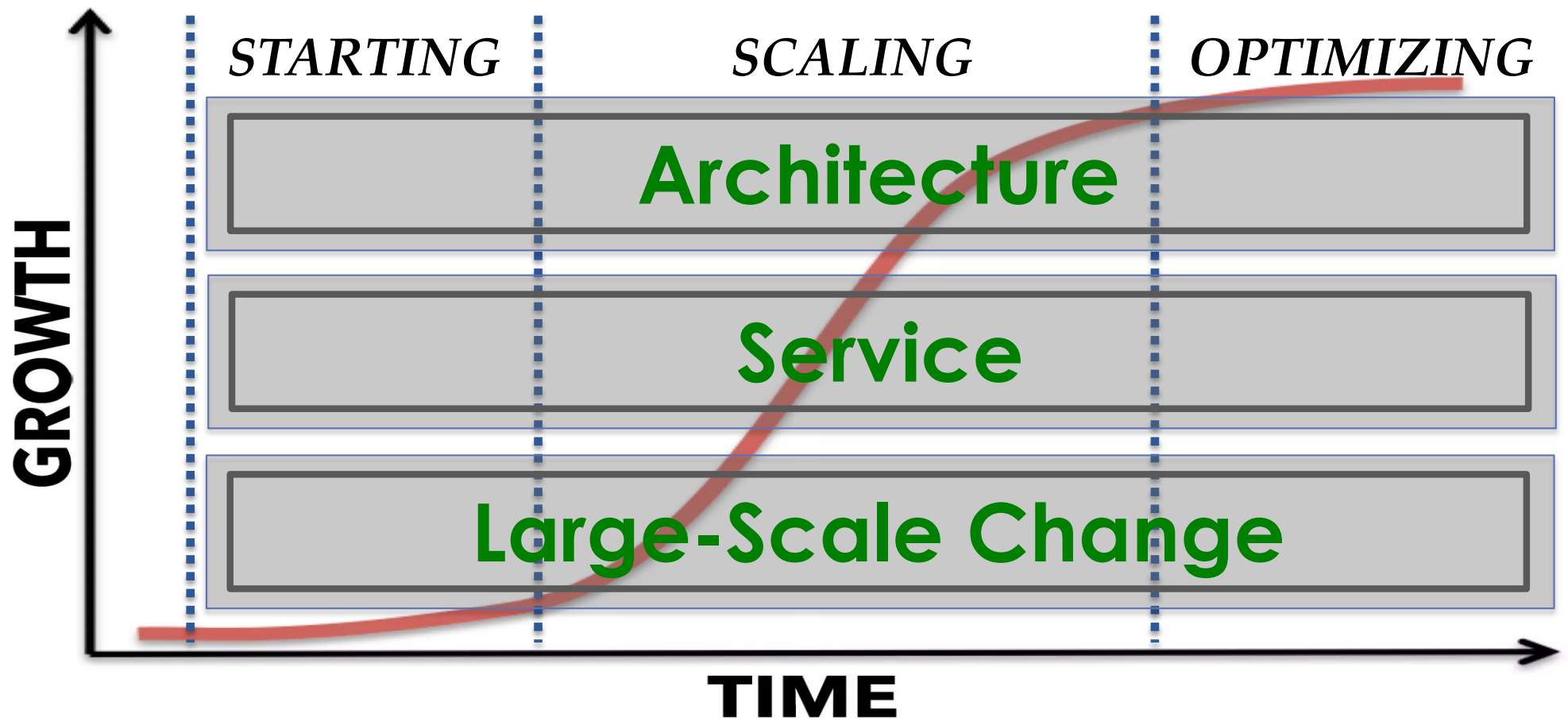




Stable Architecture

- Goal: Make a stable system more sustainable, efficient, effective
- Sustainable, incremental improvements in functionality
- Sustainable improvements in operational efficiency
- Consolidating the Team(s)





Thank you!



@randyshoup



[linkedin.com/in/randyshoup](https://www.linkedin.com/in/randyshoup)



medium.com/@randyshoup