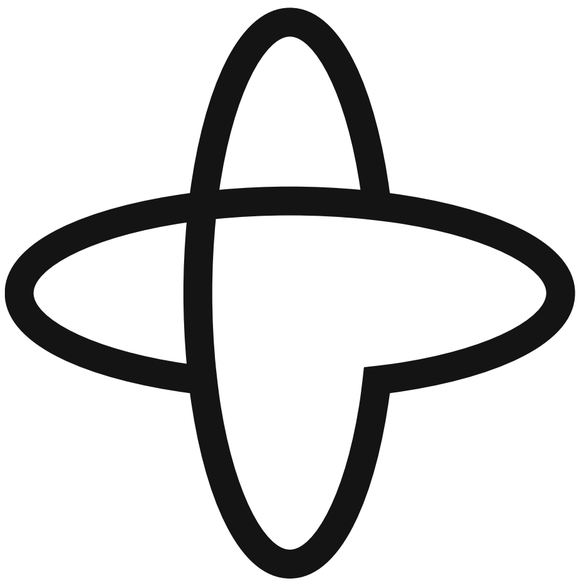


Temporal

INTRO WORKSHOP

AGENDA

- History
- Temporal
- Workshop

HISTORY

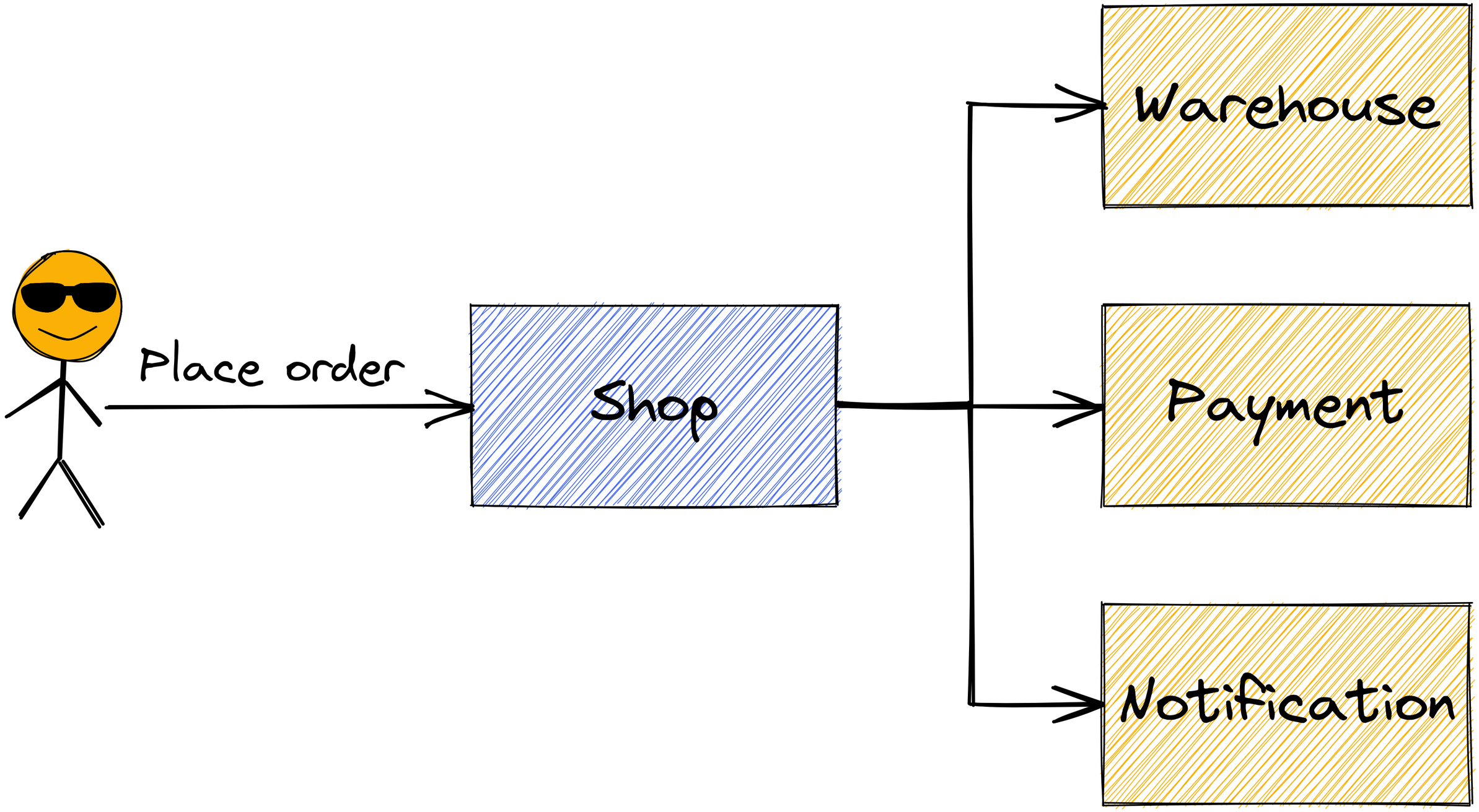
PROBLEM

Long running, complex interactions...

...in distributed systems...

...with transactional characteristics.

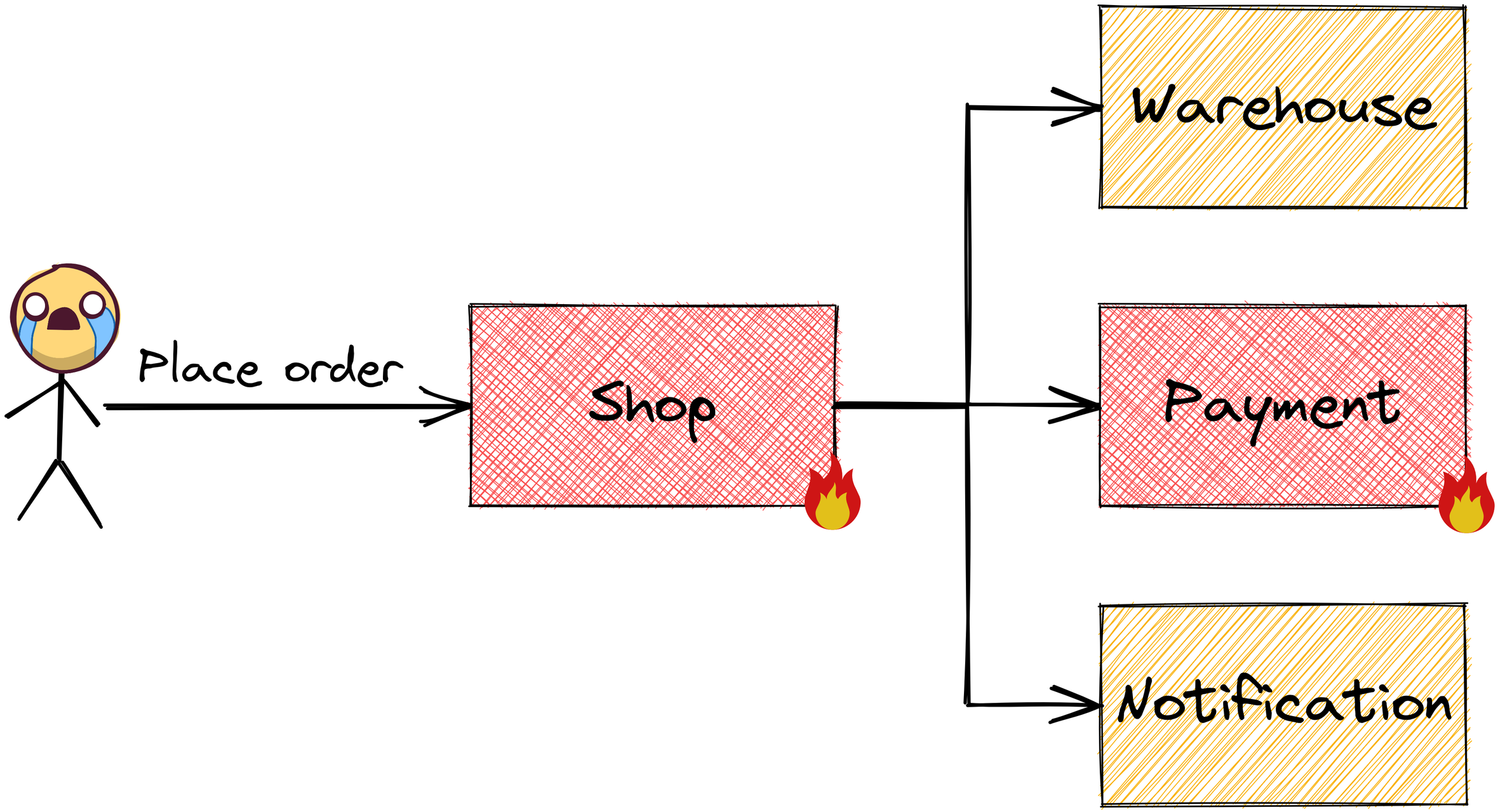
EXAMPLE



NAIVE IMPLEMENTATION

```
func placeOrder(o Order) {
    warehouse.ReserveItems(o.Items)

    payment.ProcessPayment(o)

    notifyBuyer(o, OrderPlaced)

    saveOrder(o)
}
```

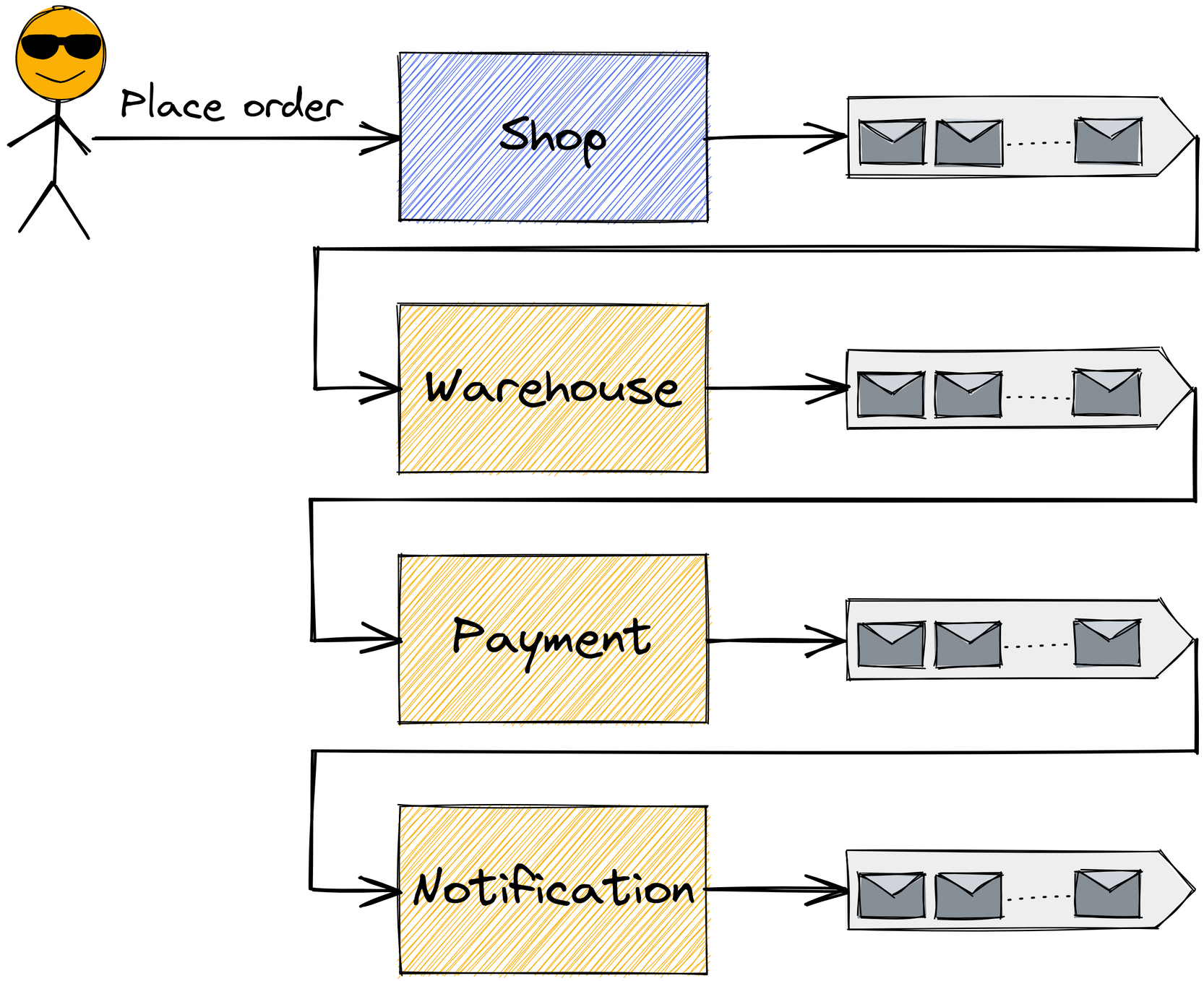
FAILURE



FAILURE MODES

```
func placeOrder(o Order) {
    reserved, err := warehouse.ReserveItems(o.Items)
    if err != nil { // Warehouse service unavailable
        // Retry?
    }

    if !reserved { // Business error requiring user intervention
        notifySeller(o, OrderFailed)
        notifyBuyer(o, OrderFailed)

        return
    }

    // ...
}
```

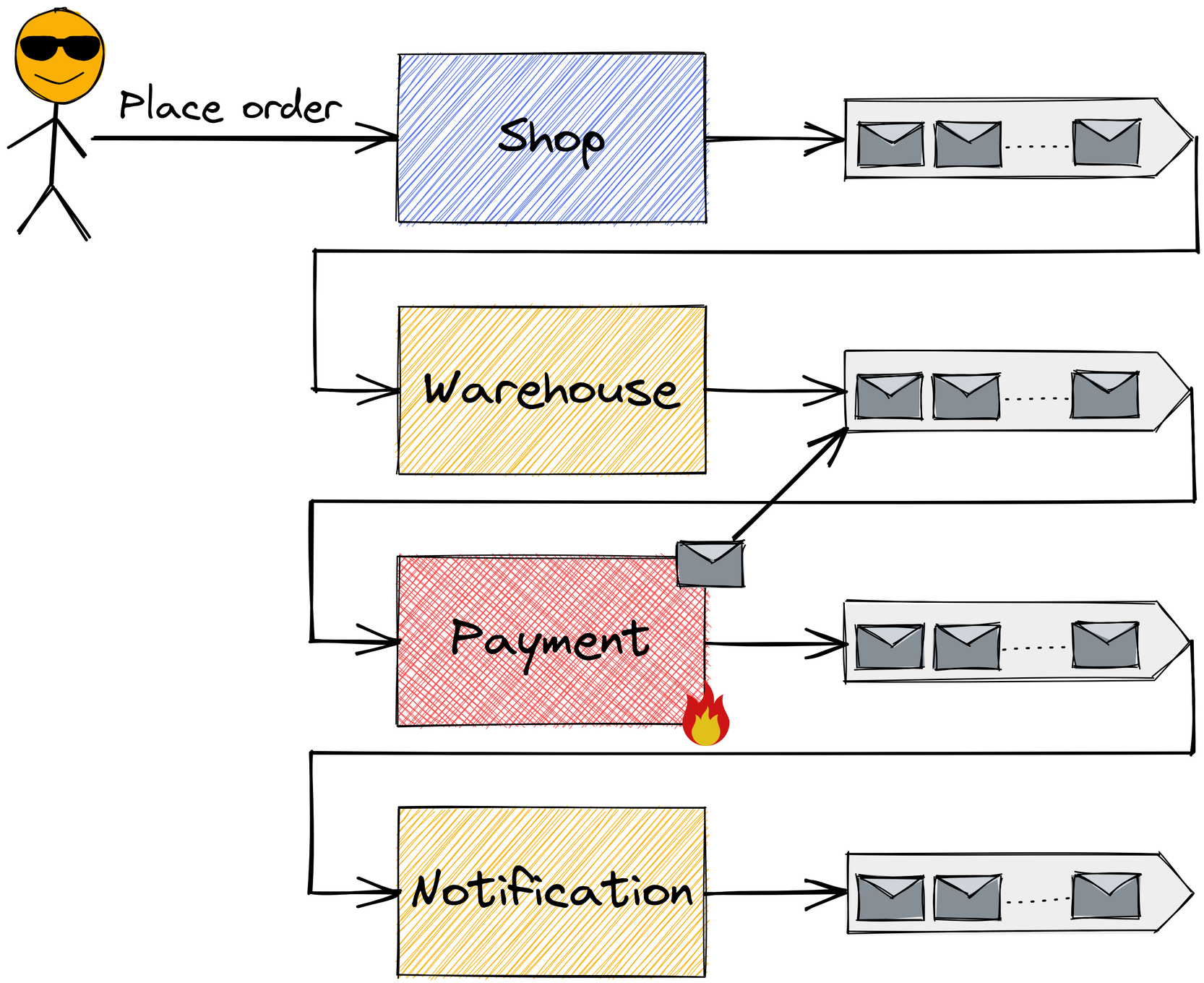
QUEUES FTW



EVENT CHOREOGRAPHY

Service	Input	Output
Shop	`PlaceOrder`*	`OrderPlaced`
Warehouse	`OrderPlaced`	`ItemsReserved`/ `ItemReservationFa`
Payment	`ItemsReserved`	`OrderPaid`/ `PaymentFailed`
Notification	`OrderPaid`/ `PaymentFailed`/ `ItemReservationFailed`	`BuyerNotified`

FAILURE WITH QUEUES

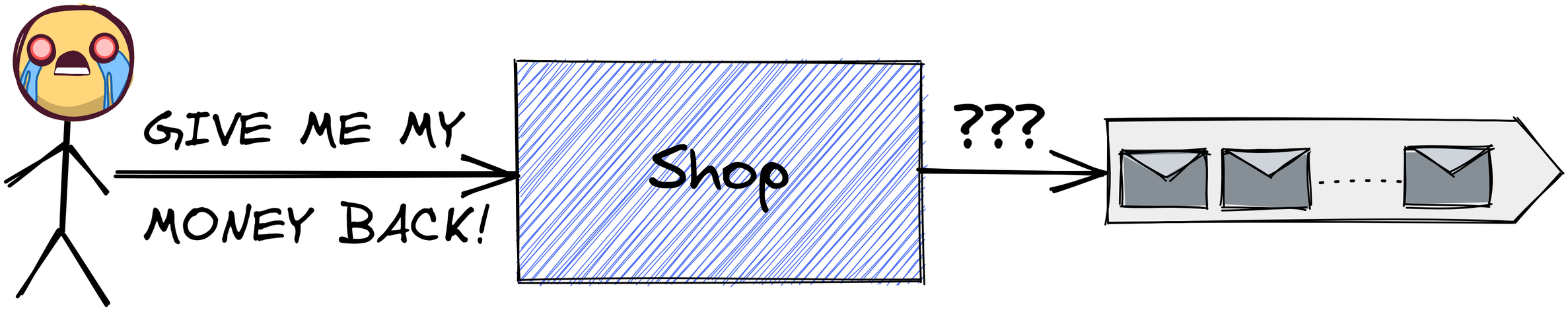


STATUS?

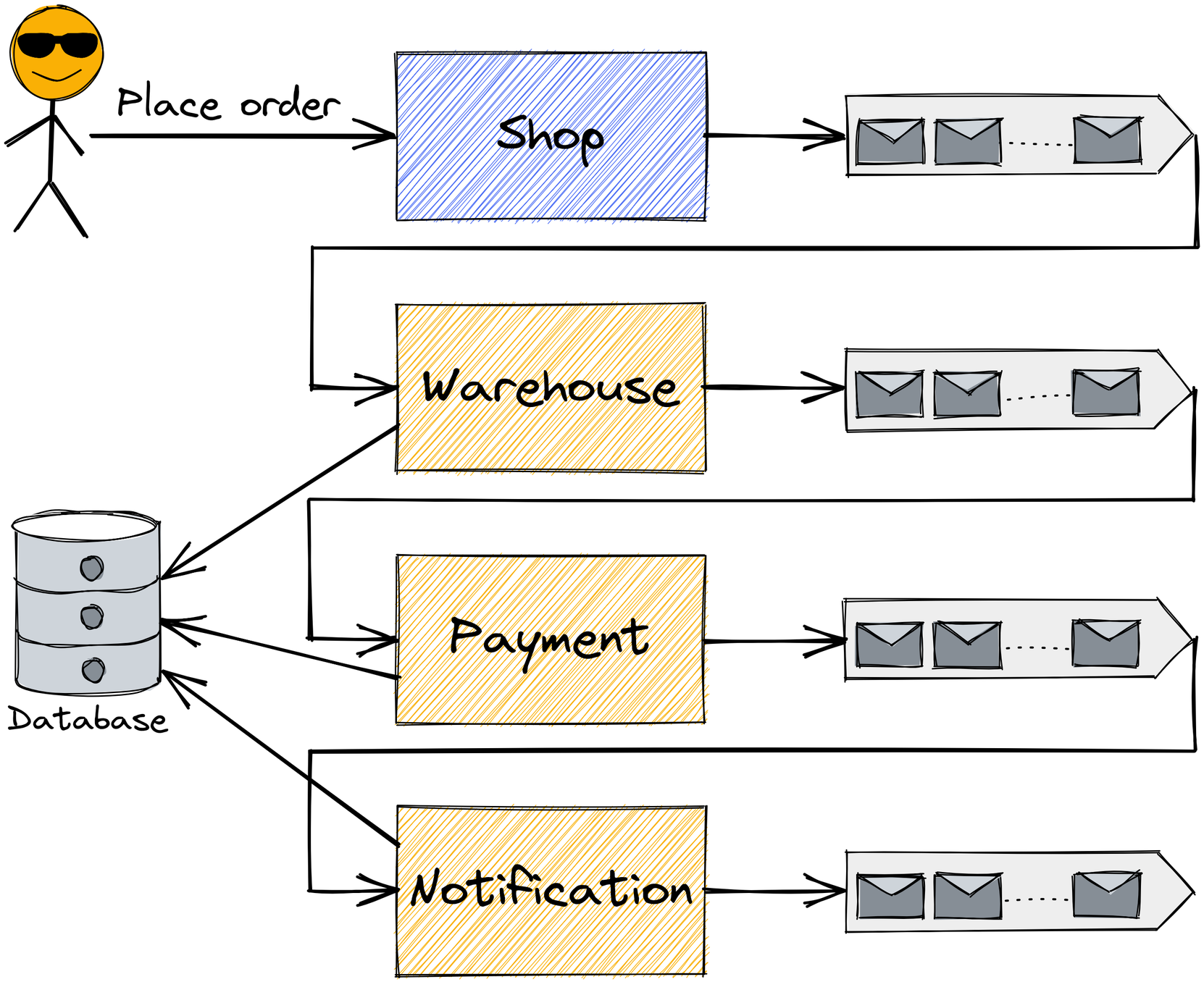
- **User:** what's going on with my order?
- **Seller:** incoming orders?
- **Developer:** debug a single execution?

CANCELLATION?

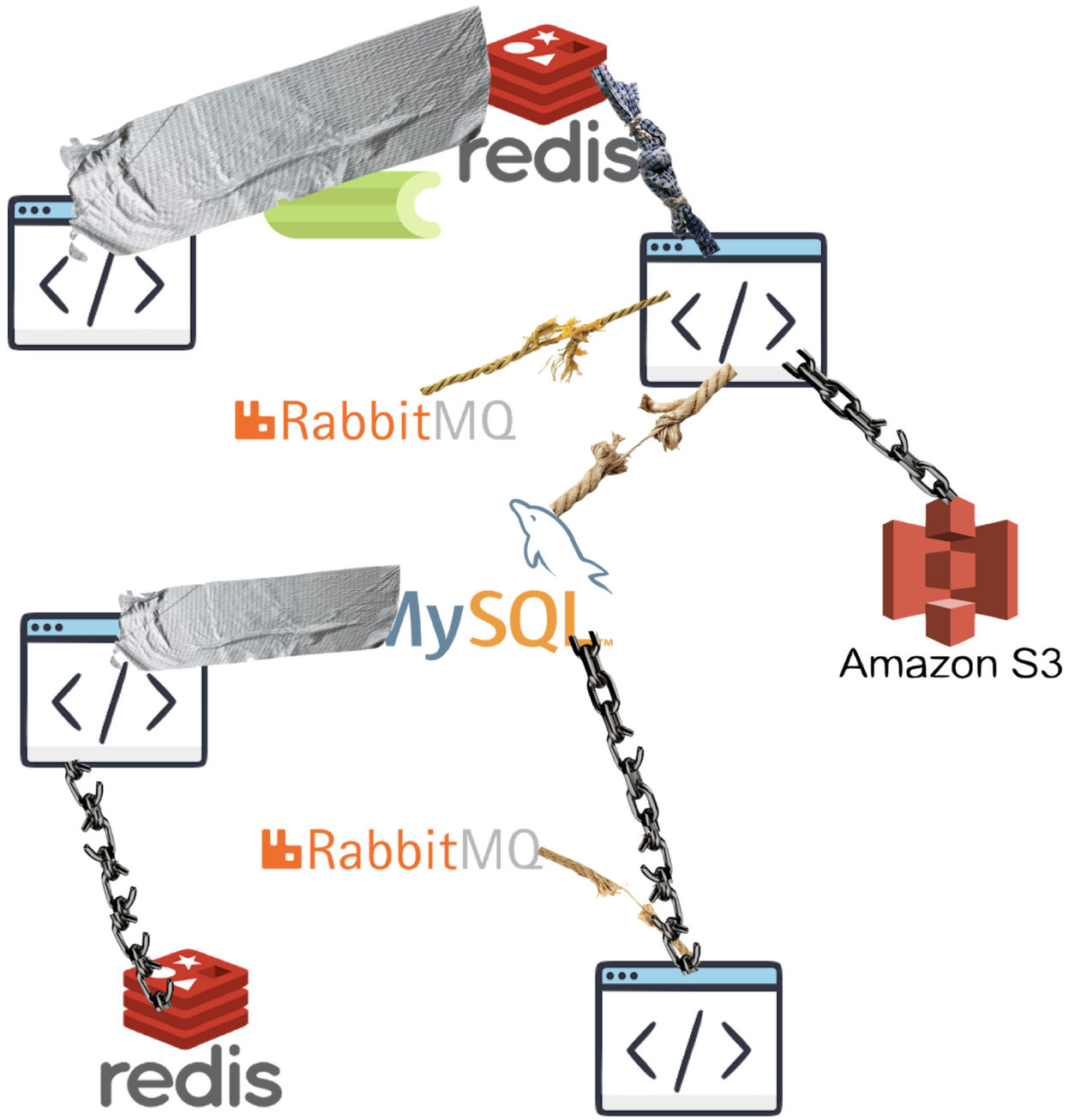
- **User:** I changed my mind, give me my money back!
- **Payment failed:** User failed to pay the order, cancel it!



STATE!


Place order
Shop
Warehouse
Payment
Notification
Database

REINVENTING WHEELS



Source: StackOverflow Blog

LESSONS LEARNT

- No *one size fits all* solution (yet)
- Fragile systems
- Lack of orchestration

LACK OF ORCHESTRATION

- Fractured business processes
- Tight coupling between components
- Cancellations?
- Compensating actions?
- Additional interactions?
- Troubleshooting?

WE WANT THIS

```
func placeOrder(o Order) {
    warehouse.ReserveItems(o.Items)

    payment.ProcessPayment(o)

    notifyBuyer(o, OrderPlaced)

    saveOrder(o)
}
```

TEMPORAL

WHAT IS TEMPORAL?

- Temporal service
- Temporal SDK *(insert your programming language here)*

ORCHESTRATION AS CODE

- Write business logic as plain code
- Orchestration framework
- Durability and reliability out-of-the-box

CONCEPTS

- Workflow
- Activity
- Worker

Documentation

WORKFLOW

- Definition
- Type
- Execution

Documentation

WORKFLOW DEFINITION

- aka. *Workflow Function*
- Encapsulates business logic
- Required to be **deterministic**
- Implemented in the *Worker*

Documentation

DETERMINISM

Output is based entirely on the input.

```go
func add(a, b int) int {
    return a + b
}
```

```go
func add(a, b int) int {
    // This is not deterministic
    resp := http.Get(fmt.Sprintf("https://add.com/%d/%d", a, b))

    return decodeBody(resp.Body)
}
```

WORKFLOW TYPE

Identifies a *Workflow Definition* (in the scope of a *Task Queue*)

Documentation

WORKFLOW EXECUTION

- Durable and reliable execution of a *Workflow Definition*
- Runs once to completion
- Executed by the *Worker*

Documentation

ACTIVITY

- Definition
- Type
- Execution

Documentation

ACTIVITY DEFINITION

- aka. *Activity Function*
- Building blocks for *Workflow(Definition)s*
- No restrictions on the code (ie. can be non-deterministic)
- Asynchronously executed
- Generally **idempotent**

Documentation

IDEMPOTENCE

Applying an operation multiple times does not change the result beyond the initial application.

ACTIVITY TYPE

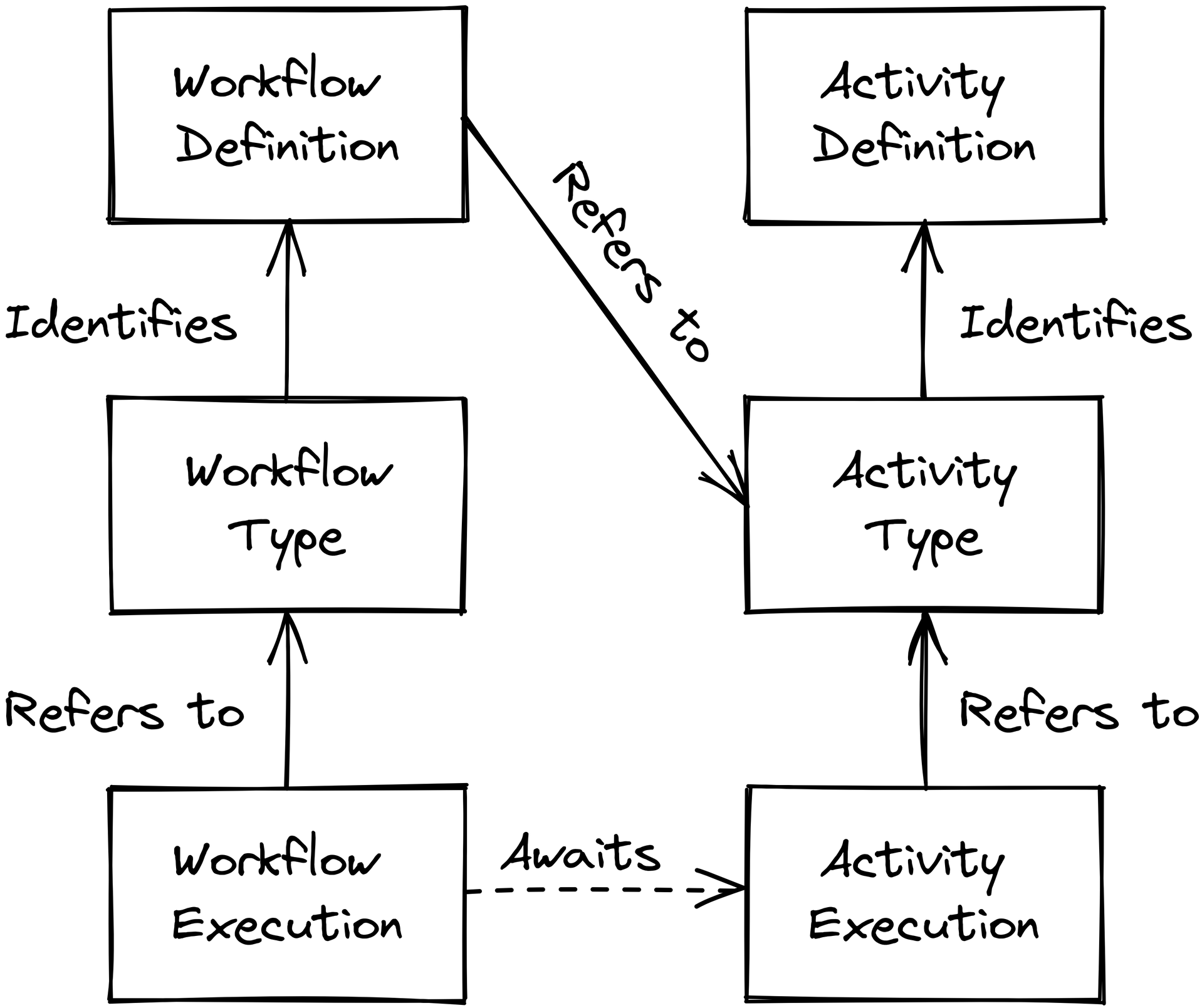
Identifies an *Activity Definition* (in the scope of a *Task Queue*)

Documentation

ACTIVITY EXECUTION

- Execution of an *Activity Definition*
- Can timeout
- Can be retried
- At least once execution guarantee
- Runs to completion or exhausts timeouts/retries
- Executed by the *Worker*

Documentation


Workflow Definition
Activity Definition
Refers to
Identifies
Identifies
Workflow Type
Activity Type
Refers to
Refers to
Workflow Execution
Awaits
Activity Execution

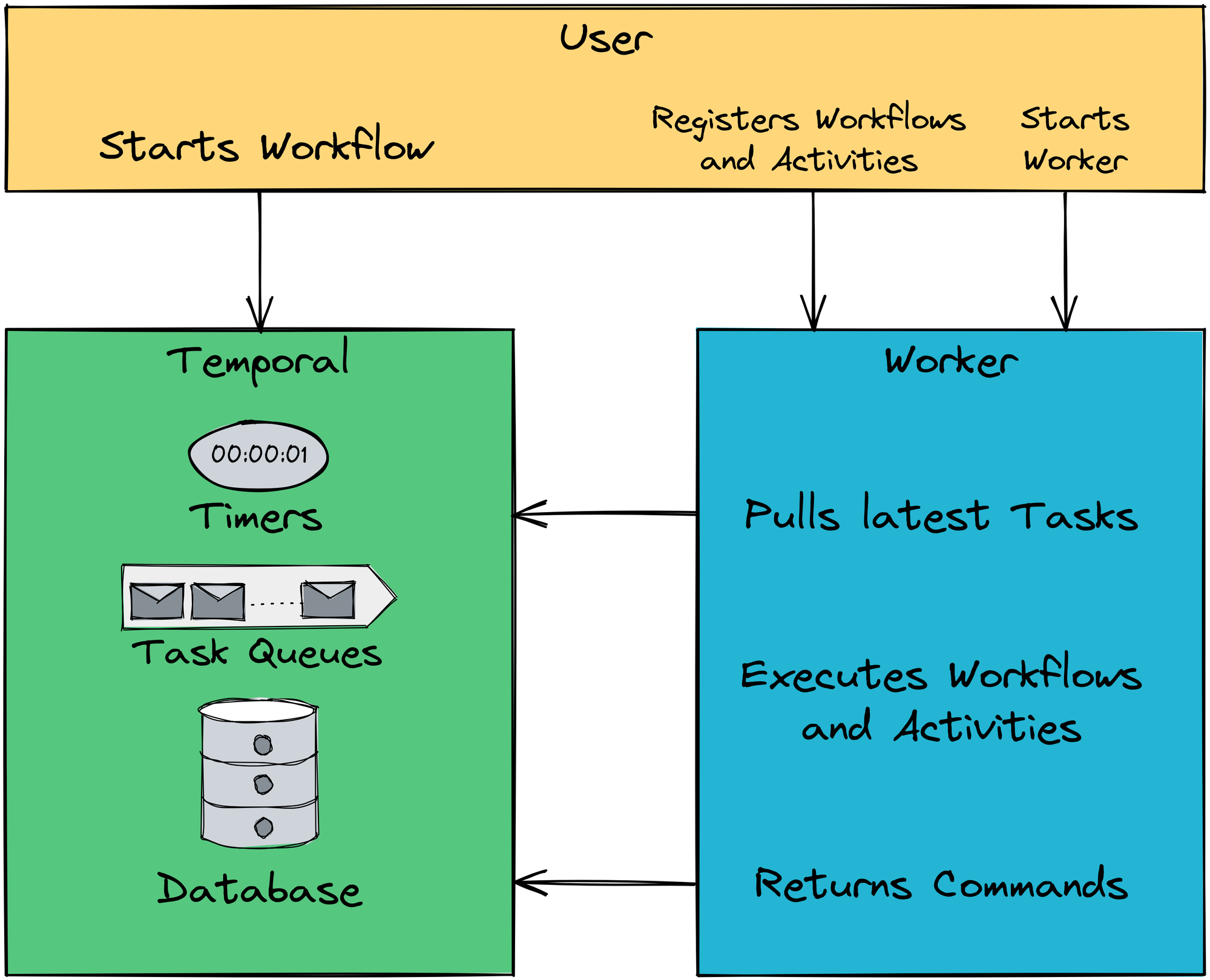
EXAMPLE

```go
// workflow type: "placeOrder"
func placeOrder(o Order) {
    for _, item := range o.Items {
        err := temporal.ExecuteActivity("warehouse.ReserveItem", item)
        // handle error
    }

    // ...
}

// activity type: "warehouse.ReserveItem"
func reserveItem(item OrderItem) error {
    if !items.IsAvailable(item.Quantity) { return ErrItemNotAvailable }

    // ..

    return nil
}
```

WORKER

- Implemented and operated by the user
- Executes *Workflows* and *Activities*
- Listens to *Task Queues*

Documentation


User
Starts Workflow
Registers Workflows and Activities
Starts Worker
Temporal
00:00:01
Timers
Task Queues
Database
Worker
Pulls latest Tasks
Executes Workflows and Activities
Returns Commands

OTHER NOTABLE CONCEPTS

- **Namespace**: unit of isolation and replication domain (analogous to a database)
- **Task Queue**: routing mechanism to different kinds of *Workers*

PREPARATION

PREPARE YOUR ENVIRONMENT

1. Git, Make, etc.
2. Make sure you have the latest Go and Docker installed

SETUP THE PROJECT

Checkout the following repository:

https://github.com/sagikazarmark/temporal-intro-workshop

Follow the instructions in the README.

CHECK THE TOOLS

- UI: http://127.0.0.1:8080
- CLI: `make shell`

WORKFLOWS

- Write business logic as code
- **MUST** be deterministic
- Parameters **MUST** be serializable

REMINDER

```
func placeOrder(o Order) {
    warehouse.ReserveItems(o.Items)

    payment.ProcessPayment(o)

    notifyBuyer(o, OrderPlaced)

    saveOrder(o)
}
```

EXAMPLE 1

Simple workflow function.

EXAMPLE 2

- Input: number (integer)
- Output: factorial of the number

EXAMPLE 3

Writing unit tests for a workflow.

EXAMPLE 4

Write a test for Example 2.

DETERMINISM

Output value is based entirely on the input.

```go
func add(a, b int) int {
    return a + b
}
```

```go
func add(a, b int) int {
    // This is not deterministic
    resp := http.Get(fmt.Sprintf("https://add.com/%d/%d", a, b))

    return decodeBody(resp.Body)
}
```

FORBIDDEN IN GO

- Time functions `time.Now`, `time.Sleep`
- Goroutines
- Channels and selects
- Iterating over maps

Use deterministic wrappers instead.

FORBIDDEN IN GENERAL

- Accessing external systems (usually over network)
- Accessing the filesystem
- Generating random values

EXAMPLE 5

Side-effects in a workflow.

EXAMPLE 6

Communicating with a running workflow.

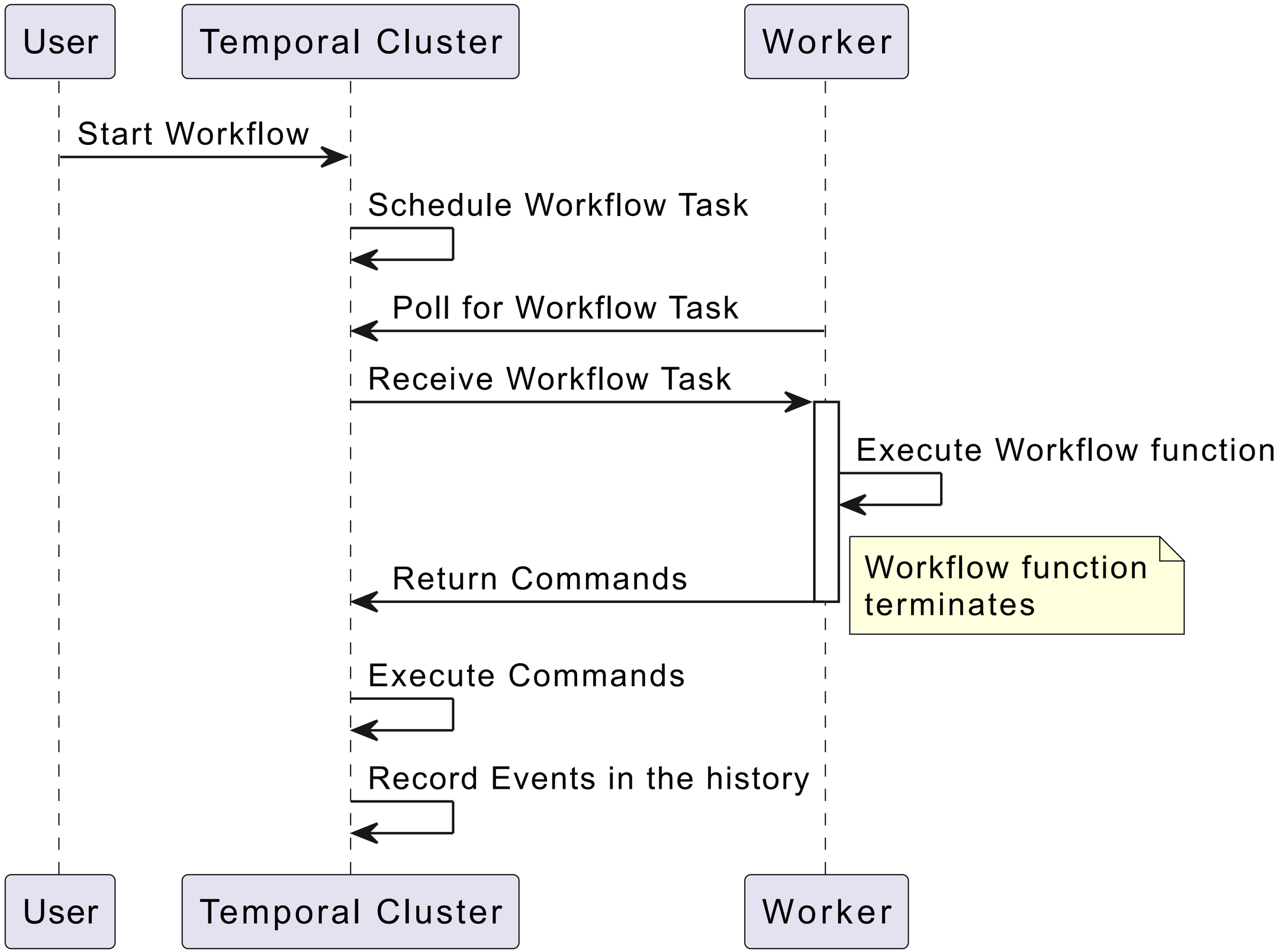
EXAMPLE 7

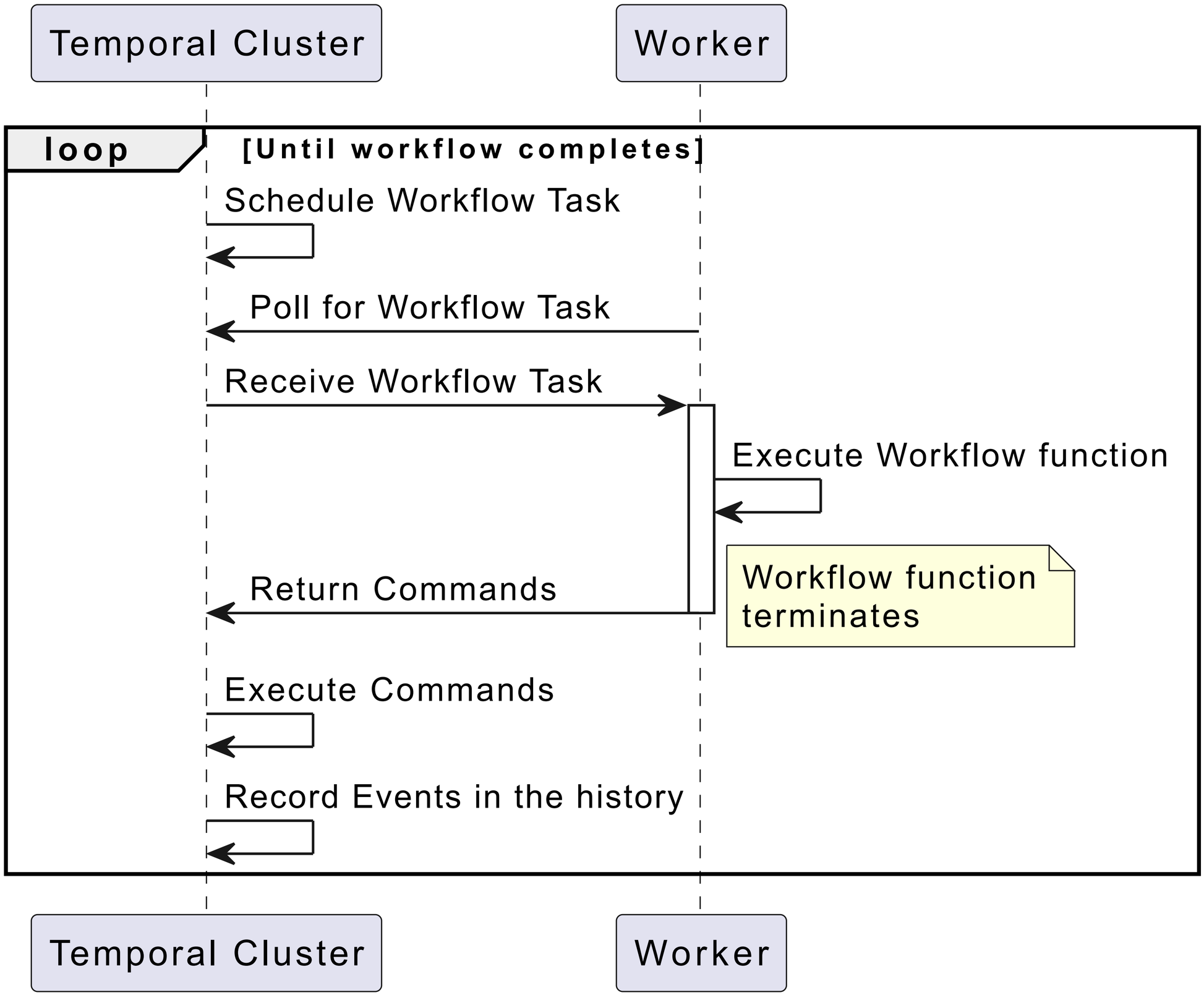
Write a query handler (for your workflow from examples 2, 4) that returns the current result in the loop.

Tip: Add sleep at the beginning of the loop so you have time to query it using the CLI.

LOG-BASED EXECUTION

- Record a history of events
- *Replay* events to get to the current state


User
Temporal Cluster
Worker
Start Workflow
Schedule Workflow Task
Poll for Workflow Task
Receive Workflow Task
Execute Workflow function
Workflow function terminates
Return Commands
Execute Commands
Record Events in the history
User
Temporal Cluster
Worker


Temporal Cluster
Worker
loop
[Until workflow completes]
Schedule Workflow Task
Poll for Workflow Task
Receive Workflow Task
Execute Workflow function
Workflow function terminates
Return Commands
Execute Commands
Record Events in the history
Temporal Cluster
Worker

WORKFLOW REPLAY

```
1 func Workflow(ctx workflow.Context) error {
2     foo := workflow.ExecuteActivity(ctx, "foo")
3
4     workflow.Sleep(ctx, 10 * time.Second)
5
6     workflow.ExecuteActivity(ctx, "bar", foo)
7 }
```

RECAP

- Workflows implement business logic
- They **MUST** be deterministic (due to log-based execution)

UNDISCUSSED TOPICS

- Child workflows
- Versioning
- Reset / Cancellation
- Search attributes
- Sessions
- Cron
- ...

Documentation: Workflows

Documentation: Workflow development in Go

ACTIVITIES

- Single task within a workflow
- Can be non-deterministic
- API calls, database access, etc
- Just regular code with regular tests

EXAMPLE 8

Simple activity function.

EXAMPLE 9

Activity retry.

EXAMPLE 10

Rewrite the factorial calculation (based on examples 2, 4, 7) as an activity (with retries and timeouts):

- It should always fail on the first attempt
- Rewrite the tests so they continue to pass

UNDISCUSSED TOPICS

- Cancellation
- Async completion
- Local activities

Documentation: Activities

Documentation: Activity development in Go

FURTHER READING

https://stackoverflow.blog/2020/11/23/the-macro-problem-with-microservices/

Documentation: Concepts

Documentation: Glossary

Documentation: Developer's guide

Temporal learning materials

THE END