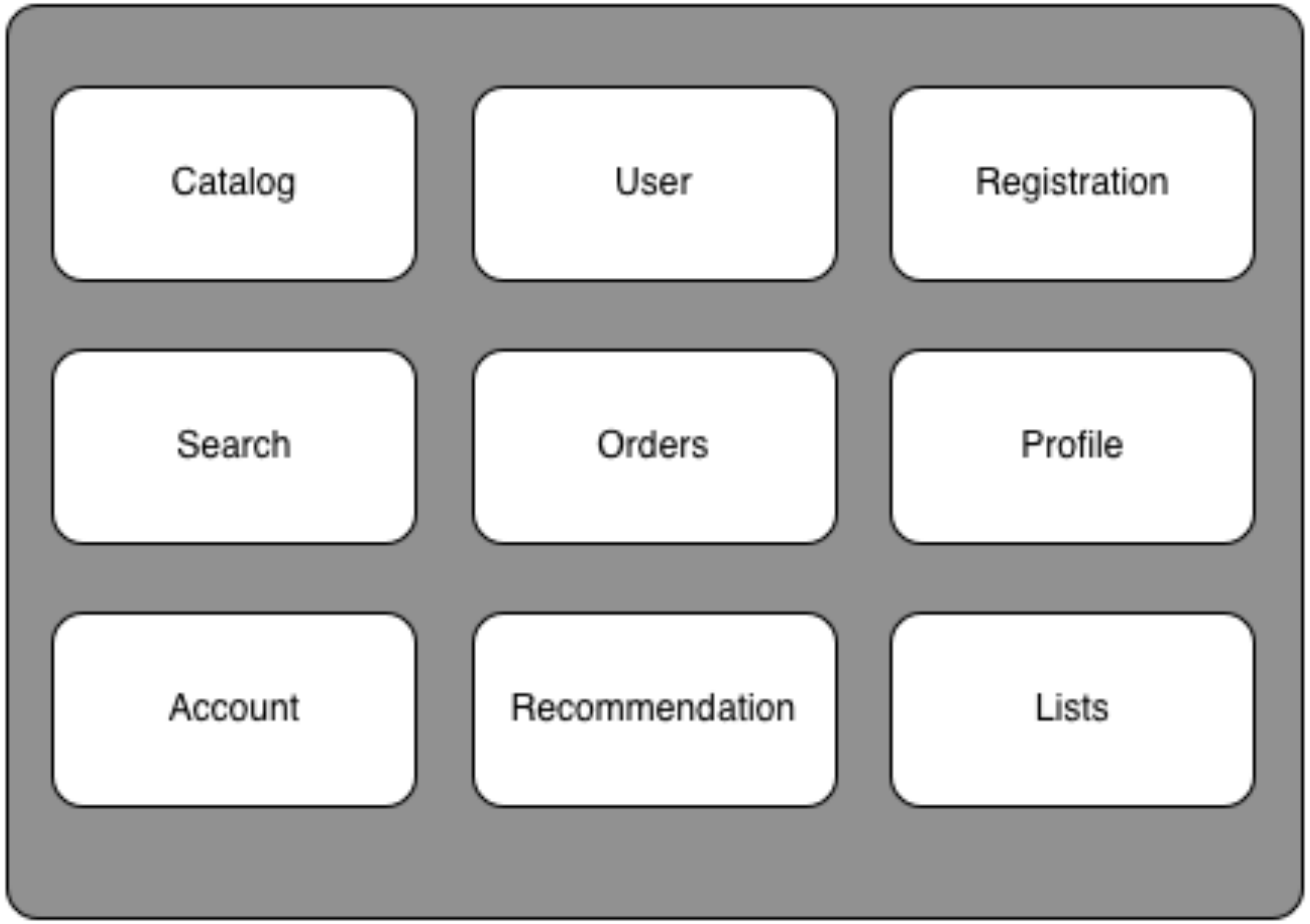
# *Characteristics of Monolithic Applications*

# The Monolith

- Application contains:
  - The UI
  - All the back-end logic to support the needs of the application
  - Often includes cross-cutting concerns:  authentication, admin user interface, dashboards, even scheduled jobs



Catalog

User

Registration

Search

Orders

Profile

Account

Recommendation

Lists

# Typified by..

- Large body of code
- Self-contained application
- Many teams could be working on a single codebase
- Coordination of work not trivial
- Releases relatively infrequent

# *Problems*

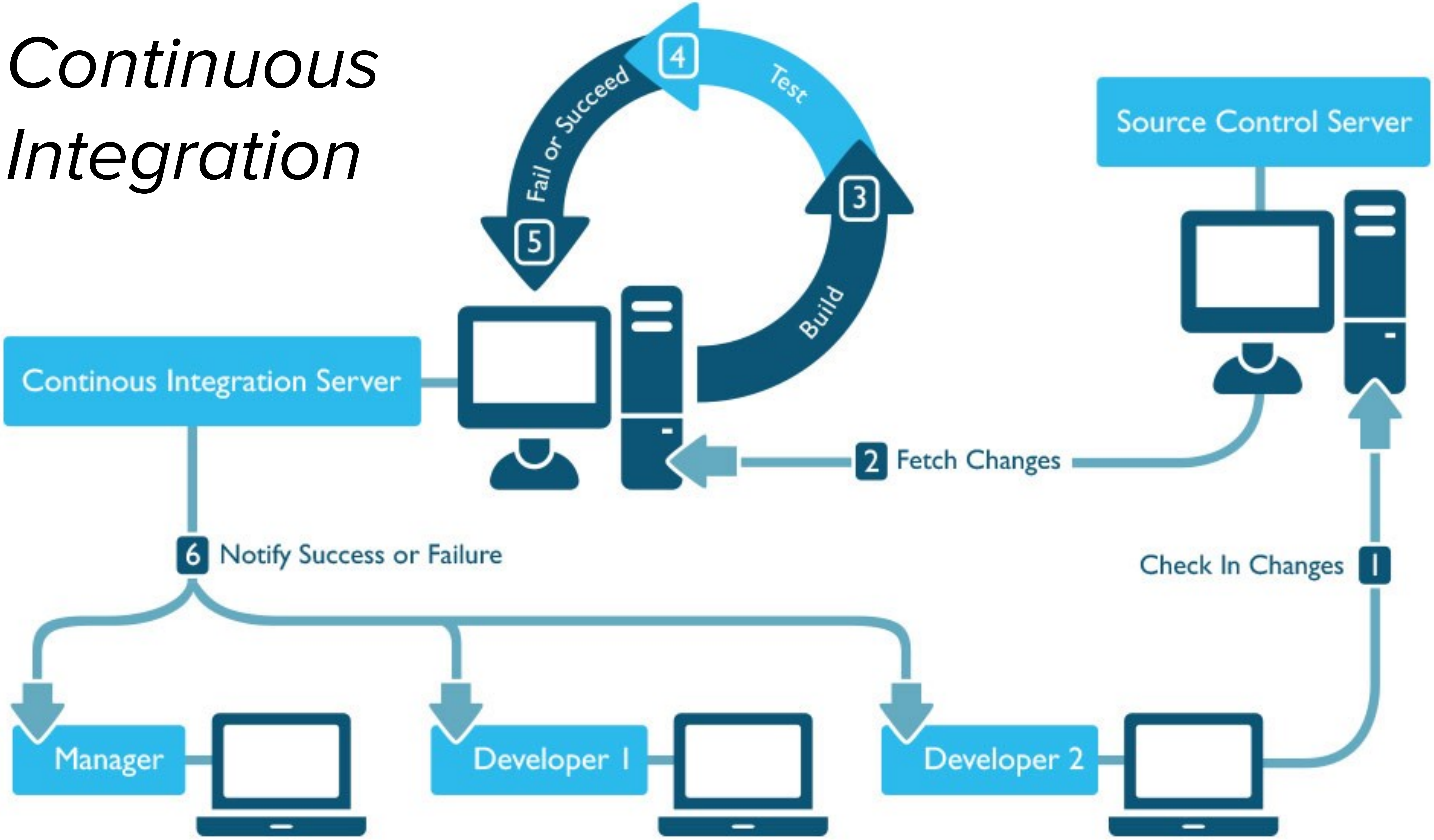
*with*

*Monolithic Applications*

# Coordinating Deployments

- When multiple teams deploy “into” the same war or ear file, their development efforts are coupled and must be coordinated
- Example: team A is ready to deploy a new feature, it often must wait until all other teams are ready as well
- There are ways to mitigate this, for example: using feature toggles

# *Continuous Integration*



# Continuous Integration

The process of integrating your work with the rest of the team:

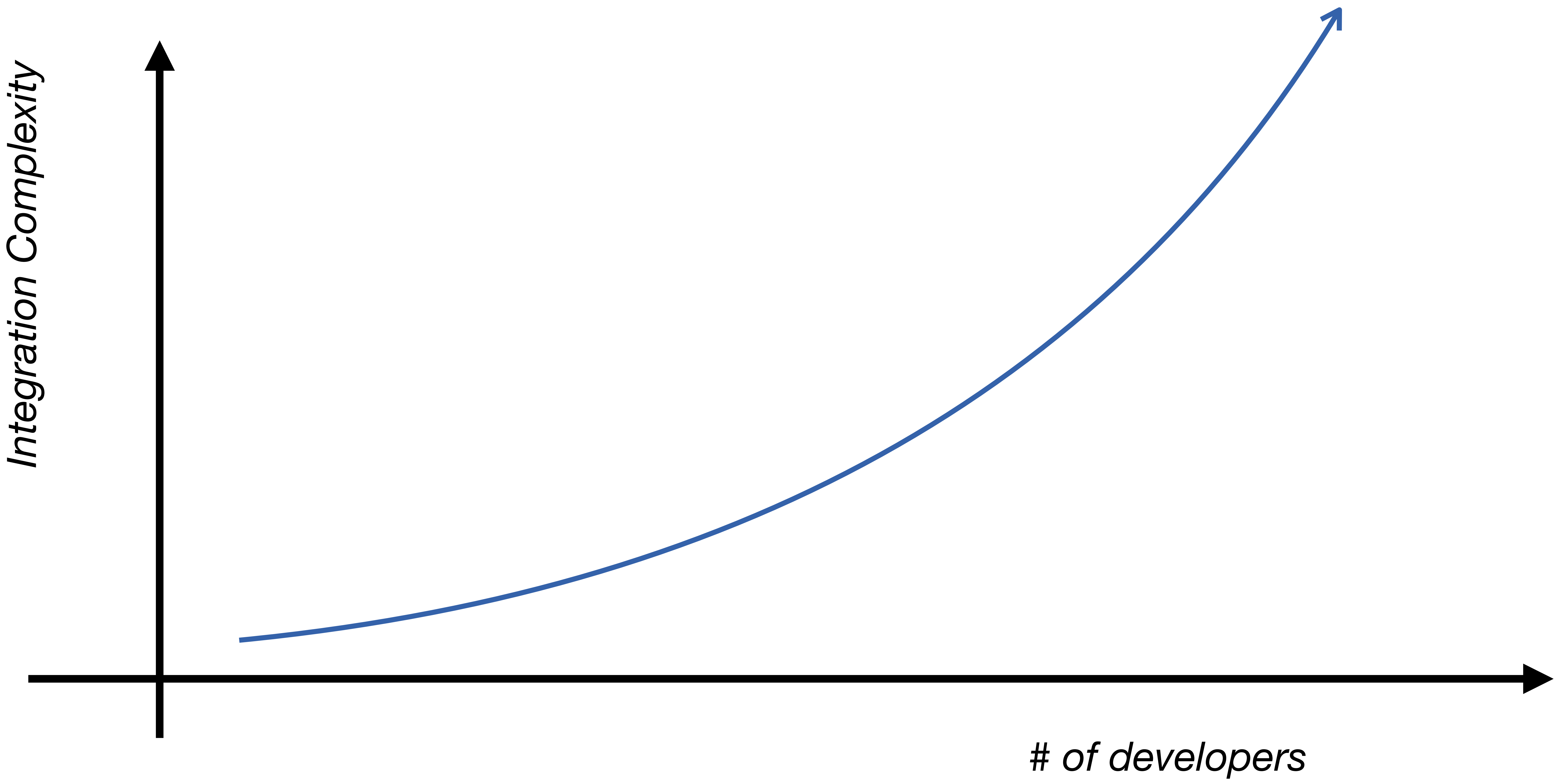
- Pull latest version of code from version control
- Implement a feature or bug fix (with tests)
- Run tests, see them pass
- Merge upstream changes
- Re-run tests, see them pass
- Push your changes upstream

*https://martinfowler.com/articles/continuousIntegration.html*

# Relationship between team size and difficulty of doing CI?

- Experiences working on a large team?
- How easy is it to commit your code into the mainline?
- How often do you have to merge others’ code changes into your copy of the codebase?
- How different would it be on a 6-person team vs a 60-person team?

# Integration Complexity as a function of # of developers



*With monolithic applications..*
*because we have a large number of developers,*
*integration is harder.*

*This affects velocity*

*With monolithic applications..*
*because coordination is harder,*
*occasions where all features can be deployed are less frequent*

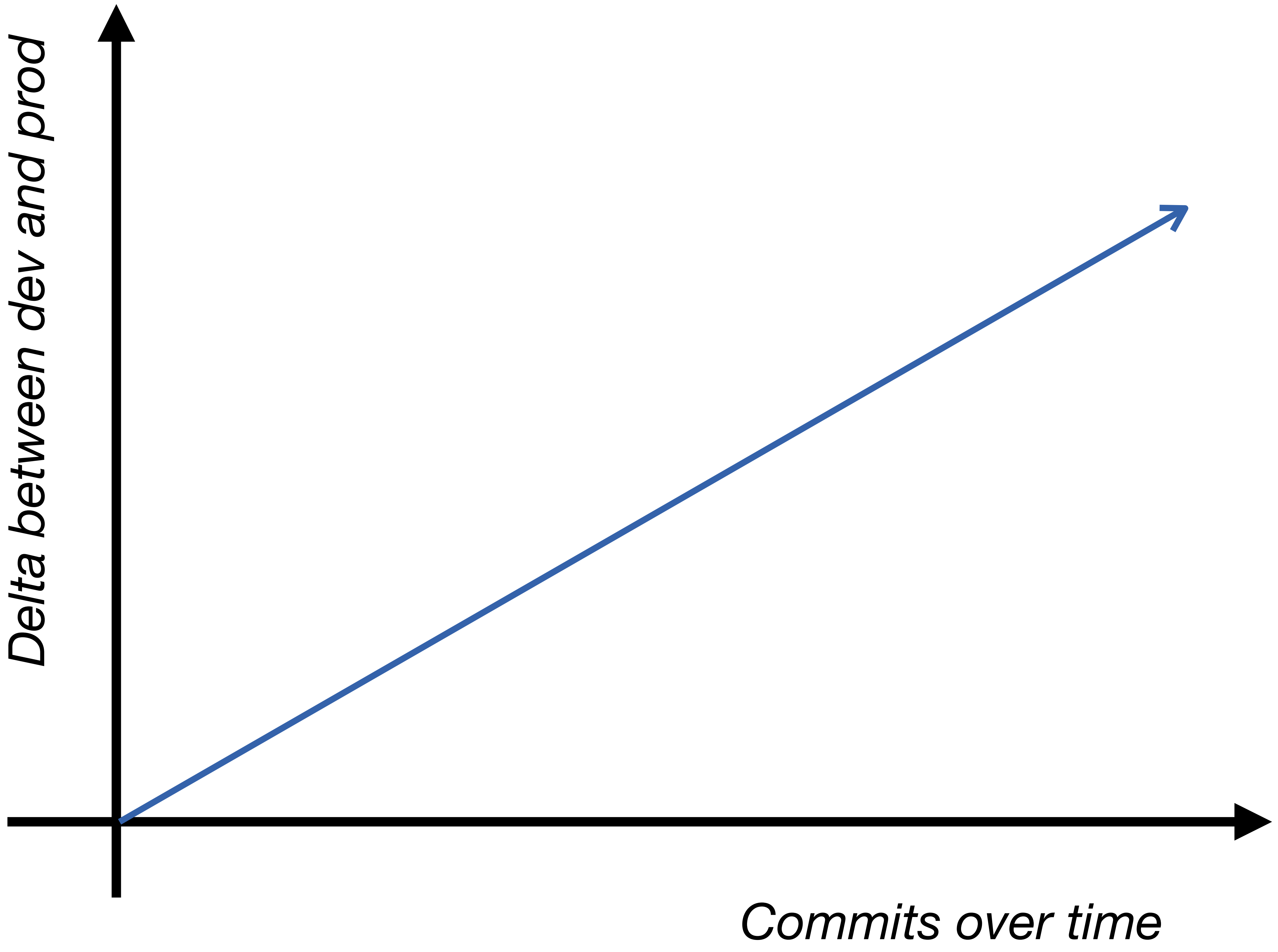
# ..requires deployment processes be put in place:

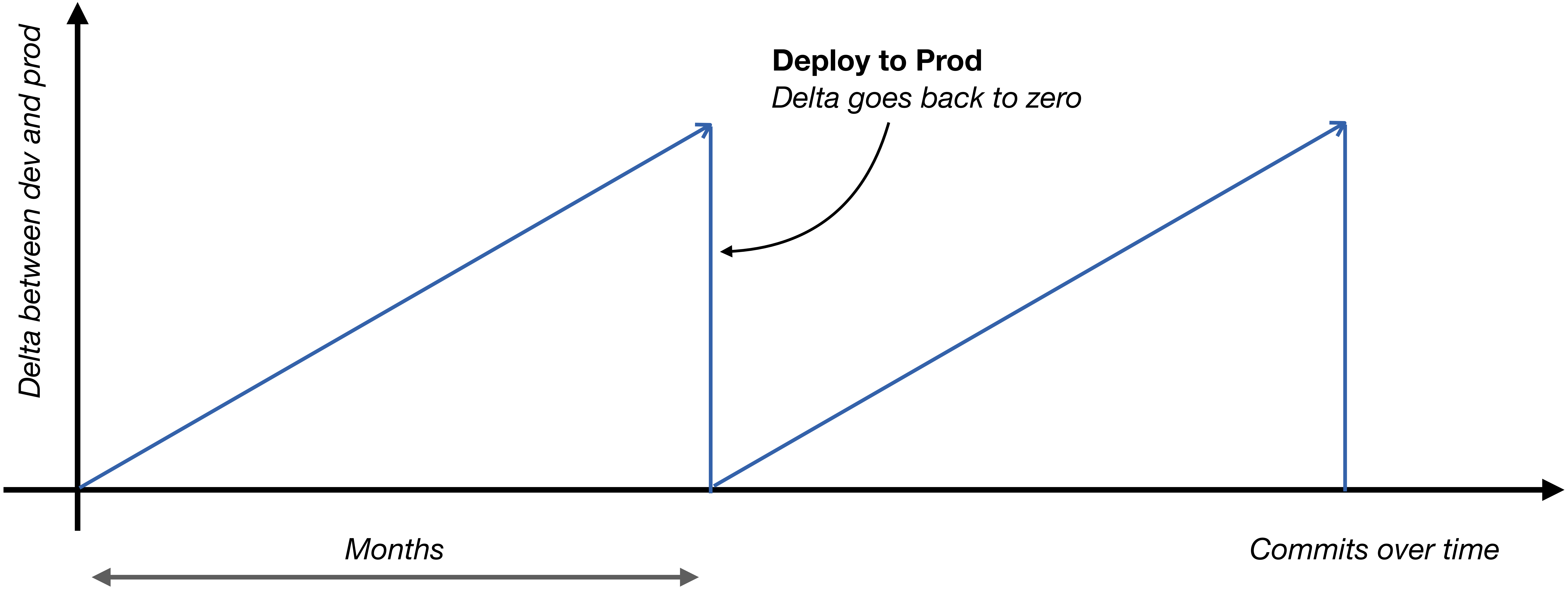
- Scheduled, coordinated deployments
- Code freezes
- Necessitates the creation of branches, which must be merged back into the mainline, and further complicates integration
- Higher likelihood of deployment delays

# How big a change are we deploying to production?

We can reason about deploying a single commit to production

It's less straightforward to reason about the effect of a deployment when it represents 1,000 commits made by a half dozen teams over a period of six months

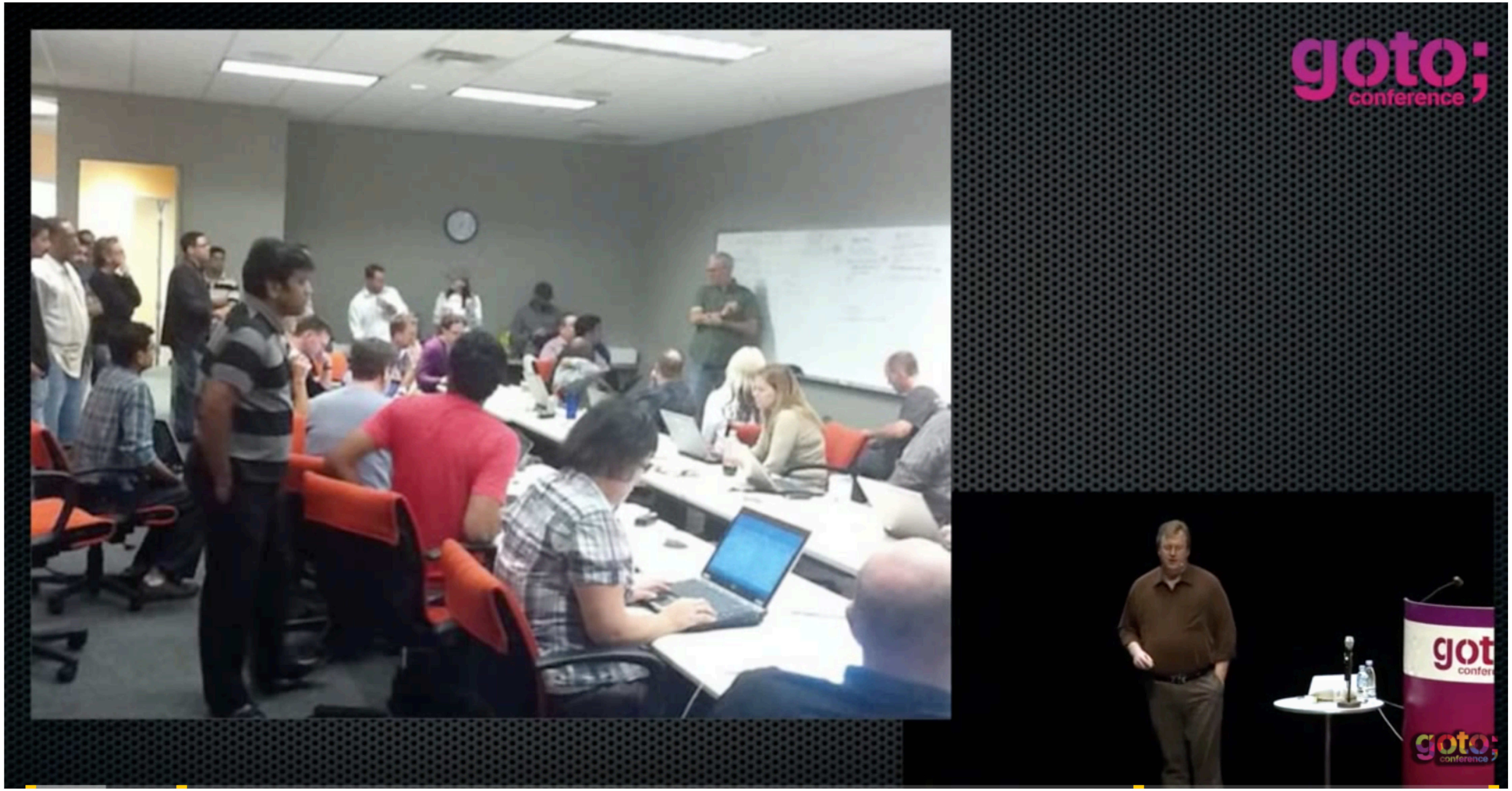



Delta between dev and prod
**Deploy to Prod**
*Delta goes back to zero*
*Months*
*Commits over time*

- Less frequent releases imply greater risks with each deployment
- A certain degree of fear associated with deployments to production
- Often involves working late hours, and a deployment becomes an event, involving a lot of people..

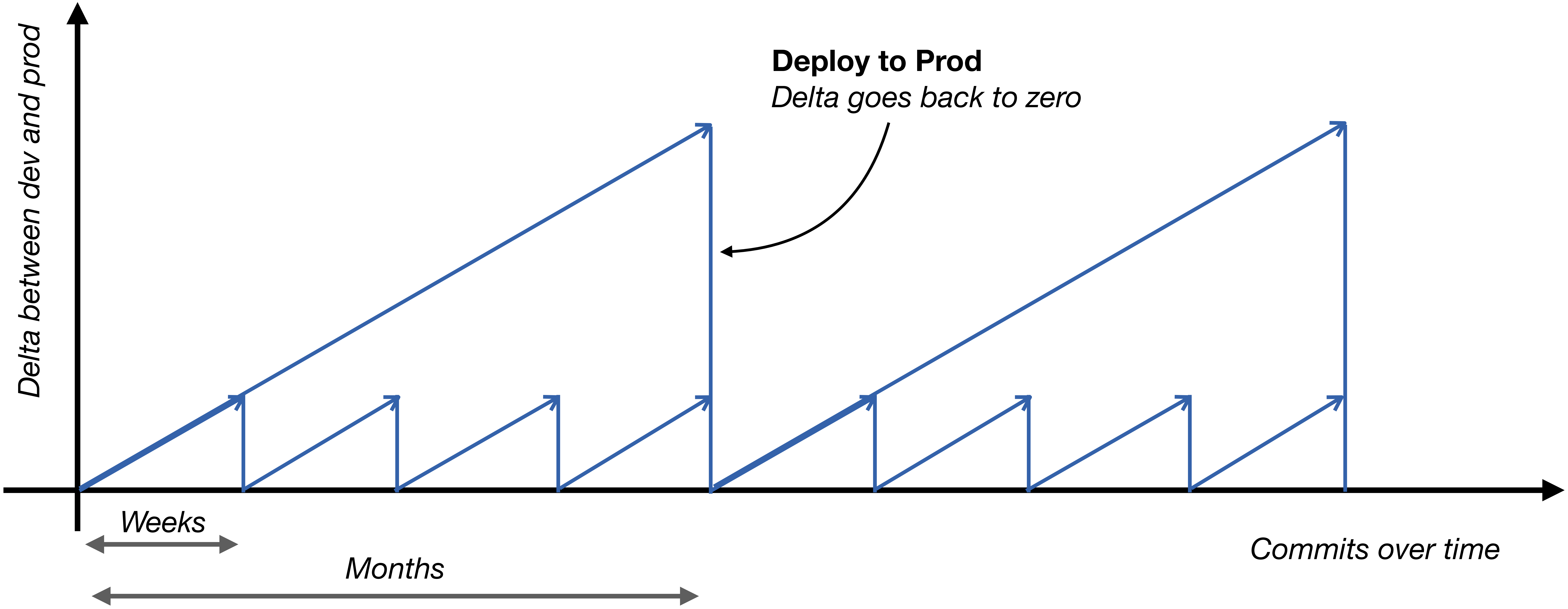
# Disband the Deployment Army

*Michael Nygard*



https://www.youtube.com/watch?v=Luskg9ES9qI

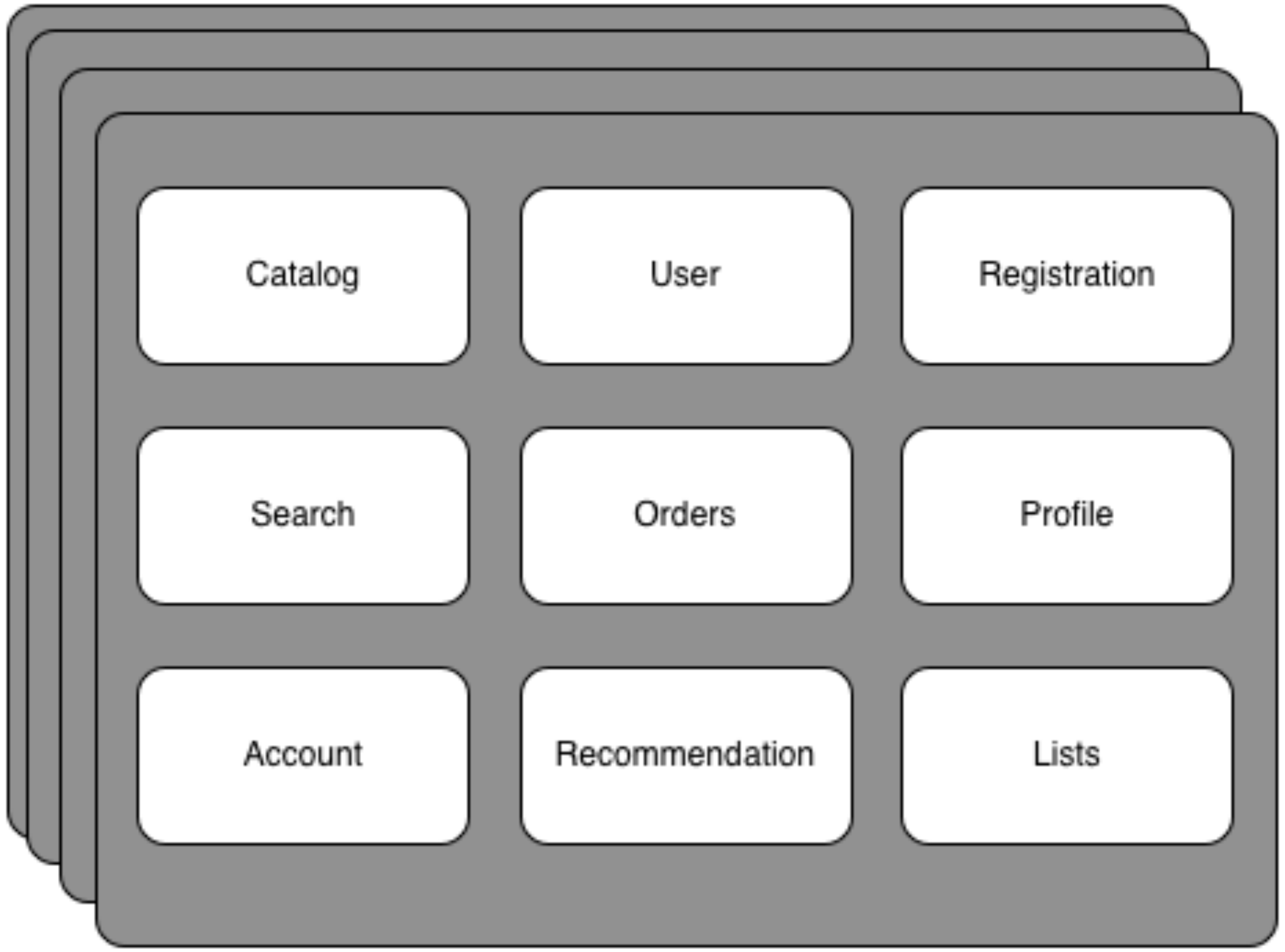
# Increasing the Frequency of Deployments



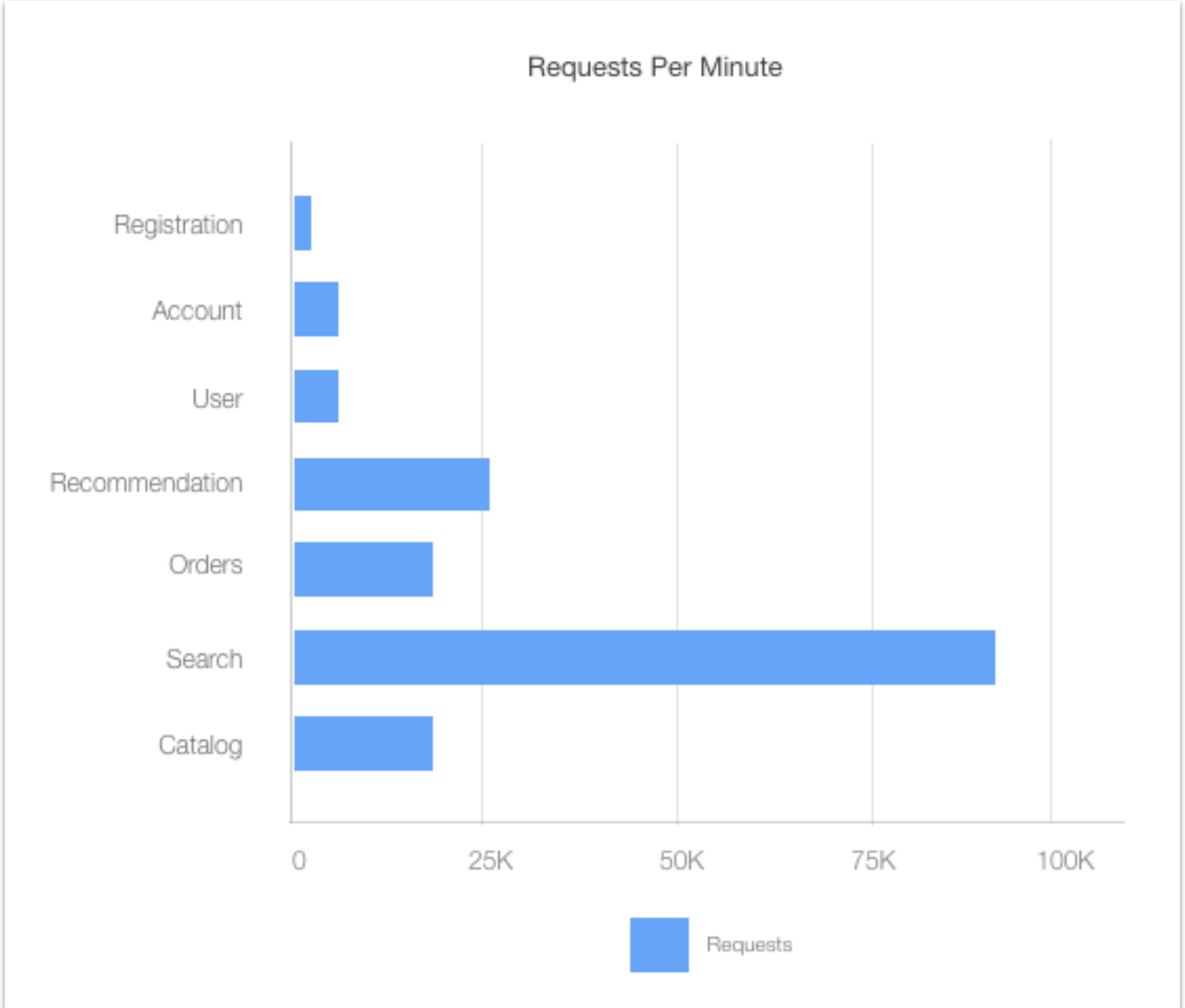
Notice how the *area under the curve* is significantly smaller when deployments are more frequent

# Scaling

- How do we scale a monolithic application?
- Cannot scale each component independently
- Scaling monoliths is usually not resource efficient



*How we should scale..*



# Tolerance for Failure

- A bug in any of the logic in the monolithic application could bring down the entire process or application instance

- Any feature exhibiting poor performance affects the entire application

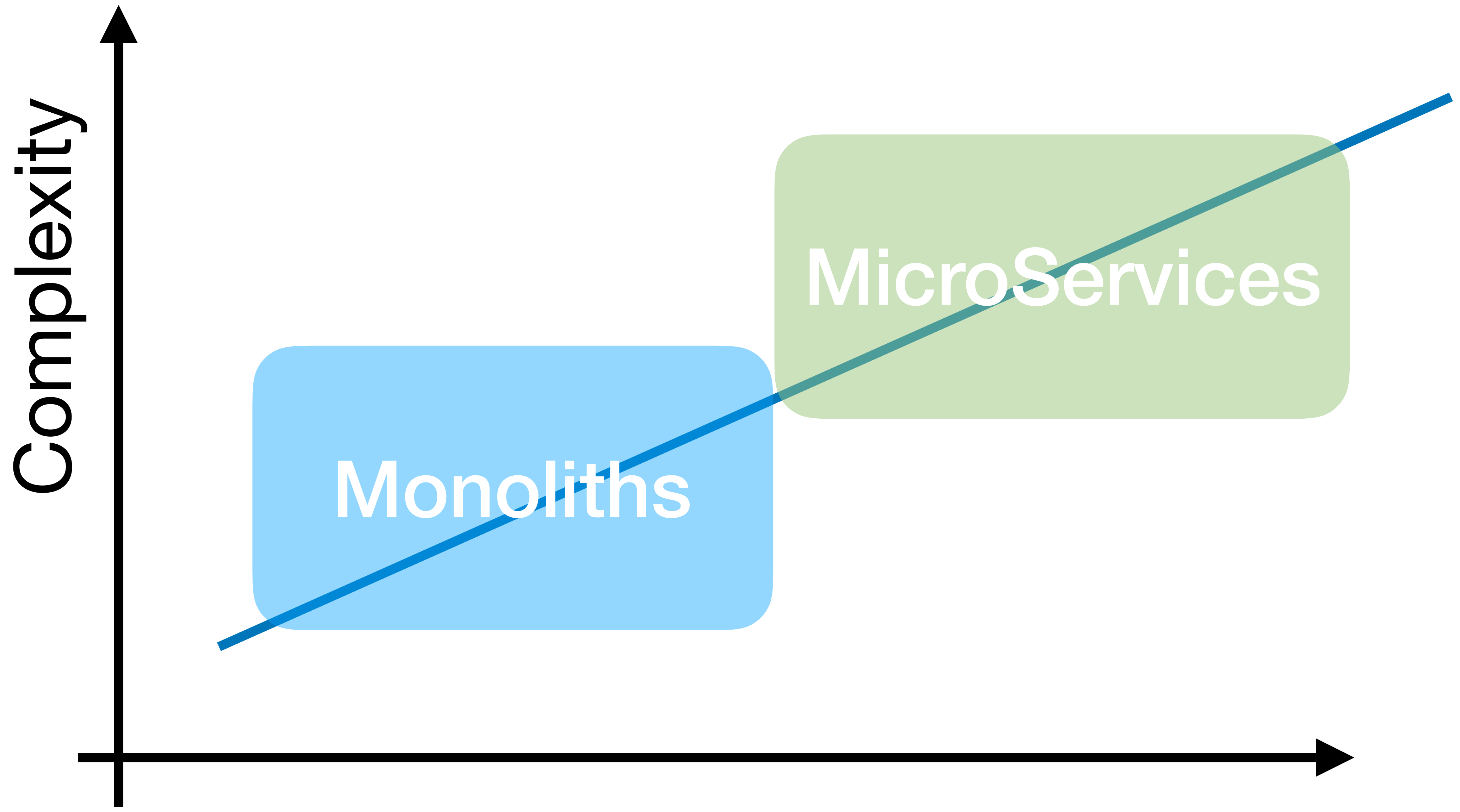
- Most monolithic applications are tested extensively, requiring time and effort, and contributing to less frequent releases

# Migrating to MicroServices

*Where to start?*

# Greenfield Applications

- It’s not always evident at the outset of a project how to organize or divide the domain into bounded contexts
- Often simpler to start with a monolith
- As the applications grows and evolves, look for obvious opportunities to extract MicroServices



# Legacy Applications

- Stop adding new features into the monolith.  Prefer to write new features as standalone microservices

- Eric Evans describes the *anticorruption layer*, an approach of integrating new code with old code in a way that does not corrupt the new model.  i.e. establish APIs and Contracts

- Refactoring approach:  the Strangler Pattern

# Strangler Pattern

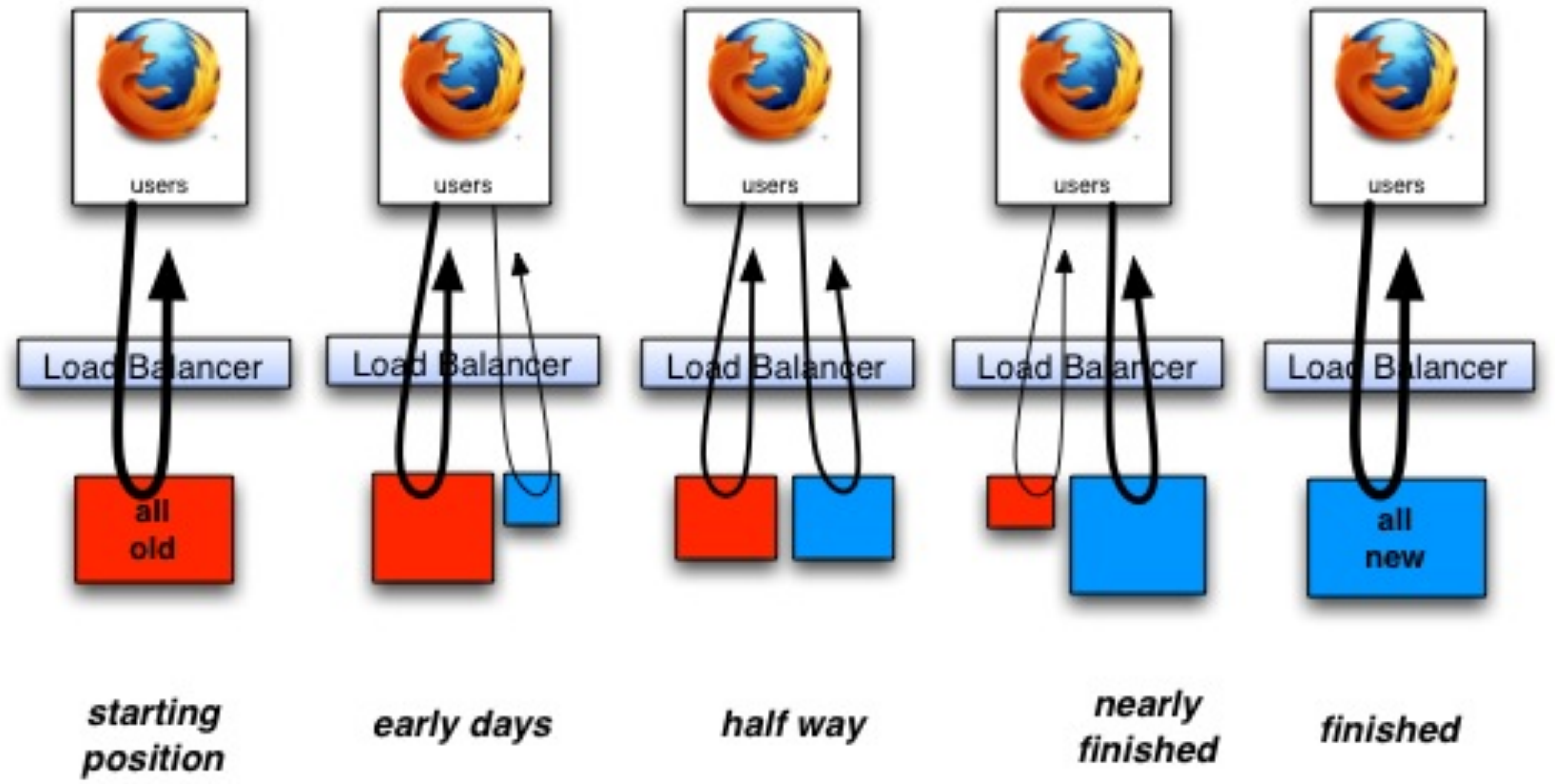
- Described by Martin Fowler in article named the “Strangler Application”
- The approach is akin to how strangler vines slowly grow around a tree, and slowly strangle the tree and take its place
- In software, it’s a refactoring strategy where we slowly replace legacy code with standalone microservices, and, over time “strangle” the monolith

*https://www.martinfowler.com/bliki/StranglerApplication.html*

- Idea of slowly shrinking a monolith by replacing some of its sub-domains with standalone microservices

- Involves the use of a proxy in front of the backing services that can be configured to direct requests to the new microservices as they are introduced

- Over time the monolith is either completely replaced or shrinks to a point where what remains is a much smaller and stable application



https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/