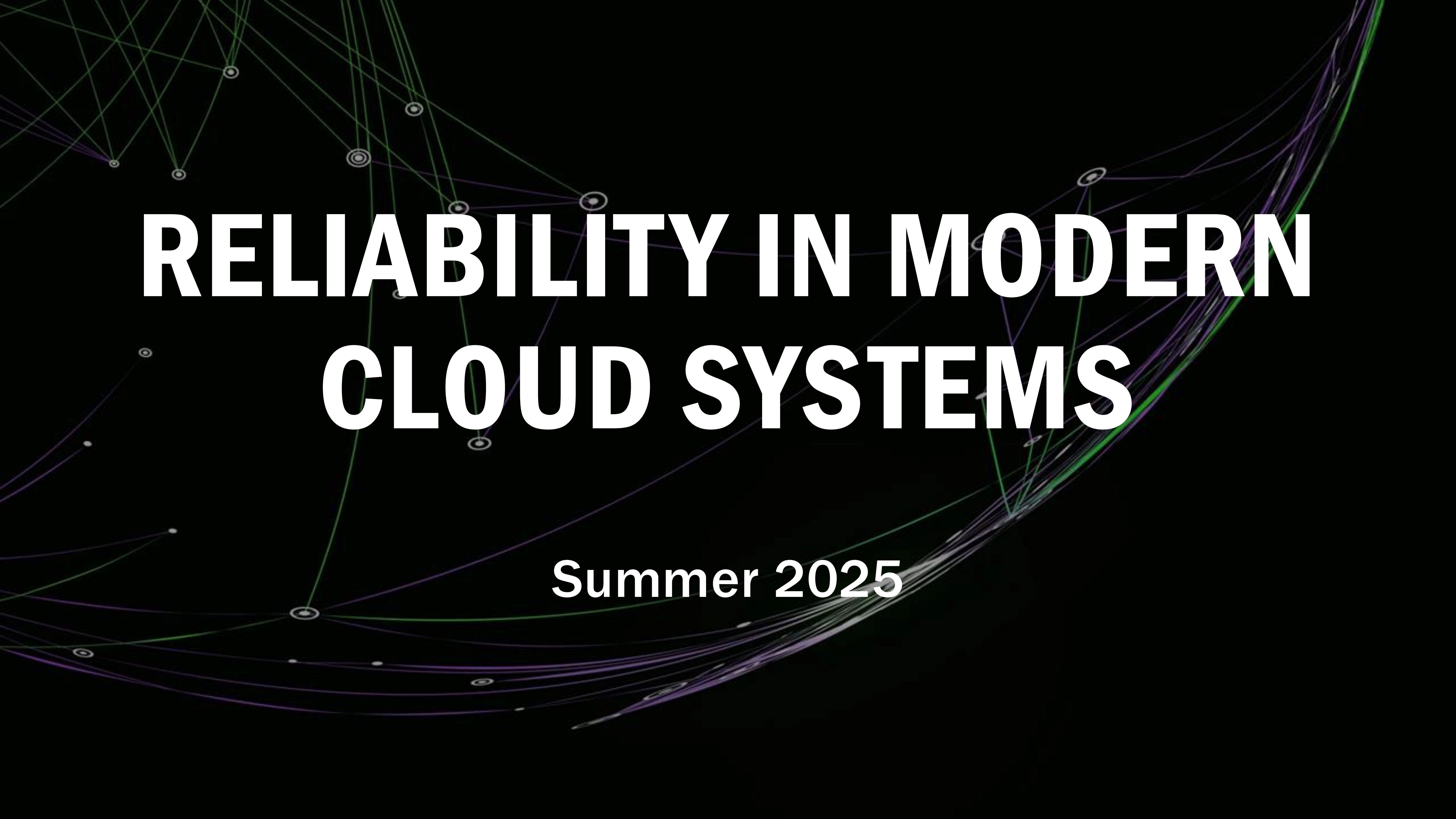


RELIABILITY IN MODERN CLOUD SYSTEMS

Summer 2025

LOGISTICS

ONLINE DISCUSSION FORUM

- Online discussion forum for the course is active

- Post questions/doubts about the assignments and course materials on the discussion forum

Link: https://os-discourse.saarland-informatics-campus.de/

ASSIGNMENT 1

- Assignment 1 will be released tonight
  - Due: Friday 10th May, 2025, 5pm CEST

- Goal of the assignment: Implement a luggage sharing microservice application
  - Implement business logic of the different services
  - Get all unit tests to pass
  - Build and run the application using Blueprint
  - Test the application with generated end-to-end tests

MONOLITHS VS MICROSERVICES DISCUSSION

DISCUSSION THEMES

❖ When to use Microservices vs Monoliths?

❖ What is the right granularity for a microservice?

❖ What are the key components of a representative microservice system?

❖ Are microservices more reliable than monoliths?

DISCUSSION THEMES

- What are the key components of a representative microservice system?

DISCUSSION THEMES

- What are the key components of a representative microservice system?

Key Takeaway: There is no 1 representative microservice system, just points in a design space

DISCUSSION THEMES

- What is the right granularity for a microservice?

DISCUSSION THEMES

- ❖ What is the right granularity for a microservice?

Key Takeaway: No fixed right answer. Varies across applications and use-cases

DISCUSSION THEMES

- ❖ When to use Microservices vs Monoliths?

DISCUSSION THEMES

❖ When to use Microservices vs Monoliths?

Monoliths have better performance + lower cost (performance benefit)

Microservices are flexible and scalable (operational benefit)

Key Takeaway: There is a fundamental tradeoff between microservices and monoliths

DISCUSSION THEMES

- ❖ Are microservices more reliable than monoliths?

DISCUSSION THEMES

- Are microservices more reliable than monoliths?

Key Takeaway: Yes, microservices offer better overall reliability at the cost of performance

THE TAIL AT SCALE

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

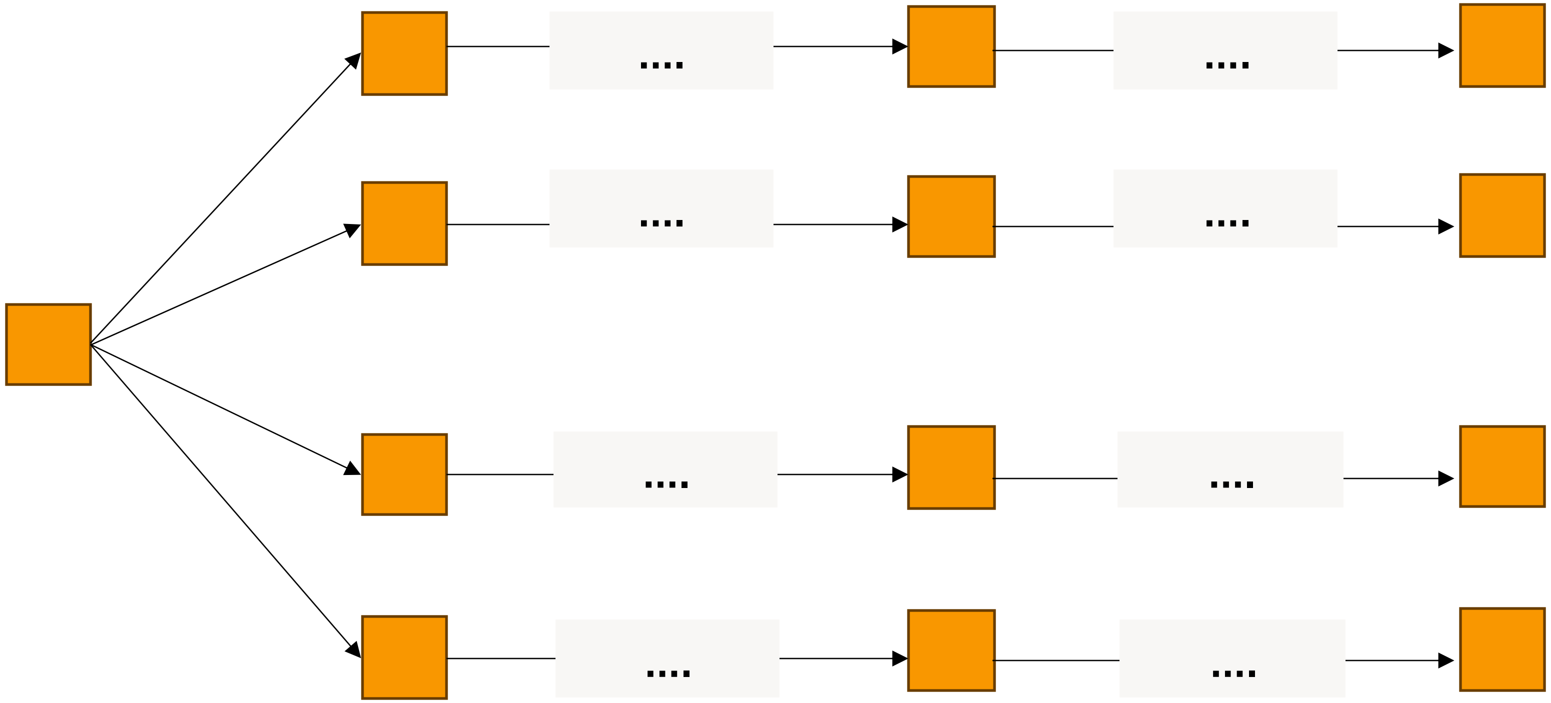
Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

THE TAIL AT SCALE

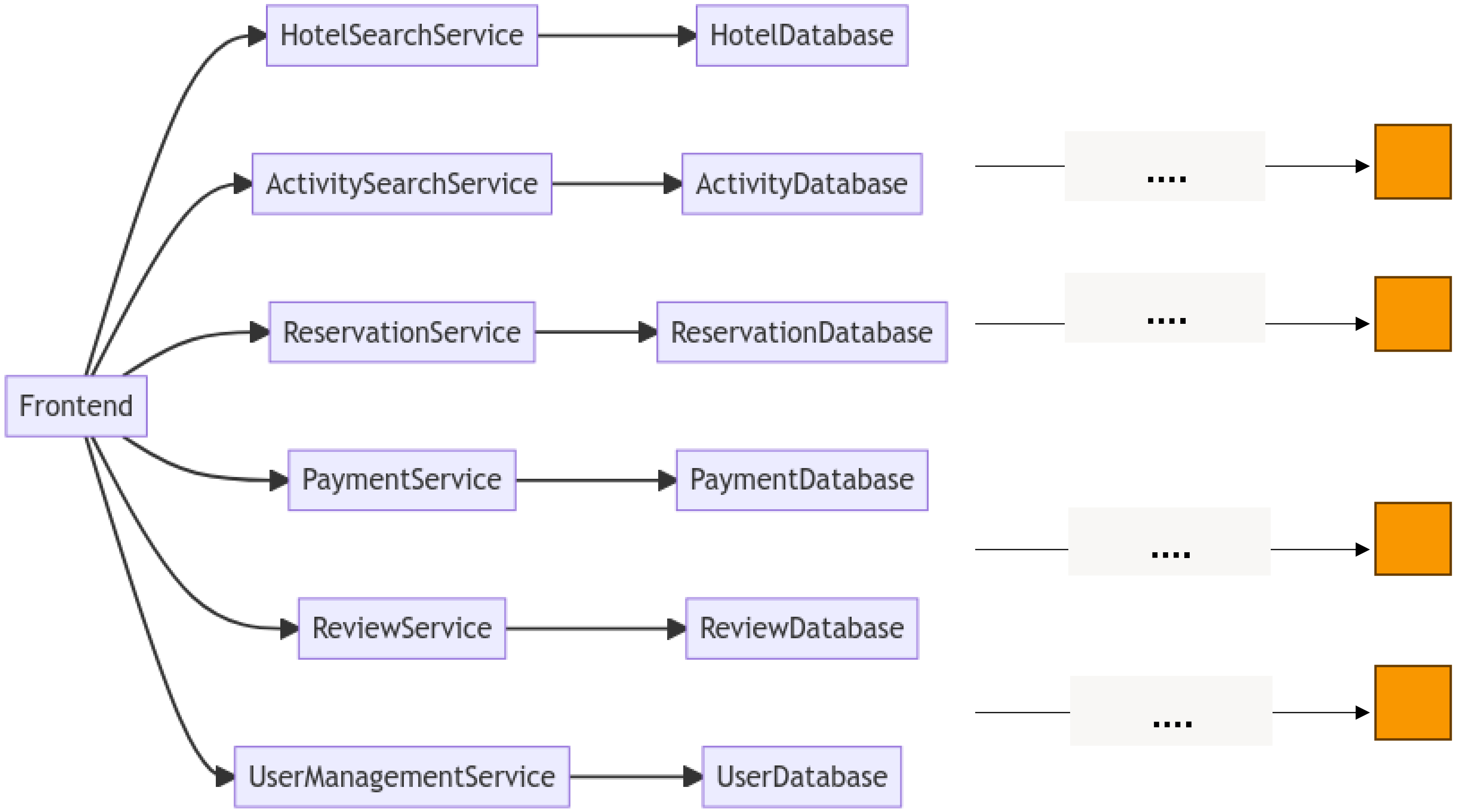
Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

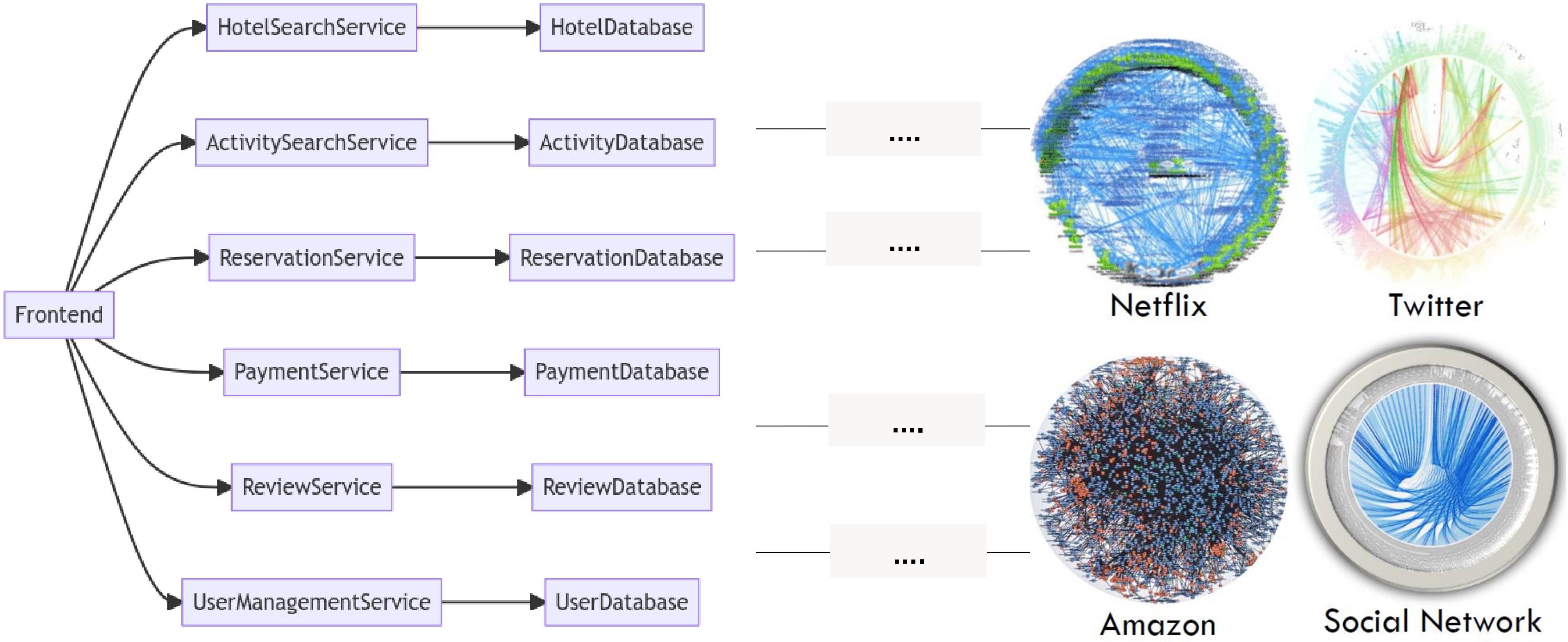
CLOUD SYSTEMS AT SCALE



CLOUD SYSTEMS AT SCALE



REAL DEPENDENCY GRAPHS



Frontend

HotelSearchService → HotelDatabase

ActivitySearchService → ActivityDatabase

ReservationService → ReservationDatabase

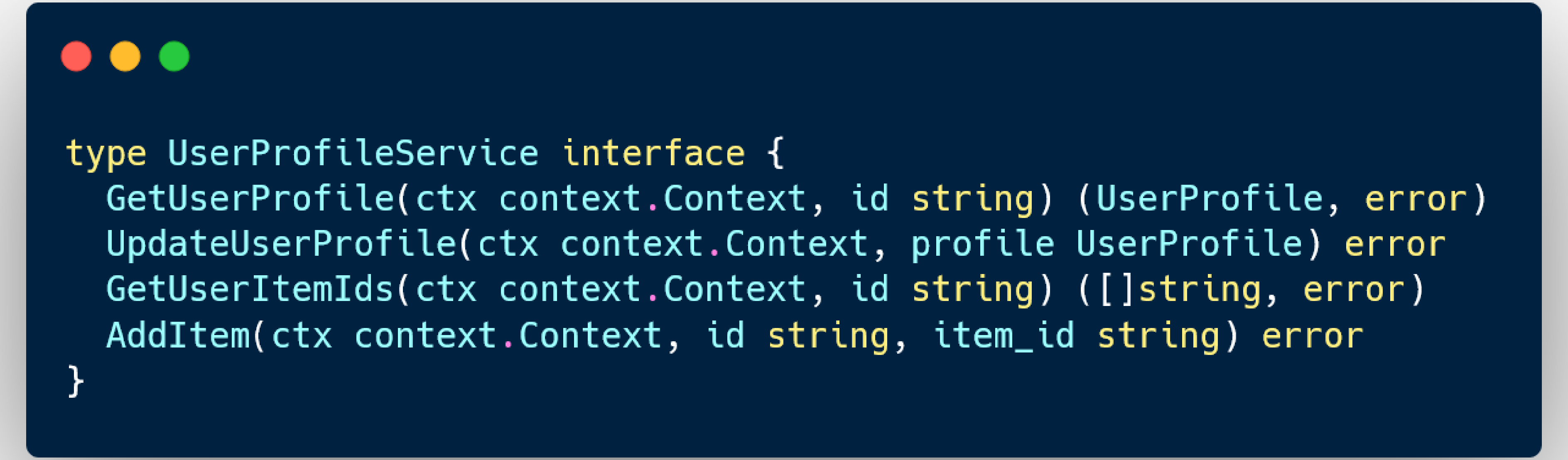
PaymentService → PaymentDatabase

ReviewService → ReviewDatabase

UserManagementService → UserDatabase

EACH SERVICE EXPORTS AN INTERFACE

EACH SERVICE EXPORTS AN INTERFACE



```go
type UserProfileService interface {
  GetUserProfile(ctx context.Context, id string) (UserProfile, error)
  UpdateUserProfile(ctx context.Context, profile UserProfile) error
  GetUserItemIds(ctx context.Context, id string) ([]string, error)
  AddItem(ctx context.Context, id string, item_id string) error
}
```

COMMUNICATION PATTERNS

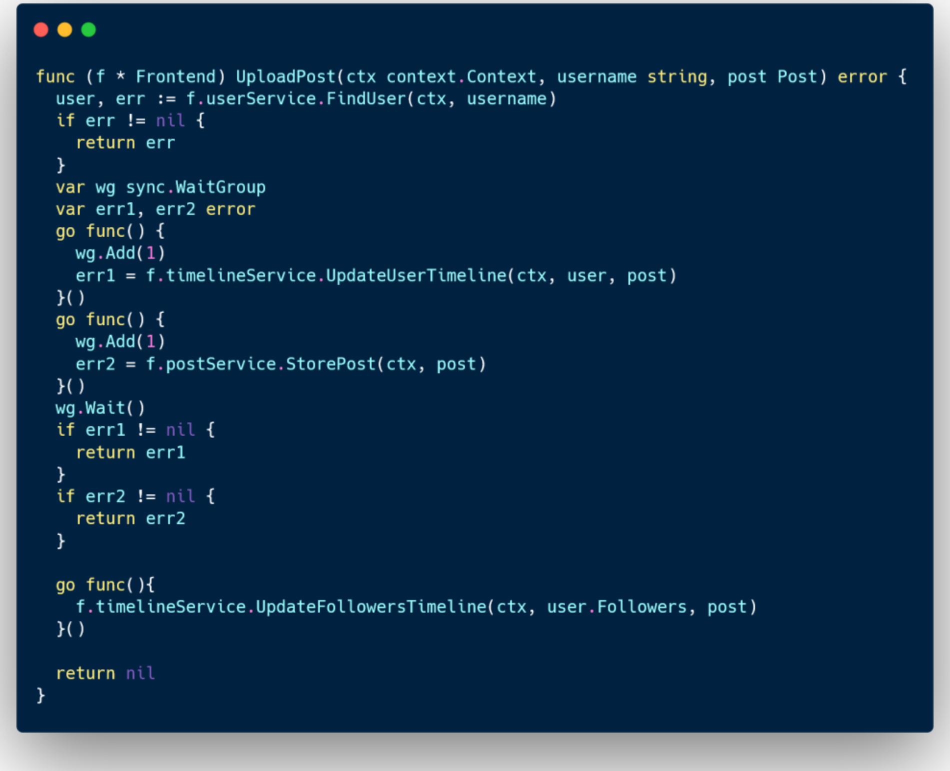
3 key/common communication patterns

- Blocked Waiting
- Non-Blocked Waiting
- Non-Blocked No-Waiting

COMMUNICATION PATTERNS

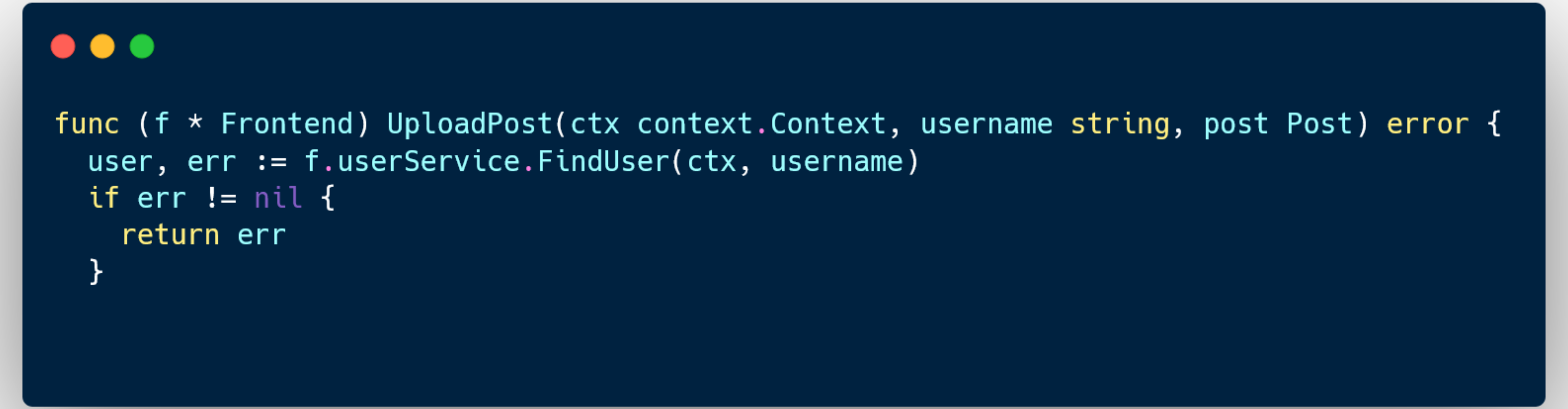
3 key/common communication patterns

- ❖ Blocked Waiting
- ❖ Non-Blocked Waiting
- ❖ Non-Blocked No-Waiting



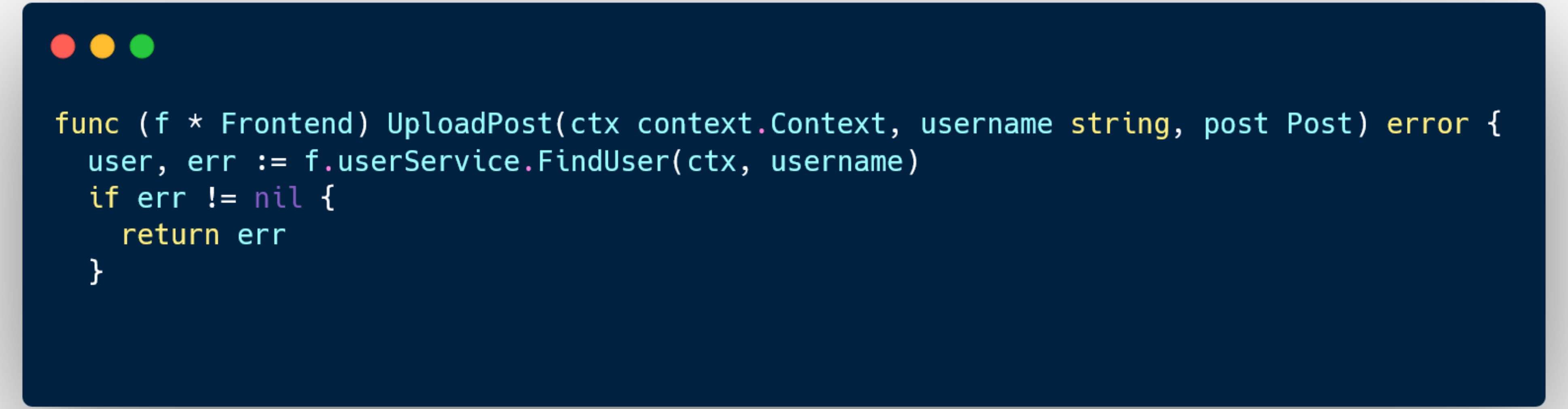
```go
func (f * Frontend) UploadPost(ctx context.Context, username string, post Post) error {
  user, err := f.userService.FindUser(ctx, username)
  if err != nil {
    return err
  }
  var wg sync.WaitGroup
  var err1, err2 error
  go func() {
    wg.Add(1)
    err1 = f.timelineService.UpdateUserTimeline(ctx, user, post)
  }()
  go func() {
    wg.Add(1)
    err2 = f.postService.StorePost(ctx, post)
  }()
  wg.Wait()
  if err1 != nil {
    return err1
  }
  if err2 != nil {
    return err2
  }

  go func(){
    f.timelineService.UpdateFollowersTimeline(ctx, user.Followers, post)
  }()

  return nil
}
```

BLOCKED WAITING



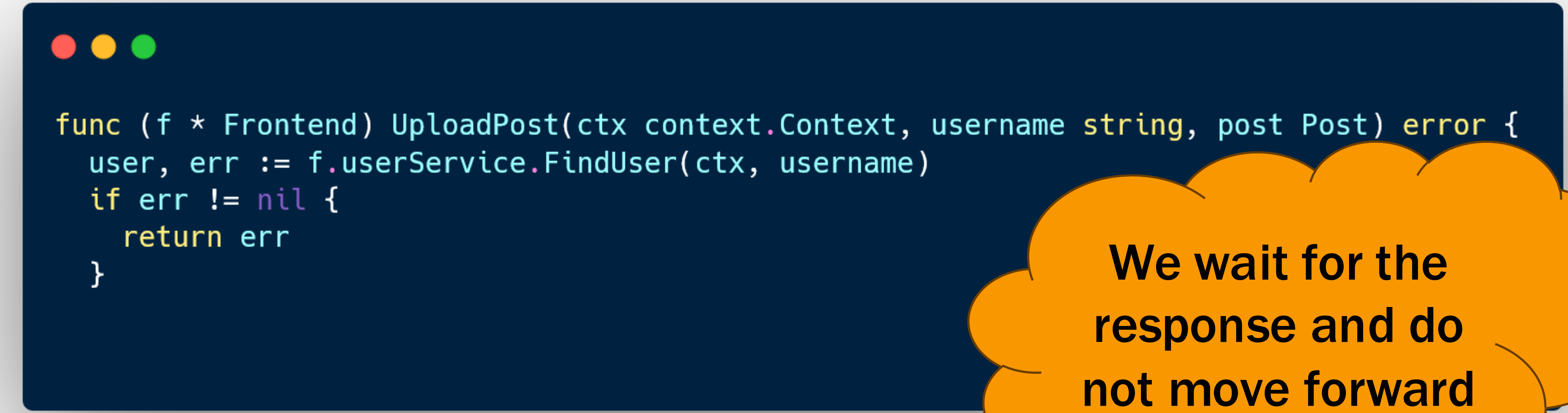
```
func (f * Frontend) UploadPost(ctx context.Context, username string, post Post) error {
  user, err := f.userService.FindUser(ctx, username)
  if err != nil {
    return err
  }
```

BLOCKED WAITING



```go
func (f * Frontend) UploadPost(ctx context.Context, username string, post Post) error {
  user, err := f.userService.FindUser(ctx, username)
  if err != nil {
    return err
  }
```

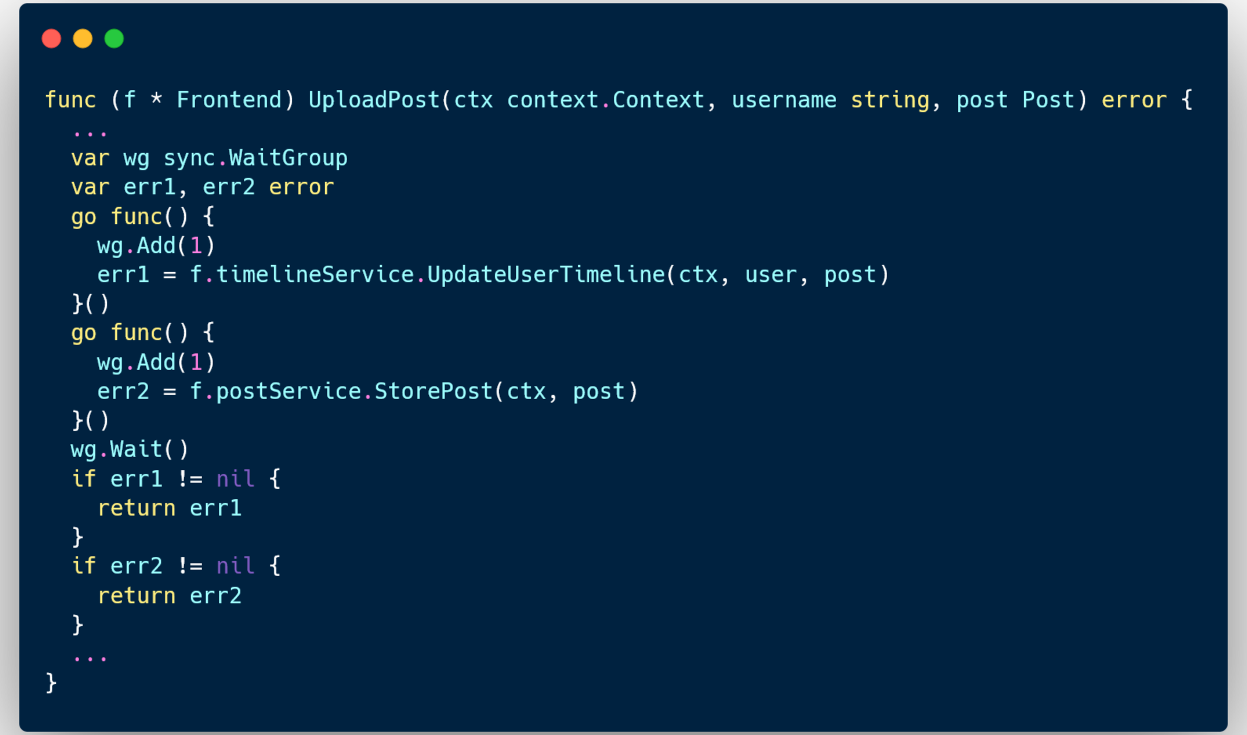
BLOCKED WAITING



```go
func (f * Frontend) UploadPost(ctx context.Context, username string, post Post) error {
  user, err := f.userService.FindUser(ctx, username)
  if err != nil {
    return err
  }
```

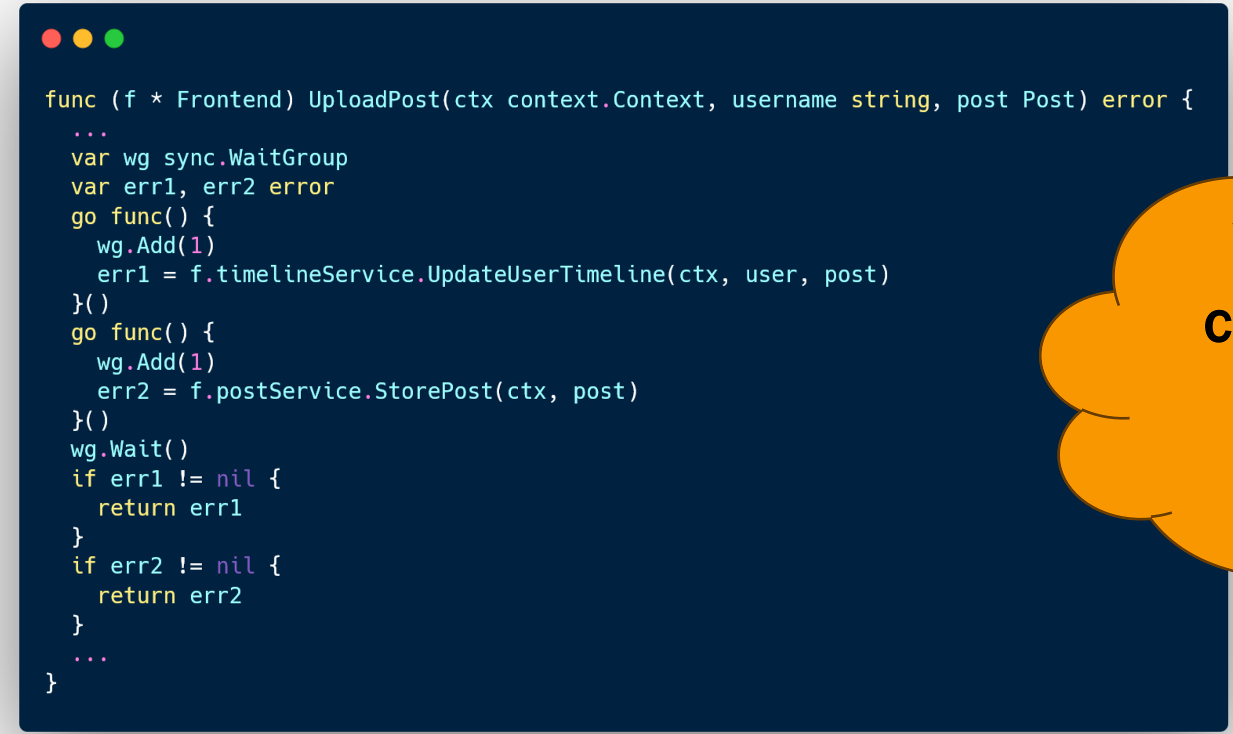
We wait for the response and do not move forward

NON-BLOCKED WAITING



```
func (f * Frontend) UploadPost(ctx context.Context, username string, post Post) error {
  ...
  var wg sync.WaitGroup
  var err1, err2 error
  go func() {
    wg.Add(1)
    err1 = f.timelineService.UpdateUserTimeline(ctx, user, post)
  }()
  go func() {
    wg.Add(1)
    err2 = f.postService.StorePost(ctx, post)
  }()
  wg.Wait()
  if err1 != nil {
    return err1
  }
  if err2 != nil {
    return err2
  }
  ...
}
```

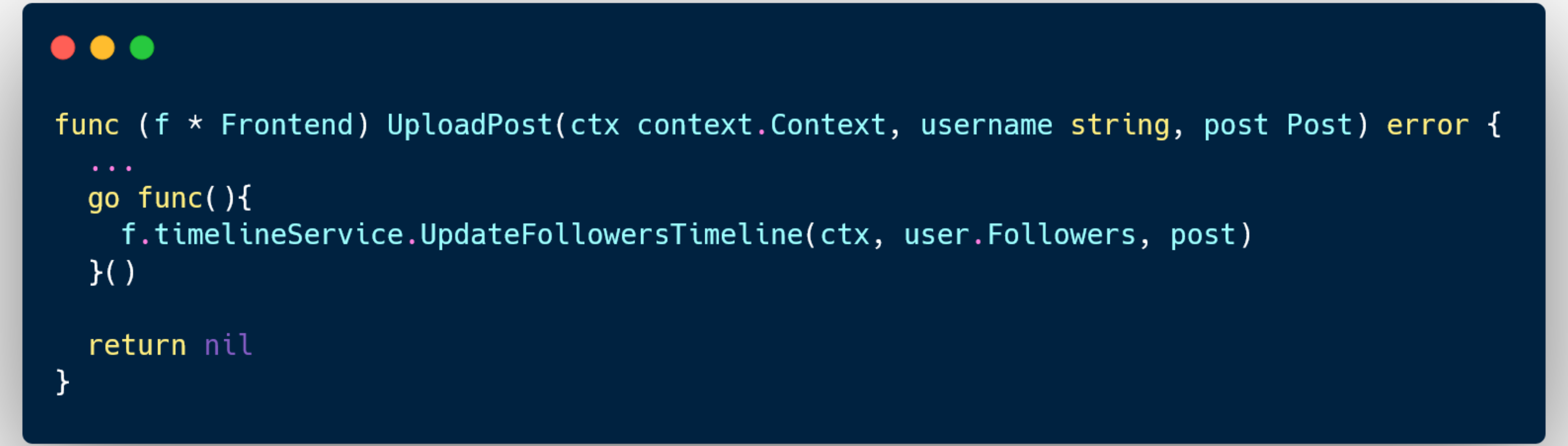
NON-BLOCKED WAITING



```
func (f * Frontend) UploadPost(ctx context.Context, username string, post Post) error {
  ...
  var wg sync.WaitGroup
  var err1, err2 error
  go func() {
    wg.Add(1)
    err1 = f.timelineService.UpdateUserTimeline(ctx, user, post)
  }()
  go func() {
    wg.Add(1)
    err2 = f.postService.StorePost(ctx, post)
  }()
  wg.Wait()
  if err1 != nil {
    return err1
  }
  if err2 != nil {
    return err2
  }
  ...
}
```

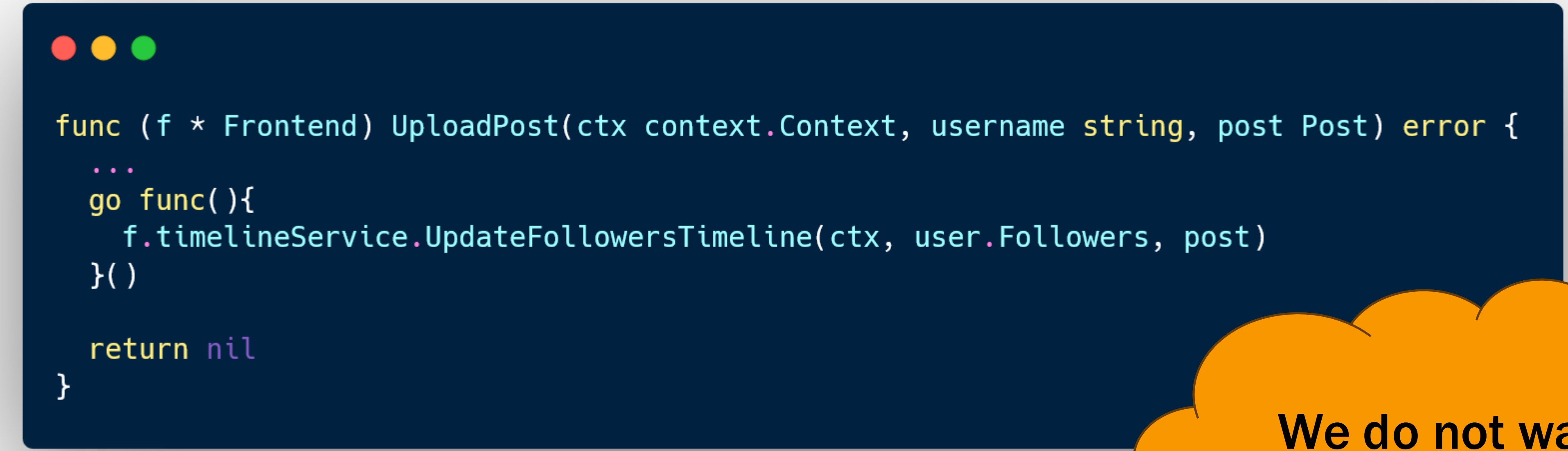
We can make concurrent calls but its not blocking

NON-BLOCKED NO WAITING



```go
func (f * Frontend) UploadPost(ctx context.Context, username string, post Post) error {
  ...
  go func(){
    f.timelineService.UpdateFollowersTimeline(ctx, user.Followers, post)
  }()

  return nil
}
```

NON-BLOCKED NO WAITING



```go
func (f * Frontend) UploadPost(ctx context.Context, username string, post Post) error {
  ...
  go func(){
    f.timelineService.UpdateFollowersTimeline(ctx, user.Followers, post)
  }()

  return nil
}
```

We do not wait
for any responses

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

PERFORMANCE

PERFORMANCE

Throughput

Total amount of work done

Typically measured in requests processed per unit of time

Latency

Total time taken by a request from start to finish

Typically measured end-to-end

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

PERFORMANCE

Throughput

Total amount of work done

Typically measured in requests processed per unit of time

Latency

Total time taken by a request from start to finish

Typically measured end-to-end

RESPONSIVE PERFORMANCE

Throughput

Total amount of work done

Typically measured in requests processed per unit of time

Latency

Total time taken by a request from start to finish

Typically measured end-to-end

RESPONSIVE PERFORMANCE

Throughput

Total amount of work done

Typically measured in requests processed per unit of time

Latency

Total time taken by a request from start to finish

Typically measured end-to-end

More Important!

HOW TO MEASURE RESPONSIVENESS?

Throughput

Total amount of work done

Typically measured in requests processed per unit of time

Mean Latency

Avg. Total time taken by a request from start to finish

Typically measured end-to-end

More Important!

IS MEAN A GOOD MEASUREMENT?

IS MEAN A GOOD MEASUREMENT?

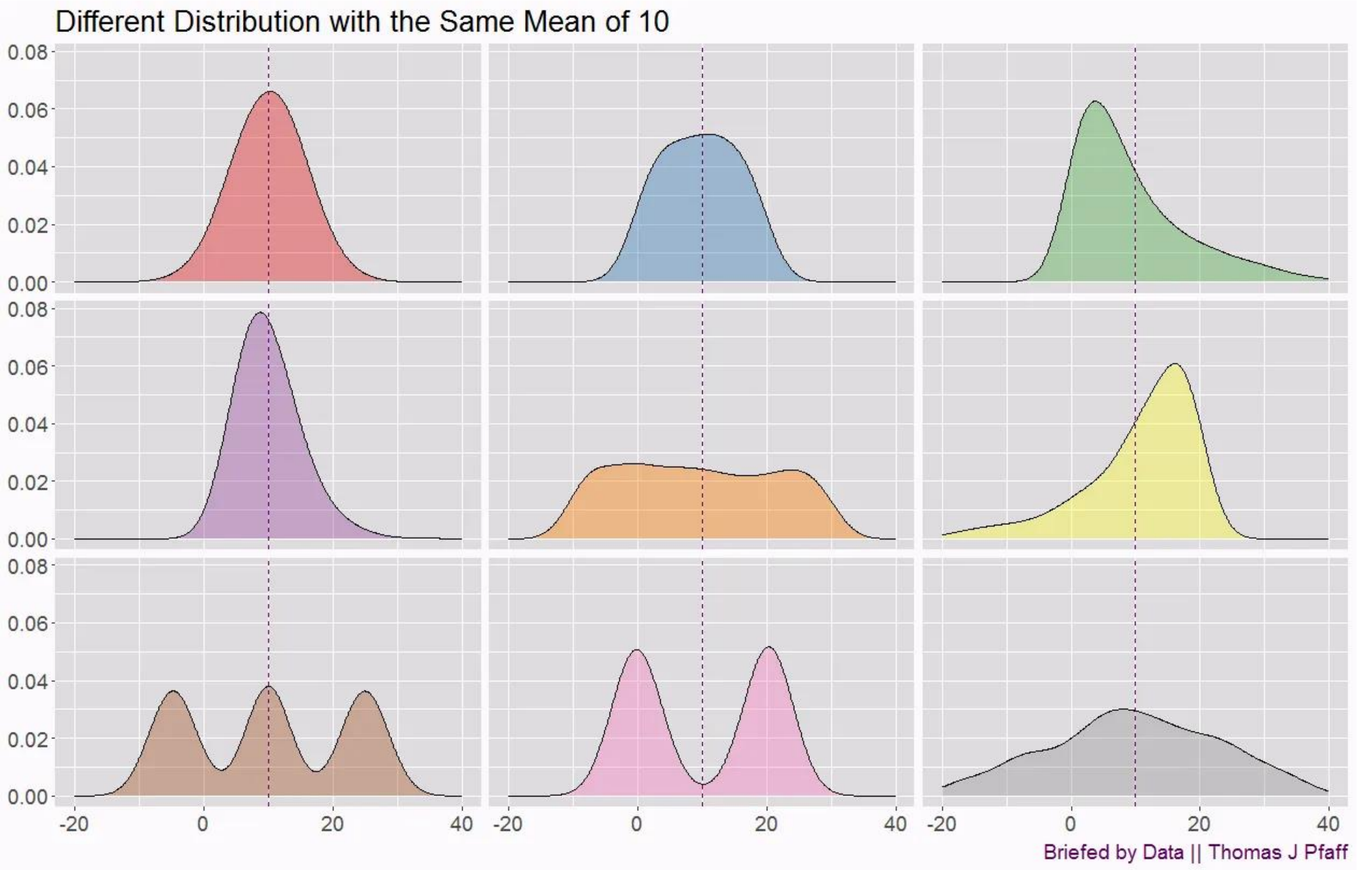
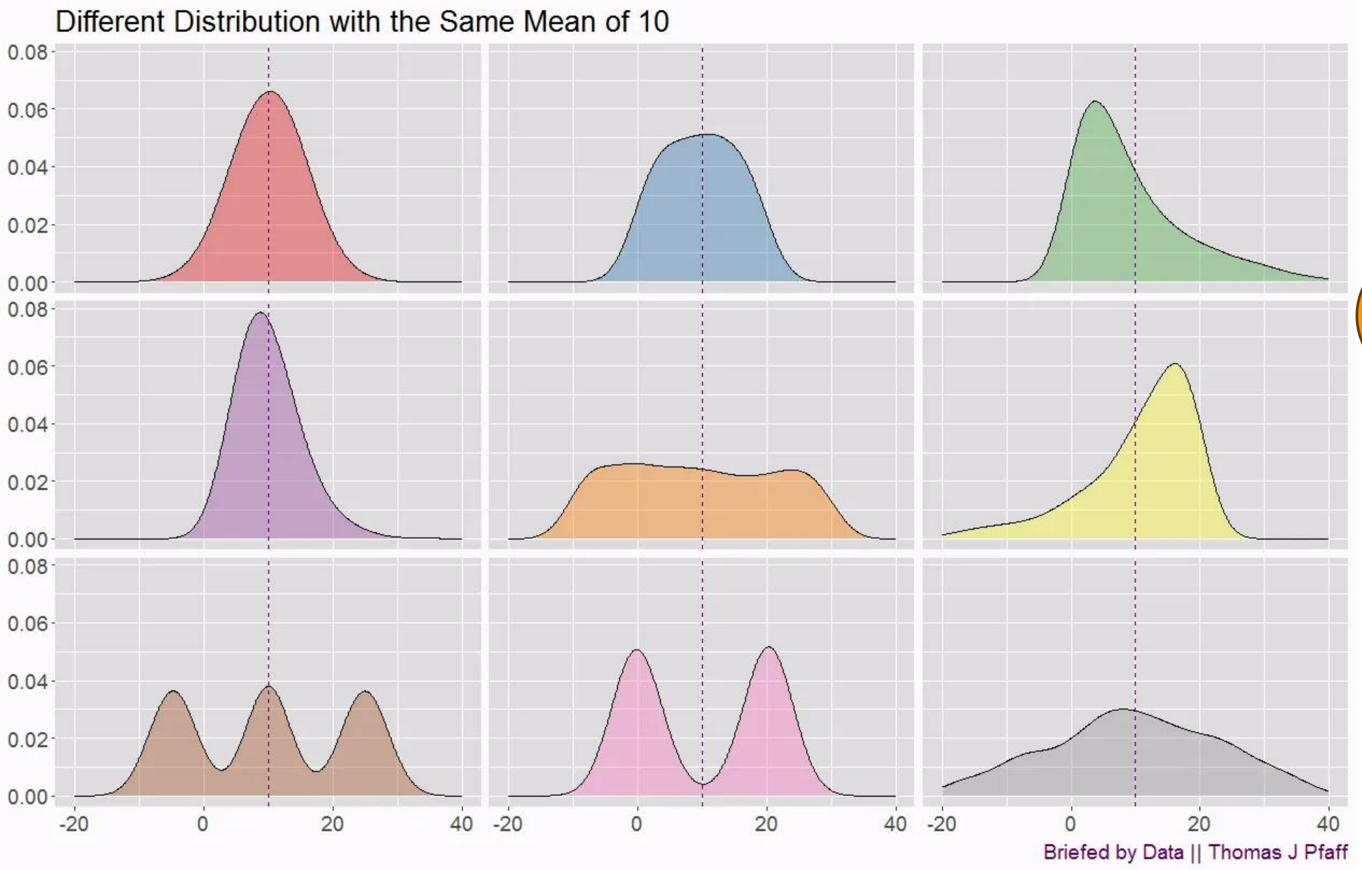


Image from https://briefedbydata.substack.com/p/same-mean-different-distribution

IS MEAN A GOOD MEASUREMENT?



Mean is not robust or indicative of the distribution

Image from https://briefedbydata.substack.com/p/same-mean-different-distribution

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

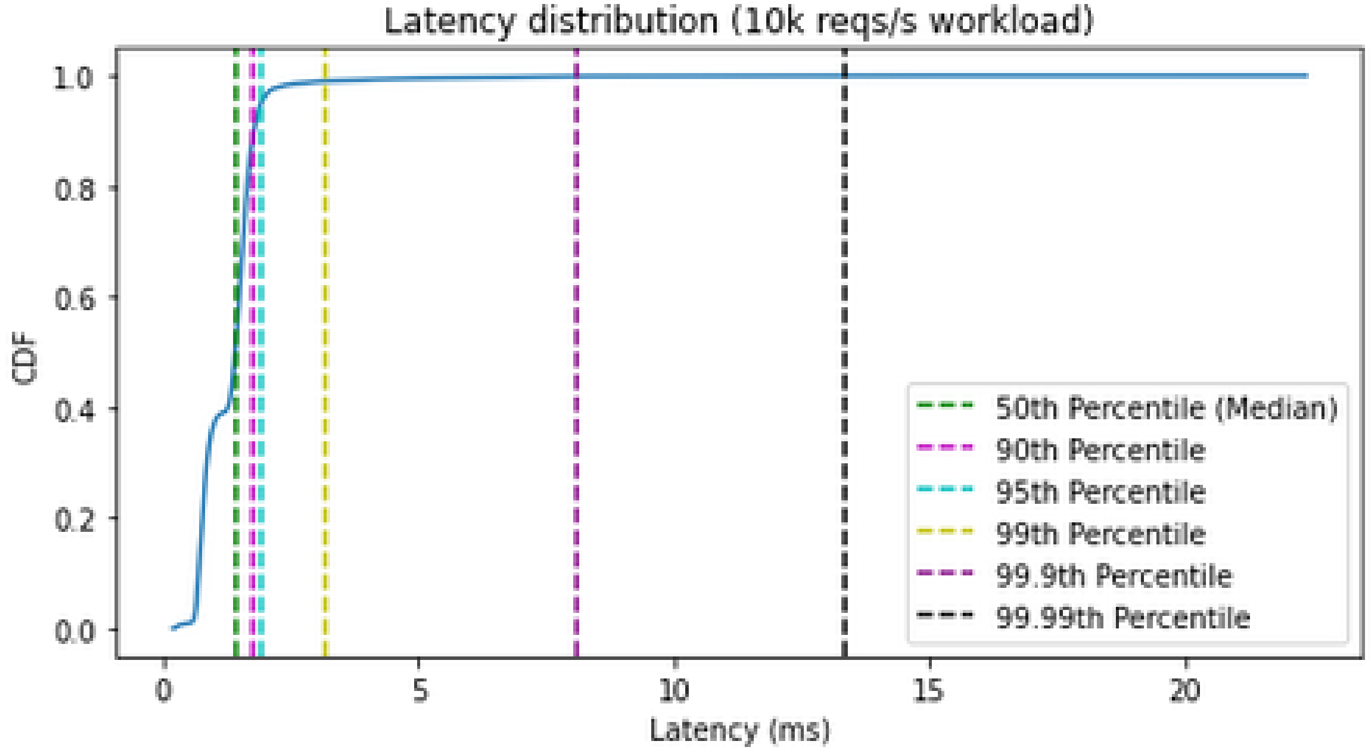
THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

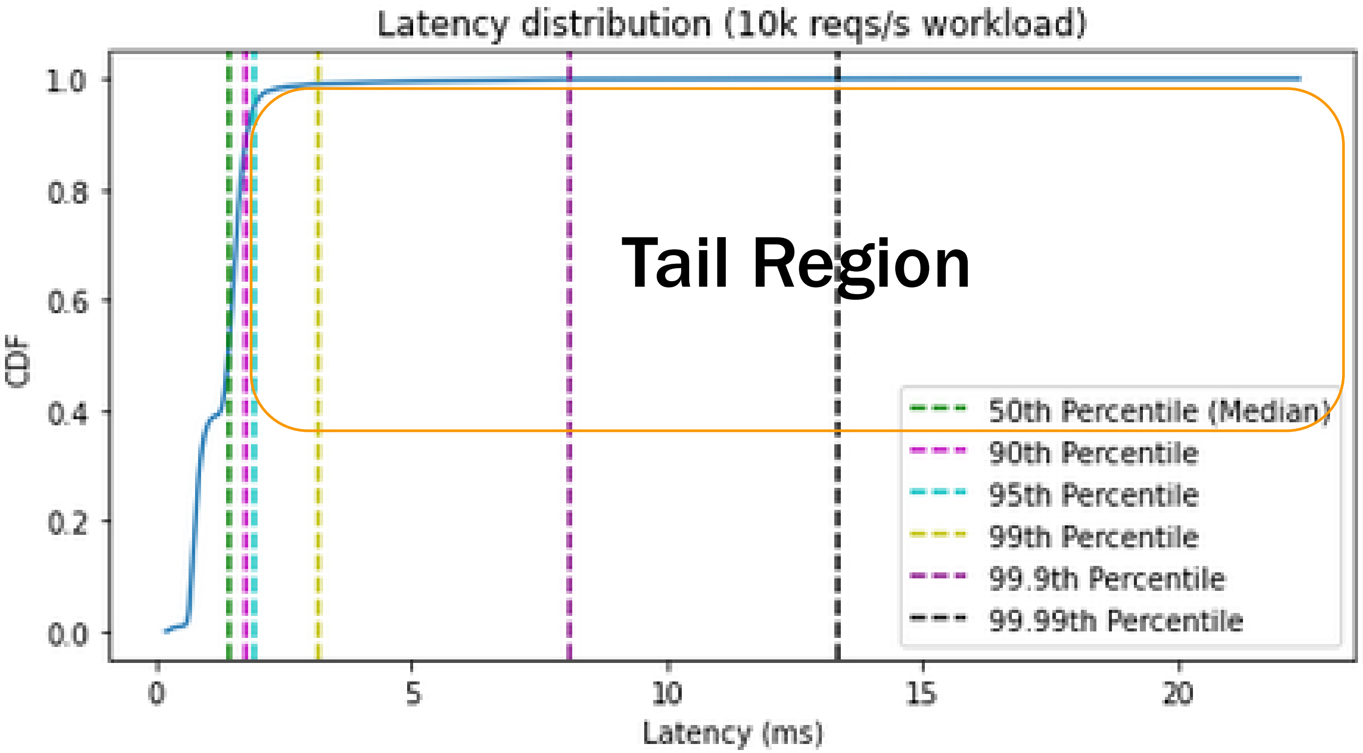
Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

TAIL PERFORMANCE

TAIL FOCUSES ON THE SLOWEST REQUESTS IN THE DISTRIBUTION



TAIL FOCUSES ON THE SLOWEST REQUESTS IN THE DISTRIBUTION



TAIL PERFORMANCE AT SCALE!

Example service characteristic:

Mean Latency: 10ms, 99th Percentile Latency: 1s

TAIL PERFORMANCE AT SCALE!

Example service characteristic:

Mean Latency: 10ms, 99th Percentile Latency: 1s

With 1 service, Prob(request latency >= 1s) =

TAIL PERFORMANCE AT SCALE!

Example service characteristic:

Mean Latency: 10ms, 99th Percentile Latency: 1s

With 1 service, Prob(request latency >= 1s) = 0.01

TAIL PERFORMANCE AT SCALE!

Example service characteristic:

Mean Latency: 10ms, 99th Percentile Latency: 1s

With 1 service, Prob(request latency >= 1s) = 0.01

With 100 services, Prob(request latency >= 1s) =

TAIL PERFORMANCE AT SCALE!

Example service characteristic:

Mean Latency: 10ms, 99th Percentile Latency: 1s

With 1 service, Prob(request latency >= 1s) = 0.01

With 100 services, Prob(request latency >= 1s) = 0.63

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

PERFORMANCE VARIABILITY AT SCALE

PERFORMANCE VARIABILITY AT SCALE

Causes of Variability

- Queuing at different layers
- Resource Sharing
  - Local sharing: co-located tasks
  - Global sharing: common dependencies
- Background tasks
- Energy + Power Management

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

NEXT WEEK

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

NEXT WEEK

DISCUSSION THEMES

- What software techniques do we need to handle performance variability?

- Why can we not eliminate all performance variability?

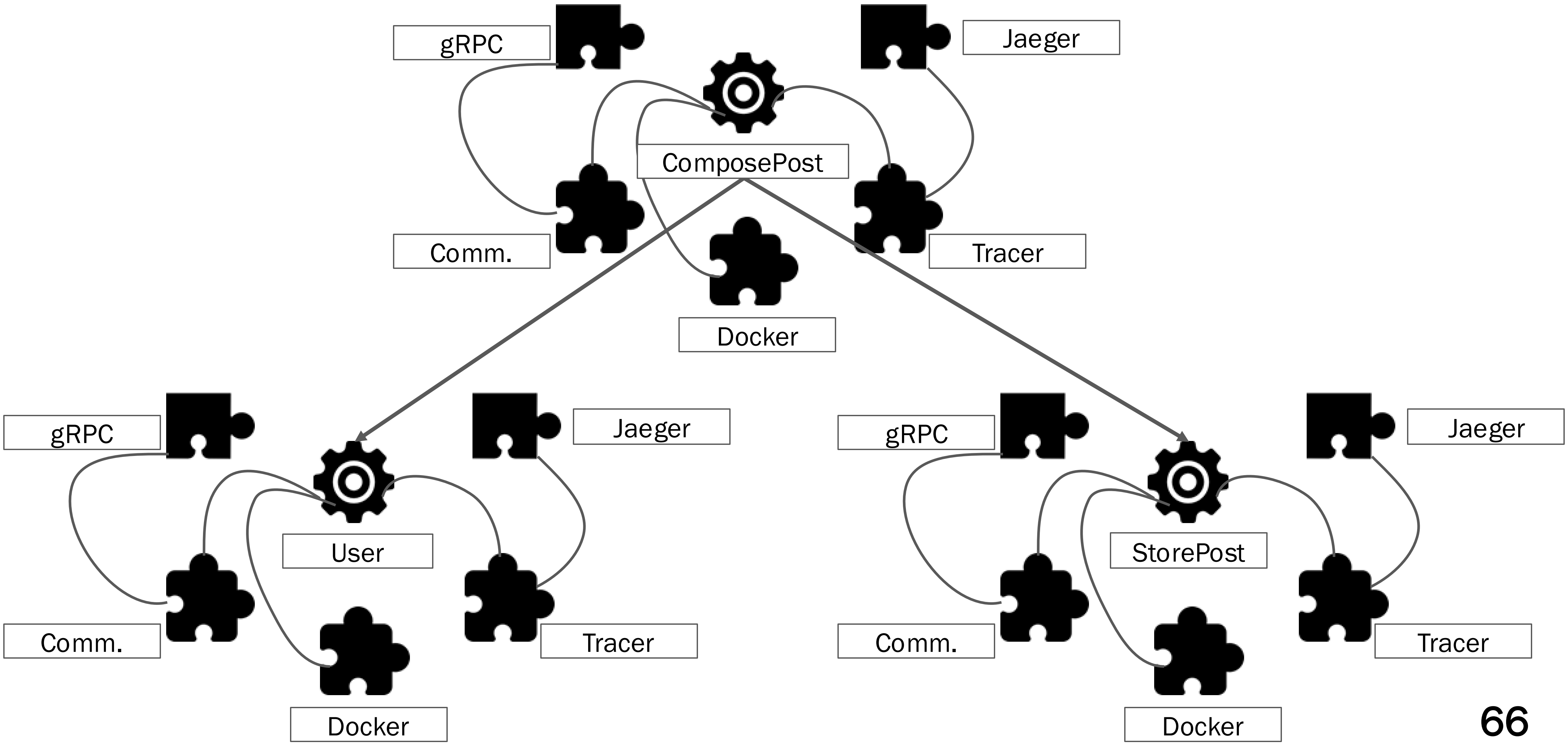
- What is the impact of new workloads and hardware on reliability?

ASSIGNMENT PRIMER BLUEPRINT 101

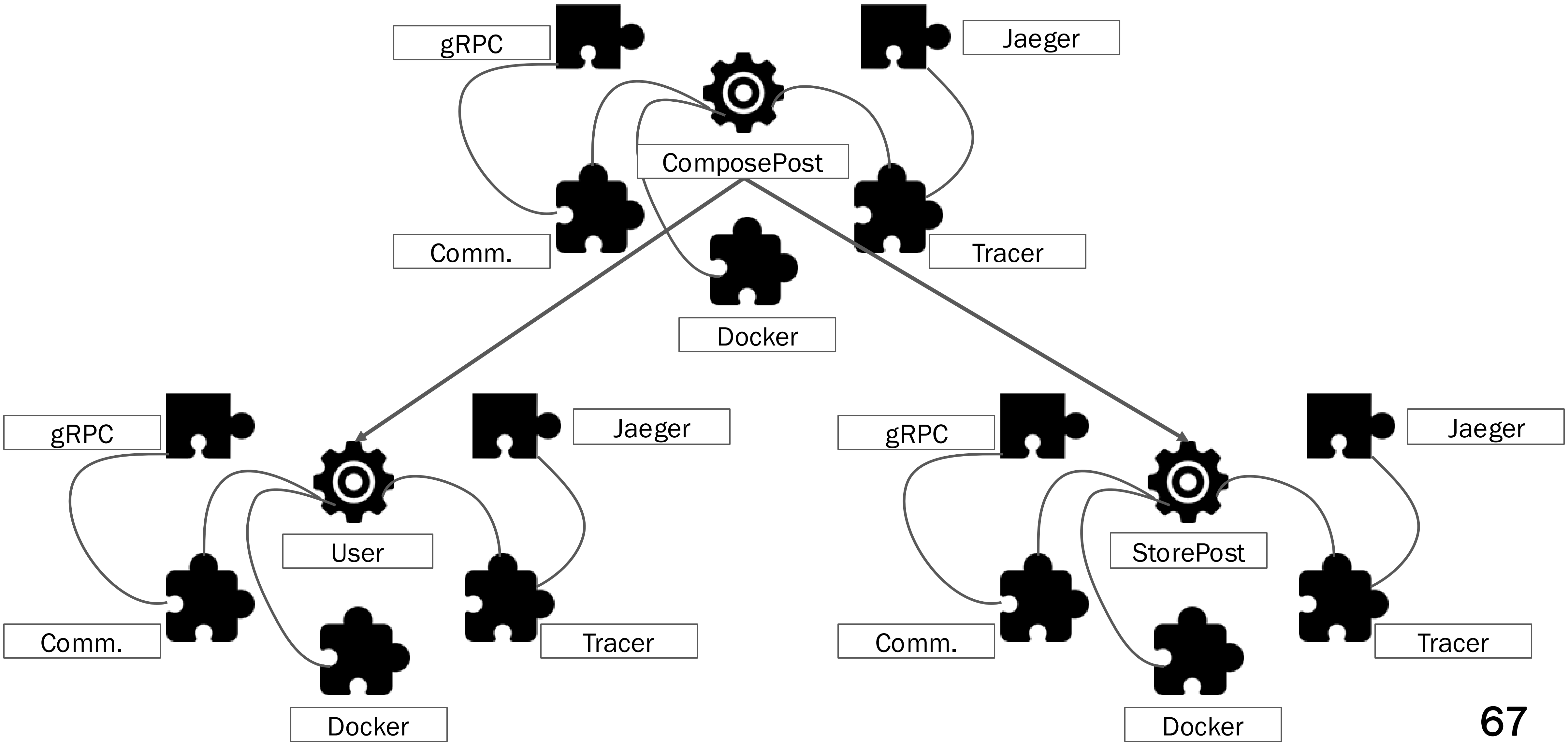
ASSIGNMENT 1

- Goal of the assignment: Implement a luggage sharing microservice application
  - Implement business logic of the different services
  - Get all unit tests to pass
  - Build and run the application using Blueprint
  - Test the application with generated end-to-end tests

A Typical Microservice

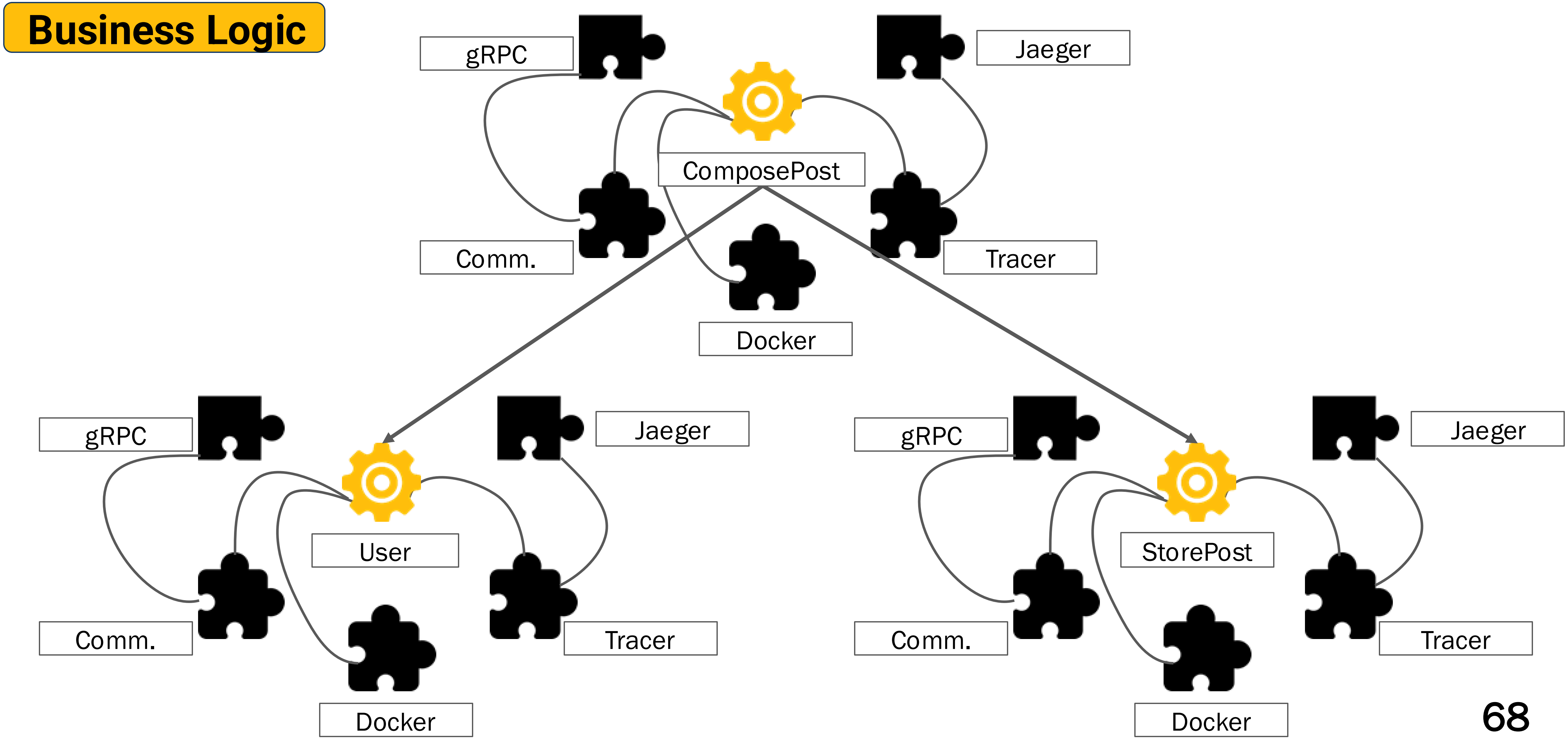


Key Idea: Decouple Application into 3 pieces



Key Idea: Decouple Application into 3 pieces

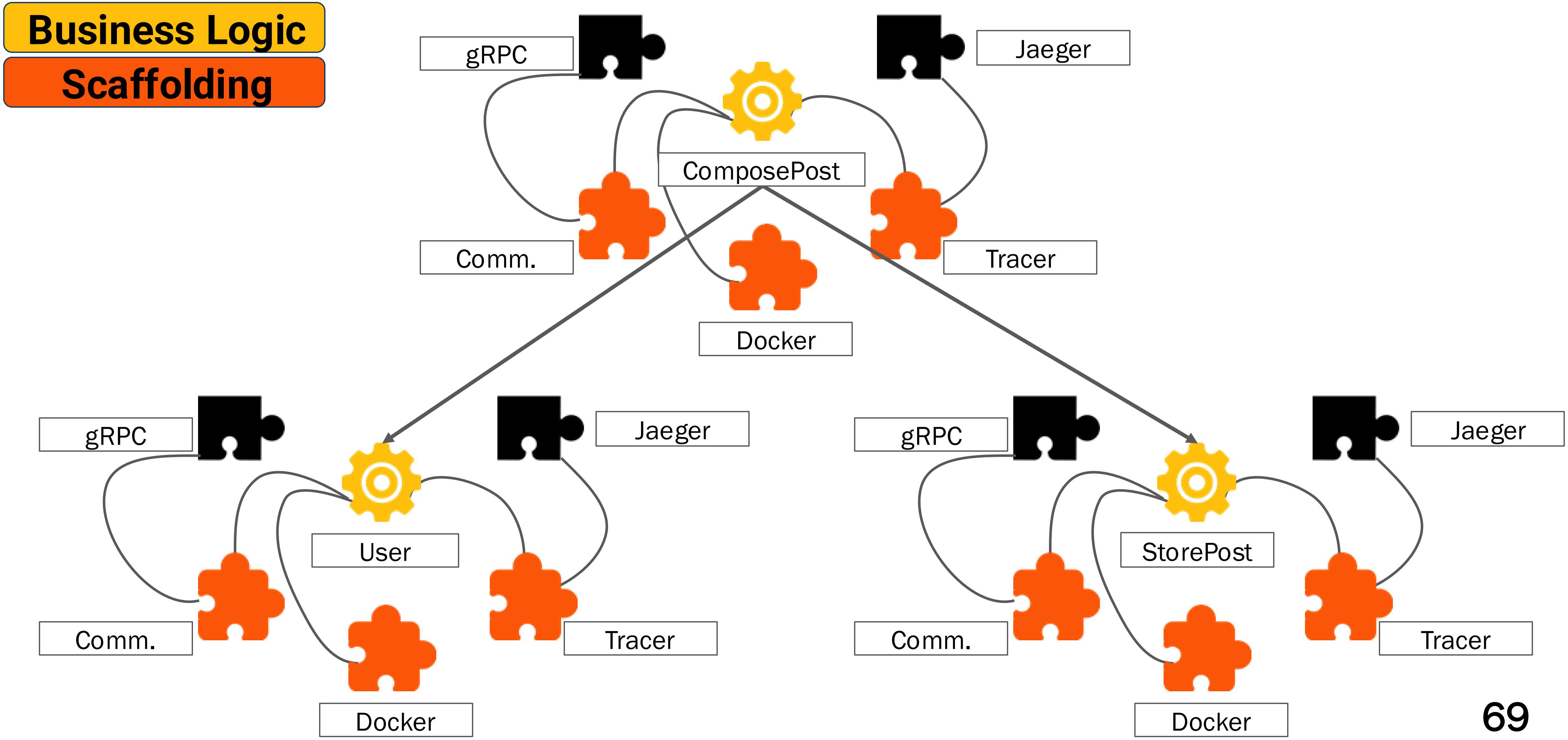
Business Logic



Key Idea: Decouple Application into 3 pieces

Business Logic

Scaffolding

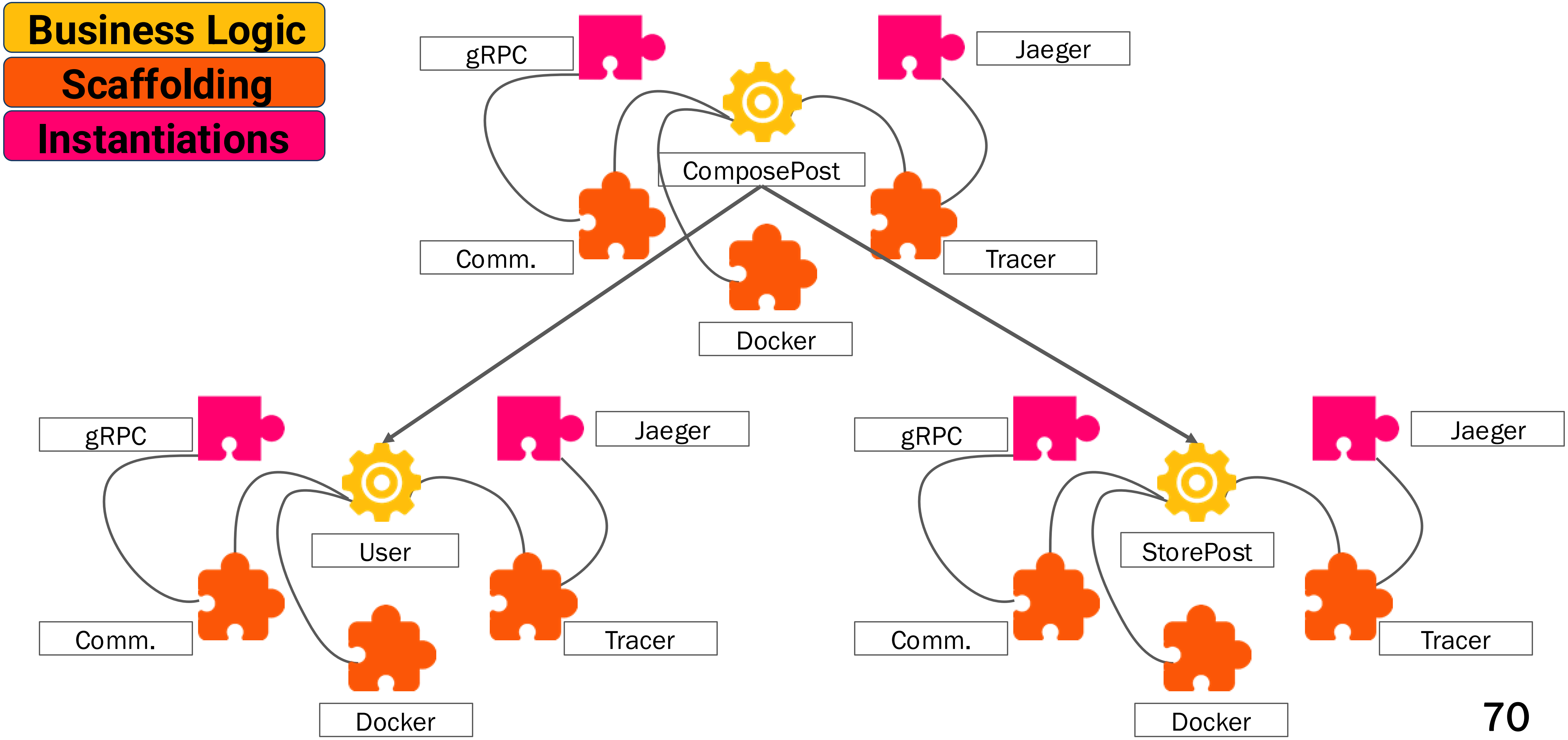


Key Idea: Decouple Application into 3 pieces

Business Logic

Scaffolding

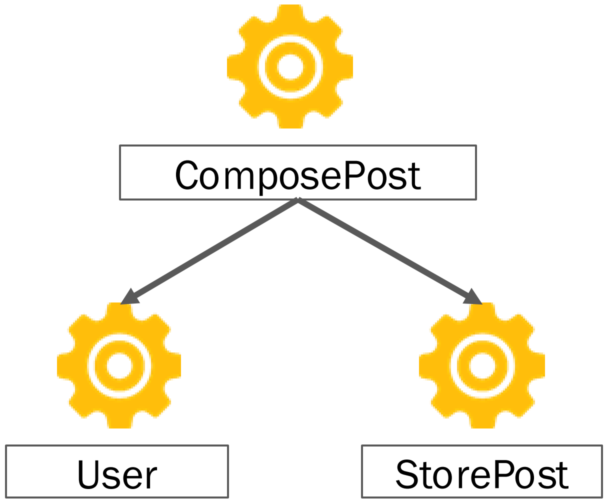
Instantiations



Separate 3 pieces into 2 input specs

Workflow Spec

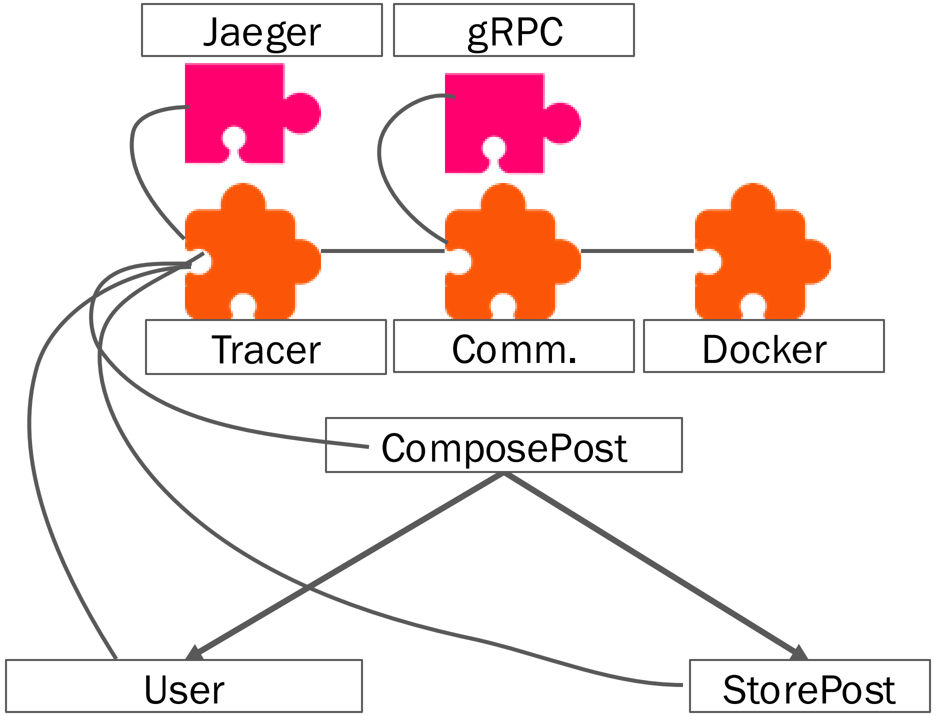
Business Logic



Wiring Spec

Scaffolding

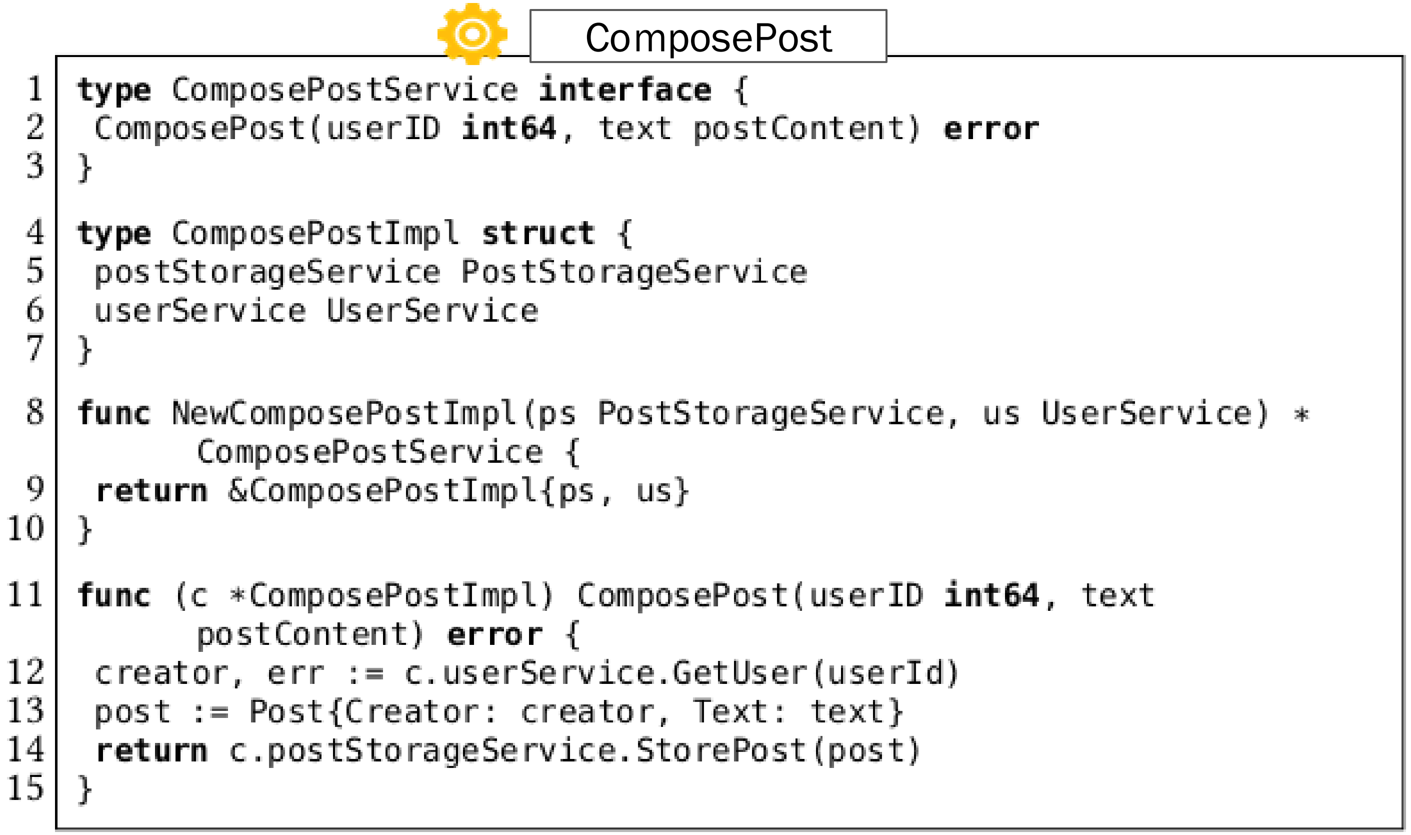
Instantiations



Workflow spec at a glance Business Logic

Workflow spec at a glance

Business Logic



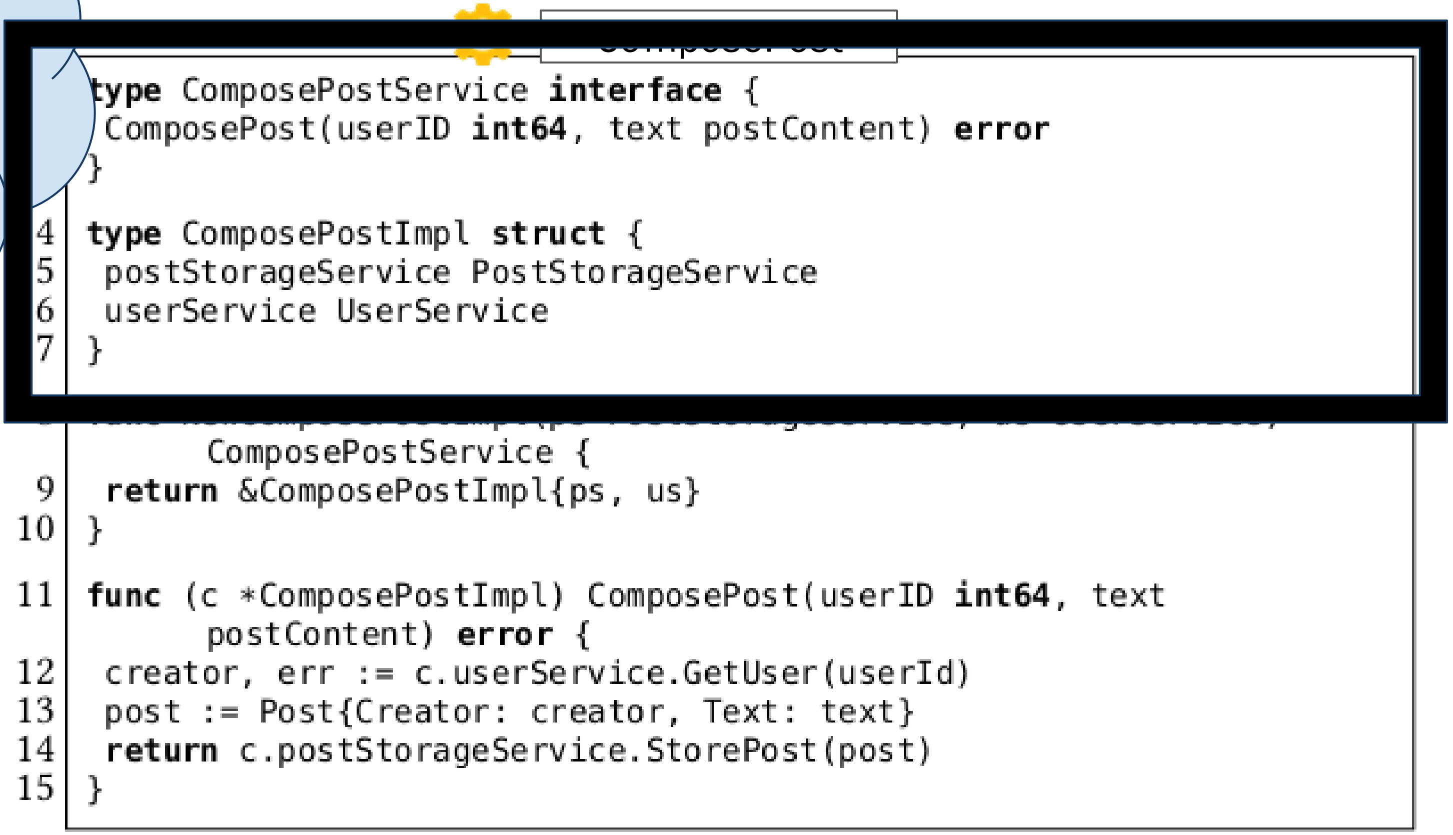
ComposePost

```
type ComposePostService interface {
 ComposePost(userID int64, text postContent) error
}

type ComposePostImpl struct {
 postStorageService PostStorageService
 userService UserService
}

func NewComposePostImpl(ps PostStorageService, us UserService) *
      ComposePostService {
 return &ComposePostImpl{ps, us}
}

func (c *ComposePostImpl) ComposePost(userID int64, text
      postContent) error {
 creator, err := c.userService.GetUser(userId)
 post := Post{Creator: creator, Text: text}
 return c.postStorageService.StorePost(post)
}
```

Workflow spec at a glance

Business Logic

Contains the service declarations

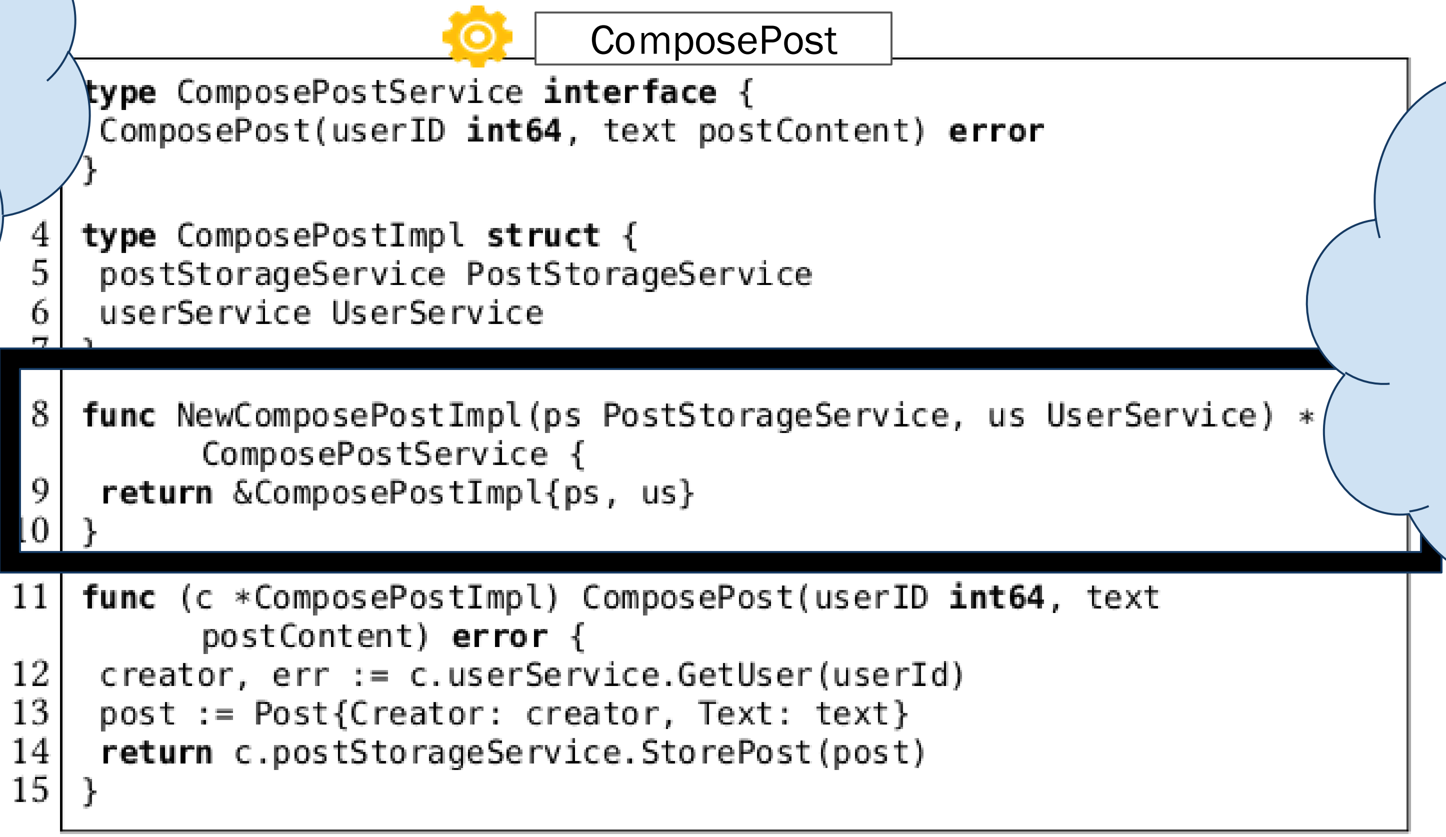


```go
type ComposePostService interface {
 ComposePost(userID int64, text postContent) error
}

type ComposePostImpl struct {
 postStorageService PostStorageService
 userService UserService
}

func NewComposePostImpl(ps PostStorageService, us UserService)
        ComposePostService {
 return &ComposePostImpl{ps, us}
}

func (c *ComposePostImpl) ComposePost(userID int64, text
        postContent) error {
 creator, err := c.userService.GetUser(userId)
 post := Post{Creator: creator, Text: text}
 return c.postStorageService.StorePost(post)
}
```

Workflow spec at a glance

Business Logic

Contains the service declarations



ComposePost

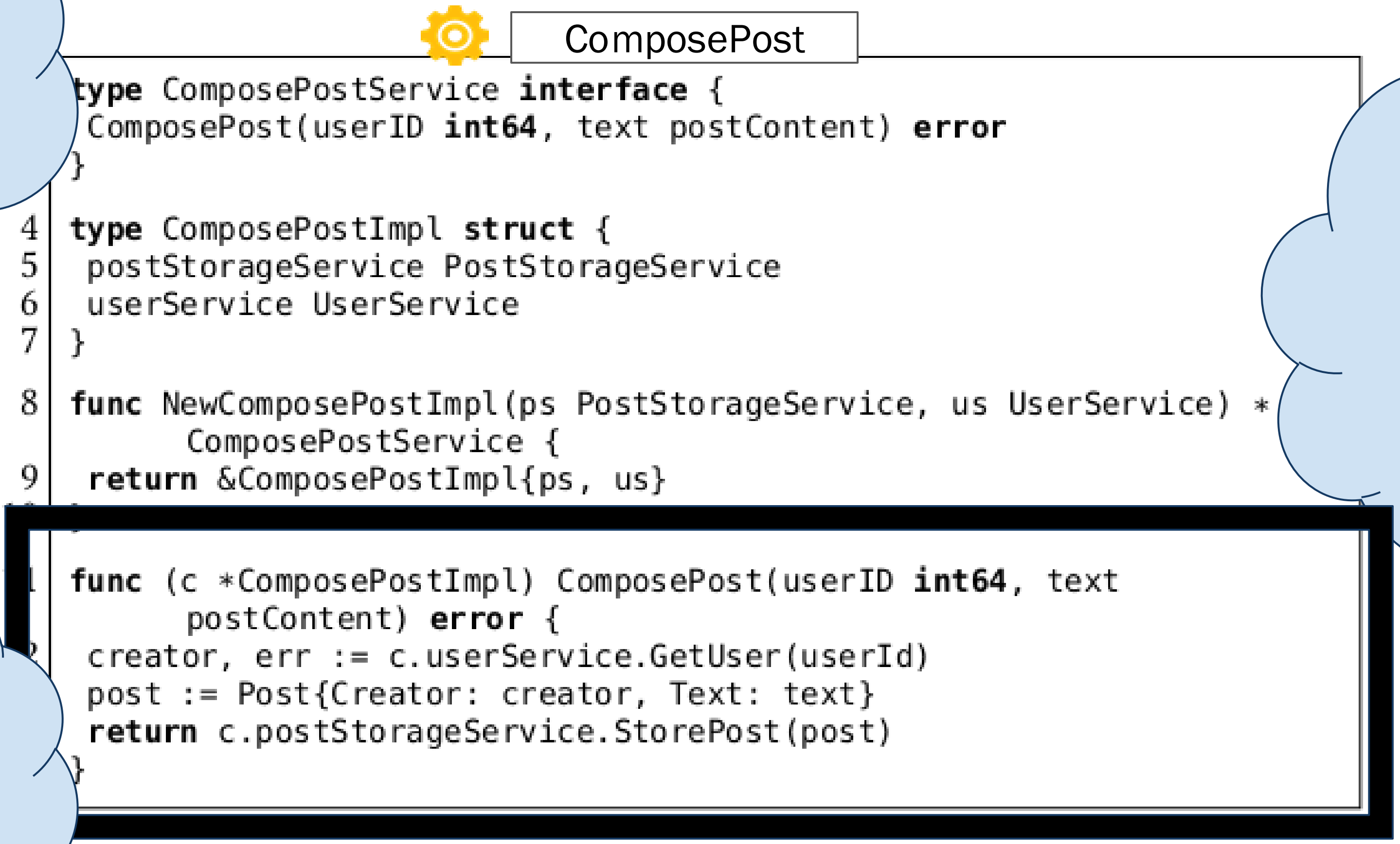
```go
type ComposePostService interface {
 ComposePost(userID int64, text postContent) error
}

type ComposePostImpl struct {
 postStorageService PostStorageService
 userService UserService
}

func NewComposePostImpl(ps PostStorageService, us UserService) *
       ComposePostService {
 return &ComposePostImpl{ps, us}
}
func (c *ComposePostImpl) ComposePost(userID int64, text
       postContent) error {
 creator, err := c.userService.GetUser(userId)
 post := Post{Creator: creator, Text: text}
 return c.postStorageService.StorePost(post)
}
```

Dependencies are passed as parameters to the constructor

Workflow spec at a glance

Business Logic

Contains the service declarations



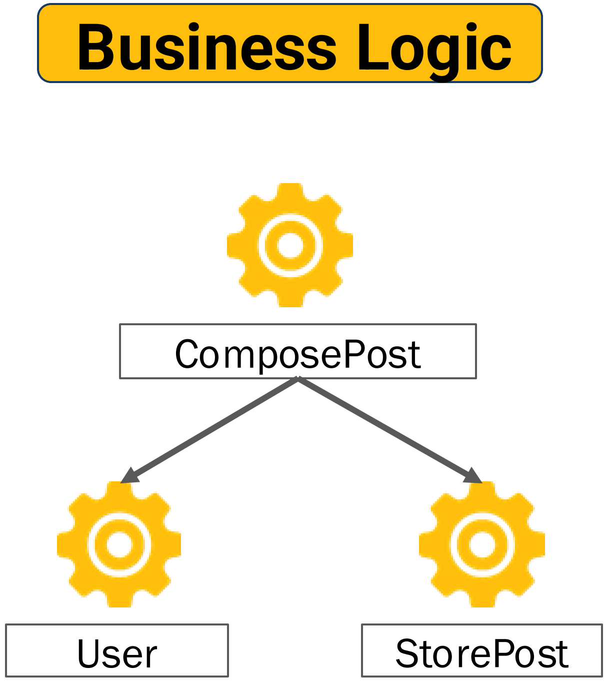
```
type ComposePostService interface {
 ComposePost(userID int64, text postContent) error
}

type ComposePostImpl struct {
 postStorageService PostStorageService
 userService UserService
}

func NewComposePostImpl(ps PostStorageService, us UserService) *
       ComposePostService {
 return &ComposePostImpl{ps, us}
}

func (c *ComposePostImpl) ComposePost(userID int64, text
       postContent) error {
 creator, err := c.userService.GetUser(userId)
 post := Post{Creator: creator, Text: text}
 return c.postStorageService.StorePost(post)
}
```

Dependencies are passed as parameters to the constructor

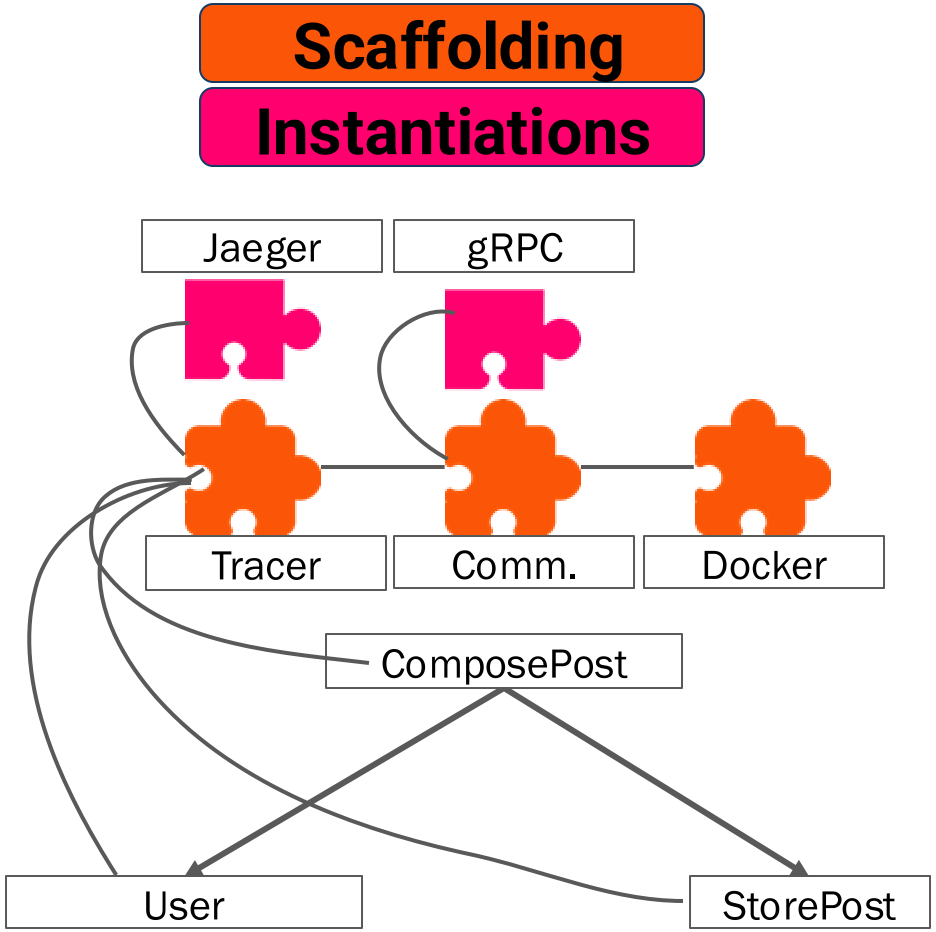
Each function can be implemented in a few lines

Separate 3 pieces into 2 input specs

Workflow Spec



Wiring Spec



Wiring spec at a glance

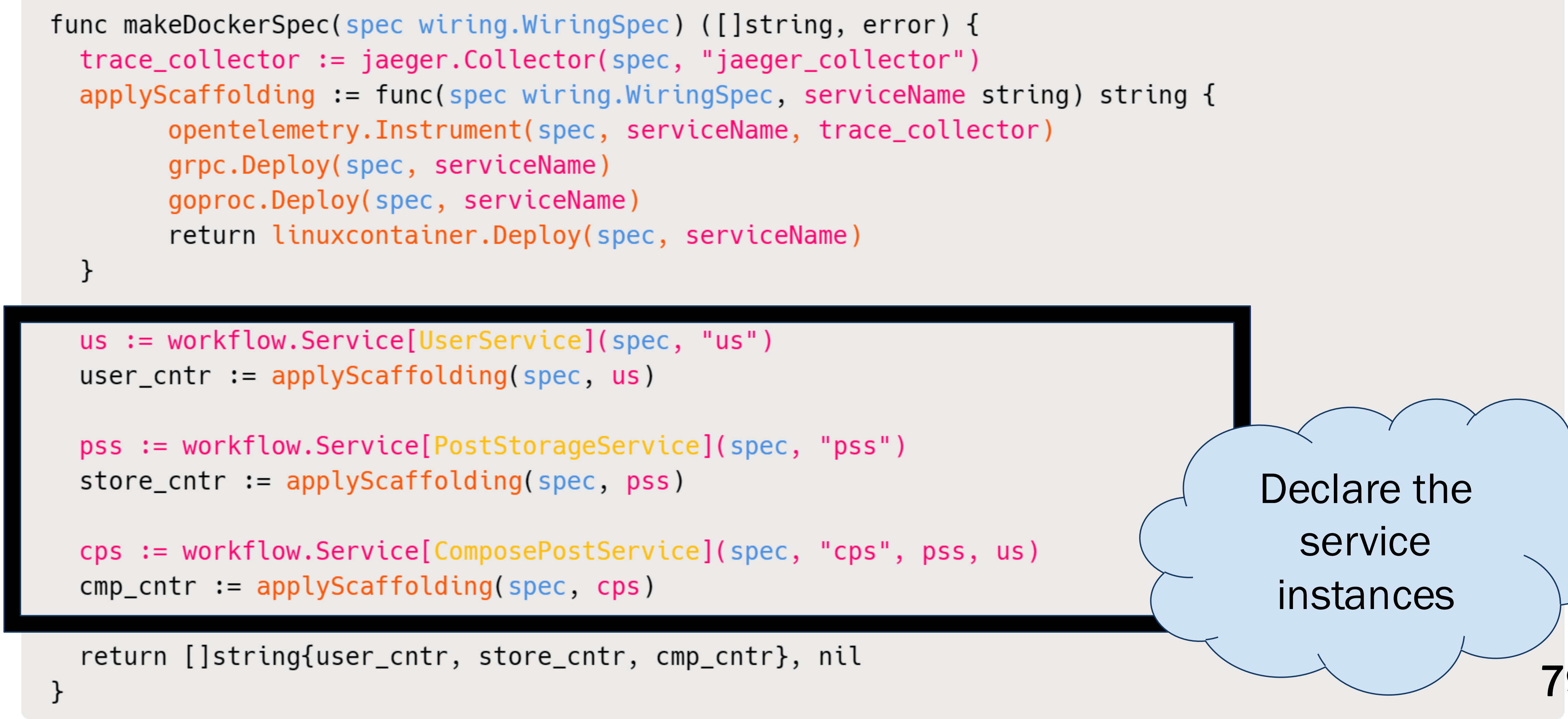
Scaffolding

Instantiations

Wiring spec at a glance

Scaffolding

Instantiations



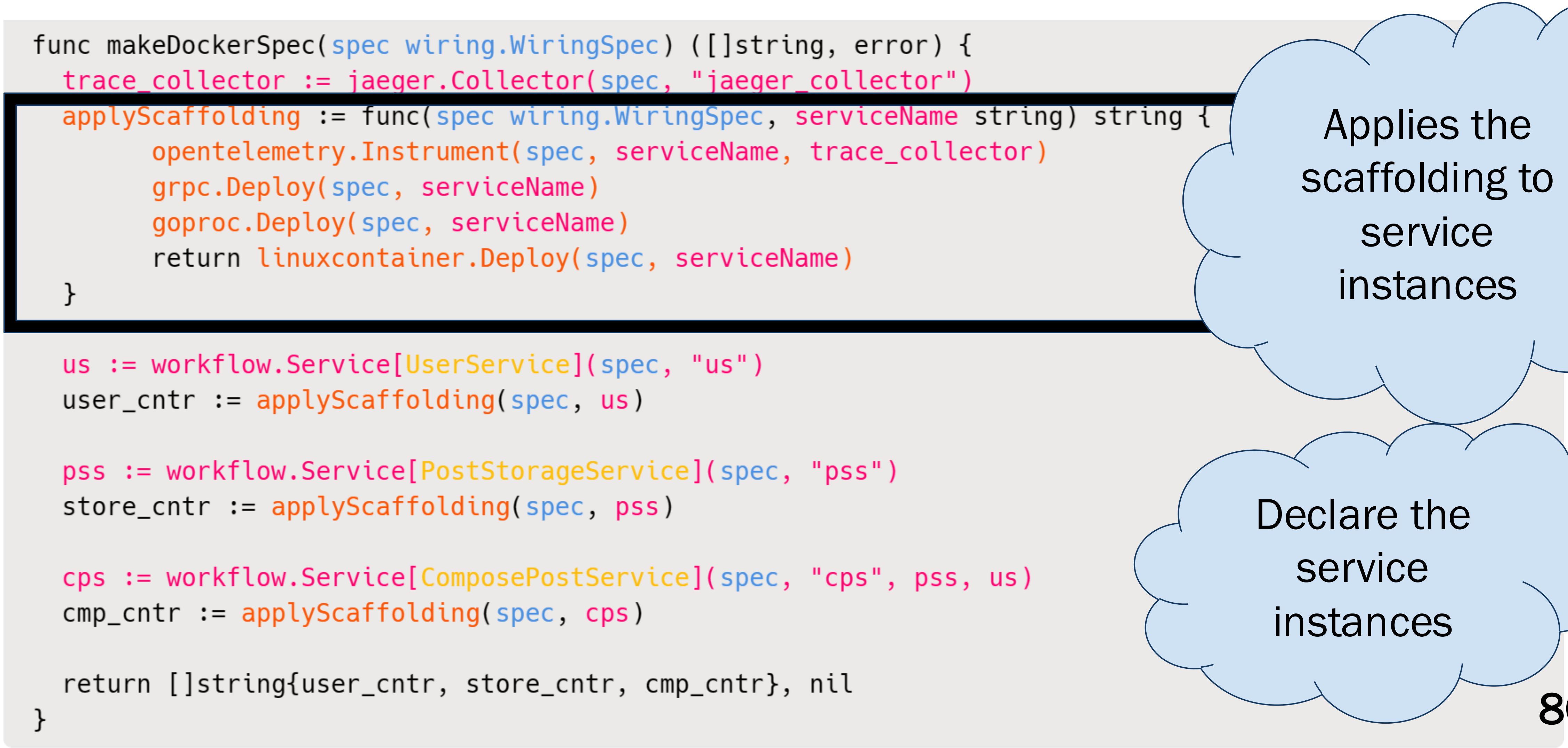
```
func makeDockerSpec(spec wiring.WiringSpec) ([]string, error) {
  trace_collector := jaeger.Collector(spec, "jaeger_collector")
  applyScaffolding := func(spec wiring.WiringSpec, serviceName string) string {
        opentelemetry.Instrument(spec, serviceName, trace_collector)
        grpc.Deploy(spec, serviceName)
        goproc.Deploy(spec, serviceName)
        return linuxcontainer.Deploy(spec, serviceName)
  }

  us := workflow.Service[UserService](spec, "us")
  user_cntr := applyScaffolding(spec, us)

  pss := workflow.Service[PostStorageService](spec, "pss")
  store_cntr := applyScaffolding(spec, pss)

  cps := workflow.Service[ComposePostService](spec, "cps", pss, us)
  cmp_cntr := applyScaffolding(spec, cps)

  return []string{user_cntr, store_cntr, cmp_cntr}, nil
}
```

Declare the service instances

Wiring spec at a glance

Scaffolding

Instantiations



```
func makeDockerSpec(spec wiring.WiringSpec) ([]string, error) {
  trace_collector := jaeger.Collector(spec, "jaeger_collector")
  applyScaffolding := func(spec wiring.WiringSpec, serviceName string) string {
        opentelemetry.Instrument(spec, serviceName, trace_collector)
        grpc.Deploy(spec, serviceName)
        goproc.Deploy(spec, serviceName)
        return linuxcontainer.Deploy(spec, serviceName)
  }

  us := workflow.Service[UserService](spec, "us")
  user_cntr := applyScaffolding(spec, us)

  pss := workflow.Service[PostStorageService](spec, "pss")
  store_cntr := applyScaffolding(spec, pss)

  cps := workflow.Service[ComposePostService](spec, "cps", pss, us)
  cmp_cntr := applyScaffolding(spec, cps)

  return []string{user_cntr, store_cntr, cmp_cntr}, nil
}
```

Applies the scaffolding to service instances

Declare the service instances