Monolift

Automating Distribution With the Tools You Have at Home

PLOS 2025

Tim Goodwin, Esteban Ramos, Lindsey Kuper, Andi Quinn

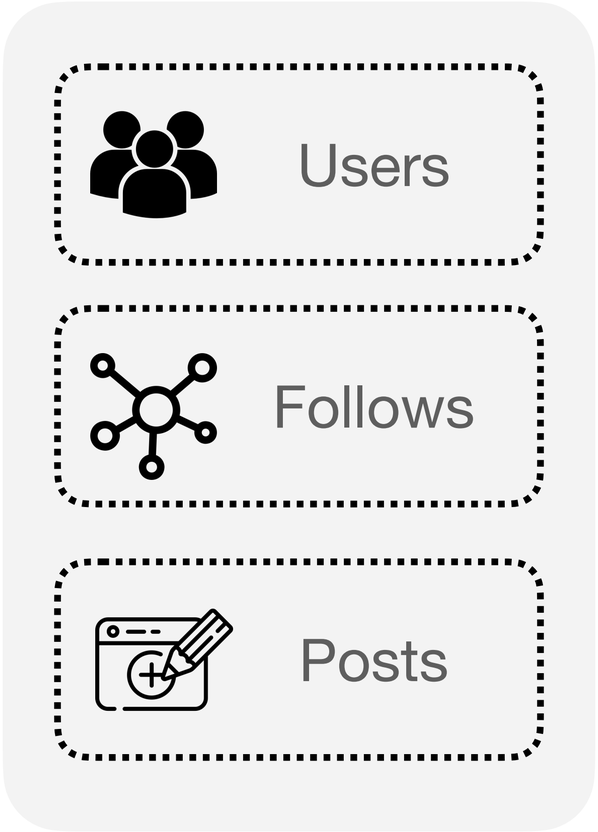
UNIVERSITY OF CALIFORNIA
SANTA CRUZ

Distributed systems in 2025

Are they any easier?

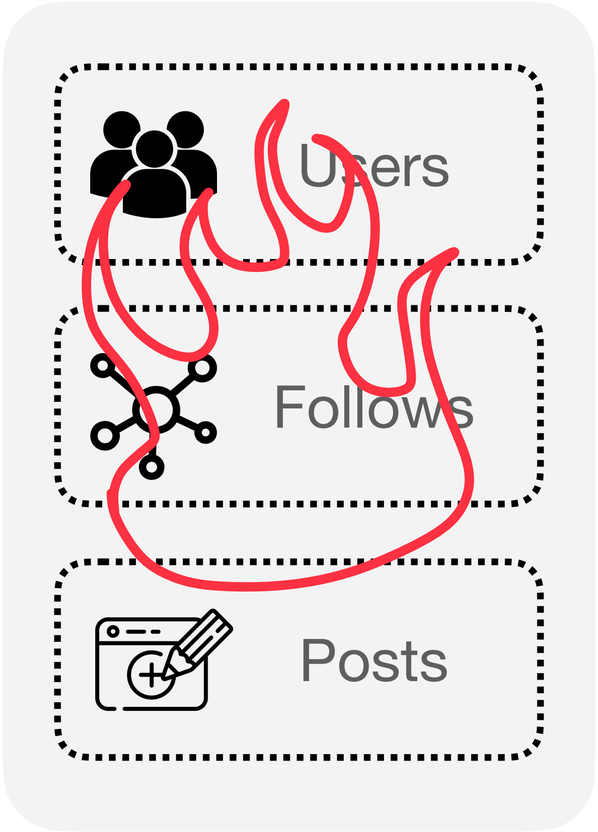
- Cloud computing makes scalability *easy*
  - Containers and orchestration platforms (e.g. Kubernetes)
  - Efficient RPC frameworks and data formats
  - Autoscalers and load balancers
- Conventional wisdom: split application into smaller pieces, deploy independently

The standard approach



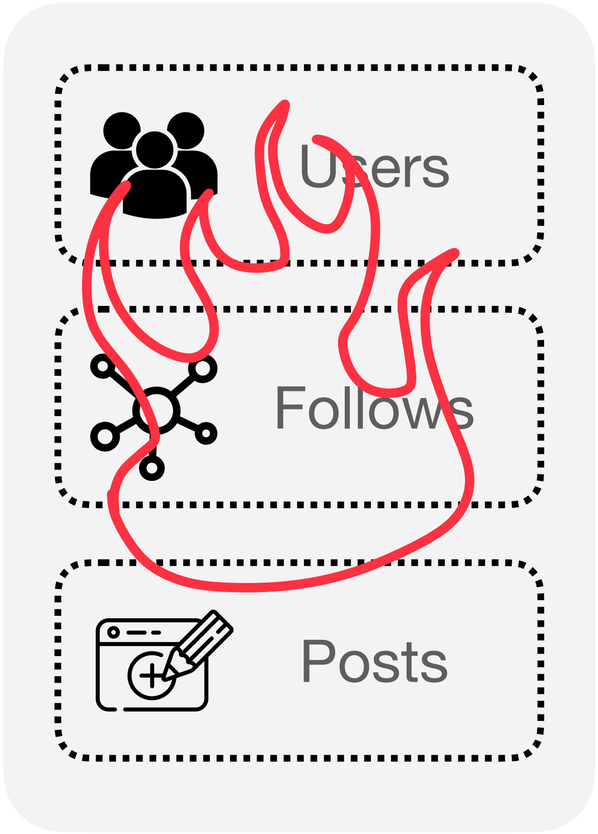
Monolith

The standard approach

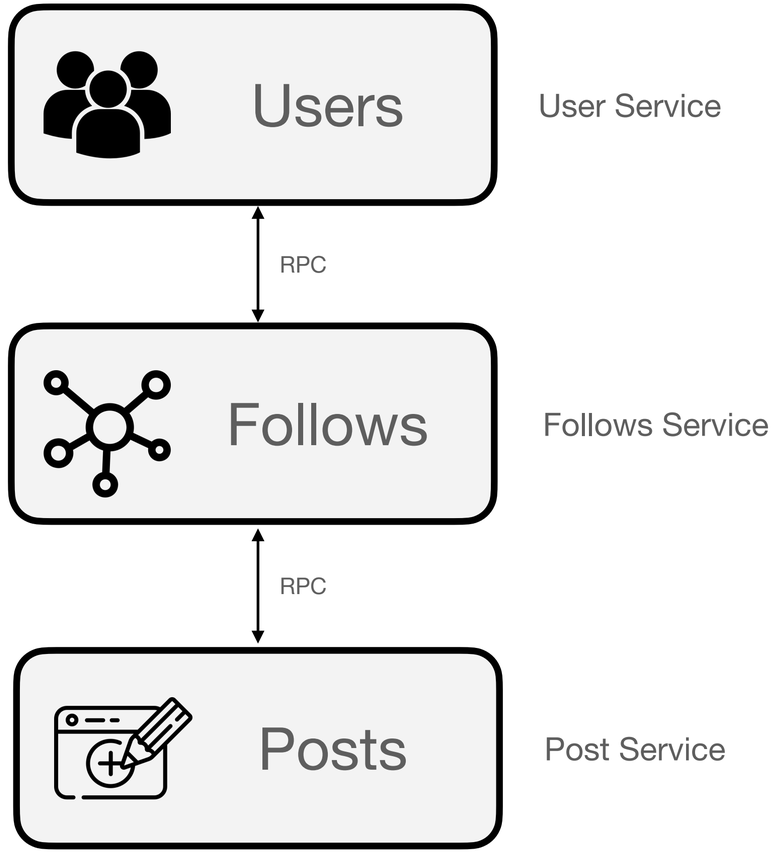


Monolith

The standard approach

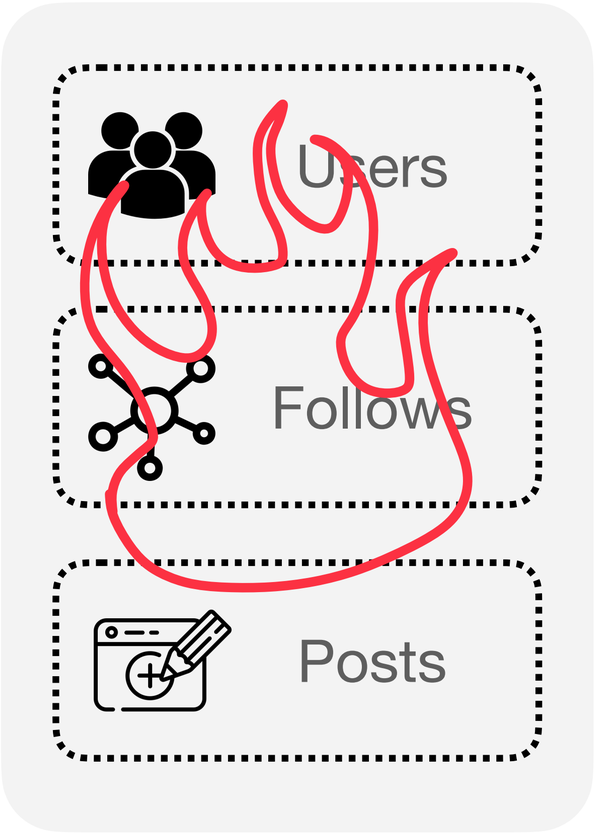


Monolith



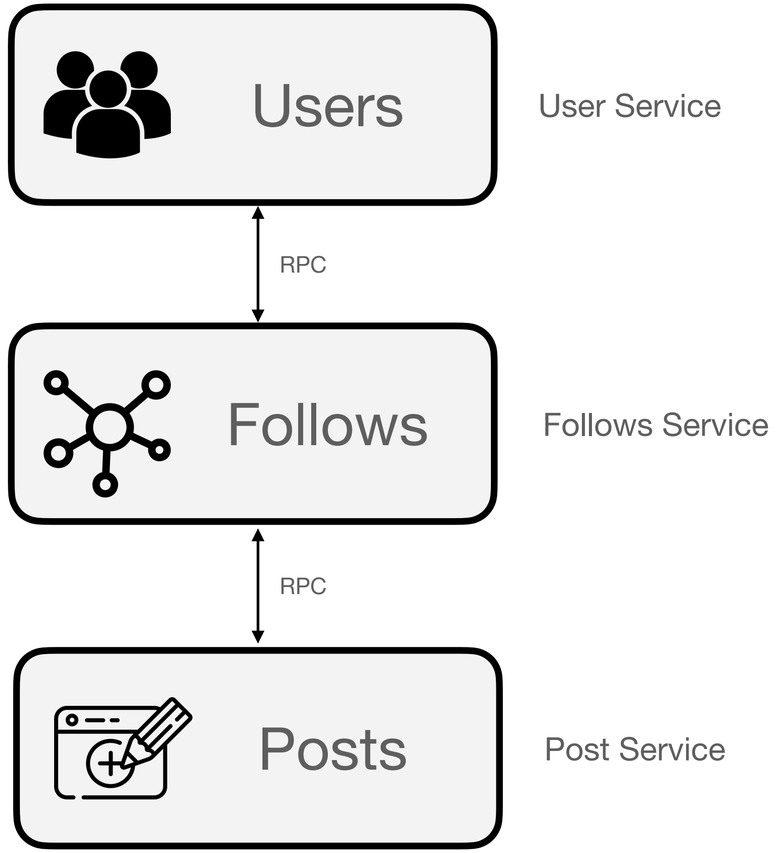
Microservices

The standard approach



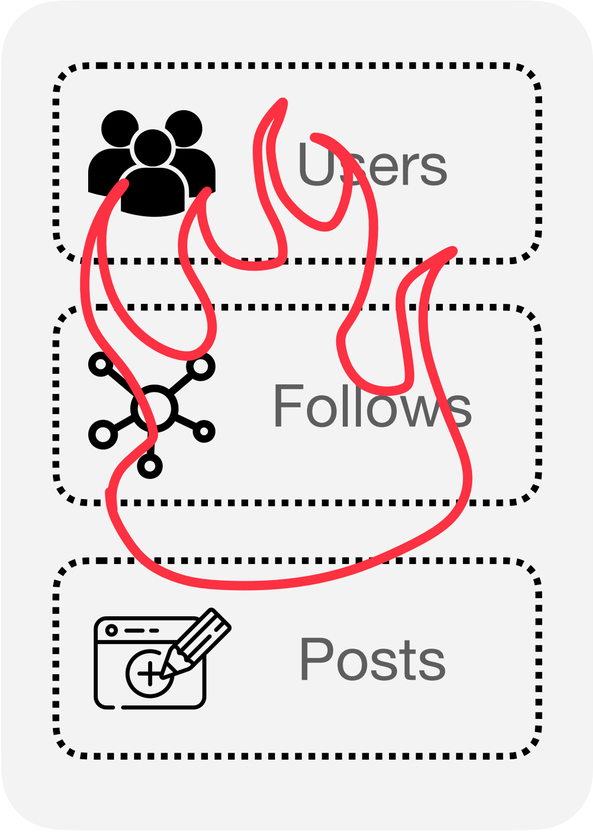
Monolith

Challenge 1: Labor-intensive refactors



Microservices

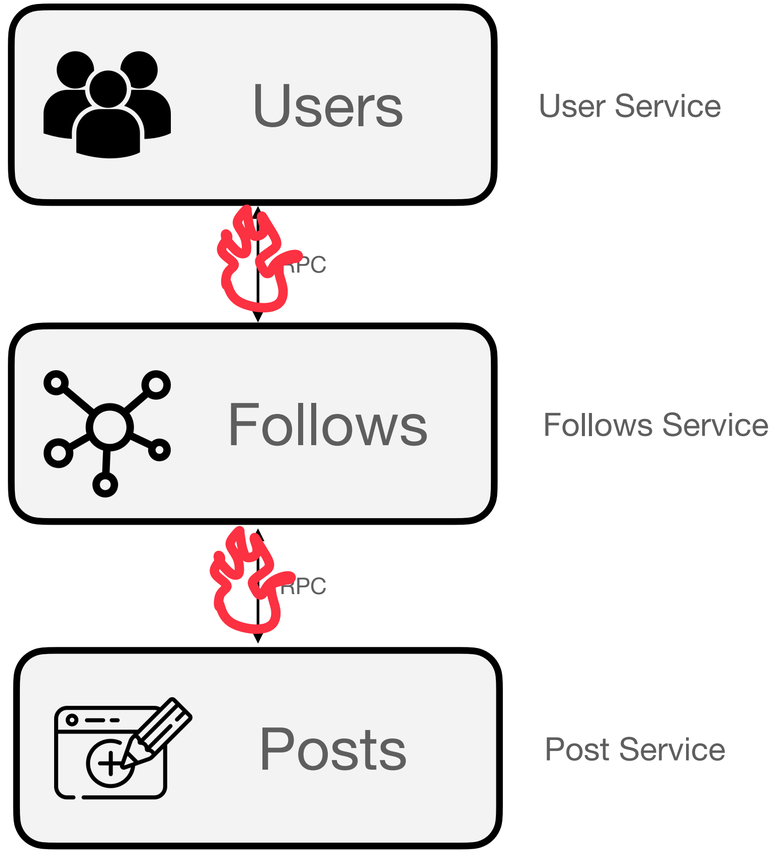
The standard approach



Monolith

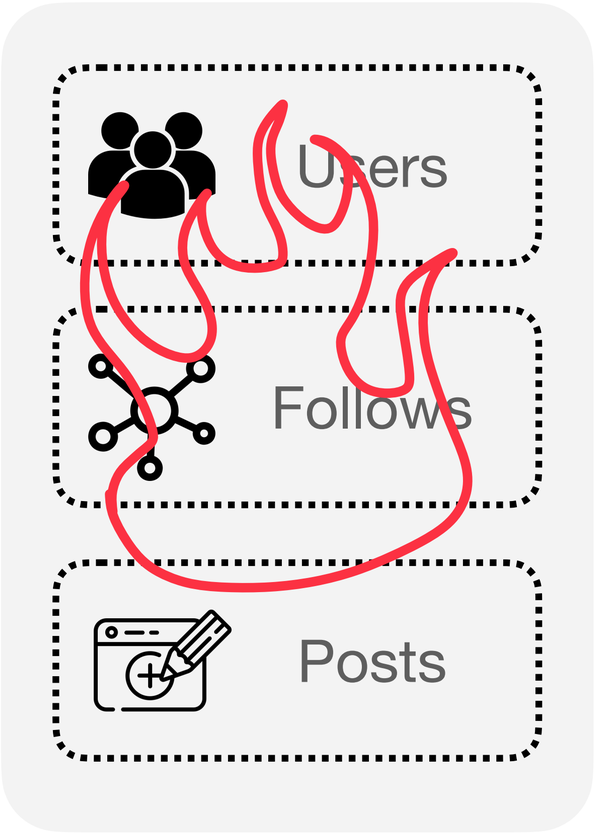
Challenge 1: Labor-intensive refactors

Challenge 2: Unpredictable outcomes



Microservices

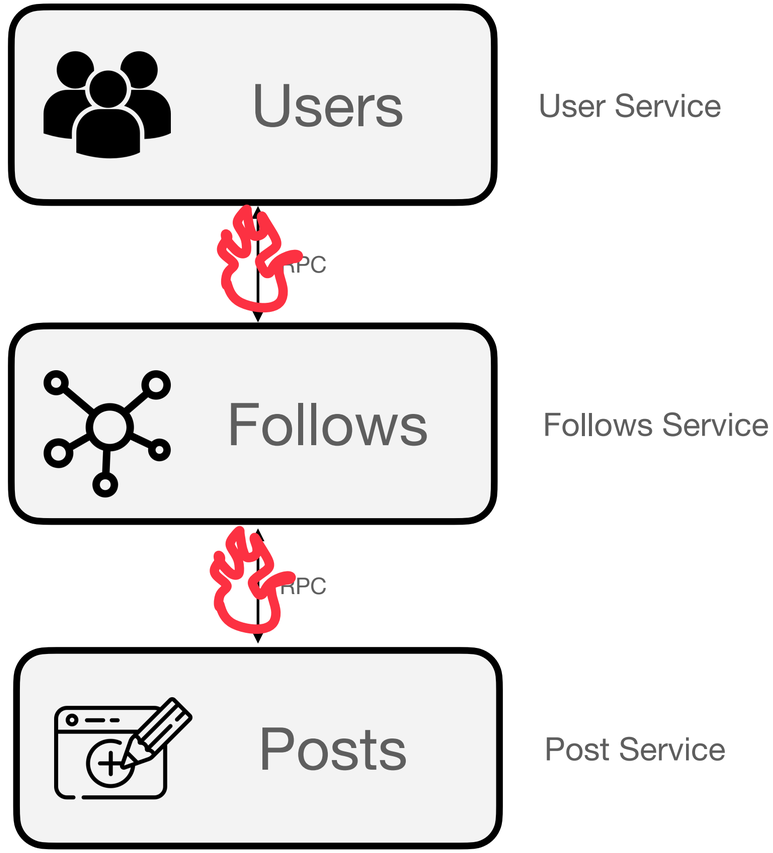
The standard approach



Monolith

Challenge 1: Labor-intensive refactors

Challenge 2: Unpredictable outcomes



Microservices

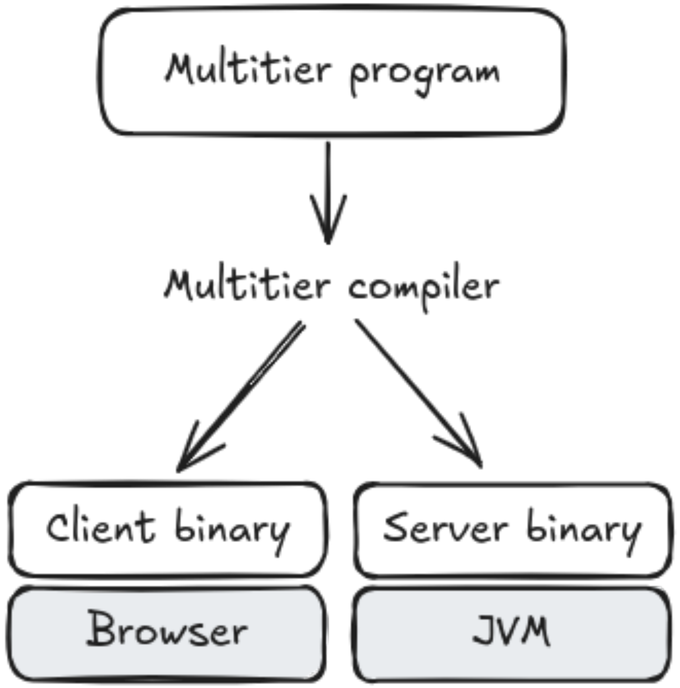
Core issue: microservices couple *development* model to *deployment* model

Decouple development from deployment with higher-level abstractions

Decoupling Development from Deployment

Multitier Programming

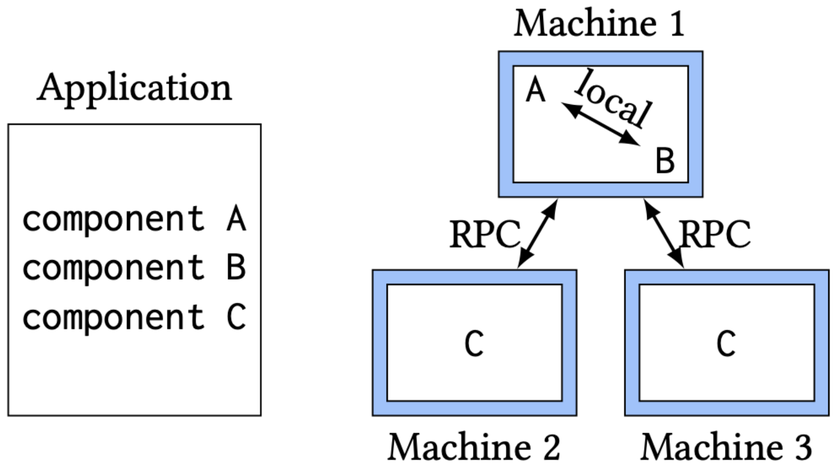
- Emerged in early 00’s to simplify web development
- Write distributed application as a *single program*, compiler generates distributed system
- Examples:
  - Hop.js (00’s)
  - Ur/Web (2015)
  - ScalaLoci (2018)



Decoupling Development from Deployment

Distributed Component Frameworks

- Express application logic in terms of predefined framework components
- Separate runtime handles how components are deployed and integrated
- Examples:
  - Microsoft DCOM (90s)
  - Akka / Orleans (2010s)
  - Google's Service Weaver (2023)



From Ghemawat et al. "Towards Modern Development of Cloud Applications"

The problem: “all or nothing” solutions

Service Weaver 

🚨 Important Announcement

> **Service Weaver** began as an exploratory initiative to understand the challenges of developing, deploying, and maintaining distributed applications. We were excited by the strong interest from the developer community, which led us to open-source the project.
>
> We greatly appreciate the continued advocacy and support of the Service Weaver community. However, we realized that it was hard for users to adopt Service Weaver directly since it required rewriting large parts of existing applications. Therefore, Service Weaver did not see much direct use, and **effective December 5, 2024**, we will transition Service Weaver into maintenance mode. After this date, for the next 6 months, we will only push critical commits to the GitHub repository, respond to critical issues, merge critical pull requests, and patch new releases. We recommend that users fork the repository and report any issues preventing them from maintaining a stable version of the code.
>
> **On June 6, 2025**, we plan to permanently freeze and archive the GitHub repository, after which no new commits or releases will be made.

Service Weaver is a programming framework for writing, deploying, and managing distributed applications. You can run, test, and debug a Service Weaver application locally on your machine, and then deploy it to the cloud with a single command.

```
$ go run .                      # Run locally.
$ weaver gke deploy weaver.toml # Run in the cloud.
```

Enter Monolift

Vision: simplify distribution with a **“pay-as-you-go”** programming model

- Prioritize incremental adoption
- Minimize code changes

Key idea: treat distribution as a **compiler pass** for **general purpose** languages

- use the tools we already have at home!

The Key Idea

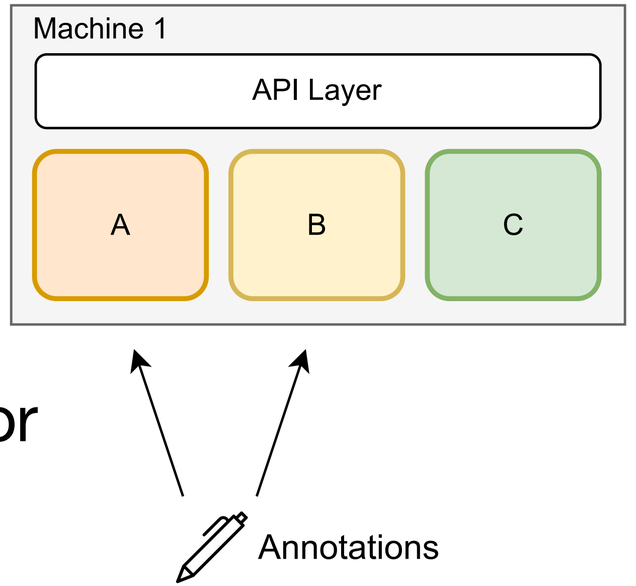
functions <~> serverless tasks

classes and interfaces <~> microservices

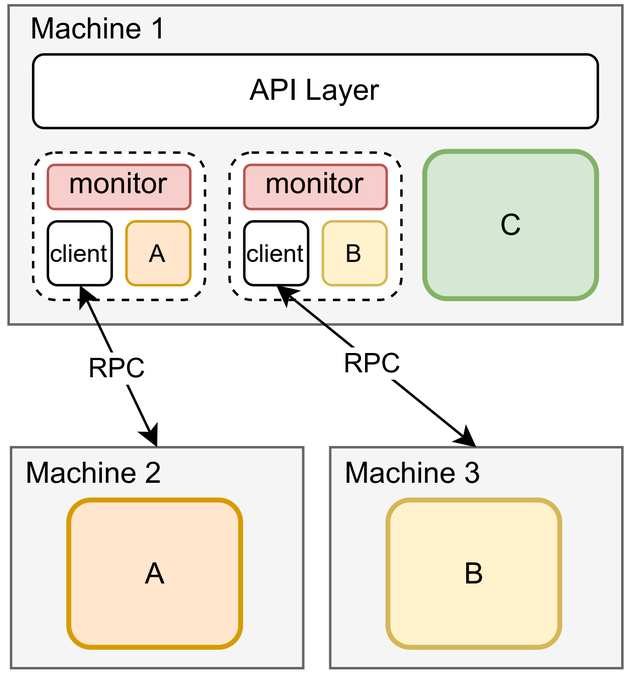
Monolift Design

Core abstraction is the **Lift**

- Code segment that can run locally or remotely
- Created by annotating source code
- Compiler extracts annotated code into standalone deployment artifacts
- Runtime monitor determines how to invoke lifts



Original Deployment



Lifted Deployment

Mental model: lifts *offload* computation onto additional resources as needed

Using Monolift

```
func hashPassword(pw string) (string, error) {
    // resource intensive hash algorithm
    // to guard against brute-force attacks
    key, err := scrypt.Key(pw, salt)
    return key, err
}

type usrMgr struct {
    db database.Client
}

func (u *usrMgr) Register(name, pw string) error {
    if pwHash, err := hashPassword(pw); err != nil {
        return err
    }
    return u.db.Insert("users", name, pwHash)
}
```

Using Monolift

- Add annotations as comments

```
//monolift:offload
func hashPassword(pw string) (string, error) {
    // resource intensive hash algorithm
    // to guard against brute-force attacks
    key, err := scrypt.Key(pw, salt)
    return key, err
}

type usrMgr struct {
    db database.Client
}

func (u *usrMgr) Register(name, pw string) error {
    if pwHash, err := hashPassword(pw); err != nil {
        return err
    }
    return u.db.Insert("users", name, pwHash)
}
```

Using Monolift

- Add annotations as comments
- Specify distribution policies via *delegate expressions*

```
//monolift:offload metric=CPU threshold=75%
func hashPassword(pw string) (string, error) {
    // resource intensive hash algorithm
    // to guard against brute-force attacks
    key, err := scrypt.Key(pw, salt)
    return key, err
}

type usrMgr struct {
    db database.Client
}

func (u *usrMgr) Register(name, pw string) error {
    if pwHash, err := hashPassword(pw); err != nil {
        return err
    }
    return u.db.Insert("users", name, pwHash)
}
```

Using Monolift

- Add annotations as comments
- Specify distribution policies via *delegate expressions*
- Compile and deploy
- Adjust and iterate if needed

```
//monolift:offload metric=CPU threshold=75%
func hashPassword(pw string) (string, error) {
    // resource intensive hash algorithm
    // to guard against brute-force attacks
    key, err := scrypt.Key(pw, salt)
    return key, err
}

type usrMgr struct {
    db database.Client
}

func (u *usrMgr) Register(name, pw string) error {
    if pwHash, err := hashPassword(pw); err != nil {
        return err
    }
    return u.db.Insert("users", name, pwHash)
}
```

```go
//monolift:offload metric=CPU threshold=75%
func hashPassword(pw string) (string, error) {
    // resource-intensive hash algorithm
    // to guard against brute-force attacks
    key, err := scrypt.Key(pw, salt)
    return key, err
}
```

<- Original

Compiled ->

```go
// Code generated by monolift. DO NOT EDIT.
func hashPasswordDelegate(pw string) (string, error) {
    shouldDelegate := monolift.EvalThreshold("CPU", 75)
    if shouldDelegate {
        params := map[string]string{"password": pw}
        return monolift.CreateTask("hashPassword", params)
    }
    return hashPassword(pw)
}
```

Monolift Prototype

- Implemented in Go, supports Go applications
- Treats Kubernetes as a compiler target
- Delegate expressions as signal/threshold policies
  - Supports CPU + memory consumption, invocation rates
- Runtime monitor embedded as background routine
  - polls cgroup interface for resource consumption

Challenges

- *How do you know what code to lift?*
  - Future work: profile guided optimization, visualization tools

Challenges

- *How do you know what code to extract?*
  - Future work: profile guided optimization, visualization tools
- *Retrofitting distribution vs. designing for it*
  - Monolift leverages modularity, can’t save your spaghetti code
  - Future work: automated refactors or compiler-aided advice

Challenges

- *How do you know what code to lift?*
  - Future work: profile guided optimization, visualization tools
- *Retrofitting distribution vs. designing for it*
  - Monolift leverages modularity, can’t save your spaghetti code
  - Future work: automated refactors or compiler-aided advice
- *How to reconcile individual delegate expressions into a global policy?*
  - Future work: formulate transition models, ML-based autoscaling

Conclusion

- Monolift automatically refactors monolithic applications into distributed architectures to unlock cloud scalability
- Supports existing code by handling distribution as a compiler pass instead of a new abstraction
- Refactors guided by a lightweight annotation language that supports dynamic distribution policies, managed by embedded runtime
- Enables rapid and low-commitment exploration of design space

https://github.com/tgoodwin/monolift