# Temporal 102

.NET

20240530AG

## Your Instructor

Angela Zhou

**Current:** **Sr. Technical Curriculum Developer @ Temporal**

Past: Software Engineer @ Palo Alto Networks

Math Teacher

## Temporal 102

# Logistics

- Introductions
- Schedule
- Facilities
- WiFi
- Course conventions ("workflow" vs. "Workflow")
- Asking questions
- Getting help with exercises

**Network:** Replay2025
**Password:** Durable!

We welcome your feedback

t.mp/replay25ws

## During this workshop, you will

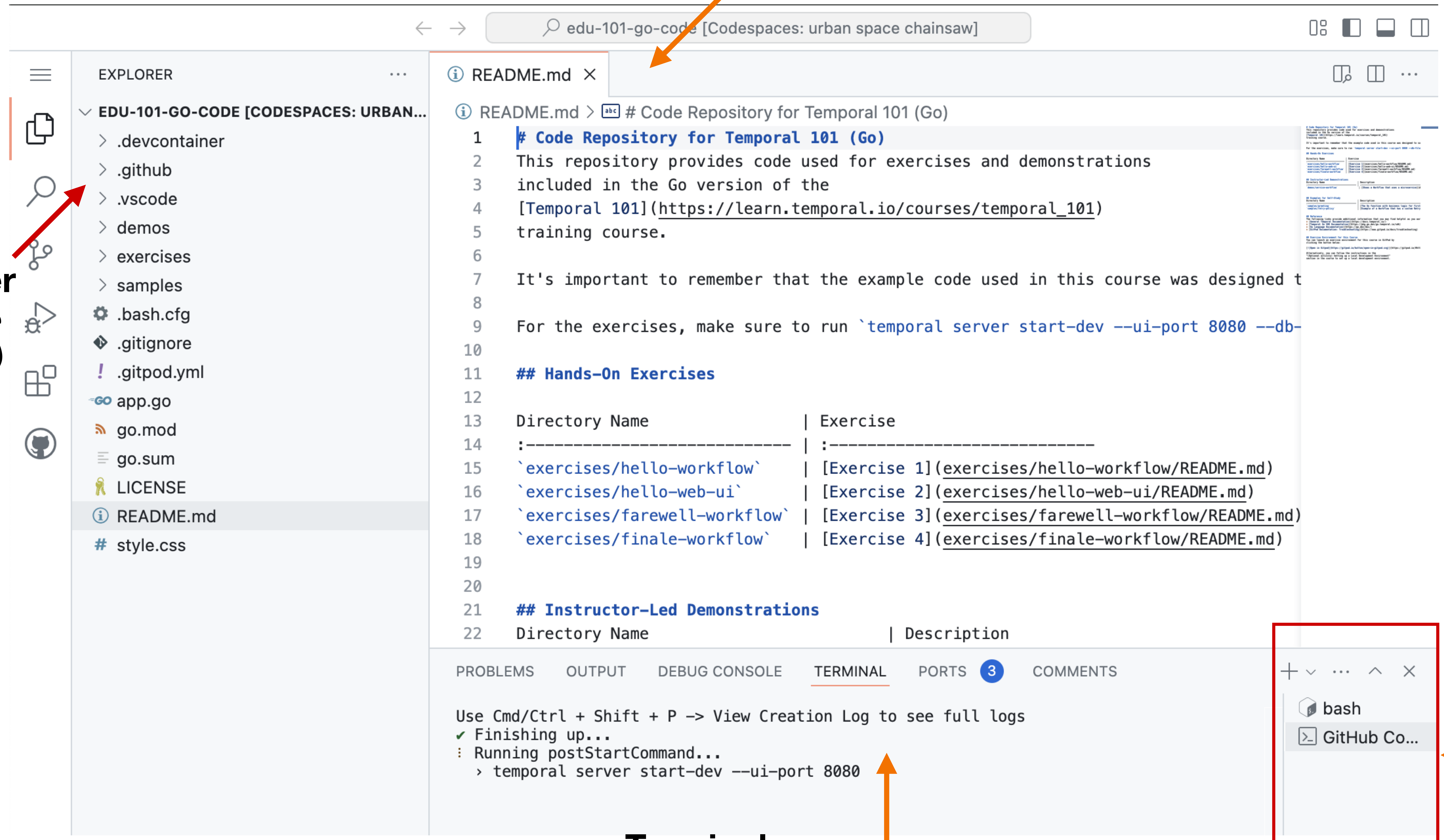
- Evaluate what a **production deployment** of Temporal looks like
- Use **Timers** to introduce delays in Workflow Execution
- Capture runtime information through **logging** in Workflow and Activity code
- Interpret **Event History** and debug problems with Workflow Execution
- Recognize **how Workflow code maps to Commands and Events** during Workflow Execution
- Differentiate **completion, failure, cancelation, and termination** of Workflow Executions
- Consider **why Temporal requires determinism** for Workflow code
- Observe **how Temporal uses History Replay** to achieve durable execution of Workflows
- Leverage the SDK's **testing support** to validate application behavior

## Exercise Environment

- **We provide a development environment for you in this workshop**
  - It uses GitHub Codespaces to deploy a Temporal Service, plus a code editor and terminal
  - You access it through your browser (requires you to log in to GitHub)
  - Your instructor will now demonstrate how to access and use it

t.mp/edu-102-dotnet-code

# Codespaces Overview



```
# Code Repository for Temporal 101 (Go)
This repository provides code used for exercises and demonstrations
included in the Go version of the
[Temporal 101](https://learn.temporal.io/courses/temporal_101)
training course.

It's important to remember that the example code used in this course was designed t

For the exercises, make sure to run `temporal server start-dev --ui-port 8080 --db-

## Hands-On Exercises

Directory Name                   | Exercise
:------------------------------- | :-----------------------------
`exercises/hello-workflow`       | [Exercise 1](exercises/hello-workflow/README.md)
`exercises/hello-web-ui`         | [Exercise 2](exercises/hello-web-ui/README.md)
`exercises/farewell-workflow`    | [Exercise 3](exercises/farewell-workflow/README.md)
`exercises/finale-workflow`      | [Exercise 4](exercises/finale-workflow/README.md)


## Instructor-Led Demonstrations
Directory Name                                   | Description
```

```
Use Cmd/Ctrl + Shift + P -> View Creation Log to see full logs
✔ Finishing up...
⋮ Running postStartCommand...
  › temporal server start-dev --ui-port 8080
```

# Temporal 102

## Temporal: A Durable Execution System

- **What is a durable execution system?**
  - Ensures that your application runs reliably despite adverse conditions
  - Automatically maintains application state and recovers from failure

# Temporal: A Durable Execution System

- **What is a durable execution system?**
  - Ensures that your application runs reliably despite adverse conditions
  - Automatically maintains application state and recovers from failure
  - Improves developer productivity by making applications easier to develop, scale, and support

# Temporal Workflows

- **Workflows are the core abstraction in Temporal**
  - It represents the sequence of steps used to carry out your business logic
  - They are durable: Temporal automatically recreates state if execution ends unexpectedly
  - In the .NET SDK, a Temporal Workflow is defined as a class marked with the Workflow attribute
  - Temporal requires that Workflows are *deterministic*

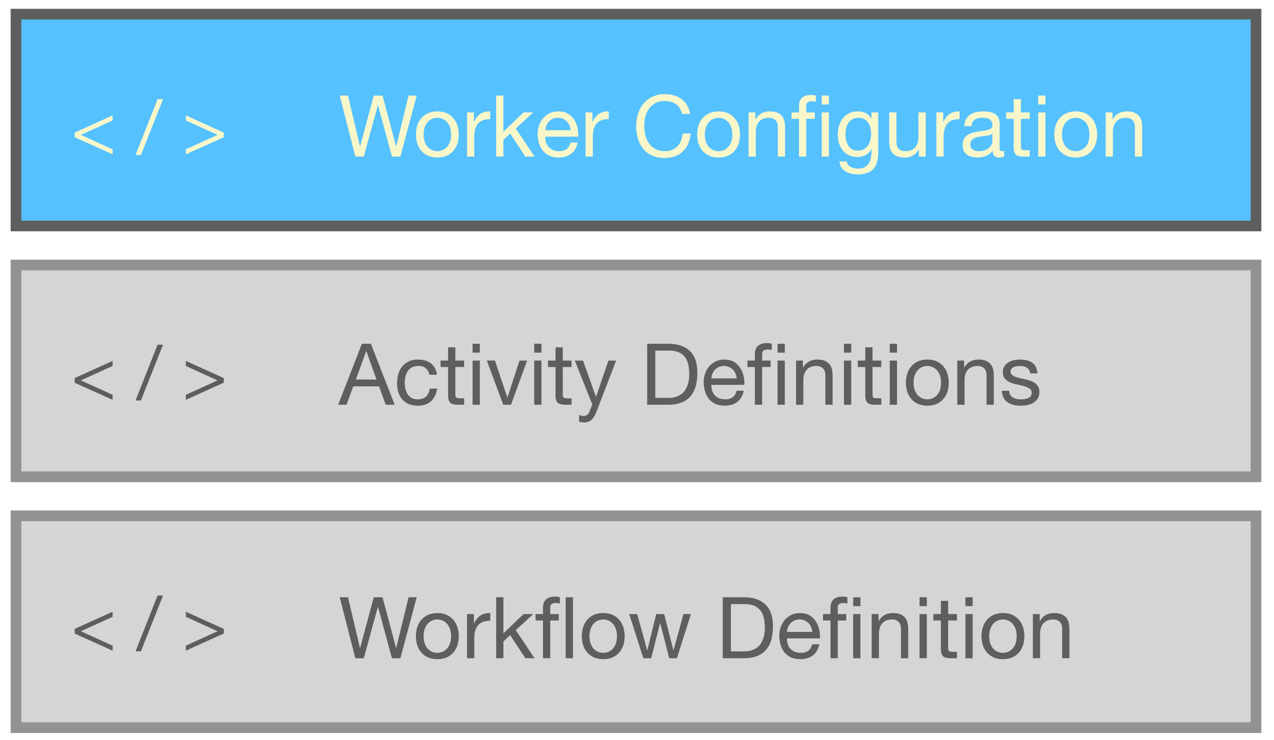
< / > Workflow Definition

## Temporal Activities

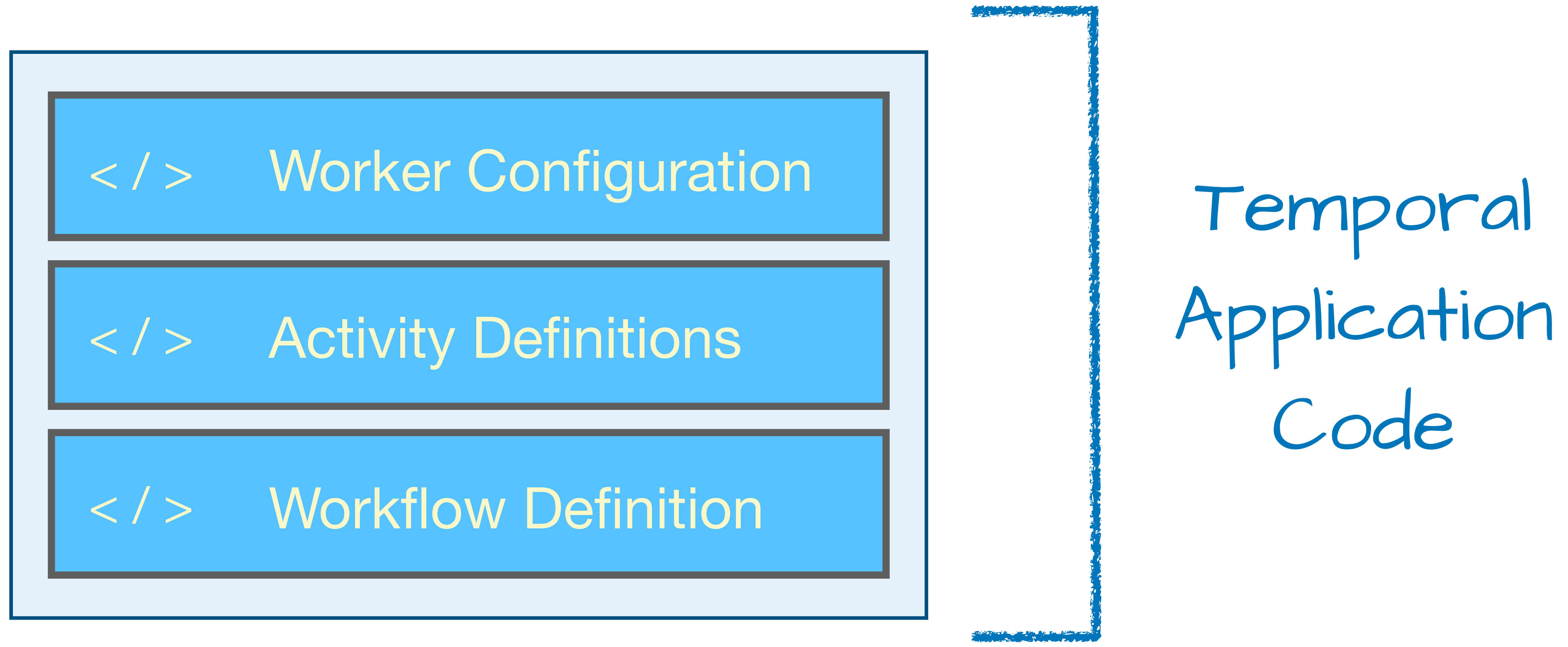
- **Activities encapsulate unreliable or non-deterministic code**
  - They are automatically retried upon failure
  - In the .NET SDK, Activities are defined as a method marked with the Activity attribute

## Temporal Workers

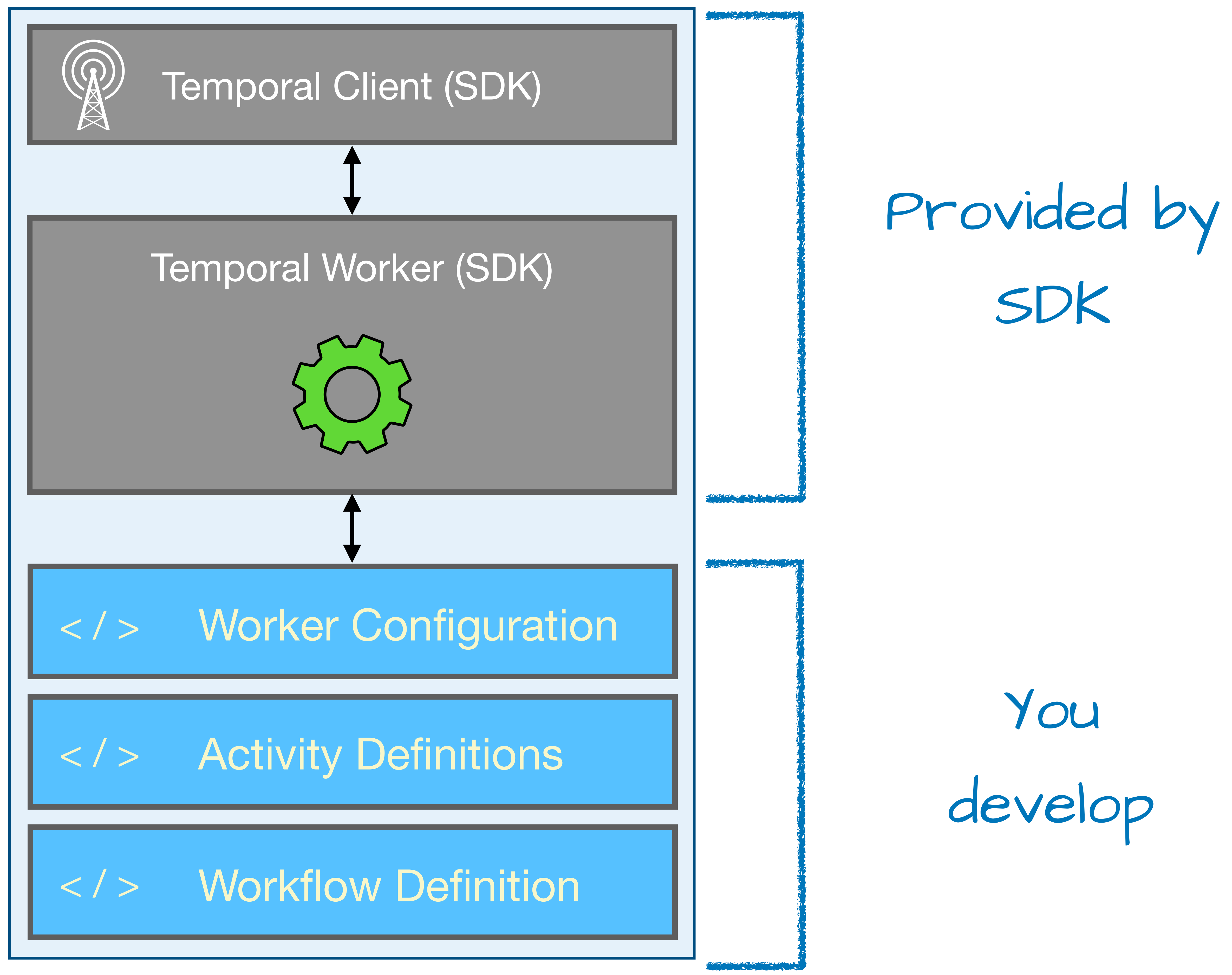
- **Workers are responsible for executing Workflow and Activity Definitions**
  - They poll a Task Queue maintained by the Temporal Service
- **The Worker implementation is provided by the Temporal SDK**
  - Your application will configure and start the Workers


< / > Worker Configuration
< / > Activity Definitions
< / > Workflow Definition

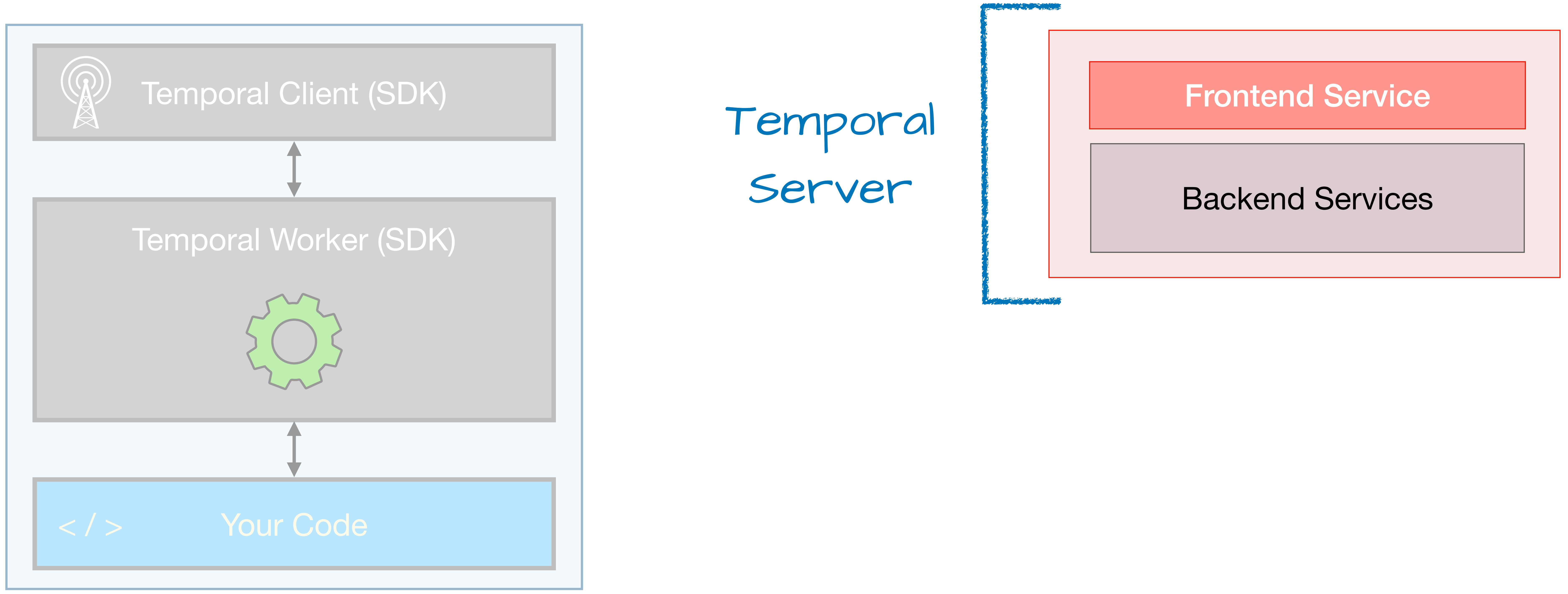
# Code You Develop



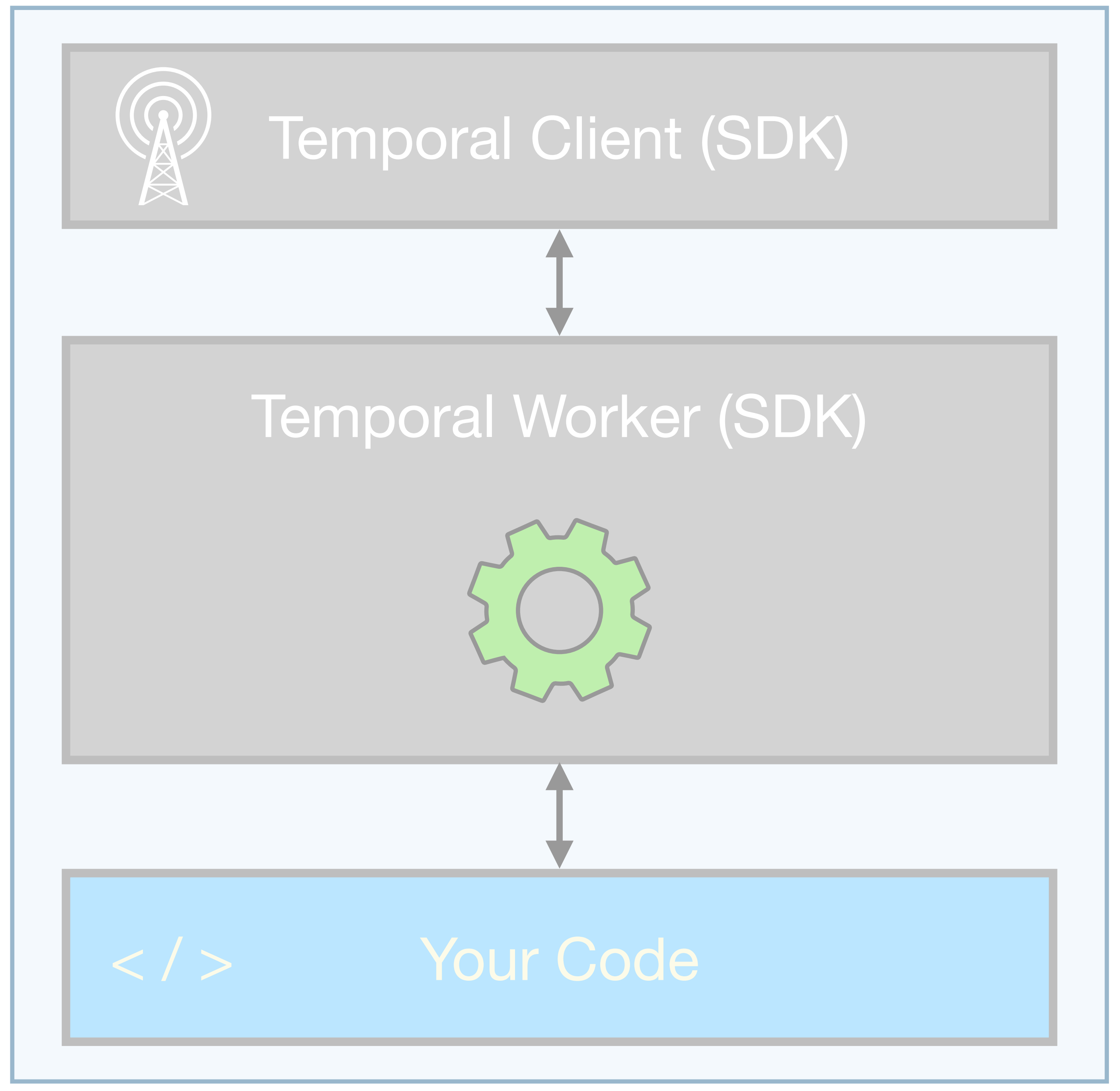
# A Complete Temporal Application

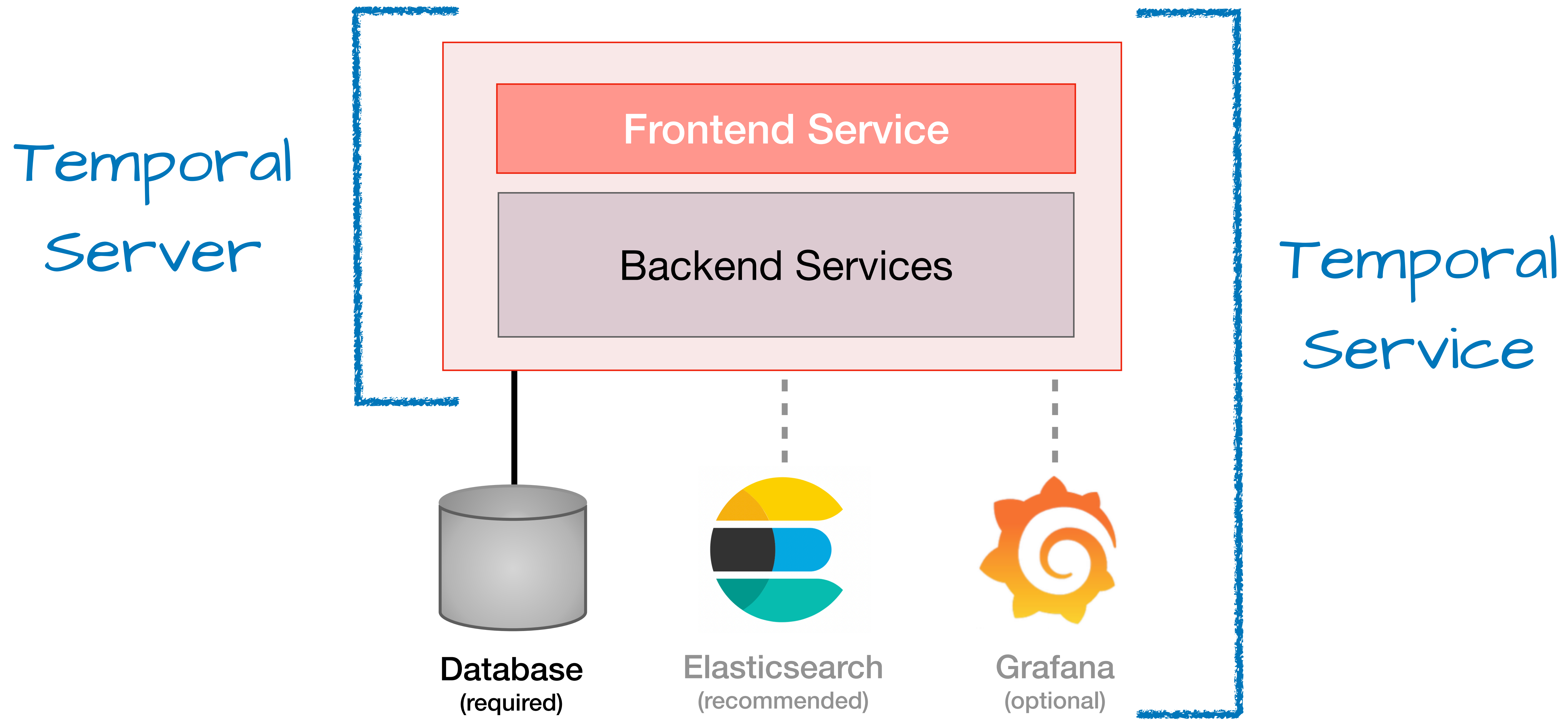


# The Role of a Local Temporal Service

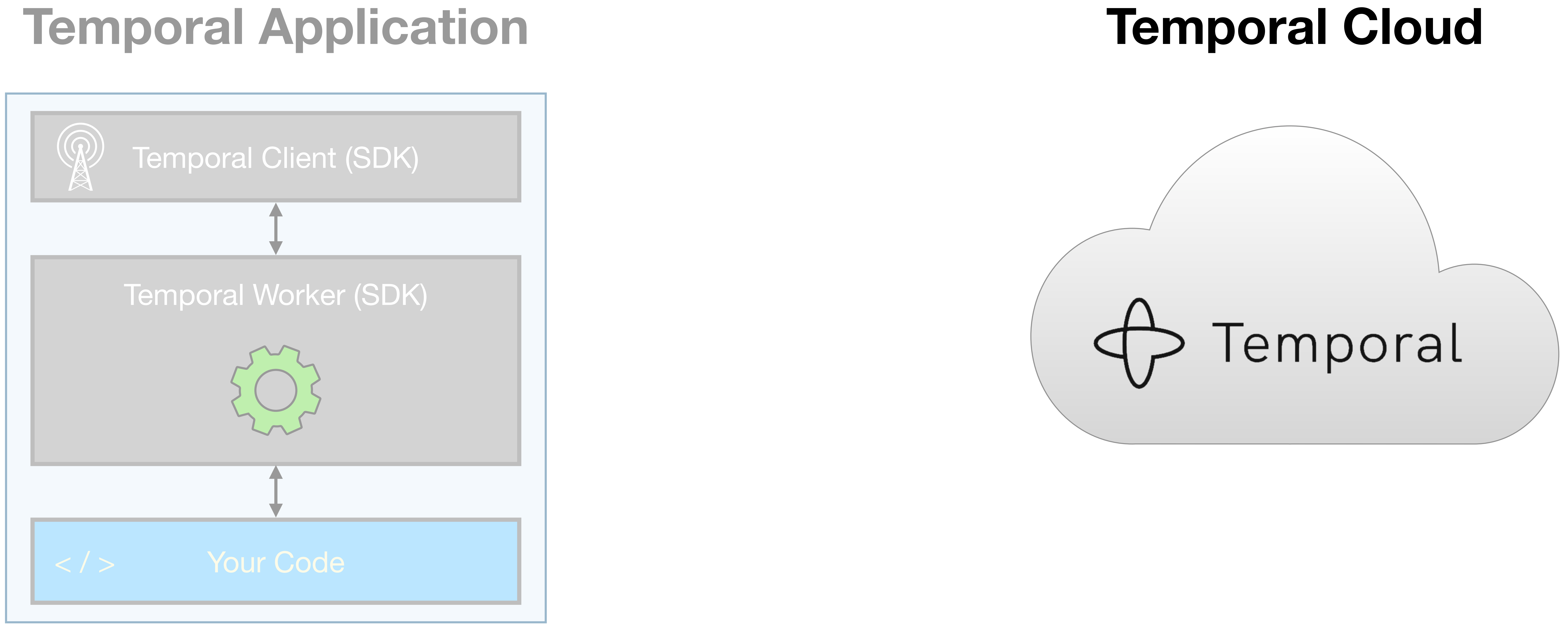


# The Role of a Local Temporal Service

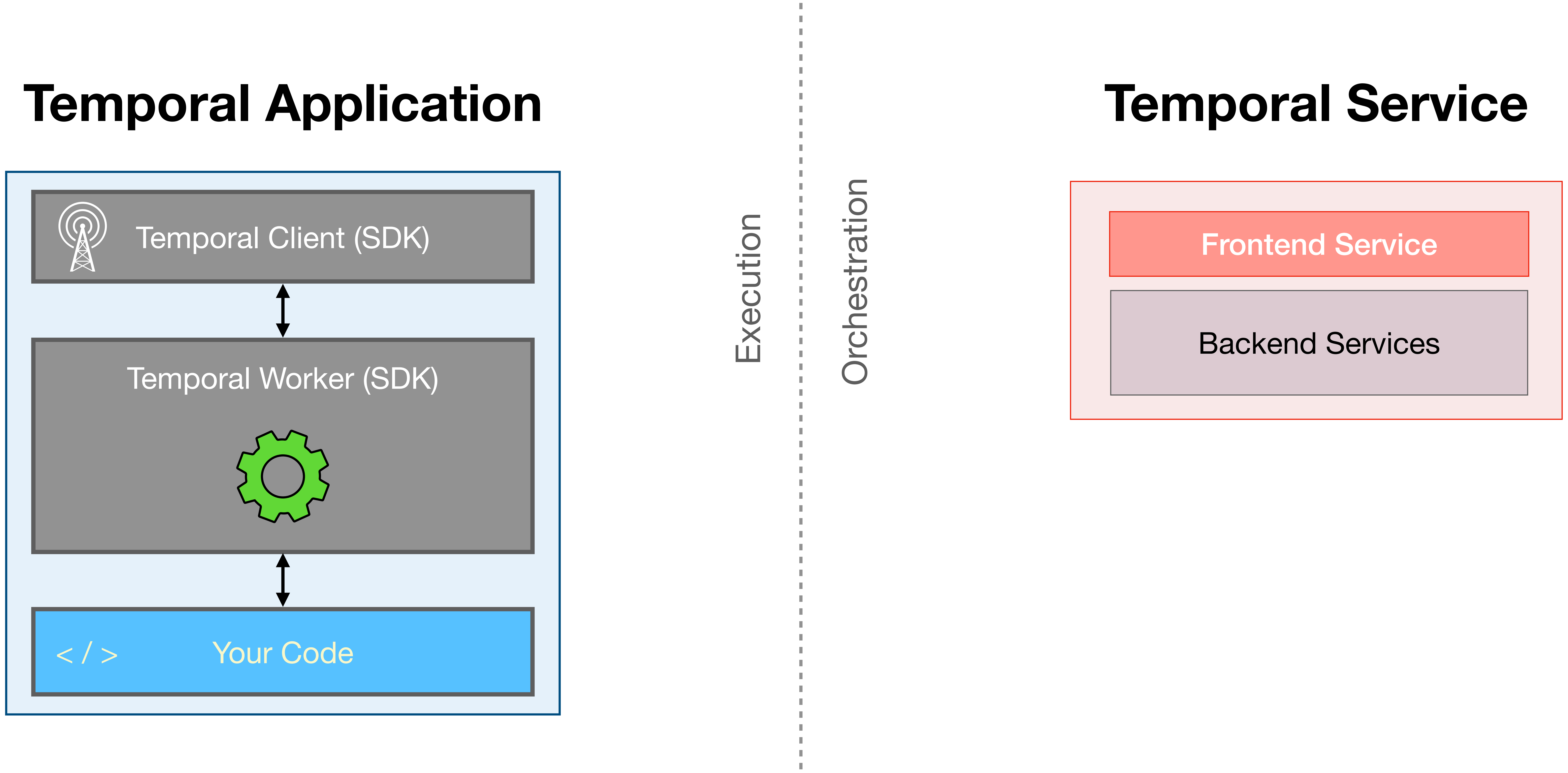




# The Role of Temporal Cloud



# Applications Are External to the Service



## Review

- **Temporal is a Durable Execution system**
  - Ensures that your application runs reliably despite adverse conditions
  - Automatically maintains application state and recovers from failure
- **Workflows represent the sequence of steps used to carry out your business logic. They must be deterministic**
- **Activities encapsulate unreliable or non-deterministic code.**
- **Workers execute Workflow and Activity Definitions by polling a Task Queue**
- **Your Workers, Workflows, and Activities make up a Temporal Application and are separate from the Temporal Service**

## Temporal 102

# Compatible Evolution of Input Parameters

- **Workflows and Activities can take any number of parameters as input**
  - Changing the number, position, or type of these parameters can affect backwards compatibility
- **It is a best practice to pass all input in a single serializable object, such as a `record`**
  - Changes to the composition of this class does not affect the method signature
  - Records can be directly serialized
  - In .NET, any new fields added to records must have default values
- **This is also the recommended approach for return values**

## Example: Using a `record` in an Activity (1)

- **Imagine that you have the following Activity**

```
public async Task<string> GetSpanishGreetingAsync(string name)
{
    // implementation omitted for brevity
}
```

output

input

- **You later need to update it to support other languages, such as Spanish**
  - Changing what is passed into or returned from the method changes its signature

# Example: Using a `record` in an Activity (2)

- **The following code sample illustrates how you could support this**

```
public record TranslationActivityInput(string Term, string LanguageCode);

public record TranslationActivityOutput(string Translation);

public async Task<TranslationActivityOutput> TranslateTermAsync(TranslationActivityInput input) {
    // implementation omitted
```

input
output

## Task Queues

- **Temporal Services coordinate with Workers through named Task Queues**
  - The name of this Task Queue is specified in the Worker configuration
  - The Task Queue name is also specified by a Client when starting a Workflow
  - Task Queues are dynamically created, so a mismatch in names does not result in an error
- **Recommendations for naming Task Queues**
  - Do not hardcode the name in multiple places: Use a shared constant if possible
  - Avoid mixed case: Task Queue names are case sensitive
  - Use descriptive names, but make them as short and simple as practical
- **Plan to run *at least* two Worker Processes per Task Queue**

## Workflow IDs

- **You specify a Workflow ID when starting a Workflow Execution**
    - This should be a value that is meaningful to your business logic

# Workflow IDs

- **You specify a Workflow ID when starting a Workflow Execution**
  - This should be a value that is meaningful to your business logic

```
// Example: An order processing Workflow might include order number in the Workflow ID
var options = new WorkflowOptions(
            id: $"process-order-number-{input.OrderNumber}",
            taskQueue: WorkflowConstants.TaskQueueName);

// Run workflow
var result = await client.ExecuteWorkflowAsync(
    (OrderProcessingWorkflow wf) => wf.RunAsync(input),
    options);
```

- **Must be unique among all *running* Workflow Executions in the namespace**

# Logging in Temporal Applications

- **The recommended way of logging in Workflows and Activities is with .NET's standard logging infrastructure using loggers that Temporal Provides**
  - Is replay aware
  - **Note:** You will not get any tracebacks from the .NET SDK without instantiating a logger

# Using the Logger

- **Set the `LoggerFactory` in the Client**

```
var client = await TemporalClient.ConnectAsync(new("localhost:7233")
{
    LoggerFactory = LoggerFactory.Create(builder =>
        builder.AddSimpleConsole(options => options.TimestampFormat =
"[HH:mm:ss] ").SetMinimumLevel(LogLevel.Information)),
});
```

- **Access and use the Workflow logger using `Workflow.Logger`**

```
var logger = Workflow.Logger;

logger.LogInformation("Preparing to execute an Activity");
logger.LogError("Preparing to charge customer {Name} for {Cost}", input.Name, input.Cost);
```

## Using the Logger

- **Access and use the Activity logger using `ActivityExecutionContext.Current.Logger`**

```
var logger = ActivityExecutionContext.Current.Logger;

logger.LogInformation("Translating term {Term} to {LangCode}", input.Term, input.LangCode);
```

## Long-Running Executions

- **Temporal Workflows may have executions that span several years**
  - Activities may also run for long periods of time
- **Workflow and Activity Executions can be synchronous or asynchronous**
  - Synchronous calls will stop progress, waiting on the result of the Workflow or Activity before continuing
  - Asynchronous calls will not stop progress and the result will have to be retrieved at a later time
- **Workflows run until all Tasks yield, and resume when there are new events.**

```
var client = await TemporalClient.ConnectAsync(new("localhost:7233"));
```

# Workflow Execution

- **Returns handle**

```
var workflowHandle = await client.StartWorkflowAsync(
    (GreetingWorkflow wf) => wf.RunAsync(),
    options);

//...

await handle.GetResultAsync();
```

- **Starts Workflow + gets result**

```
var result = await client.ExecuteWorkflowAsync(
    (TestWorkflow wf) => wf.RunAsync(input),
    options);
```

## Deferring Access to Execution Results

- **Deferring access to results *may* reduce overall execution time**
  - This is a good strategy when a Workflow needs to call unrelated Activities

```
var taskA = Workflow.ExecuteActivityAsync(
    (MyActivities act) => act.ActivityAAsync(inputA),
    activityOptions);

var taskB = Workflow.ExecuteActivityAsync(
    (MyActivities act) => act.ActivityBAsync(inputB),
    activityOptions);

var taskC = Workflow.ExecuteActivityAsync(
    (MyActivities act) => act.ActivityCAsync(inputC),
    activityOptions);

// Wait for all results at once
var results = await Workflow.WhenAllAsync(taskA, taskB, taskC);

// Or wait for results individually
var resultA = await taskA;
var resultB = await taskB;
var resultC = await taskC;
```

## Temporal 102

## What is a Timer?

- **Timers are used to introduce delays into a Workflow Execution**
  - Code that awaits the Timer pauses execution for the specified duration
  - The duration is fixed and may range from seconds to years
  - Once the time has elapsed, the Timer fires, and execution continues

## Use Cases for Timers

- Execute an Activity multiple times at predefined intervals
- Execute an Activity multiple times at dynamically-calculated intervals
- Allow time for offline steps to complete

## Timer APIs Provided by the .NET SDK

- **.NET SDK offers a Workflow-safe, replay-aware ways to start a Timer**
  - A Workflow-safe replacement for `Task.Delay`
  - Workflow code must not use .NET's methods for timers (non-deterministic)

## Pausing Workflow Execution for a Specified Duration

- **You can pause execution for a set amount of time using `Workflow.DelayAsync()`**
  - It blocks until the Timer is fired (or is canceled)

```
await Workflow.DelayAsync(TimeSpan.FromDays(3));
```

## What Happens to a Timer if the Worker Crashes?

- **Timers are maintained by the Temporal Service**
  - Once set, they fire regardless of whether any Workers are running
- **Scenario: Timer set for 10 seconds and Worker crashes 3 seconds later**
  - If Worker is restarted within 7 seconds, it will be running when the Timer fires
  - If Worker is restarted 5 *minutes* later, the Timer will have already fired
    - In this case, the Worker will resume executing the Workflow code without delay

## Exercise #1: Observing Durable Execution

- **During this exercise, you will**
  - Create Workflow and Activity loggers
  - Add logging statements to the code
  - Add a Timer to the Workflow Definition
  - Launch two Workers, run the Workflow, and kill one of the Workers, observing that the remaining Worker completes the execution
- **Refer to this exercise's `README.md` file for details**
  - Don't forget to make your changes in the `practice` subdirectory

t.mp/edu-102-dotnet-code

## Exercise #1: Observing Durable Execution SOLUTION

## Review

- Timers introduce delays into a Workflow Execution
- Timers are durable, meaning they can survive a crash of the Worker who invoked it
- Timers are maintained by the Temporal Service and recorded in the Event History
- Example Timer Use Cases:
  - Execute an Activity multiple times at predefined or calculated intervals
  - Allow time for offline steps to occur

# Temporal 102

**Workflow Definition**

```
[Workflow]
public class SayHelloWorkflow
{
    [WorkflowRun]
    public async Task<string> RunAsync(string name)
    {
        string greeting = $"Hello, {name}!";
        return greeting;
    }
}
```

**combined with**

+ +

**n Execution Requests**

```
var result = await client.ExecuteWorkflowAsync(
    (SayHelloWorkflow wf) => wf.RunAsync("Angela"),
    new(id: "my-workflow-id",
    taskQueue: "greeting-tasks"));
```

```
var result = await client.ExecuteWorkflowAsync(
    (SayHelloWorkflow wf) => wf.RunAsync(“Chad"),
    new(id: "my-workflow-id",
    taskQueue: "greeting-tasks"));
```

**results in**

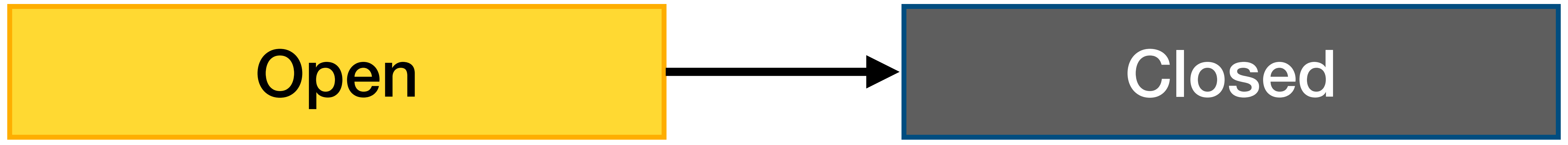
= =

**n Workflow Executions**

Workflow Execution 1

Workflow Execution 2

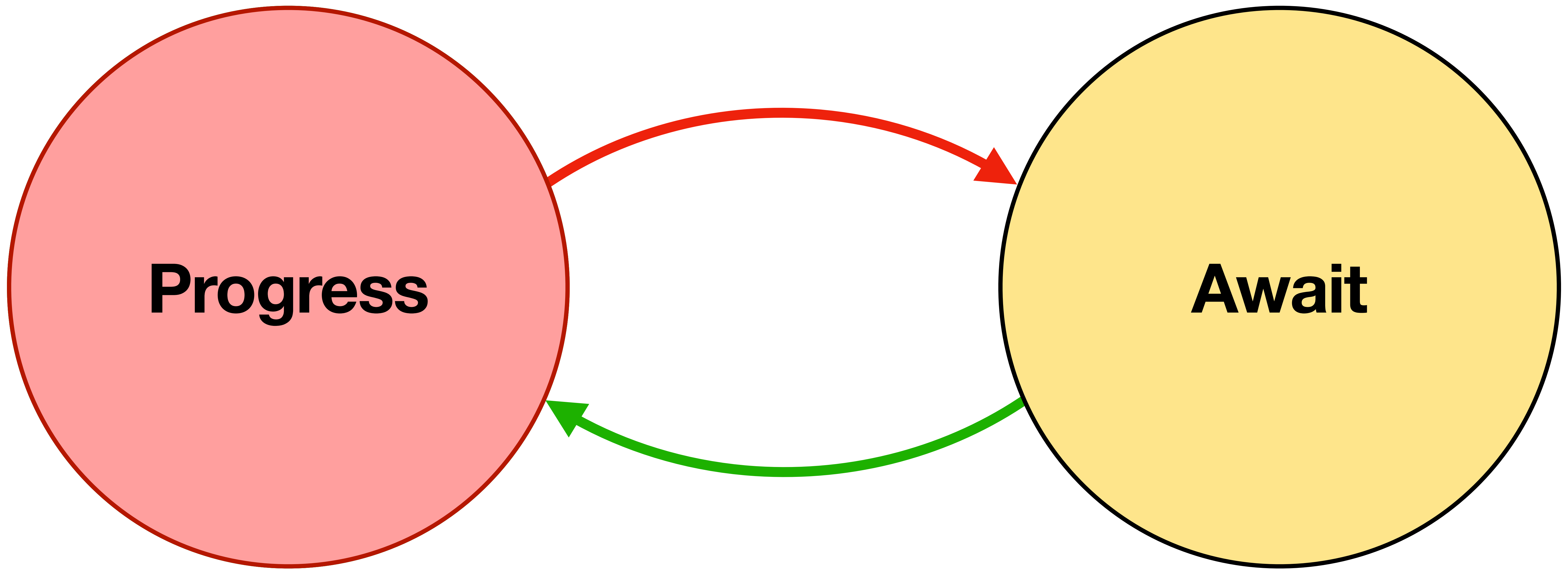
## Workflow Execution States



**This is a one-way transition**

**Each Workflow Execution has a unique Run ID**

## What Happens During Workflow Execution

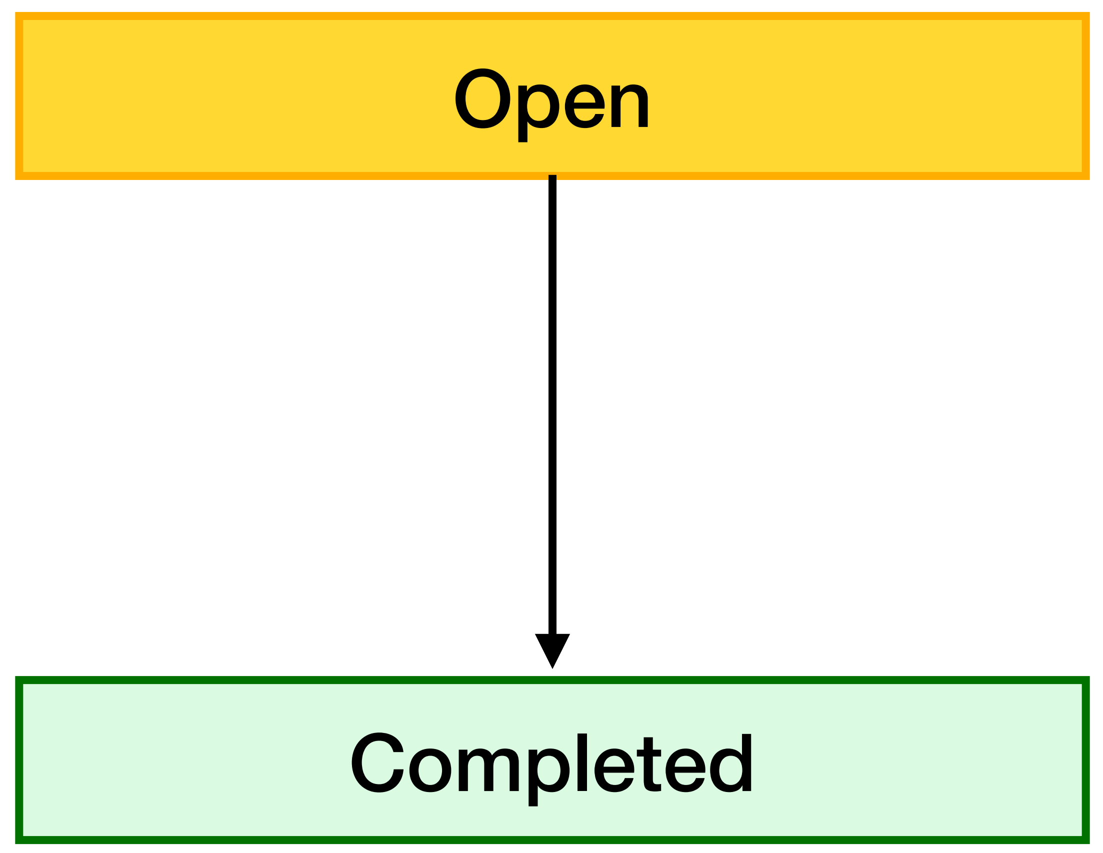


This cycle continues throughout Workflow Execution

## Workflow Execution States

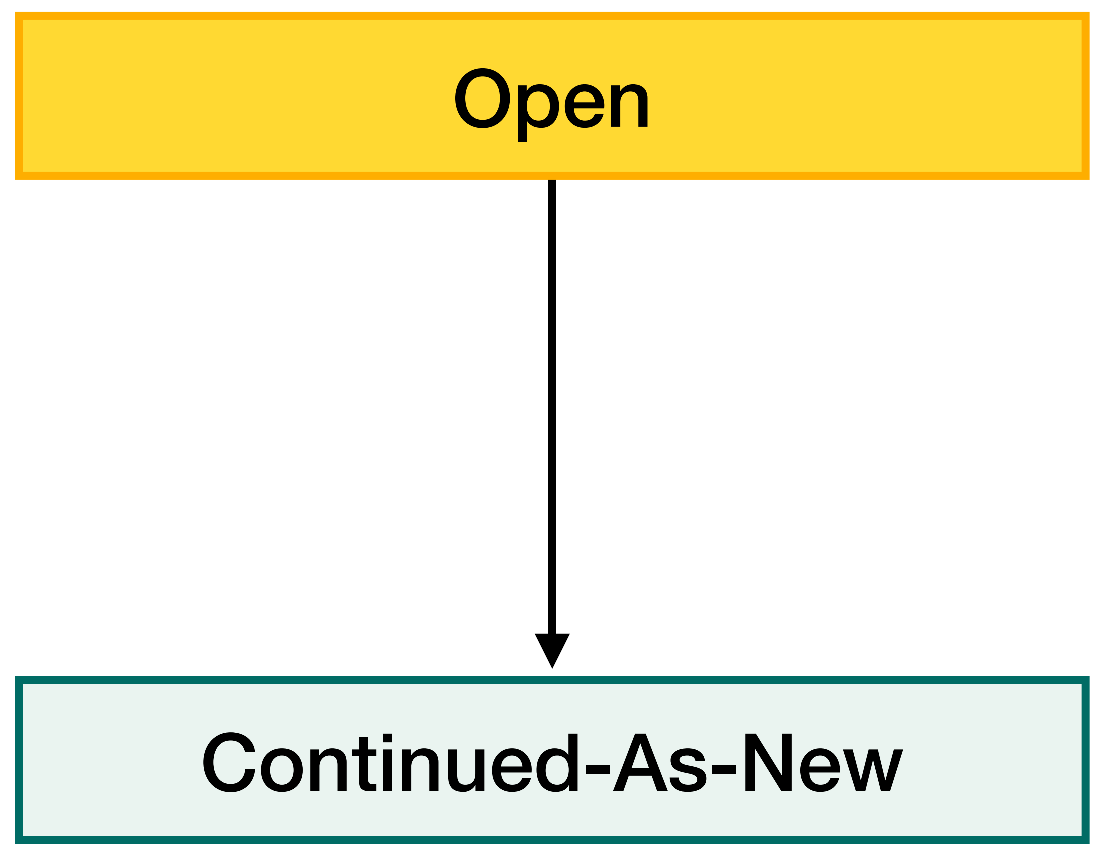
## Completed

**Meaning: The Workflow method returned a result**



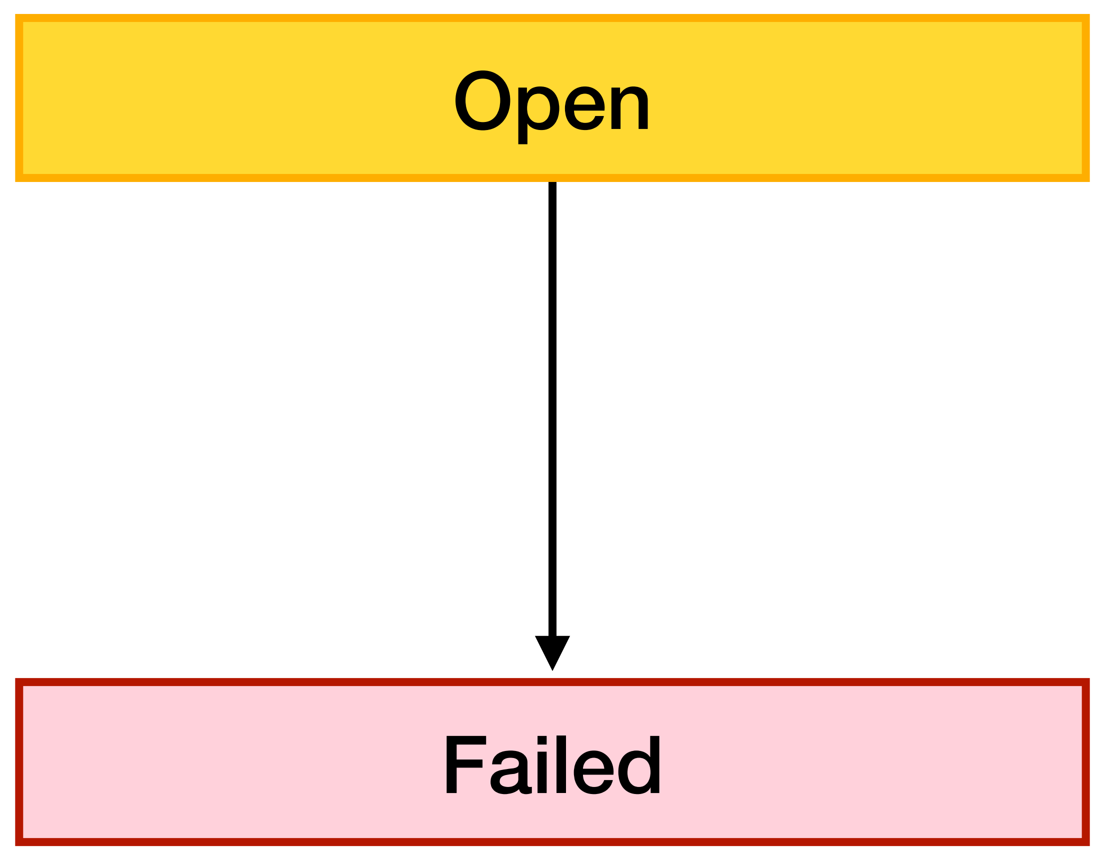
## Continued-As-New

**Meaning: Future progress will take place in a new Workflow Execution**



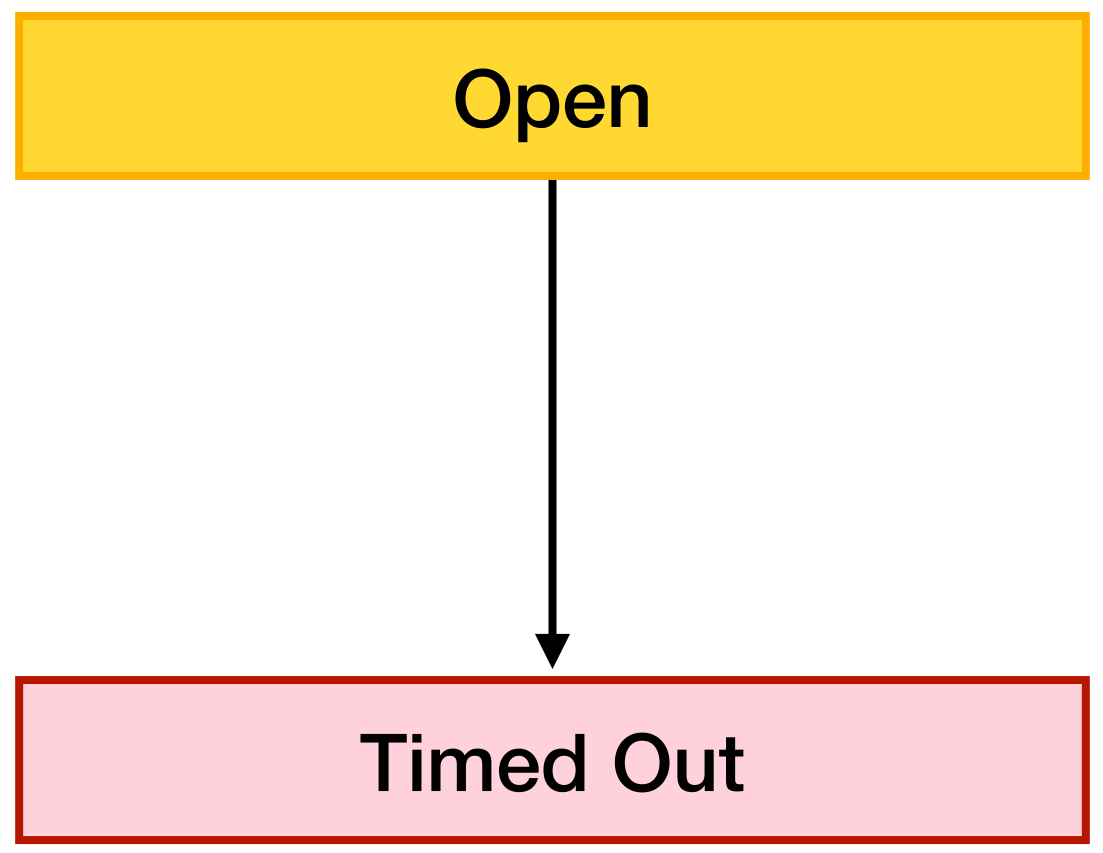
## Failed

**Meaning: The Workflow method raised an exception**



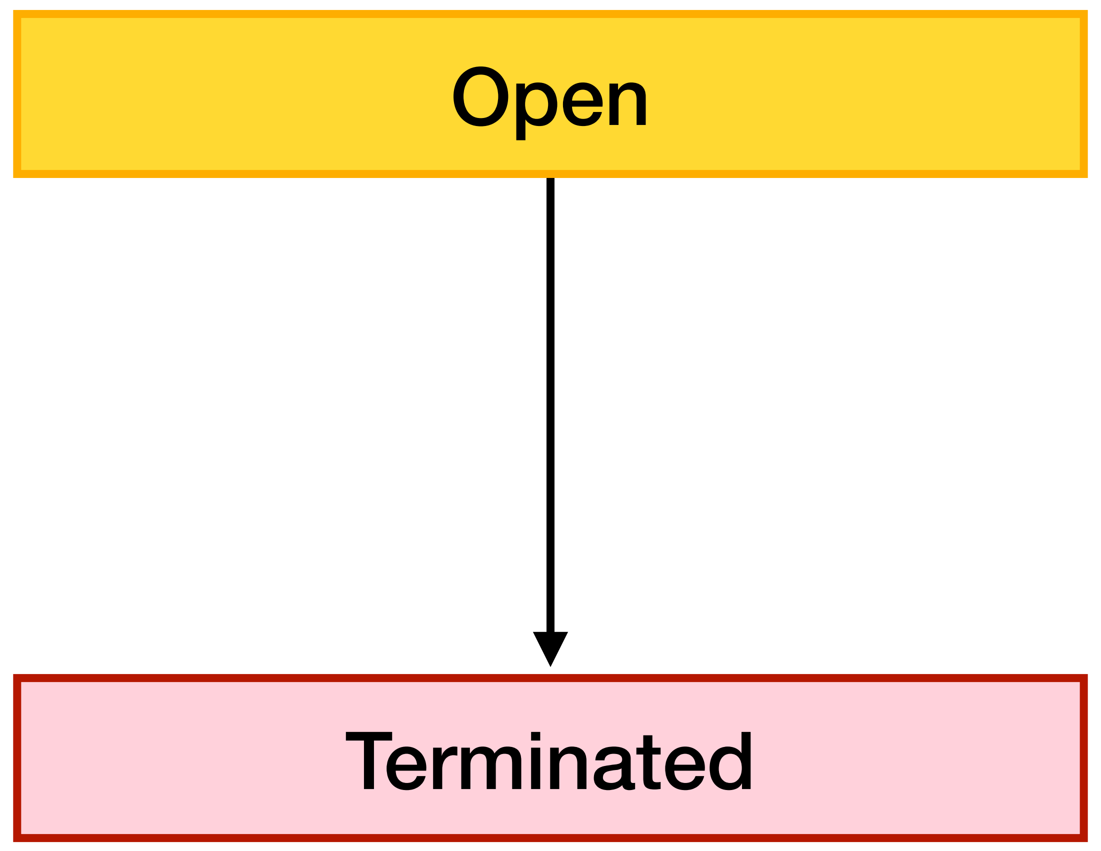
## Timed Out

**Meaning: Execution exceeded a specified time limit**



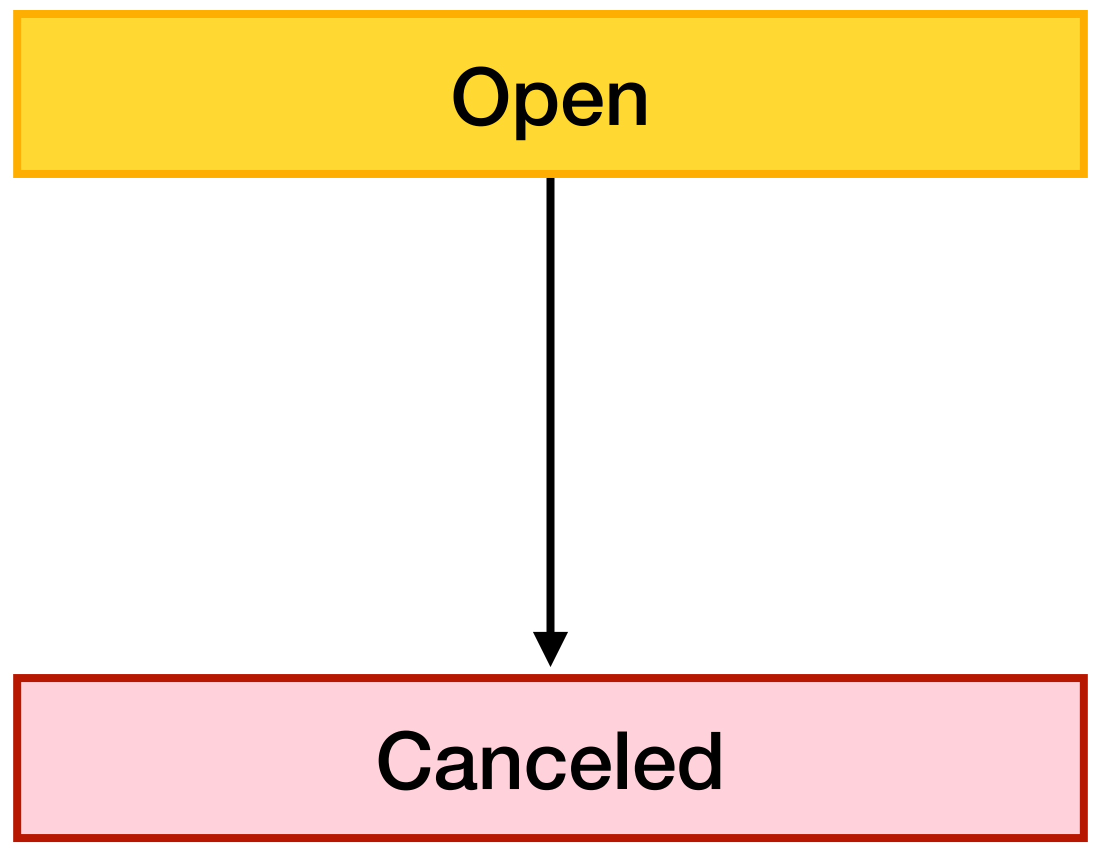
## Terminated

**Meaning: Temporal Service acted upon a termination request**

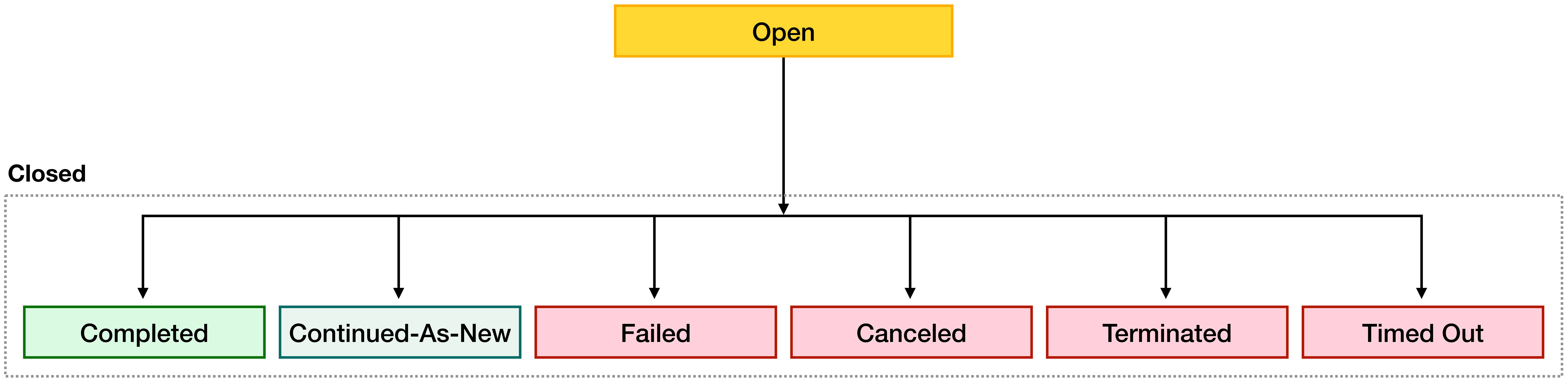


## Canceled

**Meaning: Temporal Service acted upon a request to cancel execution**

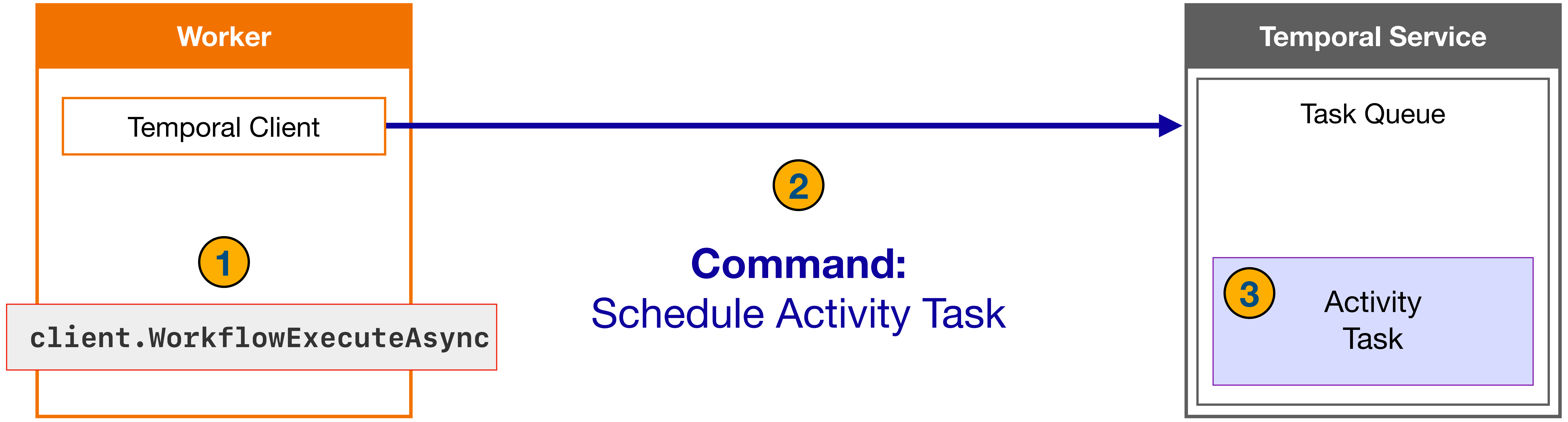


# Summary of Workflow Execution States



## How Workflow Code Maps to Commands

# Commands



- Certain API calls result in the Worker issuing a Command to the Temporal Service
- The Service acts on these Commands, but also stores them
- This allows the Service to recreate the state of a Workflow Execution following a crash

```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

## Basic Temporal Workflow Definition

- A Workflow is a sequence of steps
- Some steps are *internal to the Workflow*
  - Do not involve interaction with the Service
- Examples include
  - Performing calculations
  - Evaluating variables or expressions
  - Populating data structures
- These internal steps are highlighted in white

```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

## Basic Temporal Workflow Definition

- Other steps *do* involve interaction with the Service
- Examples include
  - Executing an Activity
  - Setting a Timer
  - Throwing an exception from the Workflow
  - Returning a value from the Workflow
- These external steps are highlighted in yellow

```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

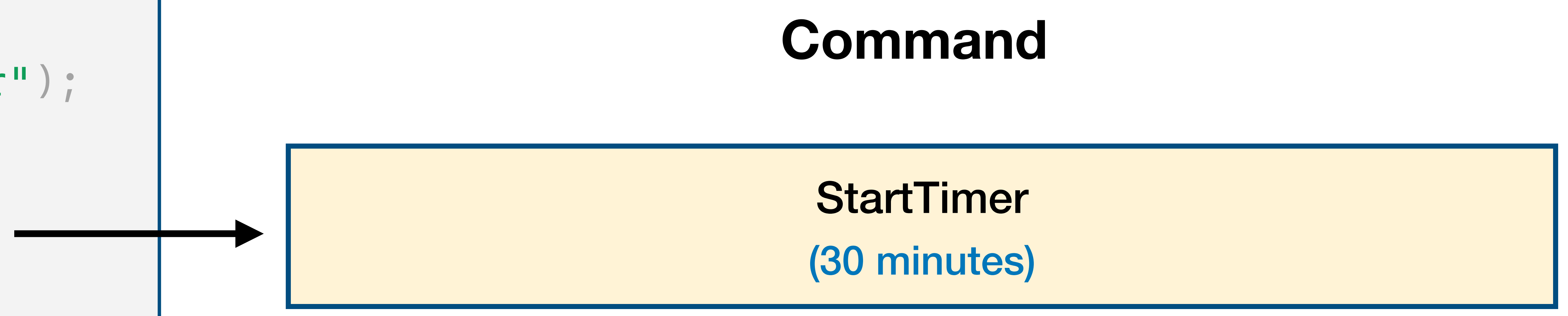
**Command**

ScheduleActivityTask

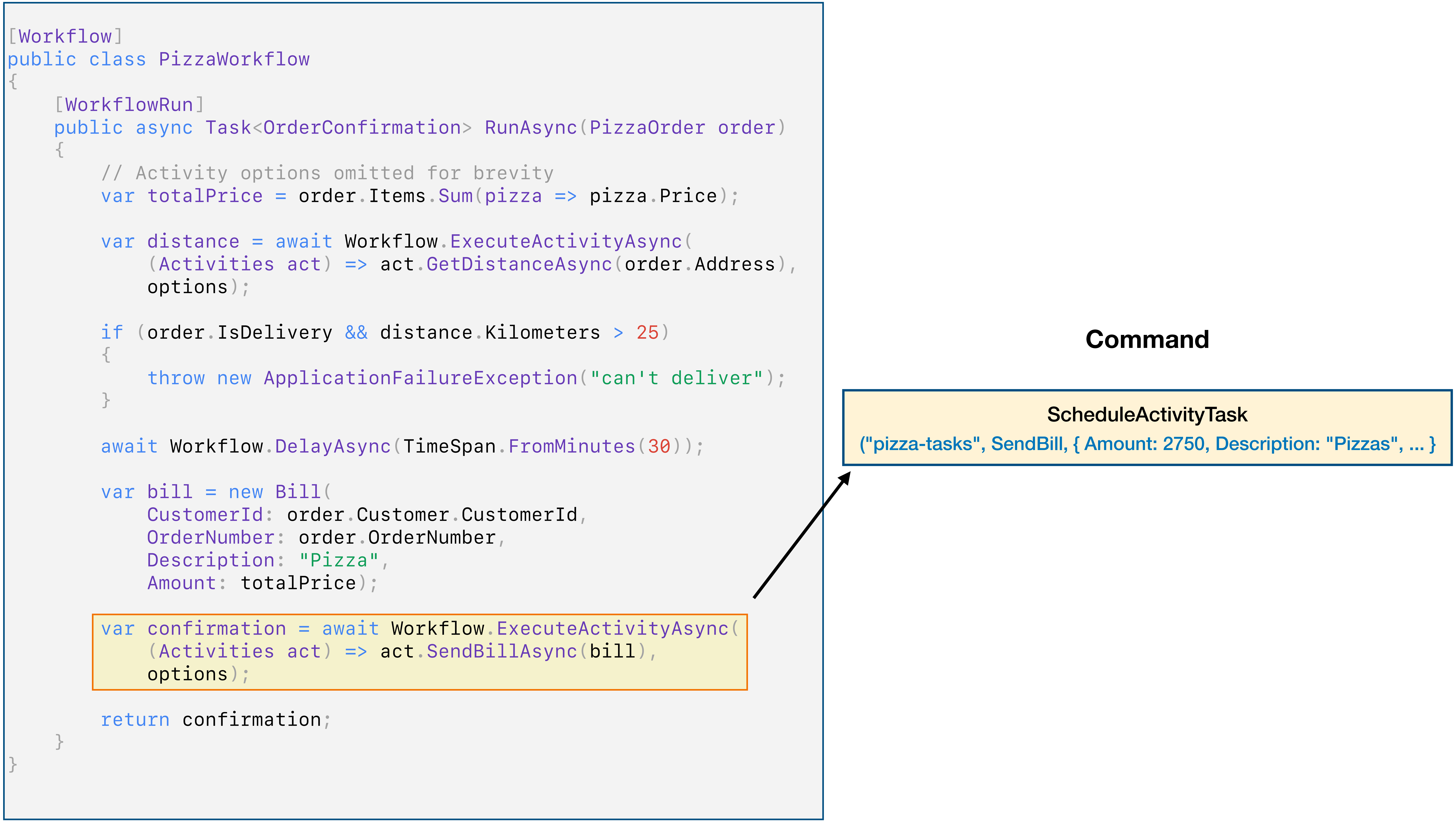
("pizza-tasks", GetDistance, { Line1: "123 Oak St.", Line2: "", ... })

```csharp
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```



```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

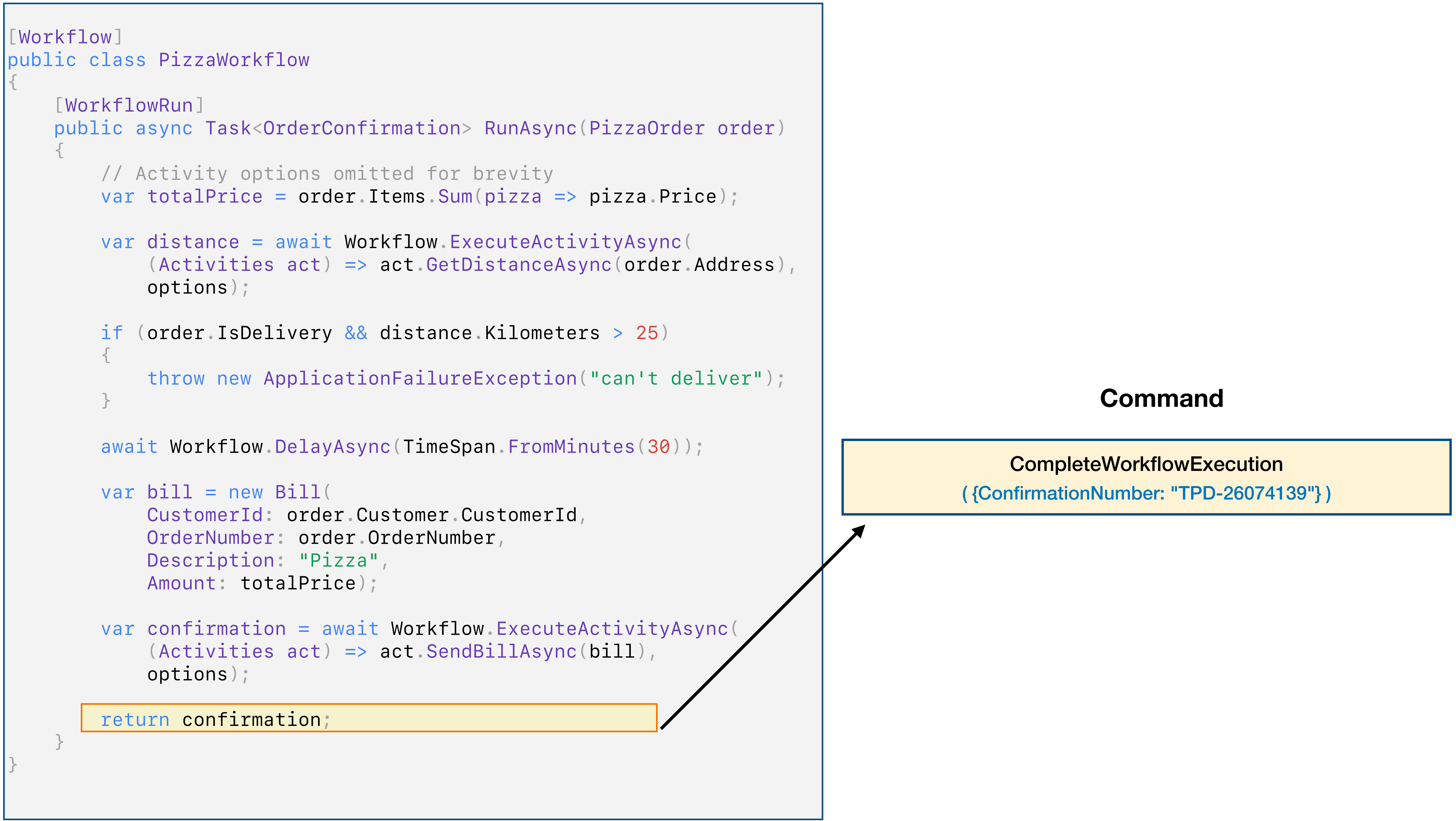


```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

**Command**

ScheduleActivityTask

("pizza-tasks", SendBill, { Amount: 2750, Description: "Pizzas", ... }



```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

Command
CompleteWorkflowExecution
( {ConfirmationNumber: "TPD-26074139"} )

## Workflow Execution Event History

- **Each Workflow Execution is associated with an Event History**
- **Represents the source of truth for what transpired during execution**
  - As viewed from the Temporal Service's perspective
  - Durably persisted by the Temporal Service
- **Event Histories serve two key purposes in Temporal**
  - Allow reconstruction of Workflow state following a crash
  - Enable developers to investigate both current and past executions
- **You can access them from code, command line, and Web UI**

## Event History Content

- **An Event History acts as an ordered append-only record of Events**
  - Begins with the `WorkflowExecutionStarted` Event
  - New Events are appended as Workflow Execution progresses
  - Ends when the Workflow Execution closes

## Event History Limits

- **Temporal places limits on a Workflow Execution's Event History**
- **Warnings begin after 10K (10,240) Events**
  - These say "history size exceeds warn limit" and will appear the Temporal Service logs
  - They identify the Workflow ID, Run ID, and namespace for the Workflow Execution
- **Workflow Execution will be *terminated* after exceeding additional limits**
  - If its Event History exceeds 50K (51,200) Events
  - If its Event History exceeds 50 MB of storage

## Event Structure and Characteristics

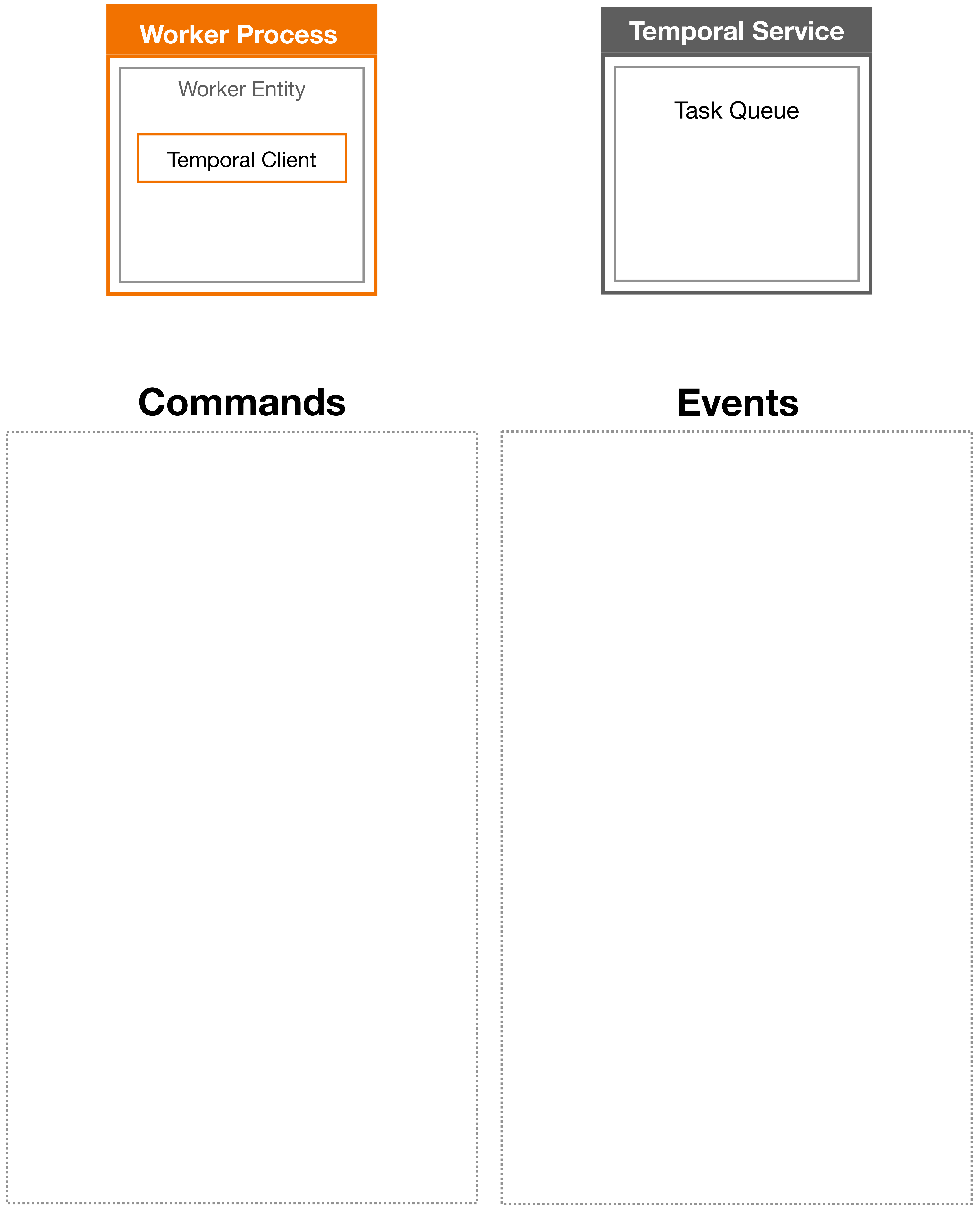
- **Every Event always contains the following three attributes**
  - ID (uniquely identifies this Event within the History)
  - Time (timestamp representing when the Event occurred)
  - Type (the kind of Event it is)

## Attributes Vary by Event Type

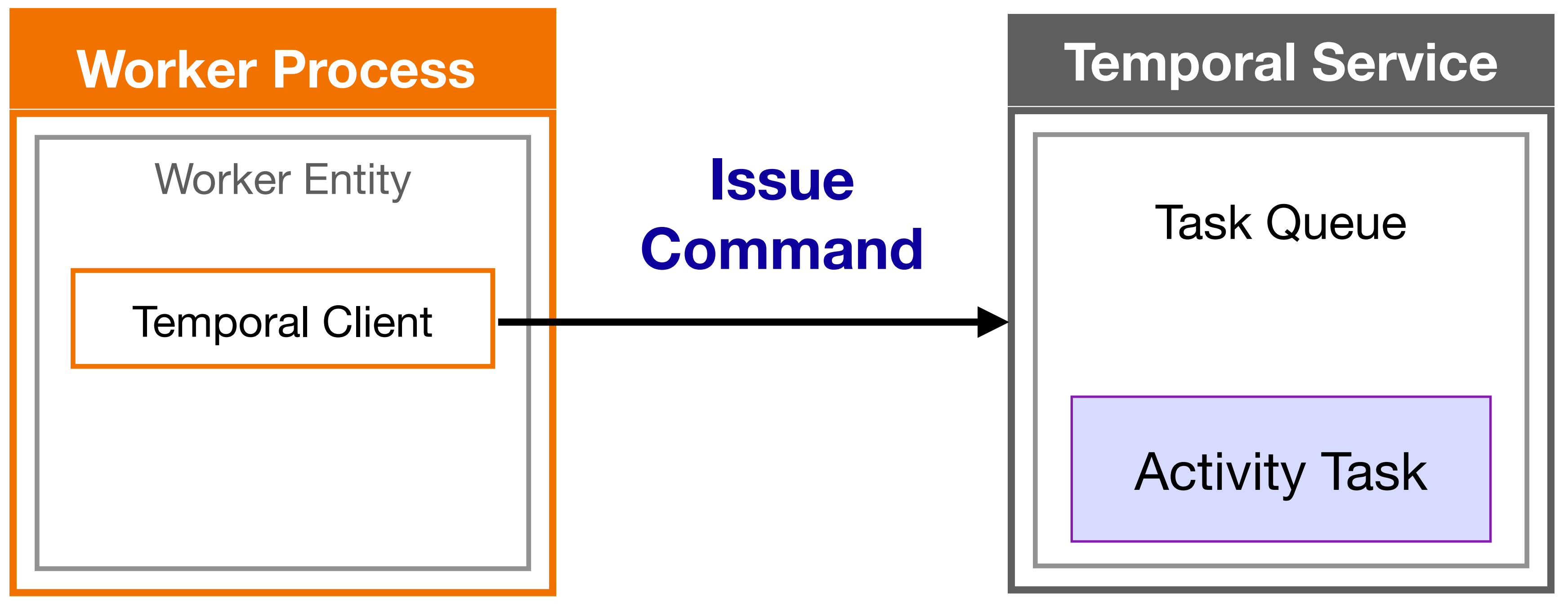
- **Additionally, each Event contains attributes specific to its type**
  - **`WorkflowExecutionStarted`** contains the Workflow Type and input parameters
  - **`WorkflowExecutionCompleted`** contains the result returned by the Workflow method
  - **`WorkflowExecutionFailed`** contains the exception thrown by the Workflow method
  - **`ActivityTaskScheduled`** contains the Activity Type and input parameters
  - **`ActivityTaskCompleted`** contains the result returned by the Activity method

## How Commands Map to Events

```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```



```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

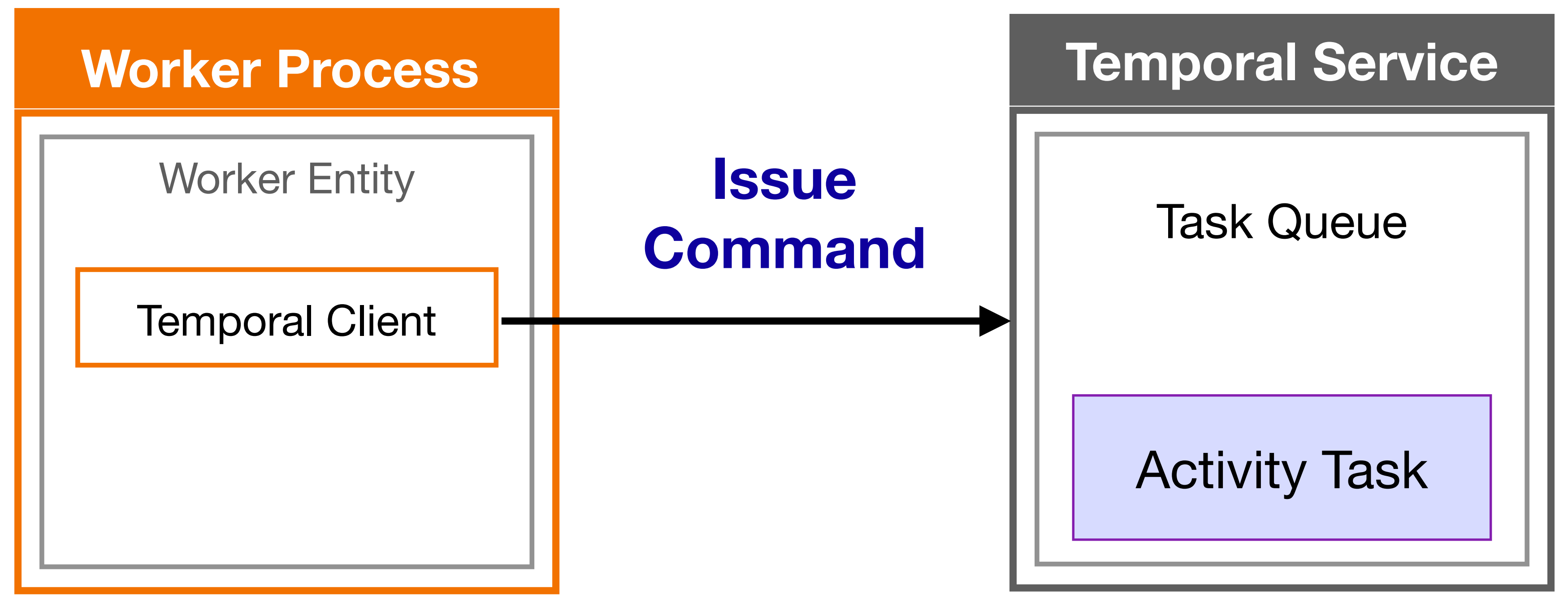


**Commands**

ScheduleActivityTask
(GetDistanceAsync)

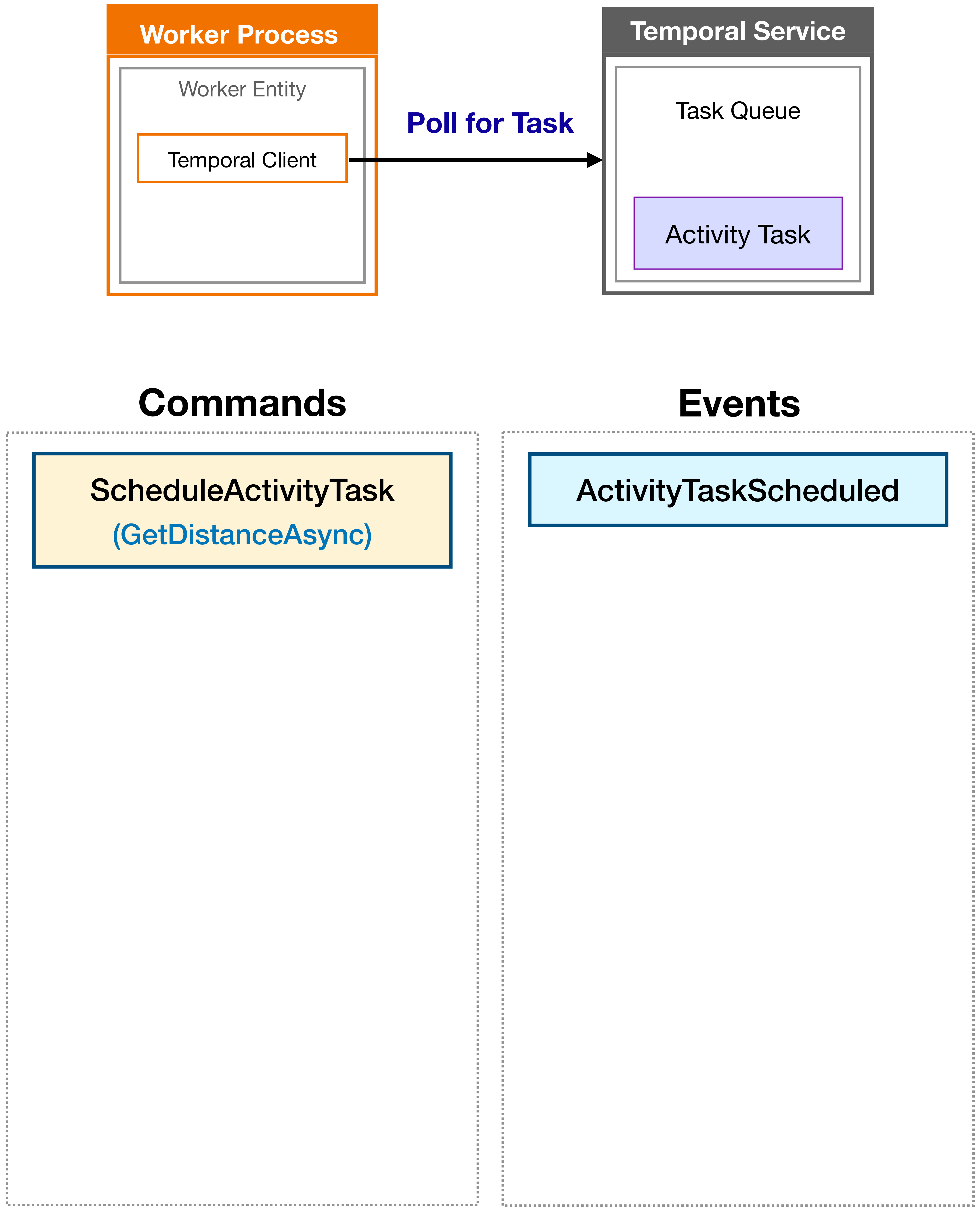
**Events**

```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

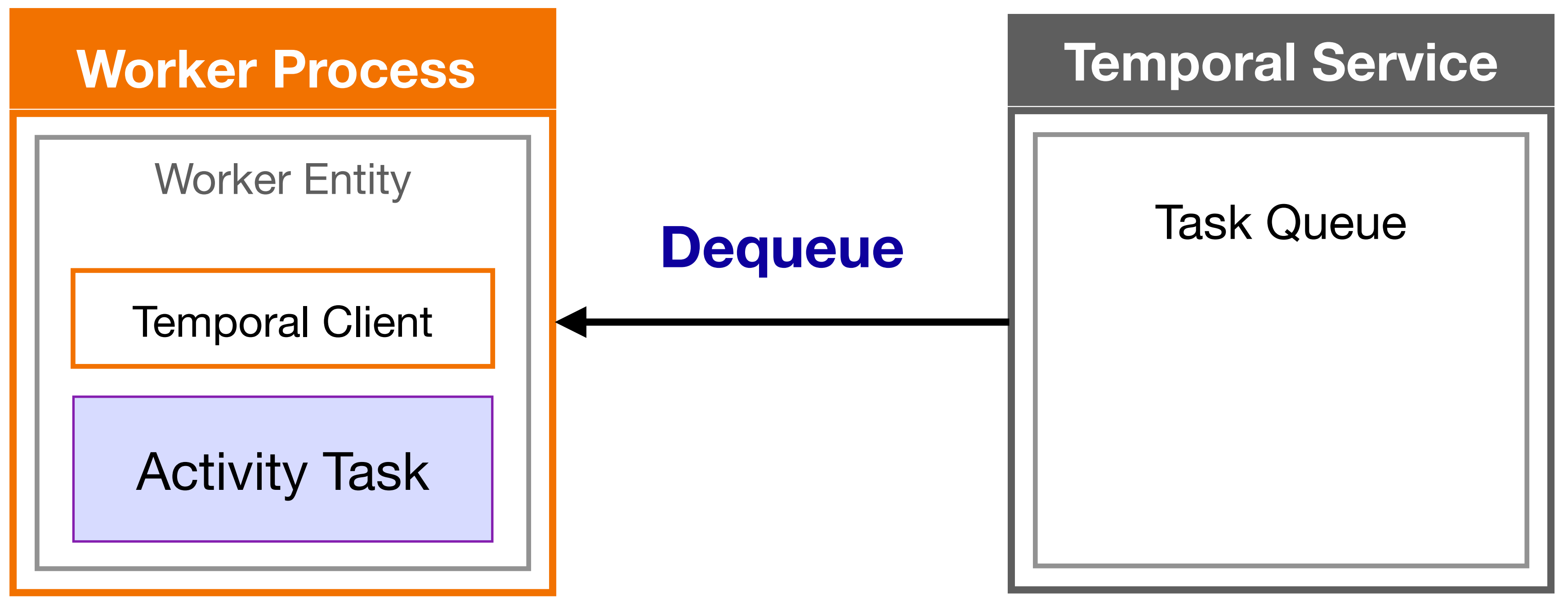


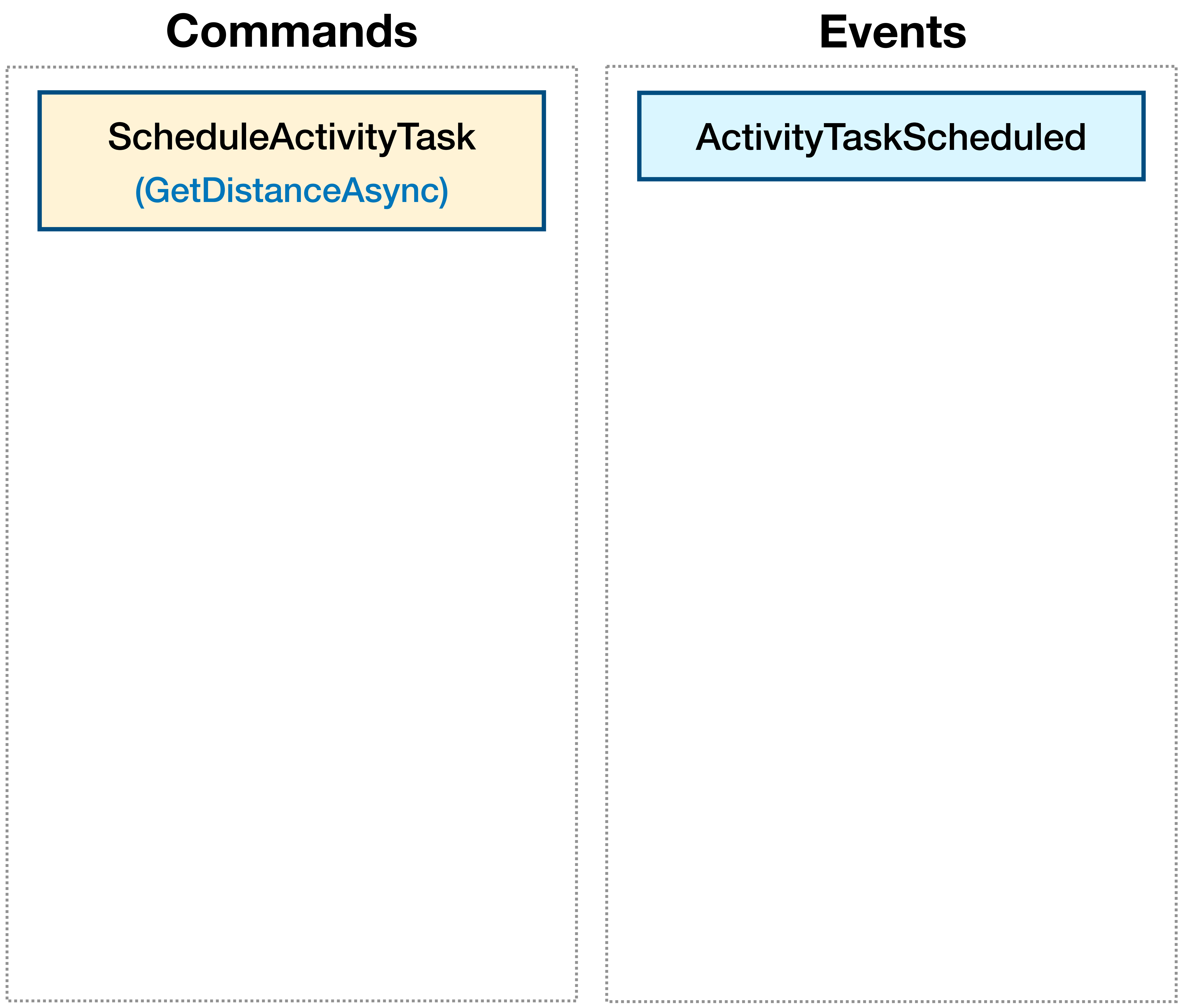
| Commands | Events |
|---|---|
| ScheduleActivityTask (GetDistanceAsync) | ActivityTaskScheduled |

```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```



```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

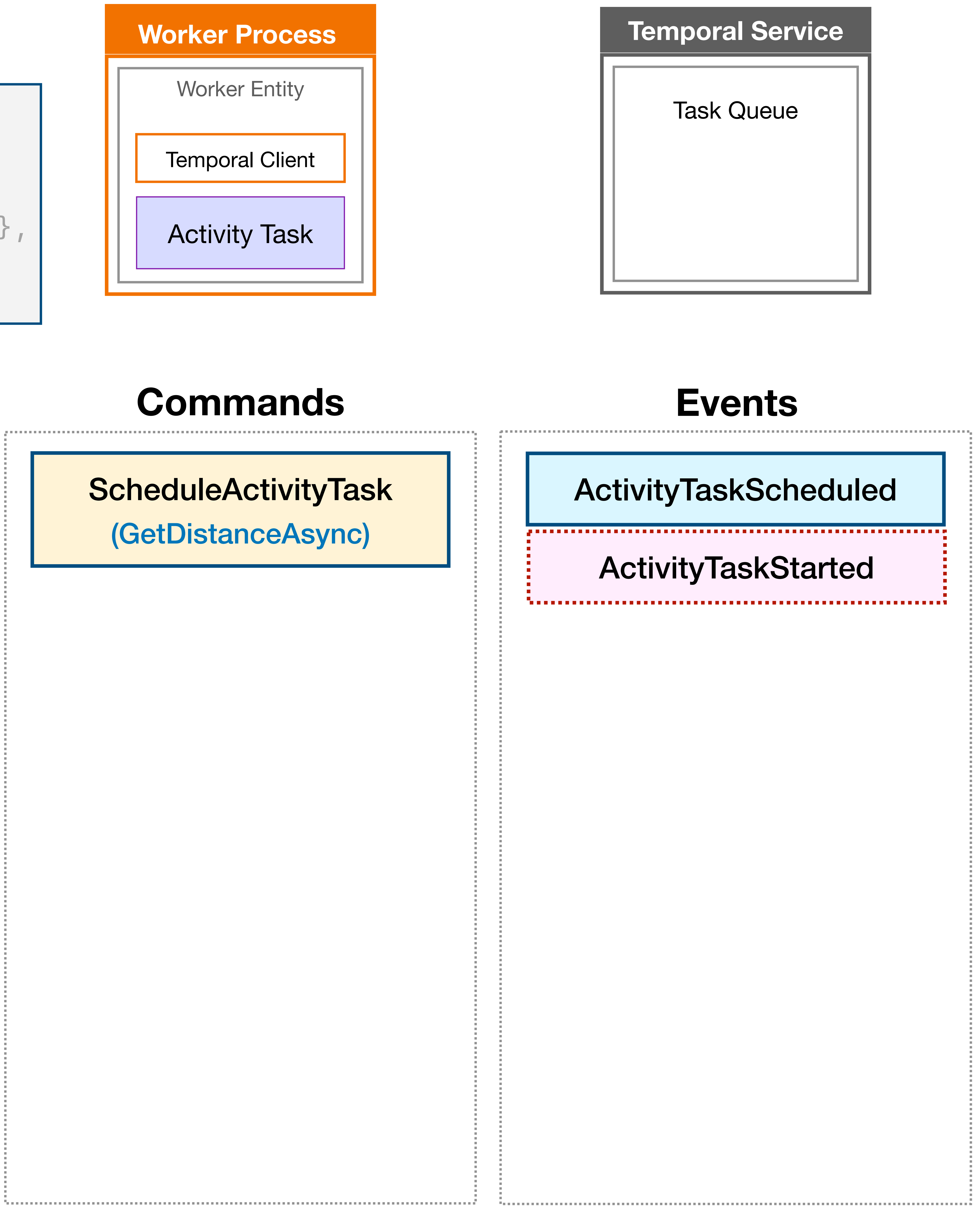




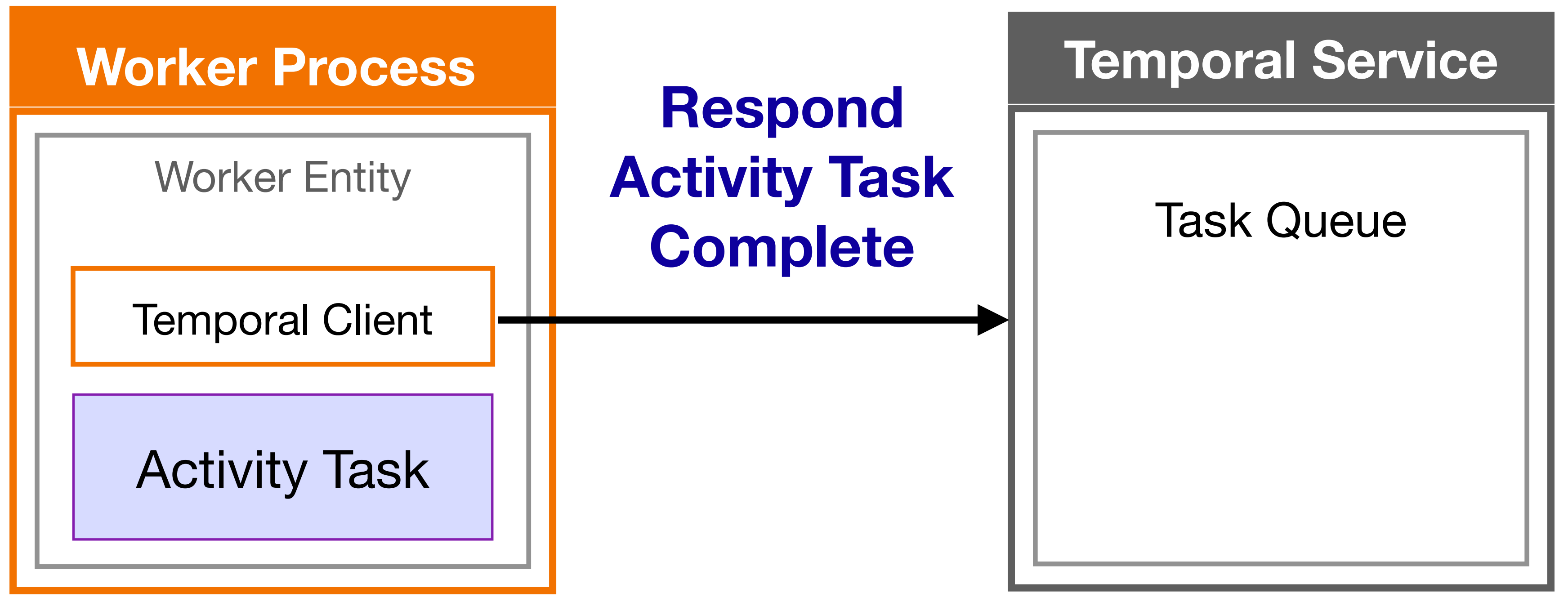
```
[Workflow]
```

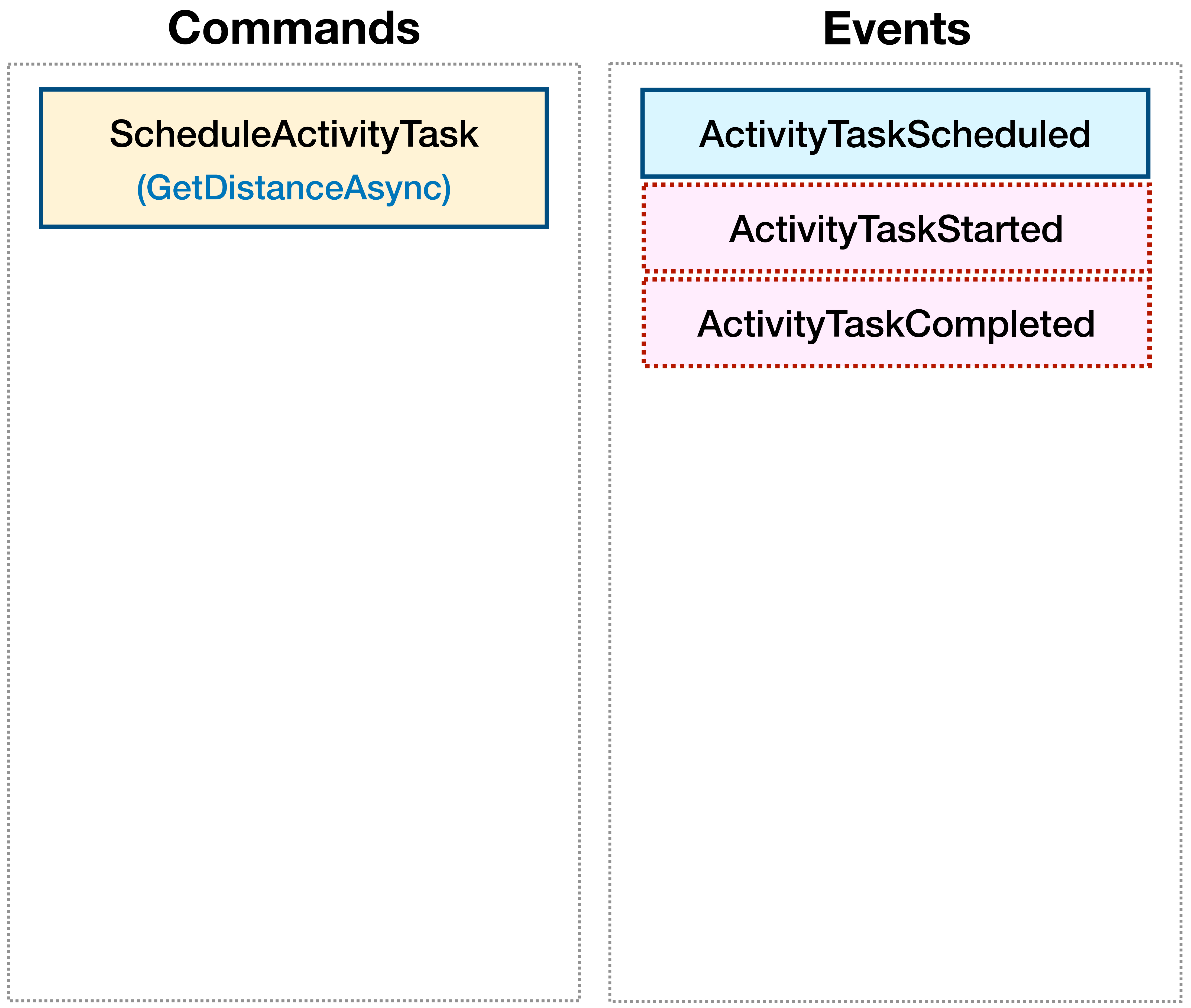
```
var options = new ActivityOptions
{
    StartToCloseTimeout = TimeSpan.FromSeconds(5),
    RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
};
```

```
        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

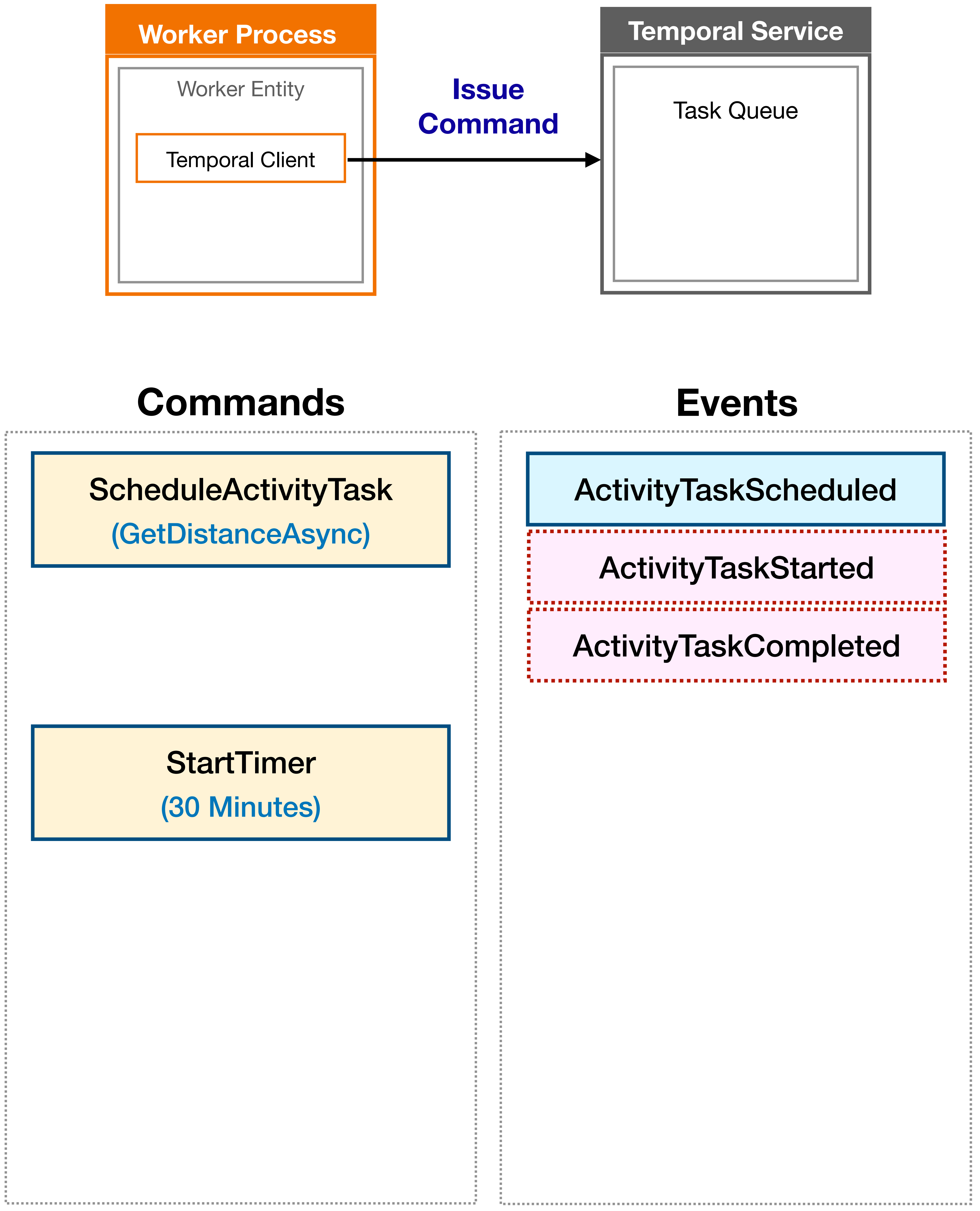


```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

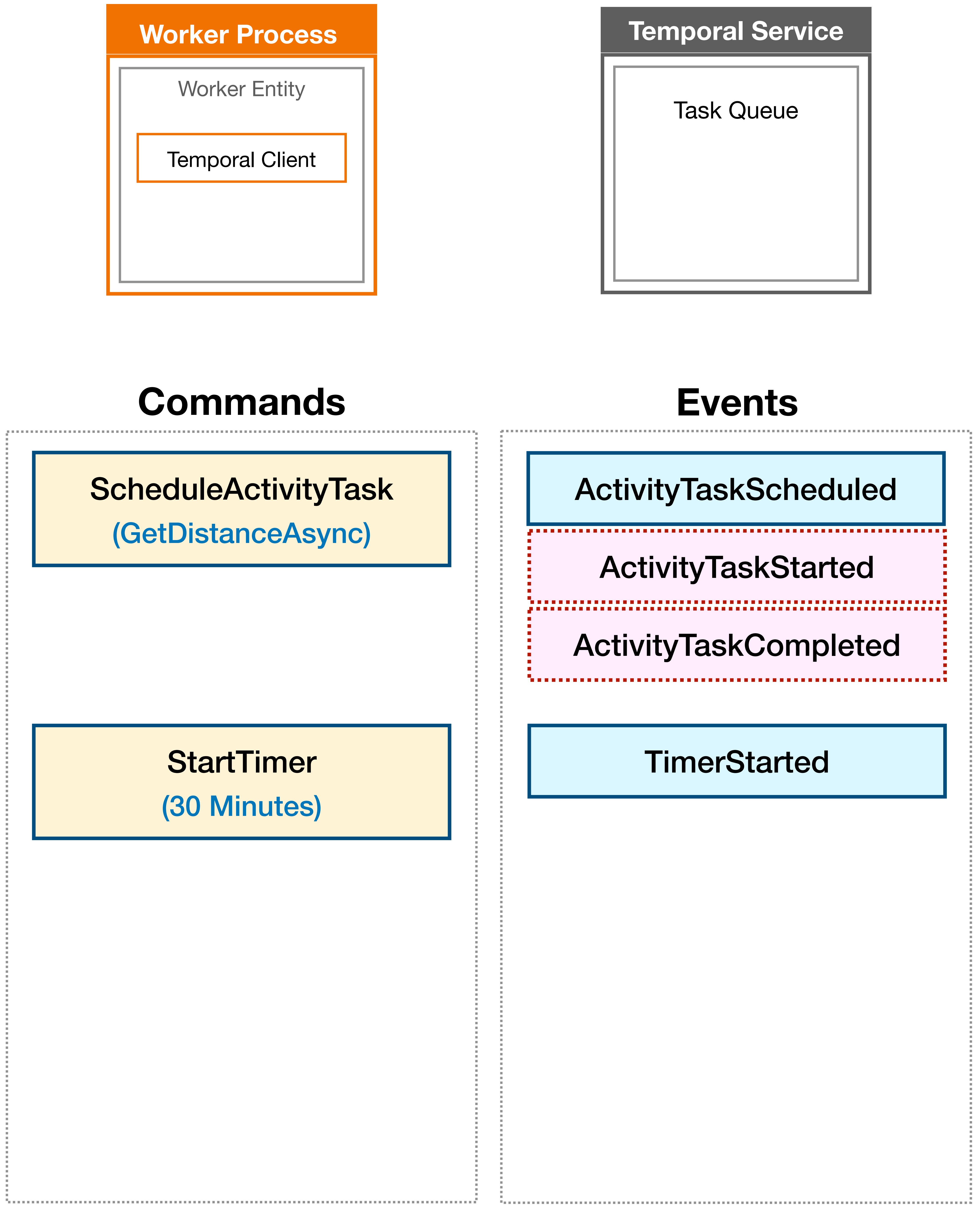




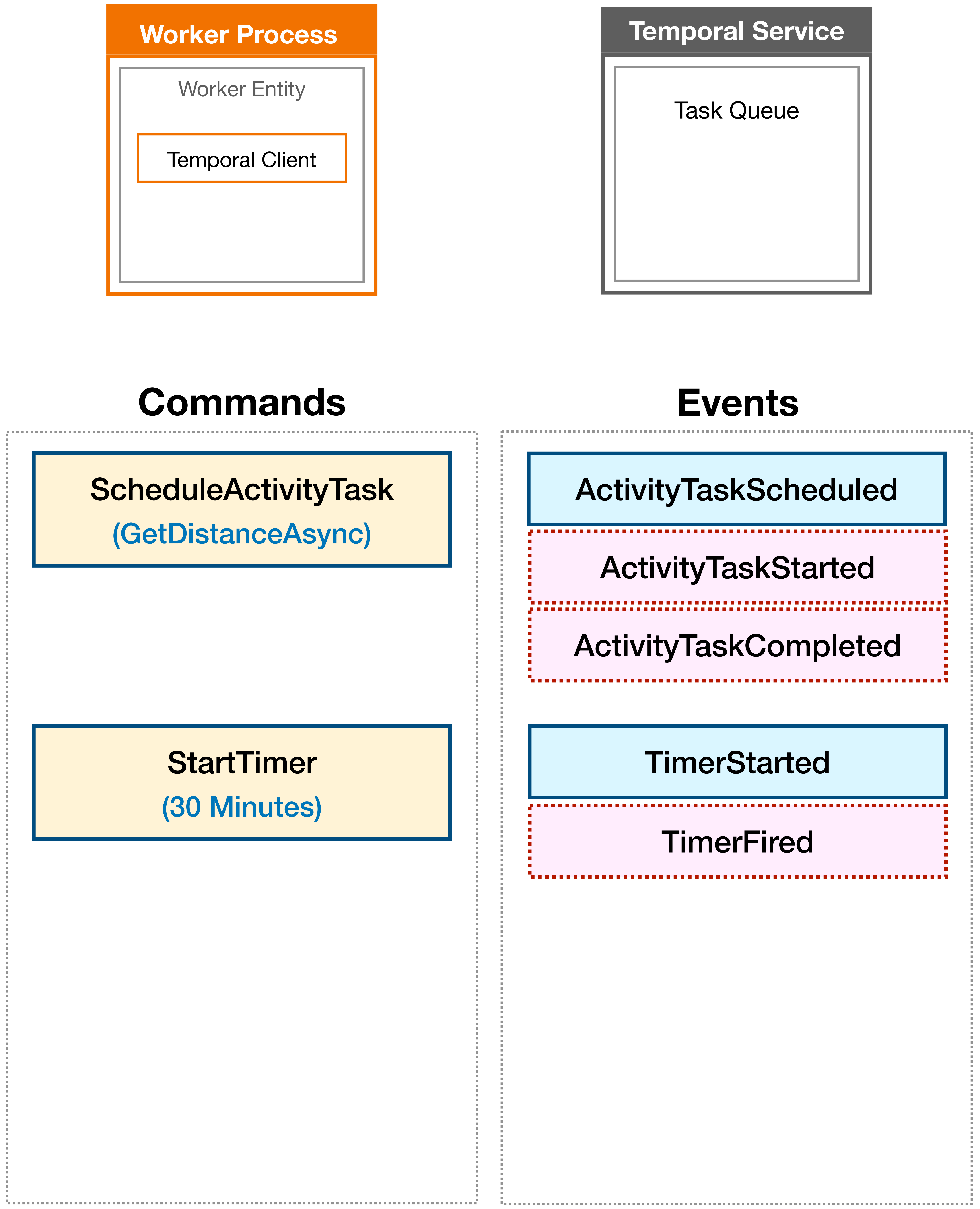
```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```



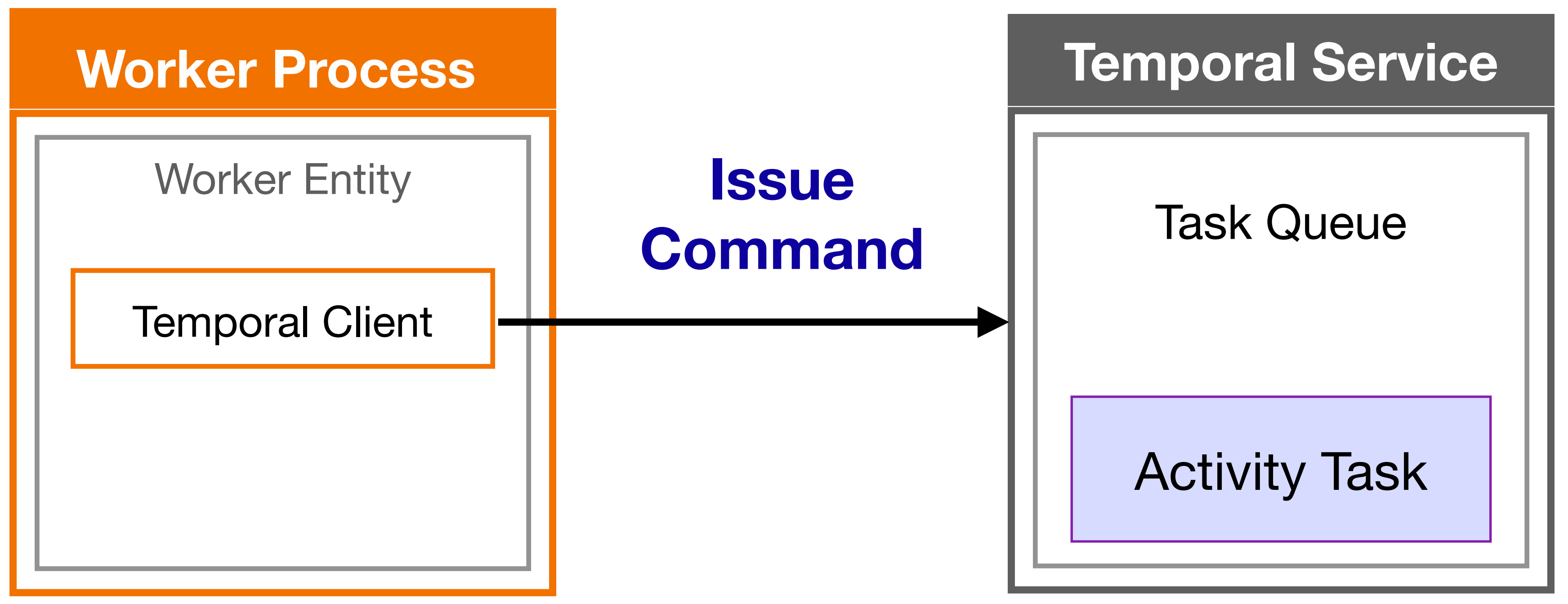
```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

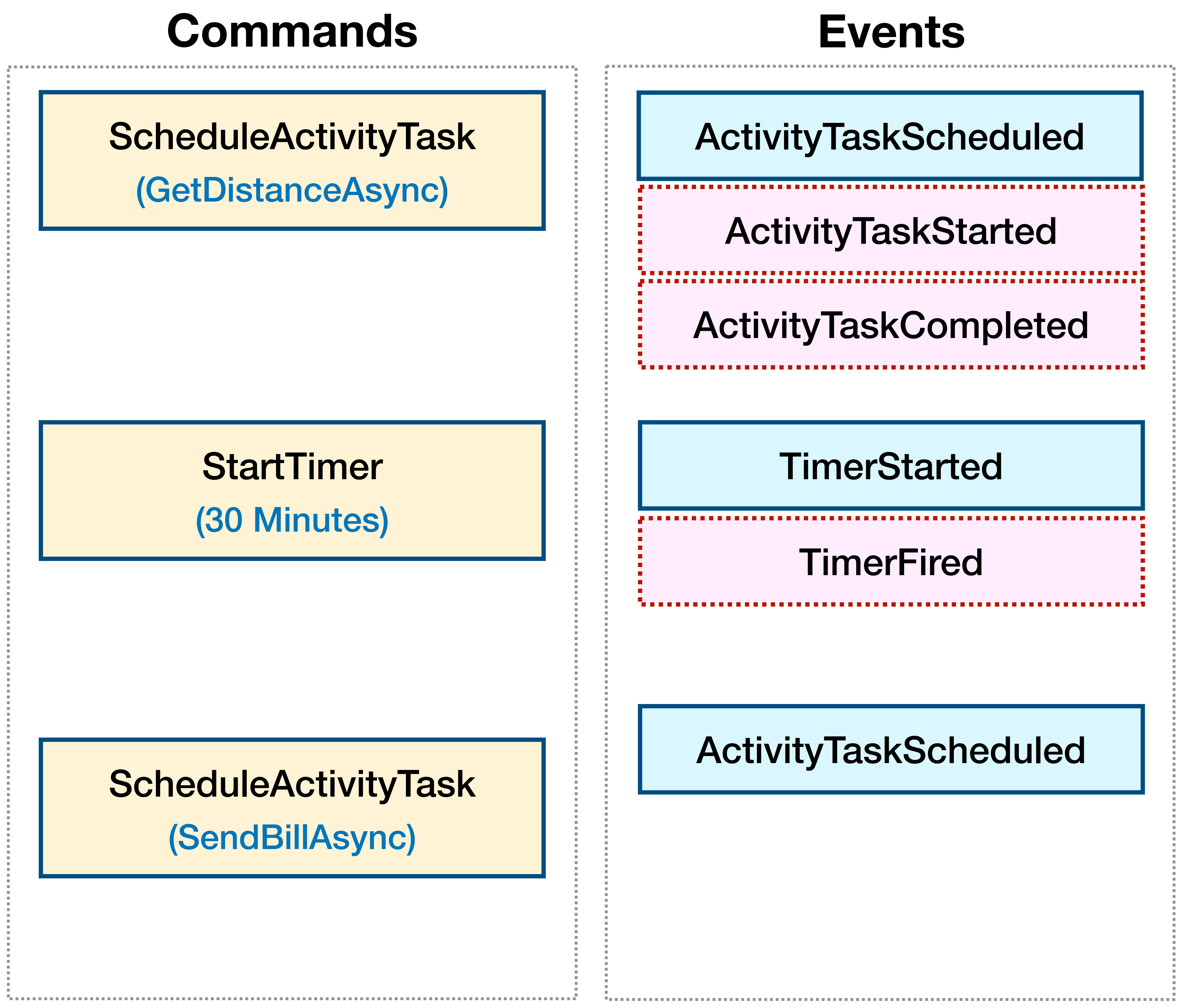


```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

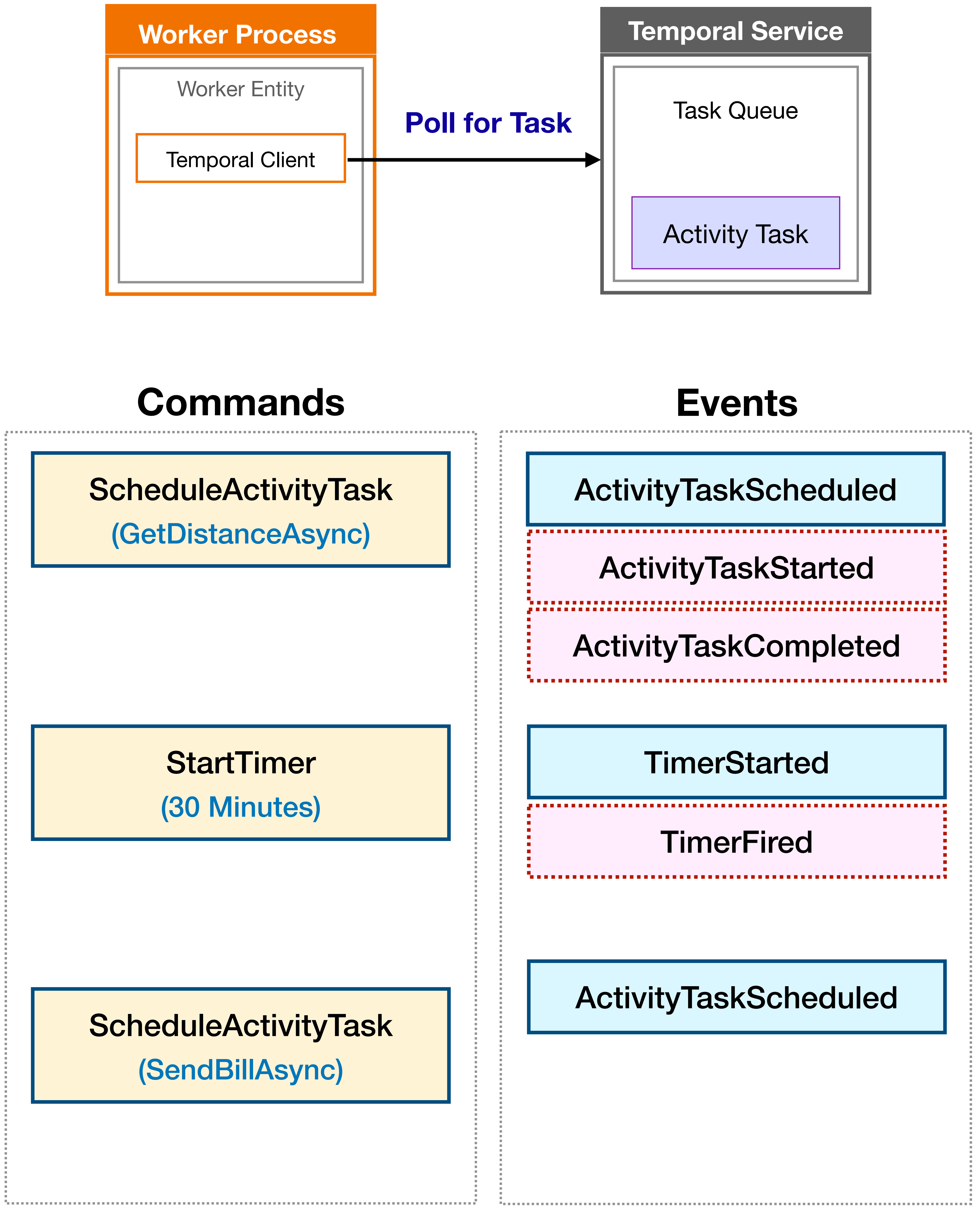


```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

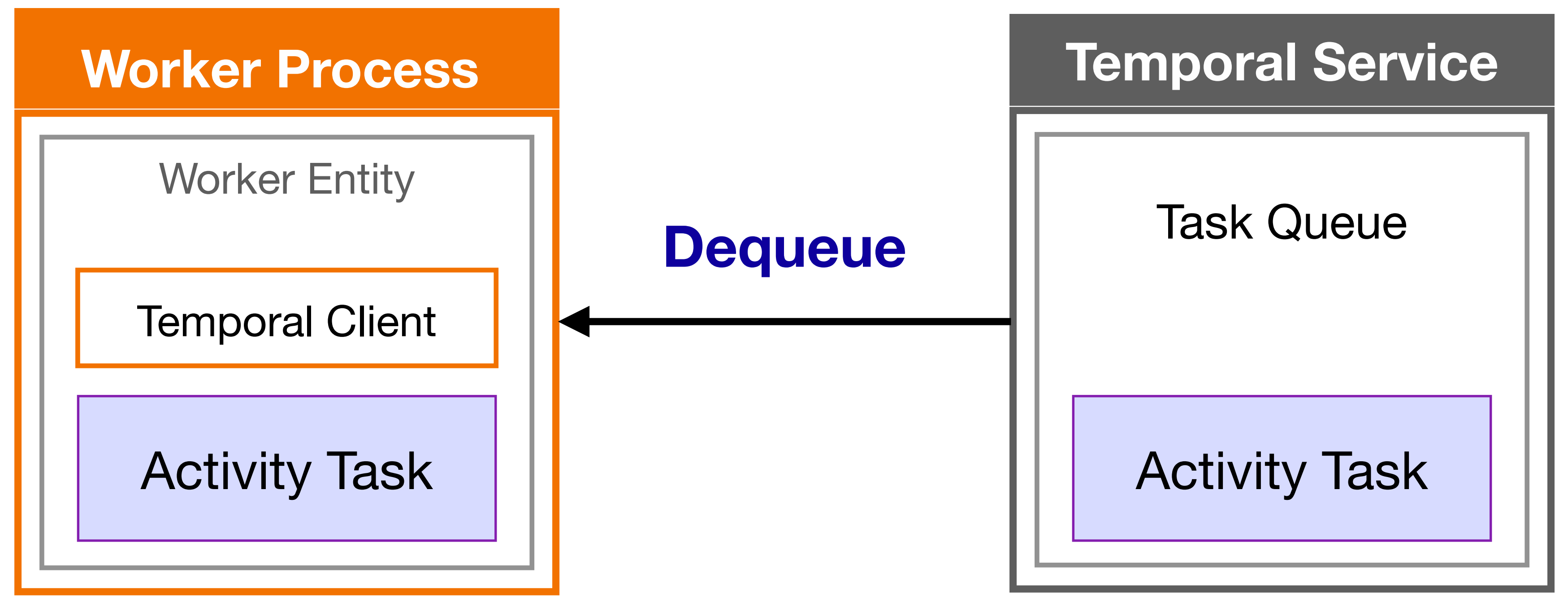


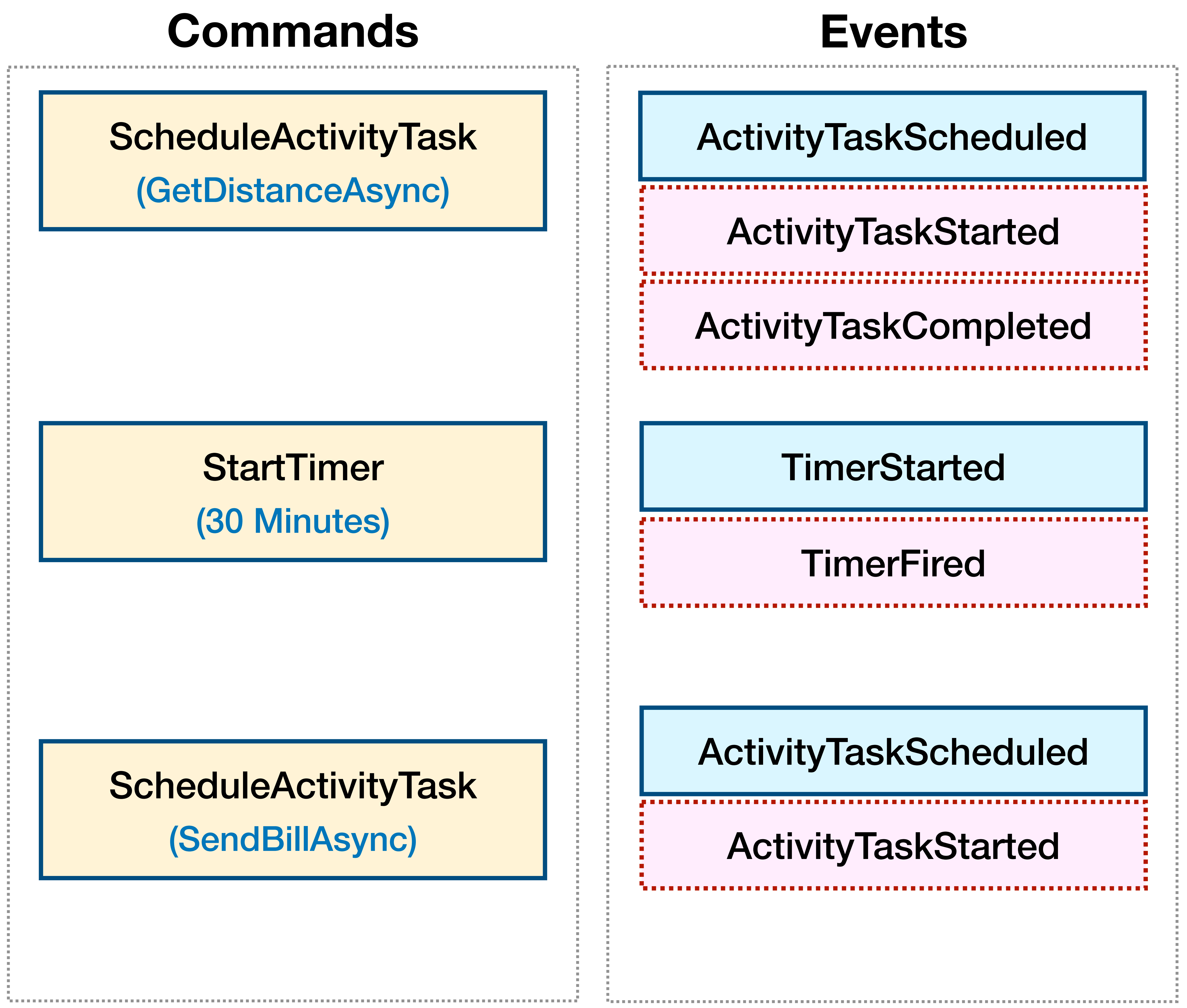


```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

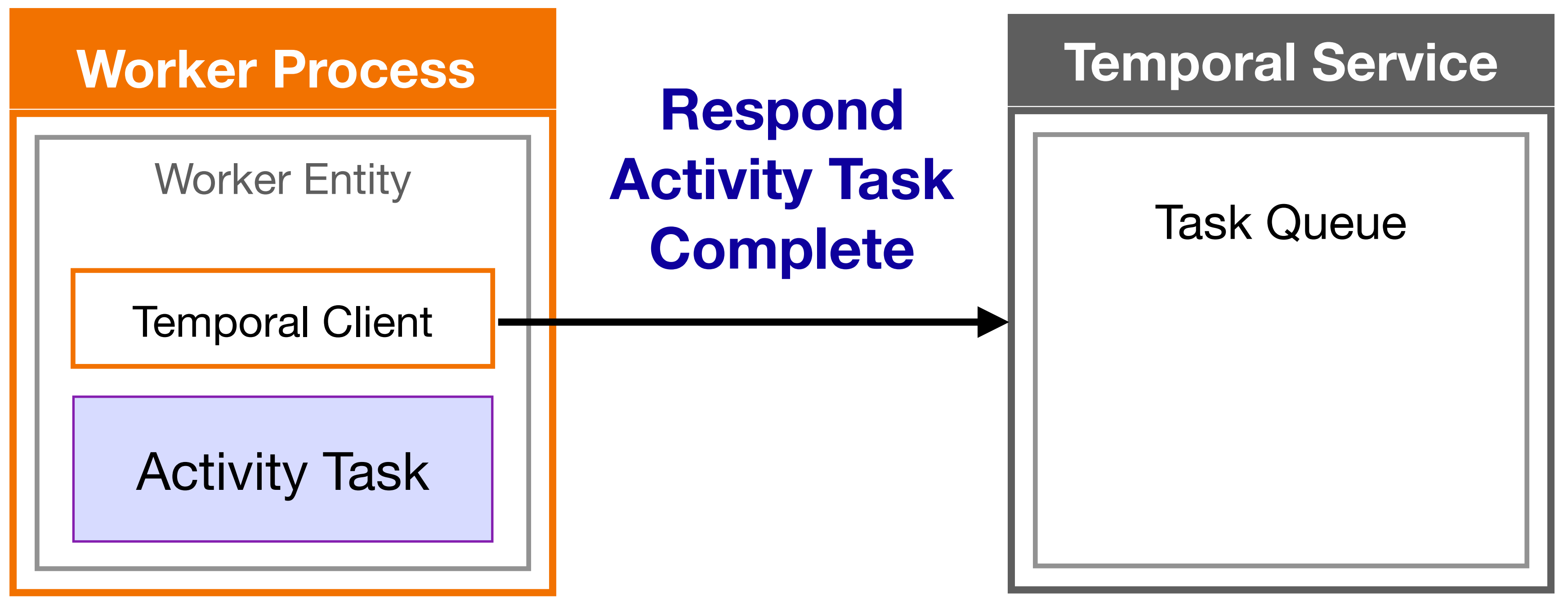


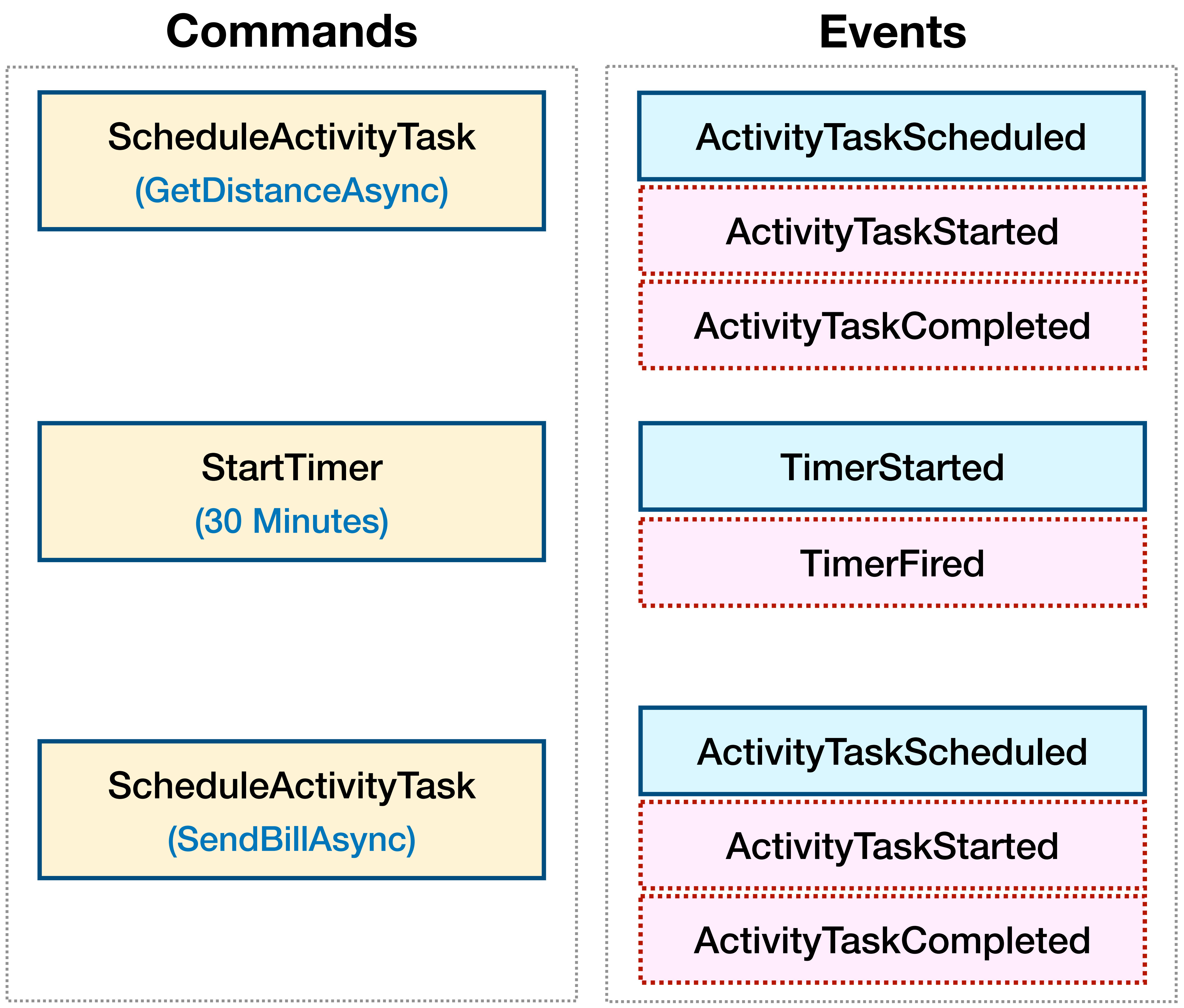
```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```



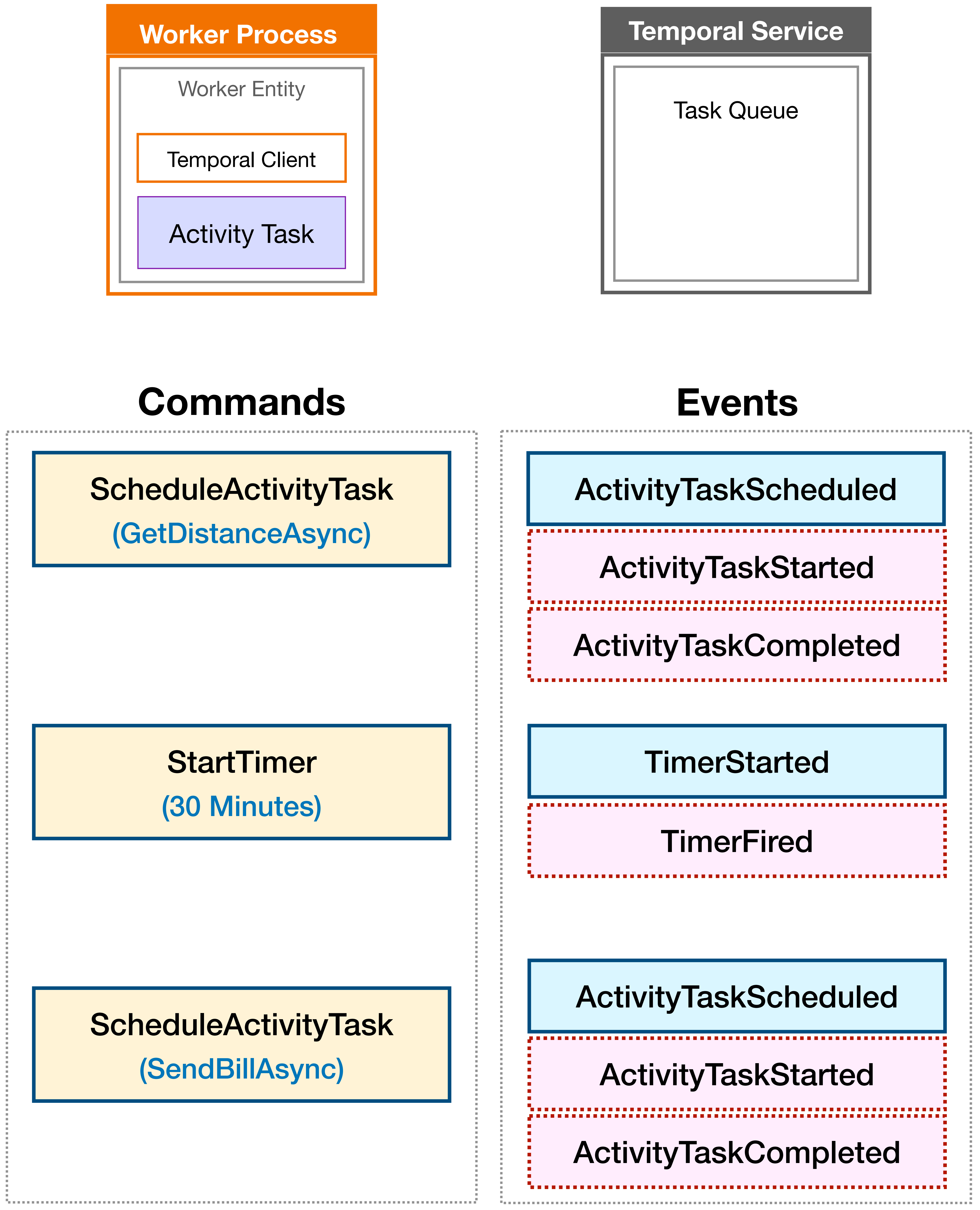


```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```



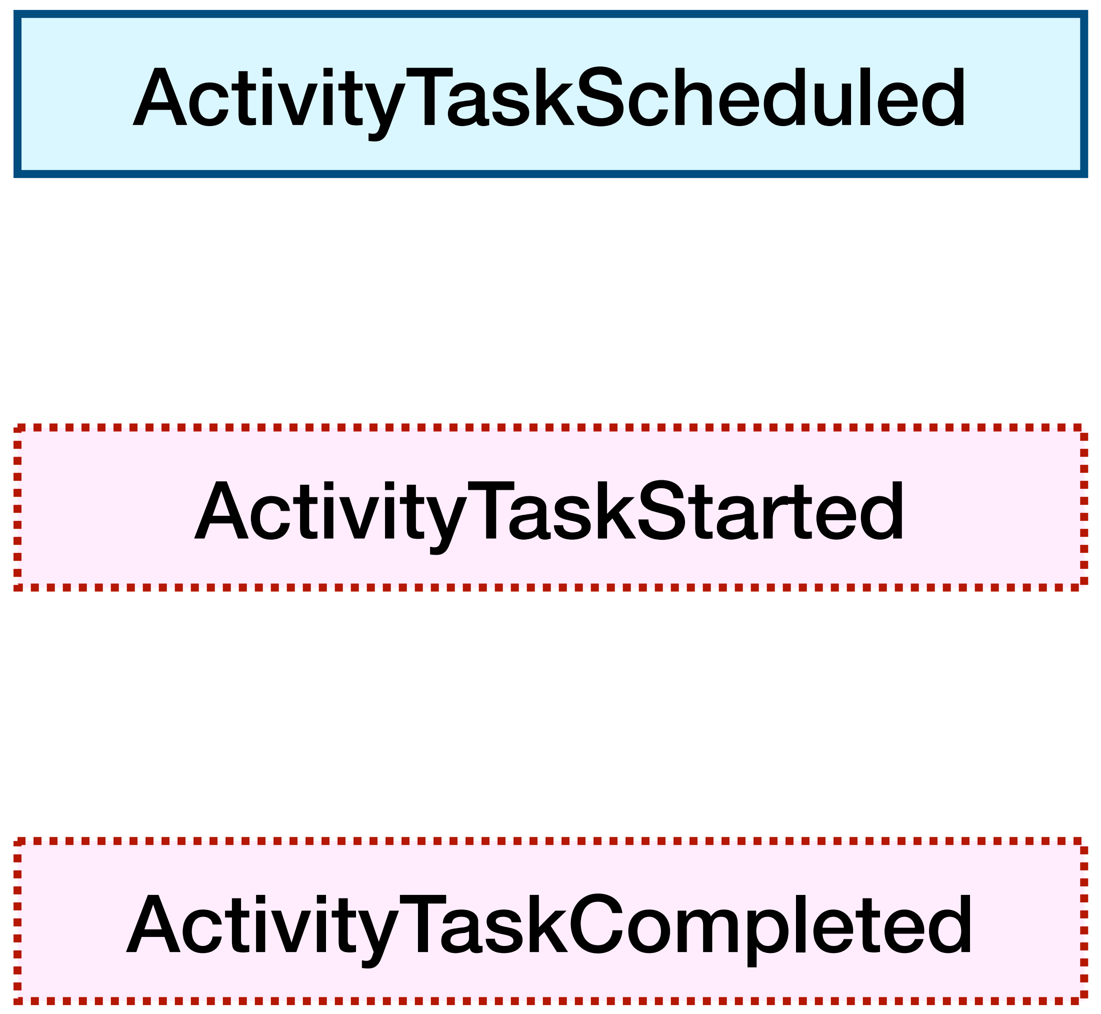


```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity options omitted for brevity
        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address),
            options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("can't deliver");
        }

        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill),
            options);

        return confirmation;
    }
}
```

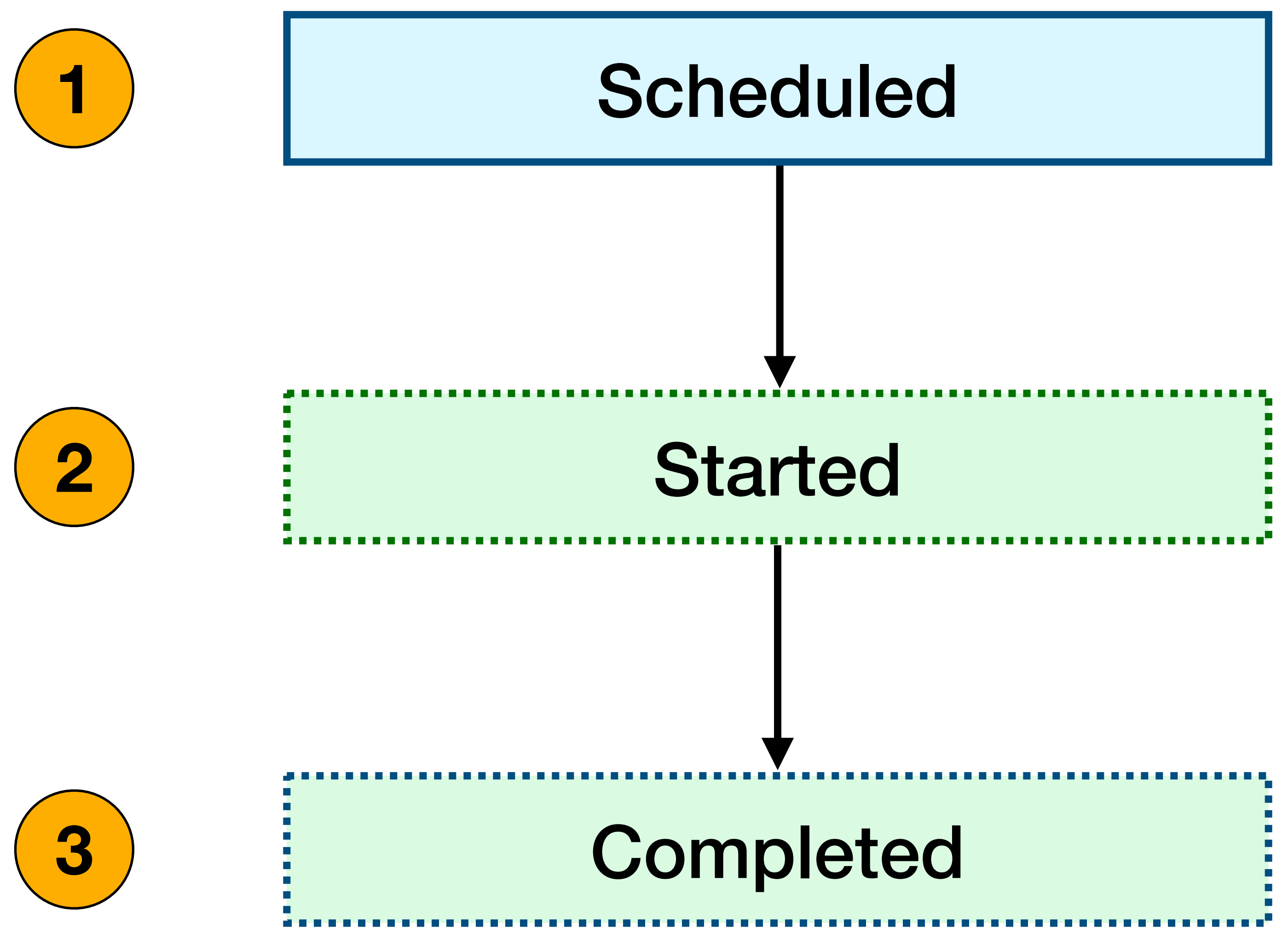


## Workflow and Activity Task States

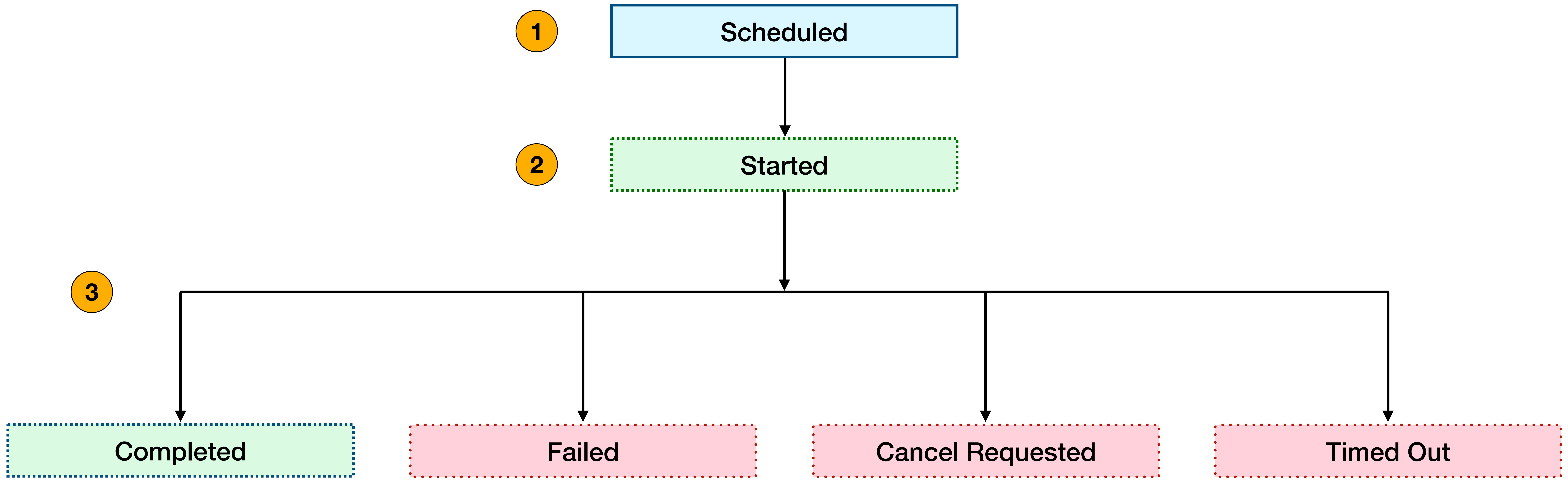
## Activity Task Event Sequence



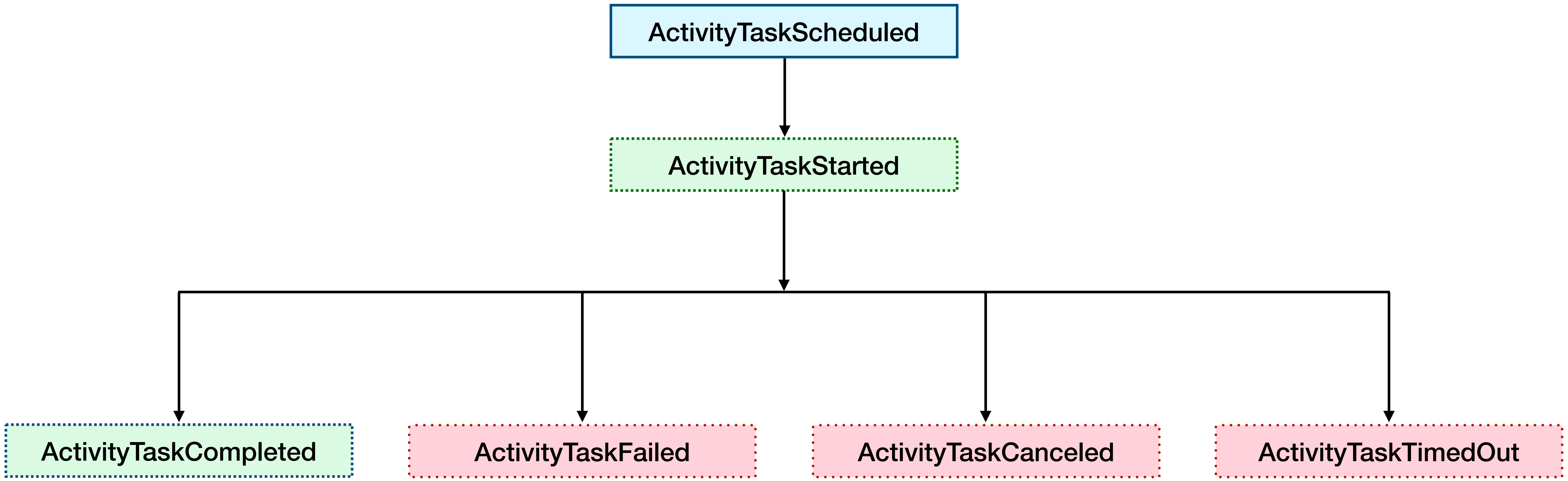
## Activity States in that Sequence



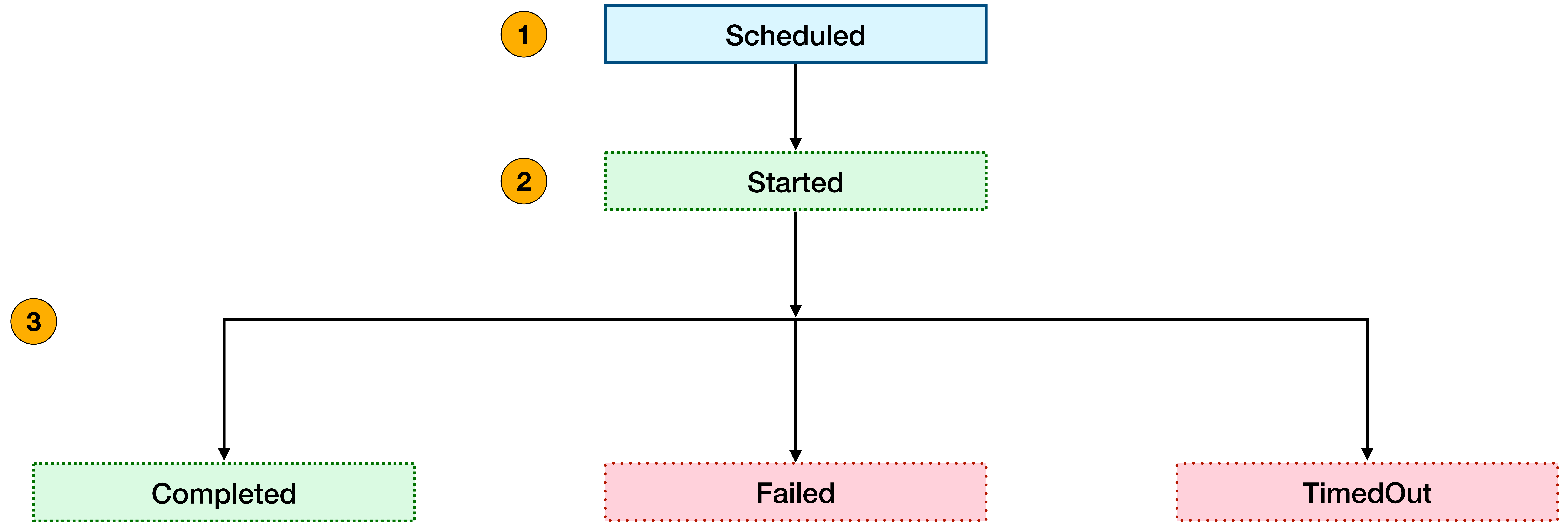
# Activity Task States



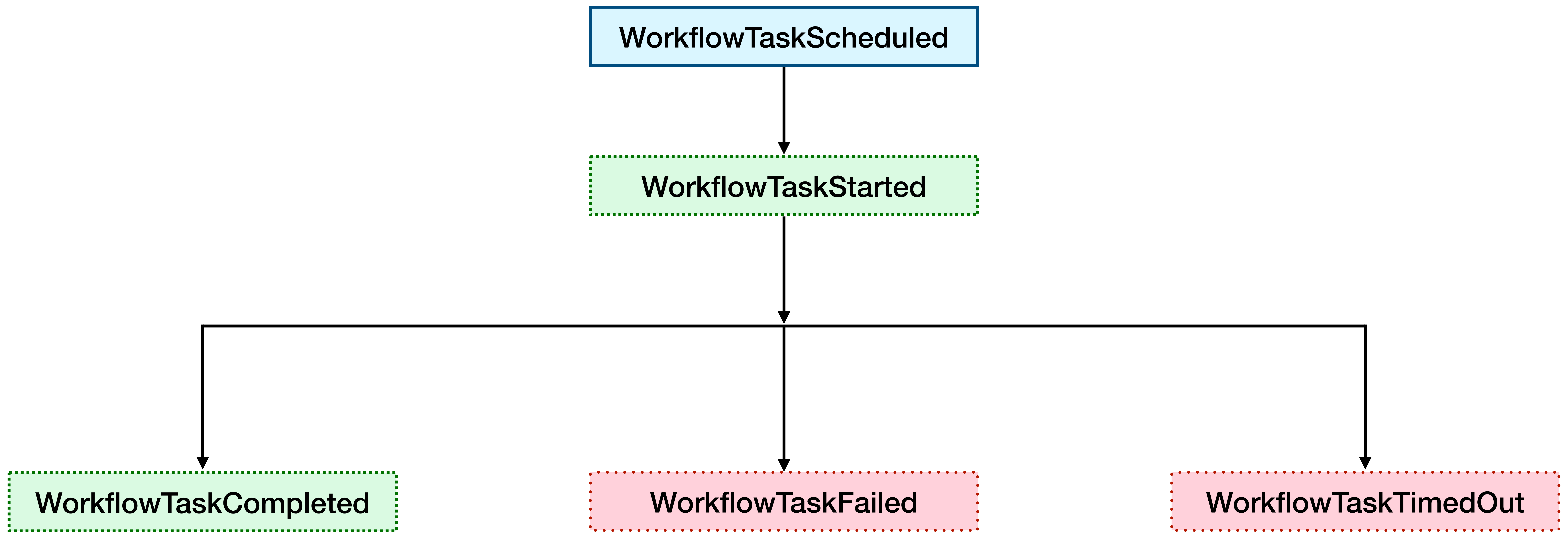
# Activity Task Events



# Workflow Task States



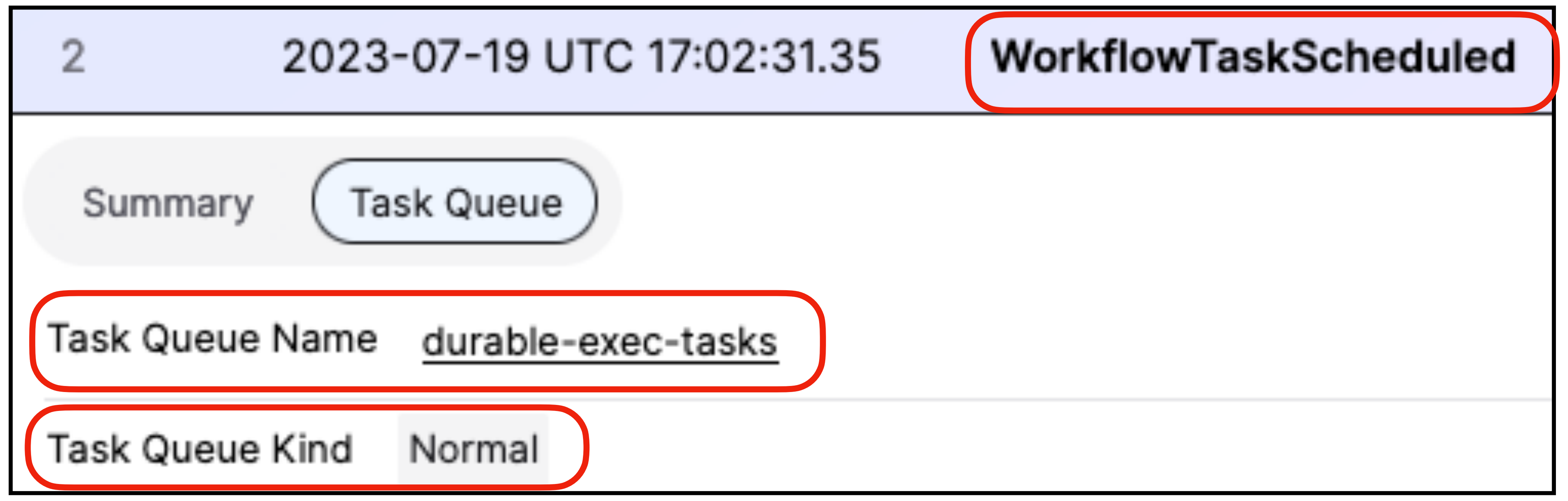
# Workflow Task Events



# Sticky Execution

- **To improve effectiveness of Worker's caching, Temporal use "sticky" execution for Workflow Tasks**
  - Directs Workflow Tasks to the same Worker that accepted them earlier in the same Workflow Execution.
- **Sticky execution is visible in the Web UI**
  - See the Task Queue Name / Kind fields
- **This does not apply to Activity Tasks**

**First Workflow Task**

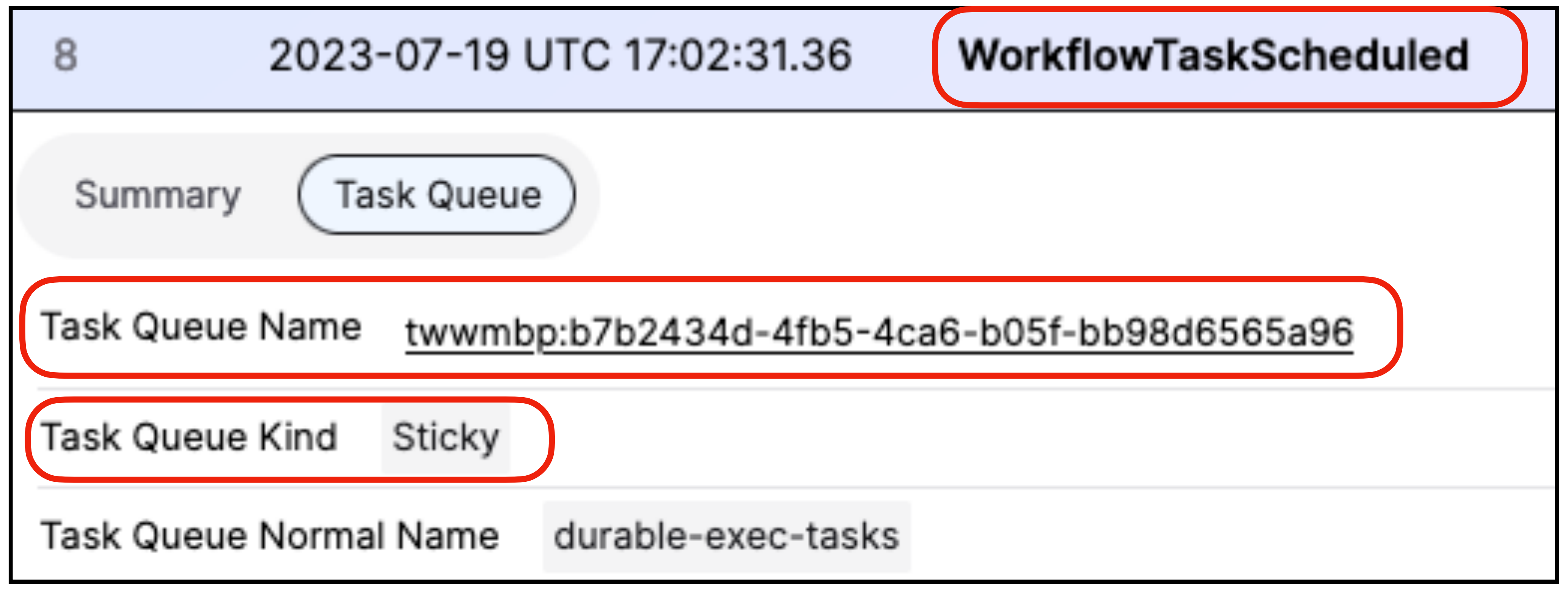


2 2023-07-19 UTC 17:02:31.35 WorkflowTaskScheduled

Summary | Task Queue

| | |
|---|---|
| Task Queue Name | durable-exec-tasks |
| Task Queue Kind | Normal |

**Later Workflow Task**



8 2023-07-19 UTC 17:02:31.36 WorkflowTaskScheduled

Summary | Task Queue

| | |
|---|---|
| Task Queue Name | twwmbp:b7b2434d-4fb5-4ca6-b05f-bb98d6565a96 |
| Task Queue Kind | Sticky |
| Task Queue Normal Name | durable-exec-tasks |

# Review

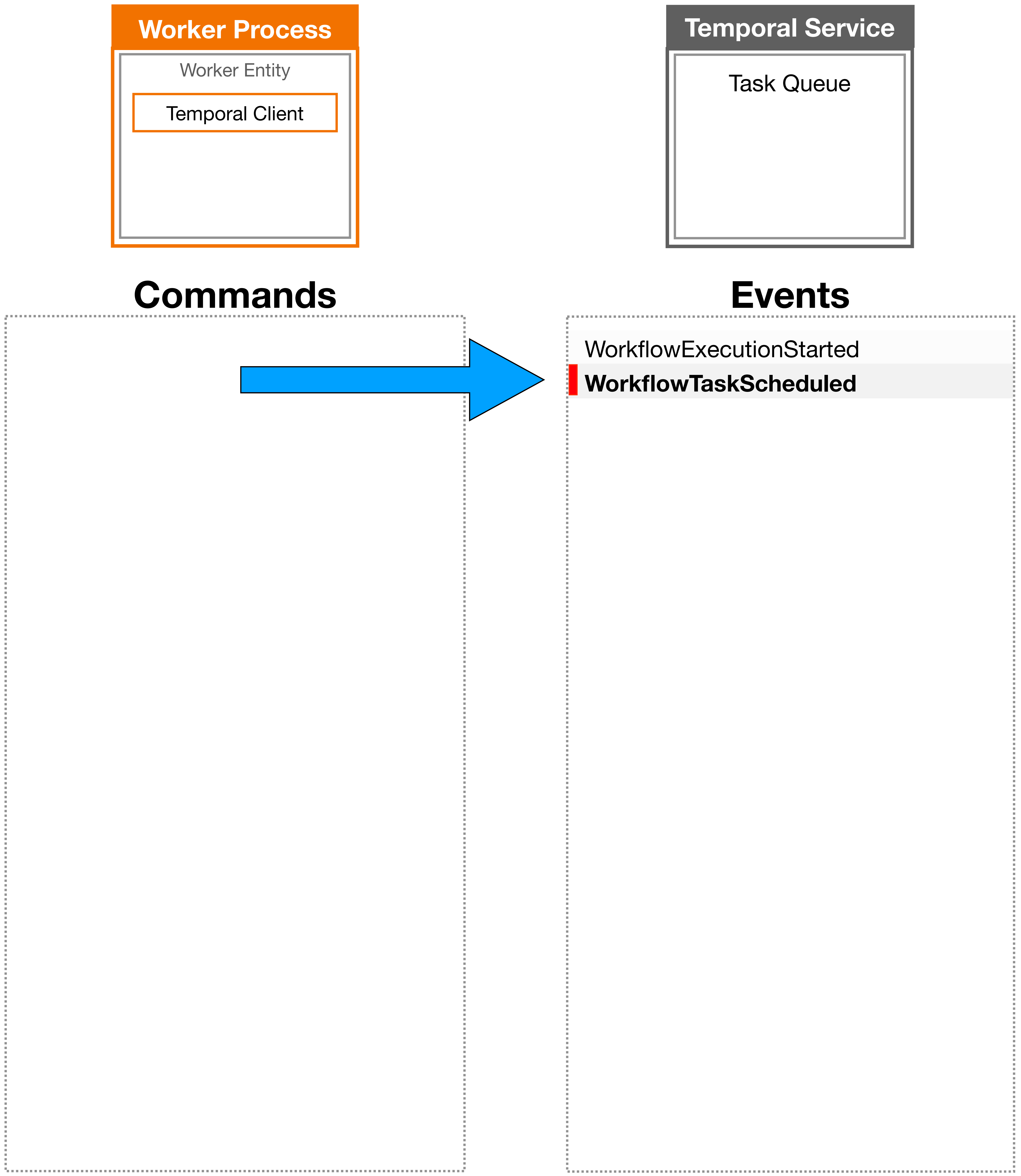
- Workflow Definition + Execution Request = Workflow Execution
- Each Workflow Execution is associated with an Event History that is the source of truth
- Executing Activities or creating Timers issues Commands to the Temporal Service, which creates Tasks, and adds Events to the Event History.
- Workflow Execution States can be Open or Closed
    - Closed means Completed, Continue-As-New, Failed, Timed Out, Cancelled, or Terminated
- Workflow and Activity Tasks can be Scheduled, Started, or Completed. They can also fail or time out.
- Sticky Execution directs Workflow Tasks to the same Worker that accepted them earlier in the same Workflow Execution

# Temporal 102

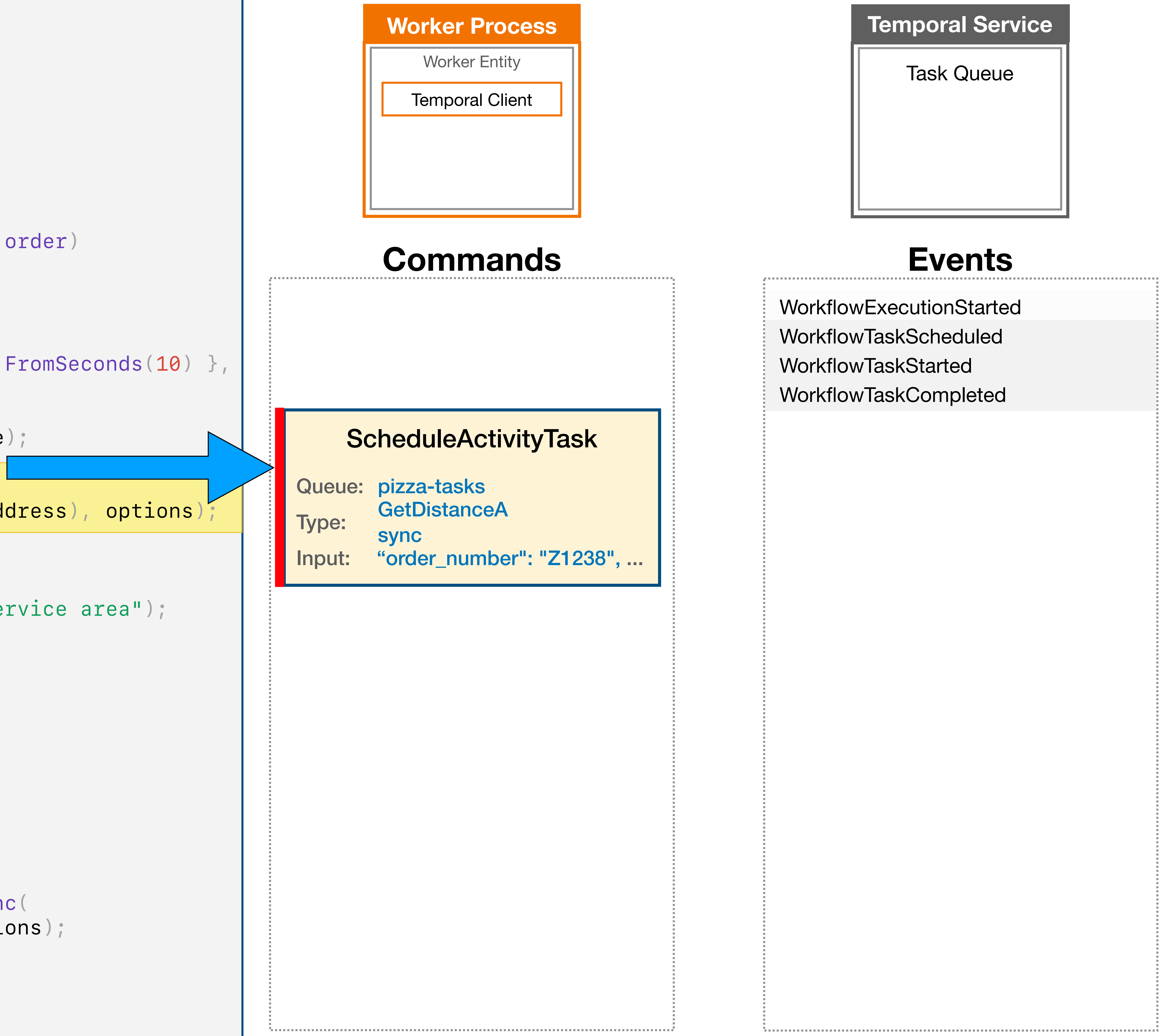
## History Replay:

How Temporal Provides Durable Execution

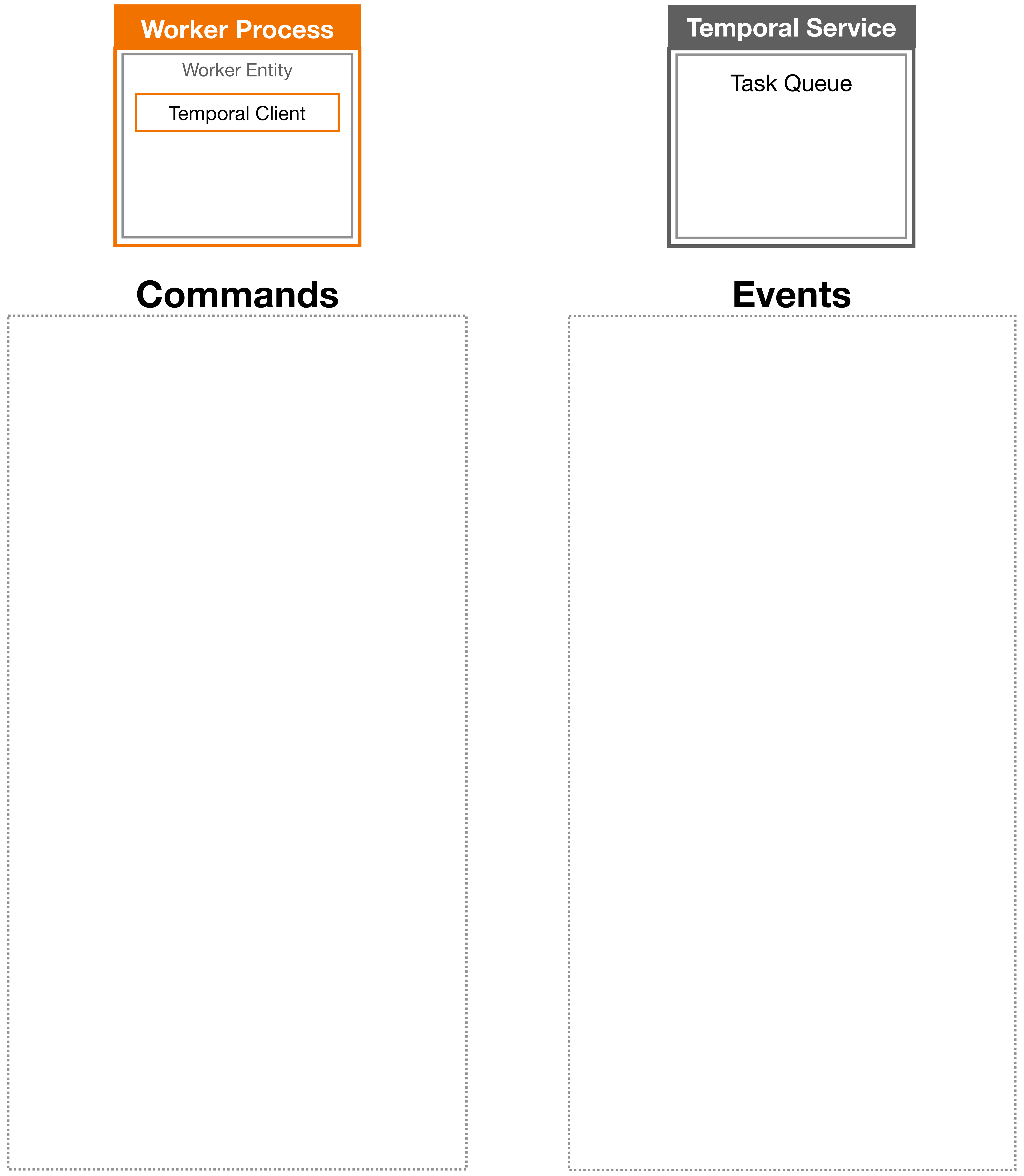
```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



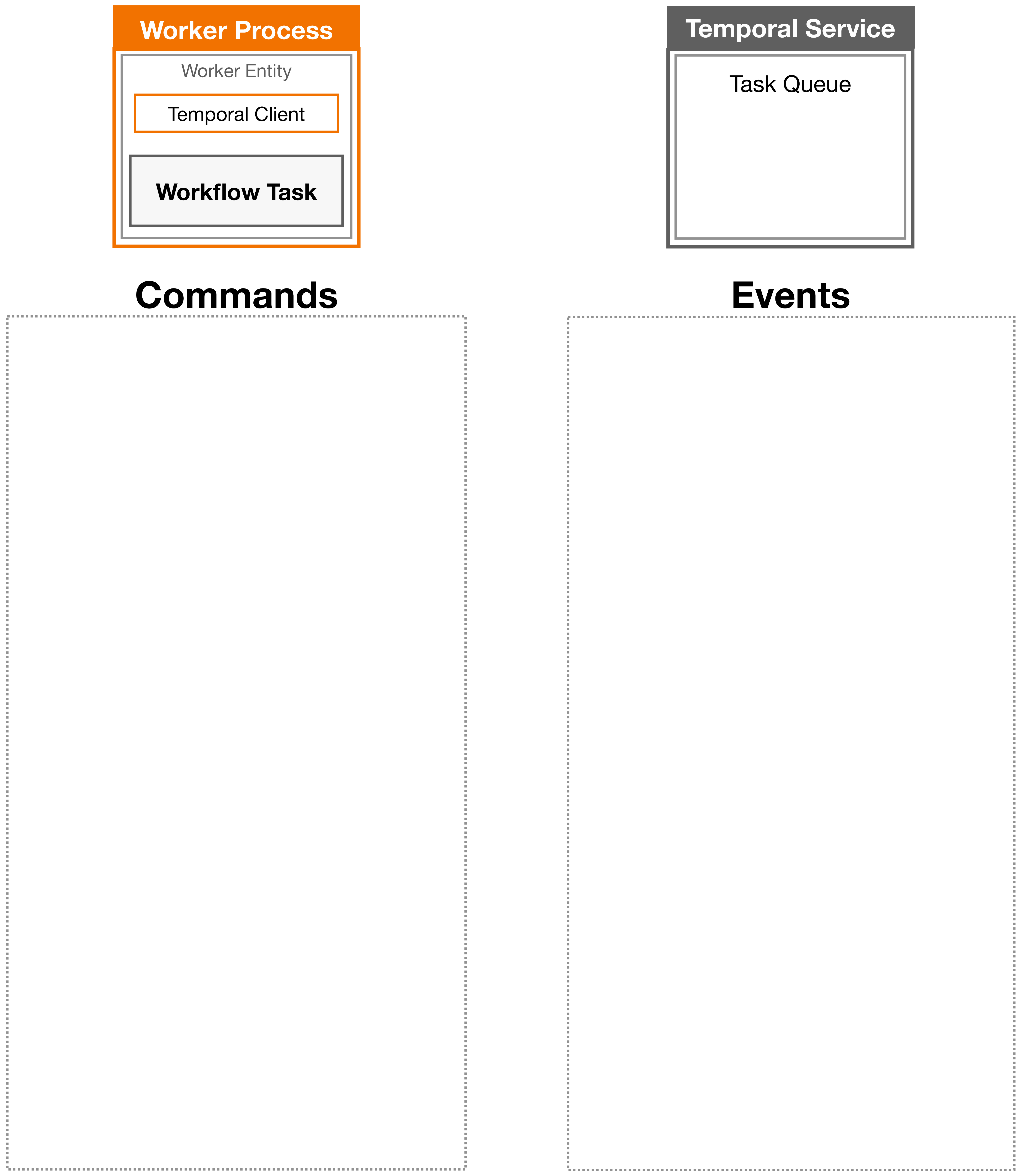
```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

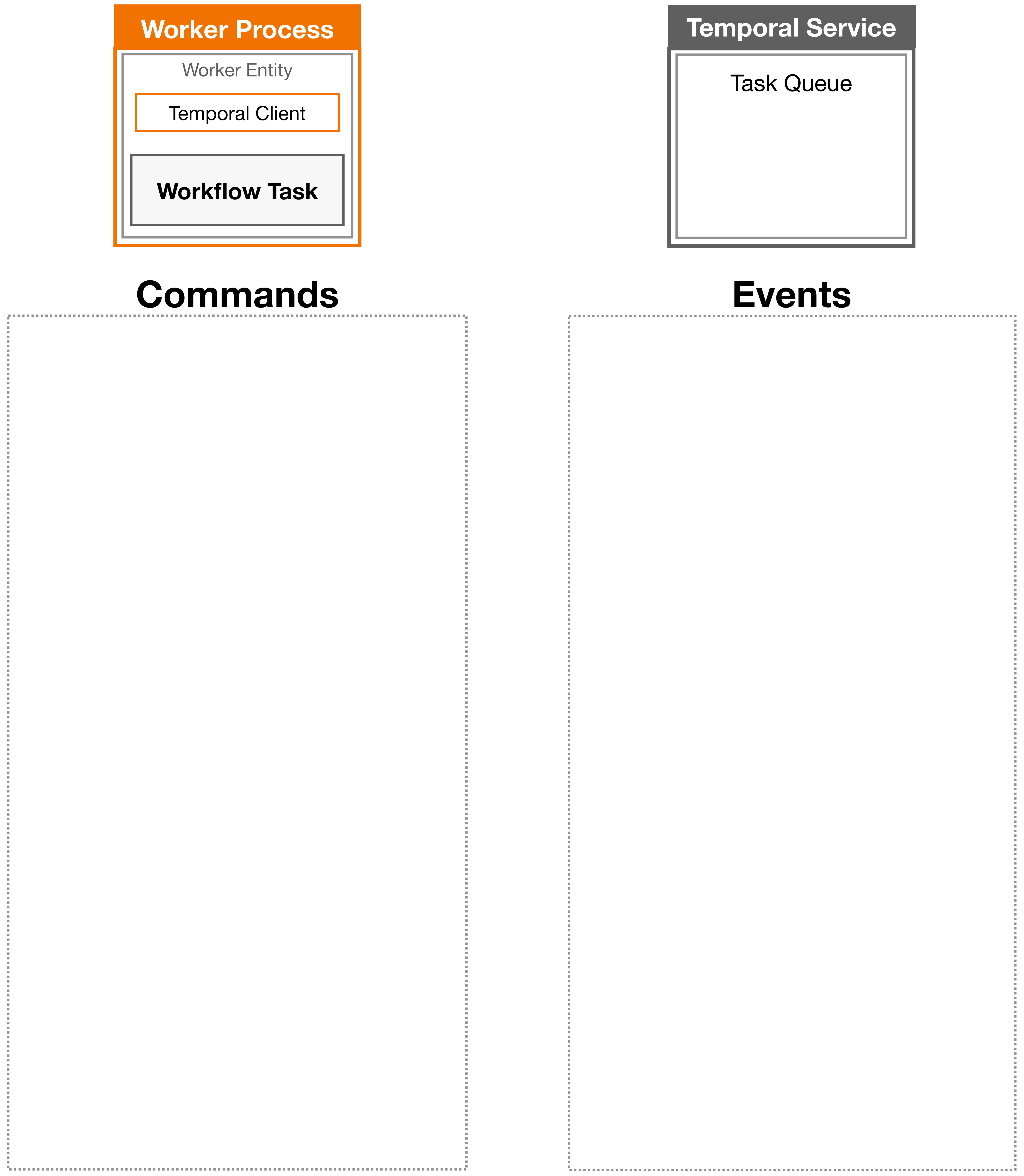


```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

**Worker crashes here**



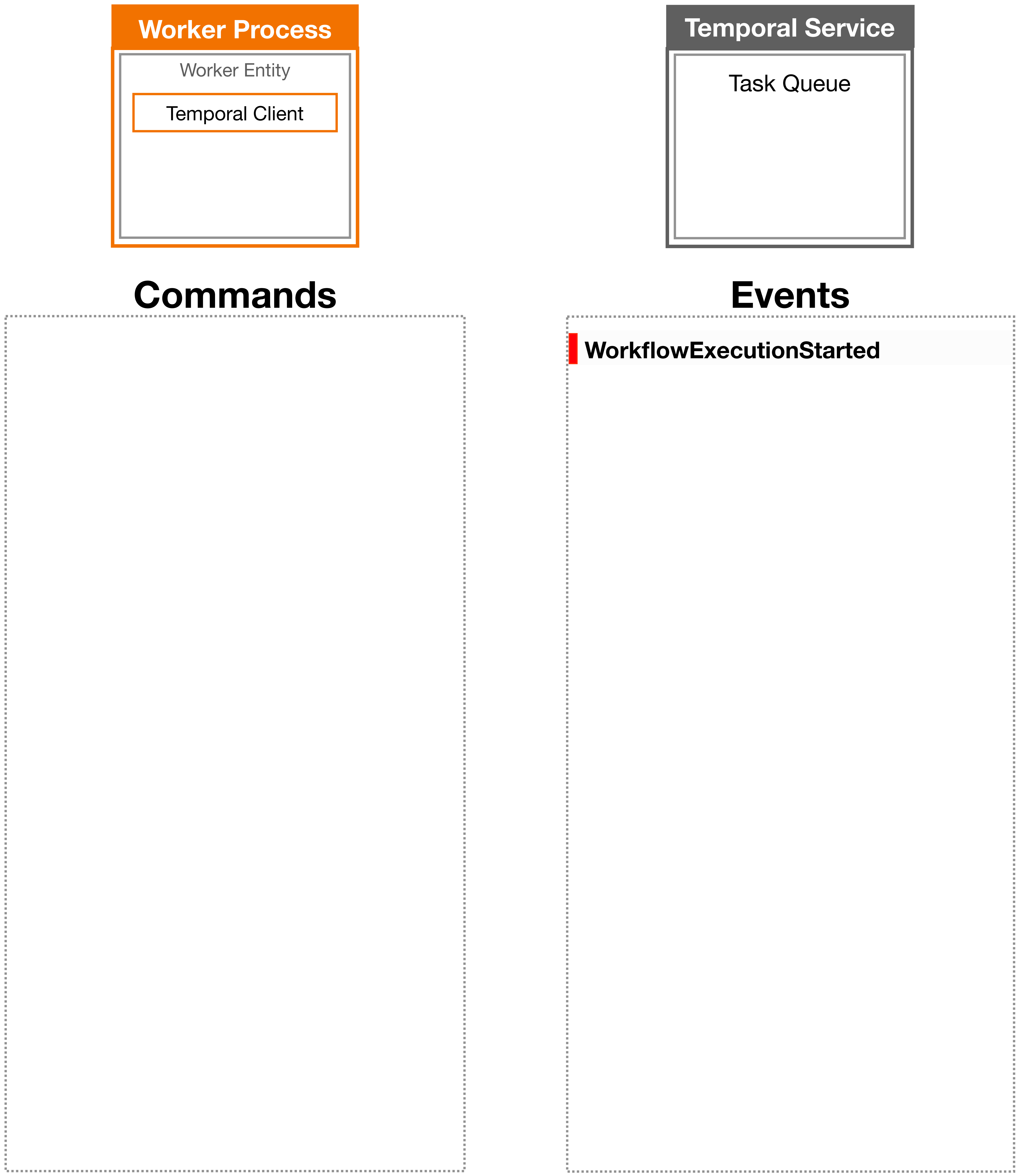
```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

# Start Workflow Execution

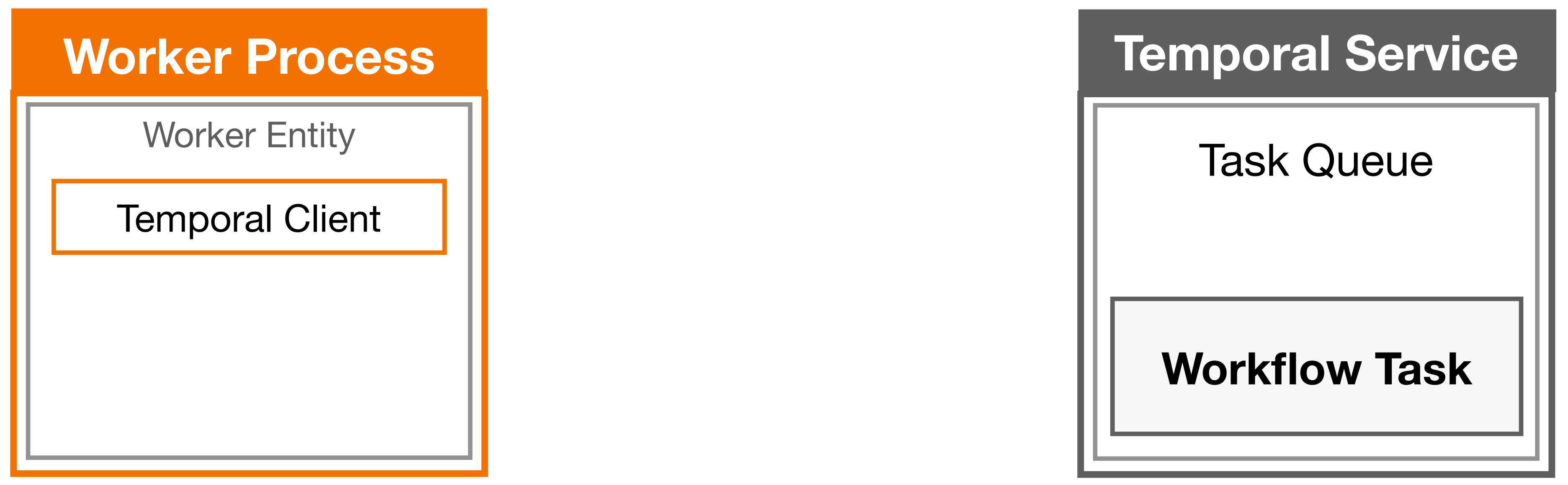
```
var result = await client.ExecuteWorkflowAsync(
    (PizzaWorkflow wf) => wf.RunAsync(order),
    new WorkflowOptions
    {
        Id = $"pizza-workflow-order-{order.OrderNumber}",
        TaskQueue = WorkflowConstants.TaskQueueName,
    });
```

```
[
    {
      "orderNumber": "Z1238",
      "customer": {
        "customerID": 12983,
        "name": "María García",
        "email": "maria1985@example.com",
        "phone": "415-555-7418"
      },
      "items": [
        {
          "description": "Large, with pepperoni",
          "price": 1500
        },
        {
          "description": "Small, with mushrooms and onions",
          “price": 1000
        }
      ],
      "isDelivery": true,
      "address": {
        "line1": "701 Mission Street",
        "line2": "Apartment 9C",
        "city": "San Francisco",
        "state": "CA",
        "postalCode": "94103"
      }
    }
]
```

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

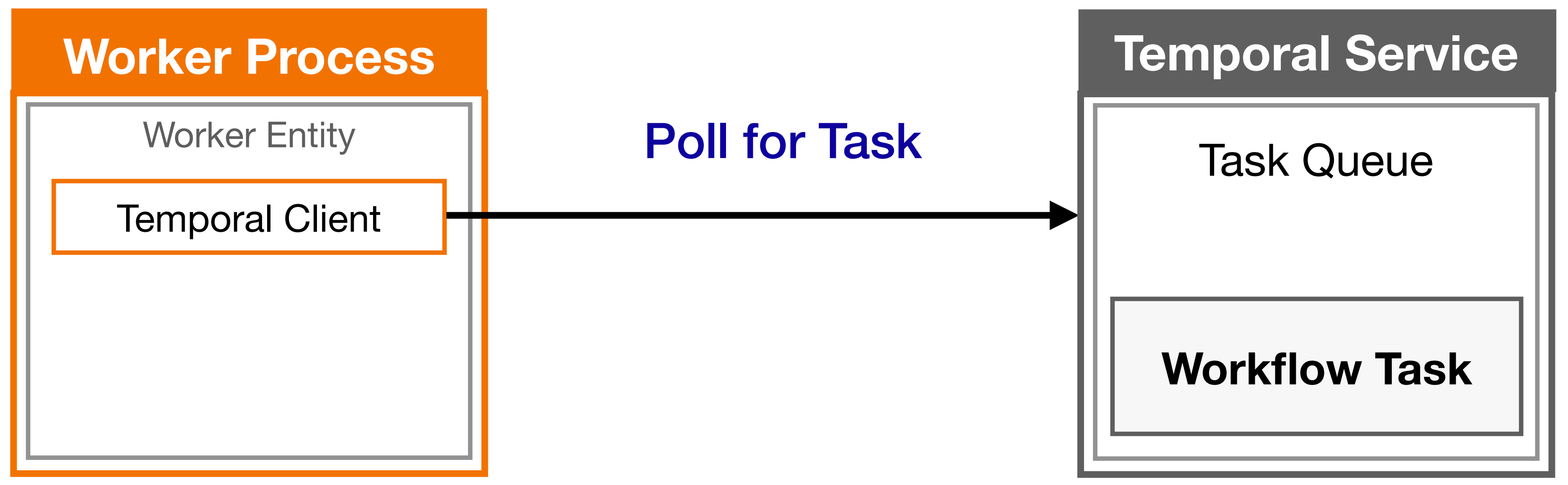


**Commands**

**Events**

WorkflowExecutionStarted
**WorkflowTaskScheduled**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

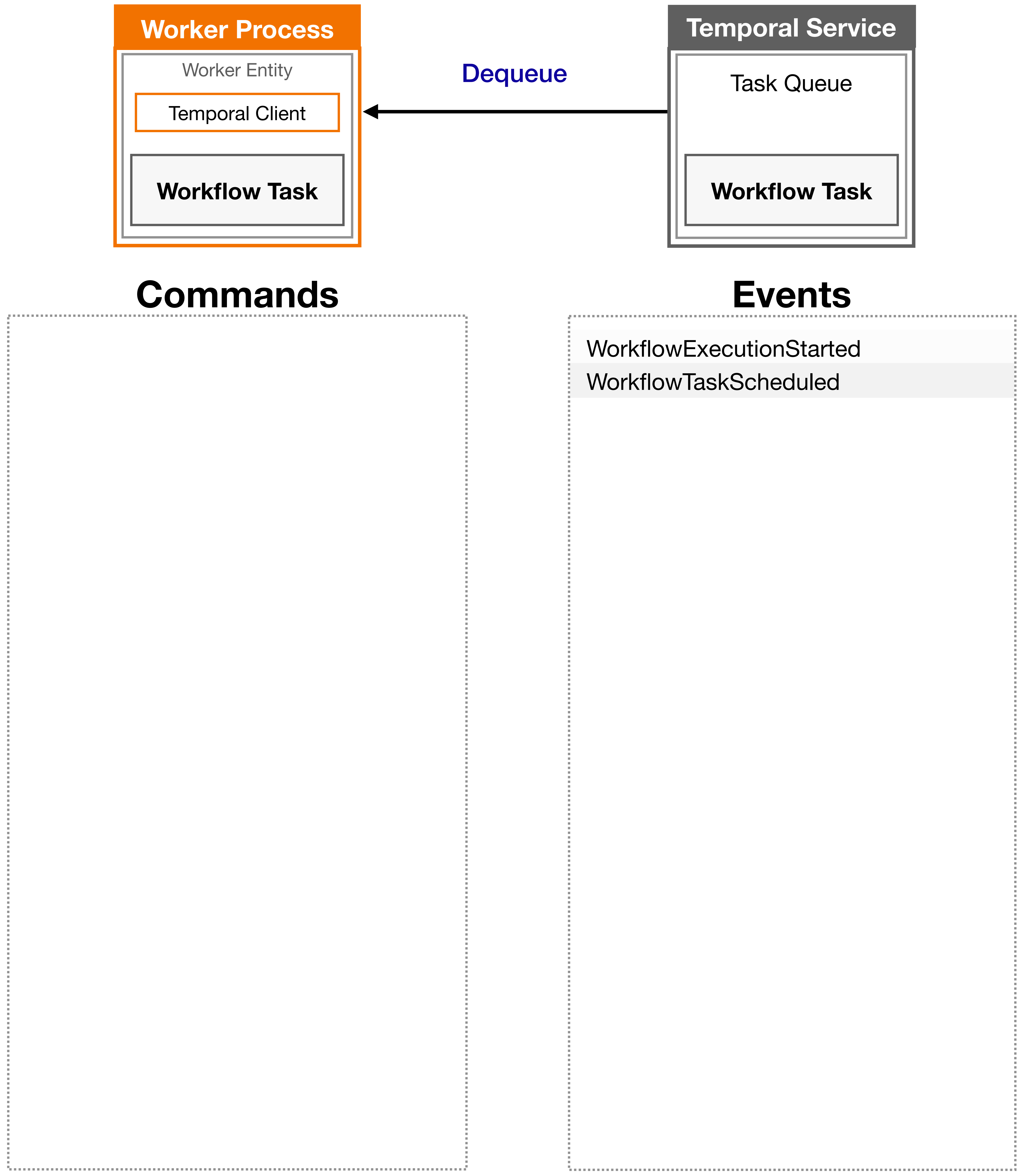


## Commands

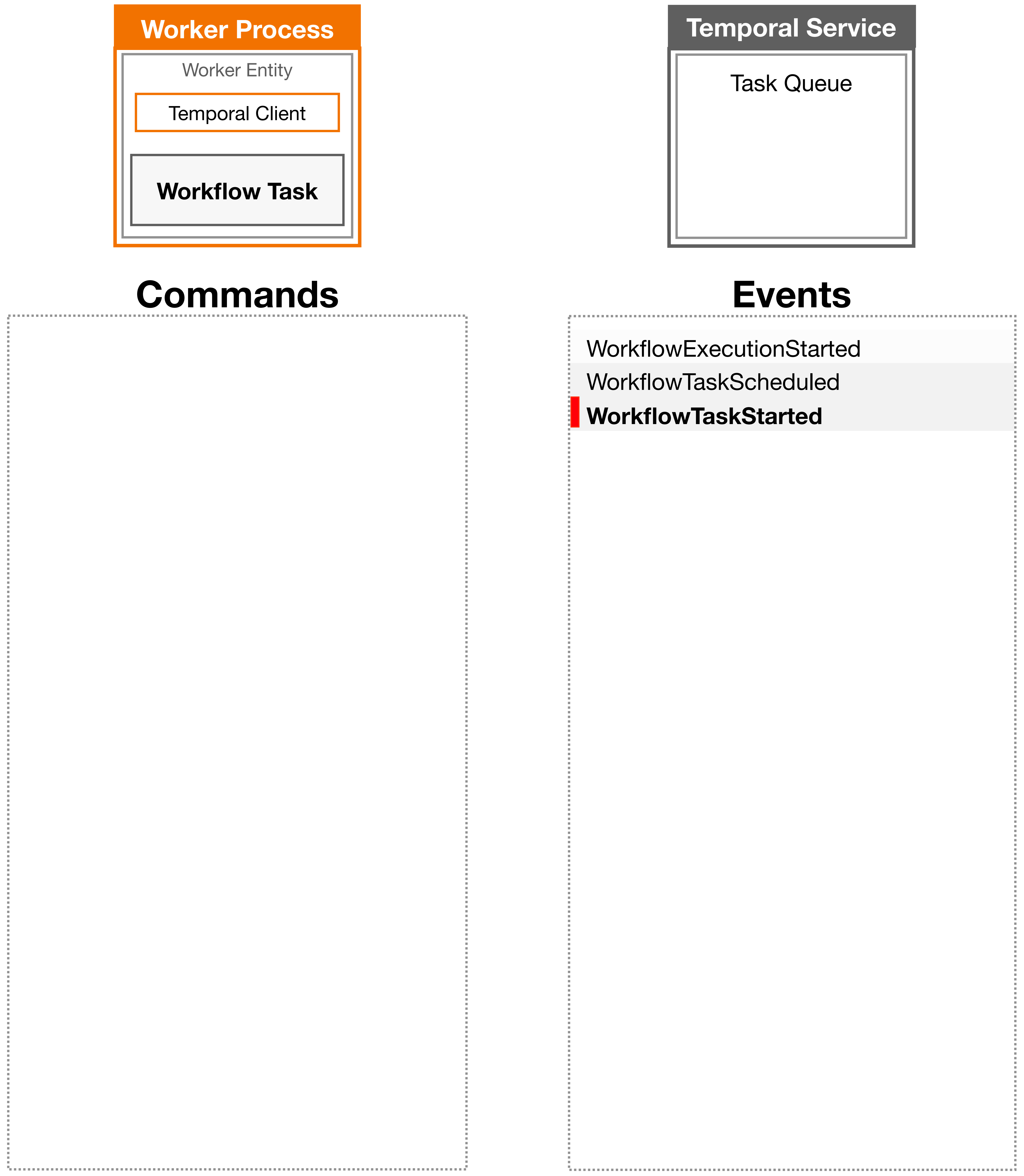
## Events

WorkflowExecutionStarted
WorkflowTaskScheduled

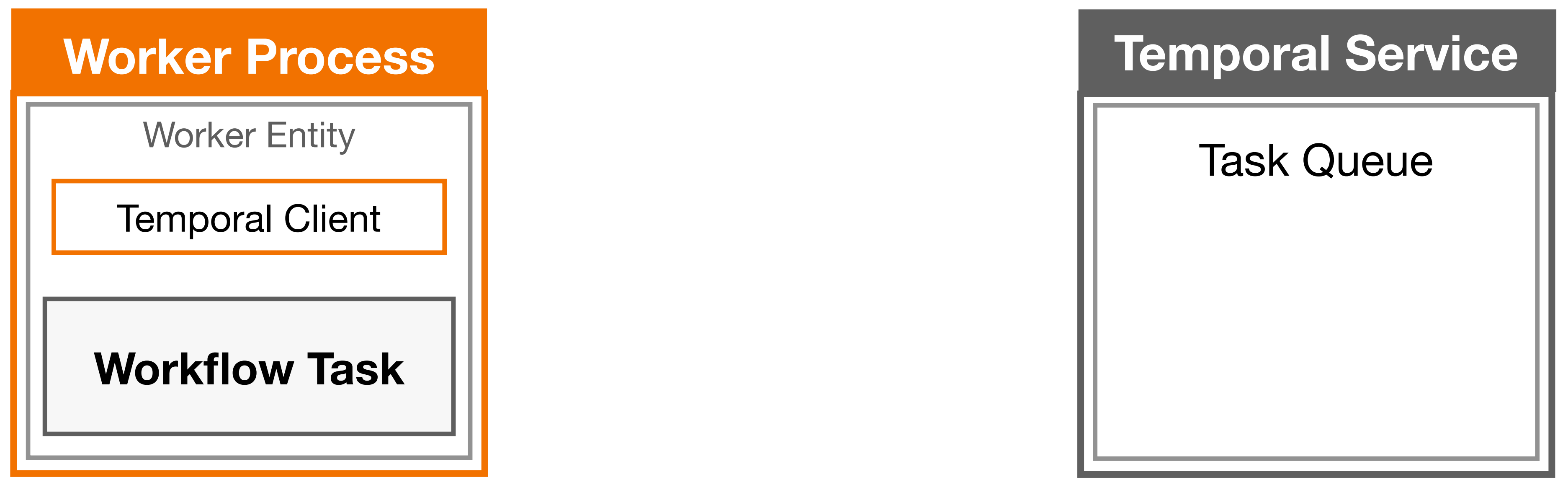
```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

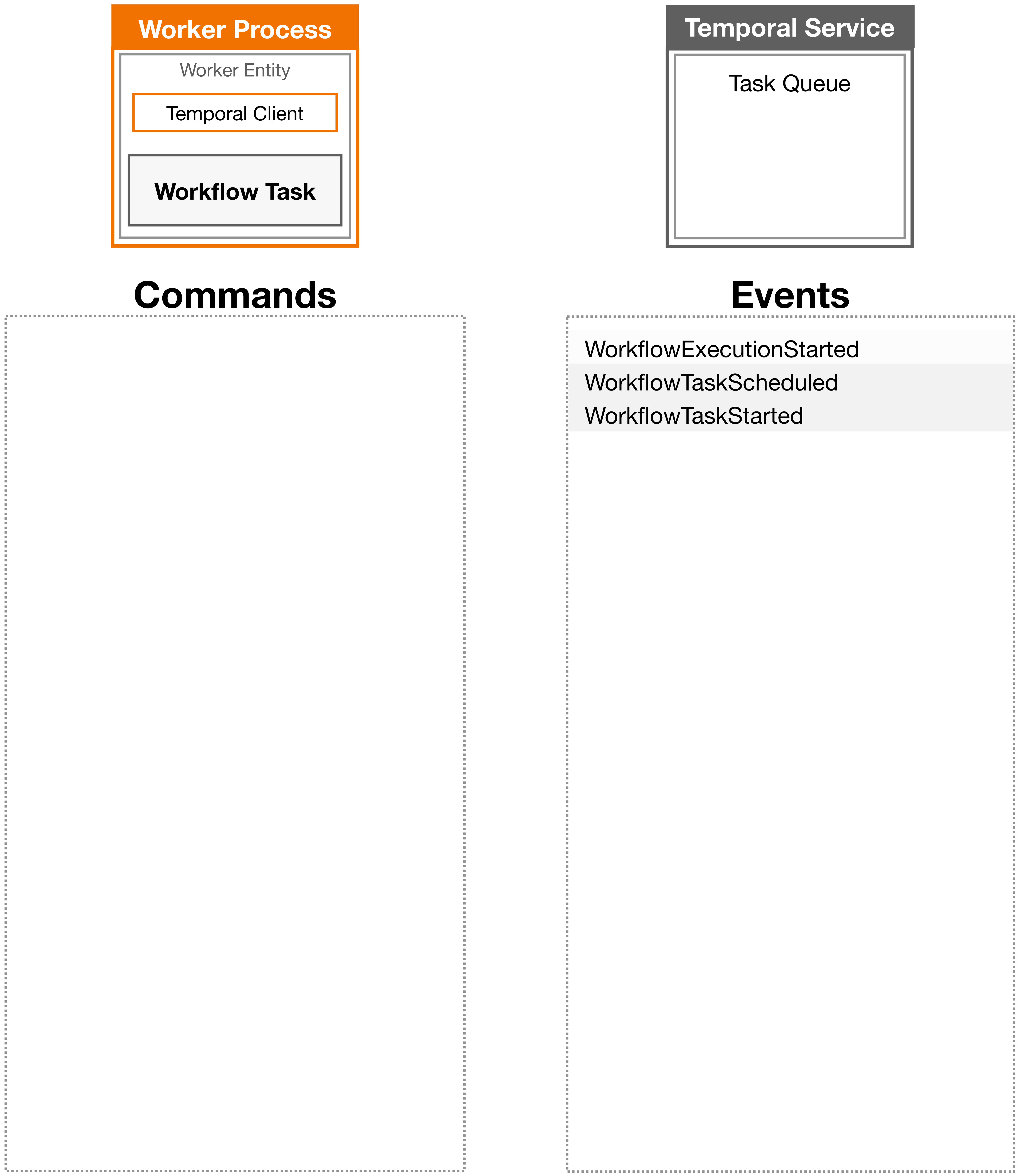


## Commands

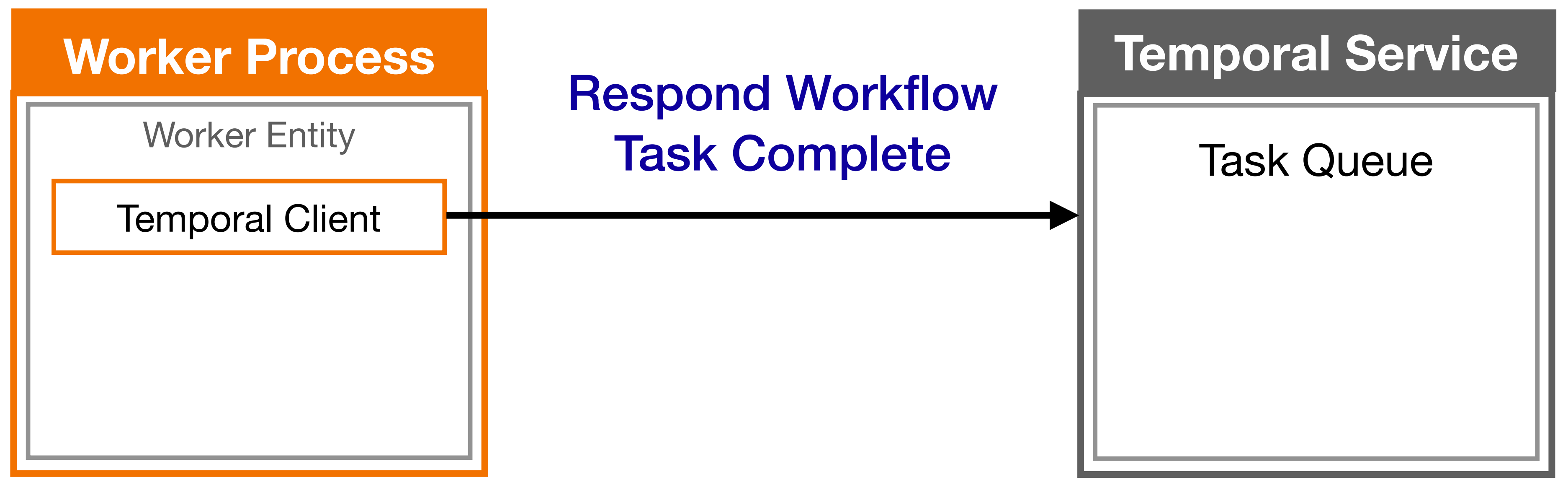
## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

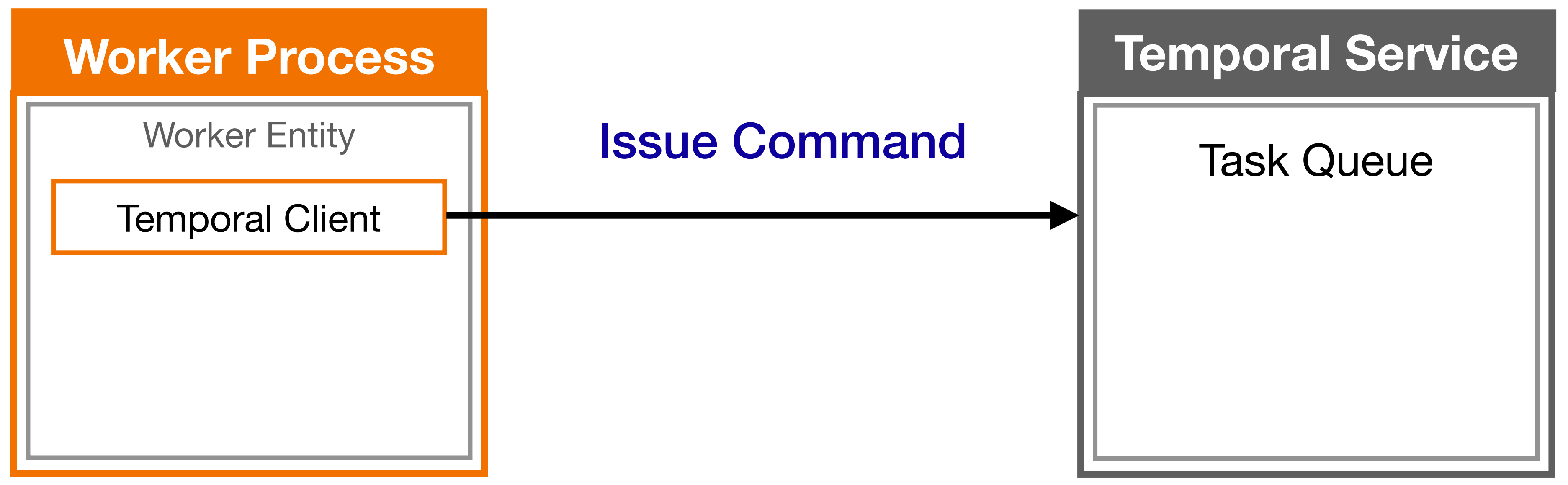


**Commands**

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
**WorkflowTaskCompleted**

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

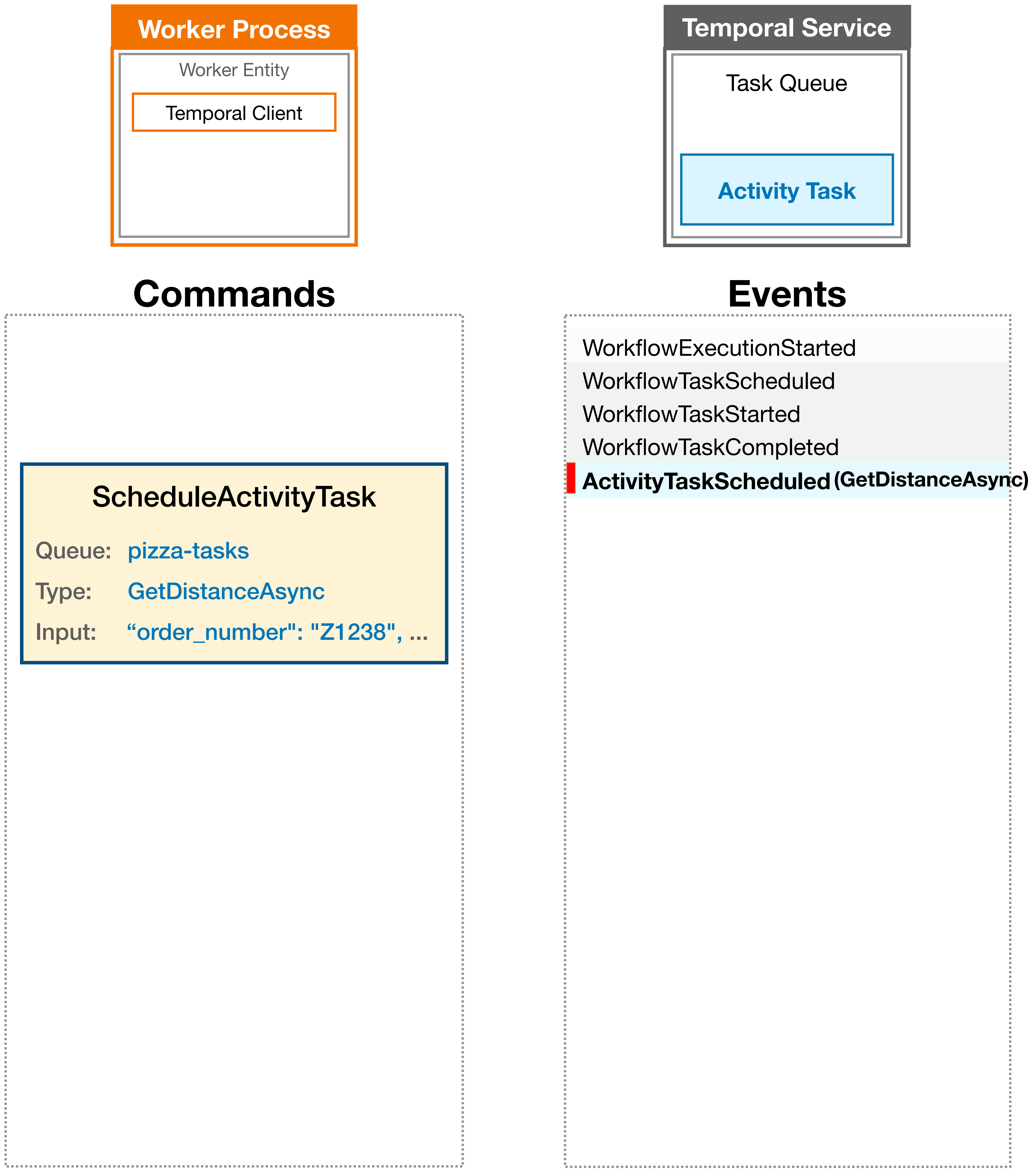
**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

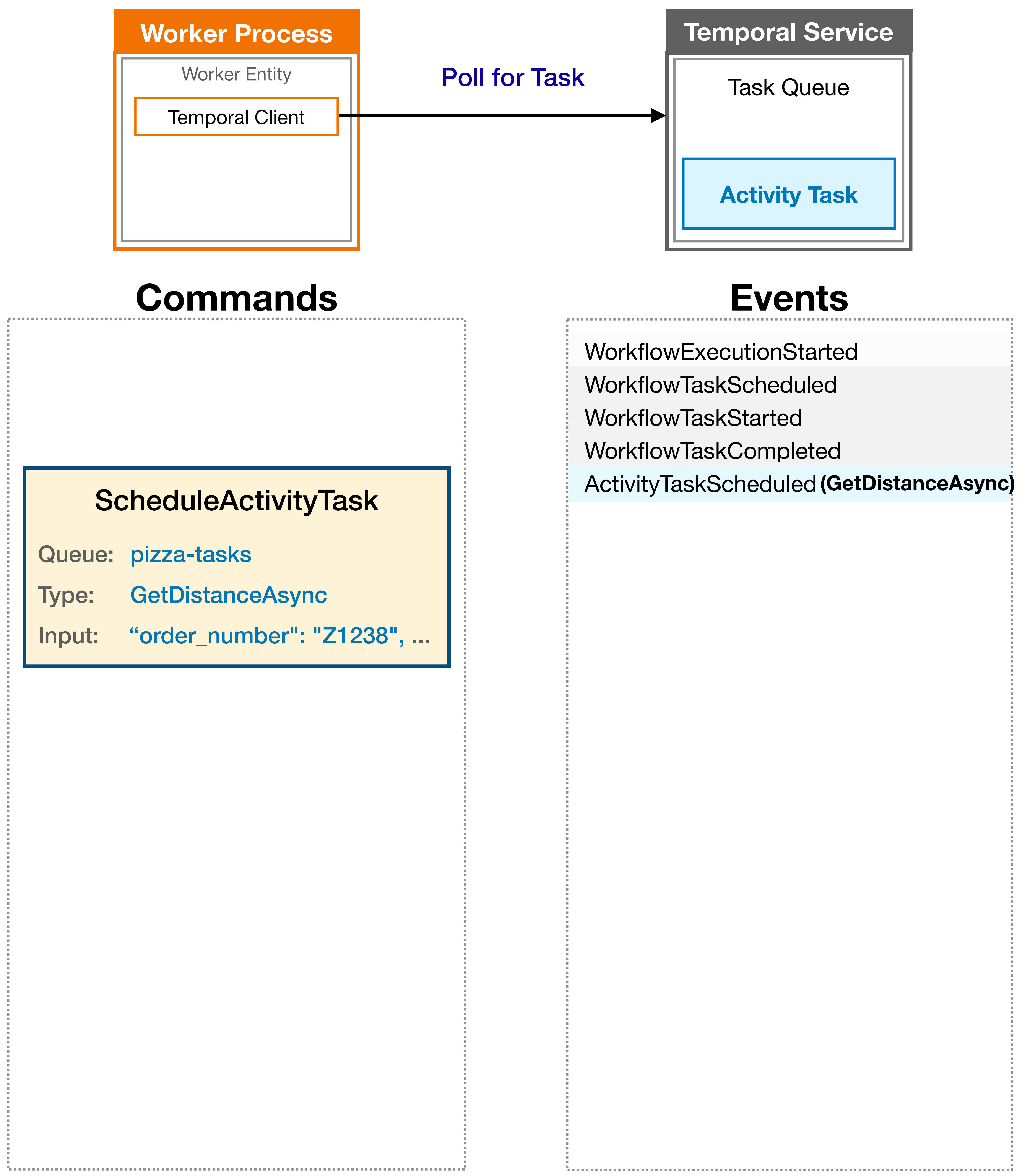
**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted

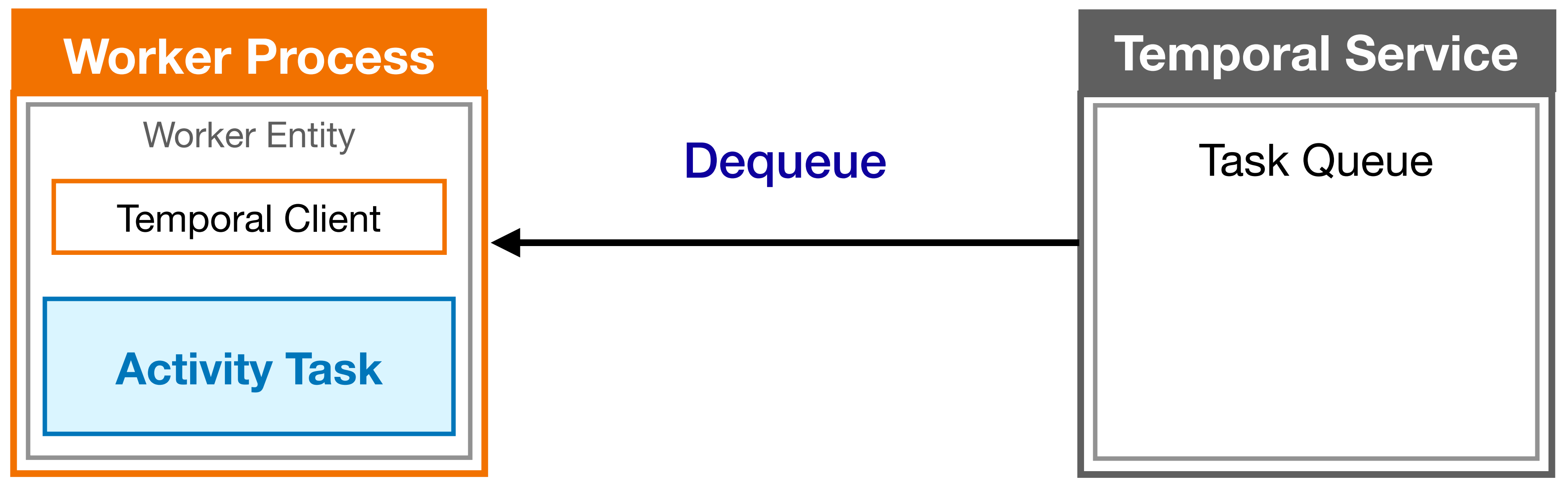
```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

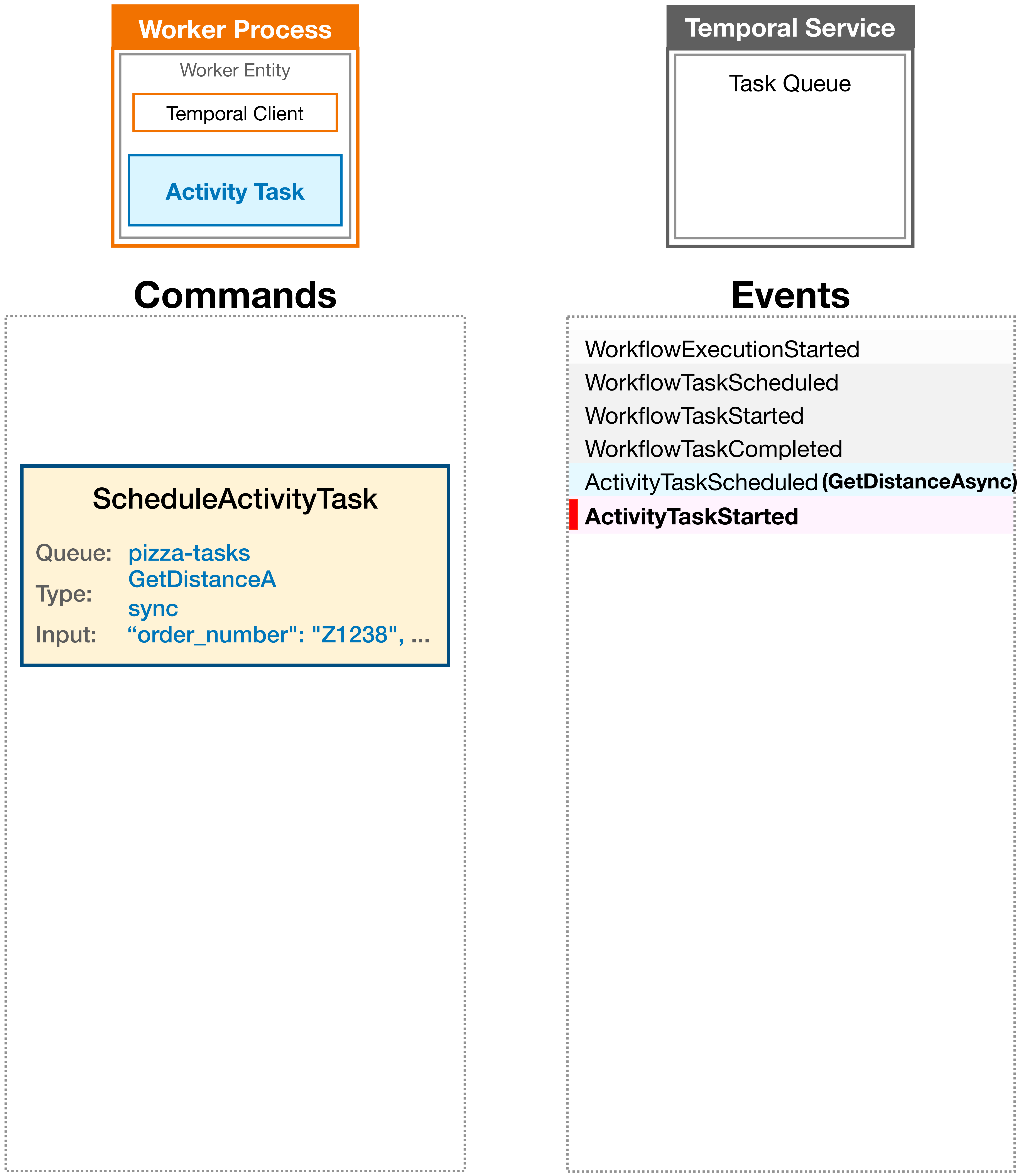
**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

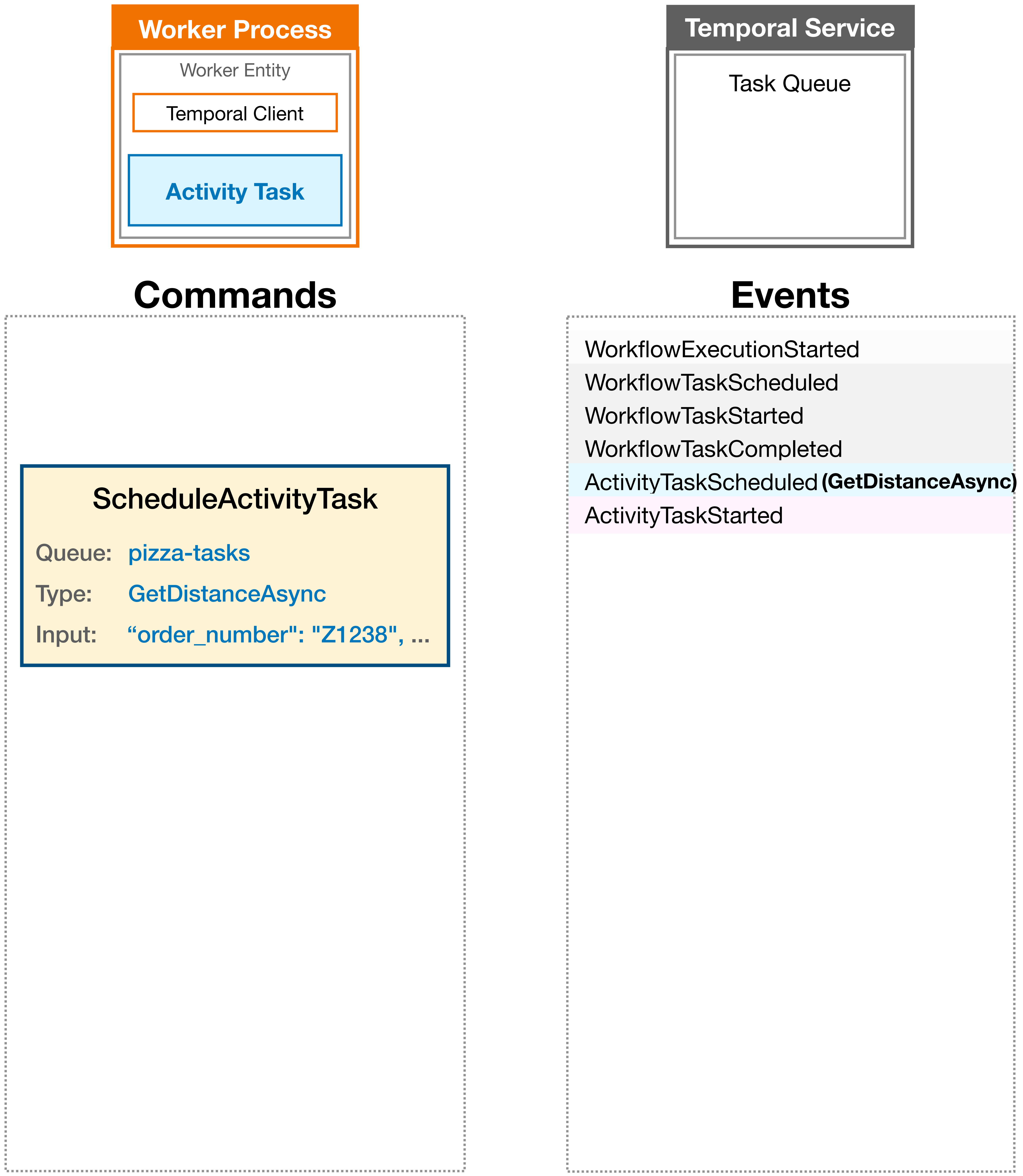
**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

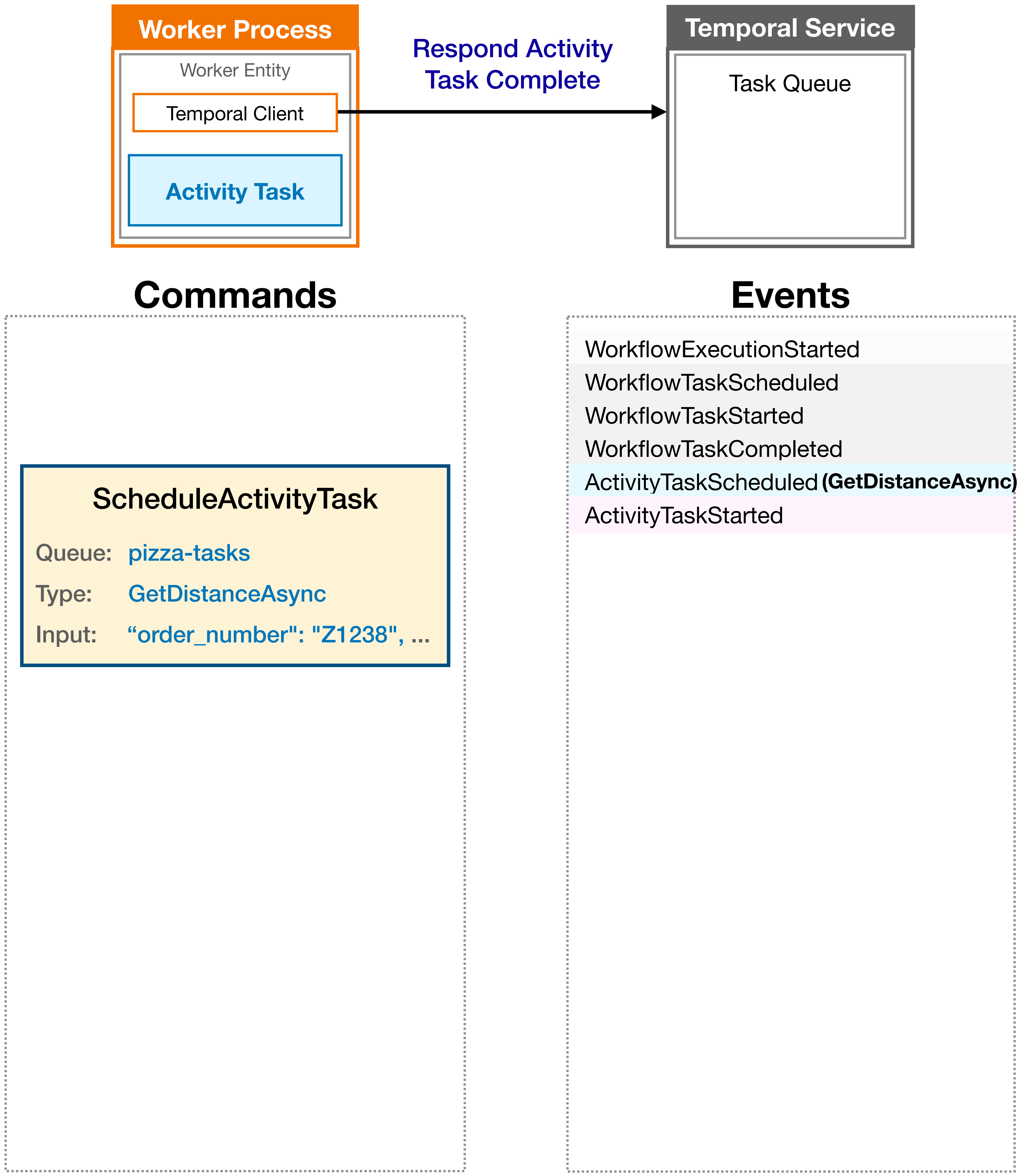


**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

Worker Process
Worker Entity
Temporal Client
Temporal Service
Task Queue

## Commands

**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: "order_number": "Z1238", ...

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
**ActivityTaskCompleted** **(distance=15)**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

Worker Process
Worker Entity
Temporal Client
Temporal Service
Task Queue
Workflow Task

## Commands

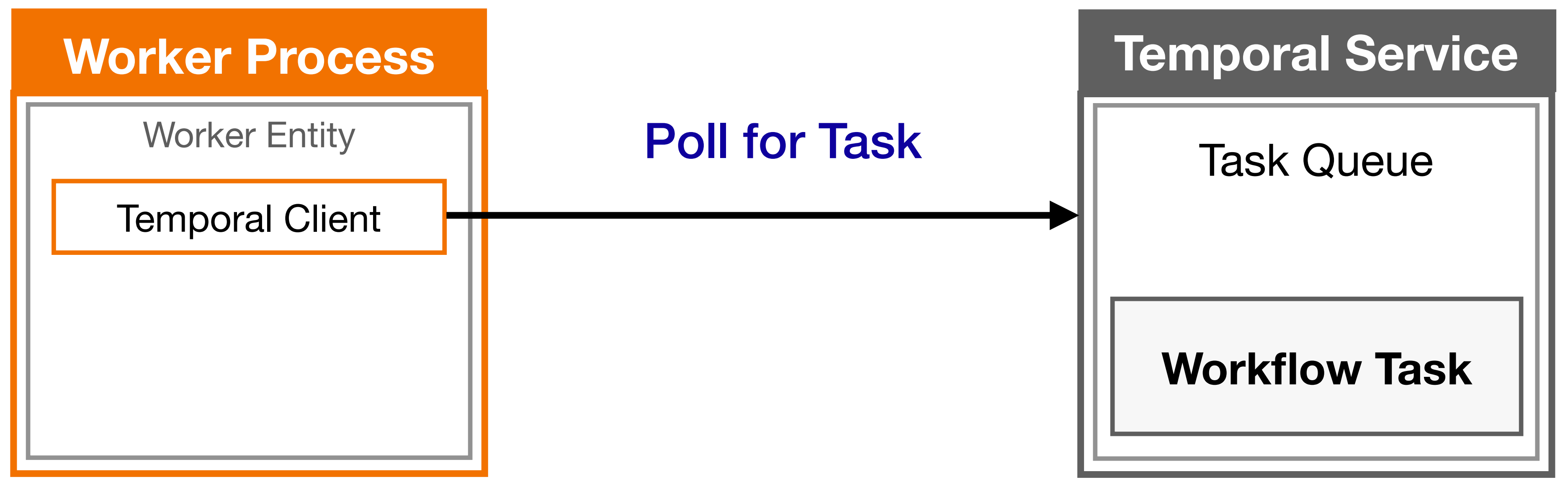
**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
**WorkflowTaskScheduled**

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

Queue: pizza-tasks

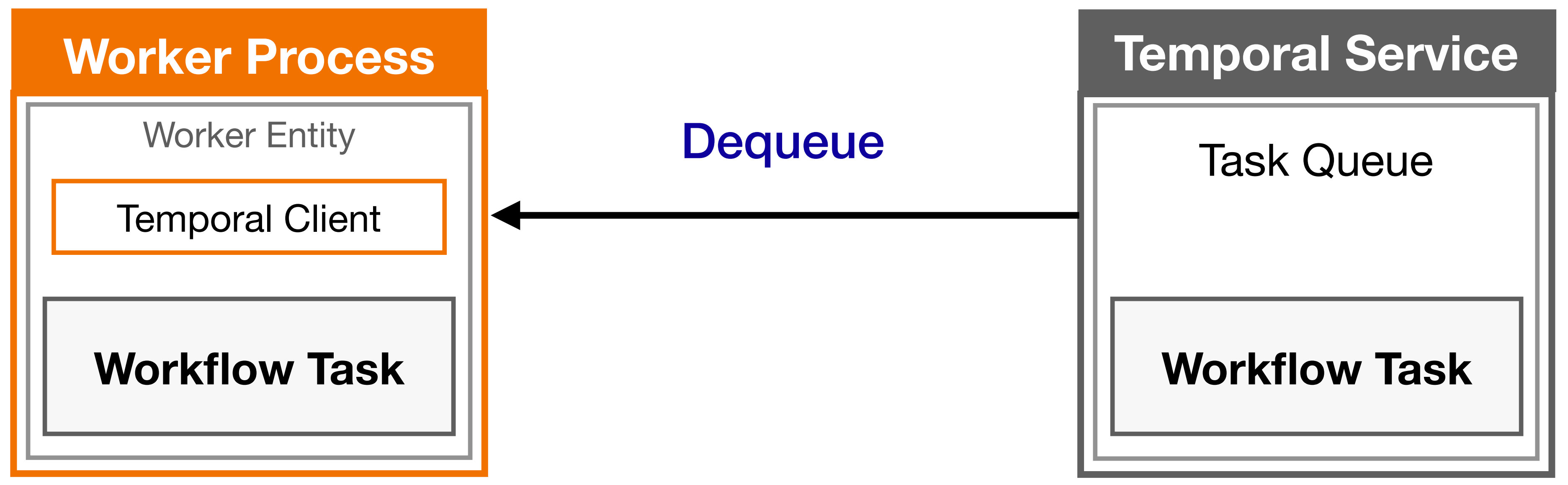
Type: GetDistanceAsync

Input: “order_number": "Z1238", ...

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

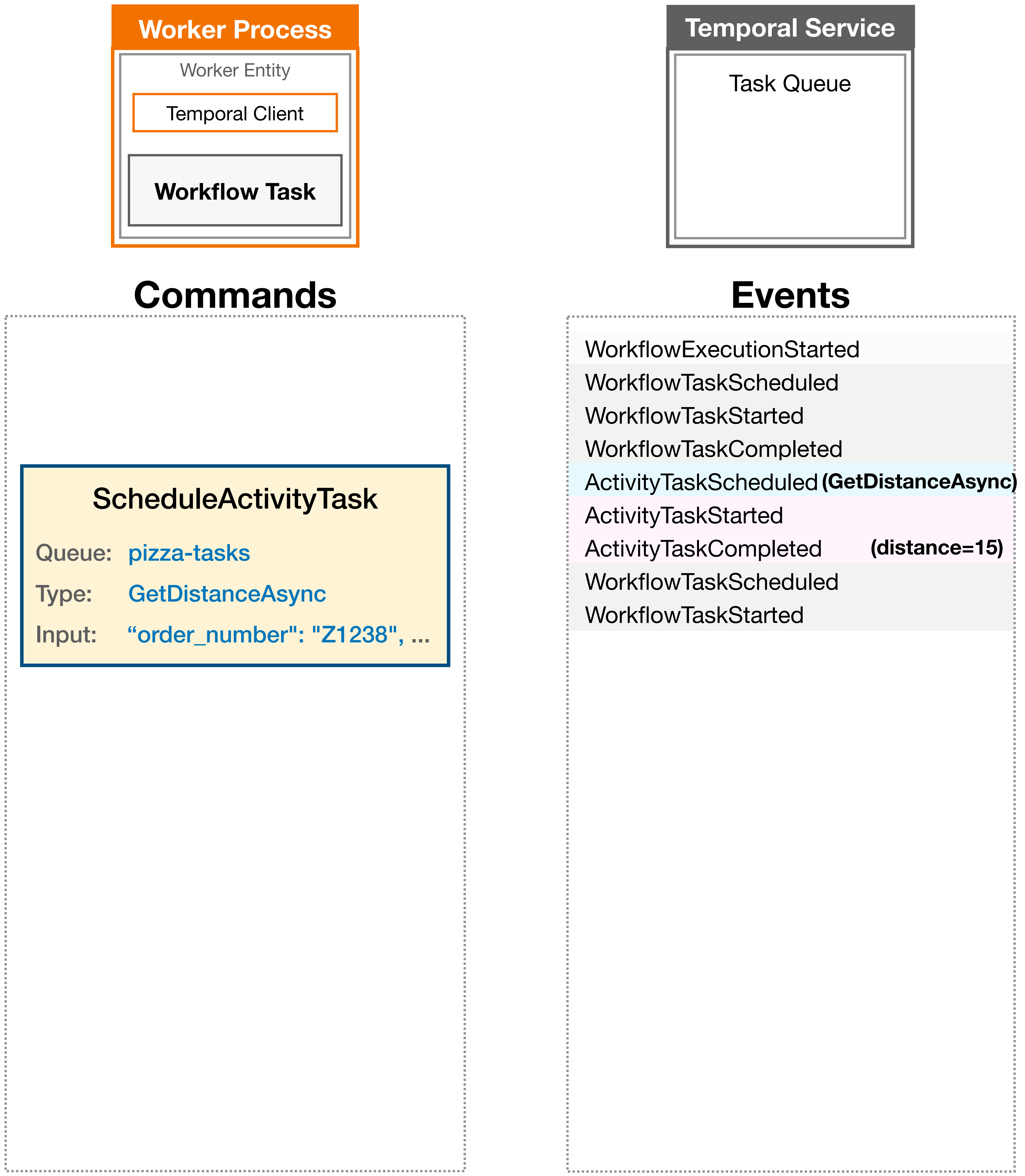
**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

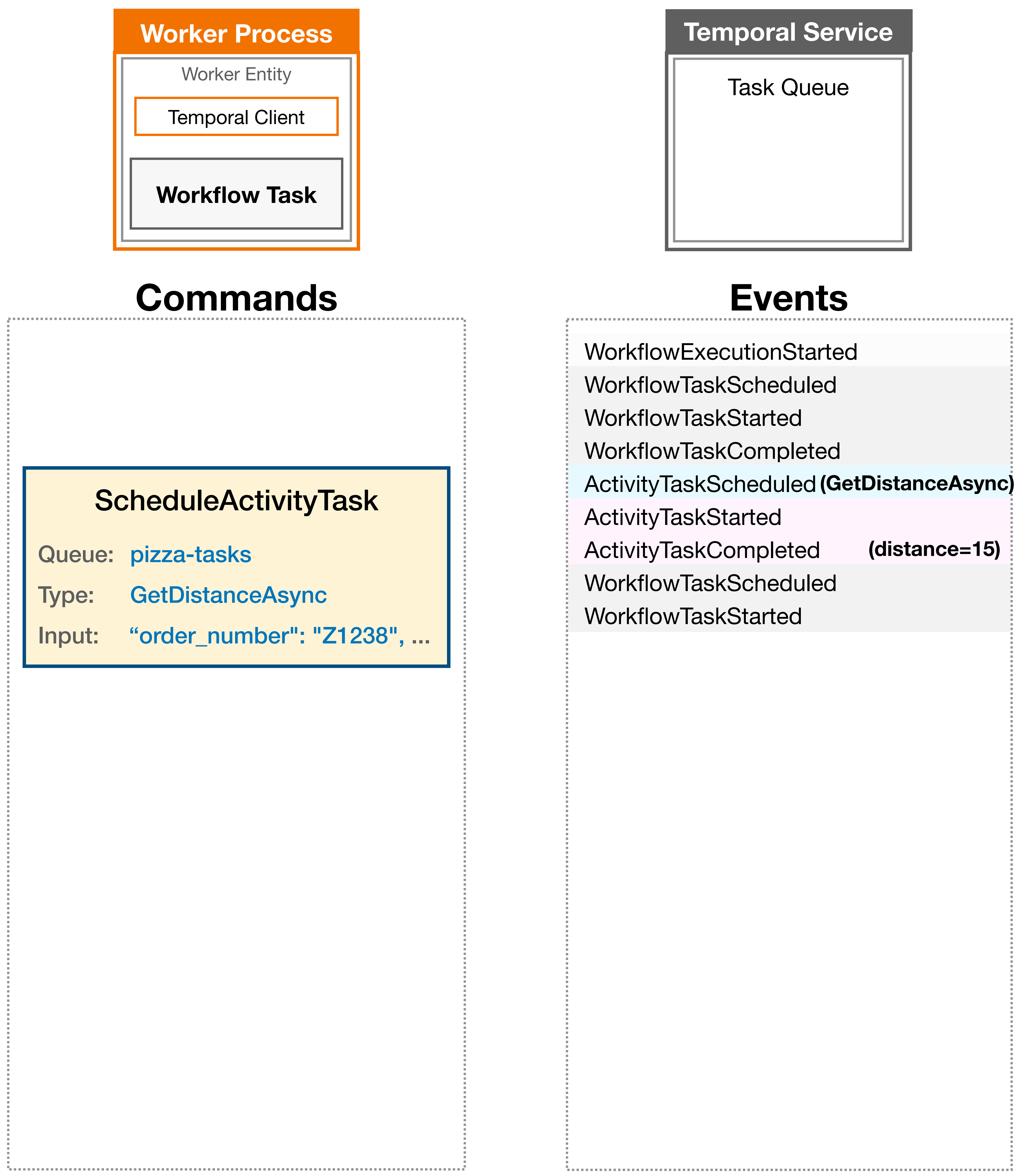
**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
**WorkflowTaskStarted**

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

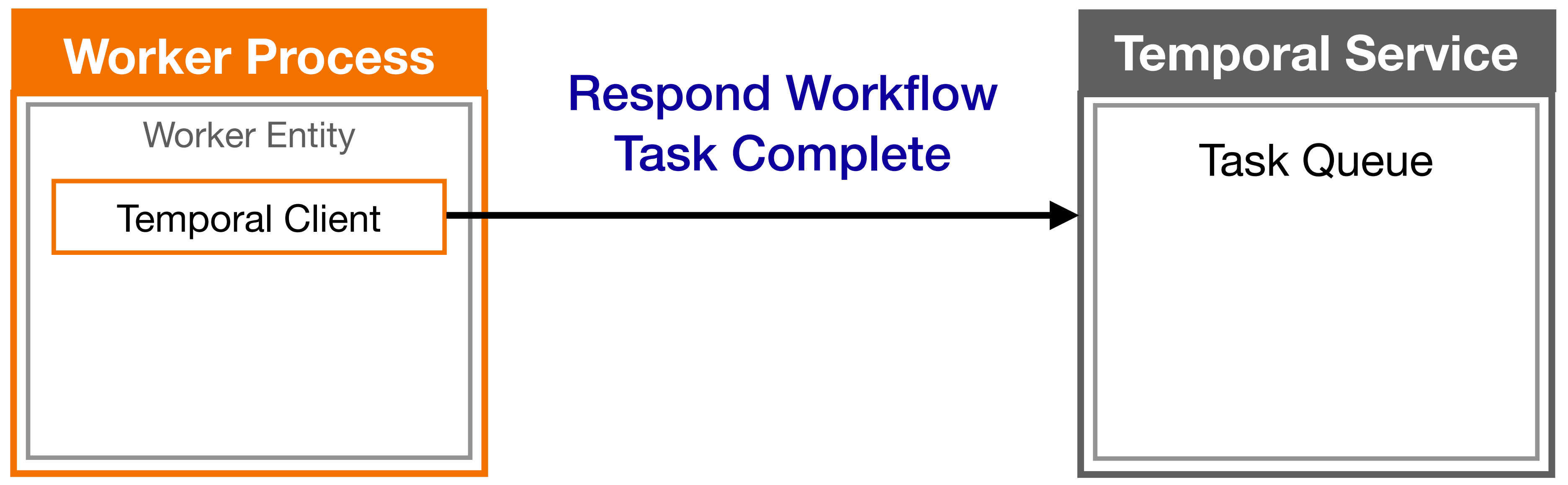


**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted        (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
```

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

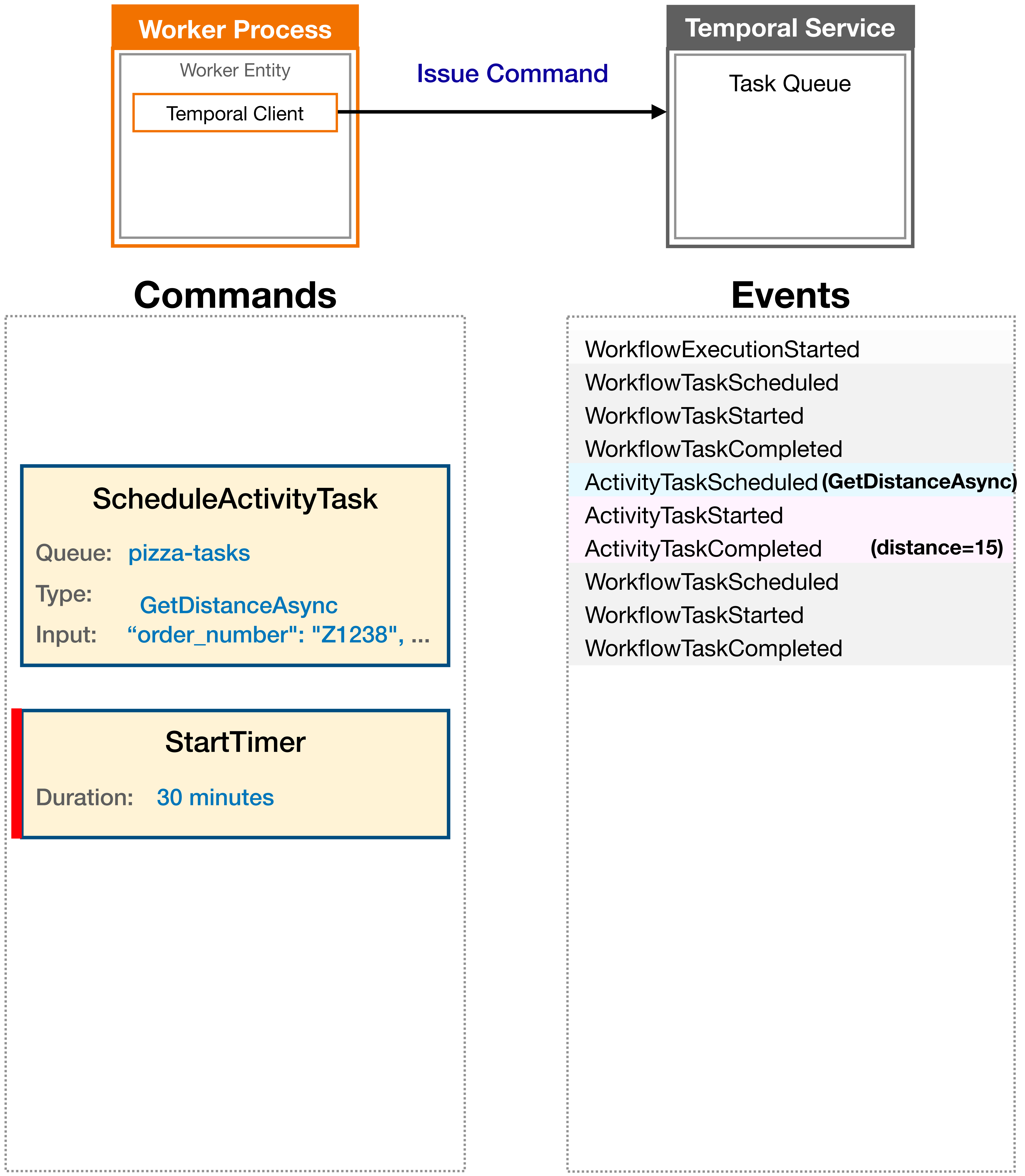
**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

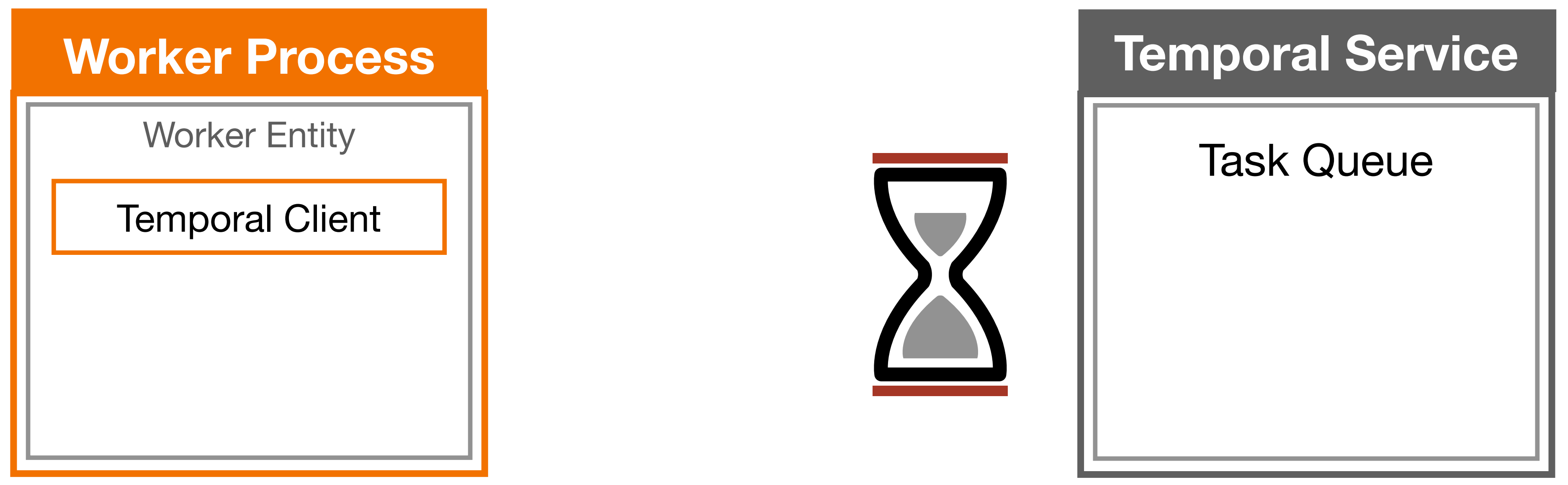
## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
**WorkflowTaskCompleted**

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

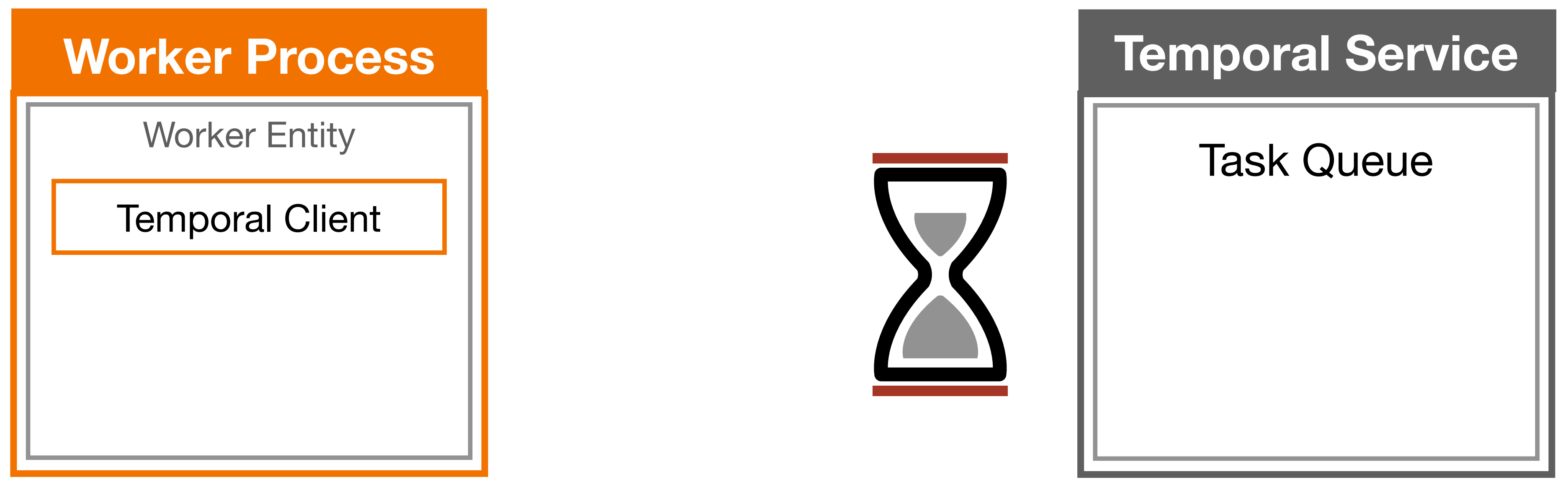
**StartTimer**

| | |
|---|---|
| Duration: | 30 minutes |

## Events

| | |
|---|---|
| WorkflowExecutionStarted | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| ActivityTaskScheduled | **(GetDistanceAsync)** |
| ActivityTaskStarted | |
| ActivityTaskCompleted | **(distance=15)** |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| **TimerStarted** | **(30 Minutes)** |

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

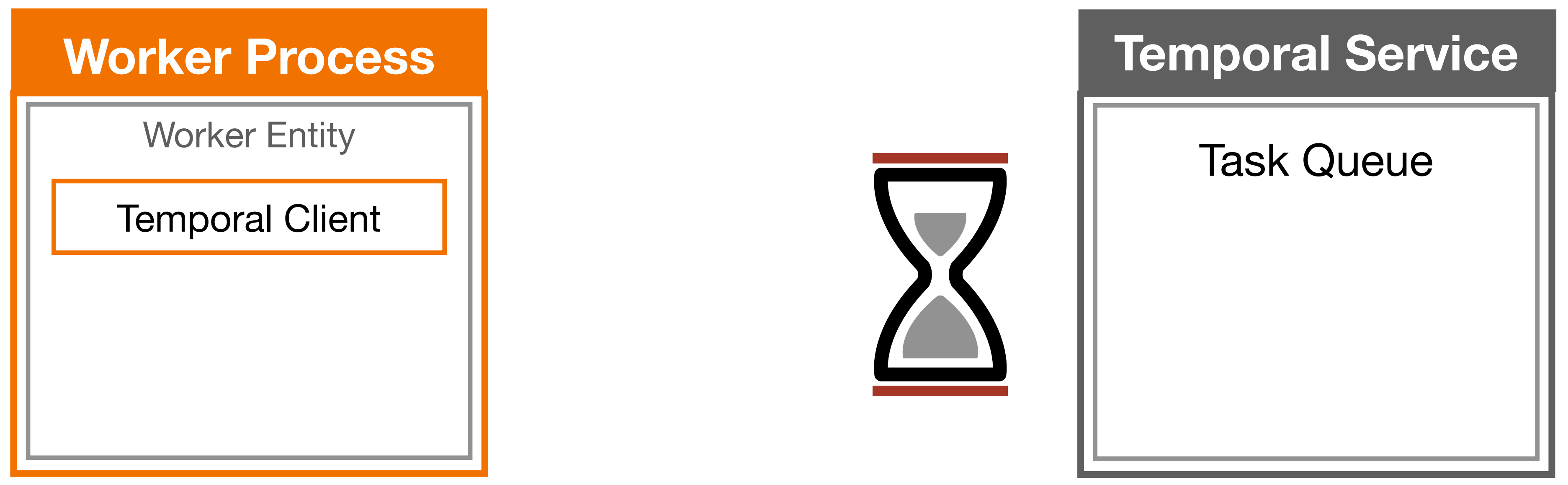
**StartTimer**

Duration: 30 minutes

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

**StartTimer**

Duration: 30 minutes

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
**TimerFired**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

Worker Process
Worker Entity
Temporal Client
Temporal Service
Task Queue
Workflow Task

## Commands

**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

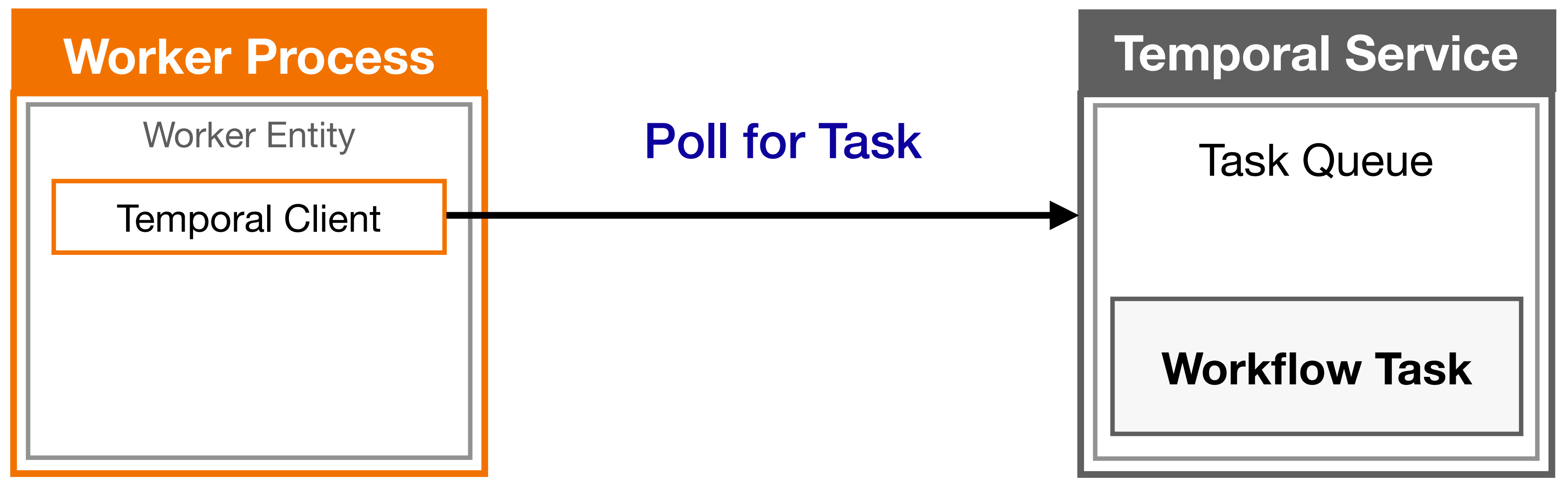
**StartTimer**

Duration: 30 minutes

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
**WorkflowTaskScheduled**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

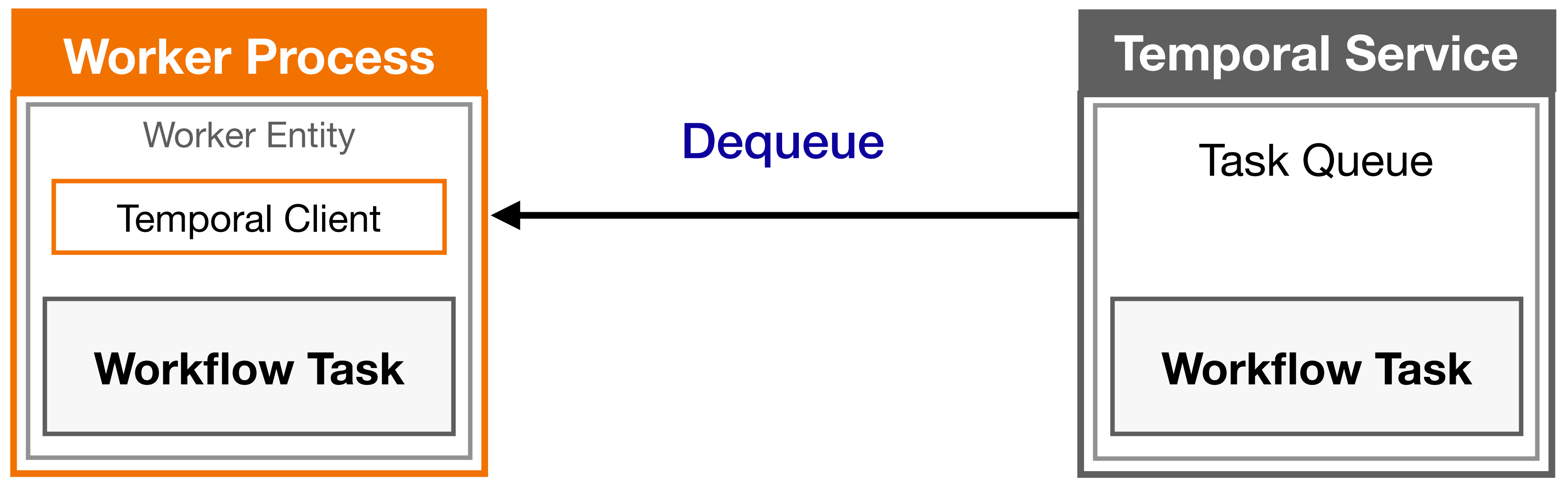
**StartTimer**

Duration: 30 minutes

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

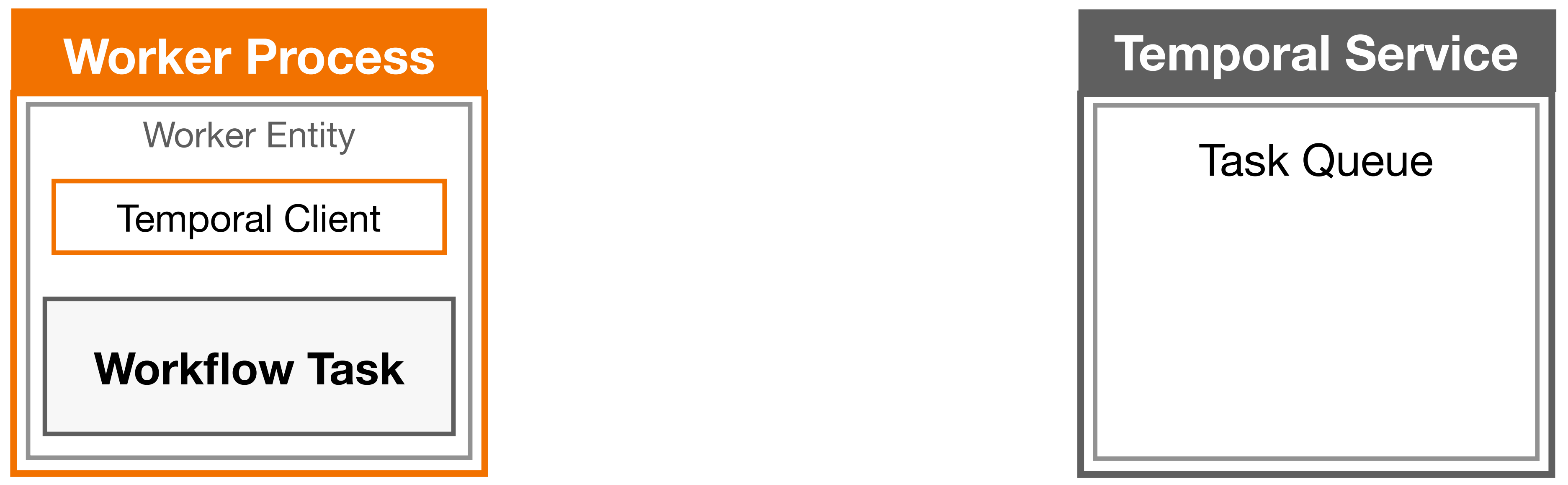
**StartTimer**

| | |
|---|---|
| Duration: | 30 minutes |

## Events

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted        (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted                  (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
```

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

**StartTimer**

Duration: 30 minutes

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

**Worker crashes here**

Worker Process
Worker Entity
Temporal Client
Workflow Task
Temporal Service
Task Queue

## Commands

**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

**StartTimer**

Duration: 30 minutes

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

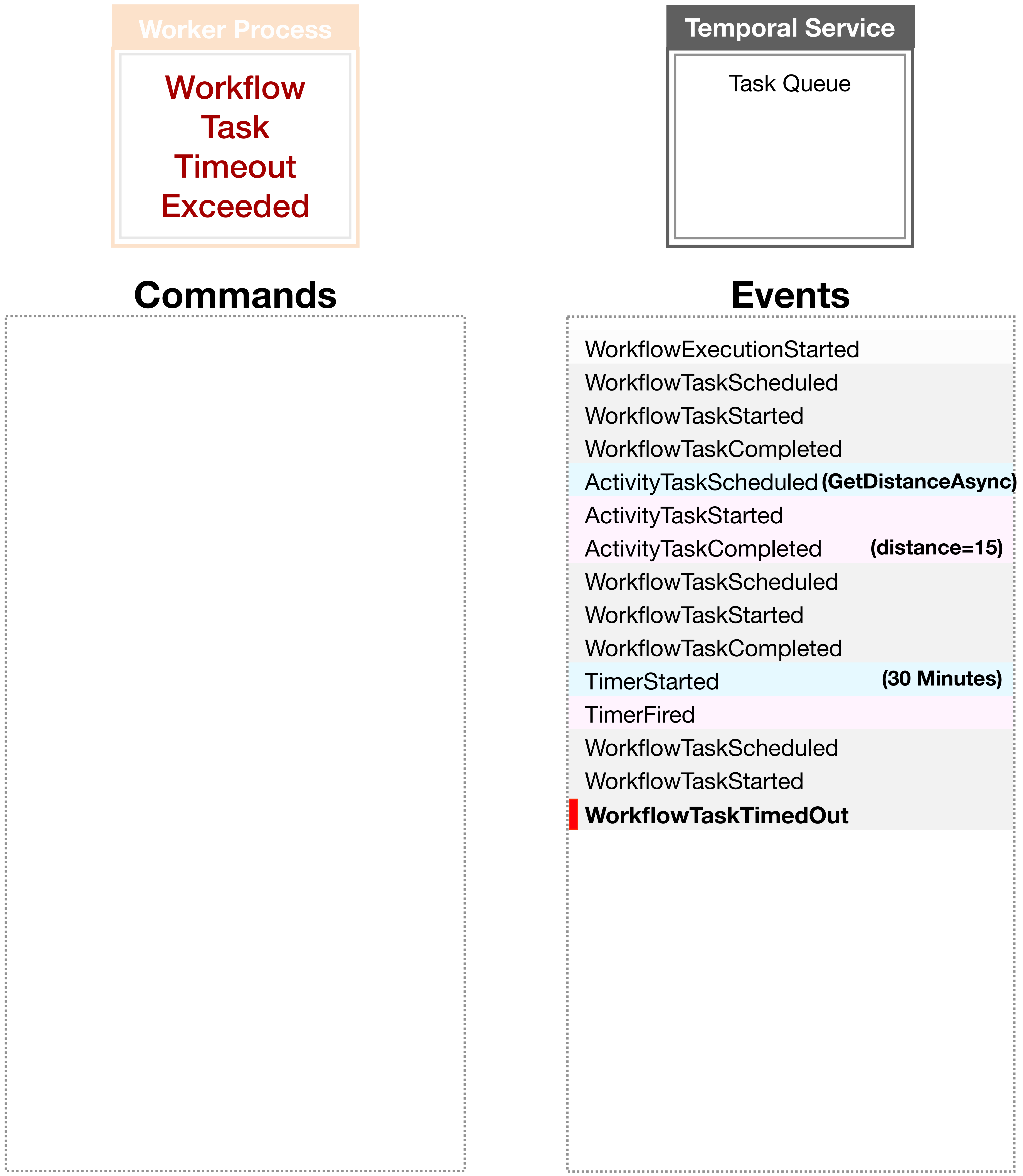
Worker Process
Worker Entity
Temporal Client
Workflow Task
Temporal Service
Task Queue

## Commands

## Events

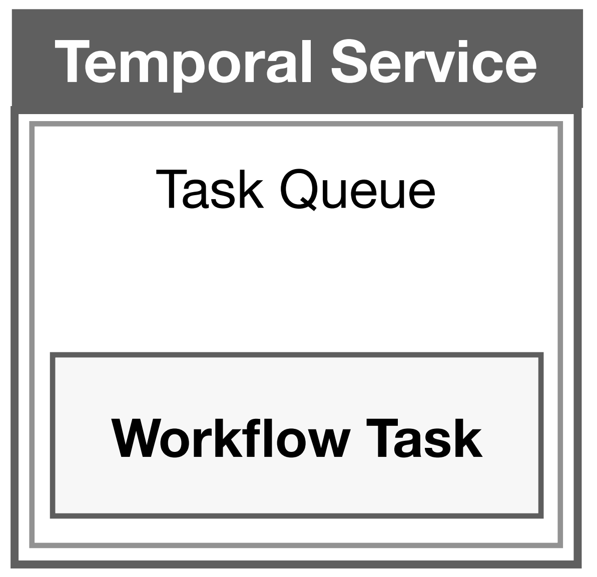
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted       (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted                (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
```

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

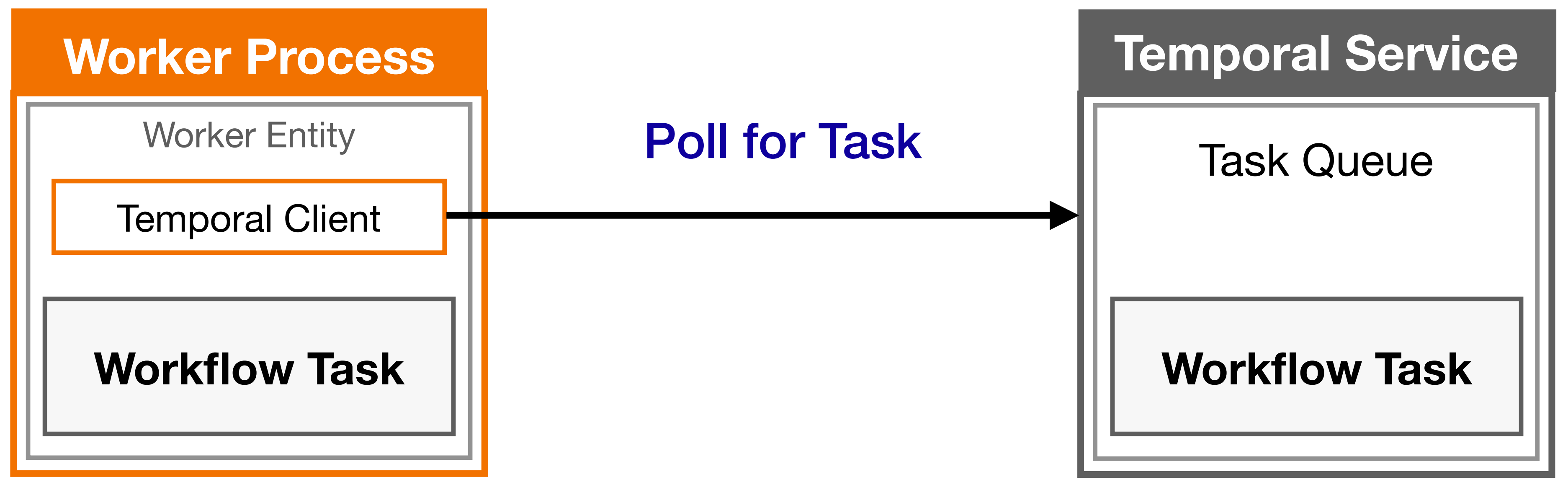


## Commands

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
**WorkflowTaskScheduled**

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

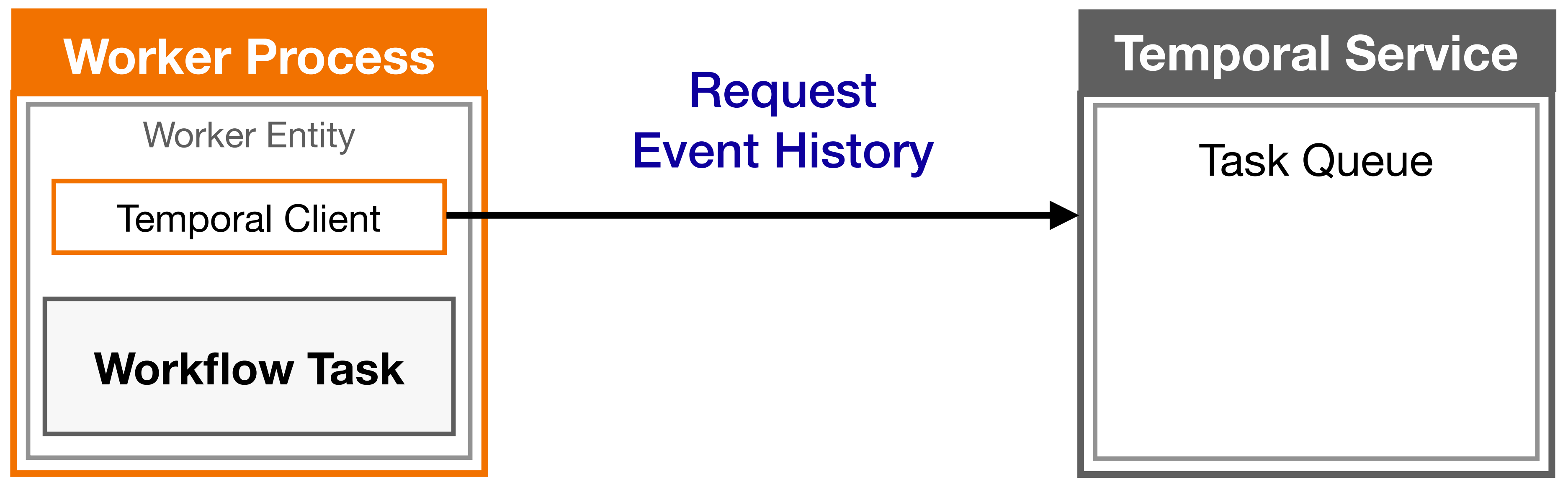


**Commands**

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
**WorkflowTaskStarted**

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

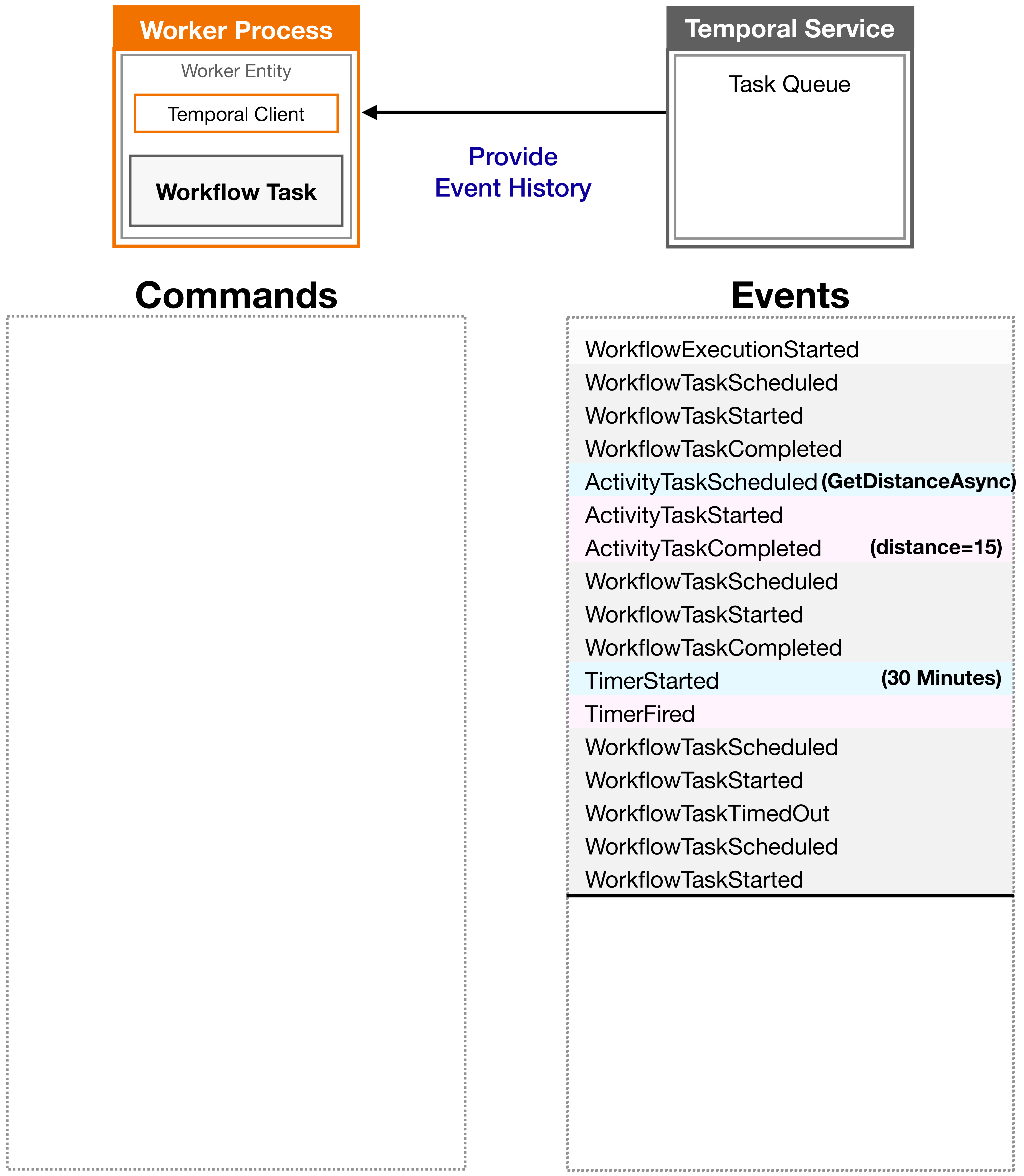


**Commands**

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

Worker Process
Worker Entity
Temporal Client
Workflow Task
Temporal Service
Task Queue

**Commands**

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

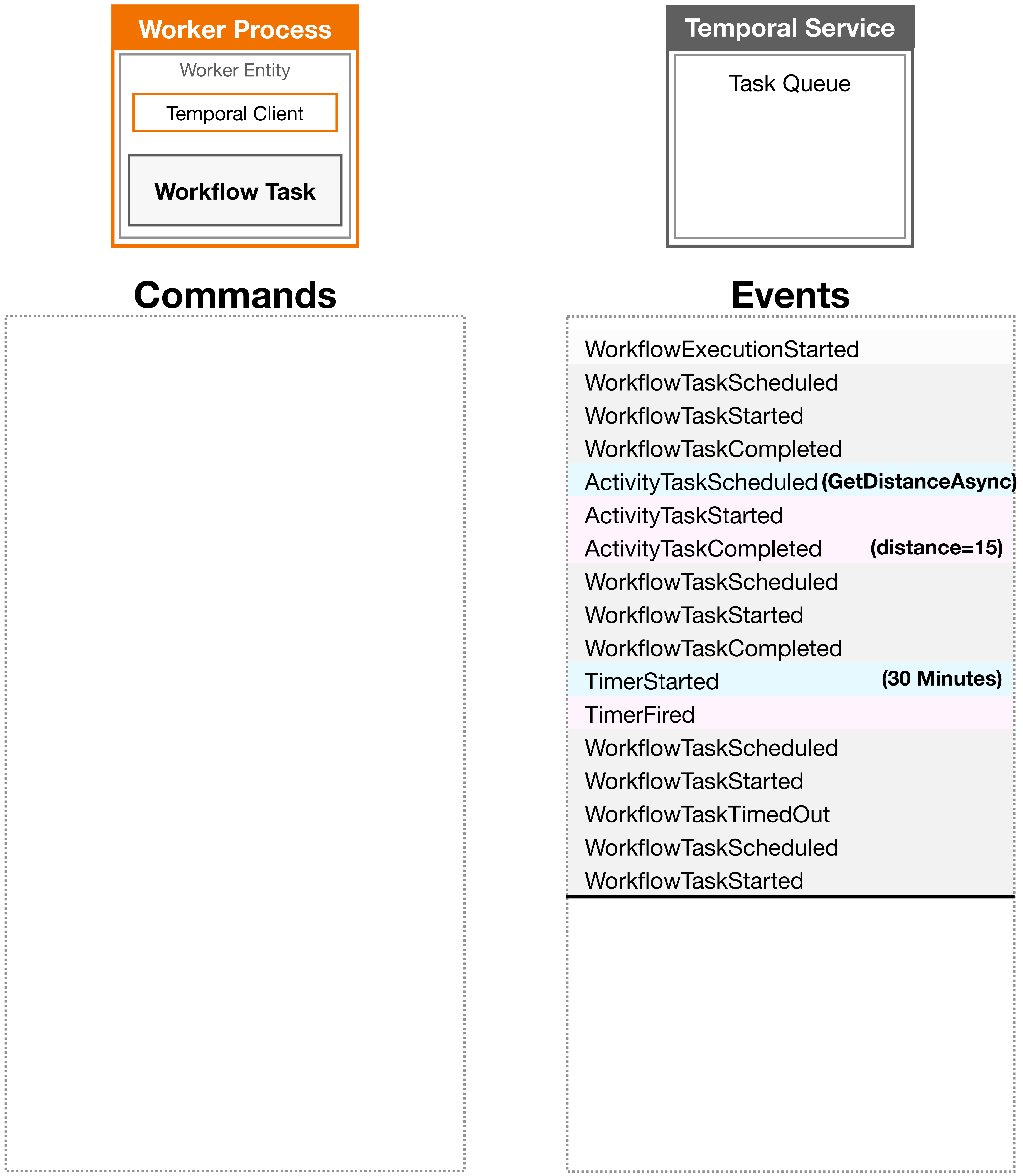
Worker Process
Worker Entity
Temporal Client
Workflow Task
Temporal Service
Task Queue

## Commands

## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```python
import asyncio
from datetime import timedelta
from temporalio import workflow
from temporalio.exceptions import ApplicationError

with workflow.unsafe.imports_passed_through():
    from activities import PizzaOrderActivities
    from shared import Bill, OrderConfirmation, PizzaOrder


@workflow.defn
class PizzaOrderWorkflow:
    @workflow.run
    async def order_pizza(self, order: PizzaOrder) -> OrderConfirmation:
        total_price = sum(pizza.price for pizza in order.items)

        workflow.logger.info(f"Calculated cost of order: {total_price}")

        distance = await workflow.execute_activity_method(
            PizzaOrderActivities.GetDistanceAsync,
            order.address,
            start_to_close_timeout=timedelta(seconds=5),
        )

        if order.is_delivery and distance.kilometers > 25:
            error_message = "customer lives outside the service area"
            raise ApplicationError(error_message)

        # Wait 30 minutes before billing the customer
        await asyncio.sleep(timedelta(minutes=1).total_seconds())

        bill = Bill(
            customer_id=order.customer.customer_id,
            order_number=order.order_number,
            description="Pizza order",
            amount=total_price,
        )

        confirmation = await workflow.execute_activity_method(
            PizzaOrderActivities.SendBillAsync,
            bill,
            start_to_close_timeout=timedelta(seconds=5),
        )

        return confirmation
```



```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted      (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted               (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
```

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

Worker Process
Worker Entity
Temporal Client
Workflow Task
Temporal Service
Task Queue

**Commands**

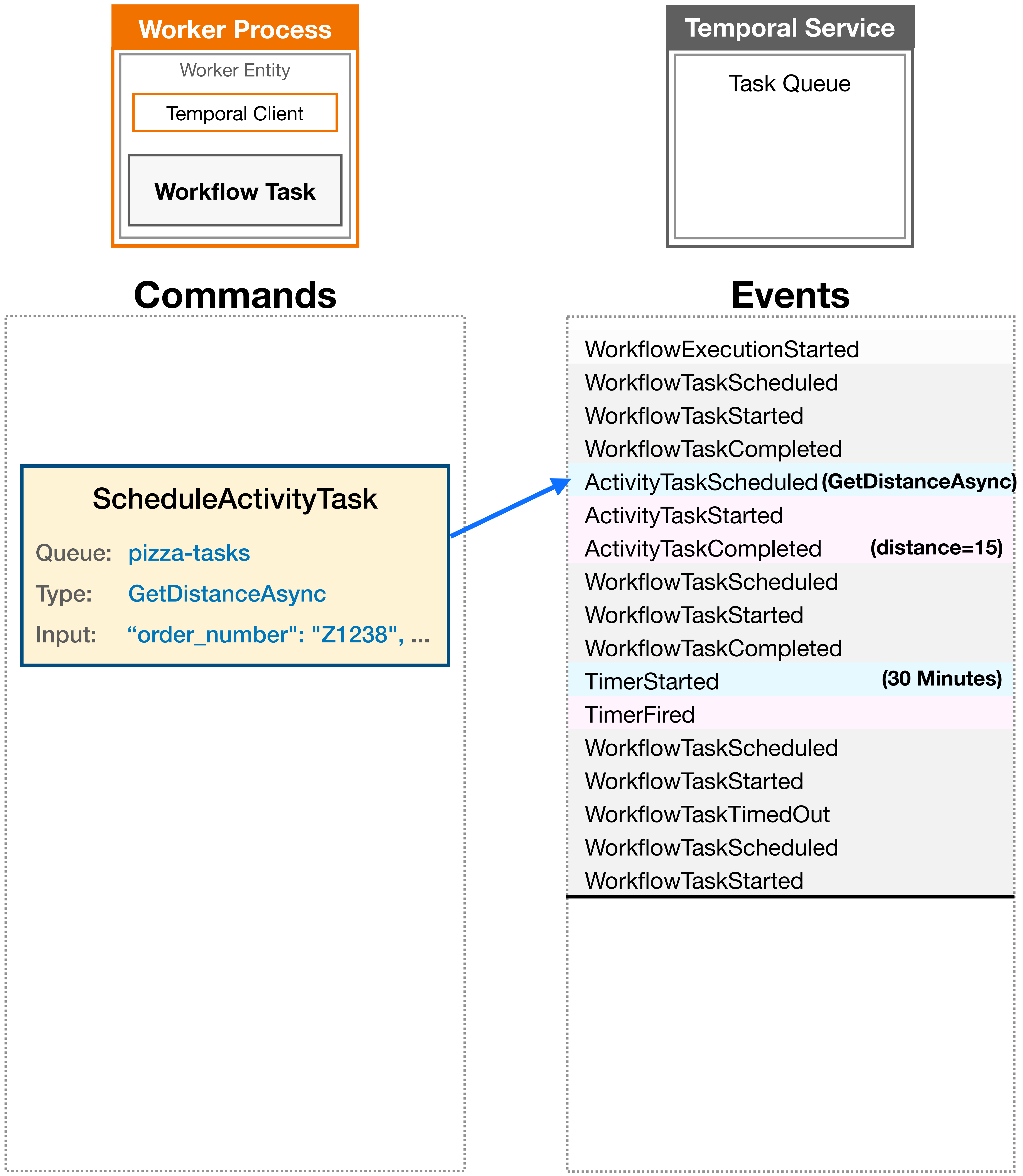
**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | "order_number": "Z1238", ... |

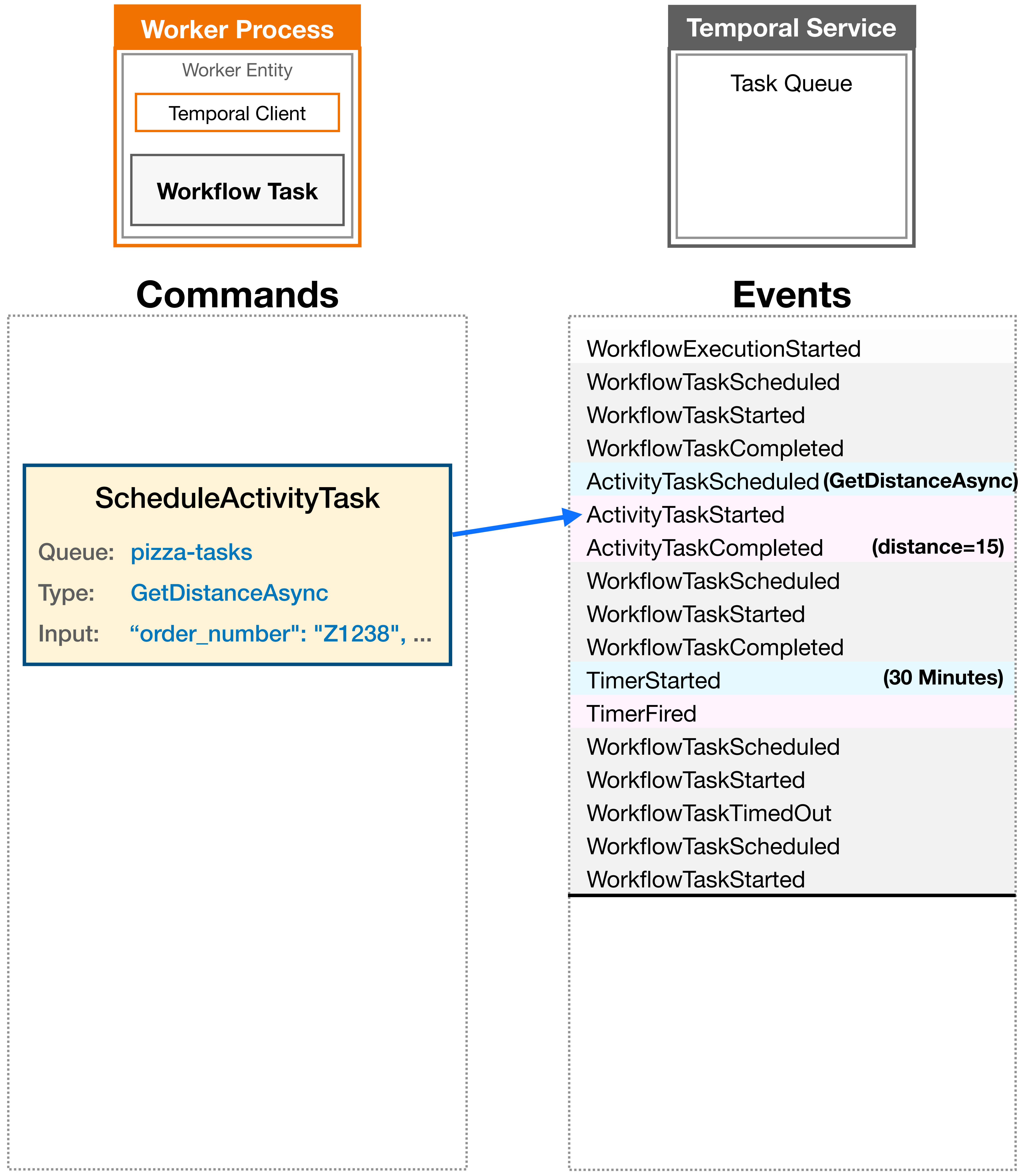
**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

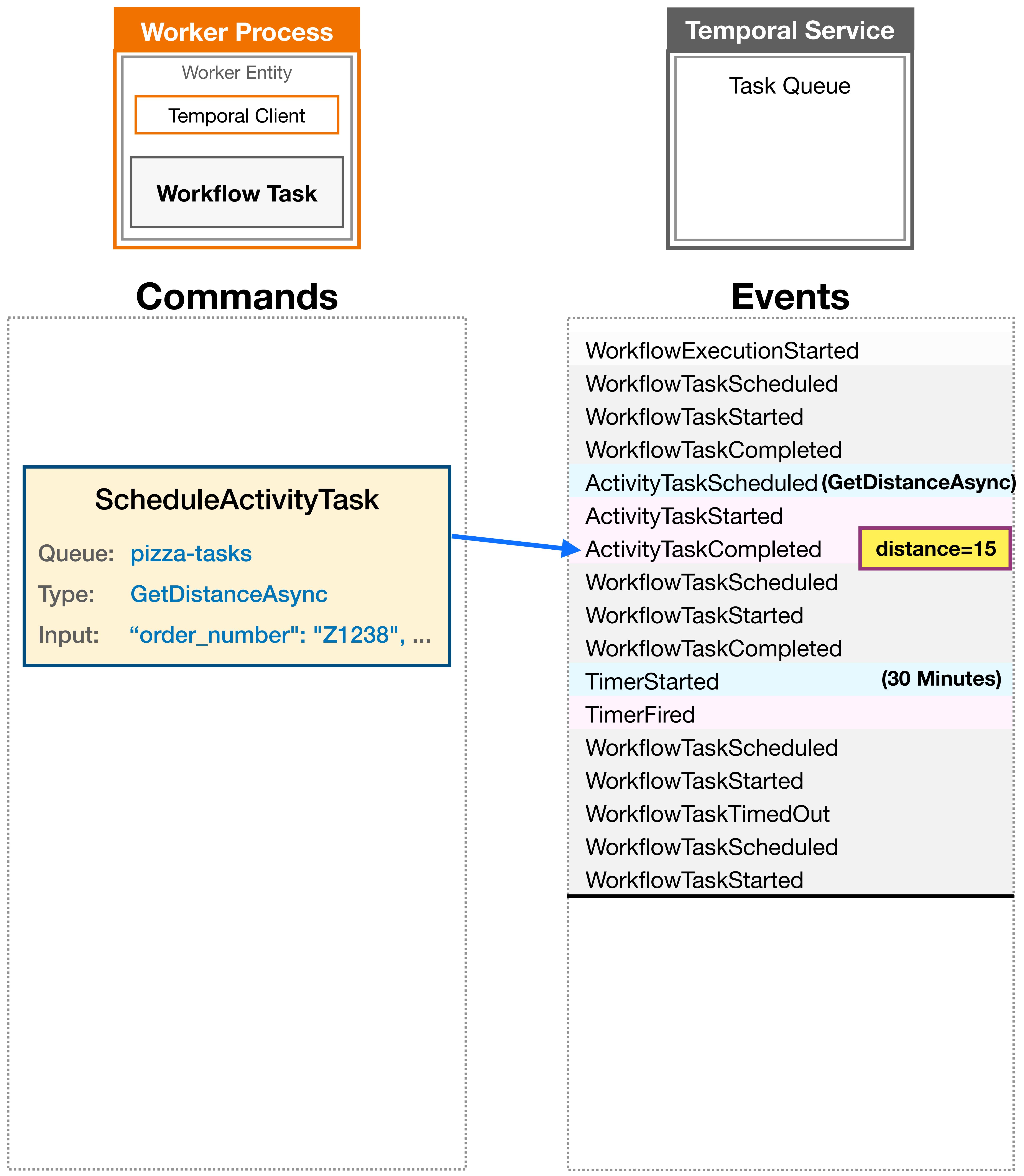


**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

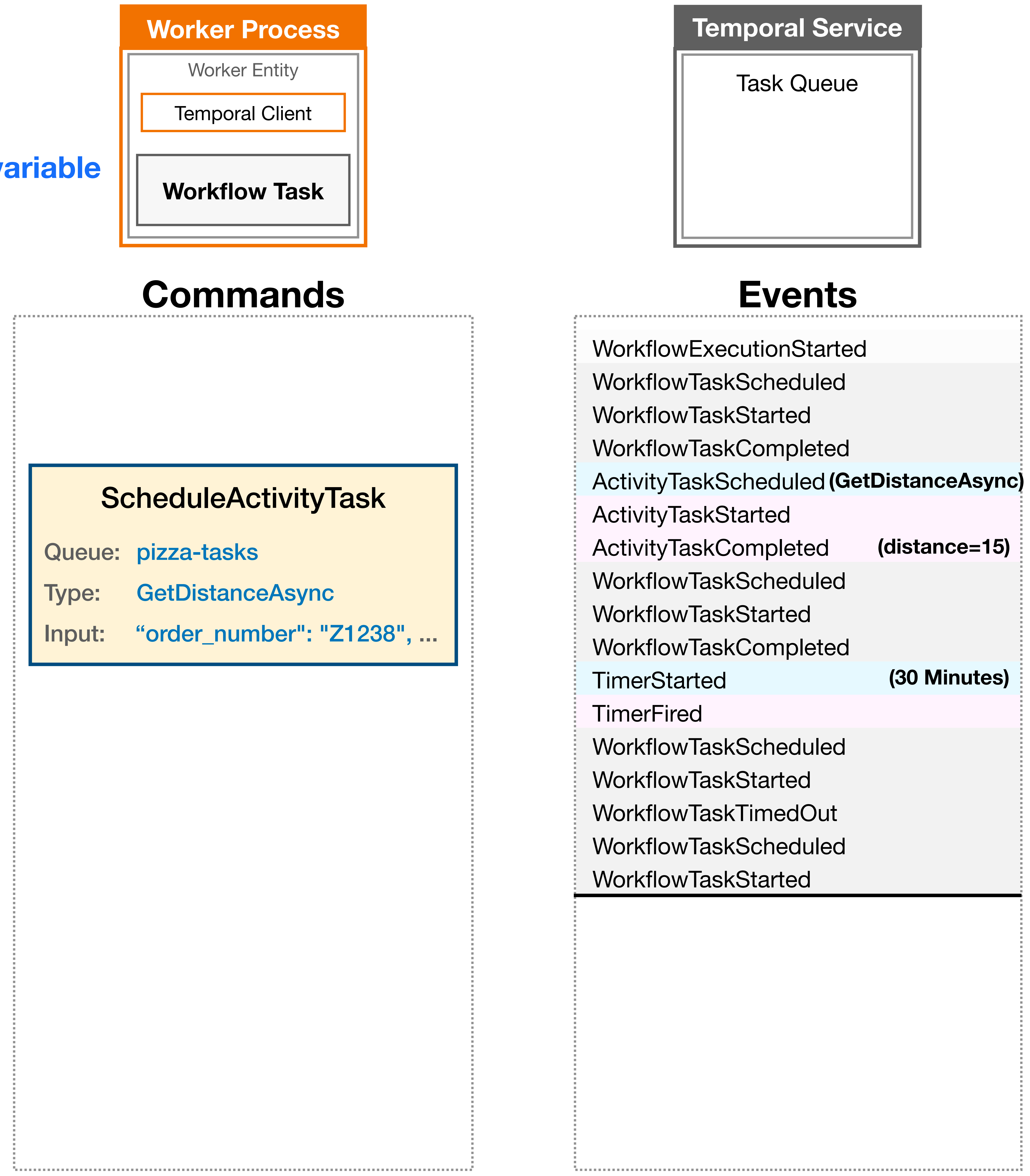
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

**Worker assigns 15 to this variable**

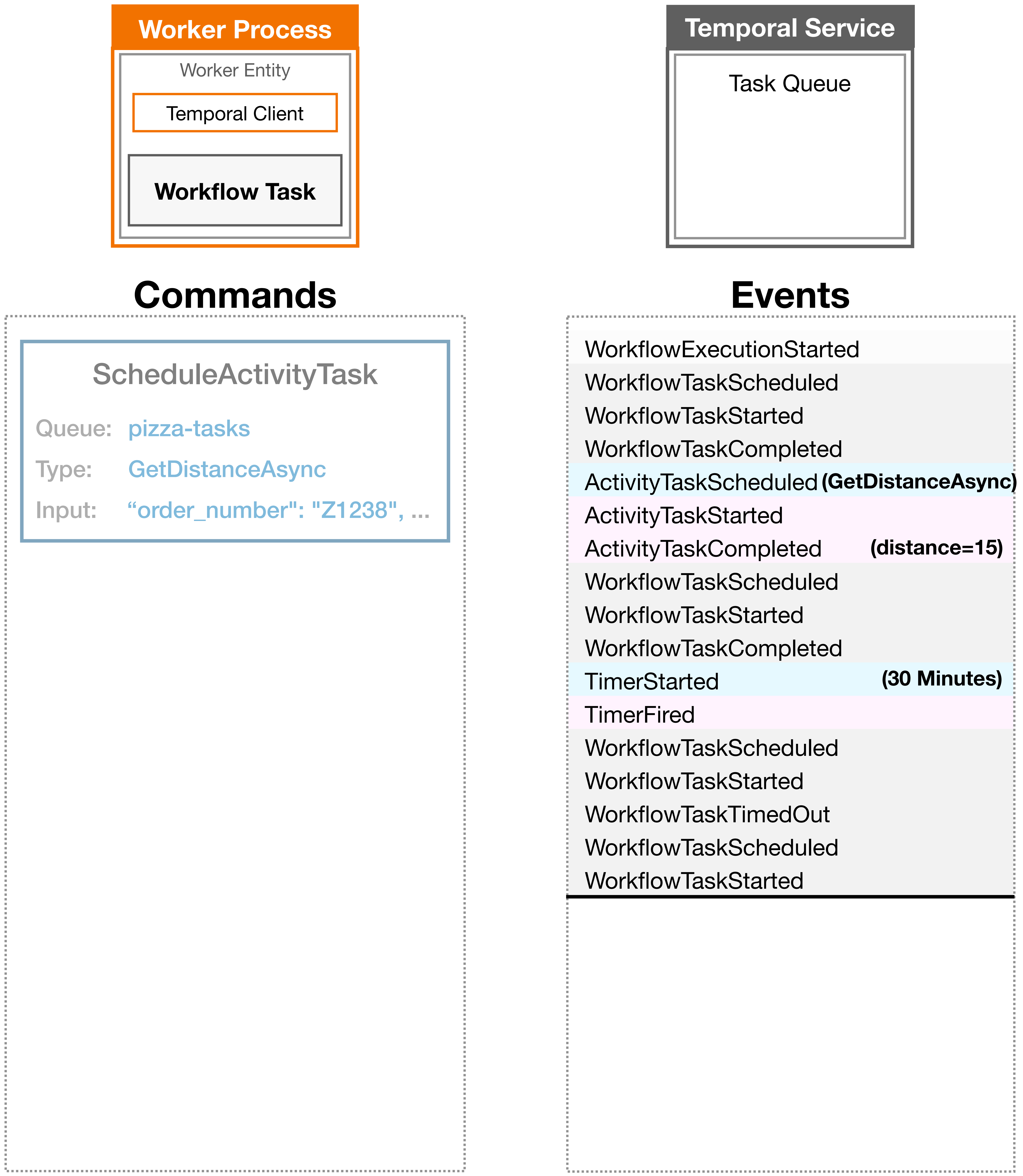


**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

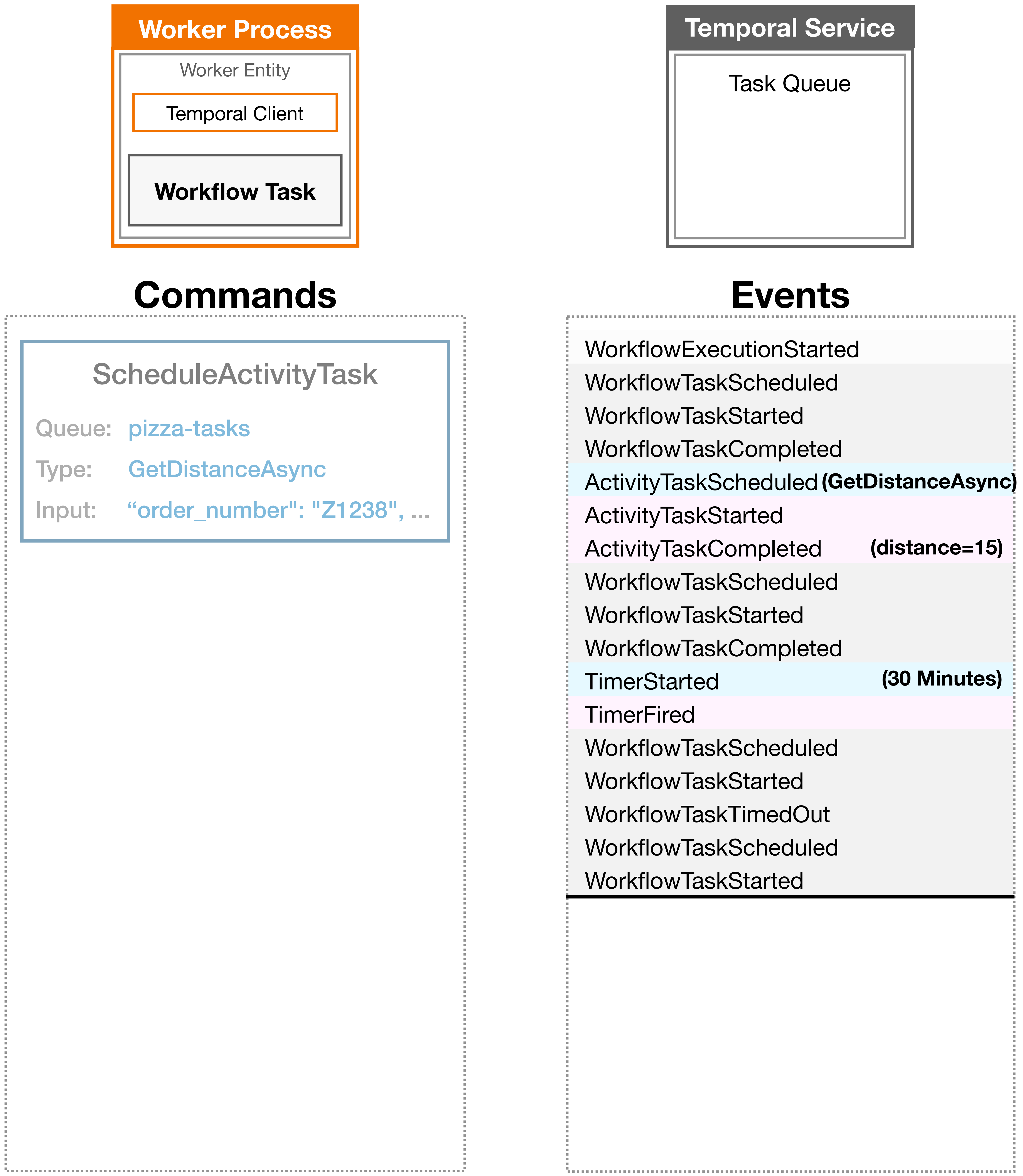
ScheduleActivityTask

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number”: "Z1238", ...

**Events**

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted        (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted                 (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
```

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

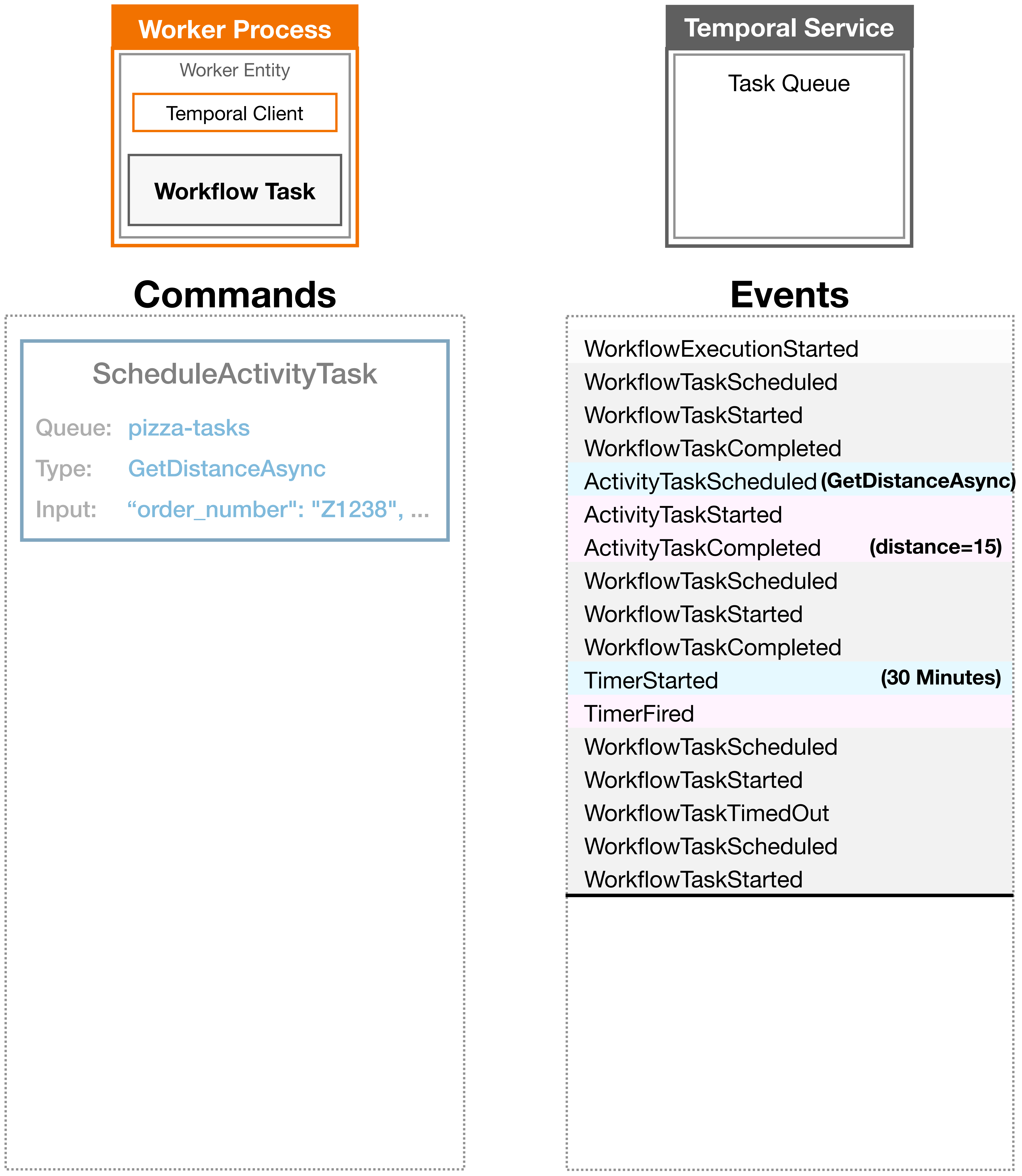


ScheduleActivityTask

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number”: "Z1238", ...

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

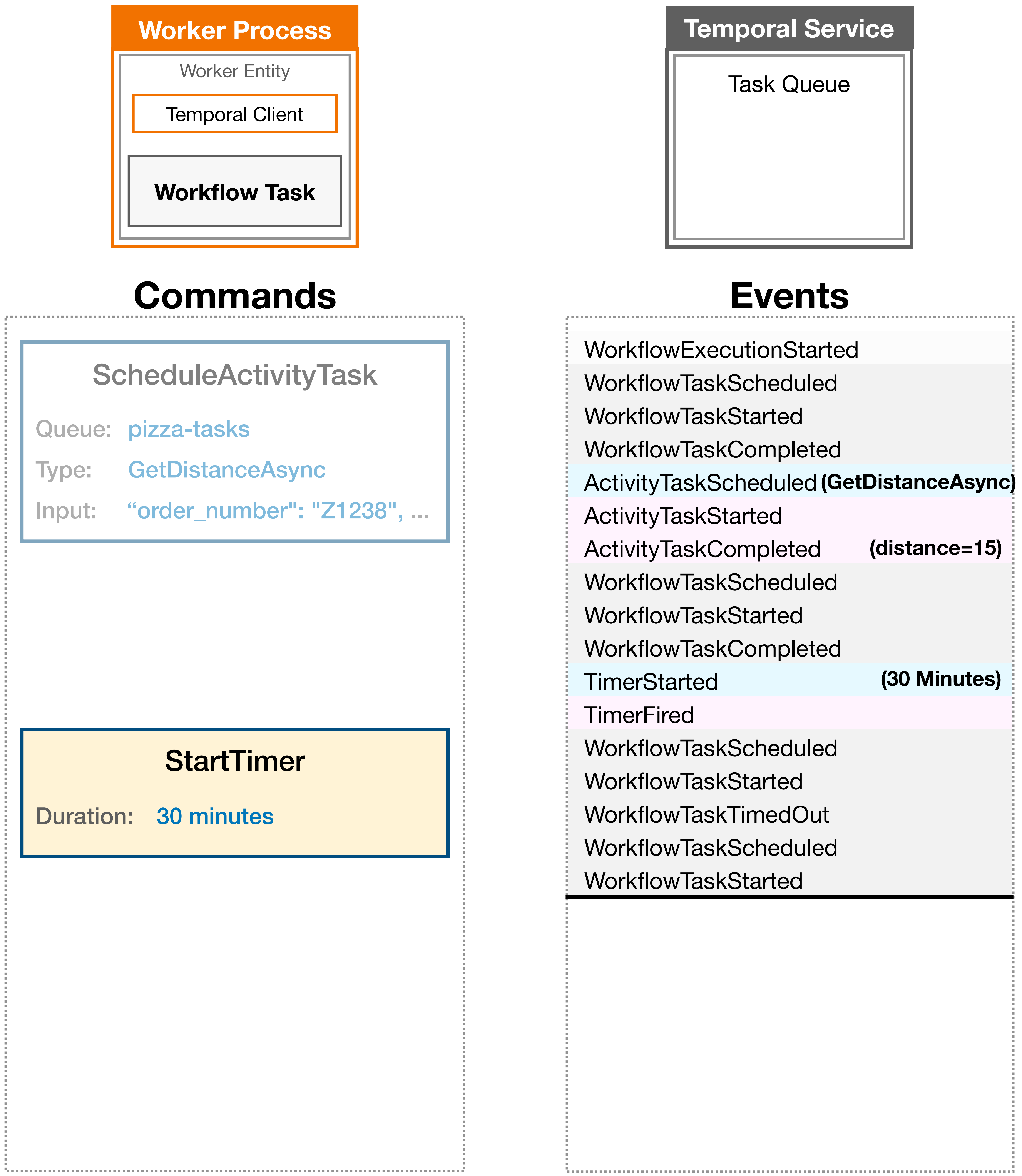
ScheduleActivityTask

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number”: "Z1238", ... |

**Events**

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted        (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted                 (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
```

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



ScheduleActivityTask

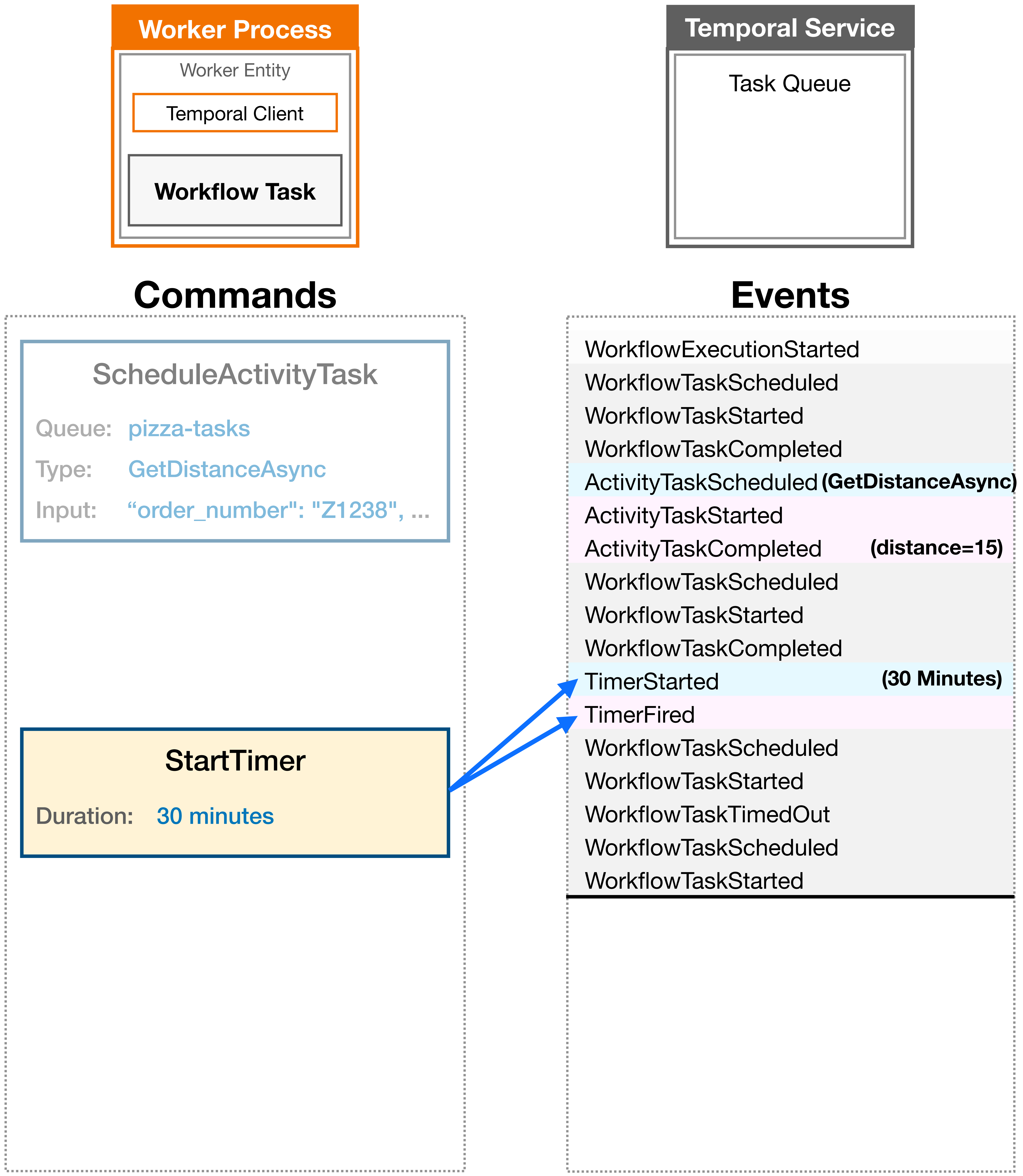
Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

StartTimer

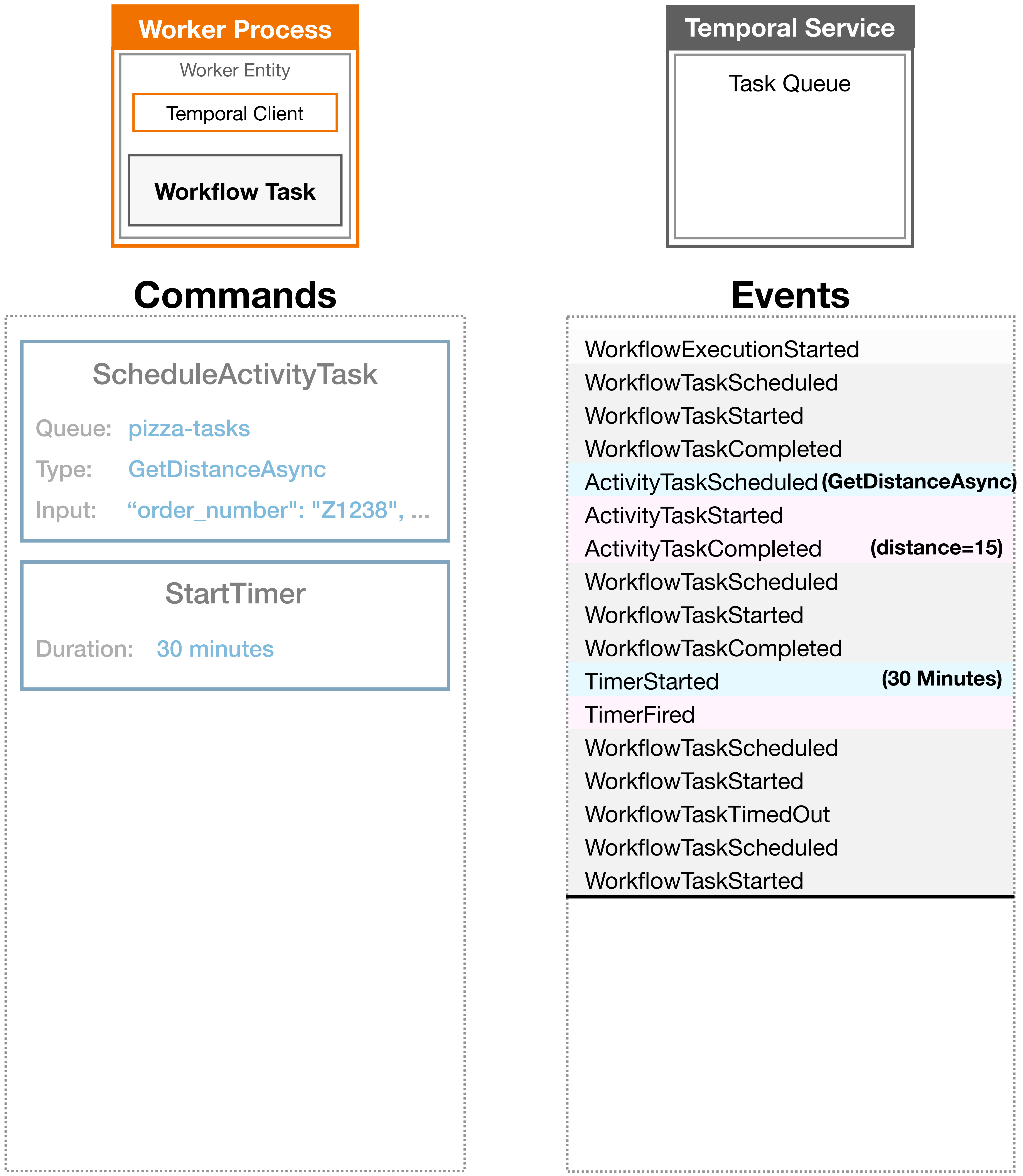
Duration: 30 minutes

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted        (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted                 (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
```

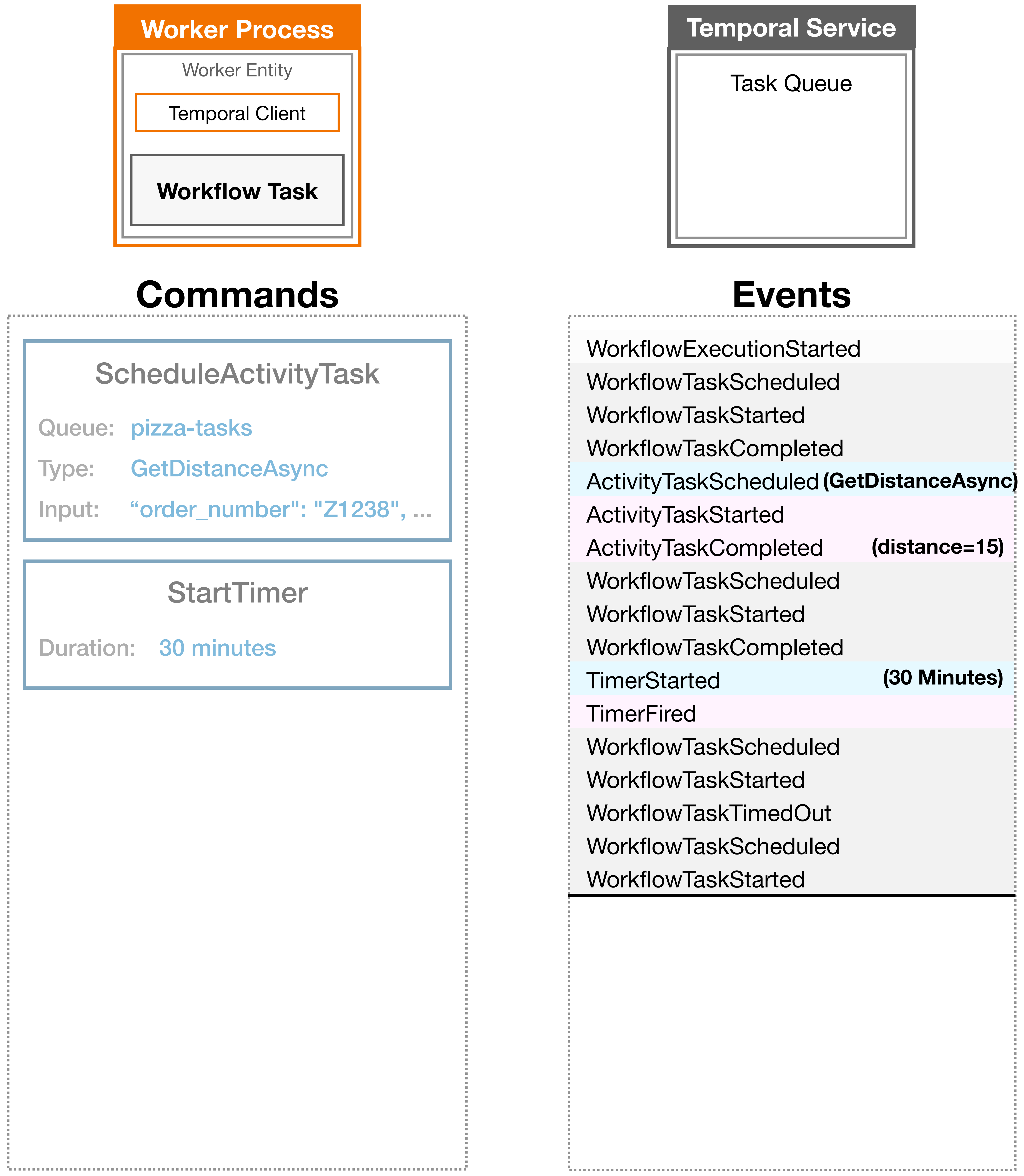
```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

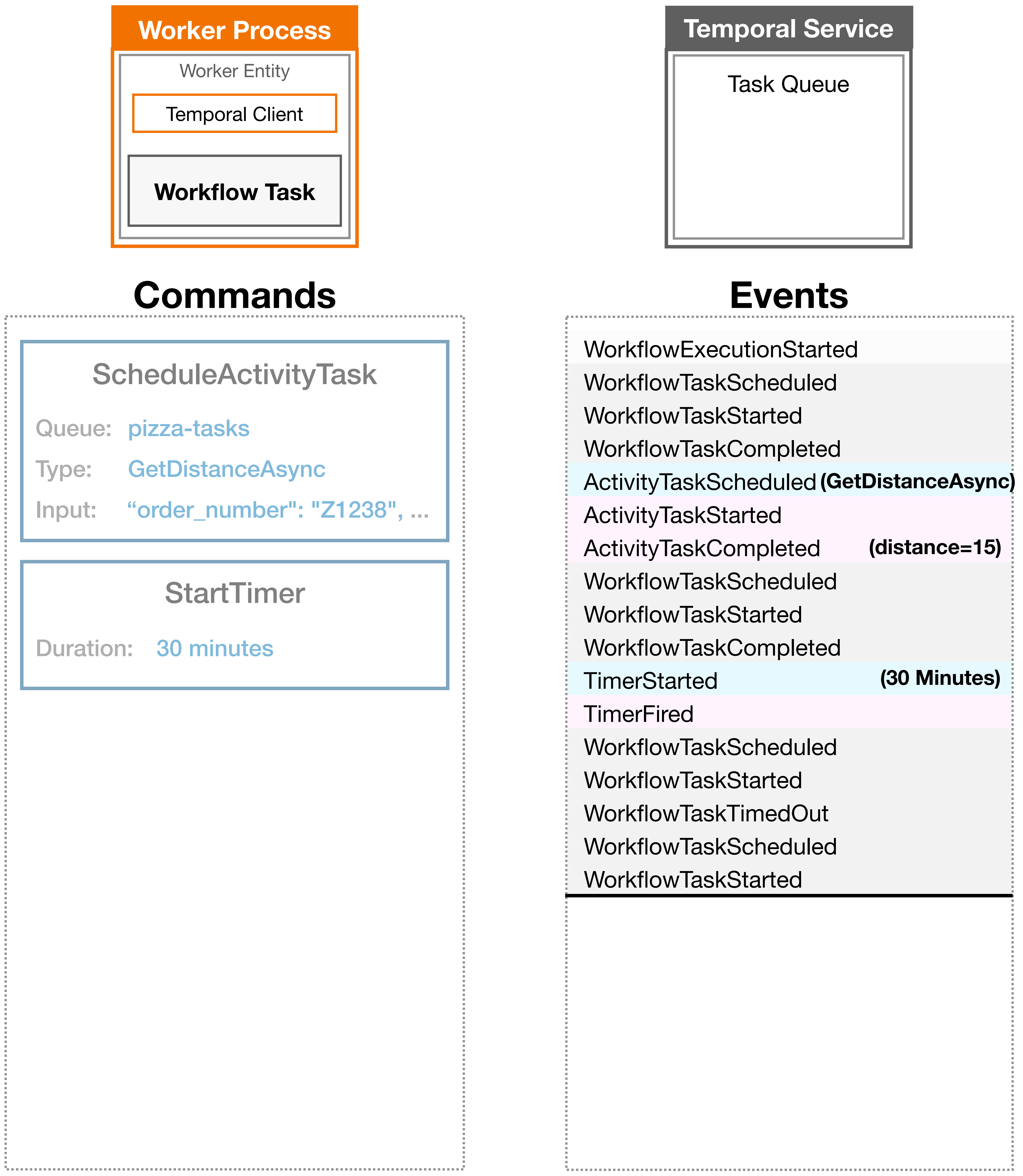
ScheduleActivityTask
Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number”: "Z1238", ...

StartTimer
Duration: 30 minutes

**Events**

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted        (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted                 (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
```

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

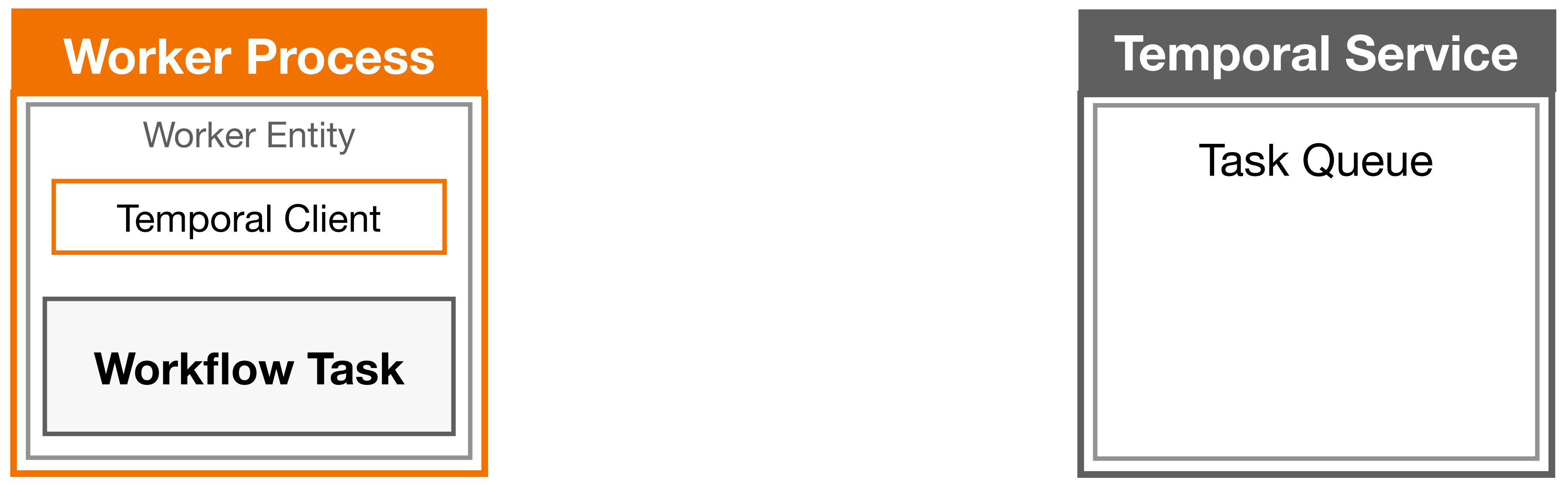
ScheduleActivityTask
Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

StartTimer
Duration: 30 minutes

## Events

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (GetDistanceAsync)
ActivityTaskStarted
ActivityTaskCompleted      (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted               (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
```

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

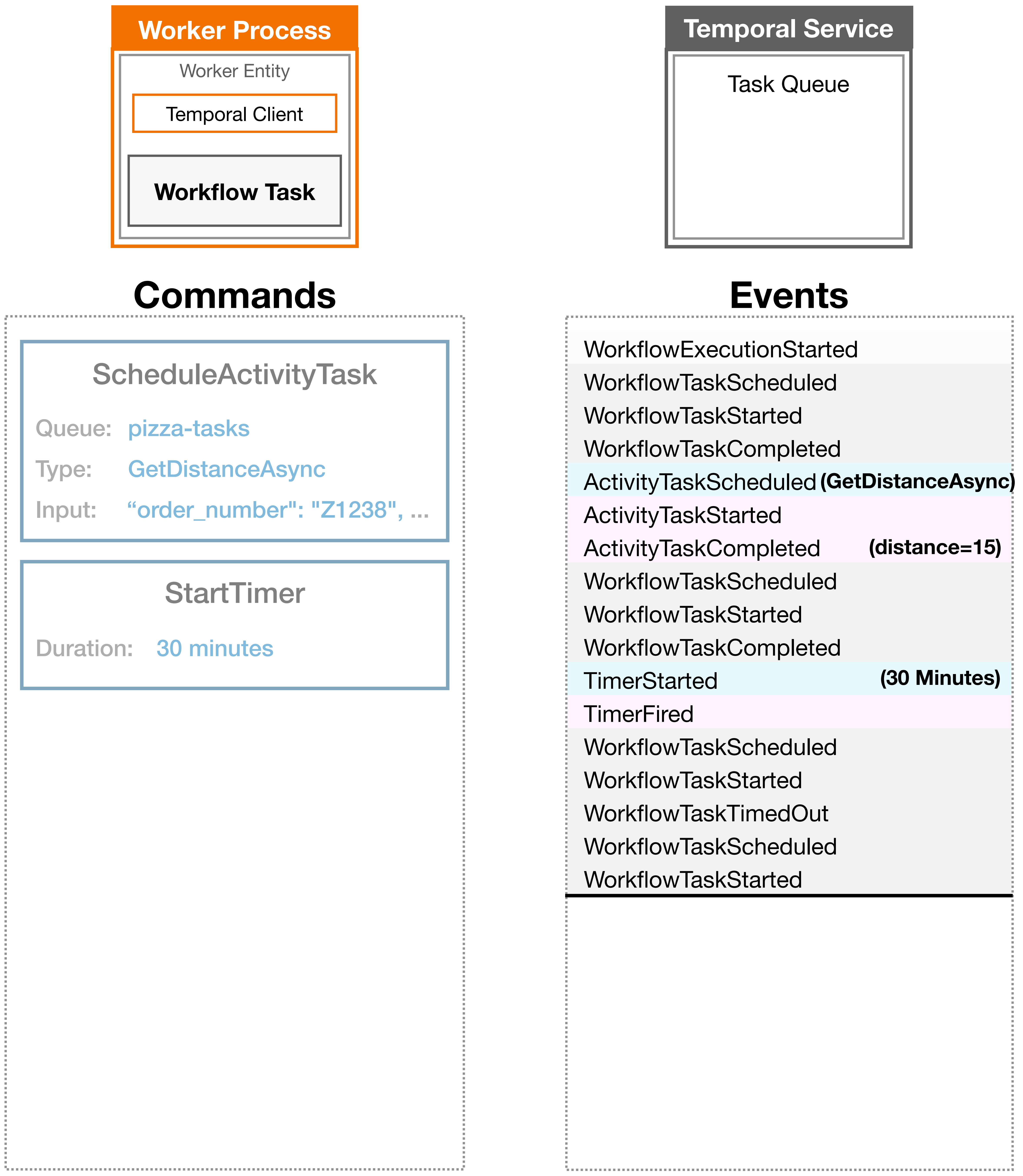
**StartTimer**

| | |
|---|---|
| Duration: | 30 minutes |

## Events

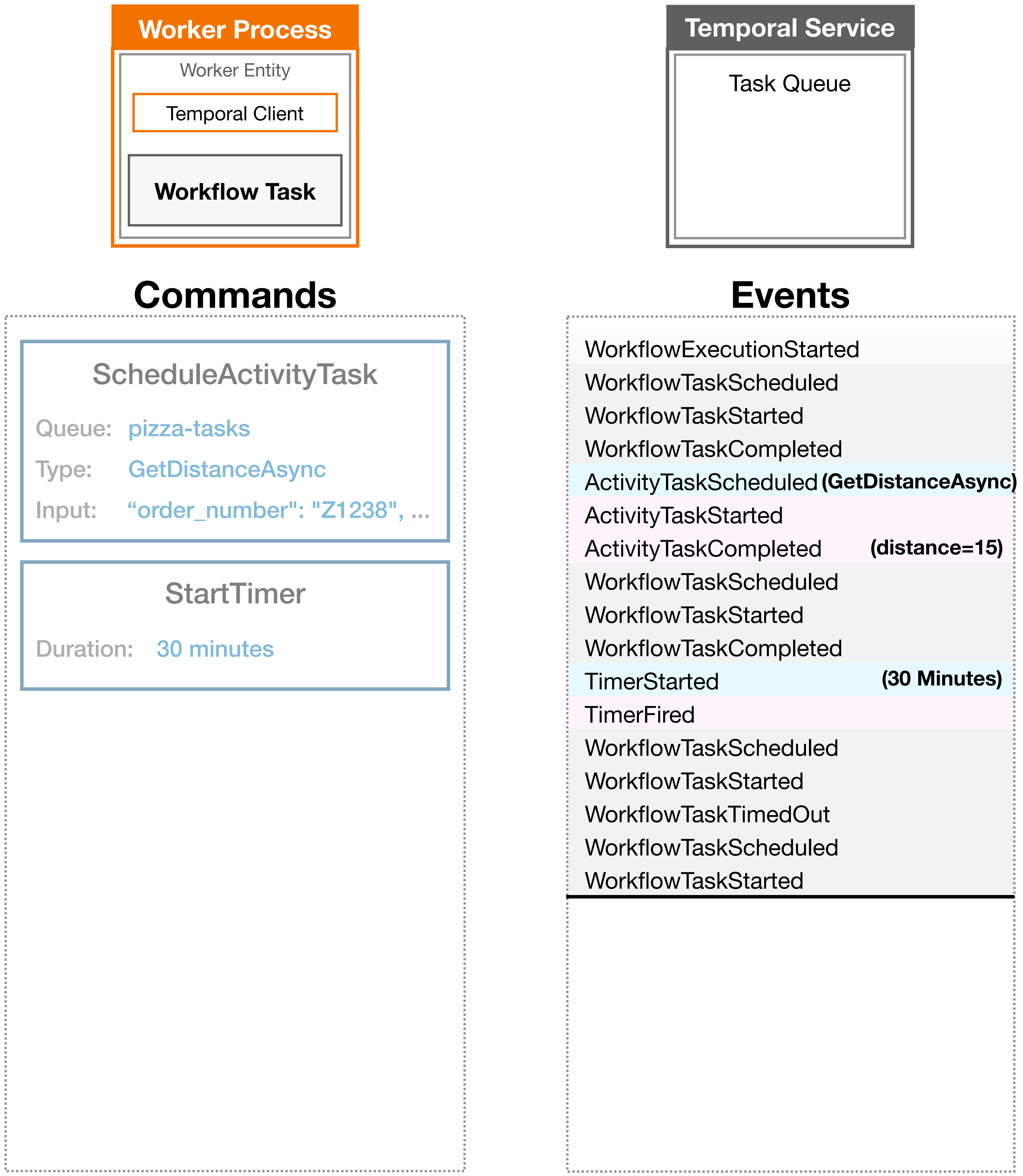
| | |
|---|---|
| WorkflowExecutionStarted | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| ActivityTaskScheduled | **(GetDistanceAsync)** |
| ActivityTaskStarted | |
| ActivityTaskCompleted | **(distance=15)** |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| TimerStarted | **(30 Minutes)** |
| TimerFired | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskTimedOut | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



ScheduleActivityTask

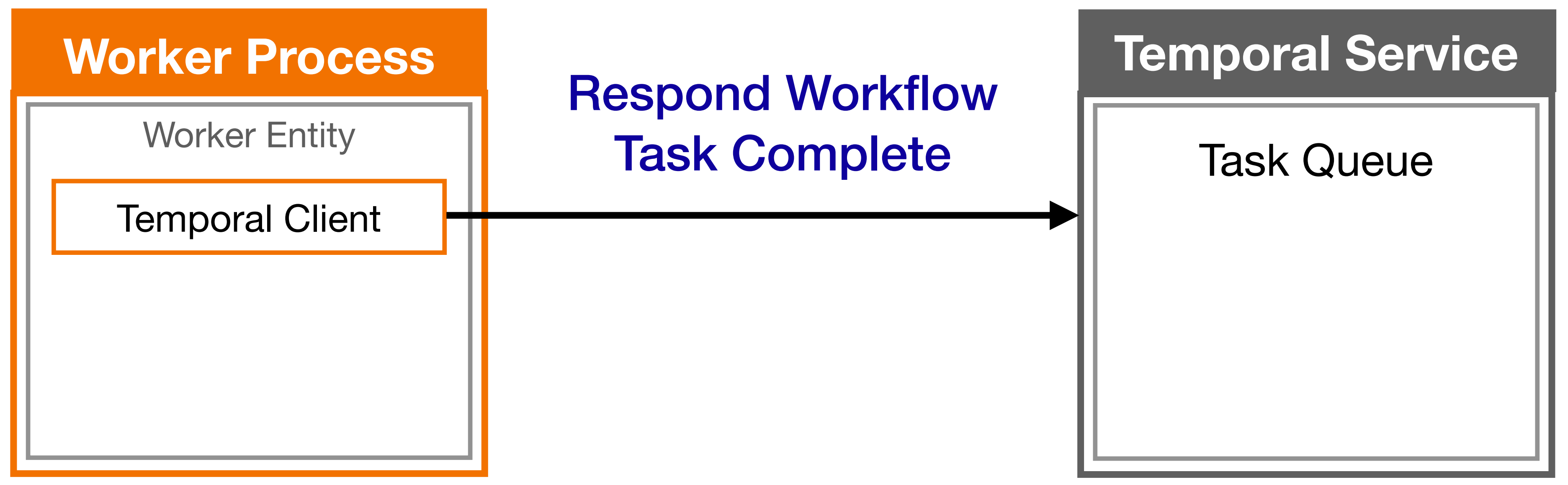
| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

StartTimer

| | |
|---|---|
| Duration: | 30 minutes |

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

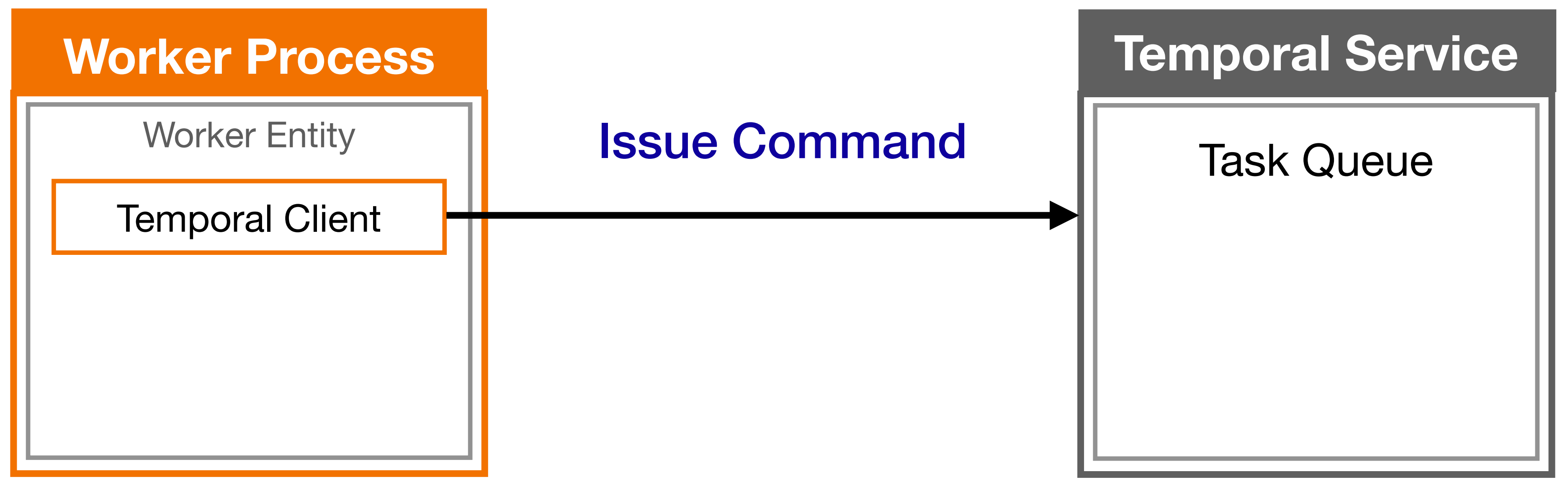
**StartTimer**

| | |
|---|---|
| Duration: | 30 minutes |

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
**WorkflowTaskCompleted**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

**StartTimer**

| | |
|---|---|
| Duration: | 30 minutes |

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | SendBillAsync |
| Input: | "customer_id": 12983, ... |

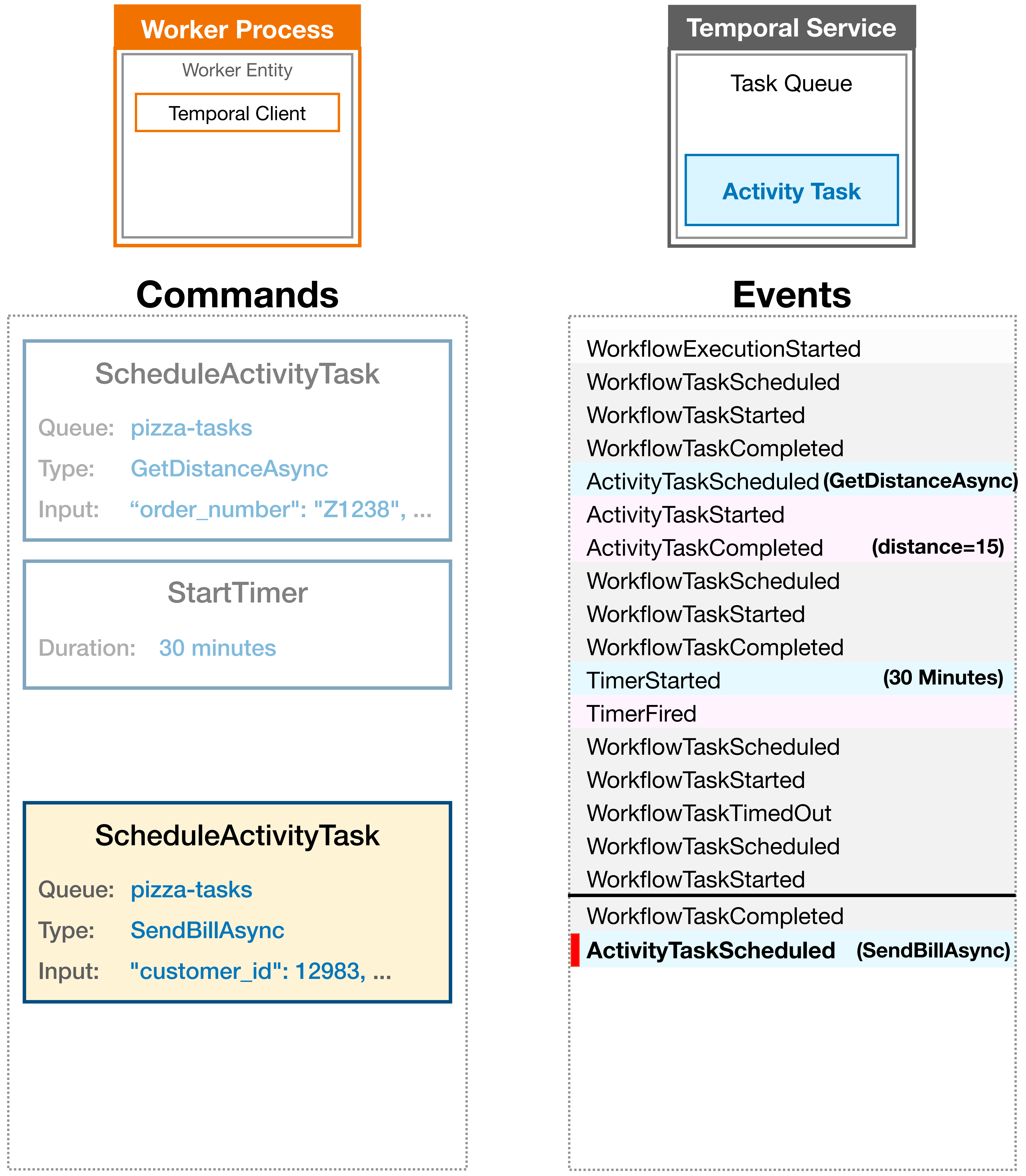
**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

---

WorkflowTaskCompleted

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

ScheduleActivityTask
Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

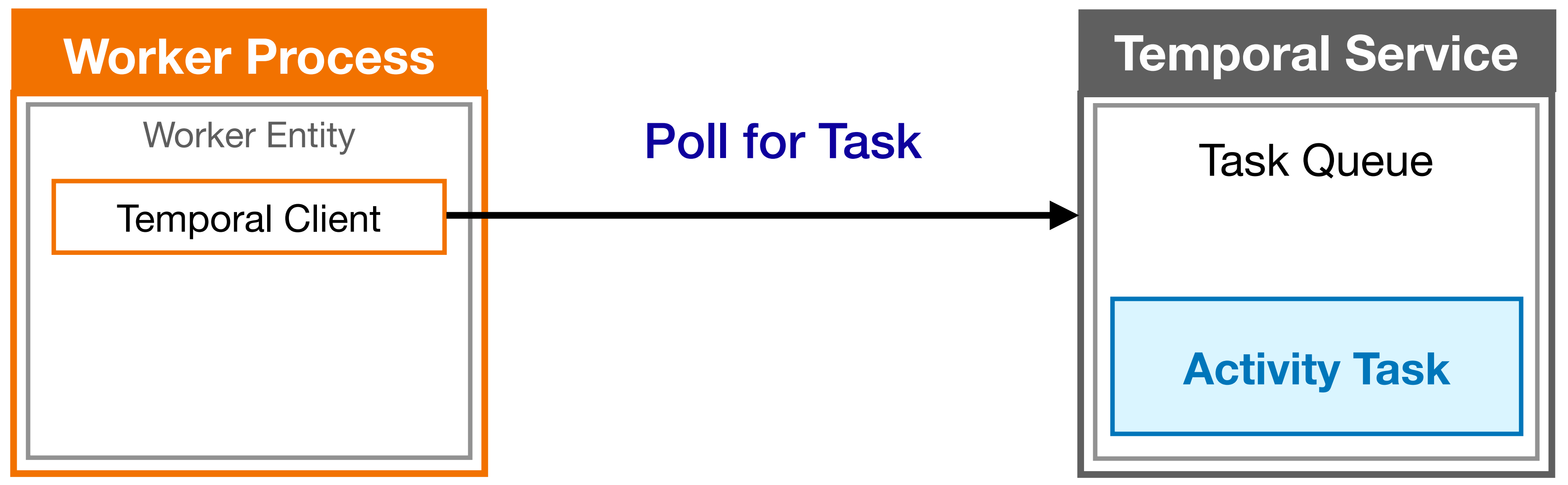
StartTimer
Duration: 30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: SendBillAsync
Input: "customer_id": 12983, ...

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
**ActivityTaskScheduled** **(SendBillAsync)**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

**StartTimer**

Duration: 30 minutes

**ScheduleActivityTask**

Queue: pizza-tasks
Type: SendBillAsync
Input: "customer_id": 12983, ...

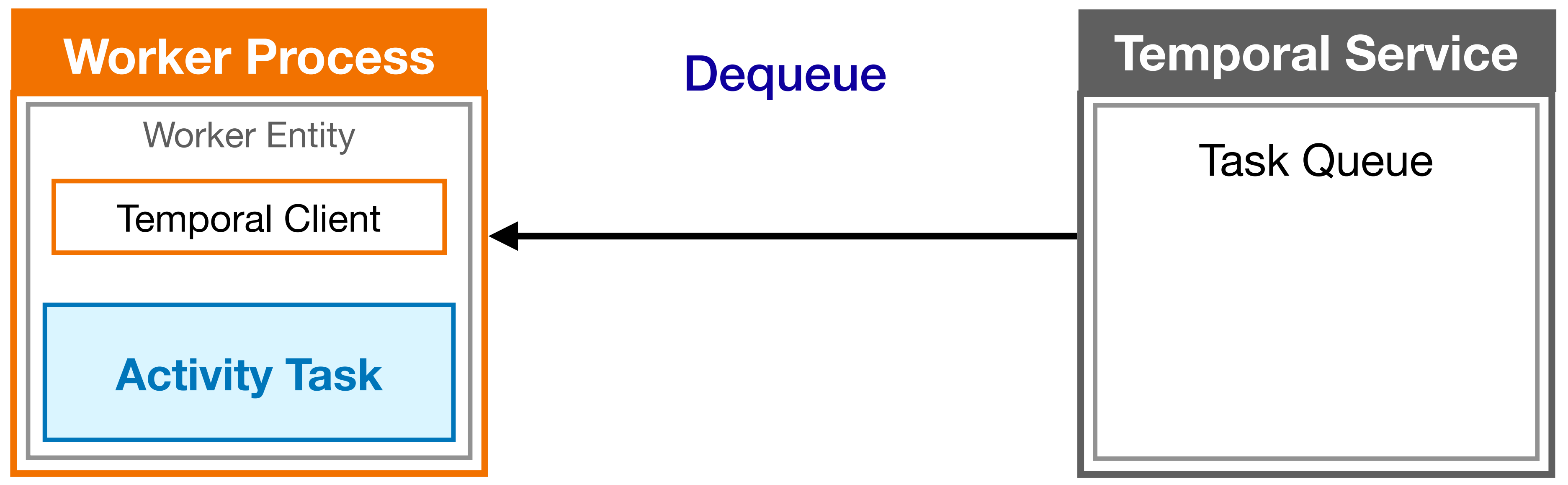
## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

---

WorkflowTaskCompleted
ActivityTaskScheduled **(SendBillAsync)**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

ScheduleActivityTask

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

StartTimer

| | |
|---|---|
| Duration: | 30 minutes |

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | SendBillAsync |
| Input: | "customer_id": 12983, ... |

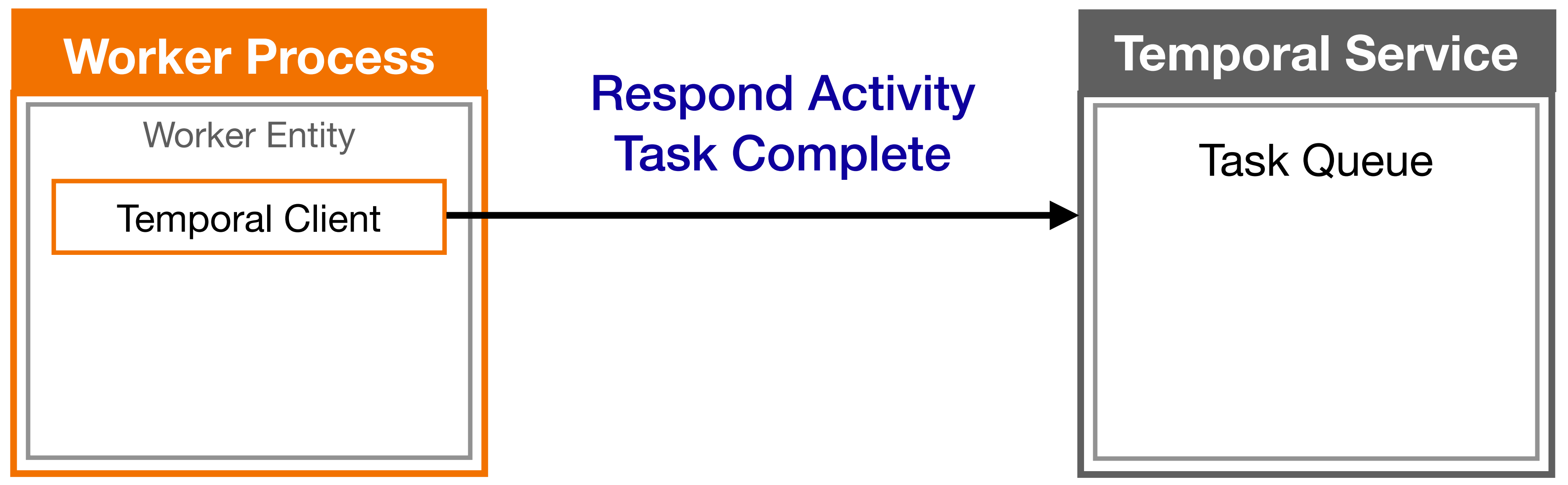
**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

---

WorkflowTaskCompleted
ActivityTaskScheduled **(SendBillAsync)**
**ActivityTaskStarted**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

**StartTimer**

| | |
|---|---|
| Duration: | 30 minutes |

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | SendBillAsync |
| Input: | "customer_id": 12983, ... |

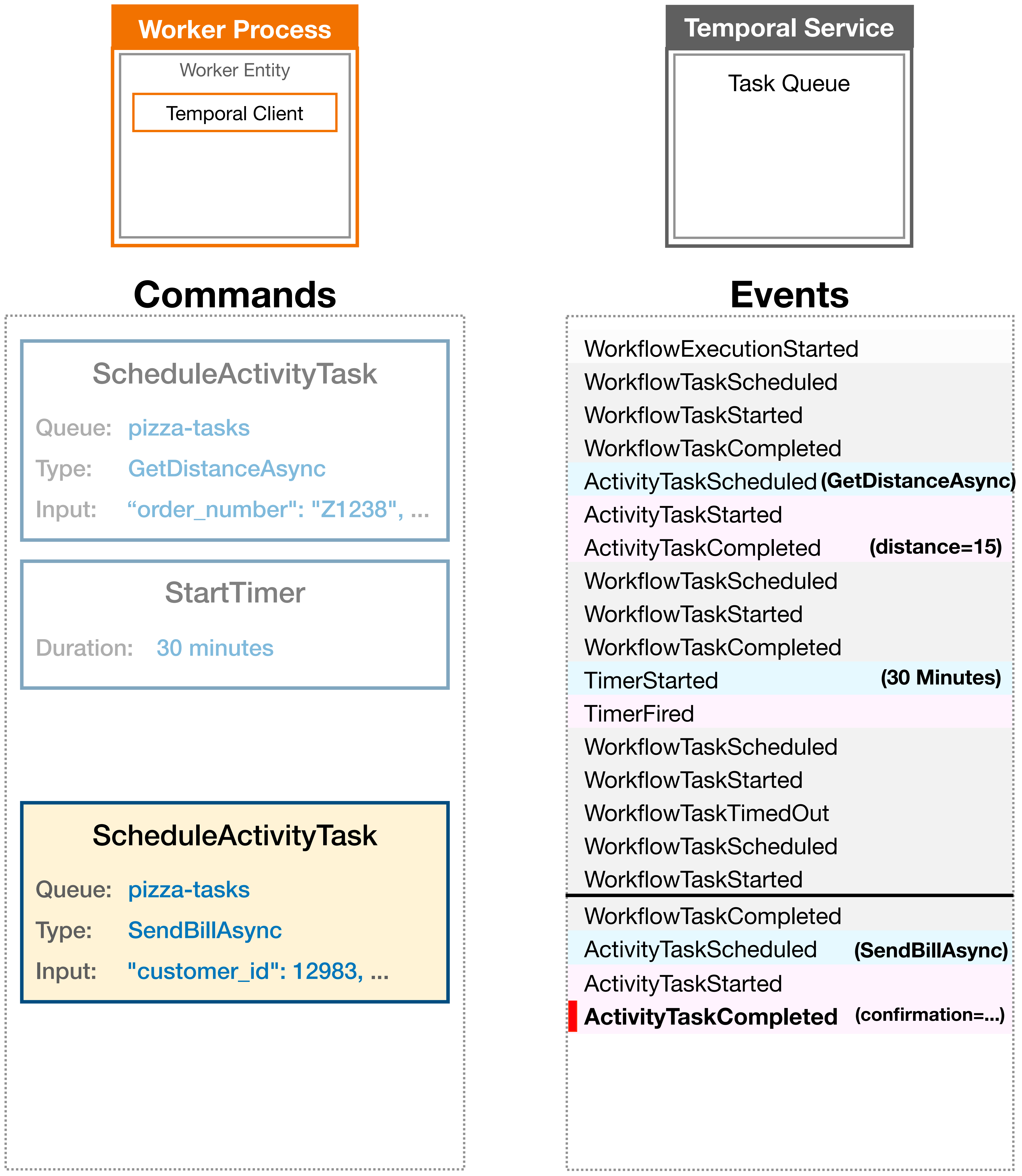
## Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted

---

WorkflowTaskCompleted
ActivityTaskScheduled **(SendBillAsync)**
ActivityTaskStarted

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

ScheduleActivityTask
Queue: pizza-tasks
Type: GetDistanceAsync
Input: "order_number": "Z1238", ...

StartTimer
Duration: 30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: SendBillAsync
Input: "customer_id": 12983, ...

**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(SendBillAsync)**
ActivityTaskStarted
**ActivityTaskCompleted** **(confirmation=...)**

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

Worker Process
Worker Entity
Temporal Client
Temporal Service
Task Queue
Workflow Task

## Commands

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | “order_number": "Z1238", ... |

**StartTimer**

| | |
|---|---|
| Duration: | 30 minutes |

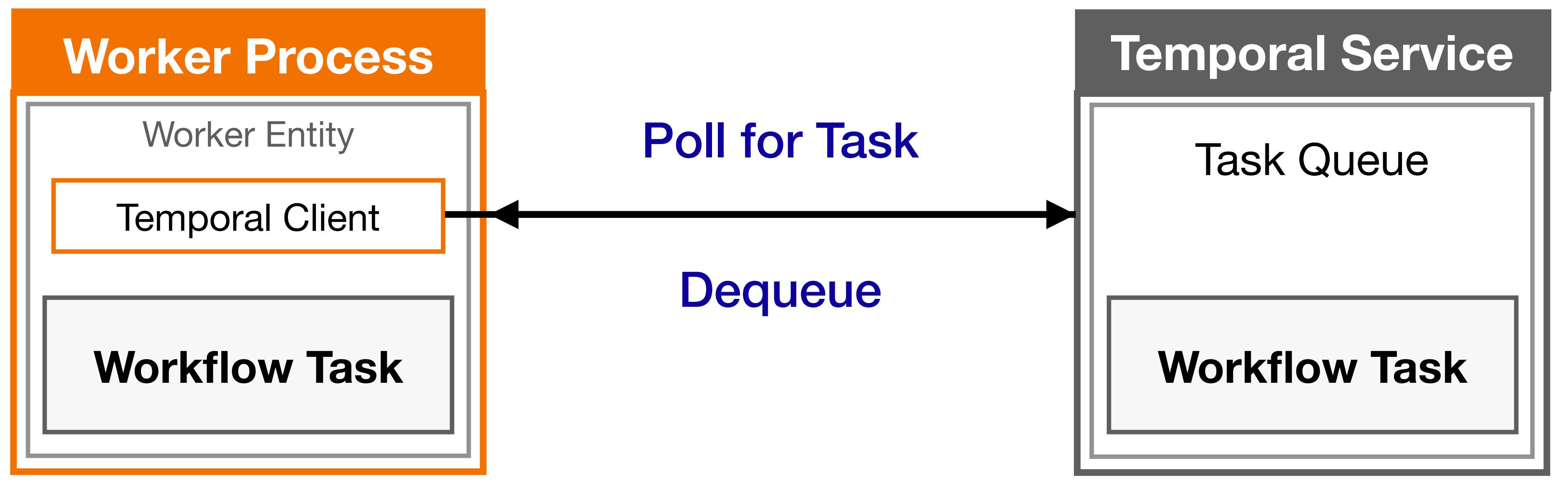
**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | SendBillAsync |
| Input: | "customer_id": 12983, ... |

## Events

| | |
|---|---|
| WorkflowExecutionStarted | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| ActivityTaskScheduled | **(GetDistanceAsync)** |
| ActivityTaskStarted | |
| ActivityTaskCompleted | **(distance=15)** |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| TimerStarted | **(30 Minutes)** |
| TimerFired | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskTimedOut | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| ActivityTaskScheduled | **(SendBillAsync)** |
| ActivityTaskStarted | |
| ActivityTaskCompleted | **(confirmation=...)** |
| **WorkflowTaskScheduled** | |

```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



## Commands

**ScheduleActivityTask**

| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | GetDistanceAsync |
| Input: | "order_number": "Z1238", ... |

**StartTimer**

| | |
|---|---|
| Duration: | 30 minutes |

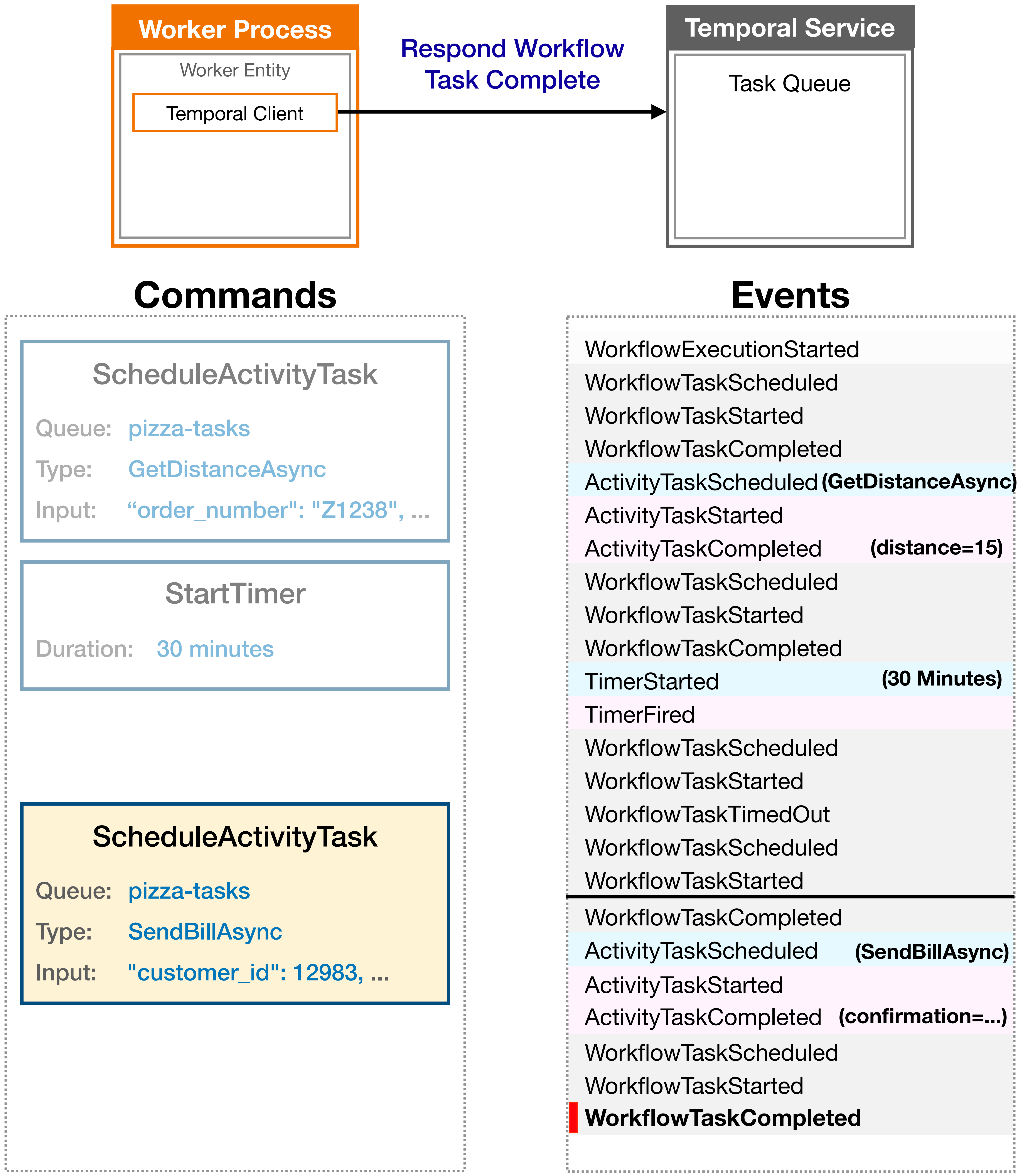
**ScheduleActivityTask**

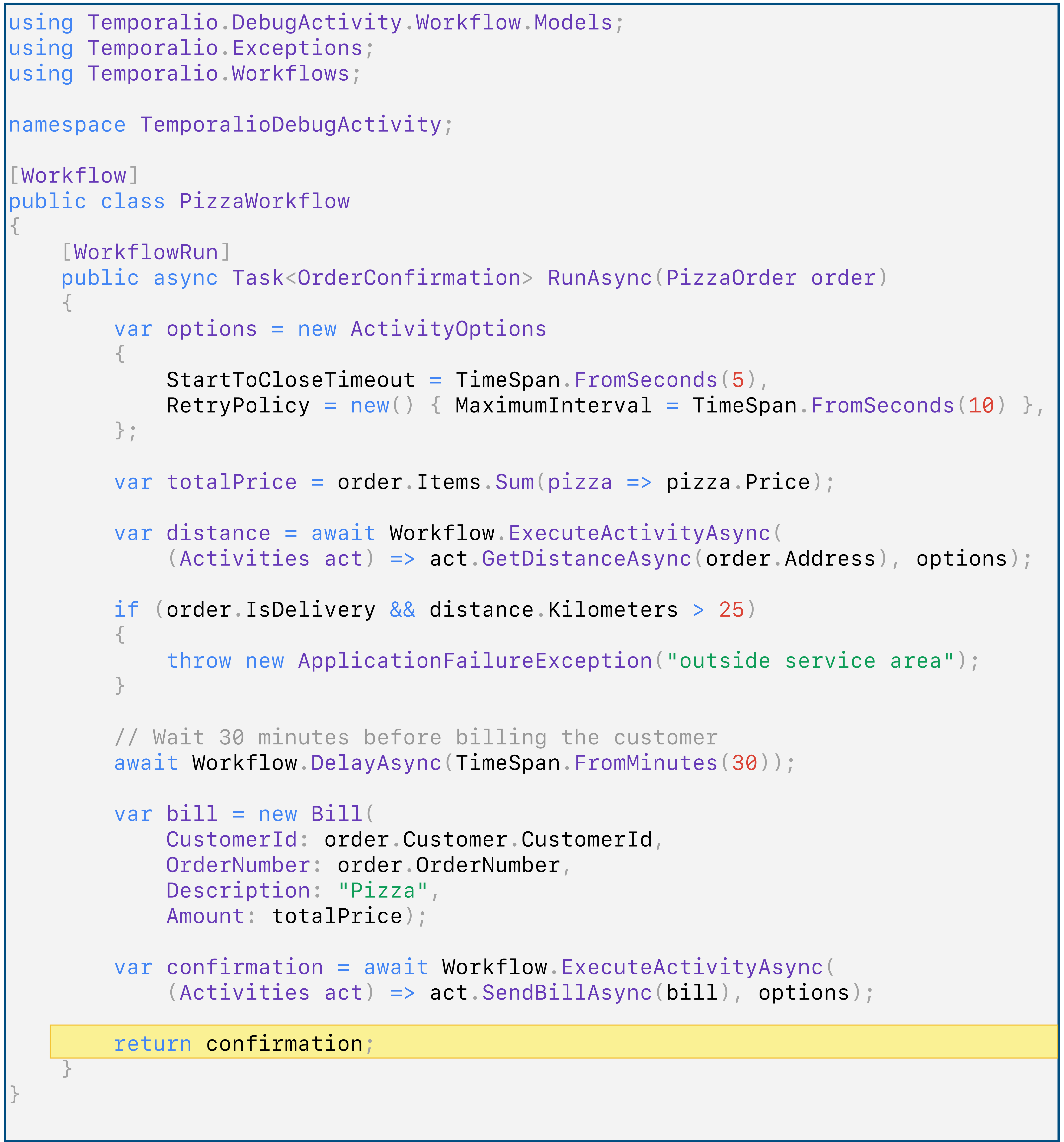
| | |
|---|---|
| Queue: | pizza-tasks |
| Type: | SendBillAsync |
| Input: | "customer_id": 12983, ... |

## Events

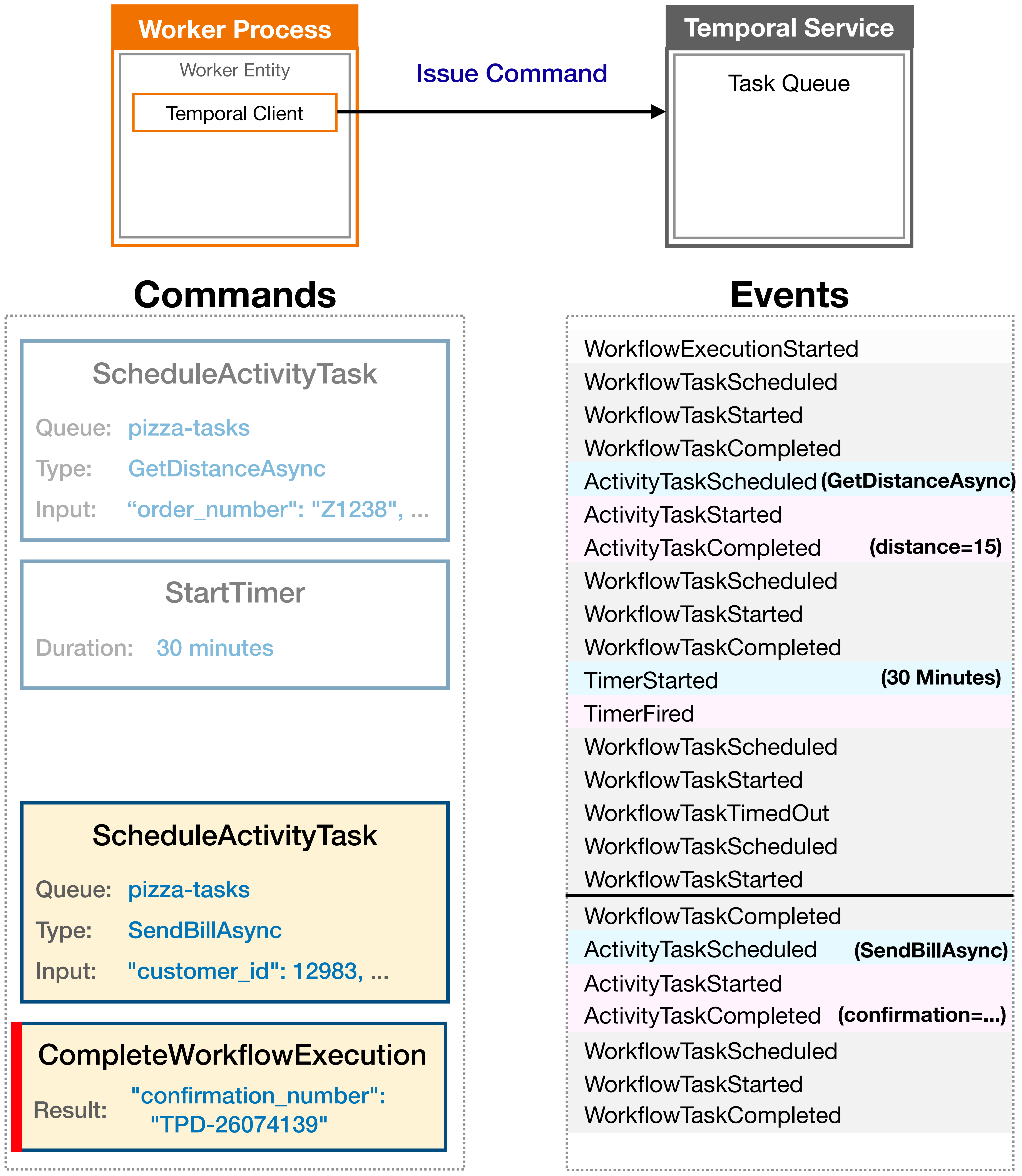
| | |
|---|---|
| WorkflowExecutionStarted | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| ActivityTaskScheduled | **(GetDistanceAsync)** |
| ActivityTaskStarted | |
| ActivityTaskCompleted | **(distance=15)** |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| TimerStarted | **(30 Minutes)** |
| TimerFired | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskTimedOut | |
| WorkflowTaskScheduled | |
| WorkflowTaskStarted | |
| WorkflowTaskCompleted | |
| ActivityTaskScheduled | **(SendBillAsync)** |
| ActivityTaskStarted | |
| ActivityTaskCompleted | **(confirmation=...)** |
| WorkflowTaskScheduled | |
| **WorkflowTaskStarted** | |

```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```

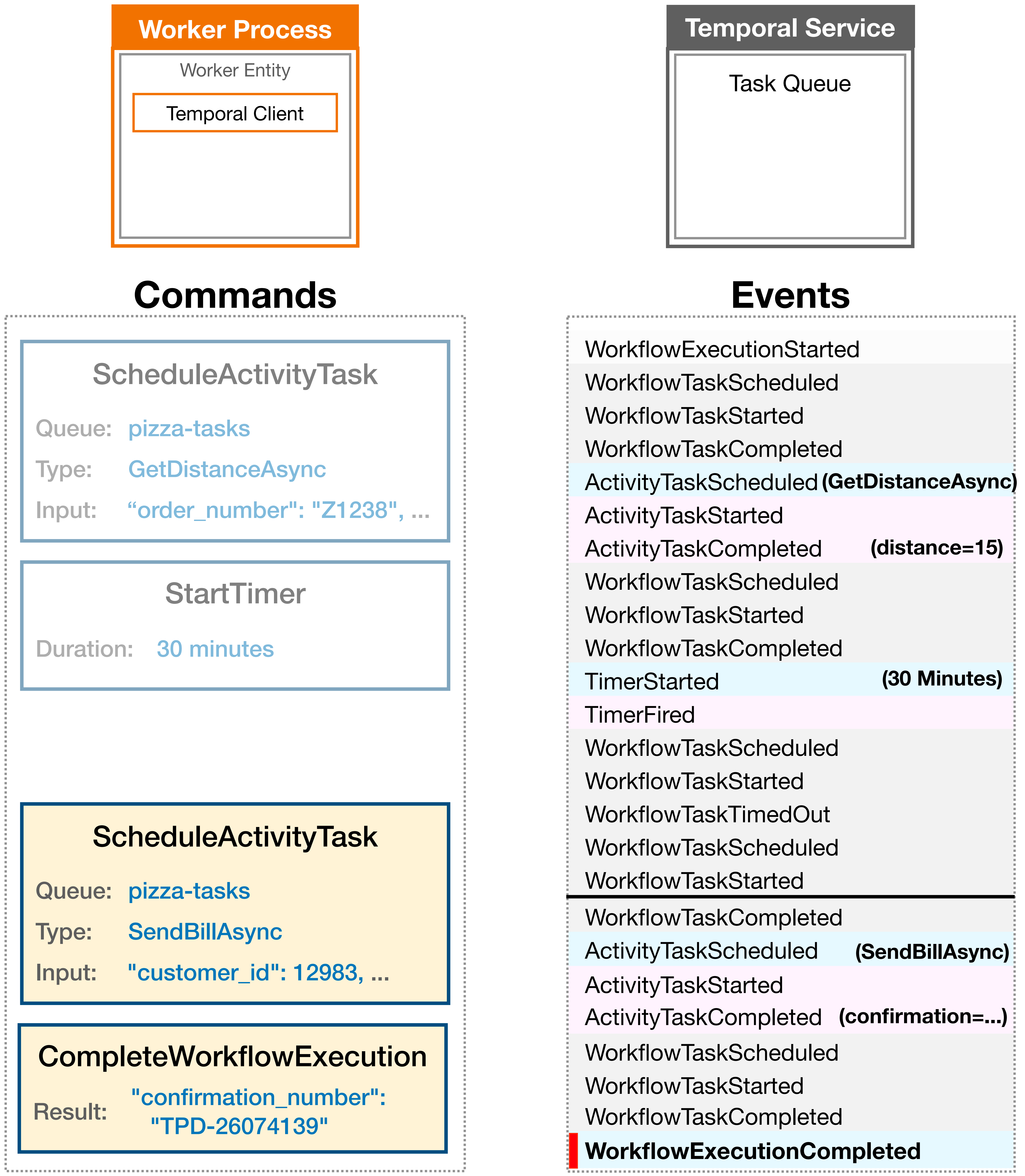




```
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



```csharp
using Temporalio.DebugActivity.Workflow.Models;
using Temporalio.Exceptions;
using Temporalio.Workflows;

namespace TemporalioDebugActivity;

[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        var options = new ActivityOptions
        {
            StartToCloseTimeout = TimeSpan.FromSeconds(5),
            RetryPolicy = new() { MaximumInterval = TimeSpan.FromSeconds(10) },
        };

        var totalPrice = order.Items.Sum(pizza => pizza.Price);

        var distance = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.GetDistanceAsync(order.Address), options);

        if (order.IsDelivery && distance.Kilometers > 25)
        {
            throw new ApplicationFailureException("outside service area");
        }

        // Wait 30 minutes before billing the customer
        await Workflow.DelayAsync(TimeSpan.FromMinutes(30));

        var bill = new Bill(
            CustomerId: order.Customer.CustomerId,
            OrderNumber: order.OrderNumber,
            Description: "Pizza",
            Amount: totalPrice);

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.SendBillAsync(bill), options);

        return confirmation;
    }
}
```



**Commands**

ScheduleActivityTask
Queue: pizza-tasks
Type: GetDistanceAsync
Input: “order_number": "Z1238", ...

StartTimer
Duration: 30 minutes

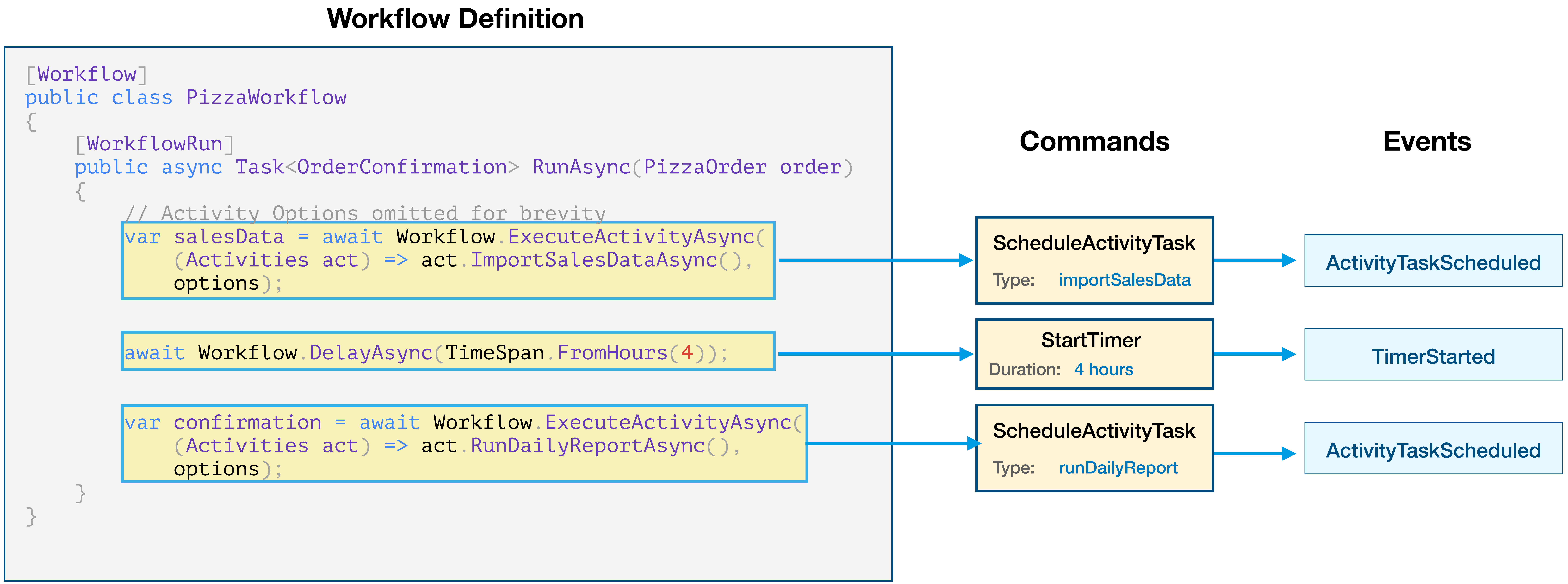
ScheduleActivityTask
Queue: pizza-tasks
Type: SendBillAsync
Input: "customer_id": 12983, ...

CompleteWorkflowExecution
Result: "confirmation_number": "TPD-26074139"

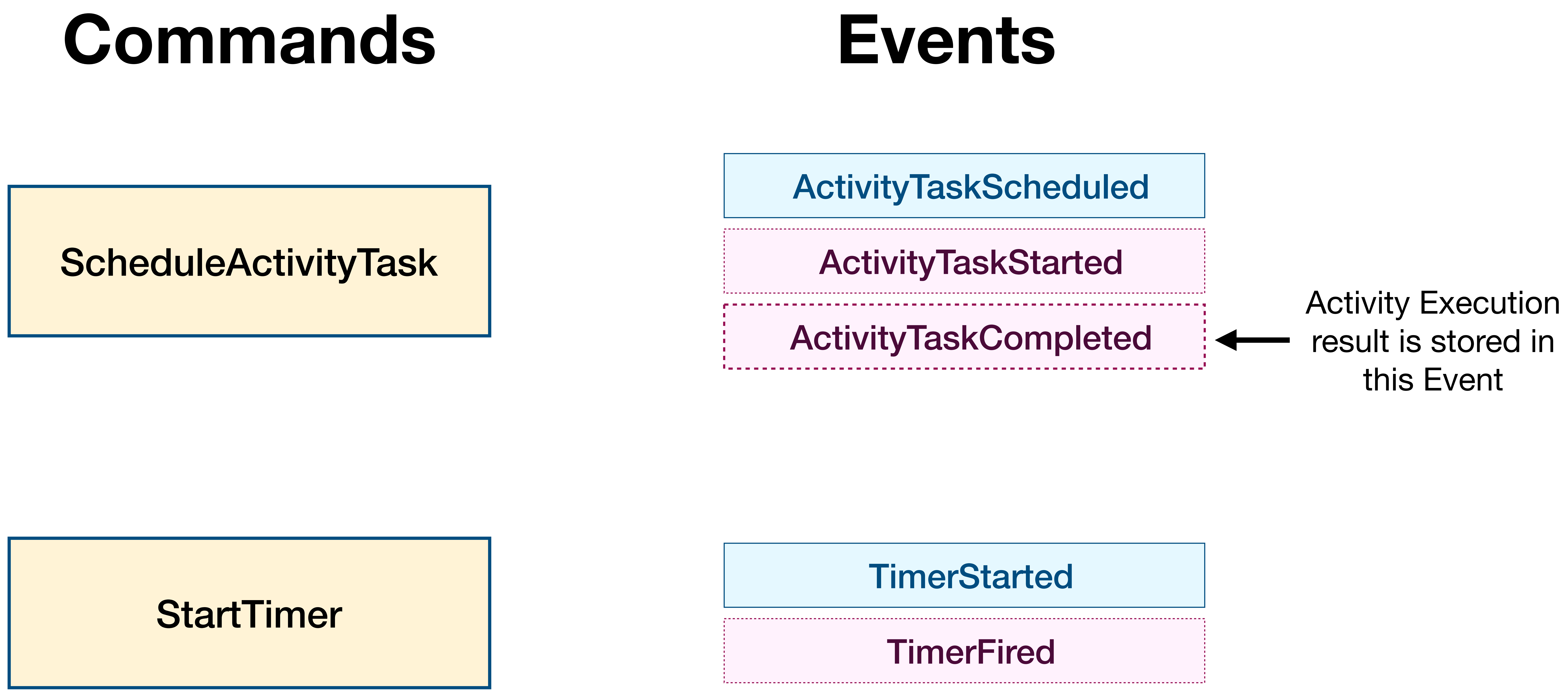
**Events**

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(GetDistanceAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(distance=15)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted **(30 Minutes)**
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled **(SendBillAsync)**
ActivityTaskStarted
ActivityTaskCompleted **(confirmation=...)**
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
**WorkflowExecutionCompleted**

## Why Temporal Requires Determinism for Workflows



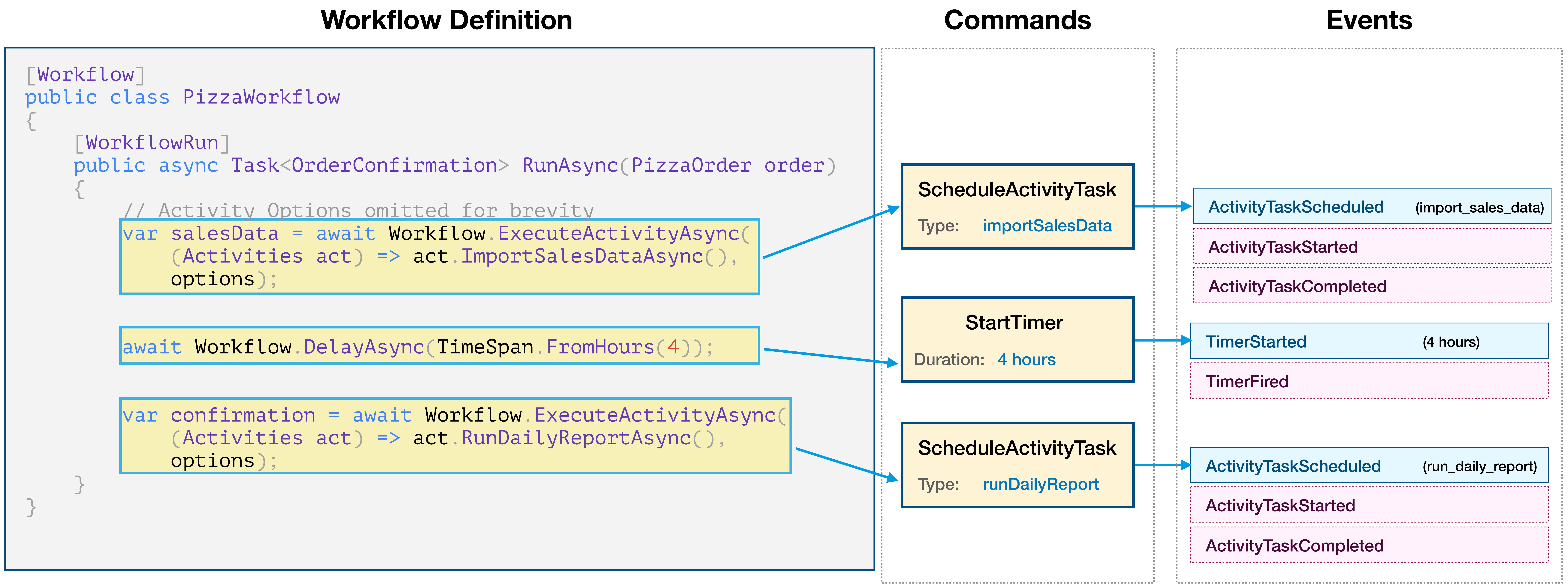
```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity Options omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        await Workflow.DelayAsync(TimeSpan.FromHours(4));

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```


Commands
Events
ScheduleActivityTask
ActivityTaskScheduled
ActivityTaskStarted
ActivityTaskCompleted
Activity Execution result is stored in this Event
StartTimer
TimerStarted
TimerFired

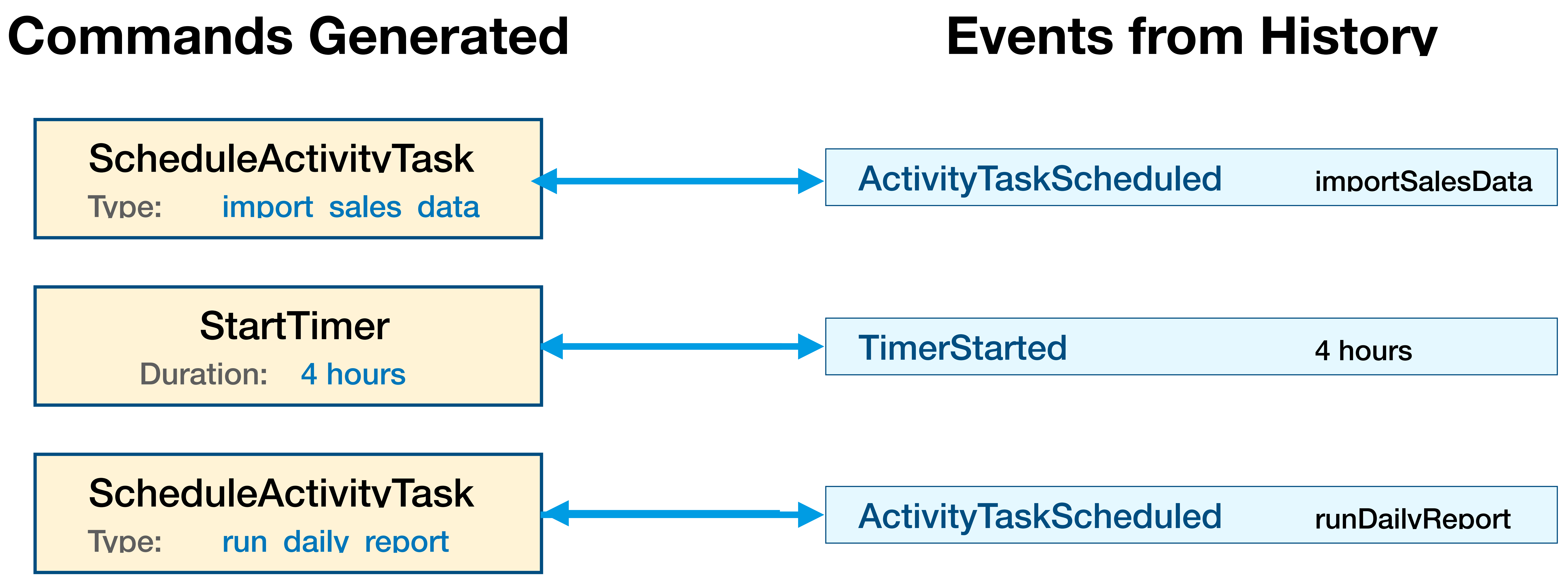
## Deterministic Workflows:

- A Workflow is deterministic if every execution of its Workflow Definition:
    - produces the same Commands
    - in the same sequence
    - given the same input

**Temporal's ability to guarantee durable execution of your Workflow depends on deterministic Workflows.**



```
[Workflow]
public class PizzaWorkflow
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {
        // Activity Options omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        await Workflow.DelayAsync(TimeSpan.FromHours(4));

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```



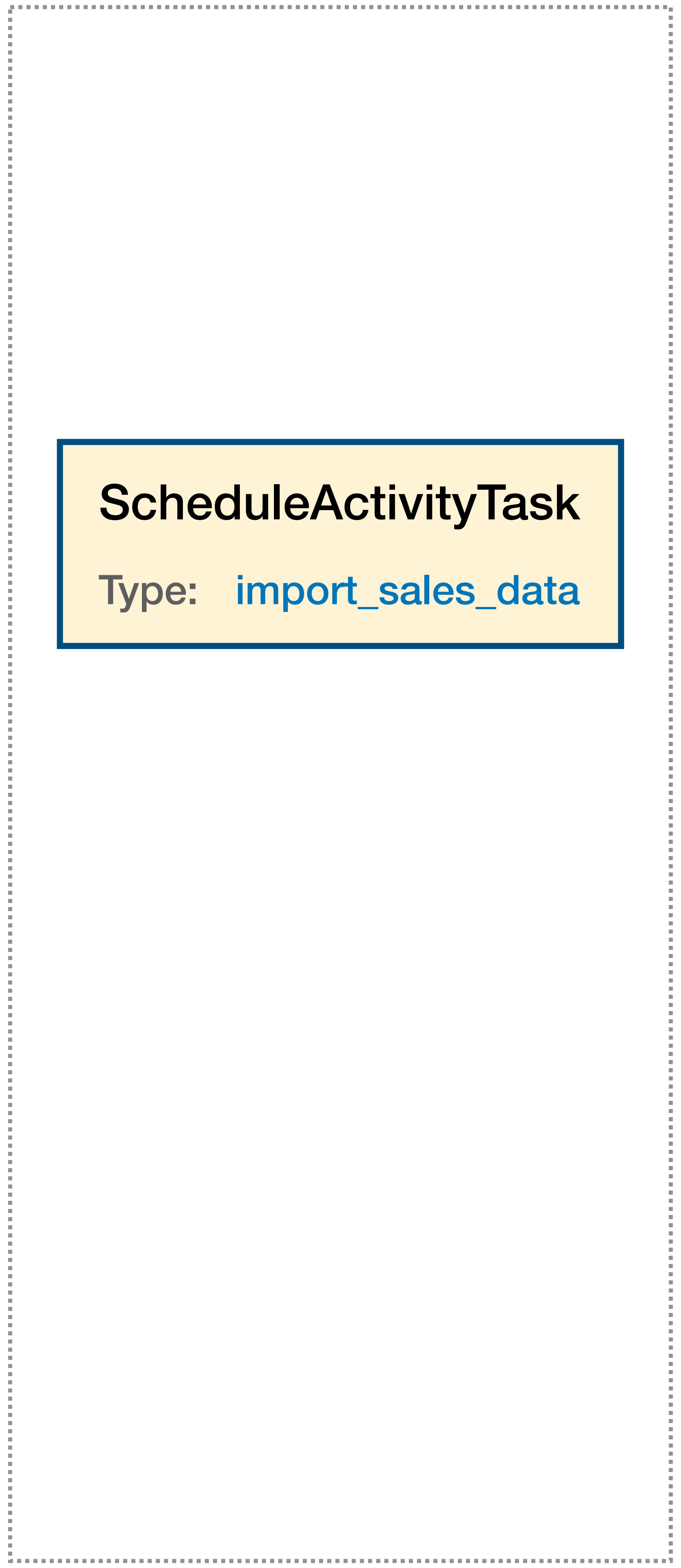
- **Given an Event, you can determine which Command led to the Event**
- **Events that are the direct result of Commands are used to create a list of Commands expected during Replay**

## Example of a Non-Deterministic Workflow

## A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```

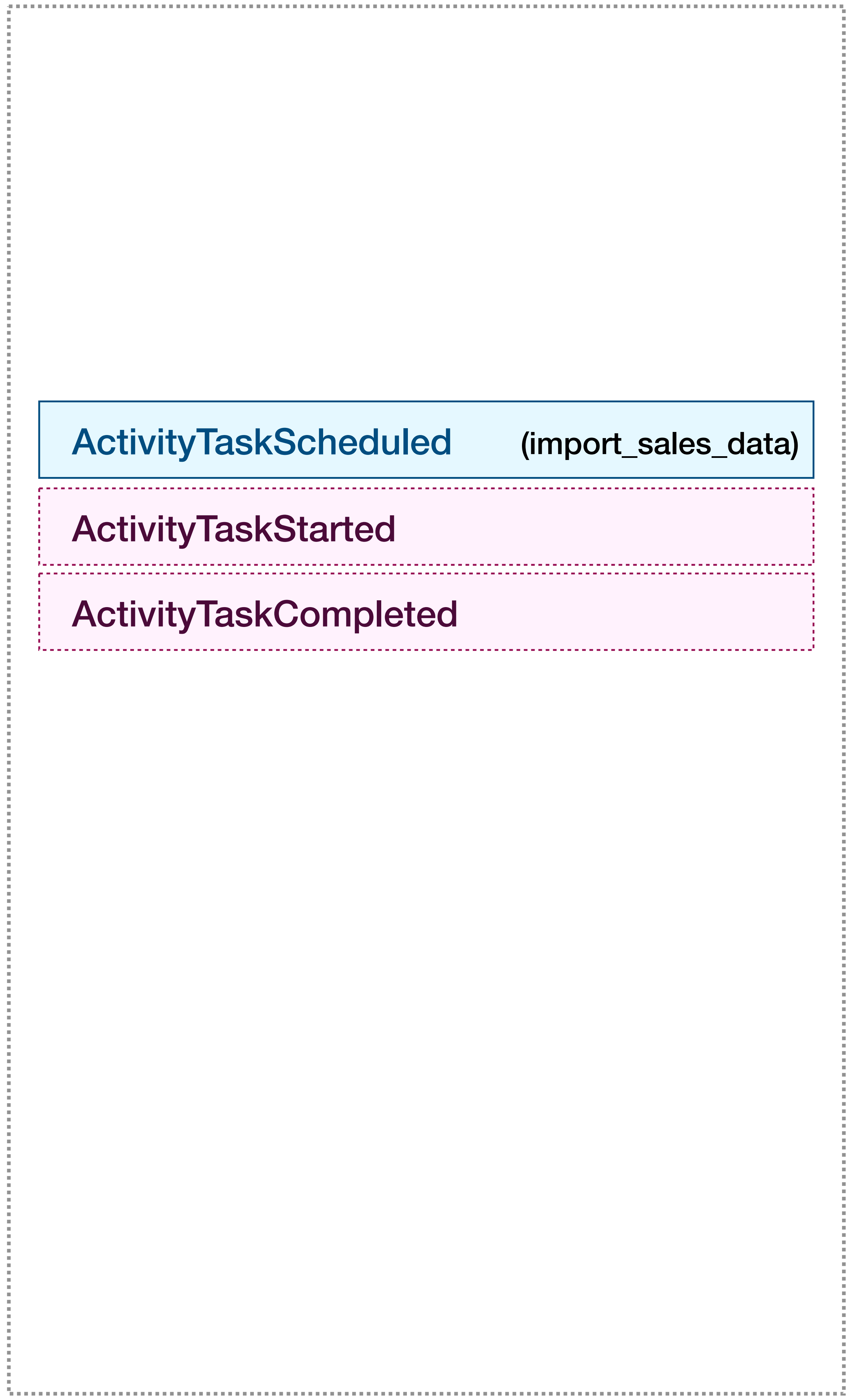
## Commands Created



**ScheduleActivityTask**

Type: import_sales_data

## Relevant Events Logged



ActivityTaskScheduled (import_sales_data)

ActivityTaskStarted

ActivityTaskCompleted

## A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAs
            (Activities act) => act.ImportSalesDataAsync
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information(“Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```

Happens to return 84 during this execution

### Commands Created

**ScheduleActivityTask**

Type: import_sales_data

### Relevant Events Logged

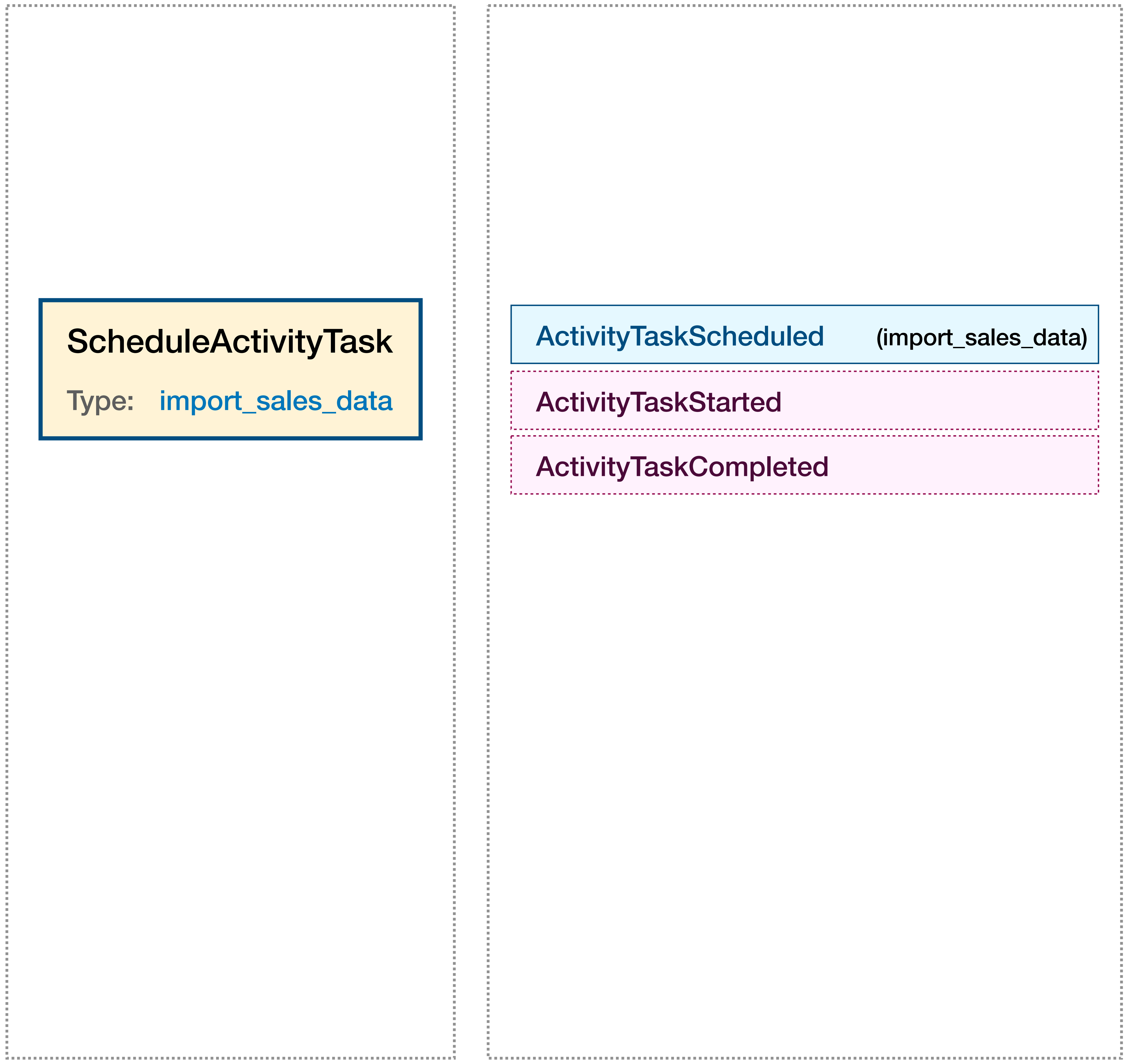
ActivityTaskScheduled (import_sales_data)

ActivityTaskStarted

ActivityTaskCompleted

## A Non-Deterministic Workflow Definition

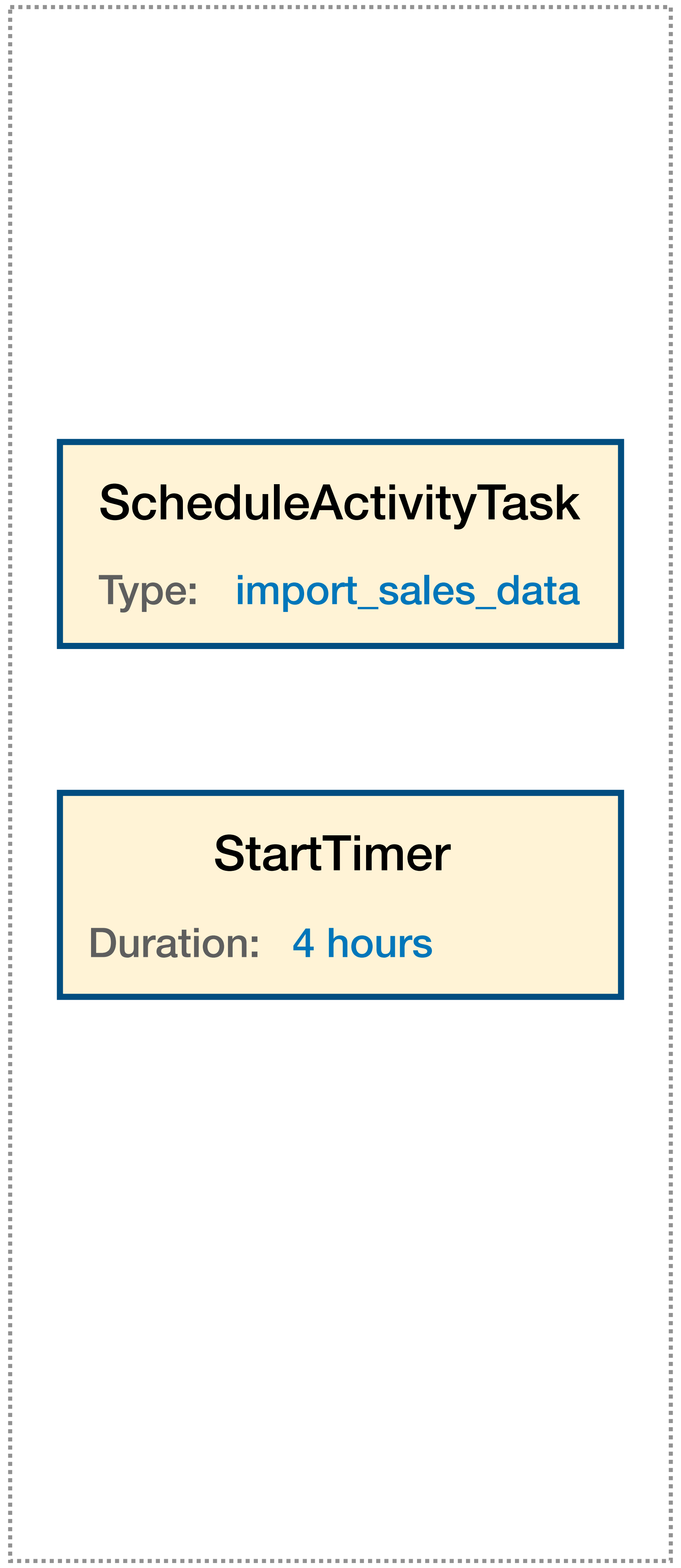
```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```



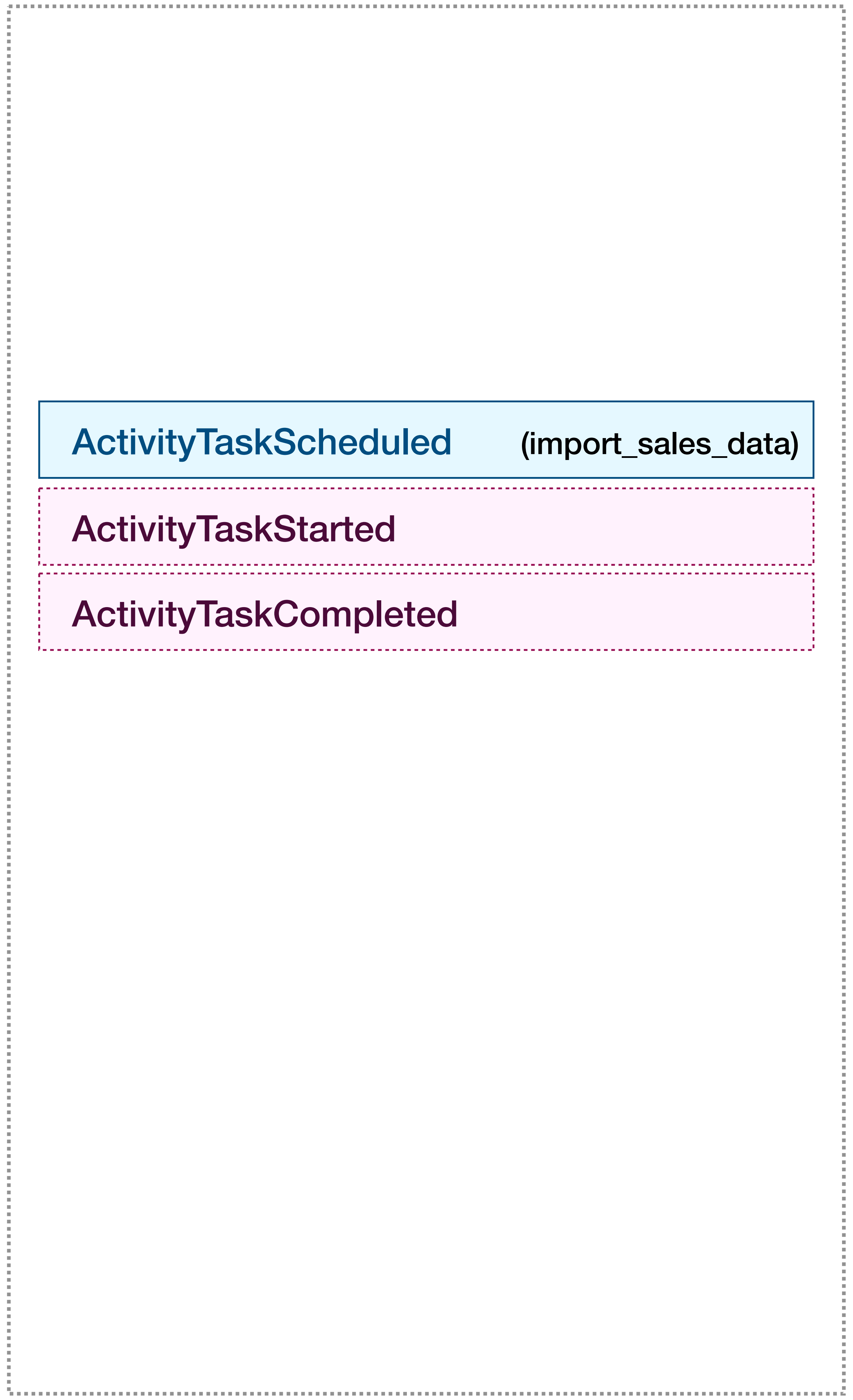
# A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```

## Commands Created



## Relevant Events Logged



# A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```

## Commands Created

**ScheduleActivityTask**
Type: import_sales_data

**StartTimer**
Duration: 4 hours

## Relevant Events Logged

ActivityTaskScheduled (import_sales_data)

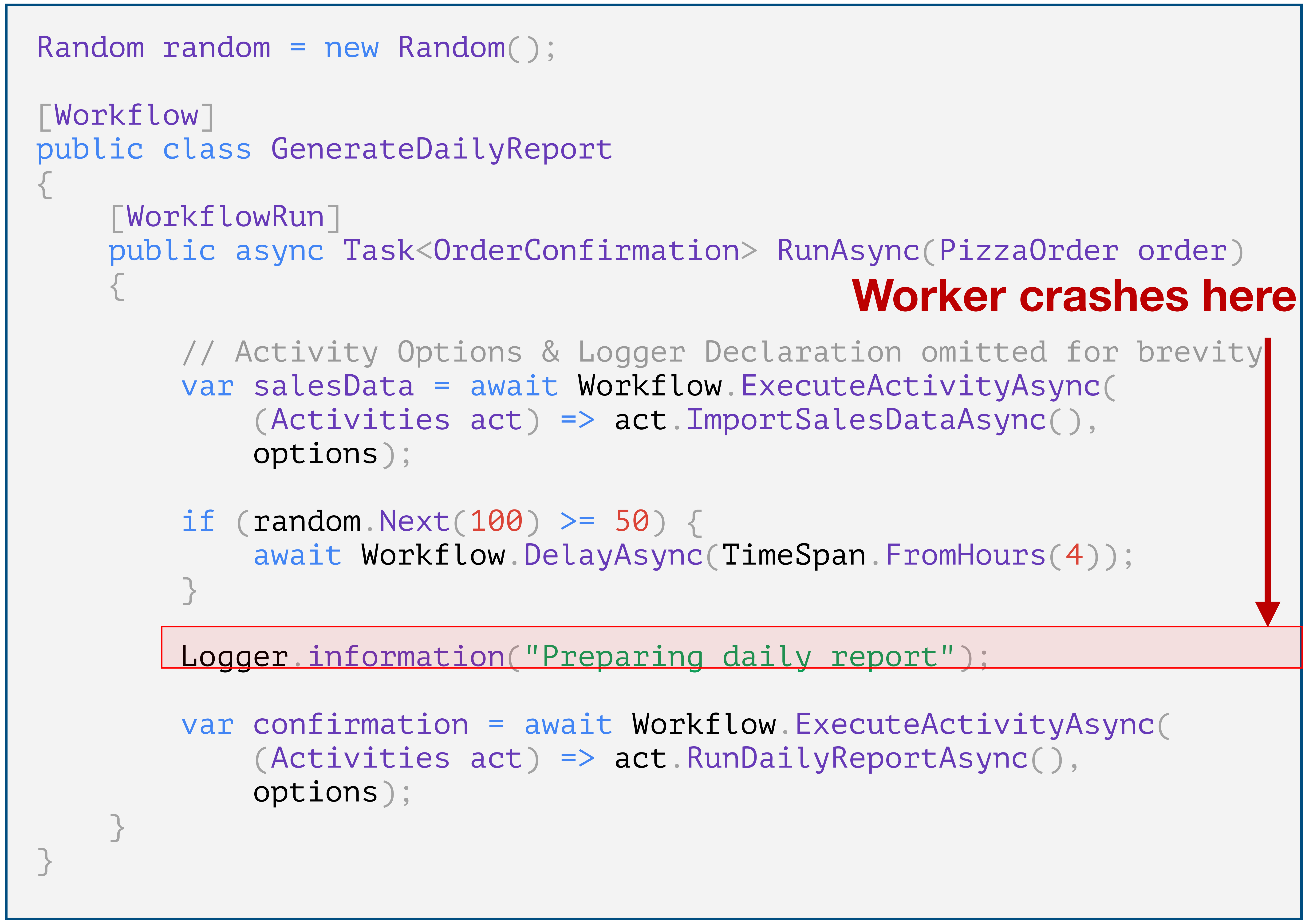
ActivityTaskStarted

ActivityTaskCompleted

TimerStarted (4 hours)

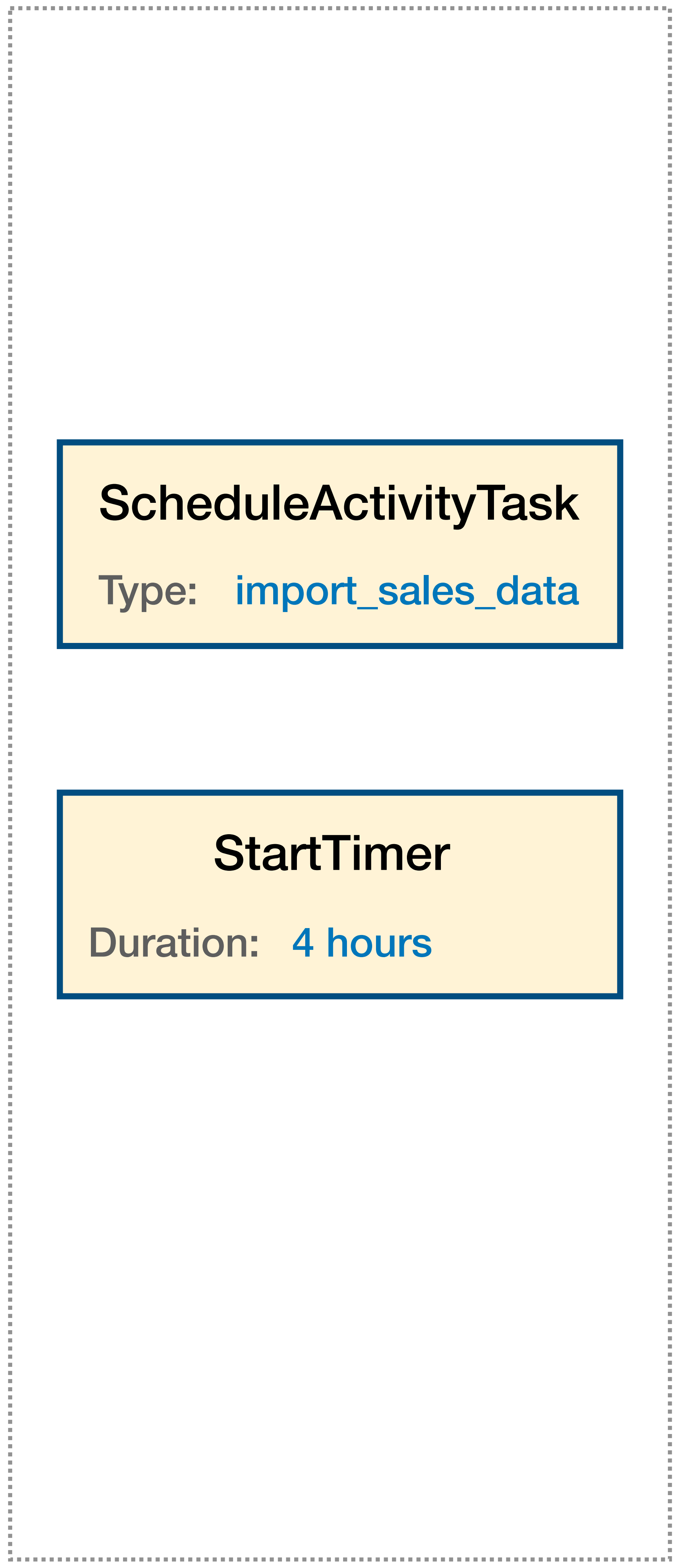
TimerFired

## A Non-Deterministic Workflow Definition

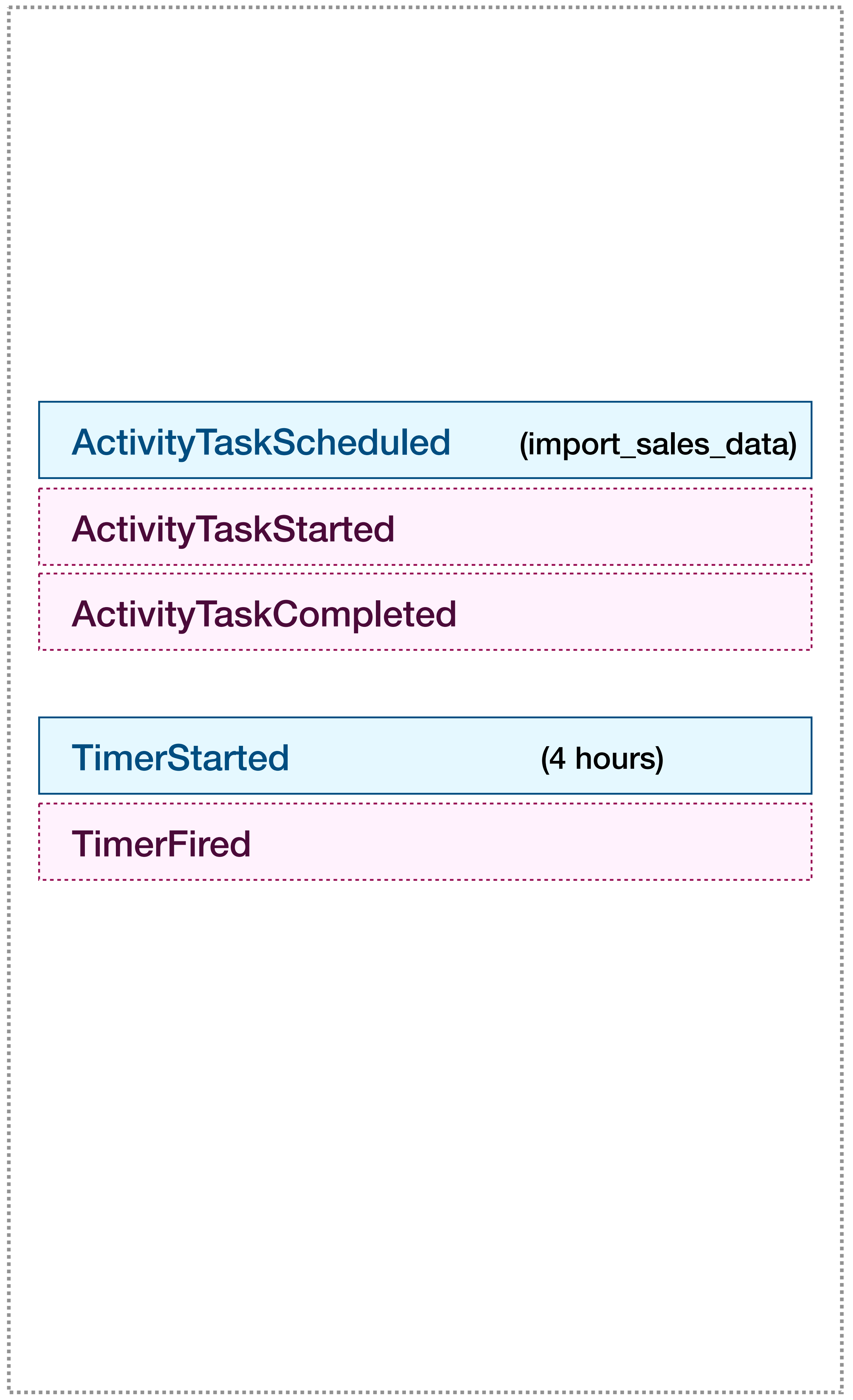


```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```

## Commands Created



## Relevant Events Logged



## A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```

## Commands Created

## Relevant History Events

| | |
|---|---|
| ActivityTaskScheduled | (import_sales_data) |
| ActivityTaskStarted | |
| ActivityTaskCompleted | |
| TimerStarted | (4 hours) |
| TimerFired | |

## Commands Expected (Based on History)

ScheduleActivityTask

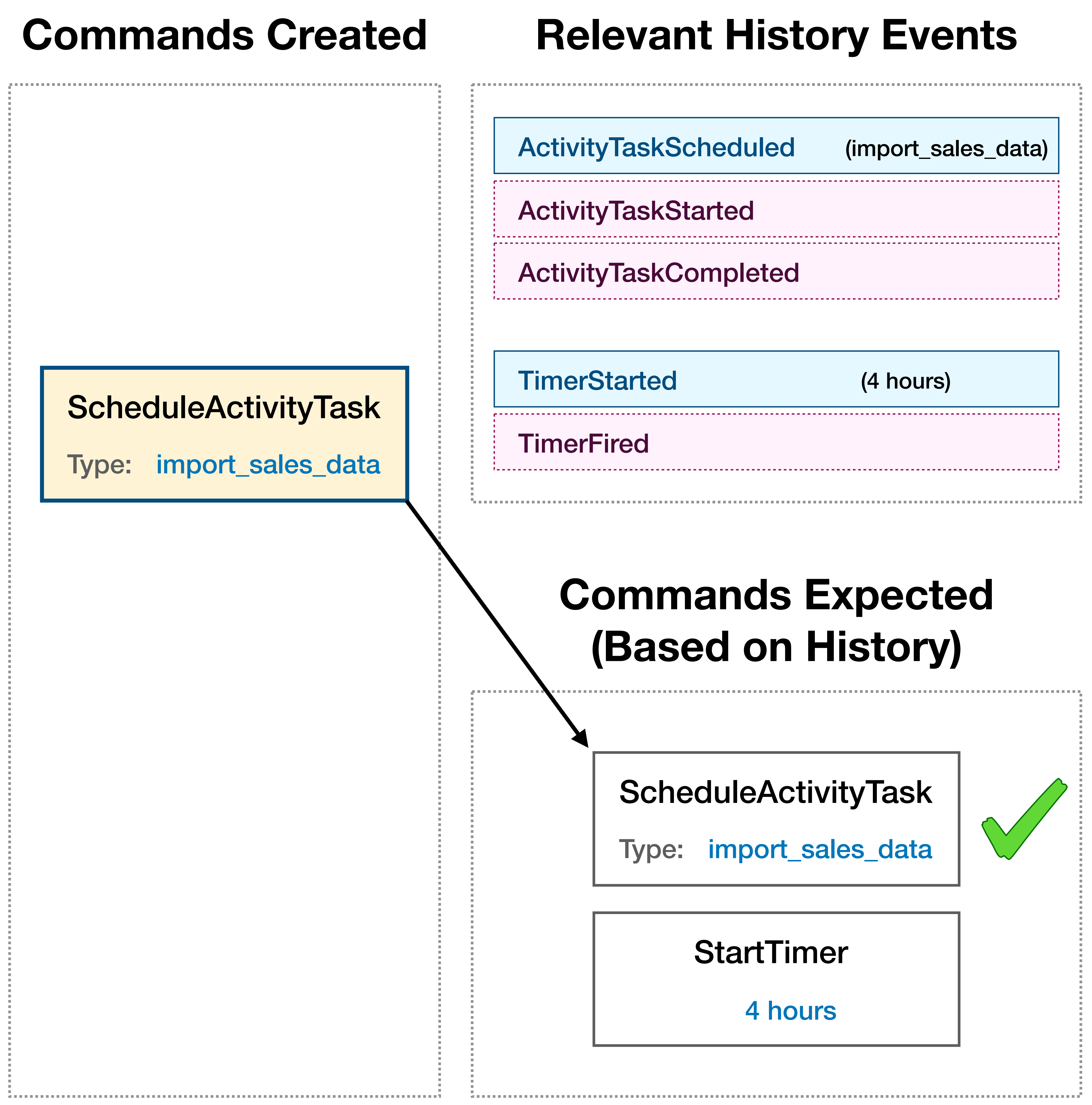
Type: import_sales_data

StartTimer

4 hours

# A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```



## A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAs
            (Activities act) => act.ImportSalesDataAsync
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```

Happens to return 14 during this execution

### Commands Created

ScheduleActivityTask
Type: import_sales_data

### Relevant History Events

| | |
|---|---|
| ActivityTaskScheduled | (import_sales_data) |
| ActivityTaskStarted | |
| ActivityTaskCompleted | |
| TimerStarted | (4 hours) |
| TimerFired | |

### Commands Expected (Based on History)

ScheduleActivityTask
Type: import_sales_data ✓

StartTimer
4 hours

## A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
  [WorkflowRun]
  public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
  {

    // Activity Options & Logger Declaration omitted for brevity
    var salesData = await Workflow.ExecuteActivityAsync(
      (Activities act) => act.ImportSalesDataAsync(),
      options);

    if (random.Next(100) >= 50) {
      await Workflow.DelayAsync(TimeSpan.FromHours(4));
    }

    Logger.information("Preparing daily report");

    var confirmation = await Workflow.ExecuteActivityAsync(
      (Activities act) => act.RunDailyReportAsync(),
      options);
  }
}
```

## Commands Created

**ScheduleActivityTask**
Type: import_sales_data

## Relevant History Events

| | |
|---|---|
| ActivityTaskScheduled | (import_sales_data) |
| ActivityTaskStarted | |
| ActivityTaskCompleted | |
| TimerStarted | (4 hours) |
| TimerFired | |

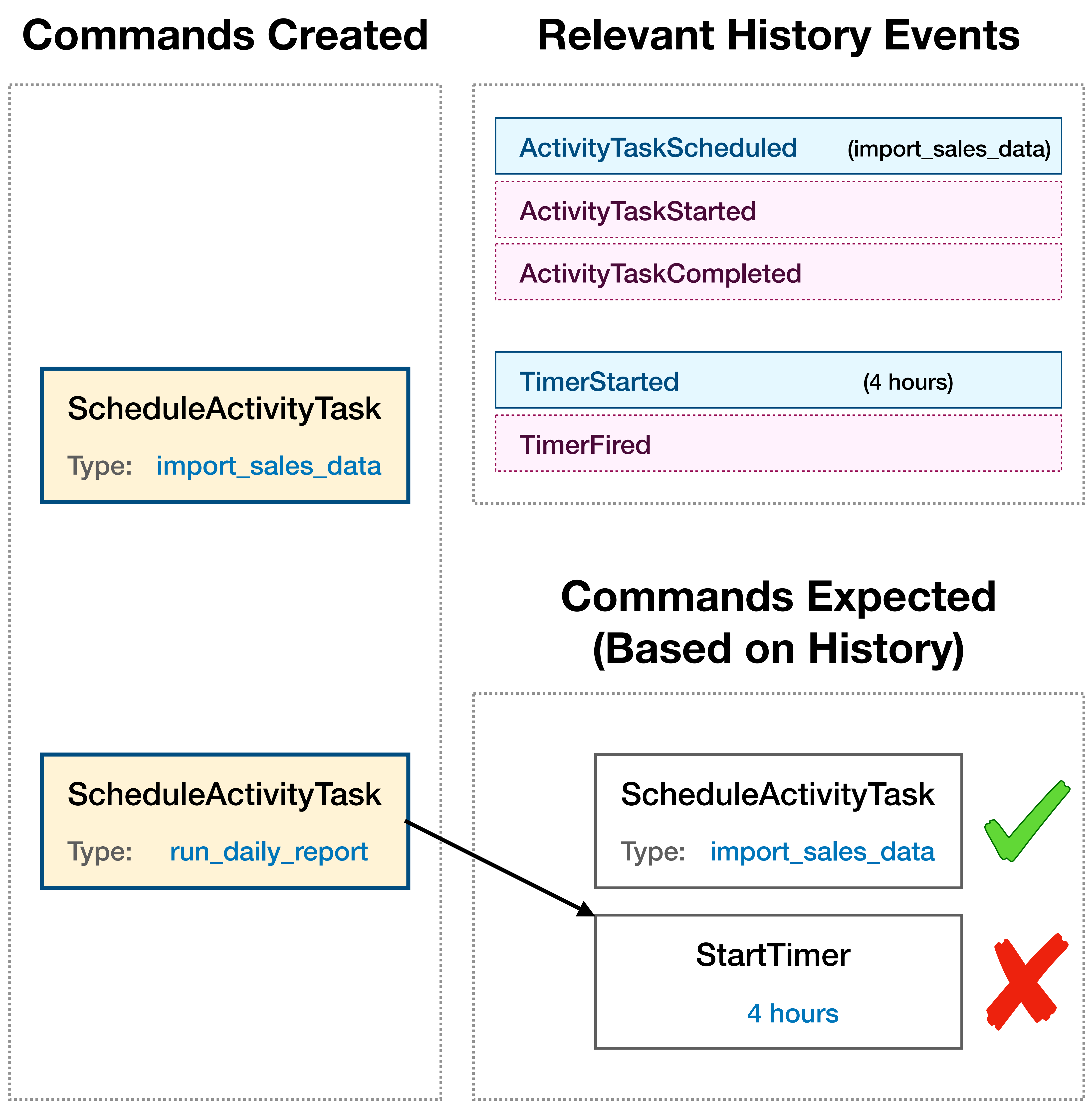
## Commands Expected (Based on History)

**ScheduleActivityTask**
Type: import_sales_data ✓

**StartTimer**
4 hours

## A Non-Deterministic Workflow Definition

```
Random random = new Random();

[Workflow]
public class GenerateDailyReport
{
    [WorkflowRun]
    public async Task<OrderConfirmation> RunAsync(PizzaOrder order)
    {

        // Activity Options & Logger Declaration omitted for brevity
        var salesData = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.ImportSalesDataAsync(),
            options);

        if (random.Next(100) >= 50) {
            await Workflow.DelayAsync(TimeSpan.FromHours(4));
        }

        Logger.information("Preparing daily report");

        var confirmation = await Workflow.ExecuteActivityAsync(
            (Activities act) => act.RunDailyReportAsync(),
            options);
    }
}
```



**Using random numbers in a Workflow Definition has resulted in Non-Deterministic Error**

Each time a particular Workflow Definition is executed with a given input, it must yield exactly the same commands in exactly the same order.

## Common Sources of Non-Determinism

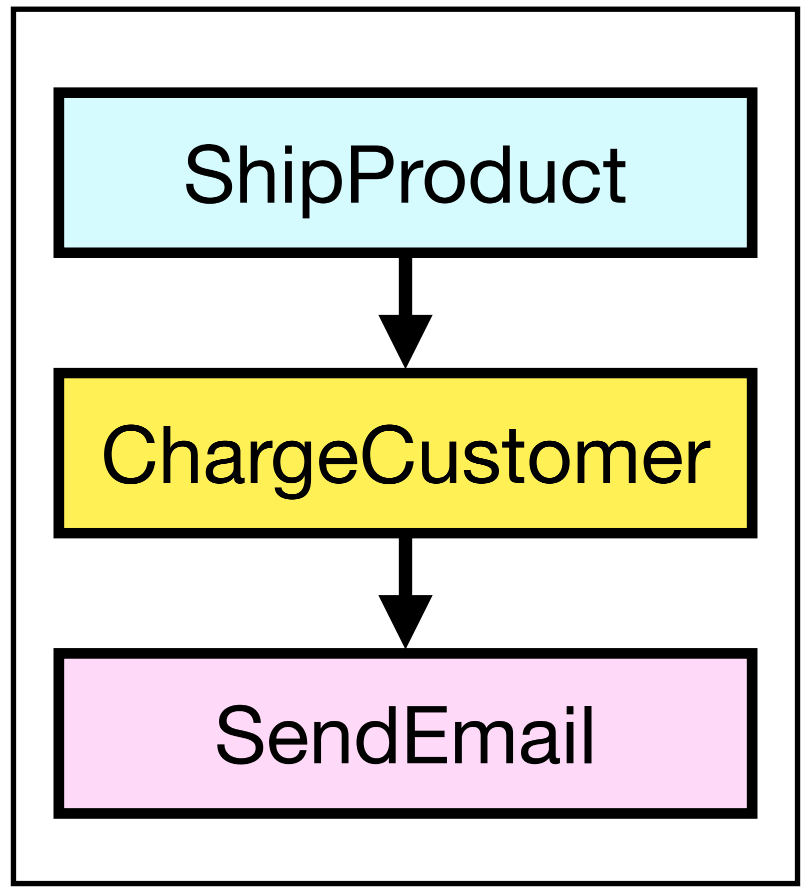
## Things to Avoid in a Workflow Definition

- **Using random numbers**
  - Use `Workflow.Random`
- **Accessing/mutating external systems, such as databases or network services**
  - Instead, use Activities to perform these operations
- **Writing business logic or calling methods that rely on system time**
  - Instead, use Workflow-safe methods such as `Workflow.UtcNow()` for system time
- **Working directly with Threads and Tasks**

## How Workflow Changes Can Lead to Non-Deterministic Errors

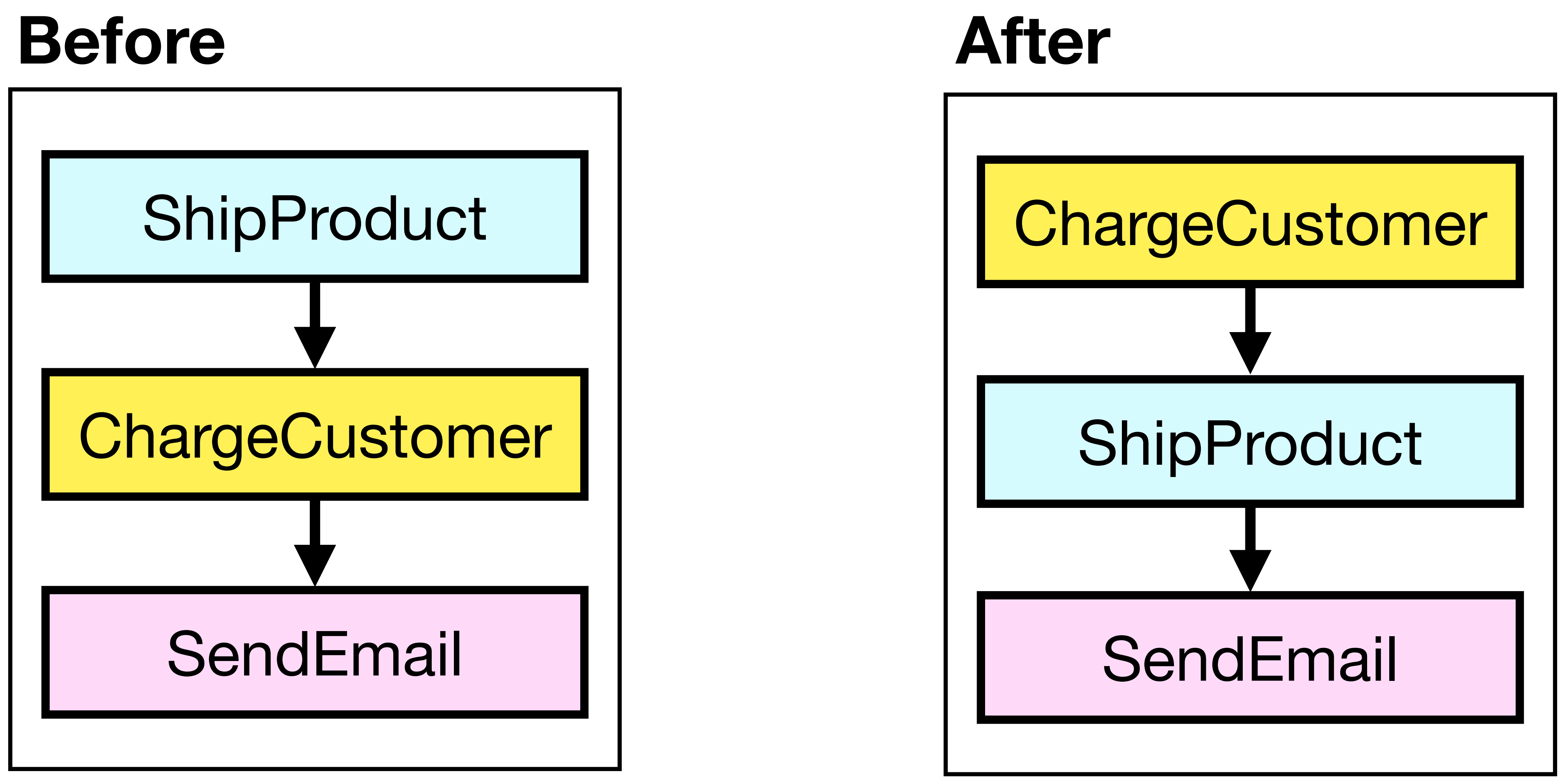
# Non-Deterministic *Code* Isn't the Only Danger

- **As you've just learned, non-deterministic code can cause problems**
  - However, there's also another source of non-deterministic errors



## Deployment Leads to Non-Deterministic Error

- **While that Workflow is running, you decide to update the code**



  - You deploy the updated code and restart the Worker(s) so that the change takes effect

- **What happens to the open execution when you restart the Worker?**

## Deployment Leads to Non-Deterministic Error

- **Problem: Worker cannot restore previous state with the updated code**
- **Only an issue if there are open executions at time of deployment**
- **How to detect?**
  - Test changes by replaying history of previous executions using new code before deploying
- **How to prevent?**
  - Versioning (see documentation for details)
- **How to remediate?**
  - Use Workflow Reset to restart execution to a point before the change was introduced

# Resetting A Workflow

- One way of overcoming a non-deterministic error that has been deployed
- Workflows can be reset to a specified point in the history
- Can be done via WebUI or CLI

```
$ temporal workflow reset \
        --workflow-id pizza-workflow-order-XD001 \
        --event-id 4 \
        --reason "Deployed an incompatible change (deleted Activity)"
```

# Temporal 102

## Validating Correctness of Temporal Application Code

- **The `Temporalio.Testing` module provides what you need**
  - It provides various tools to provide a runtime environment to test your Workflows and Activities
    - `WorkflowEnvironment` - Provides a runtime environment used to test a Workflow
      - You can "skip time" so you can test long-running Workflows without waiting
    - `ActivityEnvironment` - Similar to `WorkflowEnvironment`, but for Activities

# Testing Activities

```
public class MathActivities
{
    [Activity]
    public int Square(int number)
    {
        var logger = ActivityExecutionContext.Current.Logger;
        logger.LogInformation("Preparing to calculate the square");
        var result = number * number;
        return result;
    }
}
```

```
public class MathActivitiesTests
{
    [Fact]
    public async Task TestSquareWithPositiveNumber()
    {
        var env = new ActivityEnvironment();

        var activities = new MathActivities();

        var result = await env.RunAsync(
            () => activities.Square(3));
        Assert.Equal(9, result);
    }
}
```

## Testing Activities

```
public class MathActivities
{
    [Activity]
    public int Square(int number)
    {
        var logger = ActivityExecutionContext.Current.Logger;
        logger.LogInformation("Preparing to calculate the square");
        var result = number * number;
        return result;
    }
}
```

```
public class MathActivitiesTests
{
    [Fact]
    public async Task TestSquareWithPositiveNumber()
    {
        var env = new ActivityEnvironment();

        var activities = new MathActivities();

        var result = await env.RunAsync(
            () => activities.Square(3));
        Assert.Equal(9, result);
    }
}
```

## Testing Activities

```
public class MathActivities
{
    [Activity]
    public int Square(int number)
    {
        var logger = ActivityExecutionContext.Current.Logger;
        logger.LogInformation("Preparing to calculate the square");
        var result = number * number;
        return result;
    }
}
```

```
public class MathActivitiesTests
{
    [Fact]
    public async Task TestSquareWithPositiveNumber()
    {
        var env = new ActivityEnvironment();

        var activities = new MathActivities();

        var result = await env.RunAsync(
            () => activities.Square(3));
        Assert.Equal(9, result);
    }
}
```

# Testing Activities

```
public class MathActivities
{
    [Activity]
    public int Square(int number)
    {
        var logger = ActivityExecutionContext.Current.Logger;
        logger.LogInformation("Preparing to calculate the square");
        var result = number * number;
        return result;
    }
}
```

```
public class MathActivitiesTests
{
    [Fact]
    public async Task TestSquareWithPositiveNumber()
    {
        var env = new ActivityEnvironment();

        var activities = new MathActivities();

        var result = await env.RunAsync(
            () => activities.Square(3));
        Assert.Equal(9, result);
    }
}
```

# Testing Workflows - Definition

```
using Temporalio.Workflows;

namespace TemporalioExample;

[Workflow]
public class SumOfSquaresWorkflow
{
    [WorkflowRun]
    public async Task<int> RunAsync(int first, int second)
    {
        var options = new ActivityOptions { StartToCloseTimeout = TimeSpan.FromSeconds(5) };

        var squareOne = await Workflow.ExecuteActivityAsync(
            (MathActivities act) => act.Square(first),
            options);

        var squareTwo = await Workflow.ExecuteActivityAsync(
            (MathActivities act) => act.Square(second),
            options);

        return squareOne + squareTwo;
    }
}
```

# Testing Workflows - Test

```
public class SumOfSquaresWorkflowTests
{
    [Fact]
    public async Task TestSumOfSquaresWithPositiveNumbers()
    {
        await using var env = await WorkflowEnvironment.StartLocalAsync();

        var activities = new MathActivities();

        using var worker = new TemporalWorker(
            env.Client,
            new TemporalWorkerOptions($"task-queue-{Guid.NewGuid()}")
                .AddWorkflow<SumOfSquaresWorkflow>()
                .AddActivity(activities.Square));

        await worker.ExecuteAsync(async () =>
        {
            var result = await env.Client.ExecuteWorkflowAsync(
                (SumOfSquaresWorkflow wf) => wf.RunAsync(5, 6),
                new WorkflowOptions {
                    Id = $"wf-{Guid.NewGuid()}",
                    TaskQueue = worker.Options.TaskQueue!
                });

            Assert.Equal(61, result);
        });
    }
```

## Testing Workflows - Test

```
public class SumOfSquaresWorkflowTests
{
    [Fact]
    public async Task TestSumOfSquaresWithPositiveNumbers()
    {
        await using var env = await WorkflowEnvironment.StartLocalAsync();

        var activities = new MathActivities();

        using var worker = new TemporalWorker(
            env.Client,
            new TemporalWorkerOptions($"task-queue-{Guid.NewGuid()}")
                .AddWorkflow<SumOfSquaresWorkflow>()
                .AddActivity(activities.Square));

        await worker.ExecuteAsync(async () =>
        {
            var result = await env.Client.ExecuteWorkflowAsync(
                (SumOfSquaresWorkflow wf) => wf.RunAsync(5, 6),
                new WorkflowOptions {
                    Id = $"wf-{Guid.NewGuid()}",
                    TaskQueue = worker.Options.TaskQueue!
                });

            Assert.Equal(61, result);
        });
    }
```

# Testing Workflows - Test

```
public class SumOfSquaresWorkflowTests
{
    [Fact]
    public async Task TestSumOfSquaresWithPositiveNumbers()
    {
        await using var env = await WorkflowEnvironment.StartLocalAsync();

        var activities = new MathActivities();

        using var worker = new TemporalWorker(
            env.Client,
            new TemporalWorkerOptions($"task-queue-{Guid.NewGuid()}")
                .AddWorkflow<SumOfSquaresWorkflow>()
                .AddActivity(activities.Square));

        await worker.ExecuteAsync(async () =>
        {
            var result = await env.Client.ExecuteWorkflowAsync(
                (SumOfSquaresWorkflow wf) => wf.RunAsync(5, 6),
                new WorkflowOptions {
                    Id = $"wf-{Guid.NewGuid()}",
                    TaskQueue = worker.Options.TaskQueue!
                });

            Assert.Equal(61, result);
        });
    }
```

## Testing Workflows - Test

```
public class SumOfSquaresWorkflowTests
{
    [Fact]
    public async Task TestSumOfSquaresWithPositiveNumbers()
    {
        await using var env = await WorkflowEnvironment.StartLocalAsync();

        var activities = new MathActivities();

        using var worker = new TemporalWorker(
            env.Client,
            new TemporalWorkerOptions($"task-queue-{Guid.NewGuid()}")
                .AddWorkflow<SumOfSquaresWorkflow>()
                .AddActivity(activities.Square));

        await worker.ExecuteAsync(async () =>
        {
            var result = await env.Client.ExecuteWorkflowAsync(
                (SumOfSquaresWorkflow wf) => wf.RunAsync(5, 6),
                new WorkflowOptions {
                    Id = $"wf-{Guid.NewGuid()}",
                    TaskQueue = worker.Options.TaskQueue!
                });

            Assert.Equal(61, result);
        });
    }
```

# Mocking Activities in Workflow Tests

- **The Workflow test we wrote is an Integration Test!**
  - It invokes an Activity
  - If that Activity required external dependencies (API), that would have needed to be available
  - It's tightly coupled to both
- **Unit test Workflows by mocking Activities**
  - Define new replacement Activities

# Testing Workflows with Mocks

```
public class WorkflowMockTests
{
    [Fact]
    public async Task TestWithMockActivityAsync()
    {
        await using var env = await WorkflowEnvironment.StartTimeSkippingAsync();

        [Activity("RetrieveEstimate")]
        static Task<int> MockRetrieveEstimateAsync(string name) =>
            Task.FromResult(name == "Stanislav" ? 68 : 0);

        using var worker = new TemporalWorker(
            env.Client,
            new TemporalWorkerOptions("test-task-queue")
                .AddActivity(MockRetrieveEstimateAsync)
                .AddWorkflow<AgeEstimationWorkflow>());

        await worker.ExecuteAsync(async () =>
        {
            var result = await env.Client.ExecuteWorkflowAsync(
                (AgeEstimationWorkflow wf) => wf.RunAsync("Stanislav"),
                new WorkflowOptions
                {
                    Id = $"workflow-{Guid.NewGuid()}",
                    TaskQueue = "test-task-queue",
                });

            Assert.Equal("Stanislav has an estimated age of 68", result);
        });
    }
}
```

# Testing Workflows with Mocks

```csharp
public class WorkflowMockTests
{
    [Fact]
    public async Task TestWithMockActivityAsync()
    {
        await using var env = await WorkflowEnvironment.StartTimeSkippingAsync();

        [Activity("RetrieveEstimate")]
        static Task<int> MockRetrieveEstimateAsync(string name) =>
            Task.FromResult(name == "Stanislav" ? 68 : 0);

        using var worker = new TemporalWorker(
            env.Client,
            new TemporalWorkerOptions("test-task-queue")
                .AddActivity(MockRetrieveEstimateAsync)
                .AddWorkflow<AgeEstimationWorkflow>());

        await worker.ExecuteAsync(async () =>
        {
            var result = await env.Client.ExecuteWorkflowAsync(
                (AgeEstimationWorkflow wf) => wf.RunAsync("Stanislav"),
                new WorkflowOptions
                {
                    Id = $"workflow-{Guid.NewGuid()}",
                    TaskQueue = "test-task-queue",
                });

            Assert.Equal("Stanislav has an estimated age of 68", result);
        });
    }
}
```

## Running Tests

```
$ dotnet test
```

## Exercise #2: Testing the Translation Workflow

- **During this exercise, you will**
  - Write code to execute the Workflow in the test environment
  - Develop a Mock Activity for the translation service call
  - Observe time-skipping in the test environment
  - Write unit tests for the Activity implementation
  - Run the tests from the command line to verify correct behavior
- **Refer to this exercise's `README.md` file for details**
  - Don't forget to make your changes in the `practice` subdirectory

t.mp/edu-102-dotnet-code

## Review

- Temporal's .NET SDK provides support for testing Workflows and Activities
- You can test Activities in isolation
- You can test Workflows quickly, even if they have Timers

## Temporal 102

## Demo:
## Debugging a Workflow that Doesn’t Progress

## Demo:
## Interpreting Event History

## Demo: Terminating a Workflow Execution with the Web UI

## Exercise #3: Debugging and Fixing an Activity Failure

- **During this exercise, you will**
  - Start a Worker and run a basic Workflow for processing a pizza order
  - Use the Web UI to find details about the execution
  - Diagnose and fix a latent bug in the Activity Definition
  - Test and deploy the fix
  - Verify that the Workflow now completes successfully
- **Refer to this exercise's `README.md` file for details**
  - Don't forget to make your changes in the `practice` subdirectory

t.mp/edu-102-dotnet-code

## Temporal 102

# Temporal Service Roles

## Frontend

An API Gateway that validates and routes inbound calls

## History

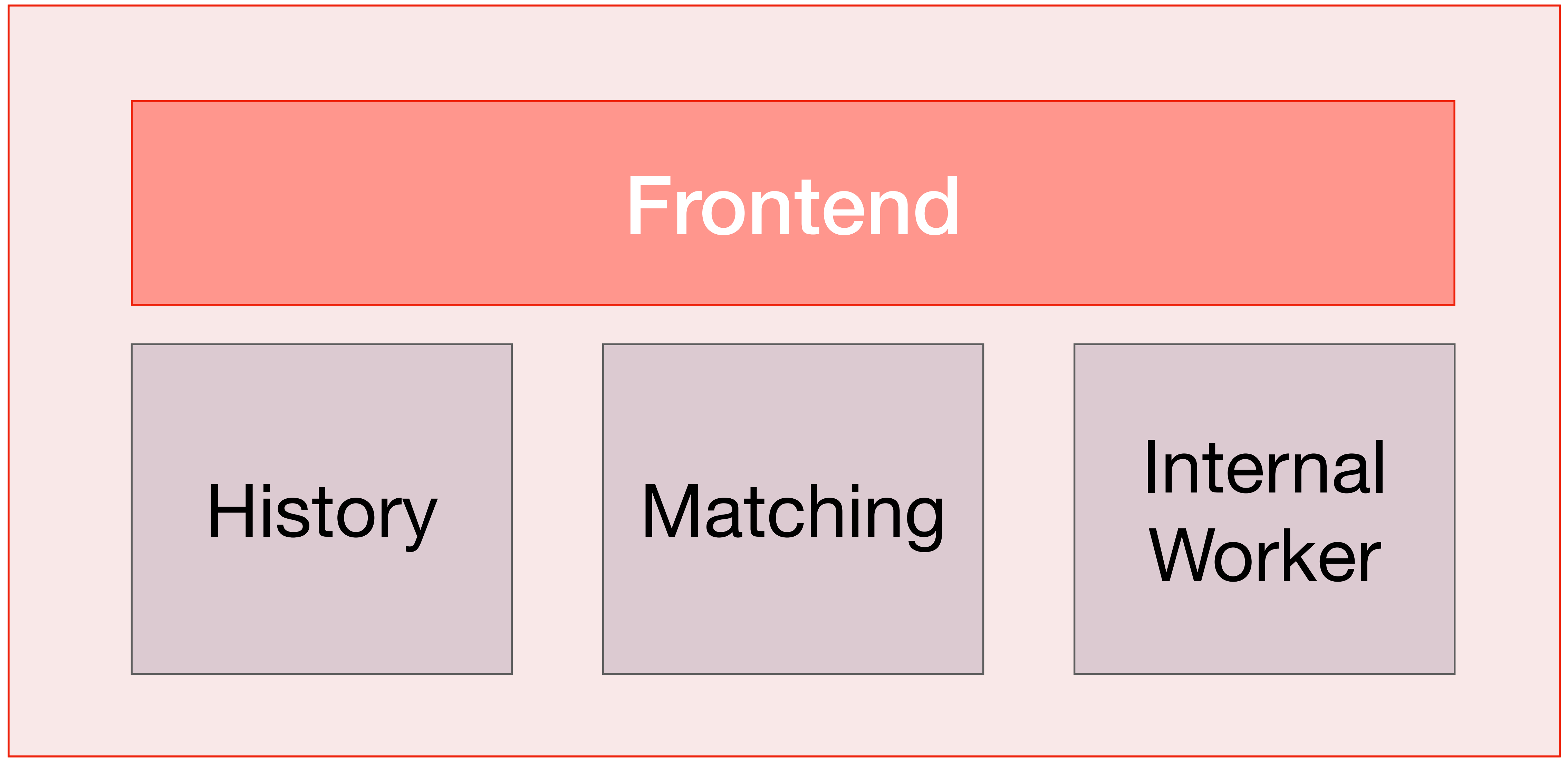
Maintains history and moves execution progress forward

## Matching

Hosts Task Queues and matches Workers with Tasks

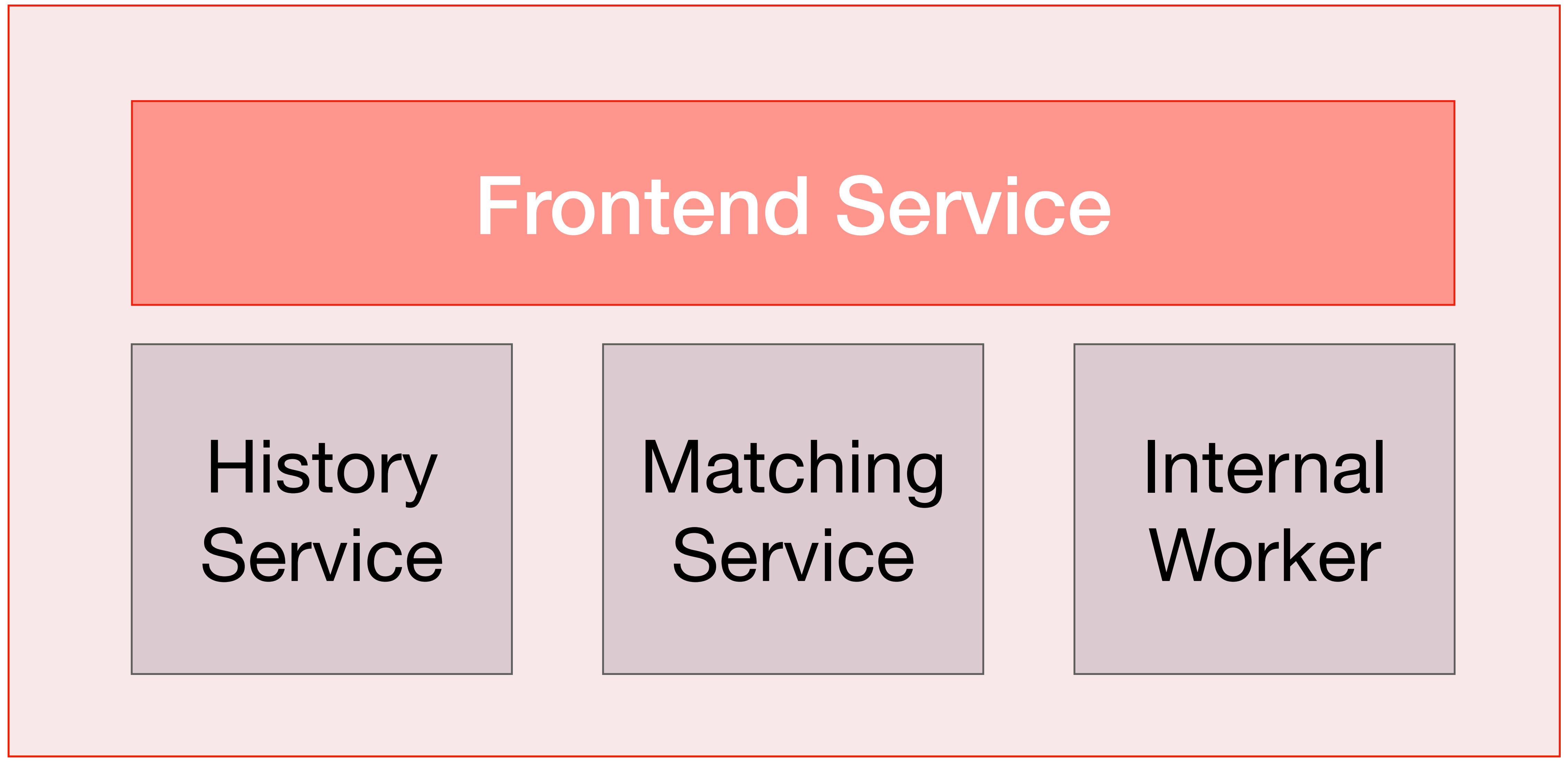
## Internal Worker

Runs Workflows that are internal to the system



## Internal Worker

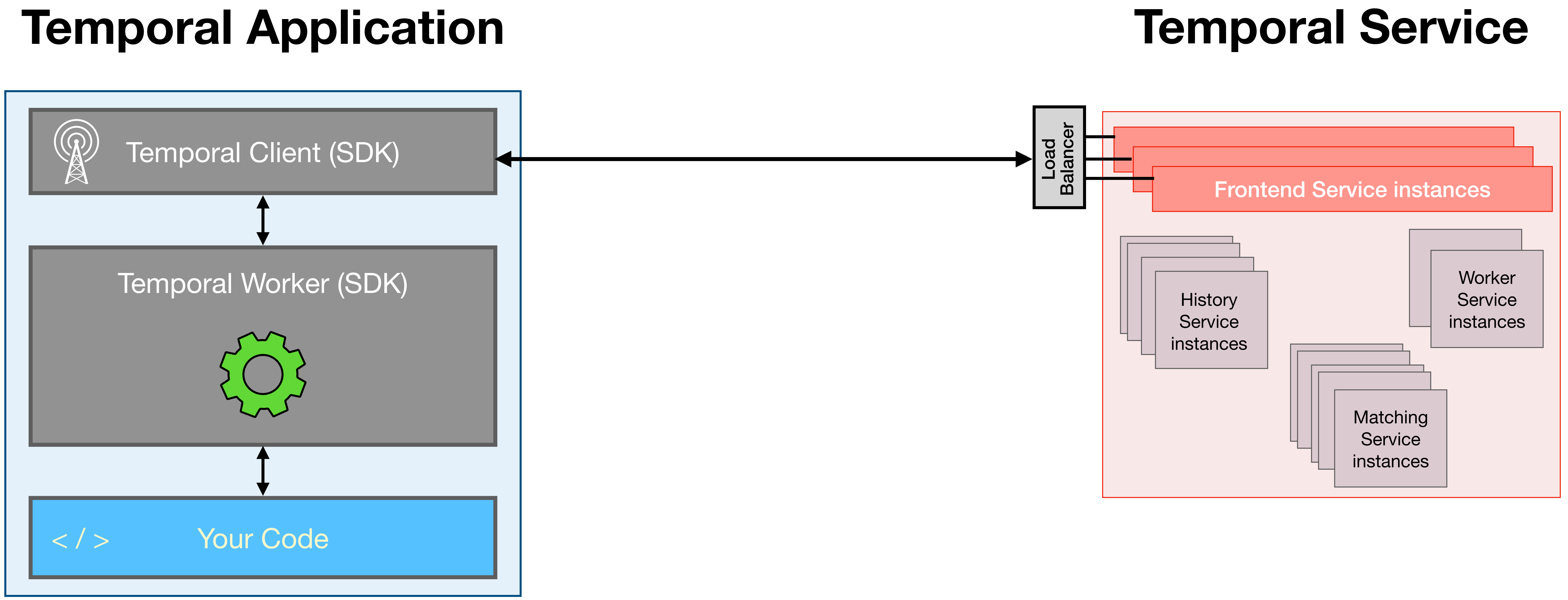
- The Internal Workflows it runs are not exposed to users.
- The service name is coincidental - it has no relationship to the Worker that's part of your application.



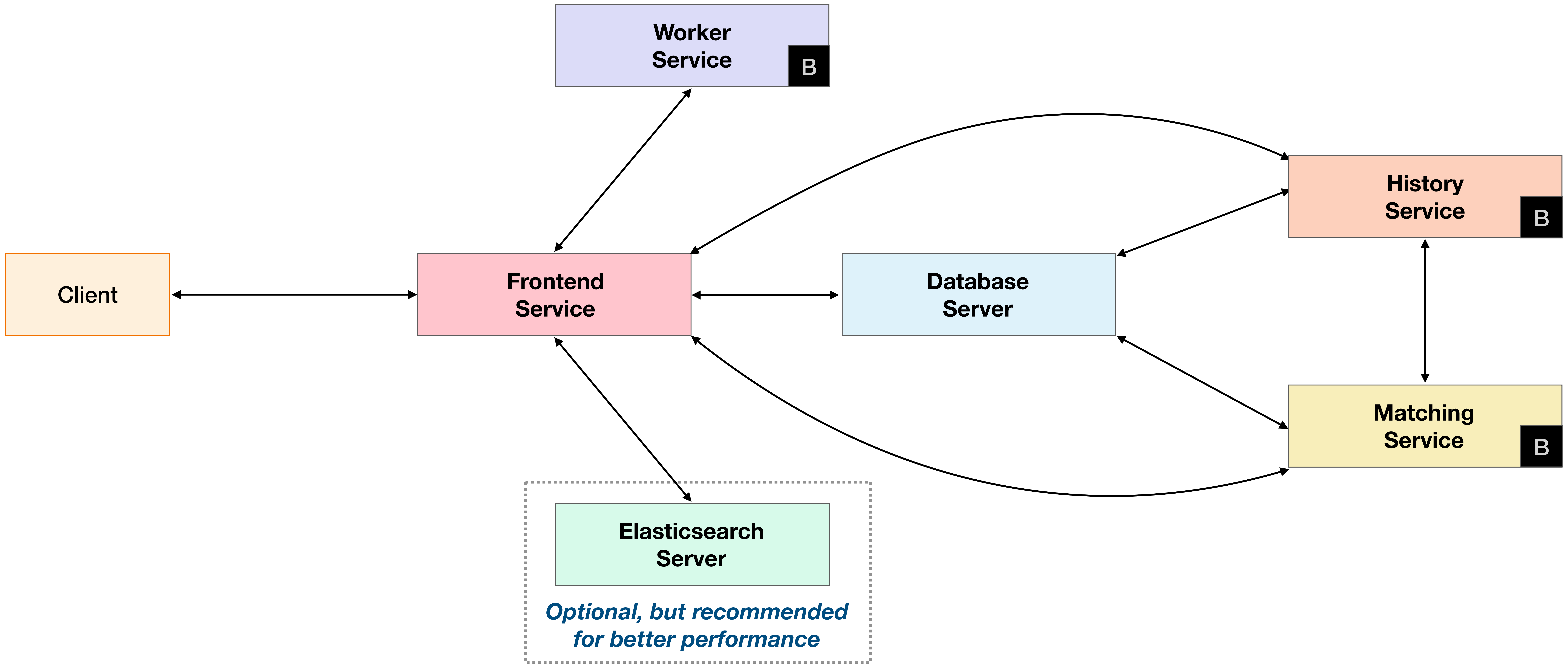
## Service Scalability

- A Temporal Service can scale with multiple instances of each service

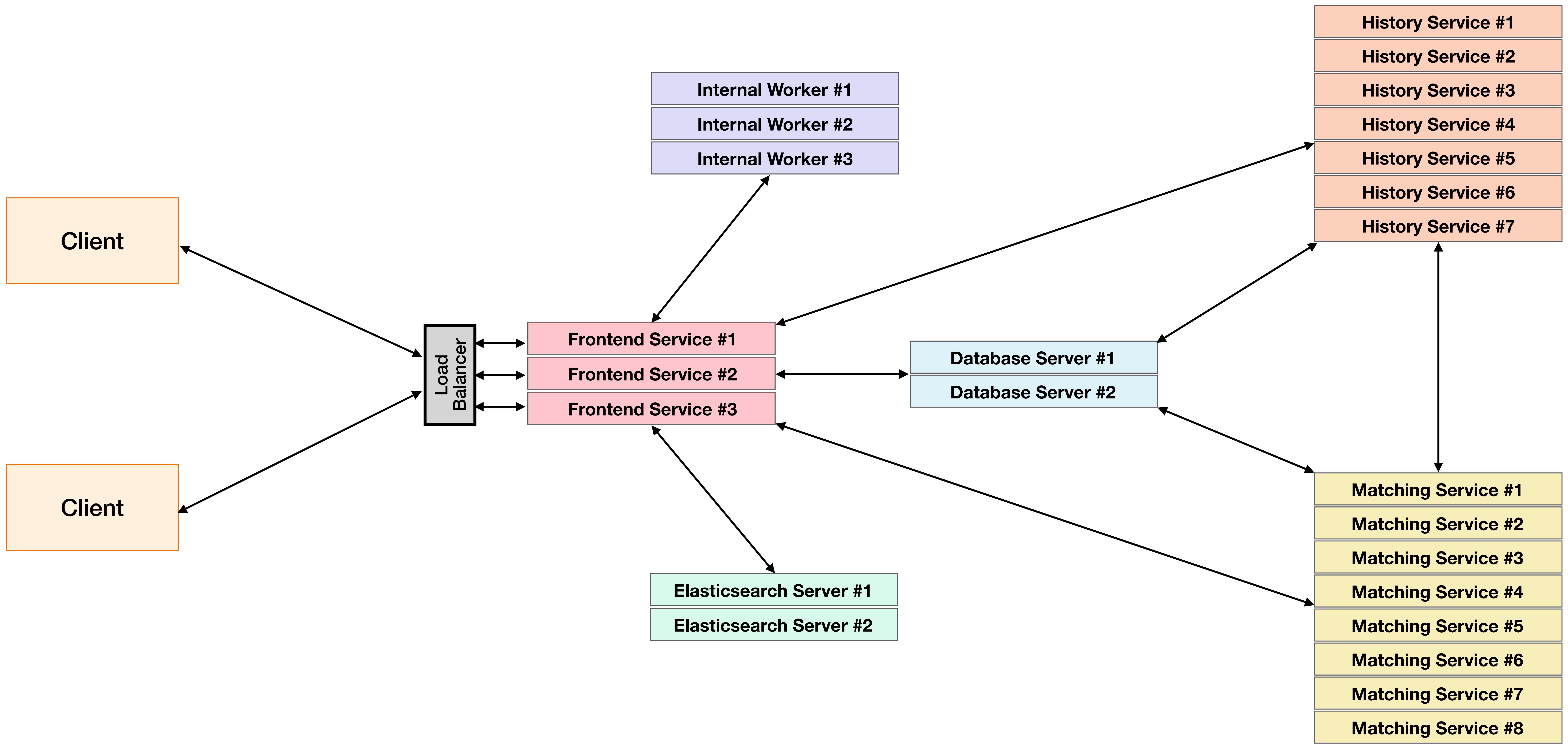
# Service Scalability



# Connectivity (Logical)



# Connectivity (Physical)



## Default Options for a Temporal Client

- **The following code example shows how to create a Temporal Client**
  - This will expect a Frontend Service running on `localhost` at TCP port 7233

```
# create the local connection
var client = await TemporalClient.ConnectAsync(new("localhost:7233"));
```

## Configuring Client for a Non-Local Service

- **This example specifies a namespace, but not parameters needed for TLS**

```
var client = await TemporalClient.ConnectAsync(new()
{
    TargetHost = "myservice.example.com:7233",
    Namespace = "my-namespace",
});
```

- The options shown above are equivalent to those in the following `temporal` command

```
$ temporal workflow list --address myservice.example.com:7233 --namespace abc
```

## Configuring Client for a Secure Service

- **This example shows Client configuration for a secure non-local service**

```
using Temporalio.Client;

var client = await TemporalClient.ConnectAsync(new("my-namespace.a1b2c.tmprl.cloud:7233")
{
    Namespace = "my-namespace.a1b2c",
    Tls = new()
    {
        ClientCert = await File.ReadAllBytesAsync("my-cert.pem"),
        ClientPrivateKey = await File.ReadAllBytesAsync("my-key.pem"),
    },
});
```

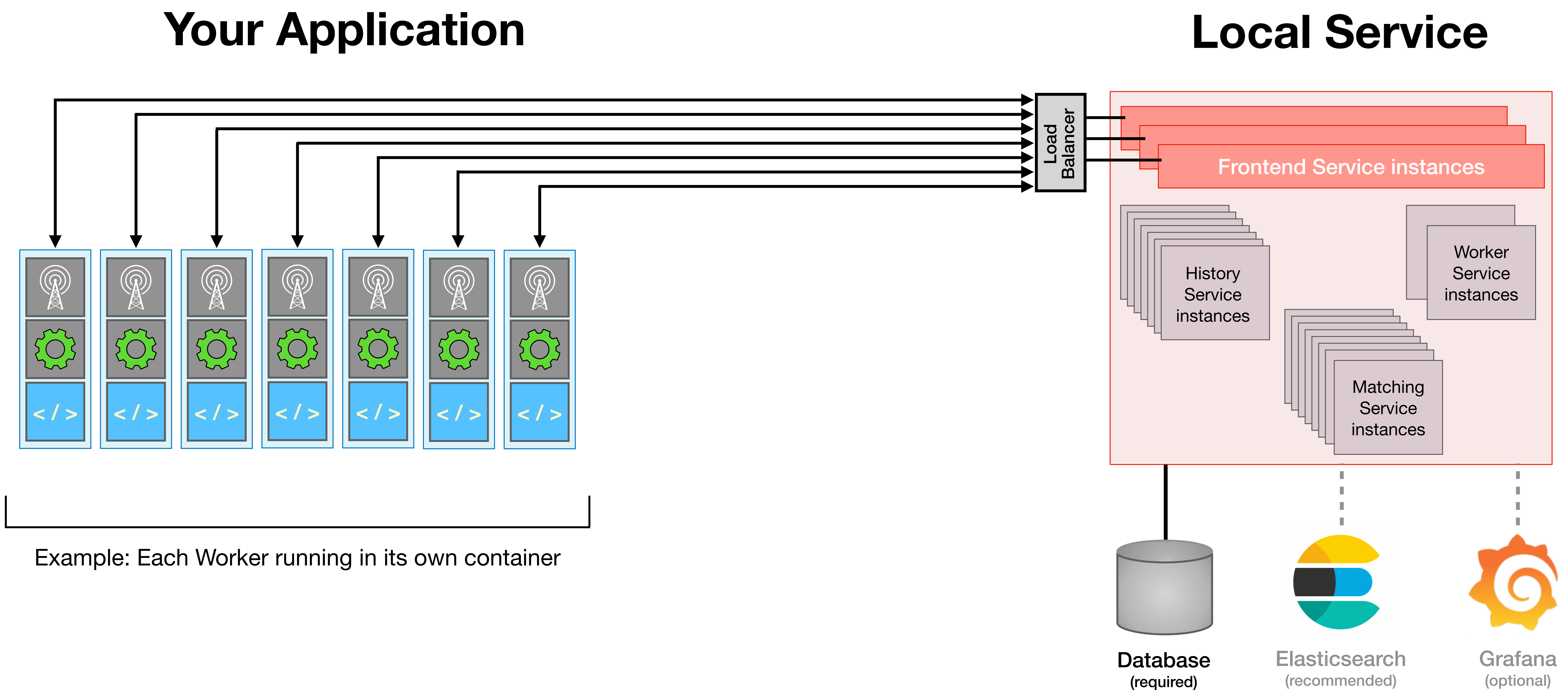
# Building a Temporal Application

- **Application deployment is usually preceded by a build process**
  - The tools used to do this vary by language, based on the SDK(s) used
  - Temporal does not require the use of any particular tools
  - You can use what is typical for the language or mandated by your organization
- **With the .NET SDK, you can package the Worker using commands such `dotnet build` or other tools of your choice**
  - The result is what you would deploy and run in production
  - It must contain all dependencies required at runtime

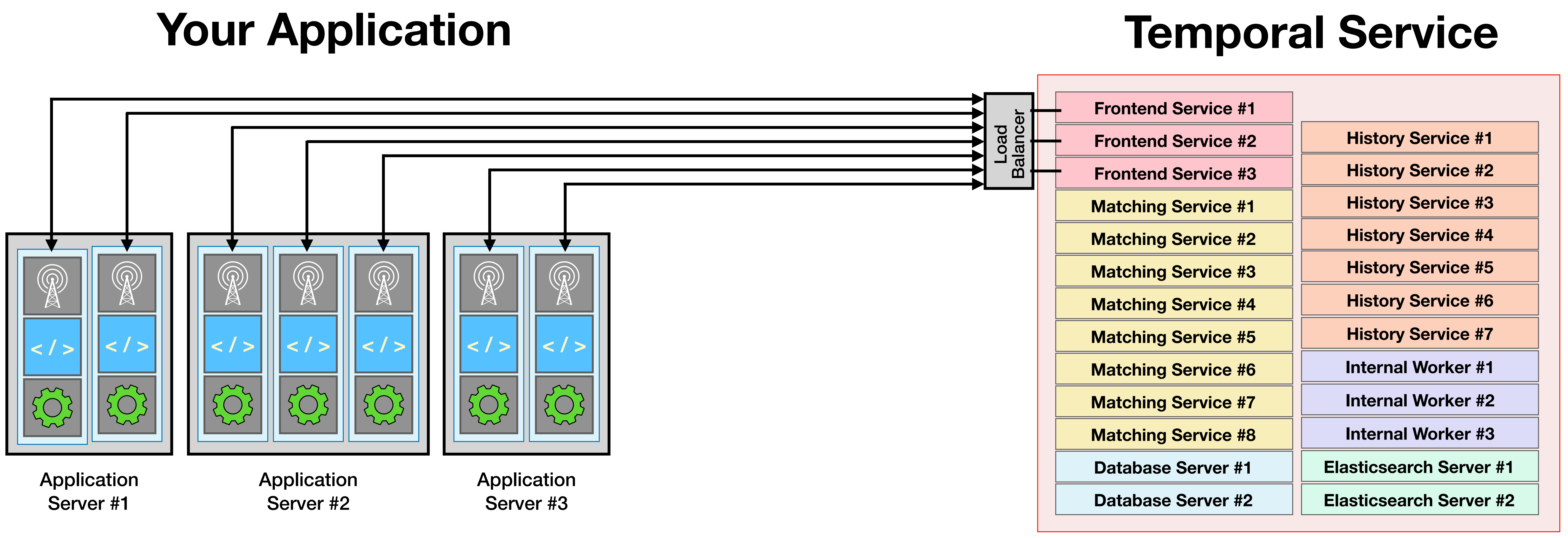
# Temporal Application Deployment

- **Once built, you'll deploy the application to production**
  - This will contain your code (e.g., Worker, Client, etc.)
  - Ensure any needed dependencies are available at runtime
    - For example, database drivers used by your application
    - For example, the Java runtime or Python interpreter for polyglot Temporal applications
- **Temporal is not opinionated about how or where you deploy the code**
  - Key point: Workers run externally to Temporal Service or Cloud
  - It's up to you how you run the Workers: bare metal, virtual machines, containers, etc.
  - Let's quickly look at two possible examples

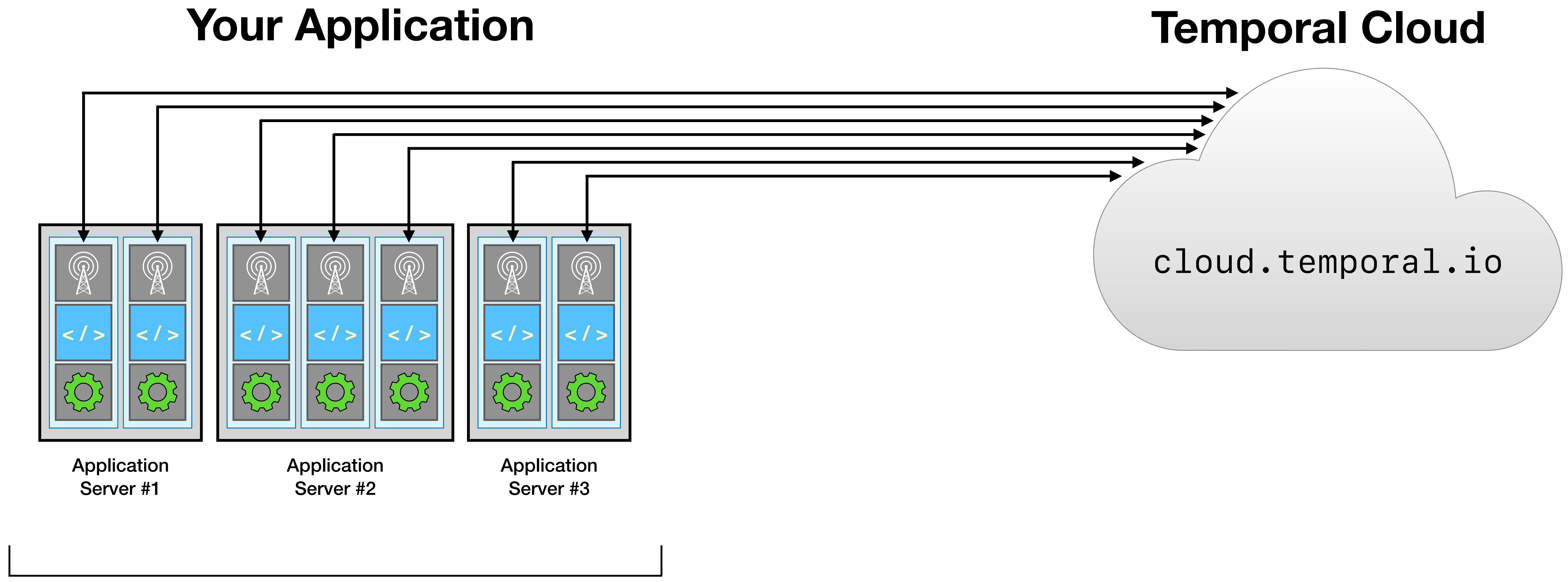
# Deployment Scenario #1



# Physical View of an Application in Production



# Deployment Scenario #2



Example: Multiple Worker Processes distributed across bare metal

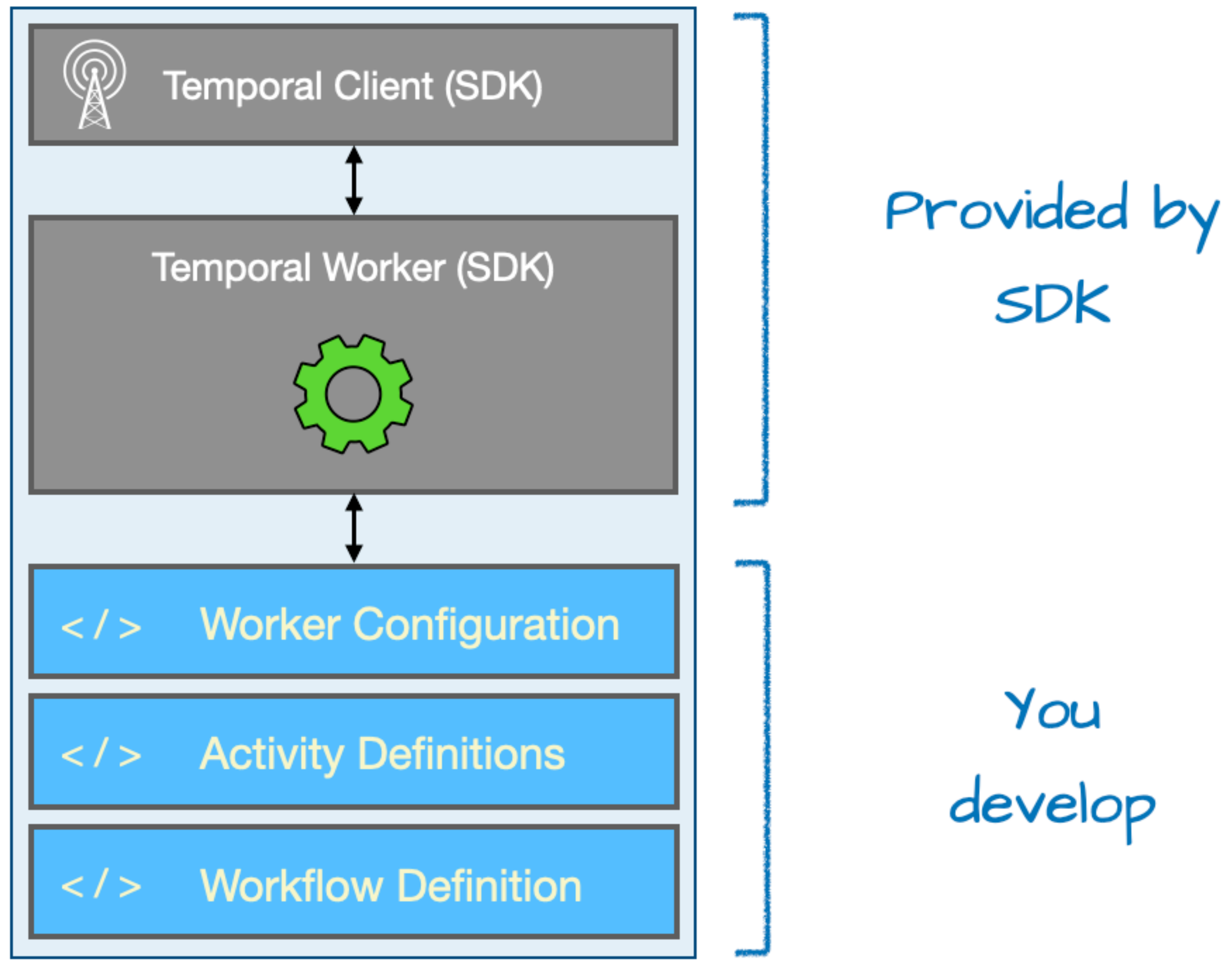
## Review

- Temporal Services have four parts:
  - Frontend Service, History Service, Matching Service, and Internal Worker
- To connect to a Temporal Service, you can specify the address, the namespace, and provide certificates and keys for mTLS connections
- Use your existing build processes to prepare your app
  - You can bundle Workflows to improve production performance
- Temporal is not opinionated about how or where you deploy the code
  - You run your Workers, Activities, and Workflows on your own servers
  - You can run a Temporal Service on your own servers or you can use Temporal Cloud.

## Temporal 102

## Essential Points (1)

- **Temporal applications contain code that you develop**
  - Workflow and Activity Definitions, Worker Configuration, etc.
- **Temporal applications also contain SDK-provided code**
  - Such as the implementations of the Worker and Temporal Client
- **Temporal guarantees durable execution of Workflows**
  - If the Worker crashes, another Worker uses History Replay to automatically recreate pre-crash state, then continues execution
  - From the developer perspective, it's as if the crash never even happened



## Essential Points (2)

- **Temporal Service / Cloud perform orchestration via Task Queues**
  - A Worker polls a Task Queue, accepts a Task, executes the code, and reports back with status/results
  - Communication takes place by Workers initiating requests via gRPC to the Frontend Service
  - **Key point**: Execution of the code is external to Temporal Service / Cloud
- **As Workers run your code, they send Commands to the Temporal Service**
  - For example, when encountering calls to Activity Methods or `Workflow.DelayAsync` or when returning a result from the Workflow Definition
- **Commands sent by the Worker lead to Events logged by the Temporal Service**

## Essential Points (3)

- **The Event History documents the details of a Workflow Execution**
    - It's an ordered append-only list of Events
    - Temporal enforces limits on the size and item count of the Event History
- **Every Event has three attributes in common: ID, timestamp, and type**
    - They will also have additional attributes, which vary by Event Type
    - Examining the Event History and attributes of individual Events can help you debug Workflow Executions

# Essential Points (4)

- **A single Workflow Definition can be executed any number of times**
  - Each time potentially having different input data and a different Workflow ID
    - At most, one open Workflow Execution with a given Workflow ID is allowed per Namespace
    - This rule applies to *all* Workflow Executions, not just ones of the same Workflow Type
- **Once started, Workflow Execution enters the Open state**
  - Execution typically alternates between making progress and awaiting a condition
  - When execution concludes, it transitions to the Closed state
  - There are several subtypes of Closed, including Completed, Failed, and Terminated

## Essential Points (5)

- **Temporal requires that your Workflow code is deterministic**
    - This constraint is what makes durable execution possible
    - Temporal's definition of determinism: Every execution of a given Workflow Definition must produce an identical sequence of Commands, given the same input
    - Non-deterministic errors can occur because of something inherently non-deterministic in the code
        - Can also occur after deploying a code change that changes the Command sequence, if there were open executions of the same Workflow Type at the time of deployment
- **Activities are used for code that interacts with the outside world**
    - Activity code isn't required to be deterministic
    - Activities are automatically retried upon failure, according to a configurable Retry Policy

## Essential Points (6)

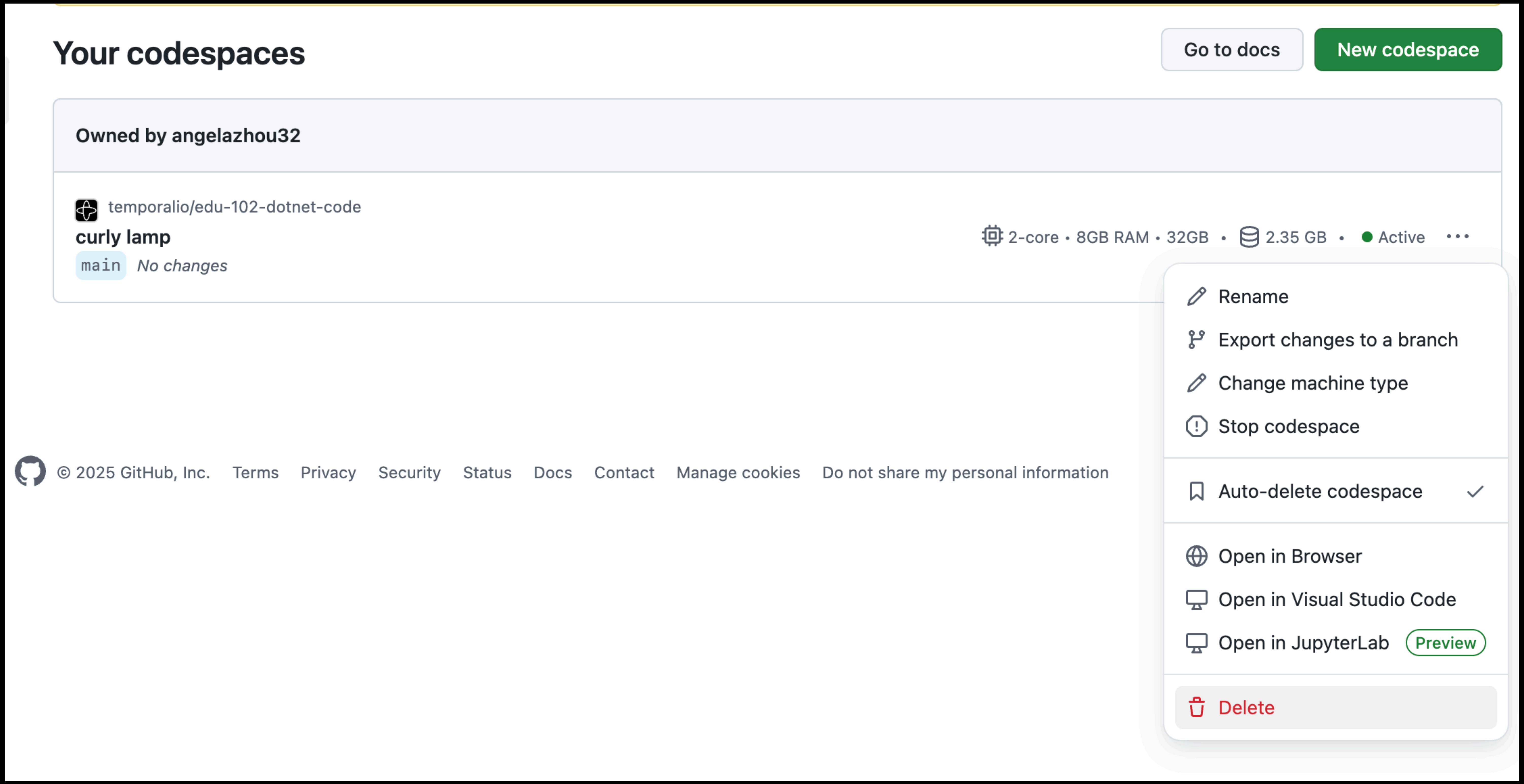
- **Recommended best practices for Temporal app development**
  - Use serializable objects such as records (not individual parameters) as input/output of your Workflow and Activity definitions
  - Be aware of the platform's limits on Event History size and item count
  - Replace non-deterministic code in Workflow Definitions with Workflow-safe counterparts
  - Use Temporal's replay-aware logging API

## Essential Points (7)

- **We don't dictate how to build, deploy, or run Temporal applications**
  - Typical advice: Build, deploy, and run as you would any other application in that language
  - However, we recommend running >= 2 Workers per Task Queue (availability/scalability)

# Don't forget to manually delete your code spaces

https://github.com/codespaces



Your codespaces
Go to docs
New codespace
Owned by angelazhou32
temporalio/edu-102-dotnet-code
curly lamp
main
No changes
2-core • 8GB RAM • 32GB • 2.35 GB • Active
Rename
Export changes to a branch
Change machine type
Stop codespace
Auto-delete codespace ✓
Open in Browser
Open in Visual Studio Code
Open in JupyterLab Preview
Delete
© 2025 GitHub, Inc. Terms Privacy Security Status Docs Contact Manage cookies Do not share my personal information

Thank you for your time and attention

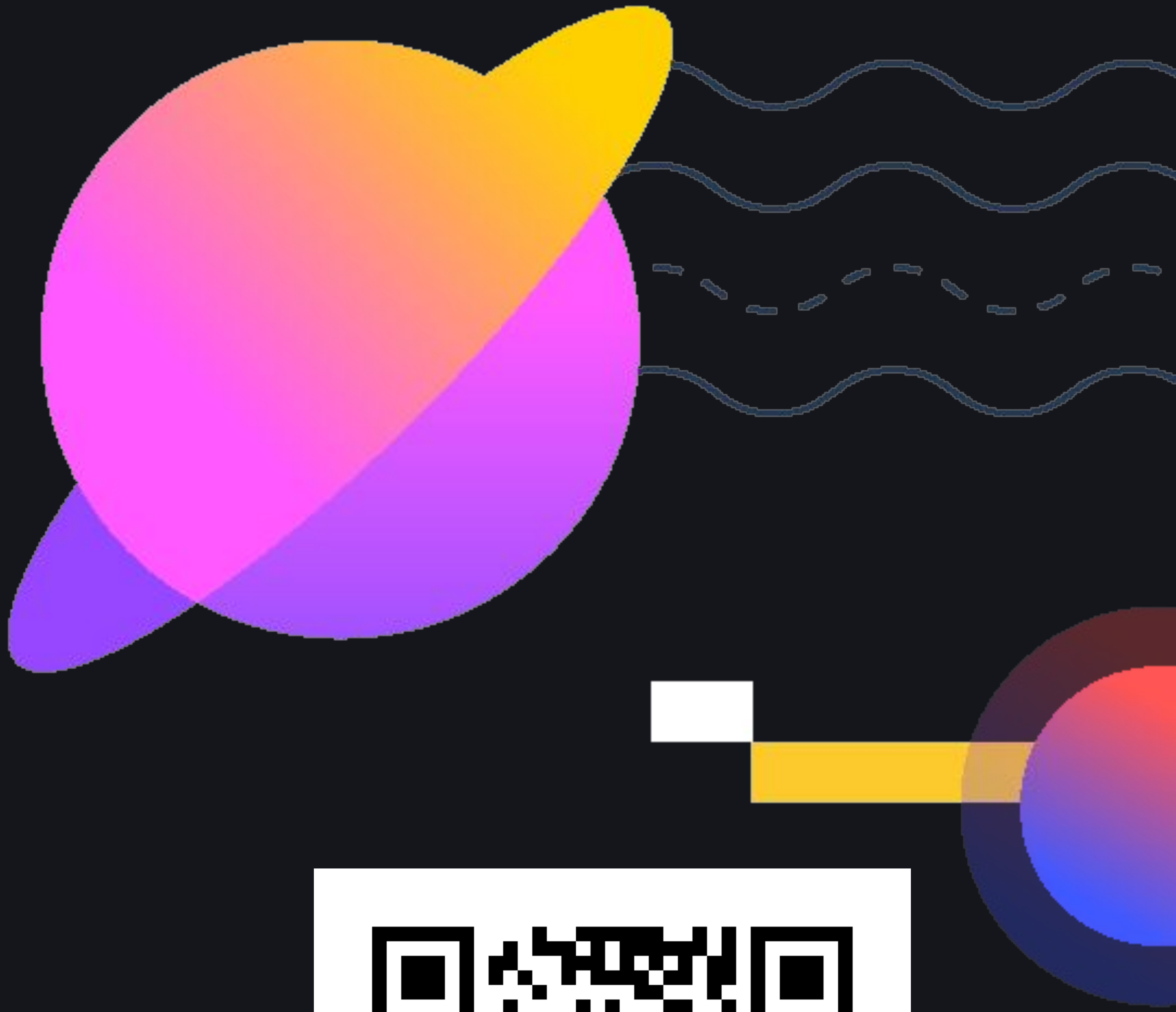
We welcome your feedback

t.mp/replay25ws

TEMPORAL'S CODE EXCHANGE

## Share what you've built with Temporal

Temporal has a thriving community building code for each other – we'd love to see what you've built!



TEMPORAL.IO/CODE-EXCHANGE