Temporal 102

Java

Live Edition for Replay

20230825a

This version of Temporal 102 is abridged from the version that is on our education site. However, after Replay this course is going to be split in two.

Temporal 102

▸ **00. About this Workshop**

01. Understanding Key Concepts in Temporal

02. Improving Your Temporal Application Code

03. Using Timers in a Workflow Definition

04. Testing Your Temporal Application Code

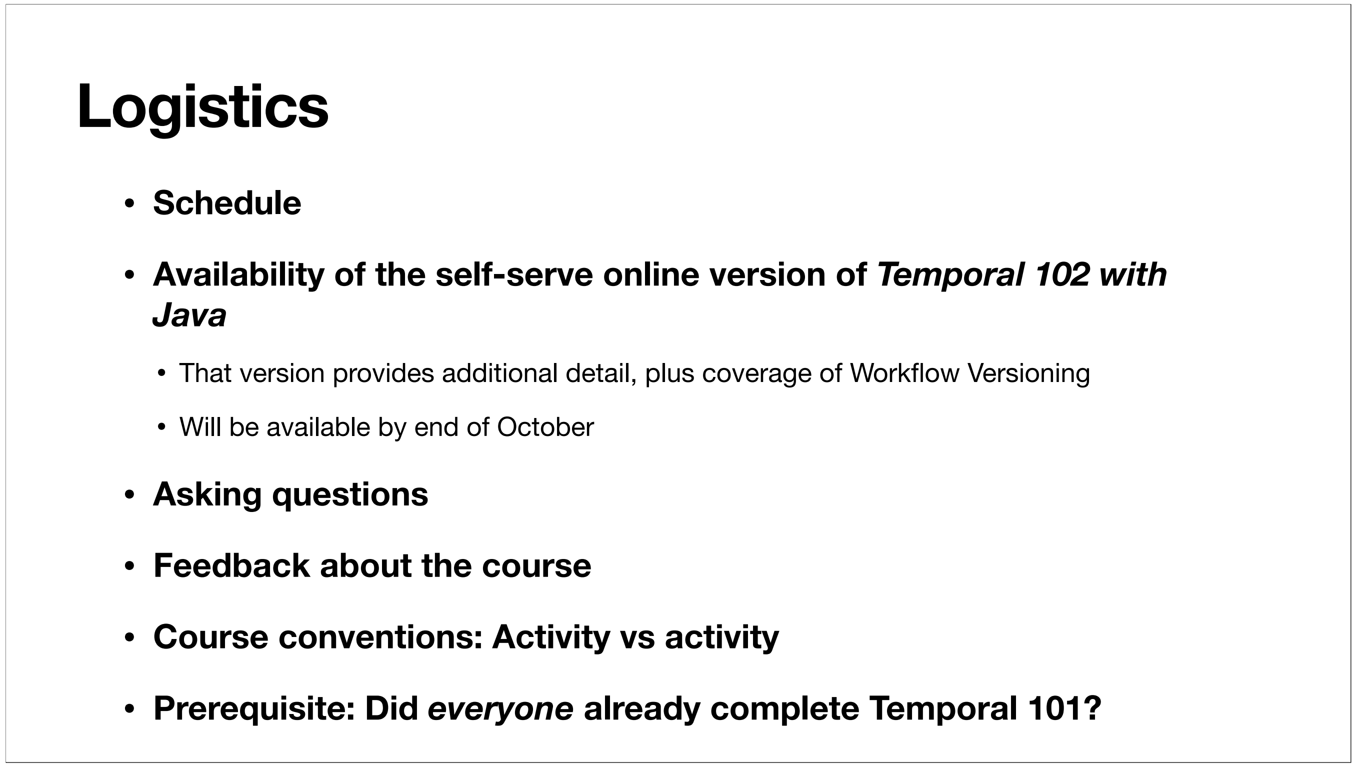
05. Understanding Event History

06. Debugging Workflow Execution

07. Deploying Your Application to Production

08. Understanding Workflow Determinism

09. Conclusion



Logistics

- **Schedule**
- **Availability of the self-serve online version of *Temporal 102 with Java***
  - That version provides additional detail, plus coverage of Workflow Versioning
  - Will be available by end of October
- **Asking questions**
- **Feedback about the course**
- **Course conventions: Activity vs activity**
- **Prerequisite: Did *everyone* already complete Temporal 101?**

First, let's go over some logistics.

This is a four hour workshop. That's a lot of time, and there's a lot to cover. But I've taught a lot of four hour workshops and classes, and I've sat through my fair share of them too. This workshop will consist of some lecture, but you'll also have four hands-on exercises to do. I'll also demo some things later on in the workshop. I've also scheduled in three ten-minute breaks, around the top of each hour, so you can get up, stretch, or do what you need to do. And of course, if you have to leave for a phone call, a bio break, or anything else, you should absolutely feel free to do so, but please make sure you do so courteously so you don't distract others.

Please put your phones on silent mode so as not to distract others. I've done the same. If you're one of the unlucky folks who has to be on call, I recommend placing the phone on on the table next to you so you can see any urgent notifications.

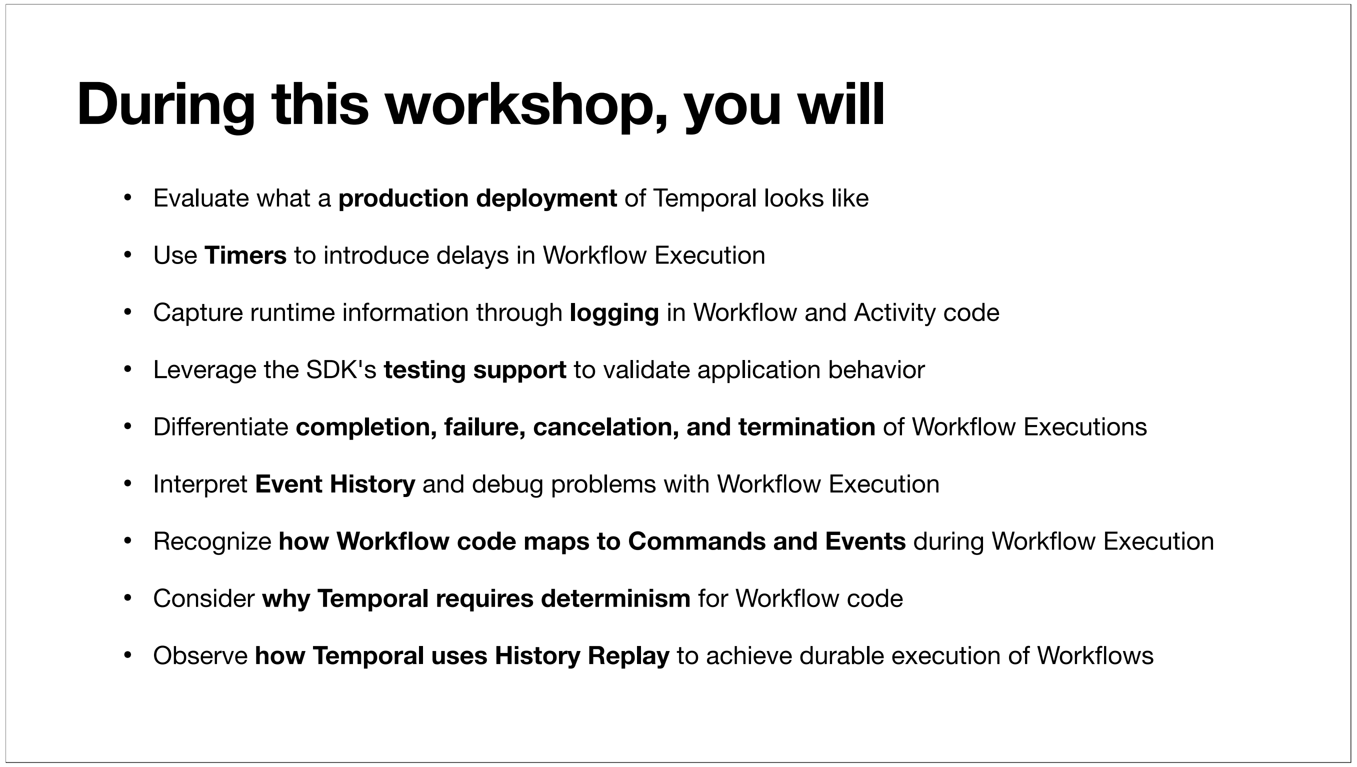
We have some amazing helpers here. They're here to answer your tough questions, and help you if you get stuck during lab time if I'm not able to do so myself.

Speaking of questions, because there are a lot of people here, I ask that you keep any questions you have scoped to the material we're covering. Definitely speak up if things aren't clear, let me know if I'm going too fast, and absolutely ask for clarification. However, if you have specific questions about your specific use-cases of Temporal, you should ask those kinds of questions during the breaks or after the course.

After the course ends, you'll receive a survey about the course. Your feedback will be incredibly helpful, as this course will eventually become a self-service online course.

In the course, you'll see some words are capitalized, like Activity or Workflow. When we are talking about the specific Temporal feature, you'll see it capitalized.

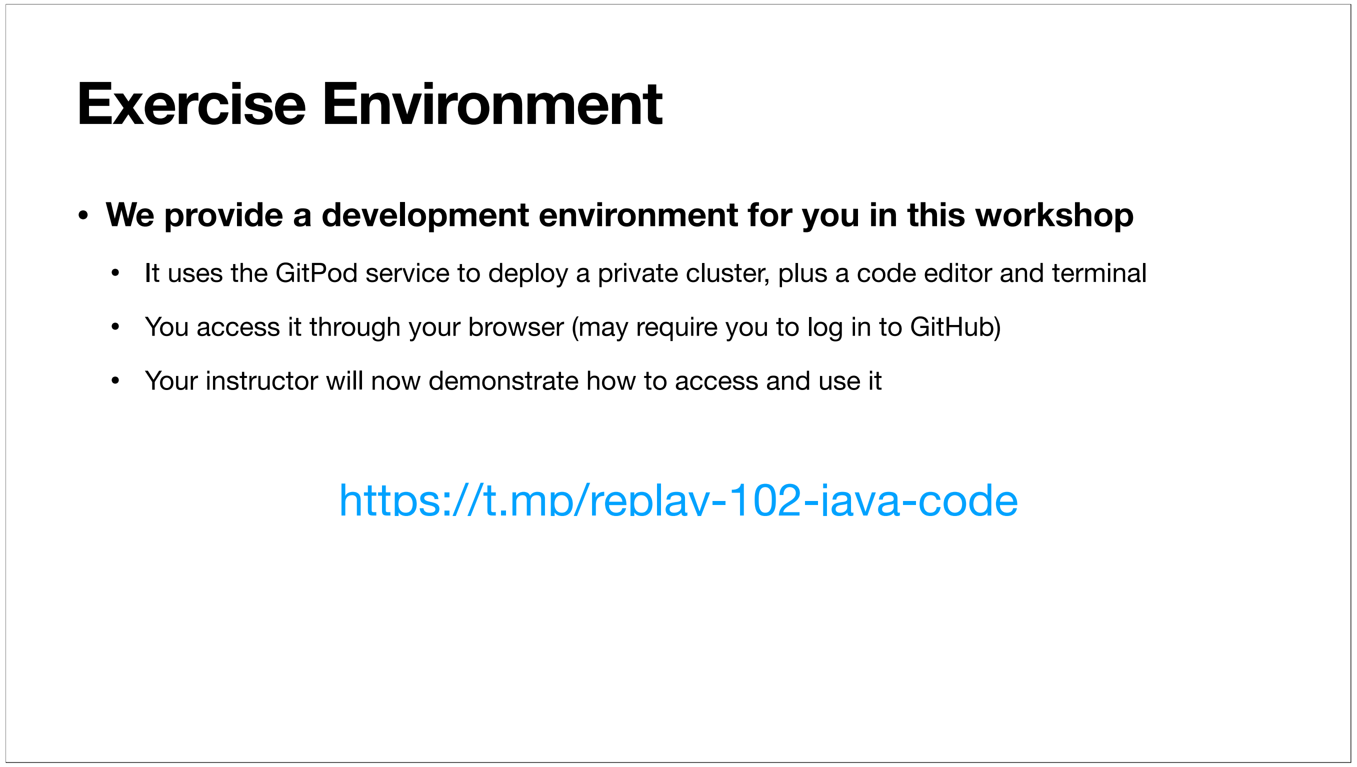
Finally, show of hands - who here took Temporal 101, either online or in the previous session? It’s not entirely required, but this course does build on the concepts from that section.



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
- Leverage the SDK's **testing support** to validate application behavior
- Differentiate **completion, failure, cancelation, and termination** of Workflow Executions
- Interpret **Event History** and debug problems with Workflow Execution
- Recognize **how Workflow code maps to Commands and Events** during Workflow Execution
- Consider **why Temporal requires determinism** for Workflow code
- Observe **how Temporal uses History Replay** to achieve durable execution of Workflows

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
Leverage the SDK's testing support to validate application behavior
Differentiate completion, failure, cancelation, and termination of Workflow Executions
Interpret Event History and debug problems with Workflow Execution
Recognize how Workflow code maps to Commands and Events during Workflow Execution
Consider why Temporal requires determinism for Workflow code
Observe how Temporal uses History Replay to achieve durable execution of Workflows



Exercise Environment

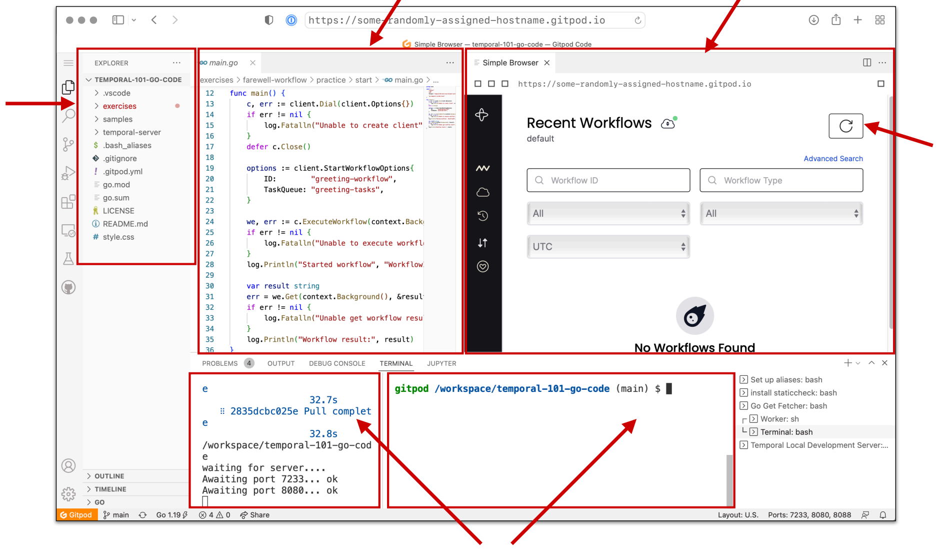
- **We provide a development environment for you in this workshop**
  - It uses the GitPod service to deploy a private cluster, plus a code editor and terminal
  - You access it through your browser (may require you to log in to GitHub)
  - Your instructor will now demonstrate how to access and use it

https://t.mp/replay-102-java-code

In this course you'll use an exercise environment powered by GitPod. You won't need to install anything on your personal machine to do the exercises. This environment will include a text editor, running terminals, and a Temporal development cluster.

Visit the URL on the screen on your local machine to start the exercise environment. While you do that, I'll go through a quick demonstration of the environment itself.

GitPod Overview



```go
func main() {
    c, err := client.Dial(client.Options{})
    if err != nil {
        log.Fatalln("Unable to create client"
    }
    defer c.Close()

    options := client.StartWorkflowOptions{
        ID:        "greeting-workflow",
        TaskQueue: "greeting-tasks",
    }

    we, err := c.ExecuteWorkflow(context.Back
    if err != nil {
        log.Fatalln("Unable to execute workfl
    }
    log.Println("Started workflow", "Workflow

    var result string
    err = we.Get(context.Background(), &resul
    if err != nil {
        log.Fatalln("Unable get workflow resu
    }
    log.Println("Workflow result:", result)
}
```

```
e
                    32.7s
   ⠿ 2835dcbc025e Pull complet
e
                    32.8s
/workspace/temporal-101-go-cod
e
waiting for server....
Awaiting port 7233... ok
Awaiting port 8080... ok
```

```
gitpod /workspace/temporal-101-go-code (main) $
```

Code editor

Embedded browser
(displays Temporal Web UI)

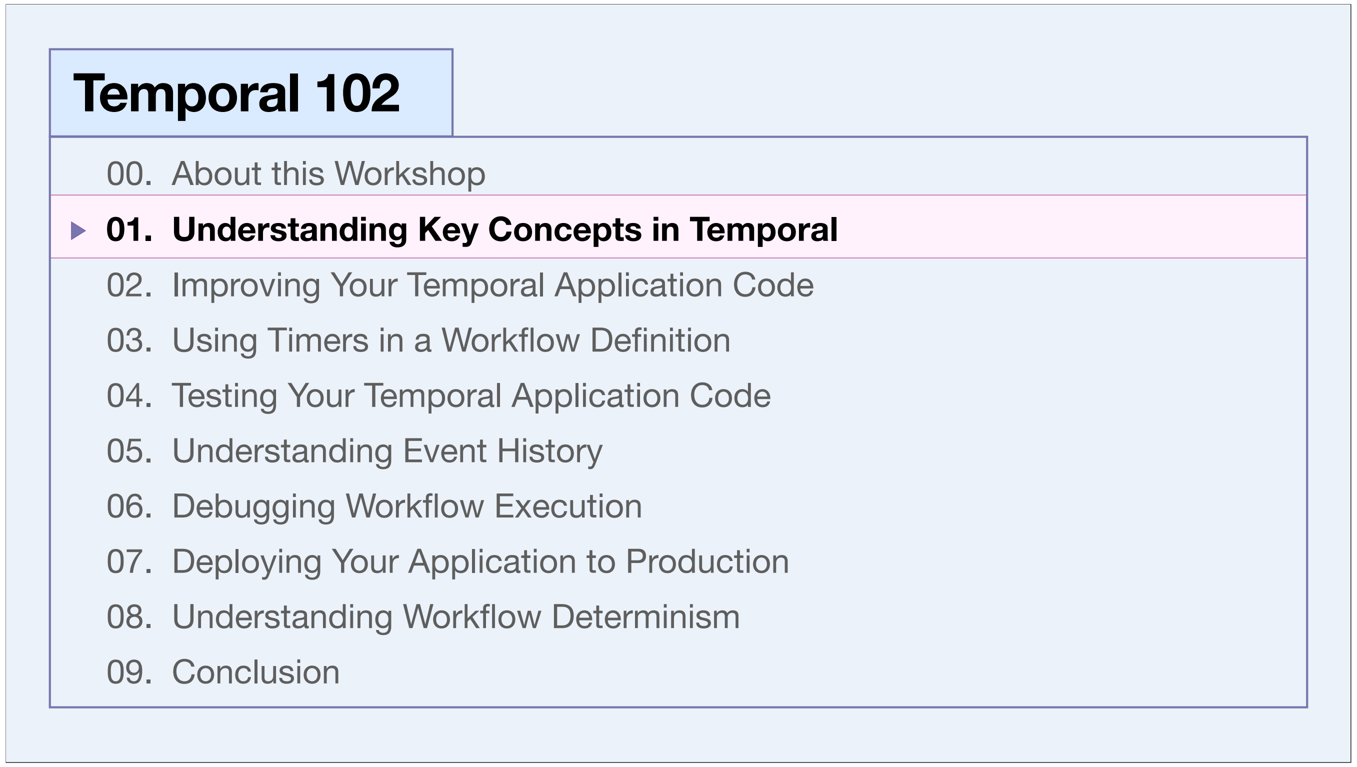
File browser
source code for exercises

Refresh button
(for Web UI)

Terminals

--

These are the key things to point out in the exercise environment. It's best to demo this live. I've included this slide for reference, but feel free to delete it.



Temporal 102

00. About this Workshop

▶ **01. Understanding Key Concepts in Temporal**

02. Improving Your Temporal Application Code

03. Using Timers in a Workflow Definition

04. Testing Your Temporal Application Code

05. Understanding Event History

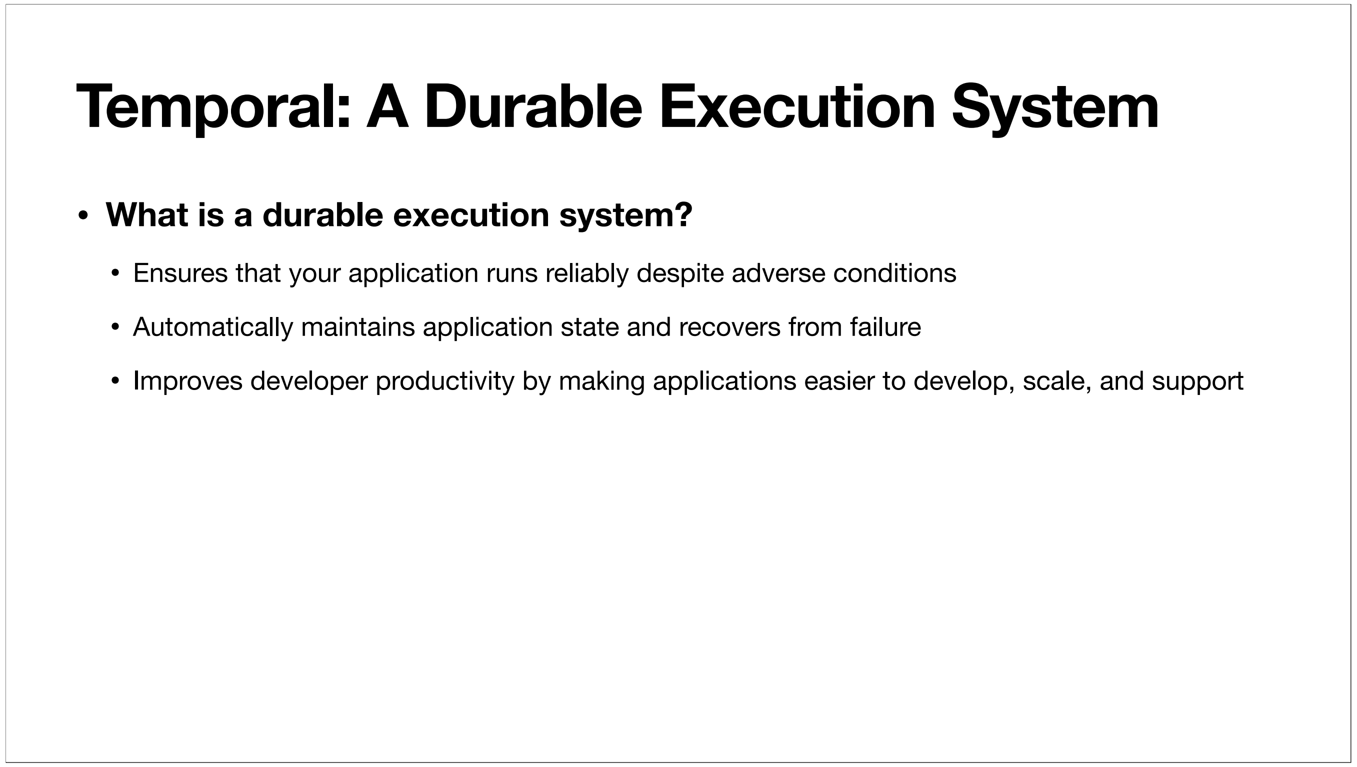
06. Debugging Workflow Execution

07. Deploying Your Application to Production

08. Understanding Workflow Determinism

09. Conclusion

We’ll start by reviewing some of the key concepts in Temporal to make sure we’re all on the same page.



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

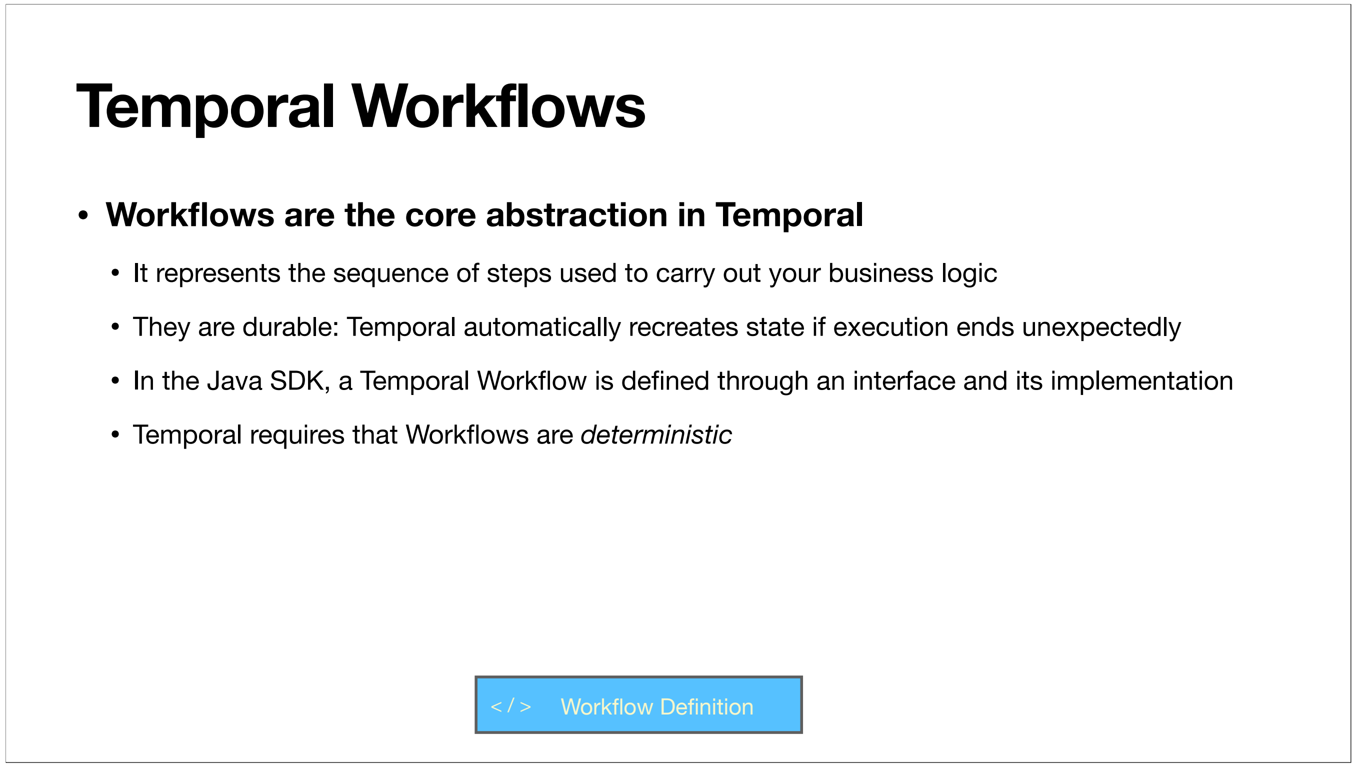
- Introduction to Temporal as a Durable Execution System
  - Ensures reliable and correct code execution
  - Maintains state for automatic recovery from failures
  - Handles small problems (e.g., network timeout) and major issues (e.g., server kernel panic)
  - Improves developer productivity
  - Provides higher-level abstractions for application development
  - Offers built-in scalability
  - Allows developers to focus on business logic
  - Provides productivity-enhancing tools, like the Web UI
  - Web UI enables viewing execution history, input parameters, and return values
  - Web UI used for debugging Workflow Execution and verifying fixes

---

Let's begin the course by introducing a new term for describing Temporal, a *durable execution system*, and then covering some key concepts that provide the foundation for what you'll learn in this course.

A durable execution system ensures that the code in your application runs reliably and correctly, even in the face of adversity. It maintains state, allowing your code to automatically recover from failure, regardless of whether that failure was caused by a small problem, such as a network timeout, or a big one, such as a kernel panic on a production application
server.

It also improves your productivity. As developers, we recognize that it's critical to handle problems such as failures and timeouts that can affect application reliability and spend significant time writing code to do so. By providing higher-level abstractions for application development and built-in scalability, Temporal enables you to instead focus on your application's business logic. Furthermore, Temporal provides tools that enhance your productivity, such as the Web UI you can use to view the execution history of your applications, both past and present, including their input parameters and return values. During this course, you'll use the Web UI to debug Workflow Execution and then ensure that your fix solved the problem.



Temporal Workflows

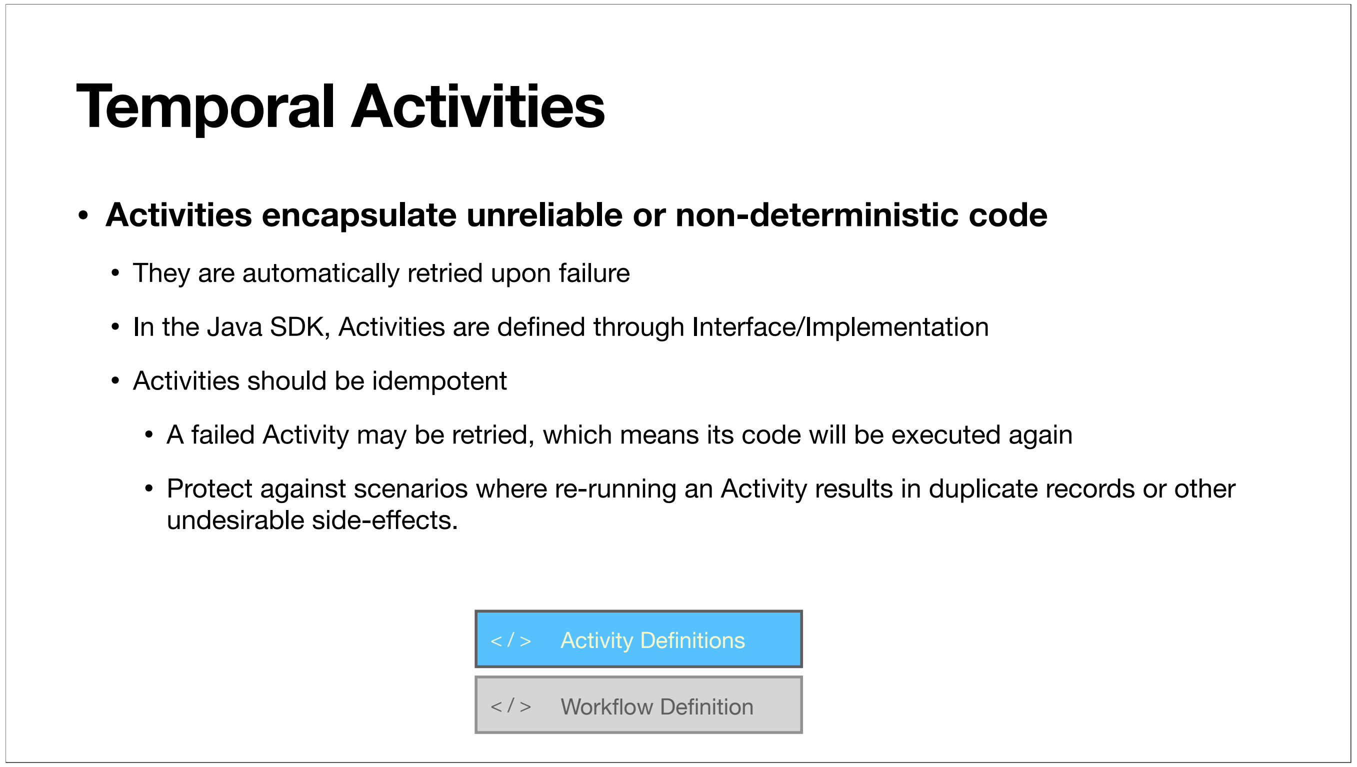
- **Workflows are the core abstraction in Temporal**
  - It represents the sequence of steps used to carry out your business logic
  - They are durable: Temporal automatically recreates state if execution ends unexpectedly
  - In the Java SDK, a Temporal Workflow is defined through an interface and its implementation
  - Temporal requires that Workflows are *deterministic*

</ >   Workflow Definition

- The main business logic of a Temporal application is called a Workflow.
- Workflows are written in general-purpose programming languages like Go, Java, TypeScript, or Python.
- Temporal provides language-specific SDKs (software development kits) with APIs and libraries to support your application.
- This course uses Temporal's TypeScript SDK for defining Workflows as functions.
- Workflows in Temporal must be deterministic, meaning each execution with the same input produces the same output.
- In this course, you'll learn a precise definition of determinism in Temporal.
- You'll also understand why Temporal requires determinism, consequences of violating this rule, and how to identify and avoid non-determinism in your Workflows.

The sequence of steps that makes up the main business logic of your Temporal application is called a Workflow. Like most other applications you develop, it is it is written in a general-purpose programming language such as go Java, TypeScript, or Python. Temporal provides language-specific software development kits, or SDKs, that provide APIs and libraries to support your application. The code in this course uses Temporal's TypeScript SDK, so Workflows are defined as functions in the TypeScript programming language.

Temporal requires that Workflows are deterministic. Temporal 101 explained this by stating that each execution of given Workflow must produce the same output given the same input. In this course, you'll learn a more precise definition for determinism in Temporal. You'll also learn why Temporal requires that Workflows are deterministic, what can happen if this rule is violated, and how to identify and avoid non-determinism in your Workflows.



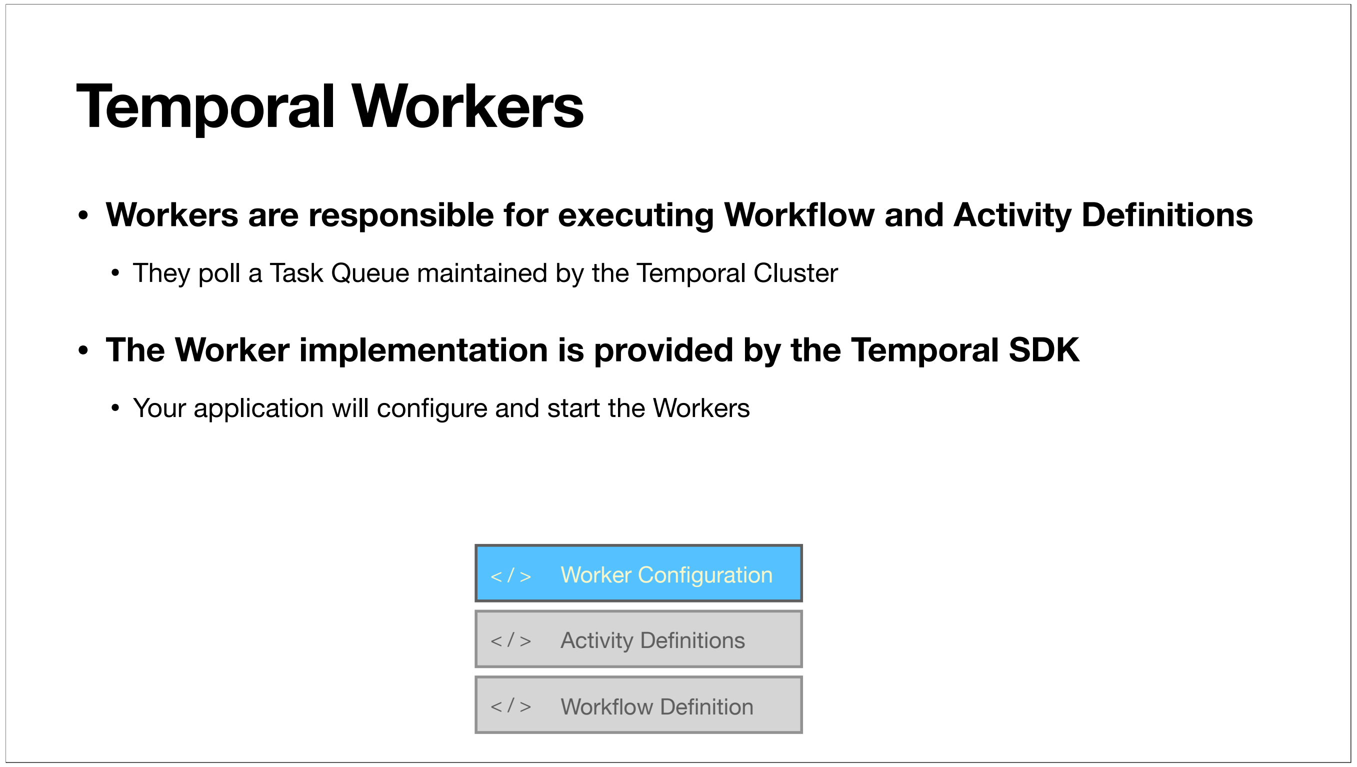
Temporal Activities

- **Activities encapsulate unreliable or non-deterministic code**
  - They are automatically retried upon failure
  - In the Java SDK, Activities are defined through Interface/Implementation
  - Activities should be idempotent
    - A failed Activity may be retried, which means its code will be executed again
    - Protect against scenarios where re-running an Activity results in duplicate records or other undesirable side-effects.

- Activities encapsulate non-deterministic or failure-prone parts of business logic in Temporal applications.
- Activities are invoked as part of Workflow Execution and are retried in case of failure.
- Temporal automatically handles transient or intermittent failures in the application.
- Activities, like Workflows, are defined as interfaces and their implmemenation when implemented in Java.

- Activities provide a means of encapsulating parts of the business logic that are non-deterministic or prone to failure. These are called as part of Workflow Execution and are retried upon failure. This means that transient or intermittent failures are handled automatically in the Temporal application. As with Workflows, Activities are also defined as functions when implemented in TypeScript .

Activity executes lines 1 - 100, crashes at line 101, retries Activity, lines 1 - 100 are executed again during the retry.



Temporal Workers

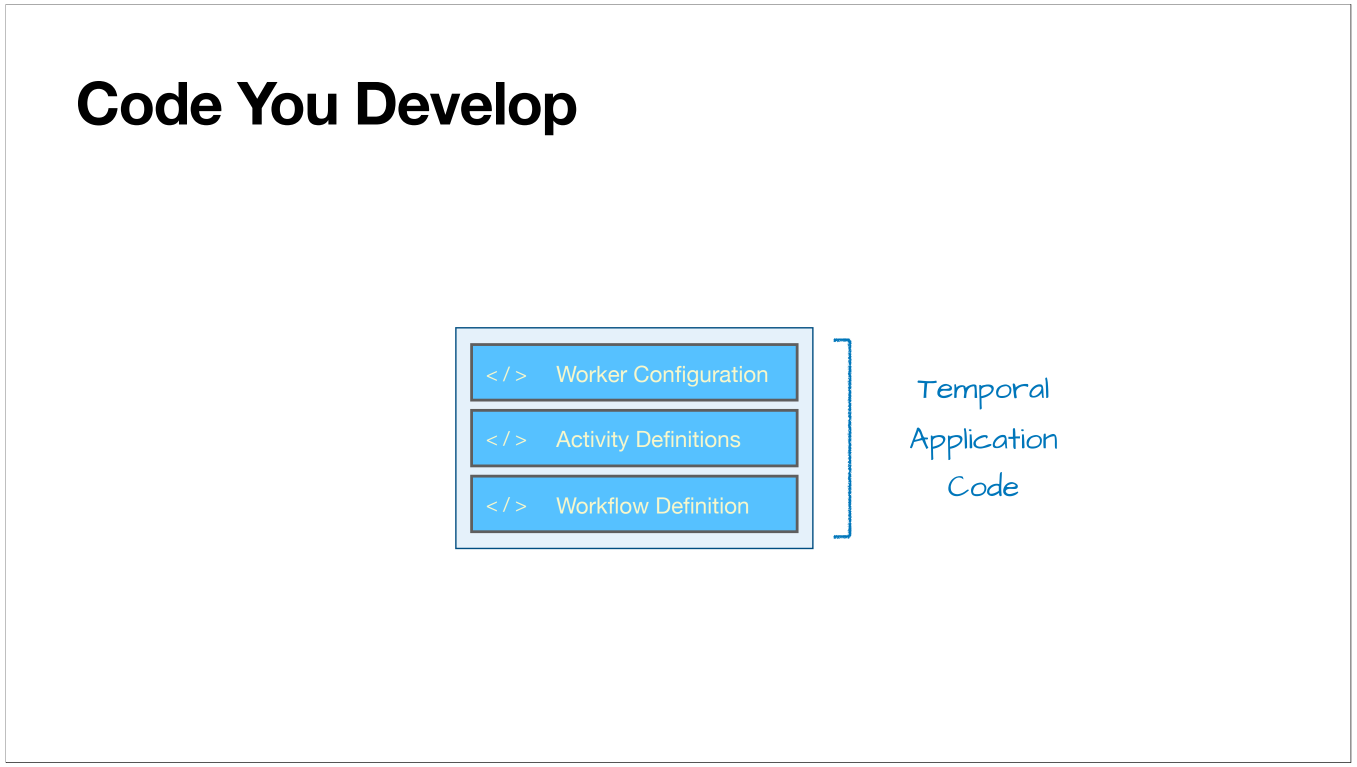
- **Workers are responsible for executing Workflow and Activity Definitions**
  - They poll a Task Queue maintained by the Temporal Cluster
- **The Worker implementation is provided by the Temporal SDK**
  - Your application will configure and start the Workers

&lt; / &gt; Worker Configuration
< / > Activity Definitions
< / > Workflow Definition

- Remember, the Temporal Cluster does not execute your workflows
- Execution is orchestrated through the workers
  - They poll a Task Queue maintained by the Temporal Cluster
- Worker implementation provided by the sdk

Although Temporal Cluster is essential for the durable execution of your Workflows, it does not execute your code. Instead it orchestrates the execution of your code by maintaining a Task Queue. The Worker, which polls this Task Queue, is responsible for executing your Workflow and Activity code.

The Worker implementation is provided by the Temporal SDK, so you don't need to write it. You will need to configure it, though, by specifying the Task Queue name and registering your Workflow and Activity functions. You'll also write code to start the Worker.

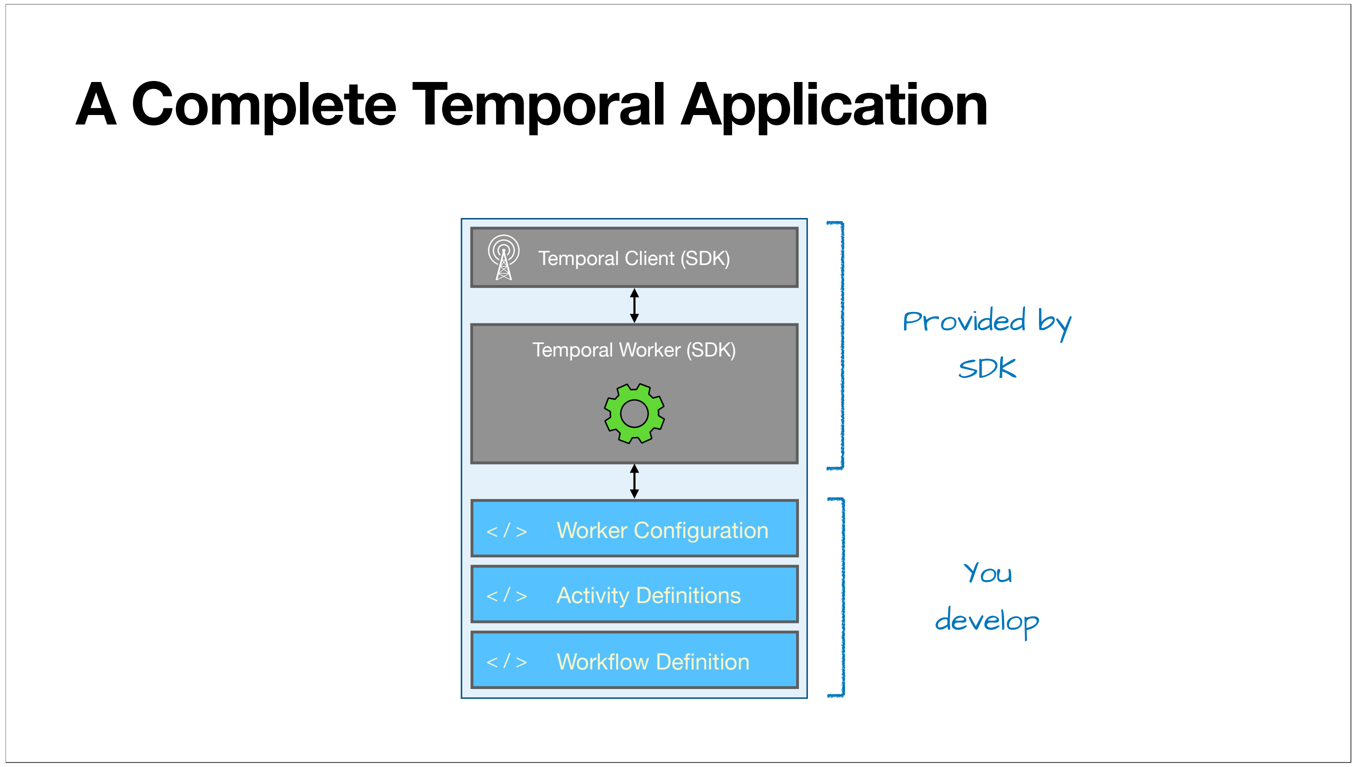


Code You Develop

</ > Worker Configuration
</ > Activity Definitions
</ > Workflow Definition
Temporal Application Code

- Workflow + Activity + Worker = Code you write
- We call this the Temporal application code
- This is not all there is to a complete application

In summary, a Temporal application developer is responsible for writing Workflow Definitions, Activity Definitions, and the code to configure and start the Workers that coordinate with a Temporal Cluster to carry execution forward. Since these represent the code that you, the developer, are responsible for writing, we'll collectively refer to them as the Temporal Application Code. However, a complete application is more than just that code.



A Complete Temporal Application

Temporal Client (SDK)

Temporal Worker (SDK)

</ > Worker Configuration

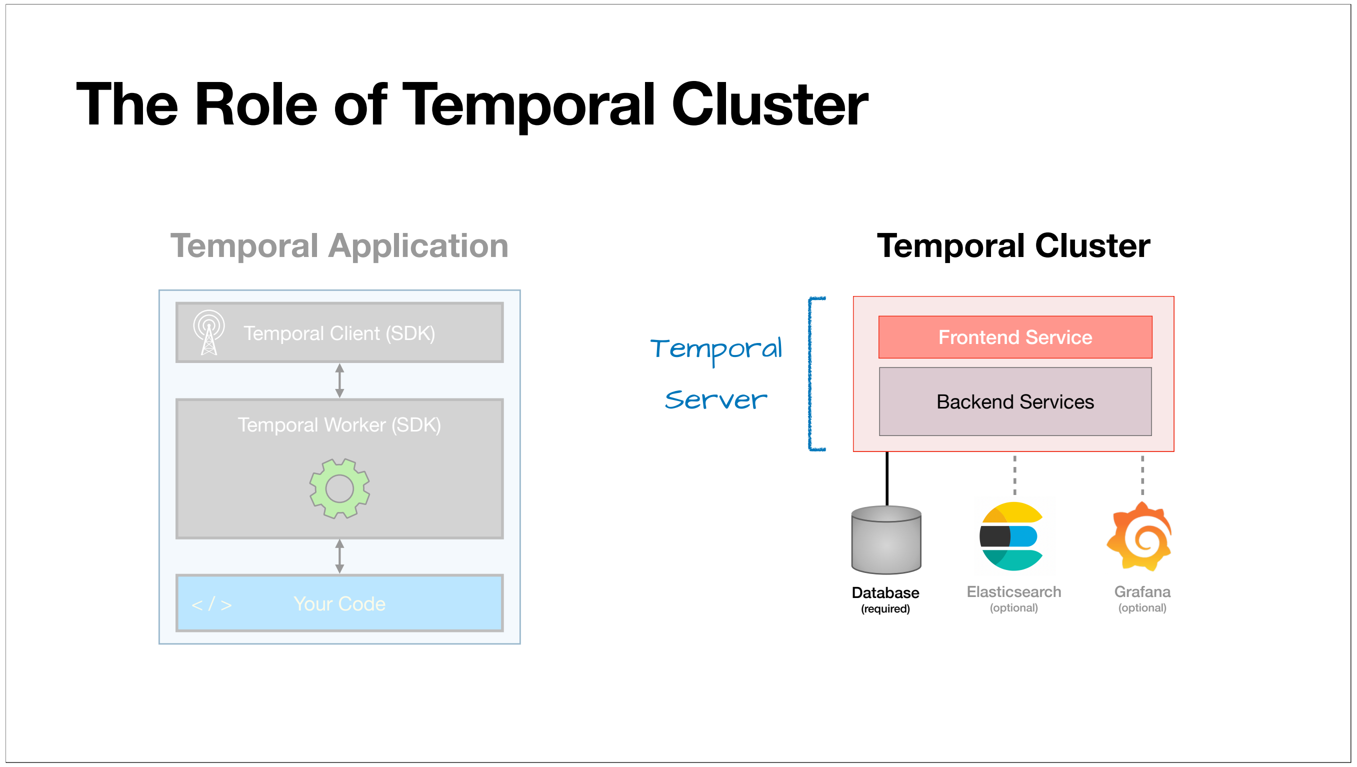
</ > Activity Definitions

</ > Workflow Definition

Provided by SDK

You develop

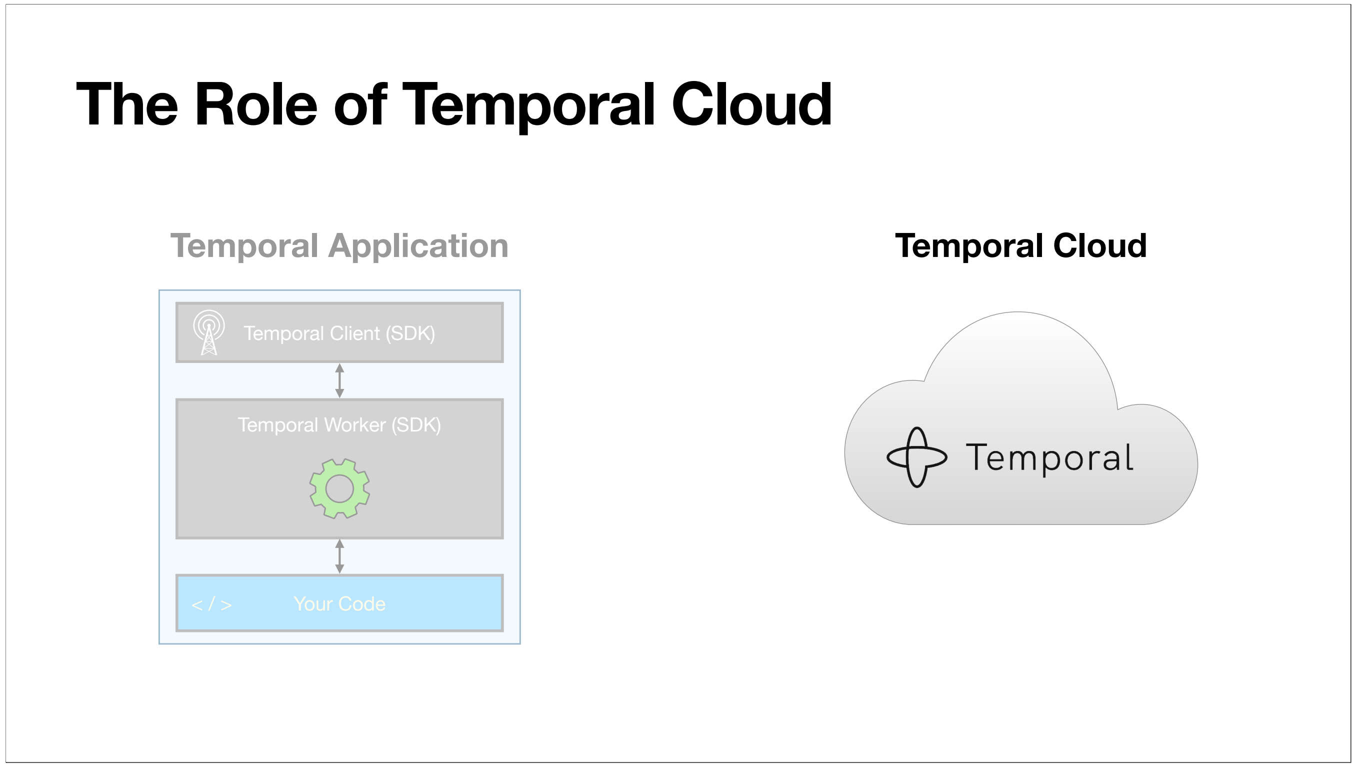
I think it's helpful to think of a Temporal
application as having two parts: one that you develop and another part provided by the SDK.



The Role of Temporal Cluster

- Durability, scalability, and reliability of a Temporal application rely on the support of the Temporal cluster.
- The Temporal cluster is a deployment of the Temporal Server, comprising a frontend and multiple backend services.
- The cluster also relies on a database for persistence.
- Optional components in a Temporal cluster can include Elasticsearch for enhanced search performance and Grafana for operational dashboards to visualize cluster and application health.

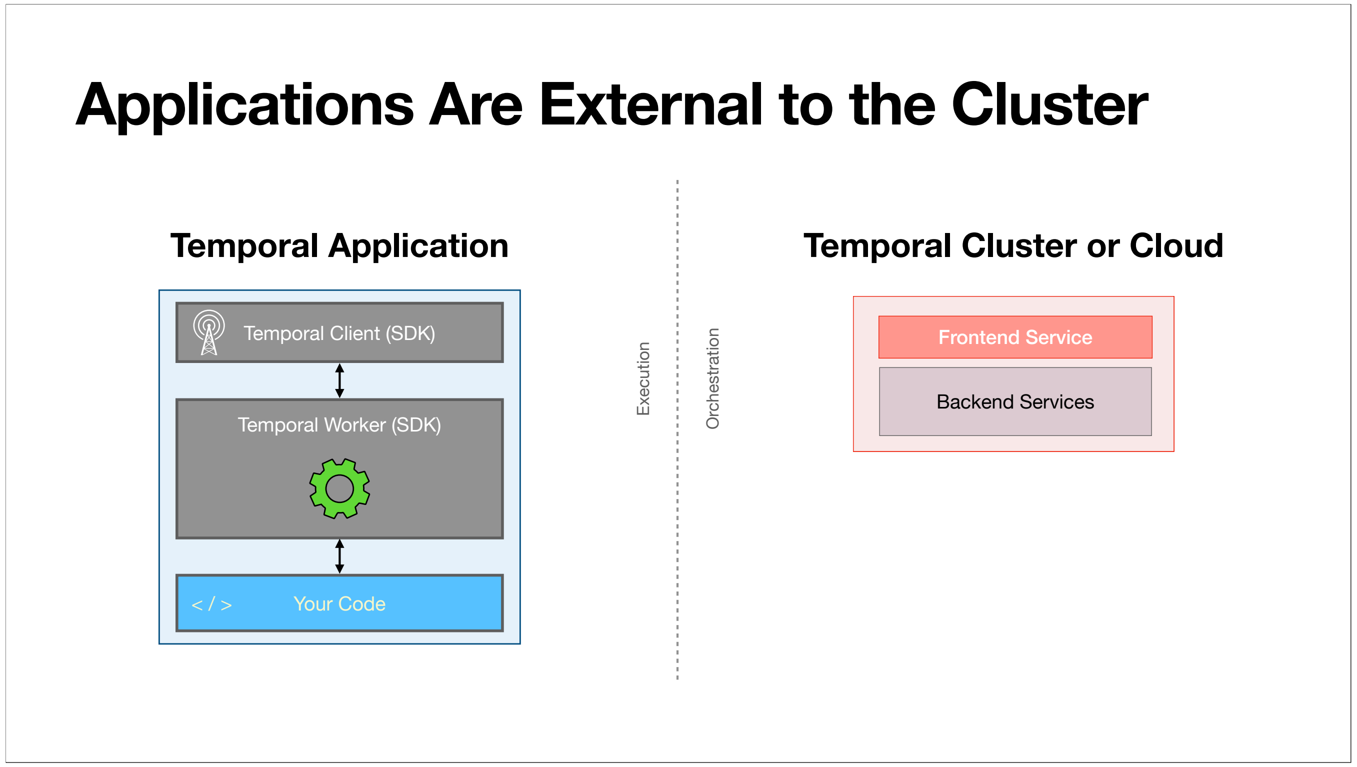
A Temporal application gains its durability, scalability, and reliability from the support provided by the Temporal cluster. This is a deployment of the Temporal Server, which consists of a frontend and multiple backend services, plus the database it relies on for persistence. A Temporal cluster may also include some optional components, such as Elasticsearch for improved search performance, or Grafana for creating operational dashboards for visualizing the health of your cluster and applications.



The Role of Temporal Cloud

Alternatively, you might use the Temporal Cloud service, in which case you won't need to deploy run and manage your own Temporal cluster. A self-hosted cluster and the Temporal Cloud service both perform the same roles. You can think of Temporal Cloud, conceptually speaking, as a very large high-performance, scalable, and secure Temporal Cluster that's managed and supported by an expert operations team.

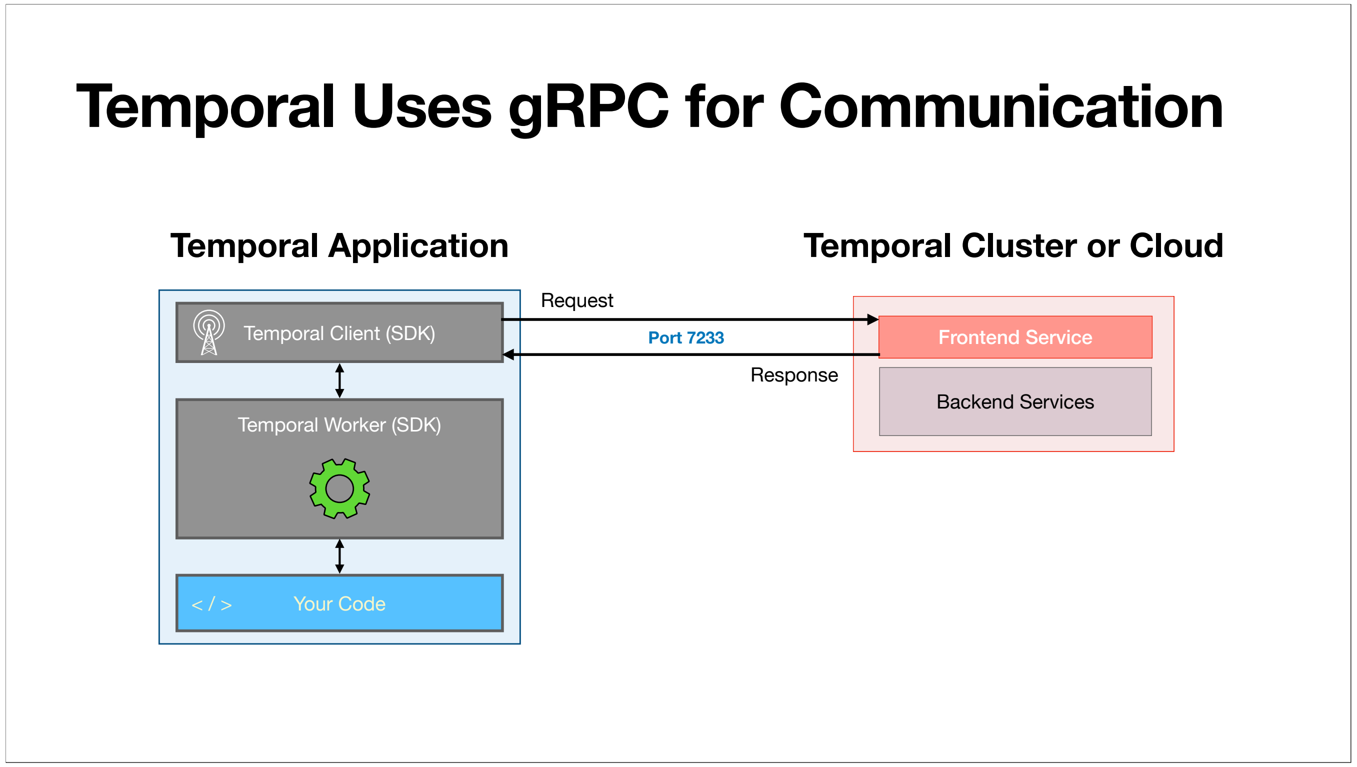
Temporal Cloud eliminates the need for your operations staff to plan deploy, secure, and manage a self-hosted cluster, so they're different from an operations perspective, but equivalent from the developer's perspective. Since this is a course for developers, it will generally refer to Temporal Cluster for the sake of brevity. It will mention Temporal Cloud specifically, when notable, but otherwise you can assume that a reference to Temporal cluster also applies to Temporal Cloud.



Applications Are External to the Cluster

Regardless of whether you're running a self-hosted Temporal Cluster or using Temporal Cloud, Workflow Execution works the same way. In fact, moving your application from a self-hosted cluster to Temporal Cloud requires minimal code change; typically just modifying a few connection parameters of the Temporal Client. You'll learn how to make those changes during this course.

Be sure to notice the separation here: The application and its execution are external to the cluster. In a production deployment, they typically run on separate machines or containers. In fact, it's possible to run the application and Temporal Cluster in different data centers.



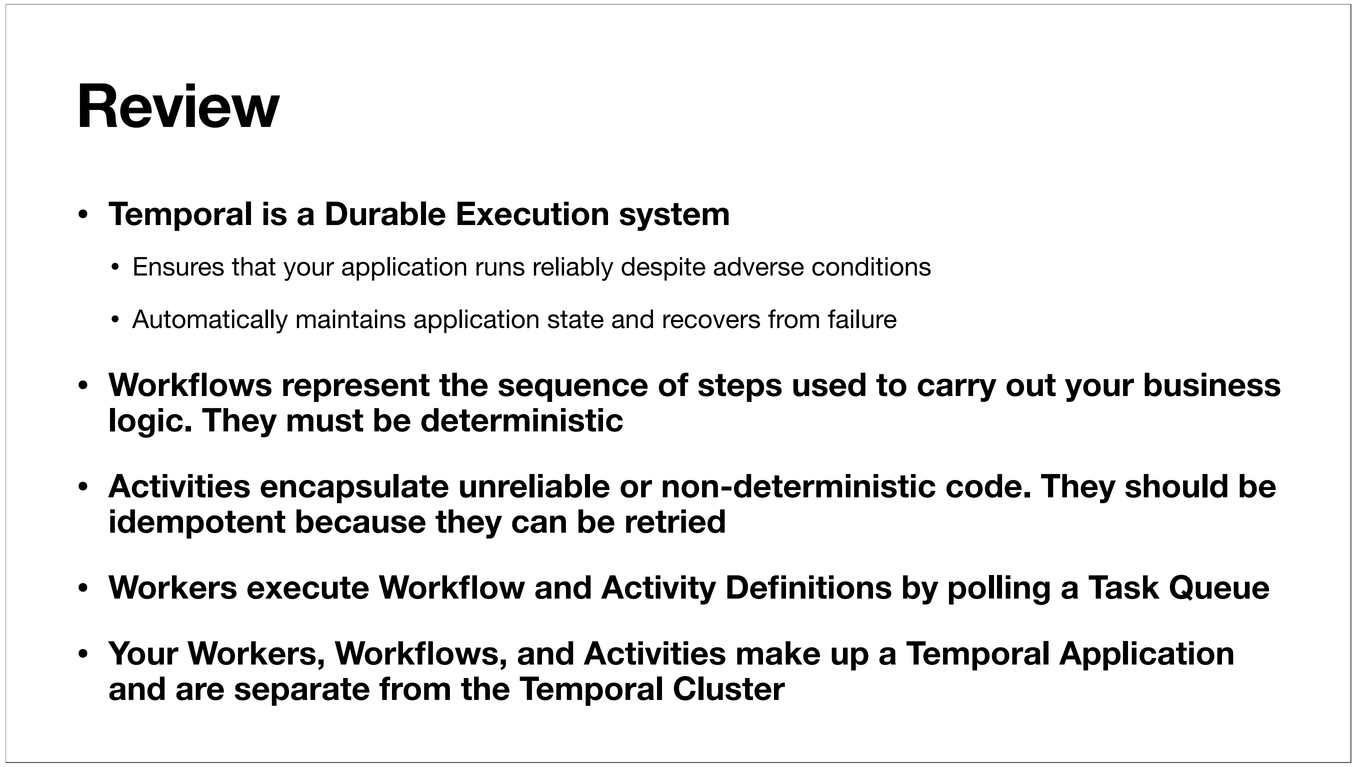
Temporal Uses gRPC for Communication

Since the interaction between the application and cluster is key to how Temporal works, it's helpful to understand how they communicate.

Requests from a Temporal Client are always directed to the Frontend Service, which serves as a gateway. This communication takes place over TCP port 7233 and uses gRPC, The messages themselves are encoded using Protocol Buffers.

When you invoke functions provided by the Temporal SDK in your code, the SDK generates a message corresponding to a Temporal Server API, encodes it using Protocol Buffers, and sends a request to the Frontend Service using gRPC. The Frontend Service handles this request, routing it to the appropriate backend services as necessary and sending a response, which also uses Protocol Buffers and gRPC, back to the client.

All of this communication can be secured with TLS, which encrypts the data as it is transmitted across the network and can also verify the identity of the client and server by validating their certificates.



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. They should be idempotent because they can be retried**
- **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 Cluster**

Temporal is a Durable Execution system

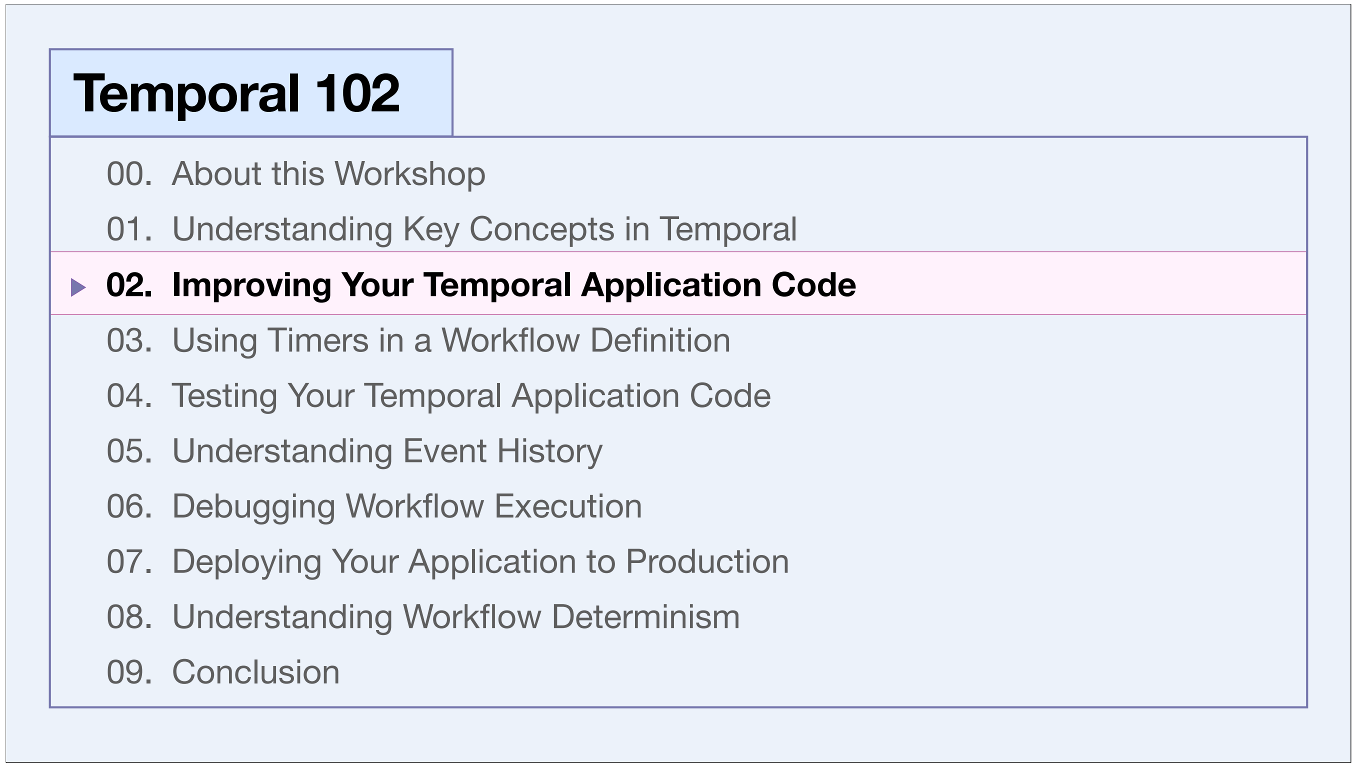
This ensures that your application runs reliably despite adverse conditions

And that it 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. They should be idempotent because they can be retried.

Workers execute Workflow and Activity Definitions by polling a Task Queue

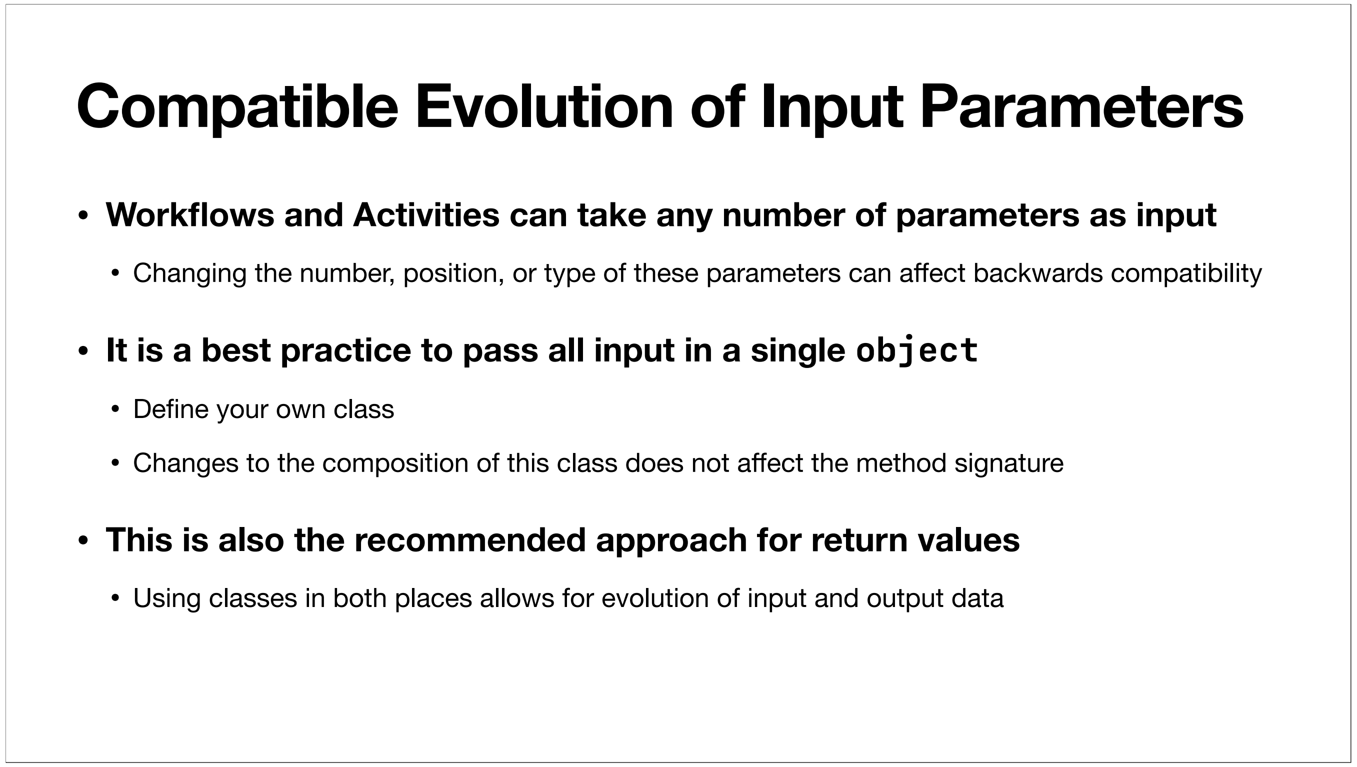
Your Workers, Workflows, and Activities make up a Temporal Application and are separate from the Temporal Cluster.



Temporal 102

00. About this Workshop
01. Understanding Key Concepts in Temporal
▶ **02. Improving Your Temporal Application Code**
03. Using Timers in a Workflow Definition
04. Testing Your Temporal Application Code
05. Understanding Event History
06. Debugging Workflow Execution
07. Deploying Your Application to Production
08. Understanding Workflow Determinism
09. Conclusion

Now let’s look at some things you can do to improve the code you write when you build Temporal applications. In Temporal 101, you built basic Workflows and Activities, and you focused on understanding important foundational concepts. Now it’s time to start thinking more about how to write Workflows and Activities that are easier to maintain.



Compatible Evolution of Input Parameters

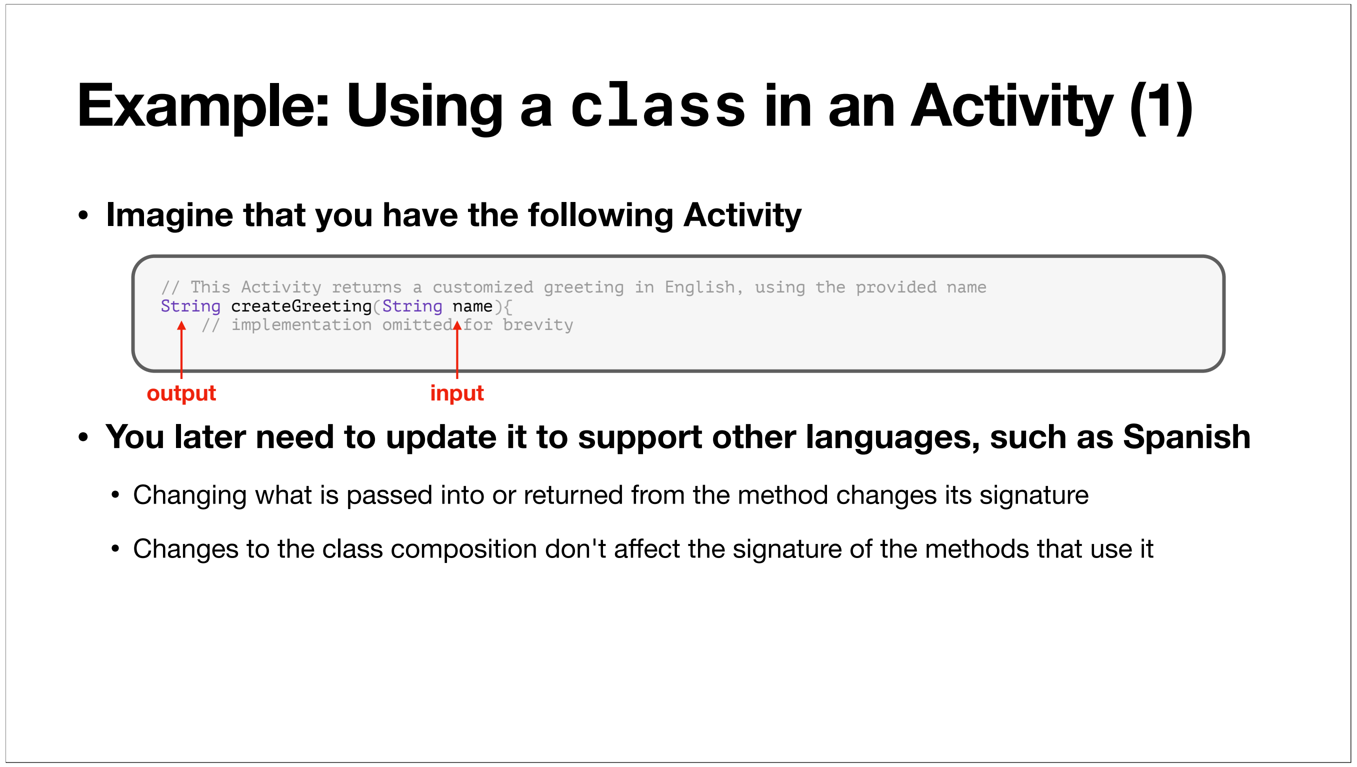
- **Workflows and Activities can take any number of parameters as input**
  - Changing the number, position, or type of these parameters can affect backwards compatibility
- **It is a best practice to pass all input in a single `object`**
  - Define your own class
  - Changes to the composition of this class does not affect the method signature
- **This is also the recommended approach for return values**
  - Using classes in both places allows for evolution of input and output data

We'll start by looking at the inputs and outputs of your Workflows and Activities.

Workflows and Activities can take any number of parameters as input. But changing the number, position, or type of these parameters can affect backwards-compatibility.

The best course of action you can take is to provide all of the inputs to a Workflow or Activity as a single object.

You should also do this for results of workflows and activities.



Example: Using a `class` in an Activity (1)

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

```
// This Activity returns a customized greeting in English, using the provided name
String createGreeting(String name){
    // implementation omitted for brevity
```

output

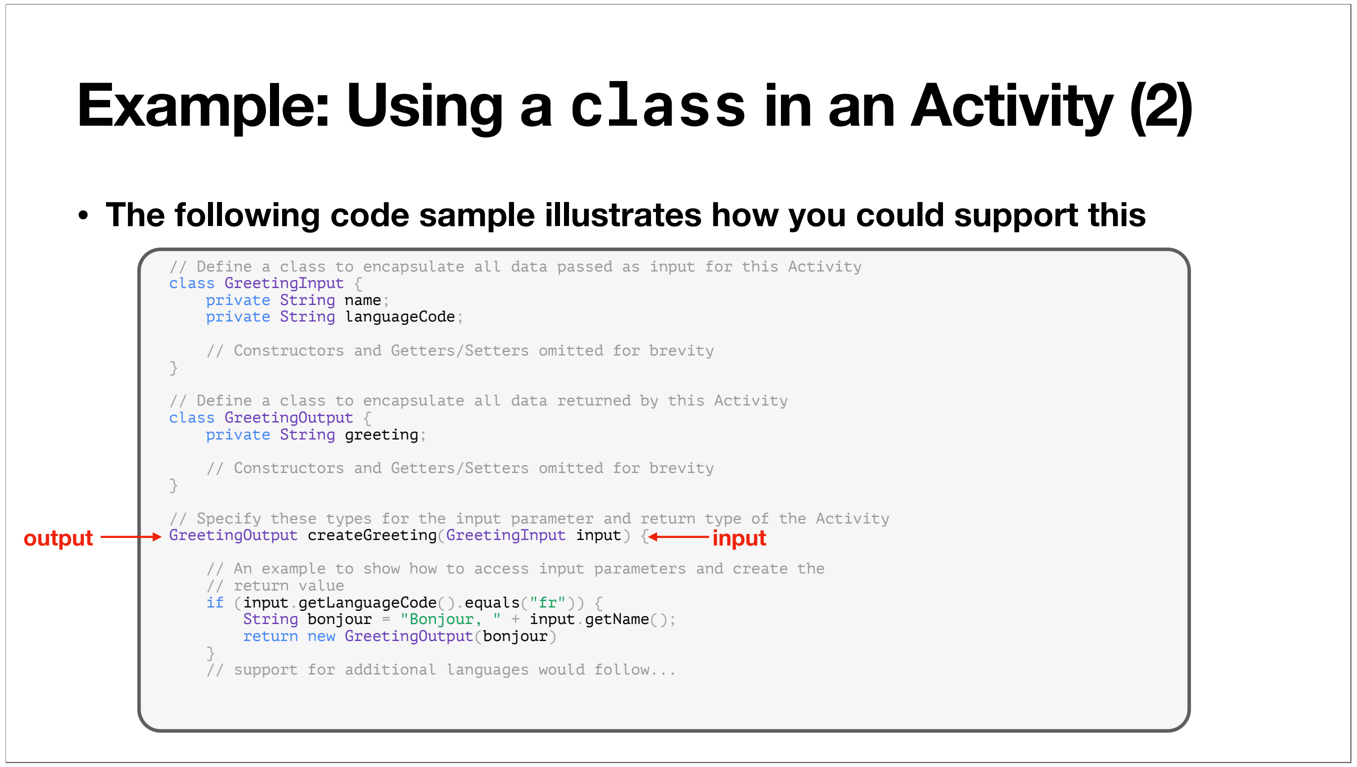
input

- **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
  - Changes to the class composition don't affect the signature of the methods that use it

To understand why this is considered a best practice, consider the "Hello World" scenario you worked with in Temporal 101, which had an Activity that called a microservice to retrieve a greeting in Spanish. This method took a string (containing a person's name) as input and returned a string (containing the customized greeting in Spanish) as output:

Although this certainly works, it is not the best approach for something you plan to deploy to production and maintain over time. Later on you'll need to update this so it supports other languages.

Changing what gets passed into or returned from a method can change the function's signature.



Example: Using a `class` in an Activity (2)

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

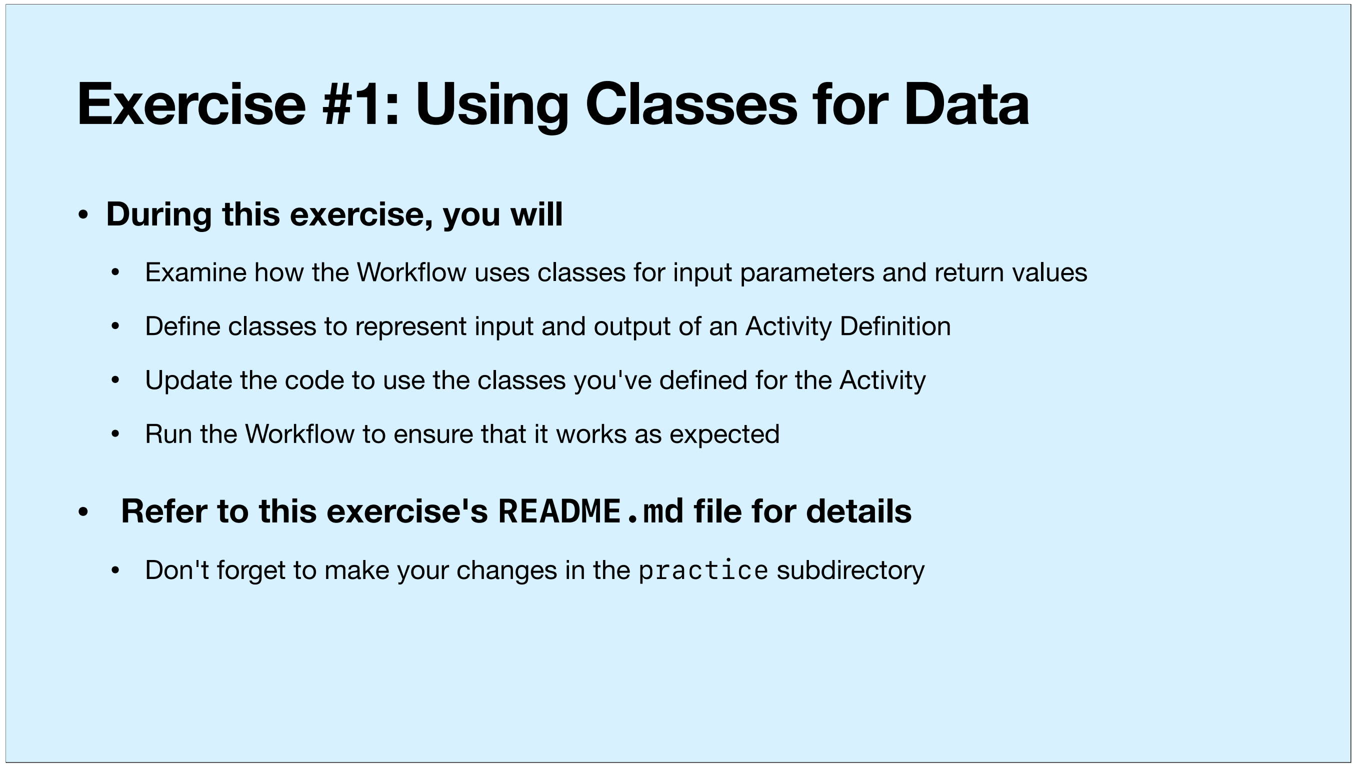
```
// Define a class to encapsulate all data passed as input for this Activity
class GreetingInput {
    private String name;
    private String languageCode;

    // Constructors and Getters/Setters omitted for brevity
}

// Define a class to encapsulate all data returned by this Activity
class GreetingOutput {
    private String greeting;

    // Constructors and Getters/Setters omitted for brevity
}

// Specify these types for the input parameter and return type of the Activity
GreetingOutput createGreeting(GreetingInput input) {

    // An example to show how to access input parameters and create the
    // return value
    if (input.getLanguageCode().equals("fr")) {
        String bonjour = "Bonjour, " + input.getName();
        return new GreetingOutput(bonjour)
    }
    // support for additional languages would follow...
```

output

input

This code example illustrates a better design for this Activity. It begins by defining a class that represents input to this Activity, which includes the original name but also the new language code field. It then defines another class, which represents the output returned by the Activity method. It just contains the translated greeting for now, but you could update this as requirements change in the future. Finally, the Activity method itself has been updated to use these new objects instead of the strings it had previously used.

While the initial move from strings to objects is not a backwards compatible change, making that change as early as possible will ensure that the code is better able to handle future evolution of the input or output data.

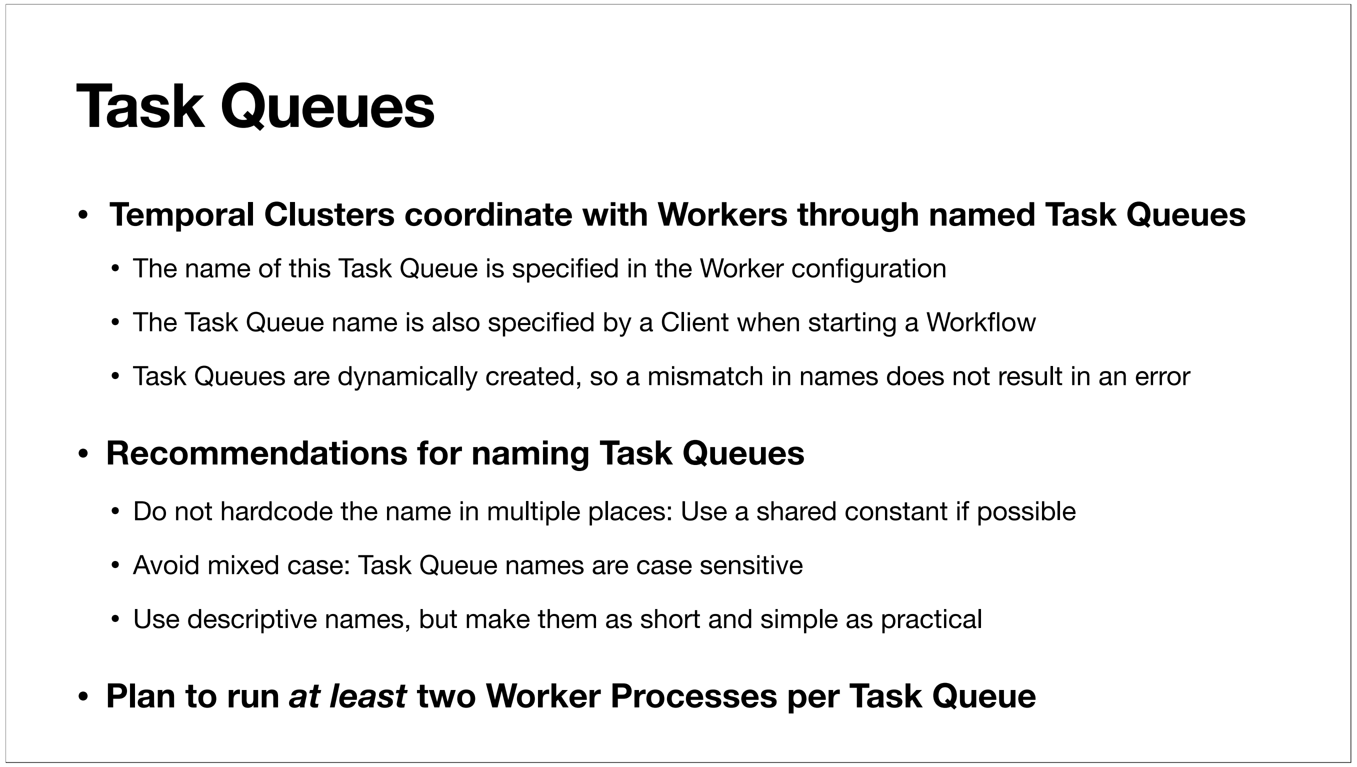


Exercise #1: Using Classes for Data

- **During this exercise, you will**
  - Examine how the Workflow uses classes for input parameters and return values
  - Define classes to represent input and output of an Activity Definition
  - Update the code to use the classes you've defined for the Activity
  - Run the Workflow to ensure that it works as expected
- **Refer to this exercise's `README.md` file for details**
  - Don't forget to make your changes in the `practice` subdirectory

Now it's your turn. In the exercise environment, you'll find instructions for a hands-on lab where you'll change the code to work with objects instead of basic strings.

Exercise #1: Using Classes for Data SOLUTION



Task Queues

- **Temporal Clusters 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**

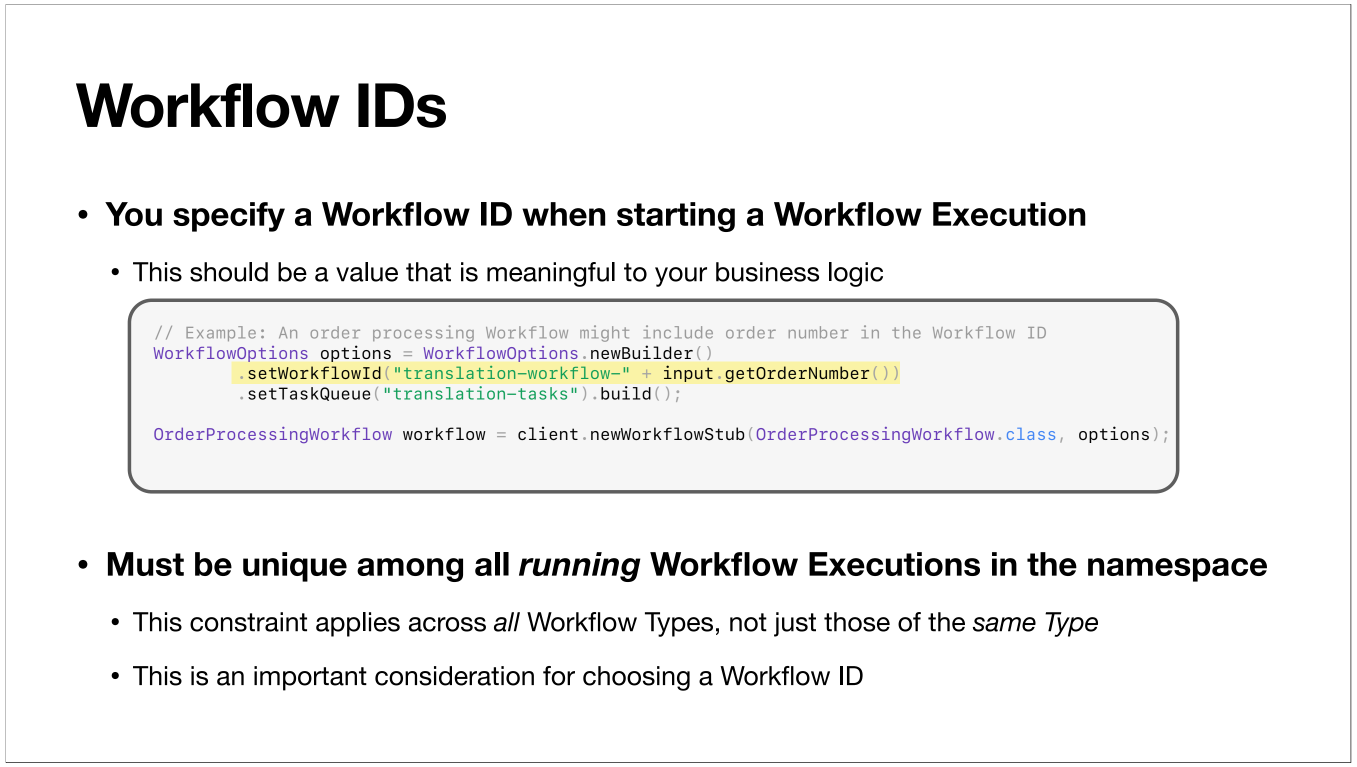
Another area to consider are task queues and how you specify them. First, let’s go over task queues.

Temporal Clusters 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

So you shouldn’t hard code the name in multiple places.
And the names are case sensitive. It’s best to use a constant.
Use descriptive names but make them as short as practical.

Also, plan to have at least two Worker processes per task queue.



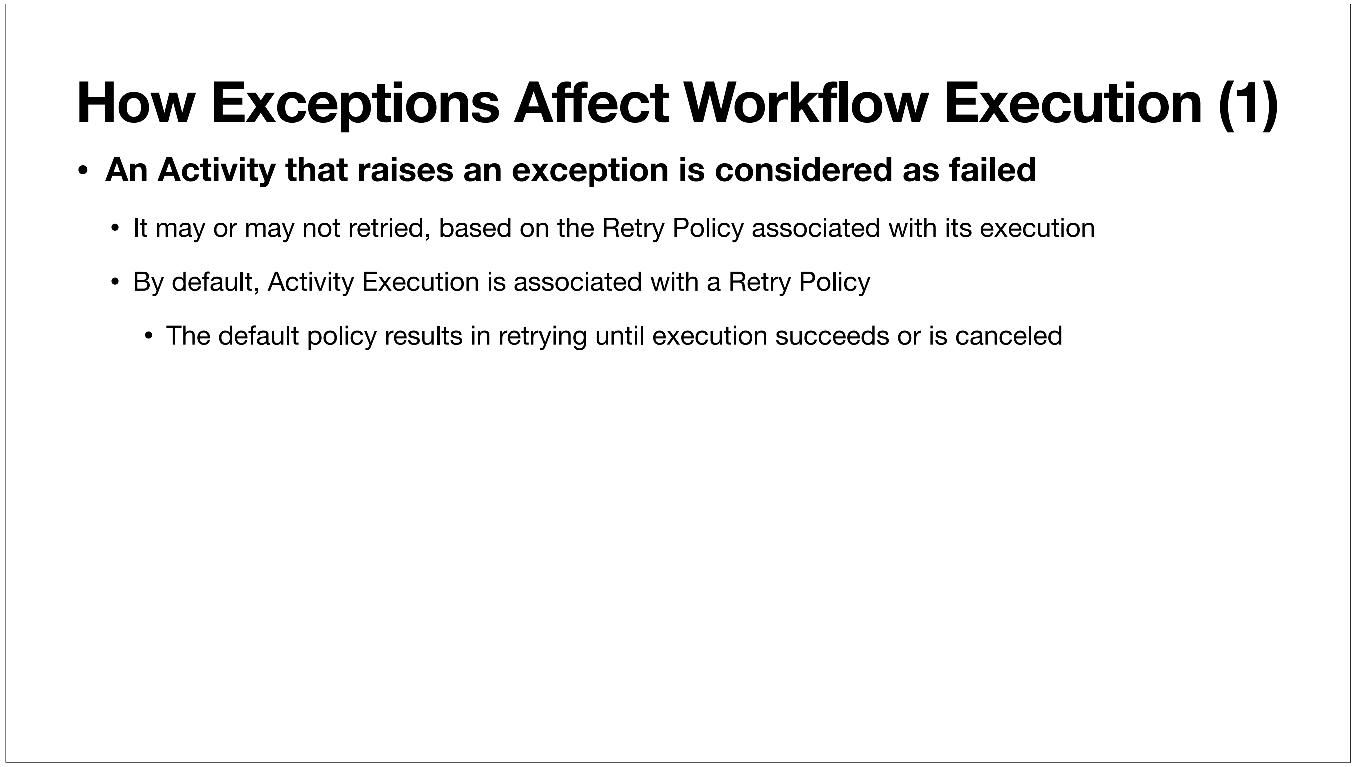
Workflow IDs

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

```
// Example: An order processing Workflow might include order number in the Workflow ID
WorkflowOptions options = WorkflowOptions.newBuilder()
        .setWorkflowId("translation-workflow-" + input.getOrderNumber())
        .setTaskQueue("translation-tasks").build();

OrderProcessingWorkflow workflow = client.newWorkflowStub(OrderProcessingWorkflow.class, options);
```

- **Must be unique among all *running* Workflow Executions in the namespace**
  - This constraint applies across *all* Workflow Types, not just those of the *same Type*
  - This is an important consideration for choosing a Workflow ID

You have to provide the task queue when you define your client and your worker. It’s best to use a constant so you are sure you’re using the same task queue name everywhere. It’s easy to accidentally create a task on a task queue that no workers are watching because the workers aren’t watching the same task queue the client started the task on!



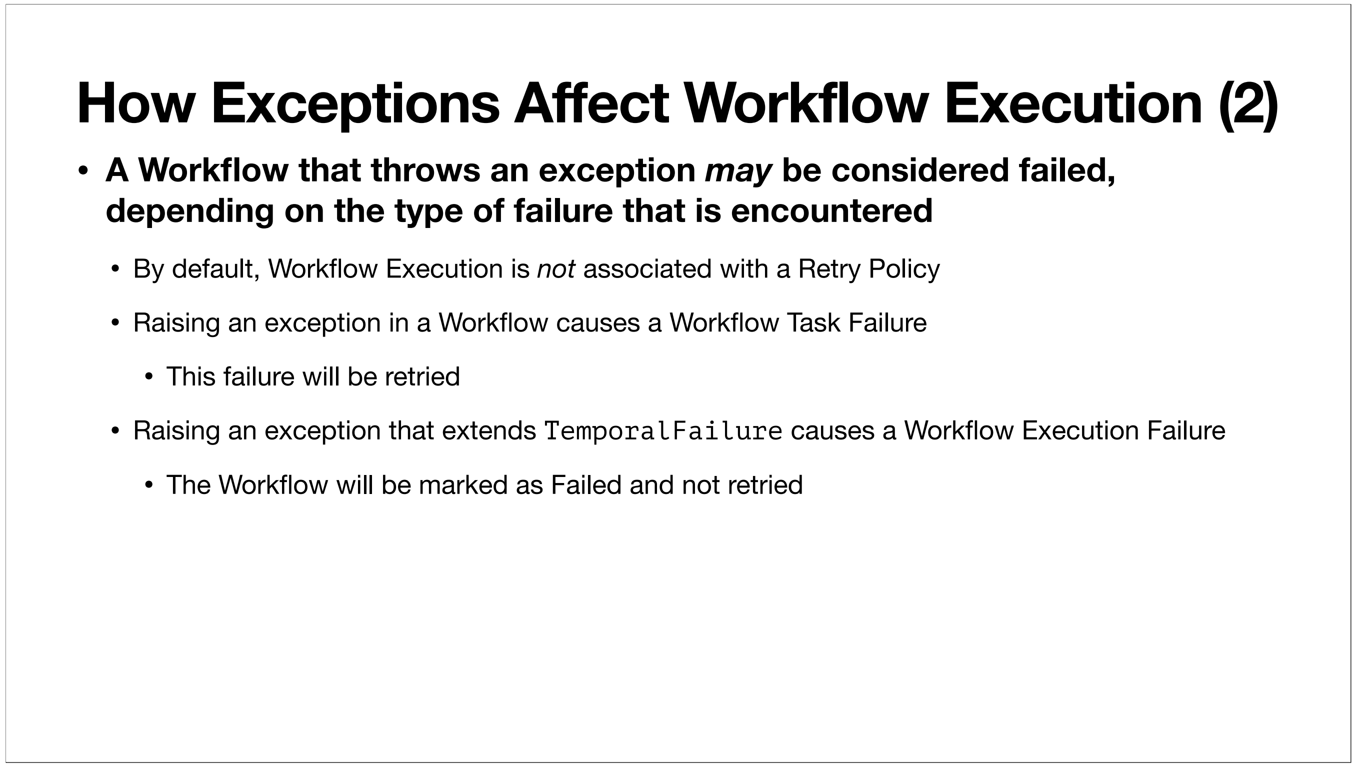
How Exceptions Affect Workflow Execution (1)

- **An Activity that raises an exception is considered as failed**
  - It may or may not retried, based on the Retry Policy associated with its execution
  - By default, Activity Execution is associated with a Retry Policy
    - The default policy results in retrying until execution succeeds or is canceled

An Activity that raises an exception is considered as failed

This is intended. We expect for Activities to possibly fail. So we want them to retry unless we explicity say not to

A Workflow that raises an exception may be considered failed, depending on the type of failure that is encountered



How Exceptions Affect Workflow Execution (2)

- **A Workflow that throws an exception *may* be considered failed, depending on the type of failure that is encountered**
  - By default, Workflow Execution is *not* associated with a Retry Policy
  - Raising an exception in a Workflow causes a Workflow Task Failure
    - This failure will be retried
  - Raising an exception that extends `TemporalFailure` causes a Workflow Execution Failure
    - The Workflow will be marked as Failed and not retried

An Activity that raises an exception is considered as failed

> This is intended. We expect for Activities to possibly fail. So we want them to retry unless we explicity say not to

A Workflow that raises an exception may be considered failed, depending on the type of failure that is encountered

How to Raise Exceptions in Activity Code

- You can throw exceptions as necessary in Activities

- Wrapping an exception allows you to throw it without explicitly declaring the Exception. These failures will be retried

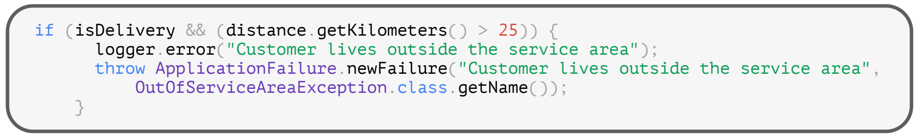
```java
try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
  String line;
  while ((line = in.readLine()) != null) {
    builder.append(line);
  }
} catch (IOException e) {
  throw Activity.wrap(e);
}
```

- If you don't want an Activity to retry, you can specify it as a Non-Retryable Failure

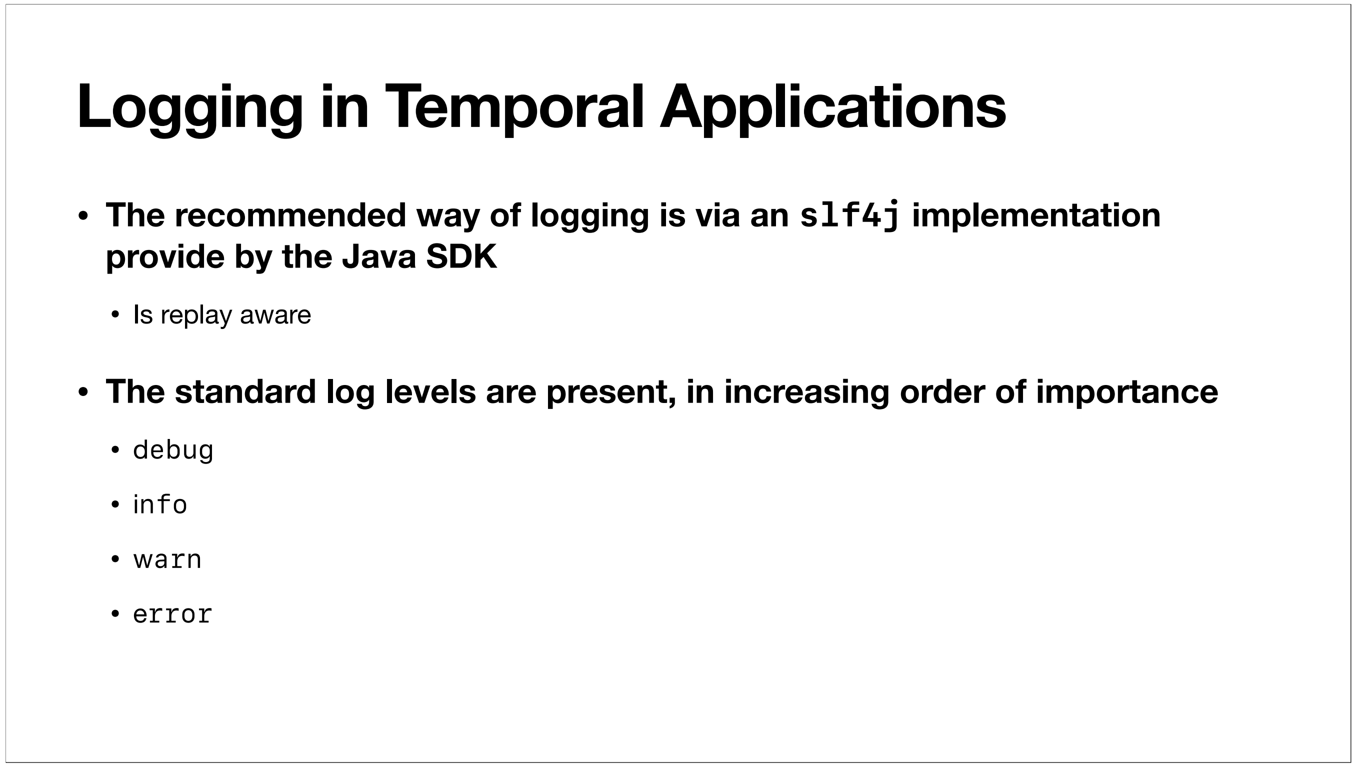
```java
RetryOptions retryOptions = RetryOptions.newBuilder()
     .setInitialInterval(Duration.ofSeconds(15))
     .setBackoffCoefficient(2.0)
     .setMaximumInterval(Duration.ofSeconds(60))
     .setMaximumAttempts(100)
     .setDoNotRetry("IOException")
     .build();
```

How to Raise Exceptions in Workflow Code

- **You can throw exceptions as necessary in Workflows**
  - If this exception does not extend `TemporalFailure` or its child classes, this will fail the Workflow Task and will be retried
    - This is known as a `WorkflowTaskFailure`
  - If this exception extends `TemporalFailure` or its child classes, this will fail the Workflow Execution and the failure will not be retried.
    - This is known as a `WorkflowExecutionFailure`
      - `ApplicationFailure` is a child of `TemporalFailure`



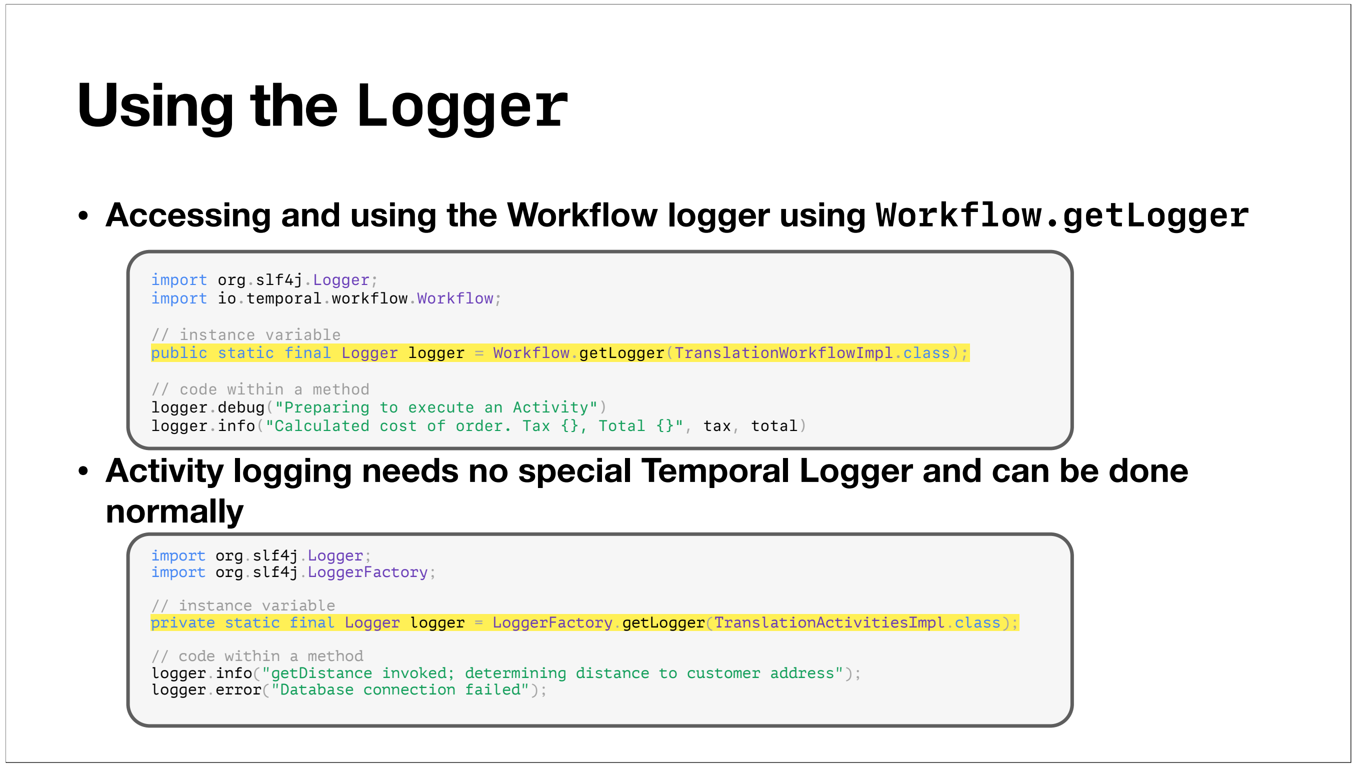
```
if (isDelivery && (distance.getKilometers() > 25)) {
      logger.error("Customer lives outside the service area");
      throw ApplicationFailure.newFailure("Customer lives outside the service area",
          OutOfServiceAreaException.class.getName());
    }
```



Logging in Temporal Applications

- **The recommended way of logging is via an `slf4j` implementation provide by the Java SDK**
  - Is replay aware
- **The standard log levels are present, in increasing order of importance**
  - `debug`
  - `info`
  - `warn`
  - `error`

Another way to improve your Temporal code is to incorporate logging into your workflows and activities. The SDK provides a logger you can use, with Debug, Info, Warn, and Error levels.



Using the Logger

- Accessing and using the Workflow logger using `Workflow.getLogger`

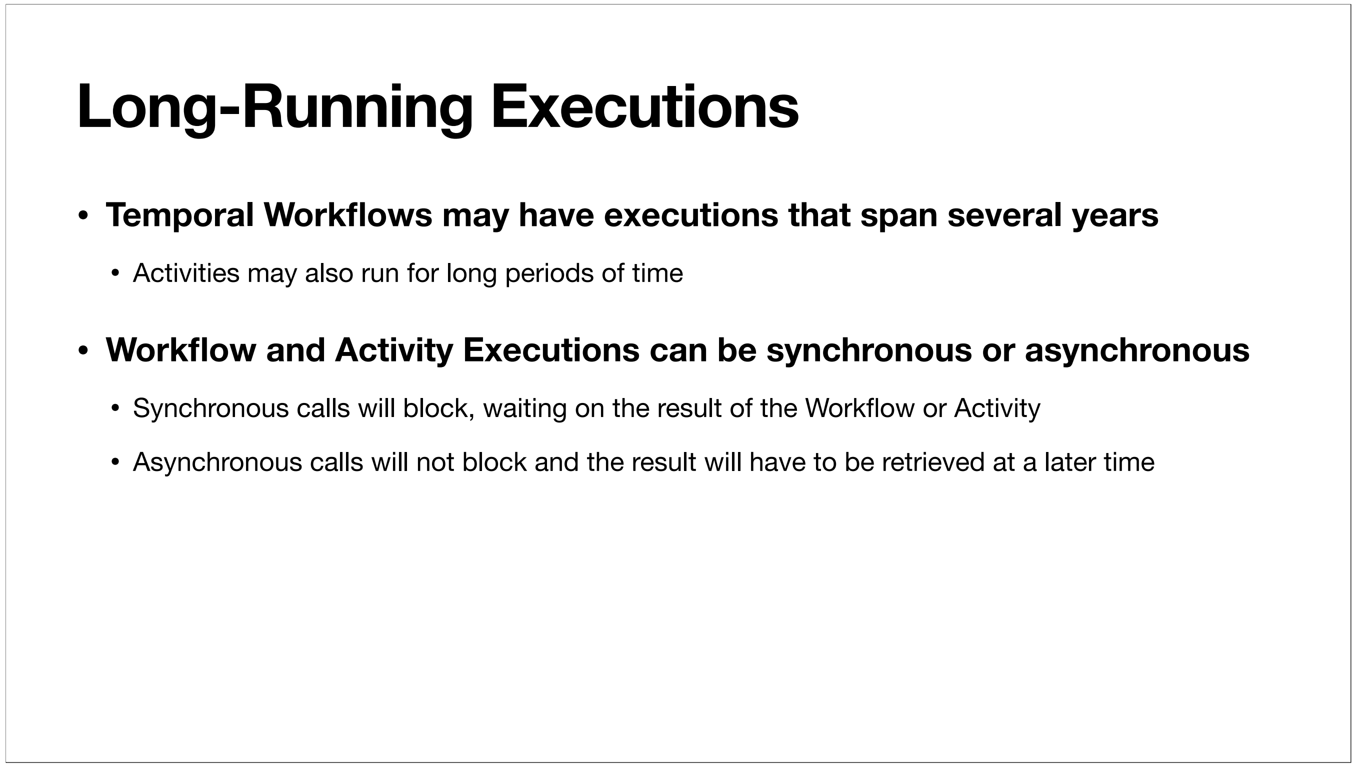
```
import org.slf4j.Logger;
import io.temporal.workflow.Workflow;

// instance variable
public static final Logger logger = Workflow.getLogger(TranslationWorkflowImpl.class);

// code within a method
logger.debug("Preparing to execute an Activity")
logger.info("Calculated cost of order. Tax {}, Total {}", tax, total)
```

- Activity logging needs no special Temporal Logger and can be done normally

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

// instance variable
private static final Logger logger = LoggerFactory.getLogger(TranslationActivitiesImpl.class);

// code within a method
logger.info("getDistance invoked; determining distance to customer address");
logger.error("Database connection failed");
```

In Workflows, you get the Logger from the Worfklow Class
You can then log as expected

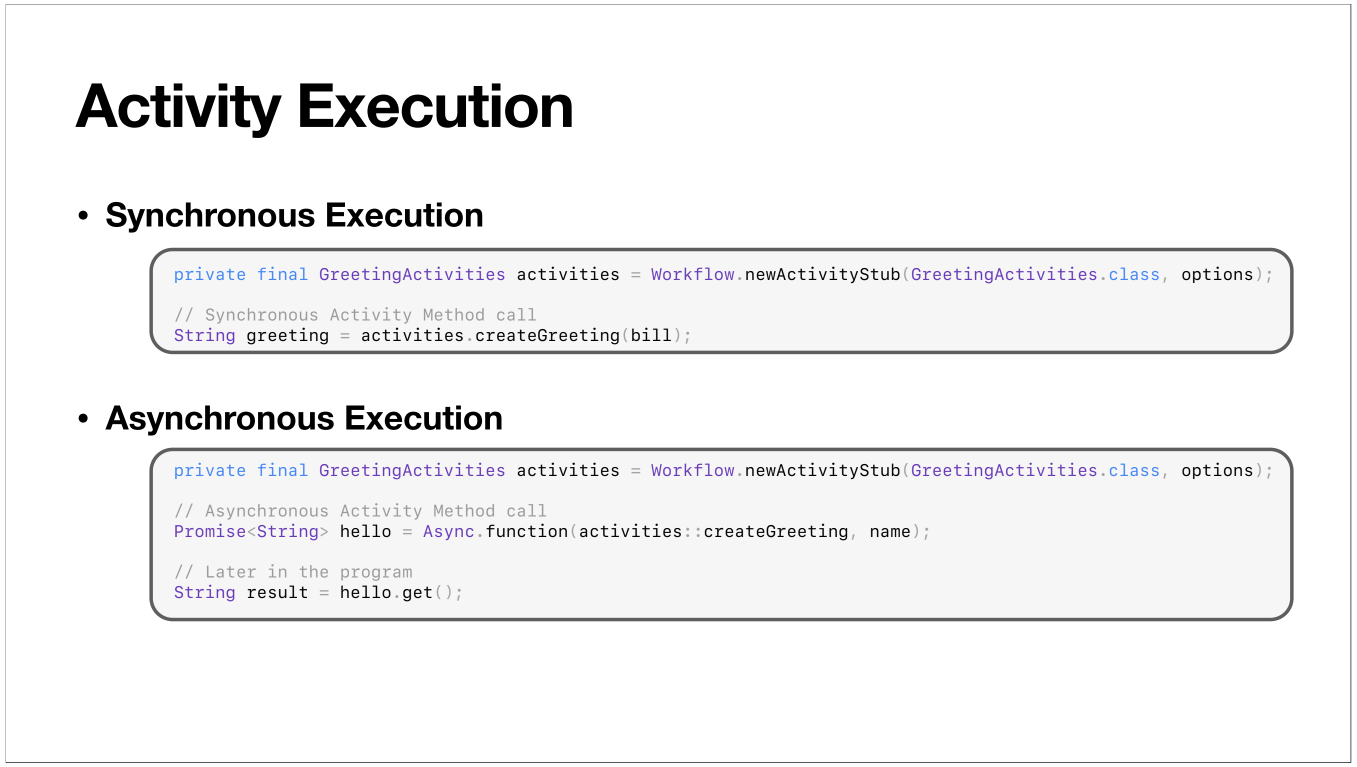
Activities don't have any special logging requirements, so you can just log as normal.



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 block, waiting on the result of the Workflow or Activity
  - Asynchronous calls will not block and the result will have to be retrieved at a later time

Sometimes you may have code that runs for extended periods of time. This may be your Workflow code or your Activity code. We'll only focus on long-running Workflows in this course. There are additional considerations for long-running Activities that are beyond the scope of this course.



Activity Execution

- **Synchronous Execution**

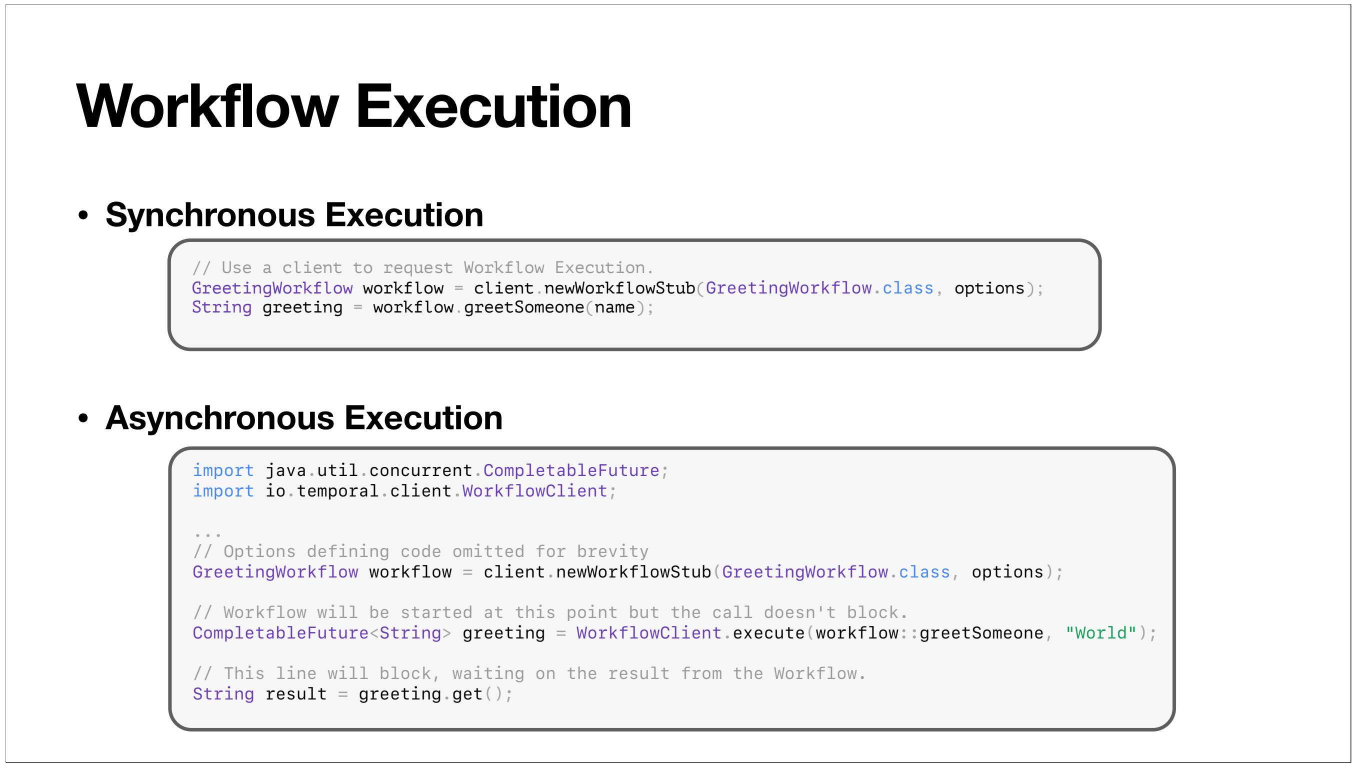
```
private final GreetingActivities activities = Workflow.newActivityStub(GreetingActivities.class, options);

// Synchronous Activity Method call
String greeting = activities.createGreeting(bill);
```

- **Asynchronous Execution**

```
private final GreetingActivities activities = Workflow.newActivityStub(GreetingActivities.class, options);

// Asynchronous Activity Method call
Promise<String> hello = Async.function(activities::createGreeting, name);

// Later in the program
String result = hello.get();
```

So far we have demonstrated Workflows and Activities. Using blocking calls.

As. You can see, we can call Activities either Synchronously or Asynchronously



Workflow Execution

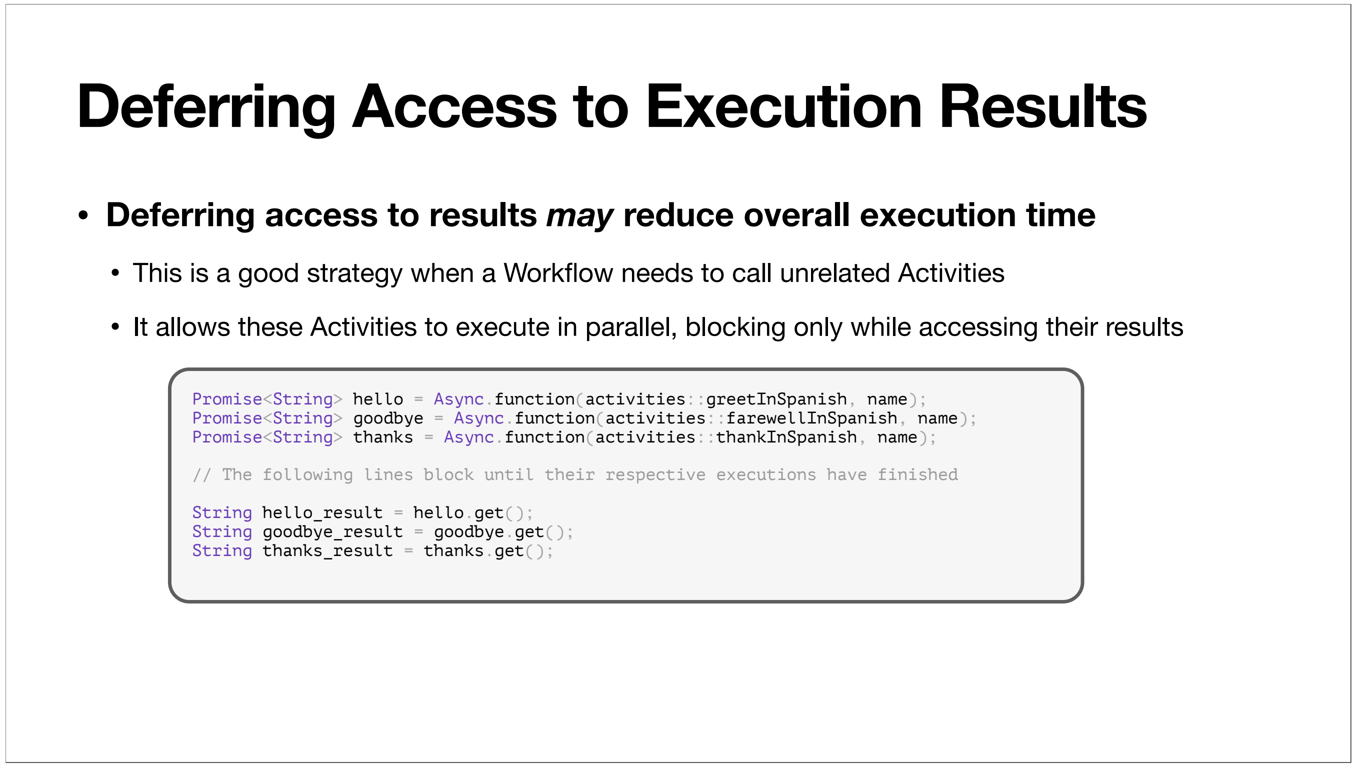
- **Synchronous Execution**

```java
// Use a client to request Workflow Execution.
GreetingWorkflow workflow = client.newWorkflowStub(GreetingWorkflow.class, options);
String greeting = workflow.greetSomeone(name);
```

- **Asynchronous Execution**

```java
import java.util.concurrent.CompletableFuture;
import io.temporal.client.WorkflowClient;

...
// Options defining code omitted for brevity
GreetingWorkflow workflow = client.newWorkflowStub(GreetingWorkflow.class, options);

// Workflow will be started at this point but the call doesn't block.
CompletableFuture<String> greeting = WorkflowClient.execute(workflow::greetSomeone, "World");

// This line will block, waiting on the result from the Workflow.
String result = greeting.get();
```

Same goes for workflows

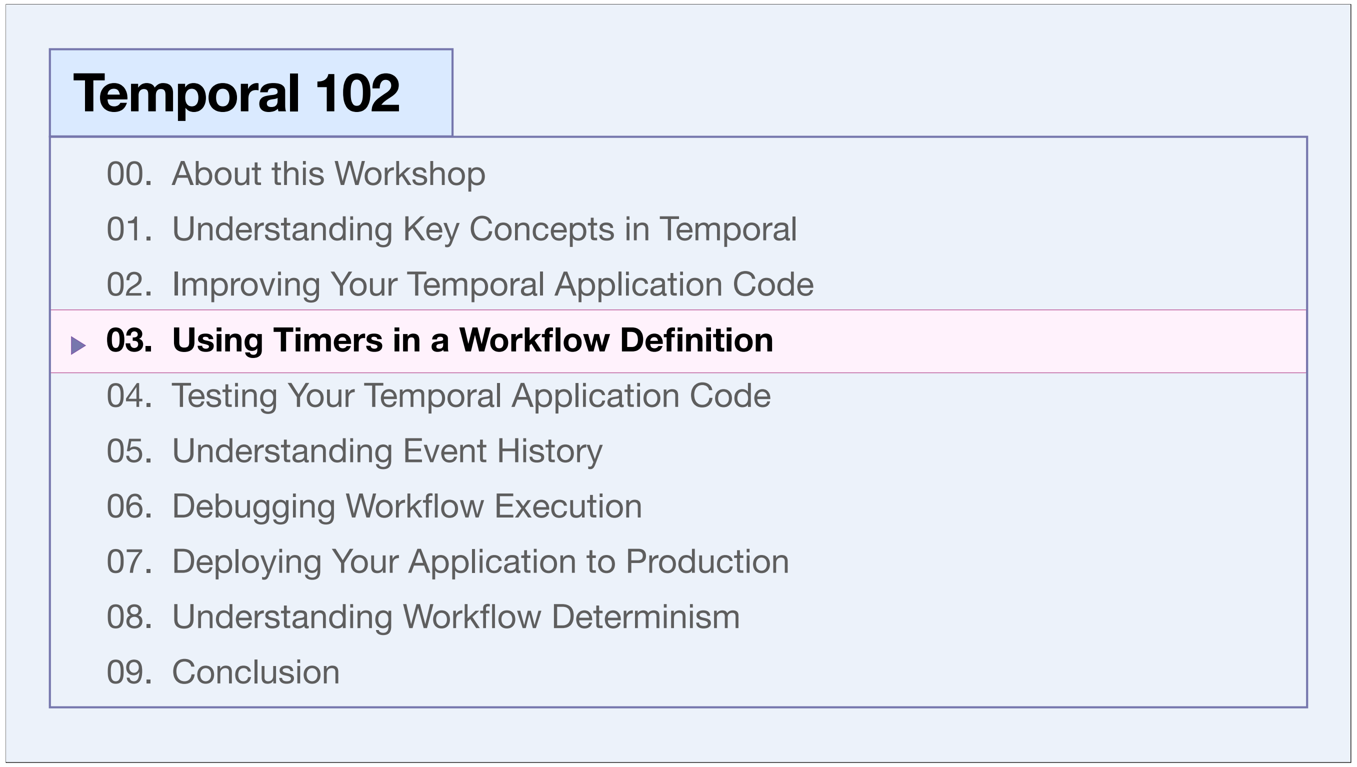


Deferring Access to Execution Results

- **Deferring access to results *may* reduce overall execution time**
  - This is a good strategy when a Workflow needs to call unrelated Activities
  - It allows these Activities to execute in parallel, blocking only while accessing their results

```
Promise<String> hello = Async.function(activities::greetInSpanish, name);
Promise<String> goodbye = Async.function(activities::farewellInSpanish, name);
Promise<String> thanks = Async.function(activities::thankInSpanish, name);

// The following lines block until their respective executions have finished

String hello_result = hello.get();
String goodbye_result = goodbye.get();
String thanks_result = thanks.get();
```

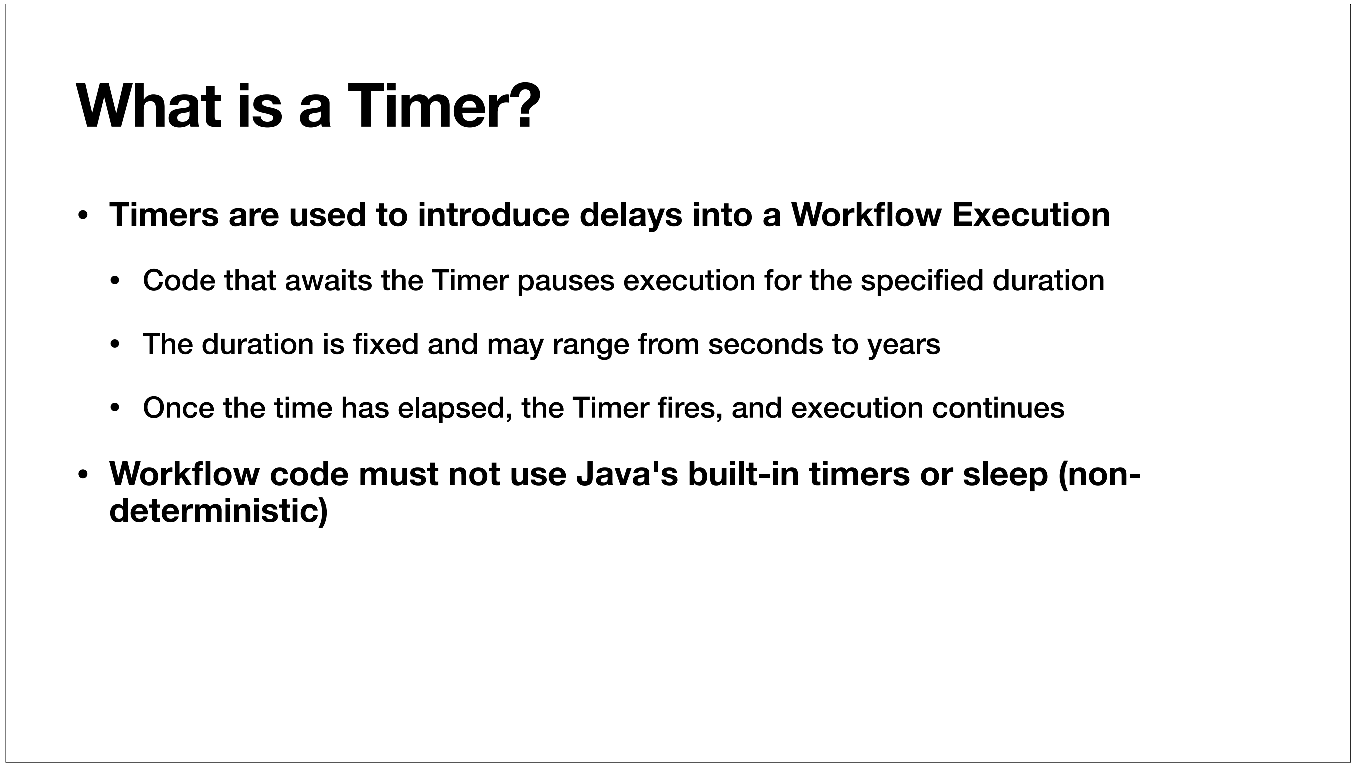
If you need to execute several Activities independently from one another, then you may be able to significantly reduce the overall execution time of the Workflow by separating the execution requests from the retrieval of the results. Assuming that your Workers and the downstream systems used in your Activities have sufficient capacity, this will execute the Activities in parallel.



Temporal 102

00. About this Workshop
01. Understanding Key Concepts in Temporal
02. Improving Your Temporal Application Code
▸ **03. Using Timers in a Workflow Definition**
04. Testing Your Temporal Application Code
05. Understanding Event History
06. Debugging Workflow Execution
07. Deploying Your Application to Production
08. Understanding Workflow Determinism
09. Conclusion

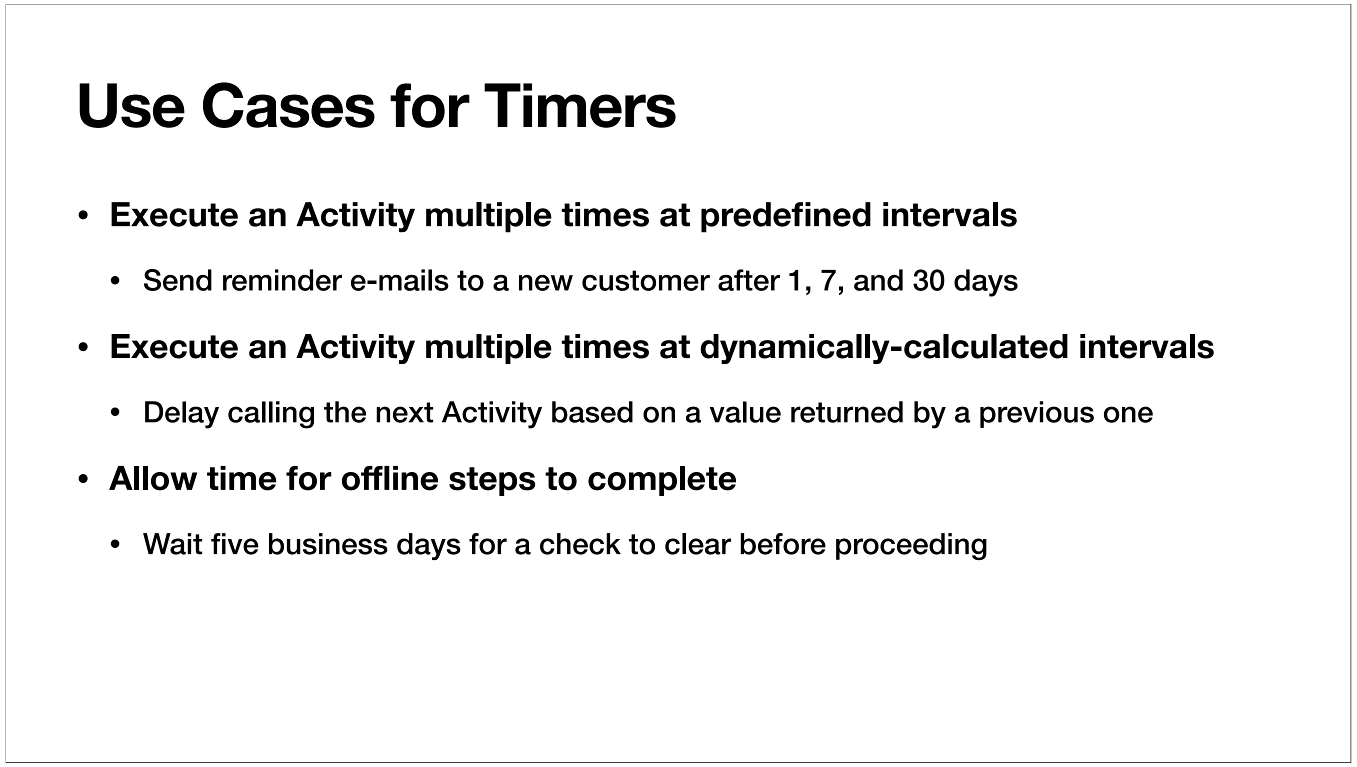
Next, we'll look at Timers,



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
- **Workflow code must not use Java's built-in timers or sleep (non-deterministic)**

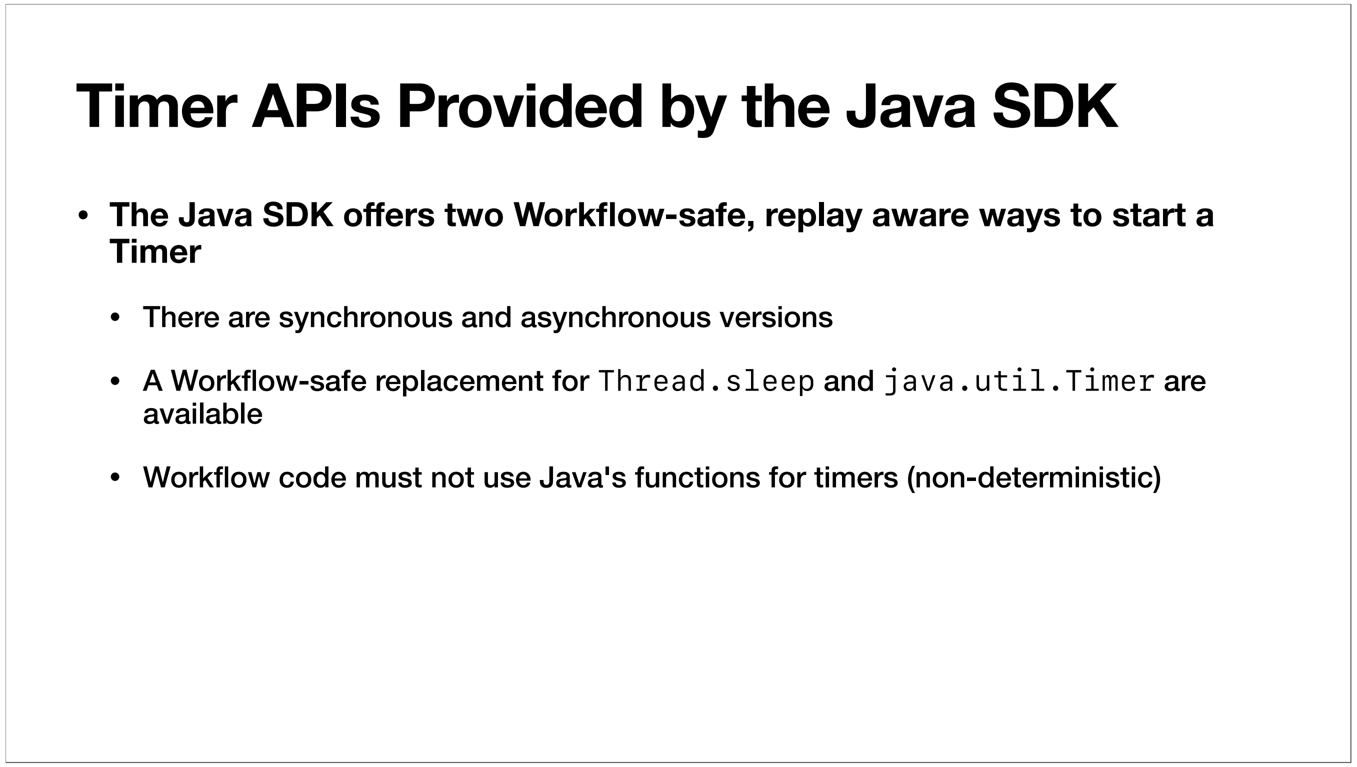
Timers are used to introduce delays into a Workflow Execution
Code that awaits the Timer pauses execution for the specified duration
The duration is fixed and may range from seconds to years
Once the time has elapsed, the Timer fires, and execution continues
Workflow code must not use Java's built-in timers or sleep (non-deterministic)



Use Cases for Timers

- **Execute an Activity multiple times at predefined intervals**
  - Send reminder e-mails to a new customer after 1, 7, and 30 days
- **Execute an Activity multiple times at dynamically-calculated intervals**
  - Delay calling the next Activity based on a value returned by a previous one
- **Allow time for offline steps to complete**
  - Wait five business days for a check to clear before proceeding

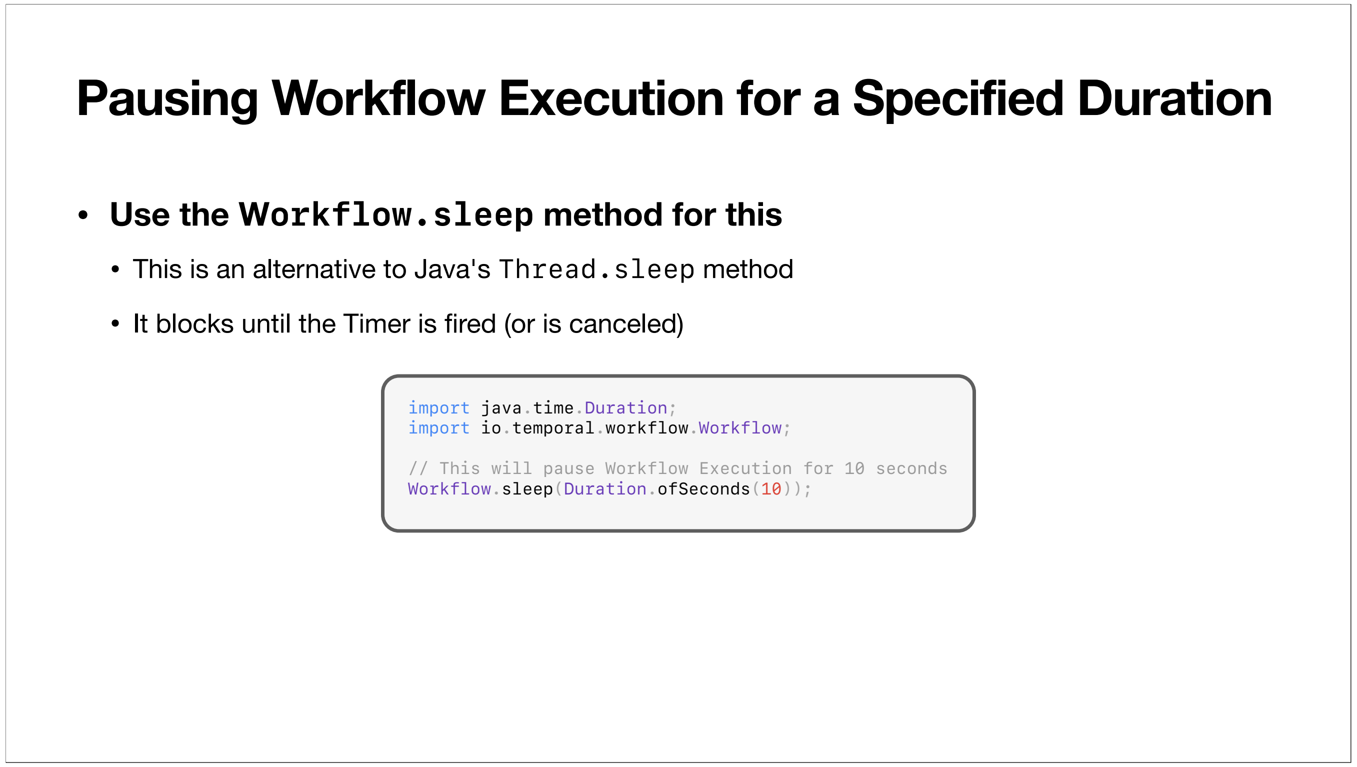
Execute an Activity multiple times at predefined intervals
Send reminder e-mails to a new customer after 1, 7, and 30 days
Execute an Activity multiple times at dynamically-calculated intervals
Delay calling the next Activity based on a value returned by a previous one
Allow time for offline steps to complete
Wait five business days for a check to clear before proceeding



Timer APIs Provided by the Java SDK

- **The Java SDK offers two Workflow-safe, replay aware ways to start a Timer**
  - There are synchronous and asynchronous versions
  - A Workflow-safe replacement for `Thread.sleep` and `java.util.Timer` are available
  - Workflow code must not use Java's functions for timers (non-deterministic)

The Java SDK offers two Workflow-safe, replay aware ways to start a Timer
There are synchronous and asynchronous versions
A Workflow-safe replacement for Thread.sleep and java.util.Timer are available
Workflow code must not use Java's functions for timers (non-deterministic)



Pausing Workflow Execution for a Specified Duration

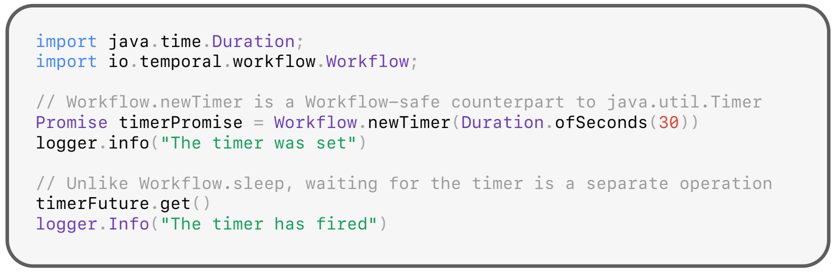
- **Use the `Workflow.sleep` method for this**
  - This is an alternative to Java's `Thread.sleep` method
  - It blocks until the Timer is fired (or is canceled)

```
import java.time.Duration;
import io.temporal.workflow.Workflow;

// This will pause Workflow Execution for 10 seconds
Workflow.sleep(Duration.ofSeconds(10));
```

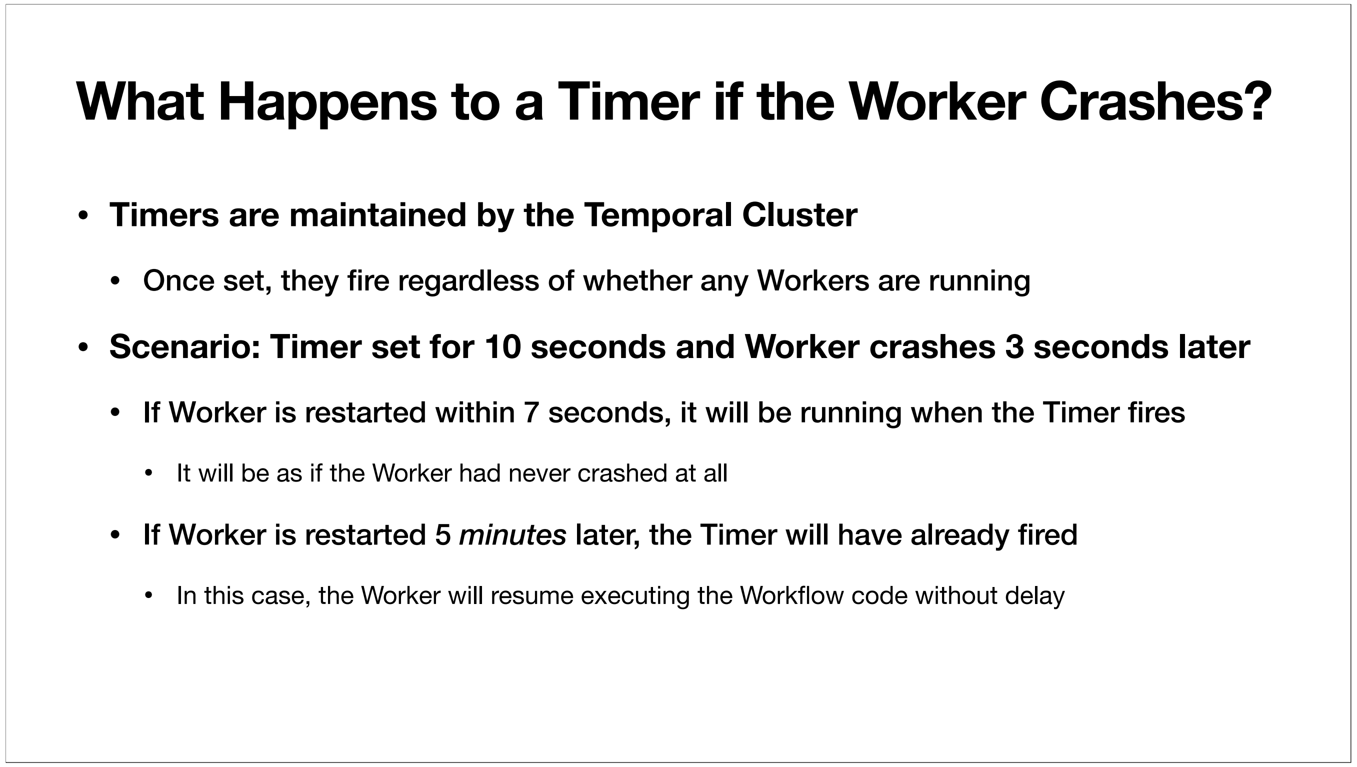
You use the sleep method in your Java workflows to create a Timer. There’s a nice string interface where you can specify the time duration.

Running Code a Specific Point in the Future

- **Use the `Workflow.newTimer` method for this**
  - This is an alternative to Java's `java.util.NewTimer` method
  - This returns a `Promise`, which becomes ready when the Timer fires (or is canceled)



```
import java.time.Duration;
import io.temporal.workflow.Workflow;

// Workflow.newTimer is a Workflow-safe counterpart to java.util.Timer
Promise timerPromise = Workflow.newTimer(Duration.ofSeconds(30))
logger.info("The timer was set")

// Unlike Workflow.sleep, waiting for the timer is a separate operation
timerFuture.get()
logger.Info("The timer has fired")
```



What Happens to a Timer if the Worker Crashes?

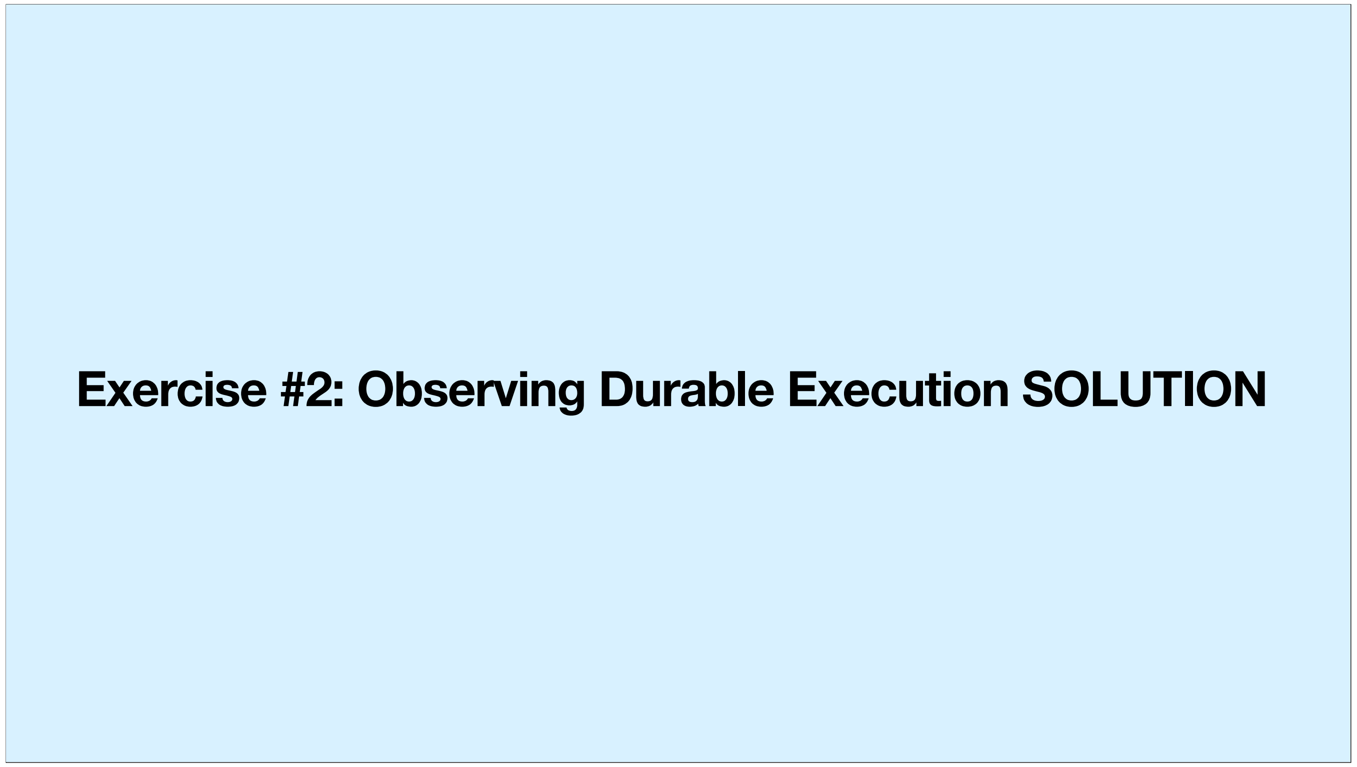
- **Timers are maintained by the Temporal Cluster**
  - Once set, they fire regardless of whether any Workers are running
- **Scenario: Timer set for 10 seconds and Worker crashes 3 seconds later**
  - If Worker is restarted within 7 seconds, it will be running when the Timer fires
    - It will be as if the Worker had never crashed at all
  - 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

Timers are maintained by the Temporal cluster. They fire even if there’s no worker running.

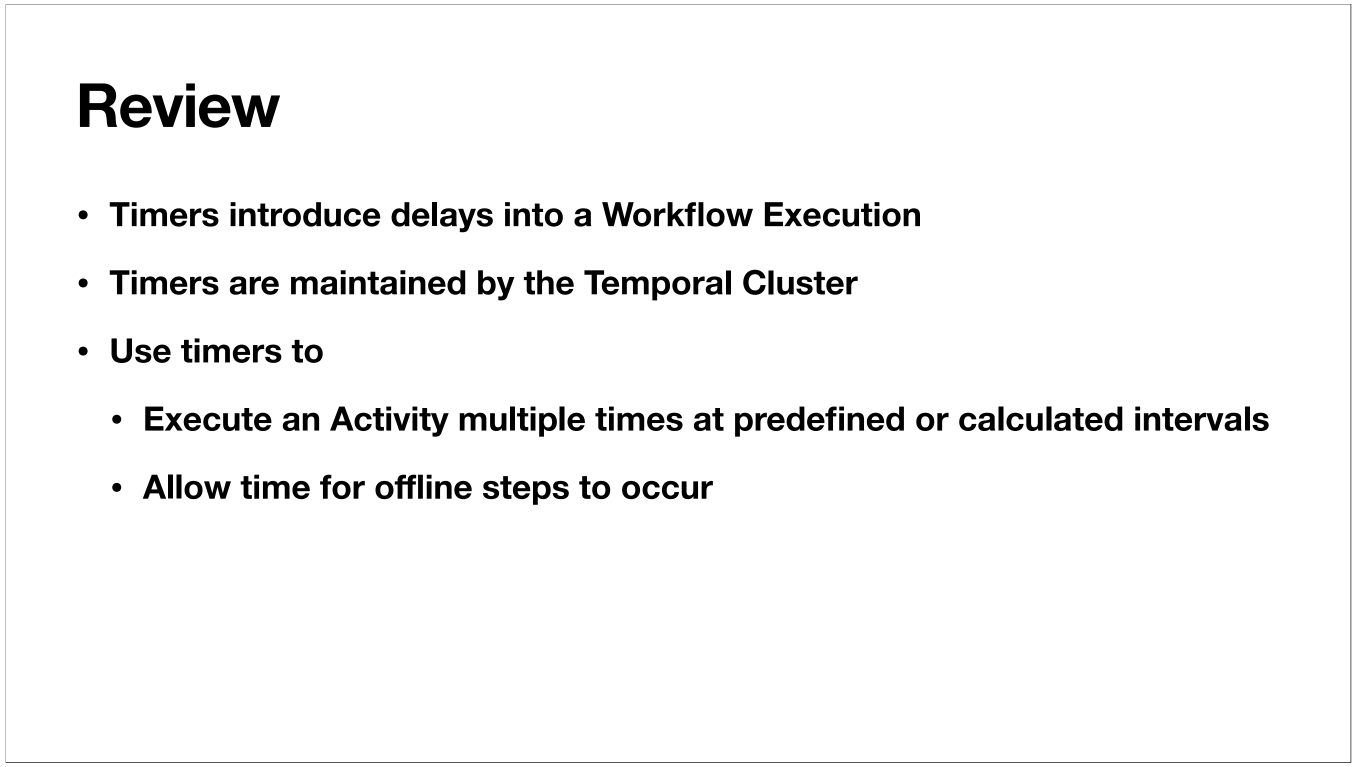
If a timer crashes and the timer has expired, the workflow resumes immediately. If there’s still time left, then the worker will be running by the time the Worker fires.

Exercise #2: 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



Exercise #2: Observing Durable Execution SOLUTION

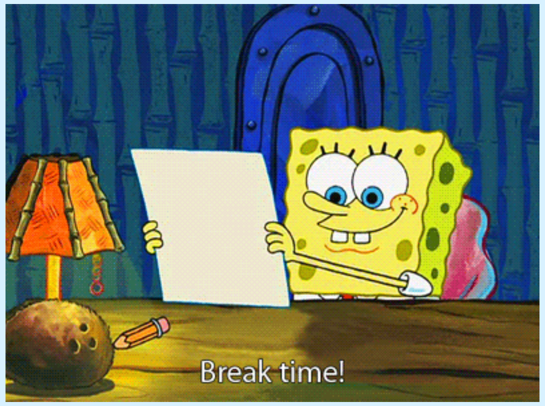


Review

- Timers introduce delays into a Workflow Execution
- Timers are maintained by the Temporal Cluster
- Use timers to
  - Execute an Activity multiple times at predefined or calculated intervals
  - Allow time for offline steps to occur

Timers introduce delays into a Workflow Execution
Timers are maintained by the Temporal Cluster
Use timers to
Execute an Activity multiple times at predefined or calculated intervals
Allow time for offline steps to occur

10 minute break



Temporal 102

00. About this Workshop

01. Understanding Key Concepts in Temporal

02. Improving Your Temporal Application Code

03. Using Timers in a Workflow Definition

▶ **04. Testing Your Temporal Application Code**

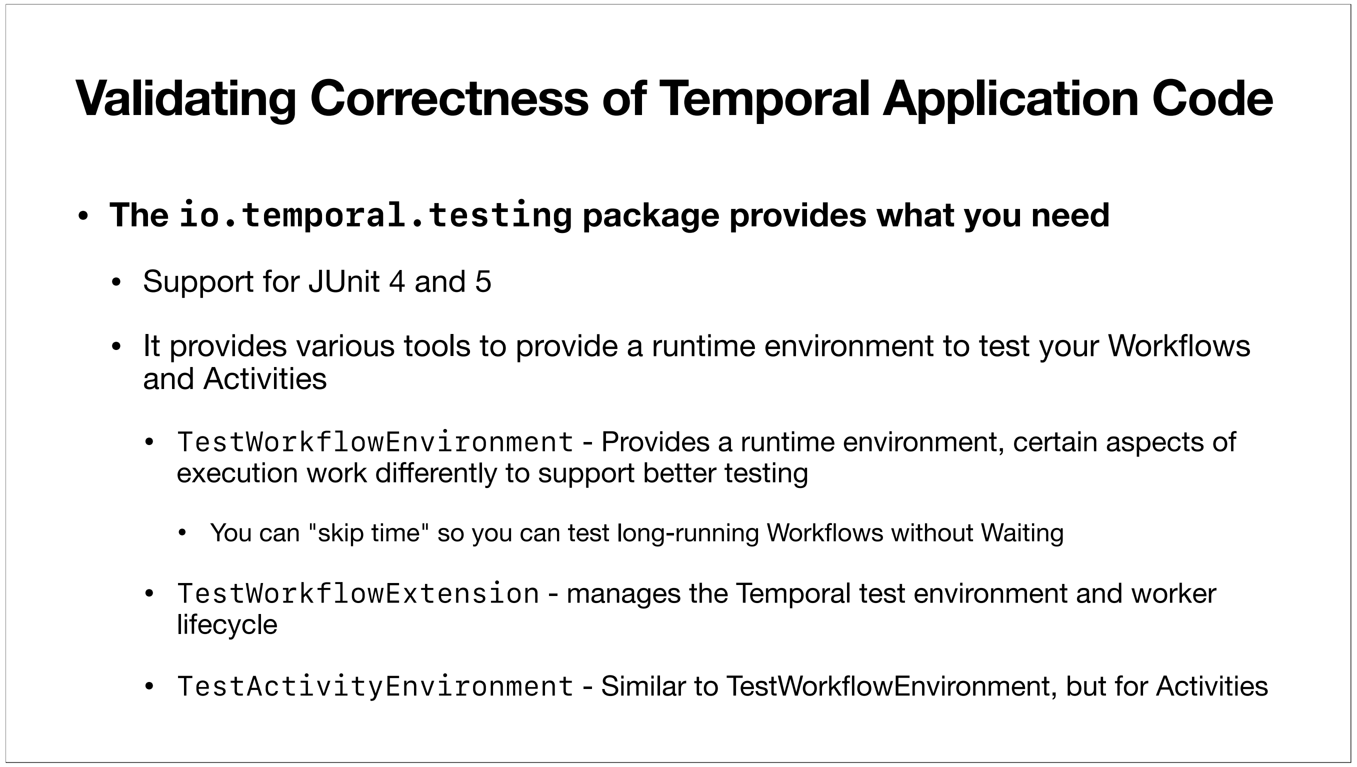
05. Understanding Event History

06. Debugging Workflow Execution

07. Deploying Your Application to Production

08. Understanding Workflow Determinism

09. Conclusion



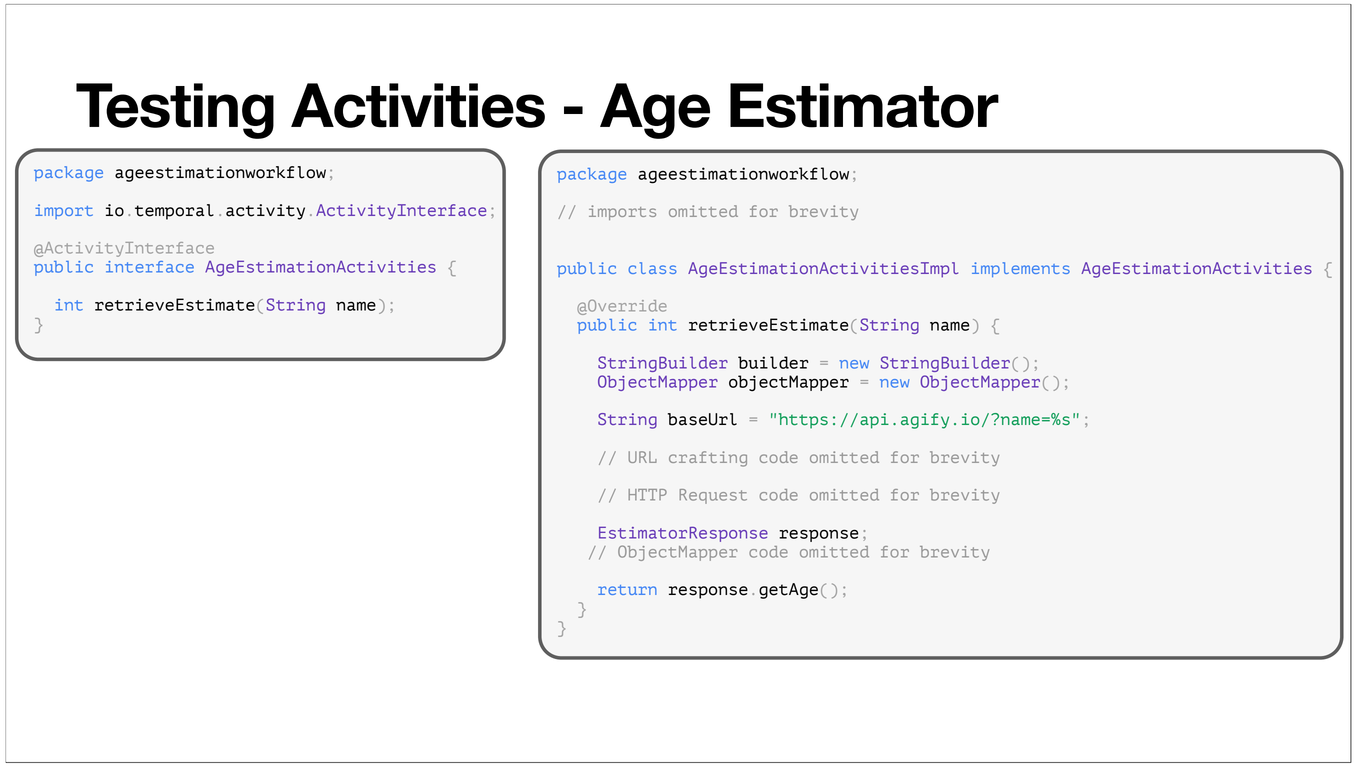
Validating Correctness of Temporal Application Code

- **The `io.temporal.testing` package provides what you need**
  - Support for JUnit 4 and 5
  - It provides various tools to provide a runtime environment to test your Workflows and Activities
    - `TestWorkflowEnvironment` - Provides a runtime environment, certain aspects of execution work differently to support better testing
      - You can "skip time" so you can test long-running Workflows without Waiting
    - `TestWorkflowExtension` - manages the Temporal test environment and worker lifecycle
    - `TestActivityEnvironment` - Similar to TestWorkflowEnvironment, but for Activities

As with other applications you develop, testing your Temporal applications helps to validate that your business logic works as you intended.

The io.temporal.testing package provides what you need to test Temporal applications.

TestWorkflowEnvironment - Provides a runtime environment, certain aspects of execution work differently to support better testing
You can "skip time" so you can test long-running Workflows without Waiting
TestWorkflowExtension - manages the Temporal test environment and worker lifecycle
TestActivityEnvironment - Similar to TestWorkflowEnvironment, but for Activities

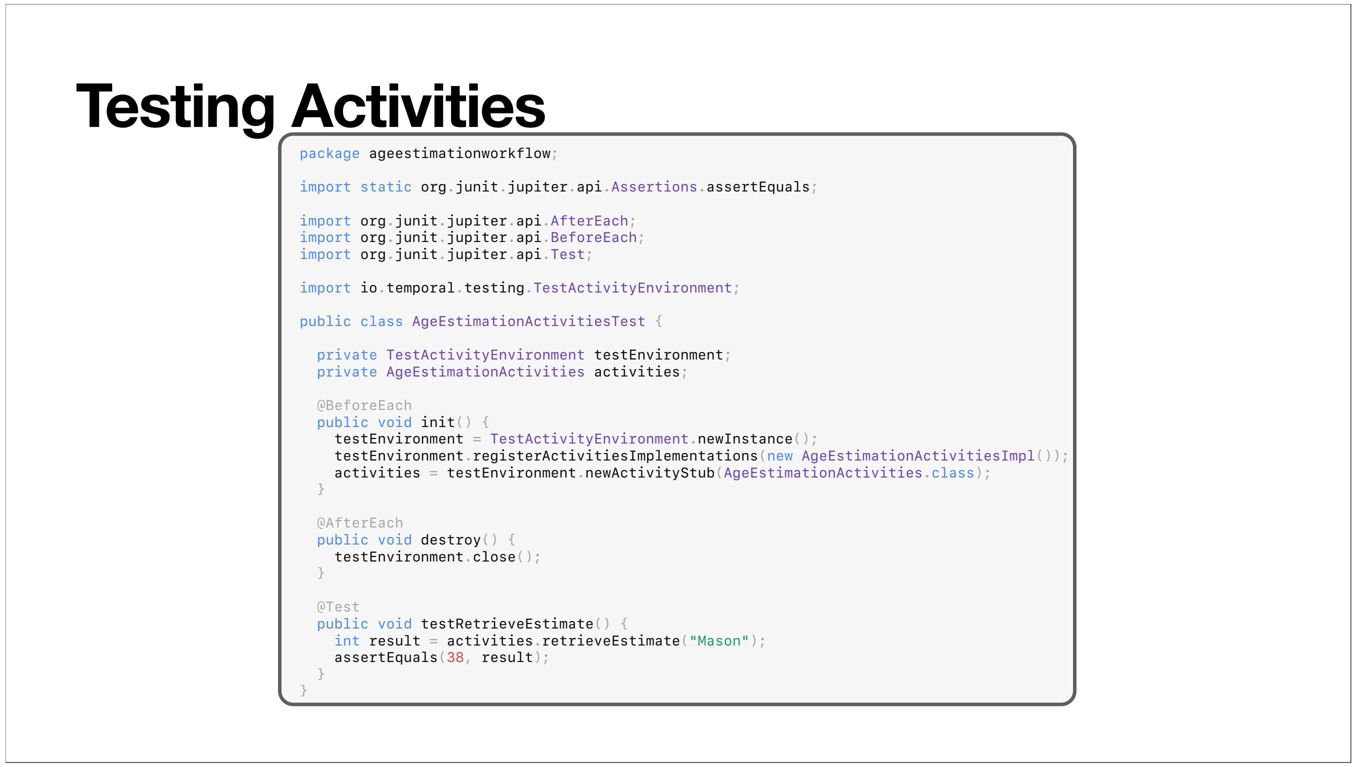


Testing Activities - Age Estimator

```java
package ageestimationworkflow;

import io.temporal.activity.ActivityInterface;

@ActivityInterface
public interface AgeEstimationActivities {

  int retrieveEstimate(String name);
}
```

```java
package ageestimationworkflow;

// imports omitted for brevity


public class AgeEstimationActivitiesImpl implements AgeEstimationActivities {

  @Override
  public int retrieveEstimate(String name) {

    StringBuilder builder = new StringBuilder();
    ObjectMapper objectMapper = new ObjectMapper();

    String baseUrl = "https://api.agify.io/?name=%s";

    // URL crafting code omitted for brevity

    // HTTP Request code omitted for brevity

    EstimatorResponse response;
   // ObjectMapper code omitted for brevity

    return response.getAge();
  }
}
```

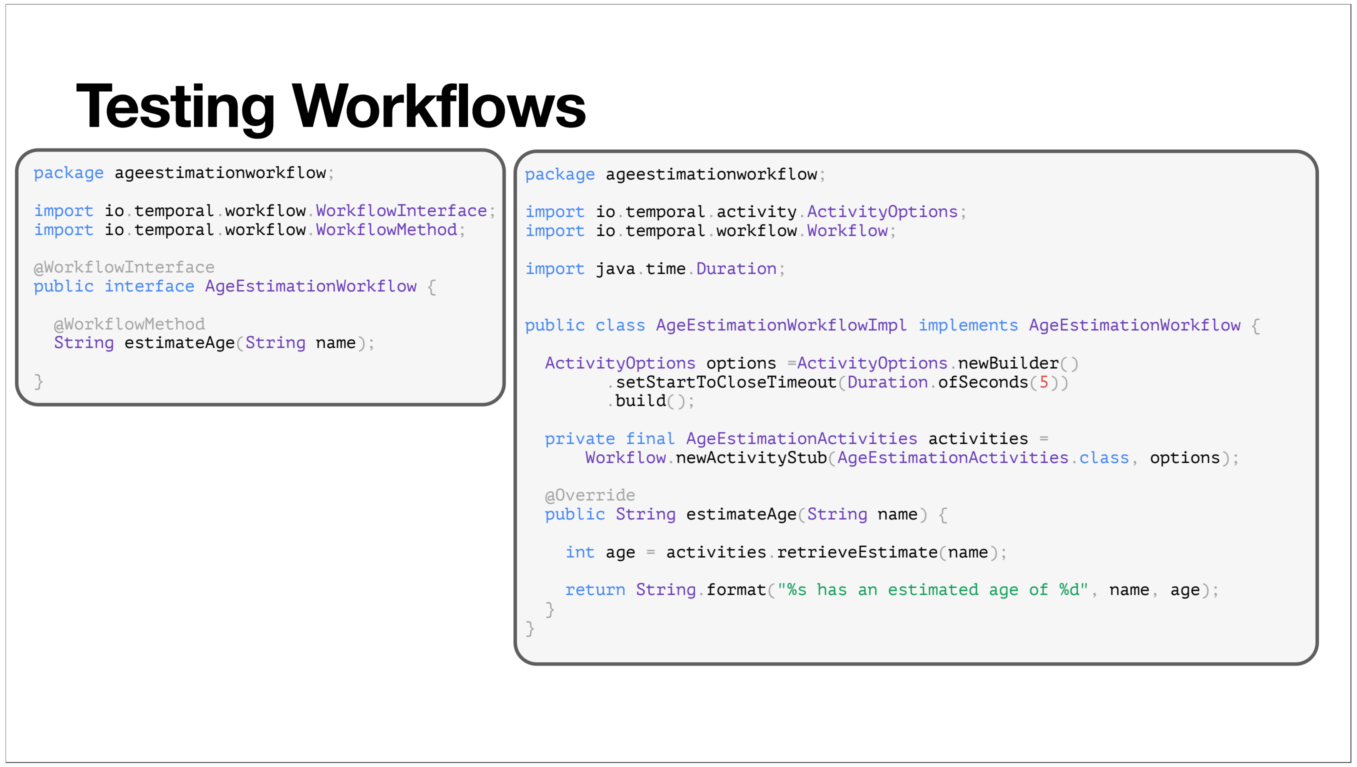
Here's a stubbed out Activity example of an application we'll test



Testing Activities

```java
package ageestimationworkflow;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import io.temporal.testing.TestActivityEnvironment;

public class AgeEstimationActivitiesTest {

  private TestActivityEnvironment testEnvironment;
  private AgeEstimationActivities activities;

  @BeforeEach
  public void init() {
    testEnvironment = TestActivityEnvironment.newInstance();
    testEnvironment.registerActivitiesImplementations(new AgeEstimationActivitiesImpl());
    activities = testEnvironment.newActivityStub(AgeEstimationActivities.class);
  }

  @AfterEach
  public void destroy() {
    testEnvironment.close();
  }

  @Test
  public void testRetrieveEstimate() {
    int result = activities.retrieveEstimate("Mason");
    assertEquals(38, result);
  }
}
```

The temporalio testing package provides TestActivityEnvironment, so you can test Activities in isolation.
You import TestActivityEnvironment, along with your activities and then use the test activity environment to run the activity You can use regular assertions to test the result.

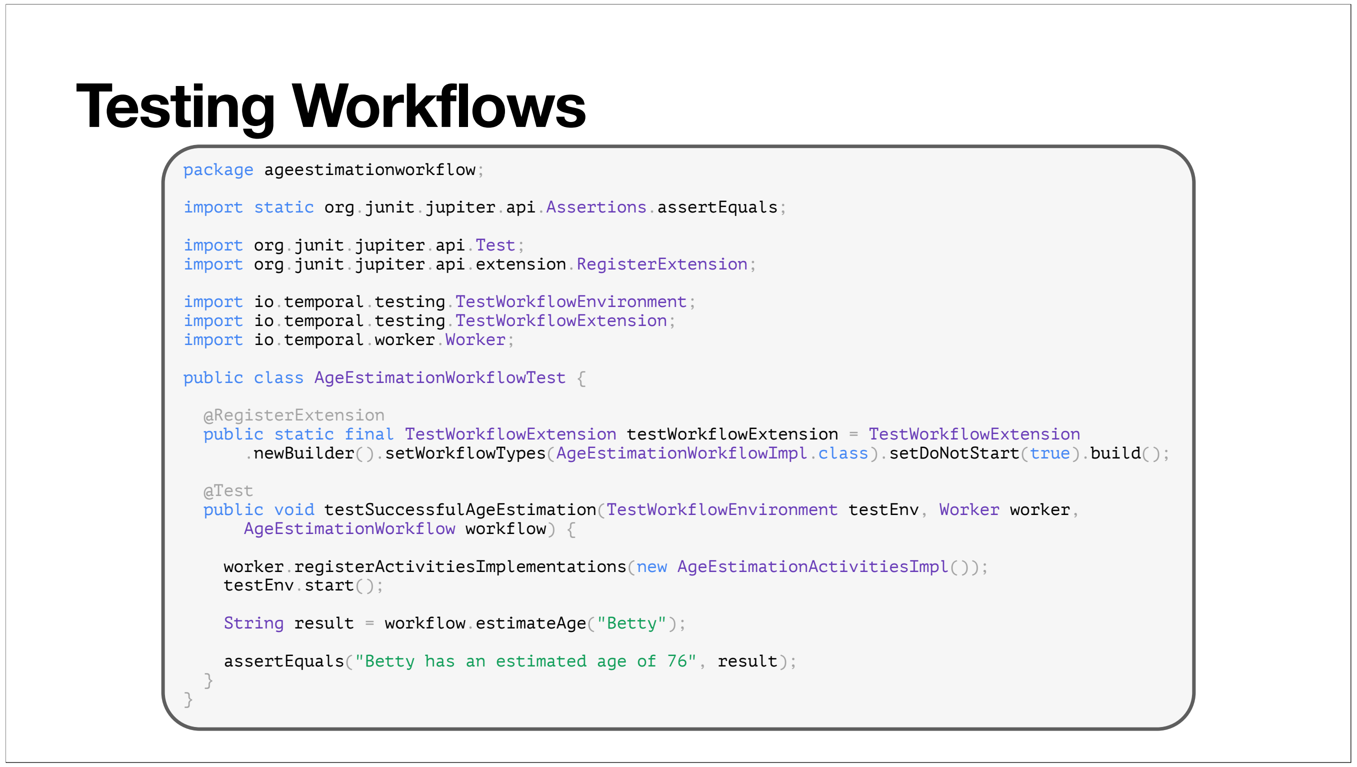


Testing Workflows

```java
package ageestimationworkflow;

import io.temporal.workflow.WorkflowInterface;
import io.temporal.workflow.WorkflowMethod;

@WorkflowInterface
public interface AgeEstimationWorkflow {

  @WorkflowMethod
  String estimateAge(String name);

}
```

```java
package ageestimationworkflow;

import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

import java.time.Duration;


public class AgeEstimationWorkflowImpl implements AgeEstimationWorkflow {

  ActivityOptions options =ActivityOptions.newBuilder()
         .setStartToCloseTimeout(Duration.ofSeconds(5))
         .build();

  private final AgeEstimationActivities activities =
      Workflow.newActivityStub(AgeEstimationActivities.class, options);

  @Override
  public String estimateAge(String name) {

    int age = activities.retrieveEstimate(name);

    return String.format("%s has an estimated age of %d", name, age);
  }
}
```

Next we'll test our Workflow that calls the retrieveEstimate Activity



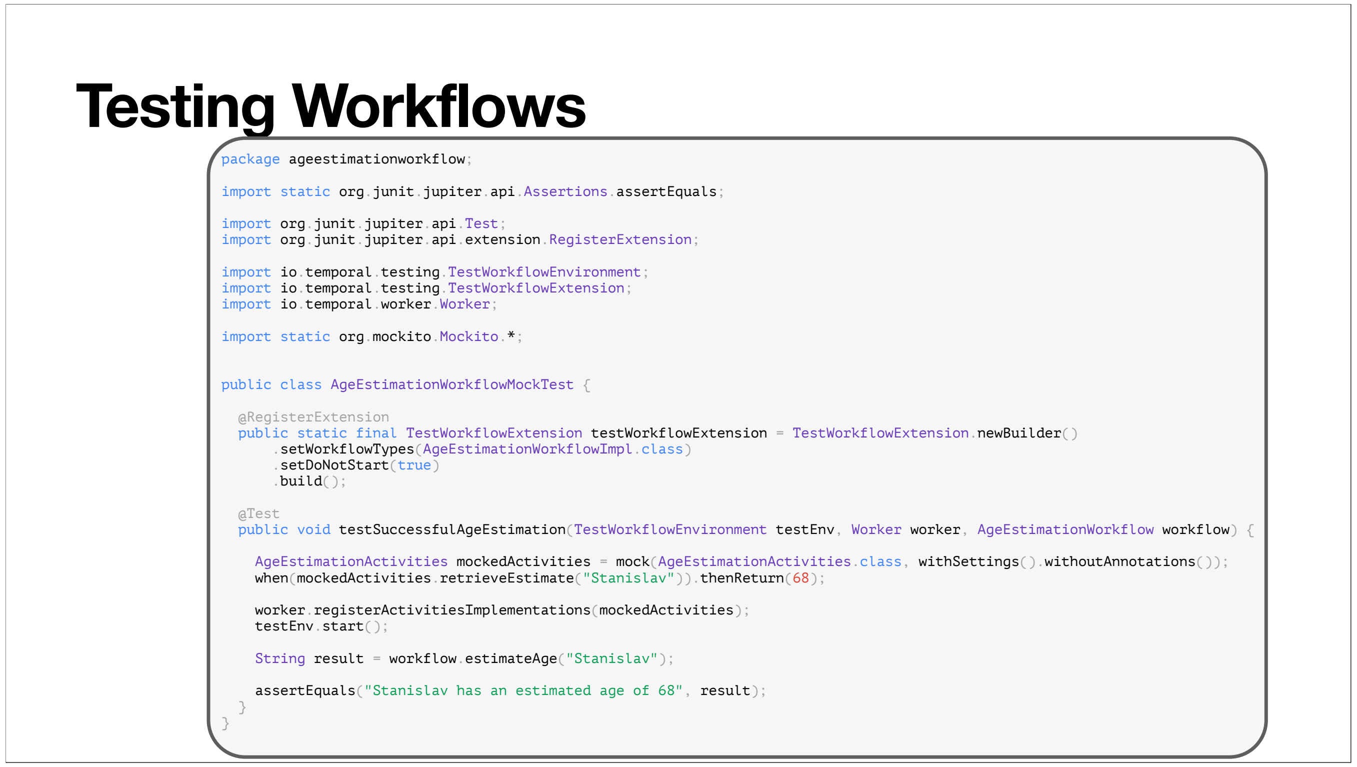
Testing Workflows

```java
package ageestimationworkflow;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;

import io.temporal.testing.TestWorkflowEnvironment;
import io.temporal.testing.TestWorkflowExtension;
import io.temporal.worker.Worker;

public class AgeEstimationWorkflowTest {

  @RegisterExtension
  public static final TestWorkflowExtension testWorkflowExtension = TestWorkflowExtension
      .newBuilder().setWorkflowTypes(AgeEstimationWorkflowImpl.class).setDoNotStart(true).build();

  @Test
  public void testSuccessfulAgeEstimation(TestWorkflowEnvironment testEnv, Worker worker,
      AgeEstimationWorkflow workflow) {

    worker.registerActivitiesImplementations(new AgeEstimationActivitiesImpl());
    testEnv.start();

    String result = workflow.estimateAge("Betty");

    assertEquals("Betty has an estimated age of 76", result);
  }
}
```

The TestWorkflowExtension is a helper class to handle the spin up and tear down of the TestWorkflowEnvironment and Worker

Mocking Activities in Workflow Tests

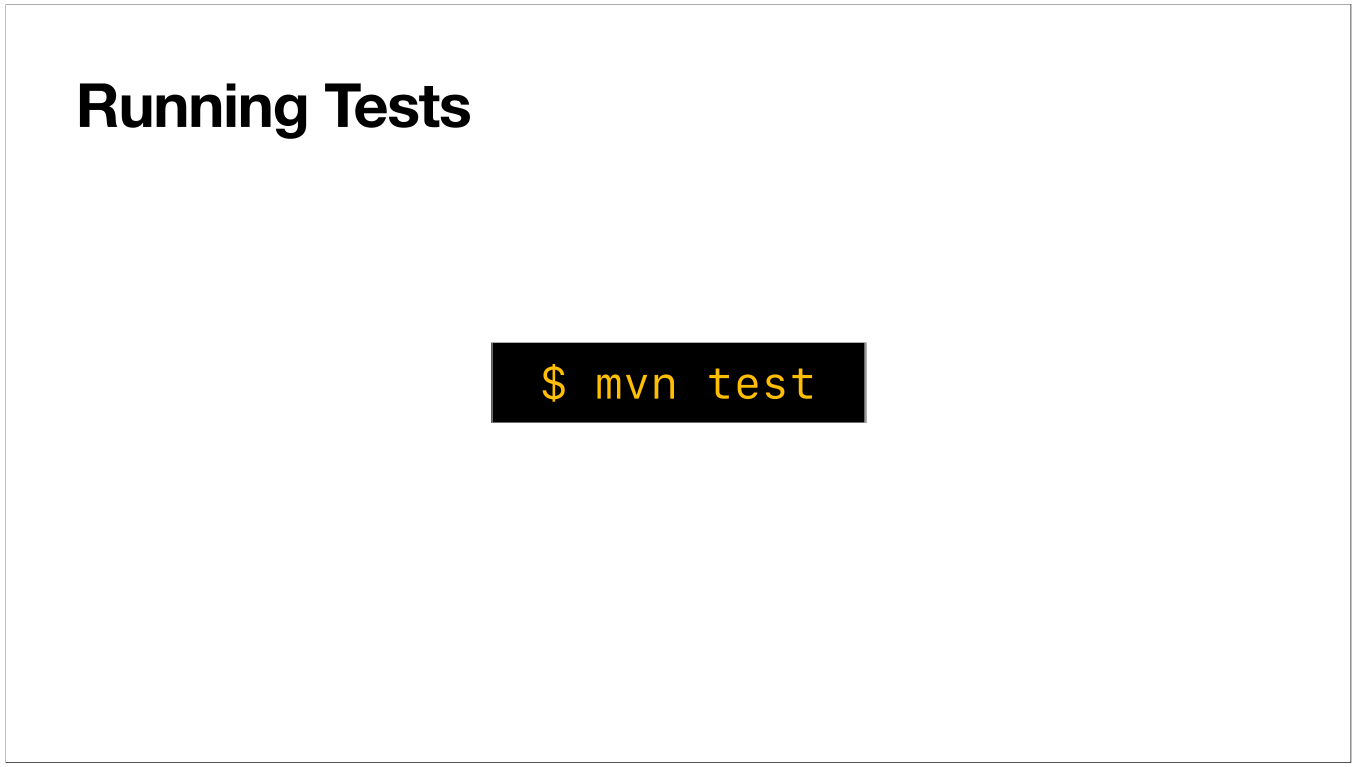
- **The Workflow test we wrote is an Integration Test!**
  - It invokes an Activity
  - If that Activity required external dependencies (API), that would have needed to be available
  - It's tightly coupled to both
- **Unit test Workflows by mocking Activities**
    - Define new replacement Activities
    - Use the Mockito package to create mocks



Testing Workflows

```java
package ageestimationworkflow;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;

import io.temporal.testing.TestWorkflowEnvironment;
import io.temporal.testing.TestWorkflowExtension;
import io.temporal.worker.Worker;

import static org.mockito.Mockito.*;


public class AgeEstimationWorkflowMockTest {

  @RegisterExtension
  public static final TestWorkflowExtension testWorkflowExtension = TestWorkflowExtension.newBuilder()
      .setWorkflowTypes(AgeEstimationWorkflowImpl.class)
      .setDoNotStart(true)
      .build();

  @Test
  public void testSuccessfulAgeEstimation(TestWorkflowEnvironment testEnv, Worker worker, AgeEstimationWorkflow workflow) {

    AgeEstimationActivities mockedActivities = mock(AgeEstimationActivities.class, withSettings().withoutAnnotations());
    when(mockedActivities.retrieveEstimate("Stanislav")).thenReturn(68);

    worker.registerActivitiesImplementations(mockedActivities);
    testEnv.start();

    String result = workflow.estimateAge("Stanislav");

    assertEquals("Stanislav has an estimated age of 68", result);
  }
}
```

The TestWorkflowExtension is a helper class to handle the spin up and tear down of the TestWorkflowEnvironment and Worker



Running Tests

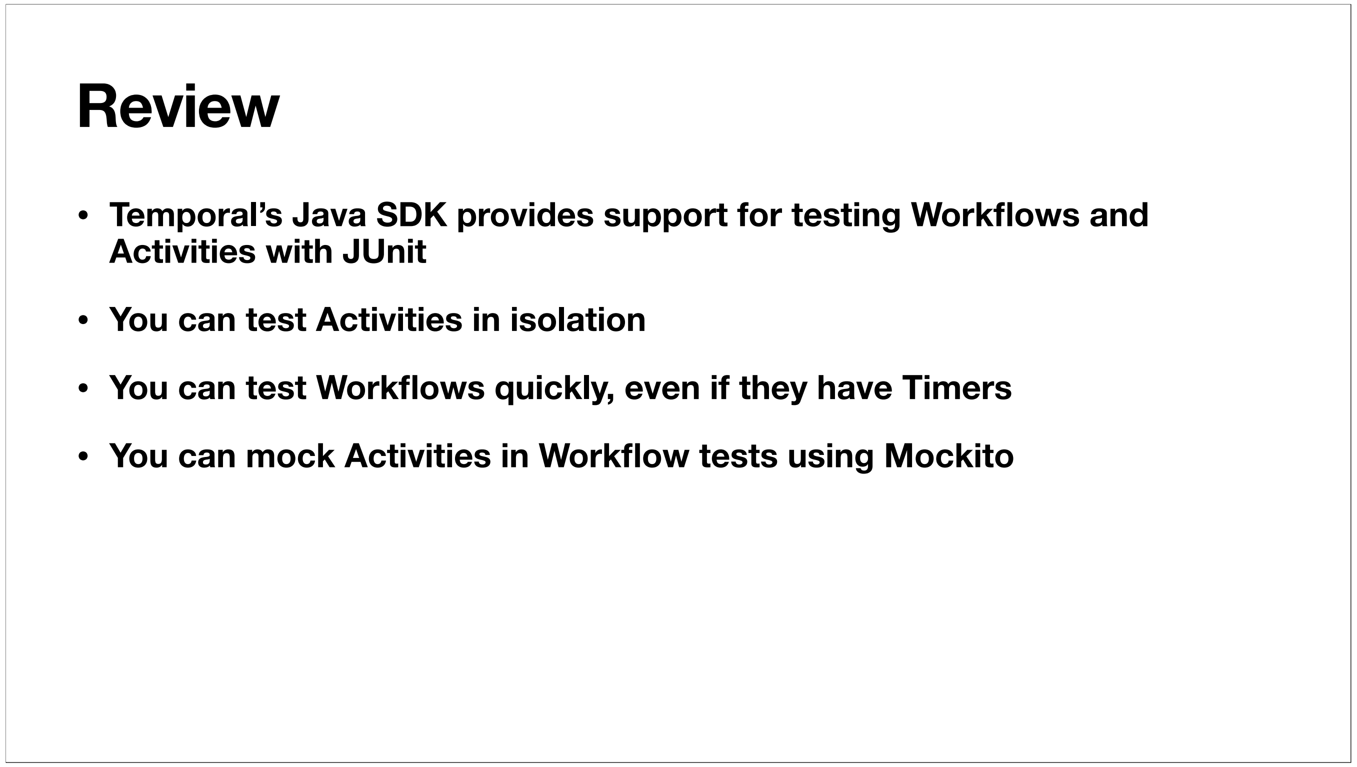
```
$ mvn test
```

Run your tests with your dependency manager, in this case "mvn test"

Exercise #3: 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

Exercise #3: Testing the Translation Workflow SOLUTION



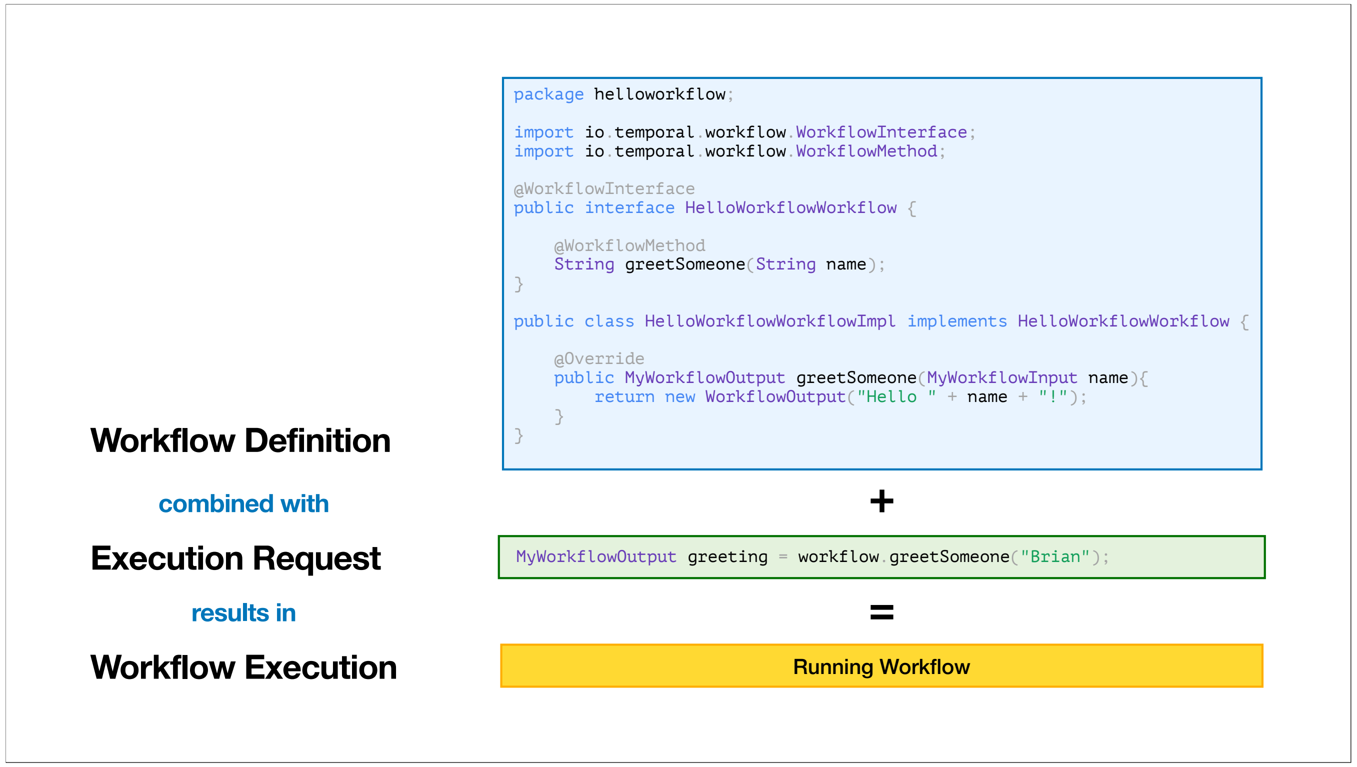
Review

- Temporal’s Java SDK provides support for testing Workflows and Activities with JUnit
- You can test Activities in isolation
- You can test Workflows quickly, even if they have Timers
- You can mock Activities in Workflow tests using Mockito

Temporal’s Java SDK provides support for testing Workflows and Activities with JUnit
You can test Activities in isolation
You can test Workflows quickly, even if they have Timers
You can mock Activities in Workflow tests using Mockito

Temporal 102

00. About this Workshop
01. Understanding Key Concepts in Temporal
02. Improving Your Temporal Application Code
03. Using Timers in a Workflow Definition
04. Testing Your Temporal Application Code
▶ **05. Understanding Event History**
06. Debugging Workflow Execution
07. Deploying Your Application to Production
08. Understanding Workflow Determinism
09. Conclusion



```
package helloworkflow;

import io.temporal.workflow.WorkflowInterface;
import io.temporal.workflow.WorkflowMethod;

@WorkflowInterface
public interface HelloWorkflowWorkflow {

    @WorkflowMethod
    String greetSomeone(String name);
}

public class HelloWorkflowWorkflowImpl implements HelloWorkflowWorkflow {

    @Override
    public MyWorkflowOutput greetSomeone(MyWorkflowInput name){
        return new WorkflowOutput("Hello " + name + "!");
    }
}
```

Workflow Definition

combined with

+

Execution Request

```
MyWorkflowOutput greeting = workflow.greetSomeone("Brian");
```

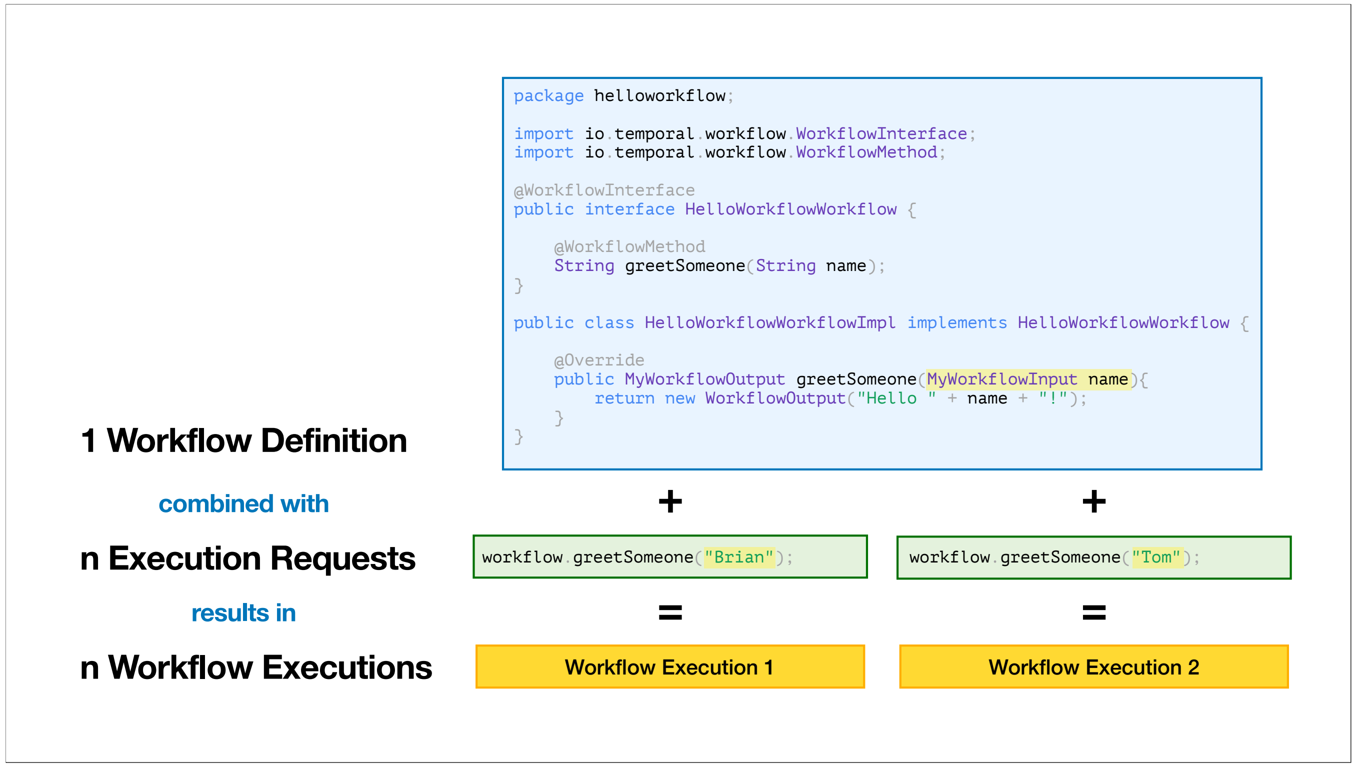
results in

=

Workflow Execution

Running Workflow

In Temporal, the code that defines your main business logic is implemented in a function referred to as a Workflow Definition. As with any other code you write, it doesn't actually do anything until you execute it. You do this by using a Client to initiate an execution request, which you could do with the command-line tool or by using code like what you see here. Either approach results in the same thing: a running Workflow. In Temporal, we refer to this as a Workflow Execution.



```
package helloworkflow;

import io.temporal.workflow.WorkflowInterface;
import io.temporal.workflow.WorkflowMethod;

@WorkflowInterface
public interface HelloWorkflowWorkflow {

    @WorkflowMethod
    String greetSomeone(String name);
}

public class HelloWorkflowWorkflowImpl implements HelloWorkflowWorkflow {

    @Override
    public MyWorkflowOutput greetSomeone(MyWorkflowInput name){
        return new WorkflowOutput("Hello " + name + "!");
    }
}
```

1 Workflow Definition

combined with

n Execution Requests

```
workflow.greetSomeone("Brian");
```

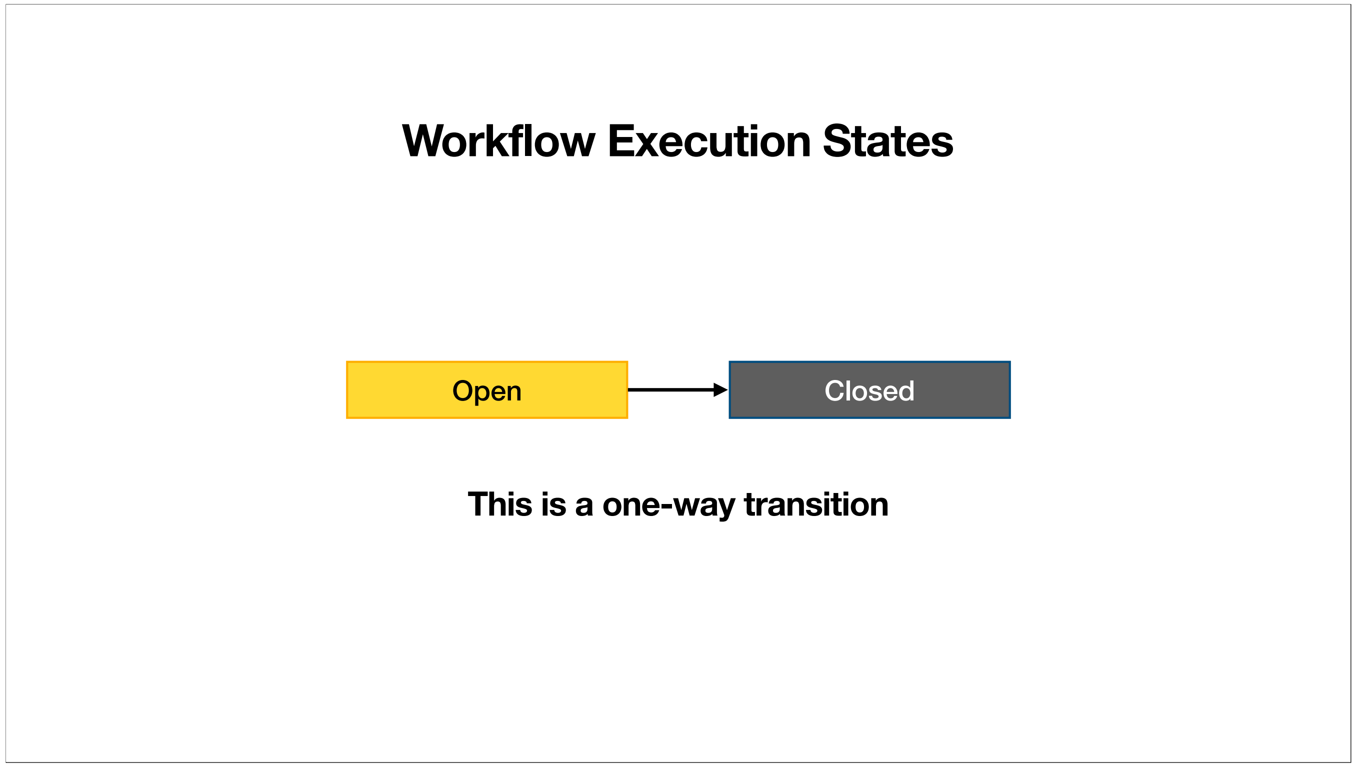
```
workflow.greetSomeone("Tom");
```

results in

n Workflow Executions

A single Workflow Definition can be executed any number of times. Each results in a new Workflow Execution. For example, you might run the same Workflow Definition each morning to generate some type of daily report.

Also notice that while the type of input is specified in the Workflow Definition, the value of that input is supplied in the execution request. It's very common to run the same Workflow Definition multiple times, with each execution having a different input. For example, I might start the same Workflow Definition five thousand times in a row, supplying a different customer ID as input each time, in order to send out monthly statements to each customer.



Workflow Execution States

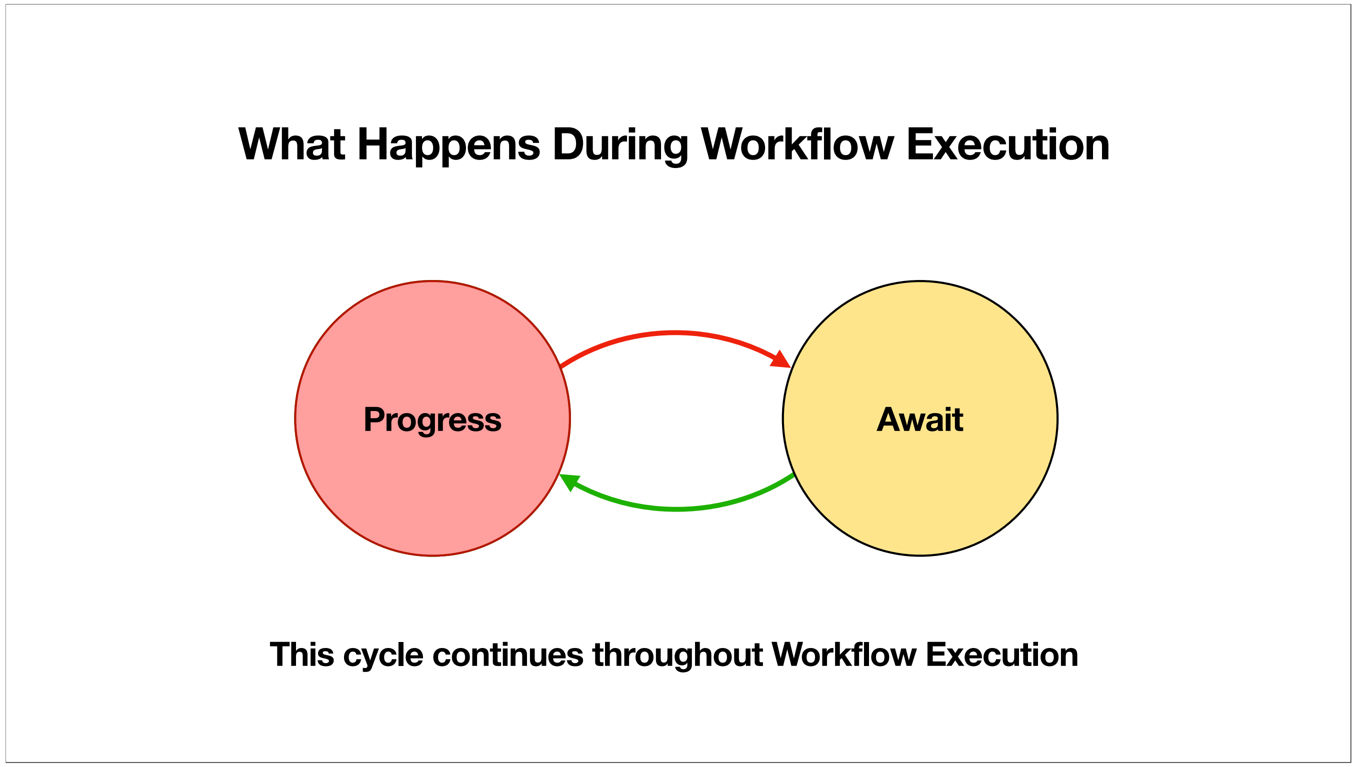
This is a one-way transition

A Workflow Execution has two possible states: open or closed. A Workflow Execution is one that is currently running, while a closed Workflow Execution is one that has stopped running for one reason or another, perhaps because it completed successfully, failed, or was terminated.

The state of a Workflow Execution can, and eventually will, change from open to closed, but this is a one-way transition. Once a Workflow Execution enters the closed state, it remains there.

Also notice that while the type of input is specified in the Workflow Definition, the value of that input is supplied in the execution request. It's very common to run the same Workflow Definition multiple times, with each execution having a different input. For example, I might start the same Workflow Definition five thousand times in a row, supplying a different customer ID as input each time, in order to send out monthly statements to each customer.

Also notice that while the type of input is specified in the Workflow Definition, the value of that input is supplied in the execution request. It's very common to run the same Workflow Definition multiple times, with each execution having a different input. For example, I might start the same Workflow Definition five thousand times in a row, supplying a different customer ID as input each time, in order to send out monthly statements to each customer.



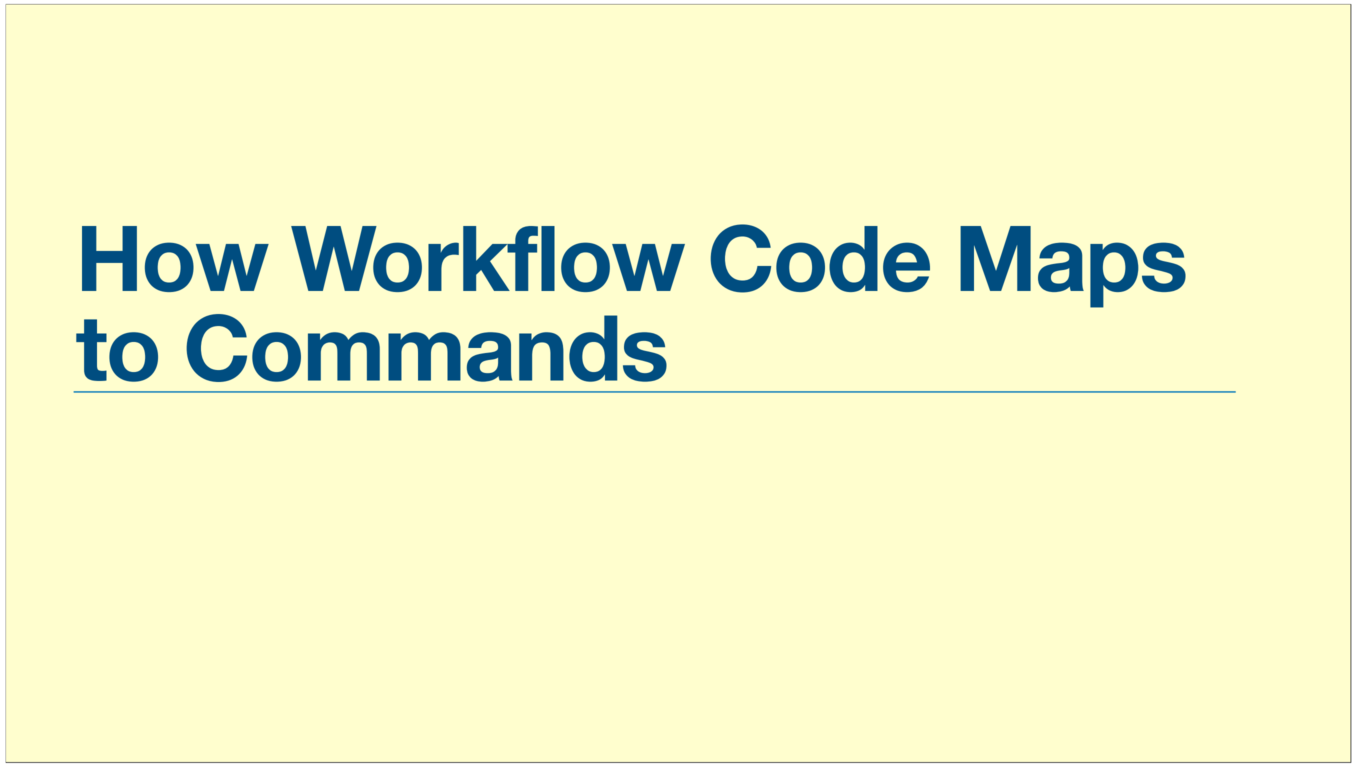
What Happens During Workflow Execution

This cycle continues throughout Workflow Execution

While in the open state, the Workflow is essentially doing one of two things. It's either actively making progress or it's awaiting something that's required for progress to continue.

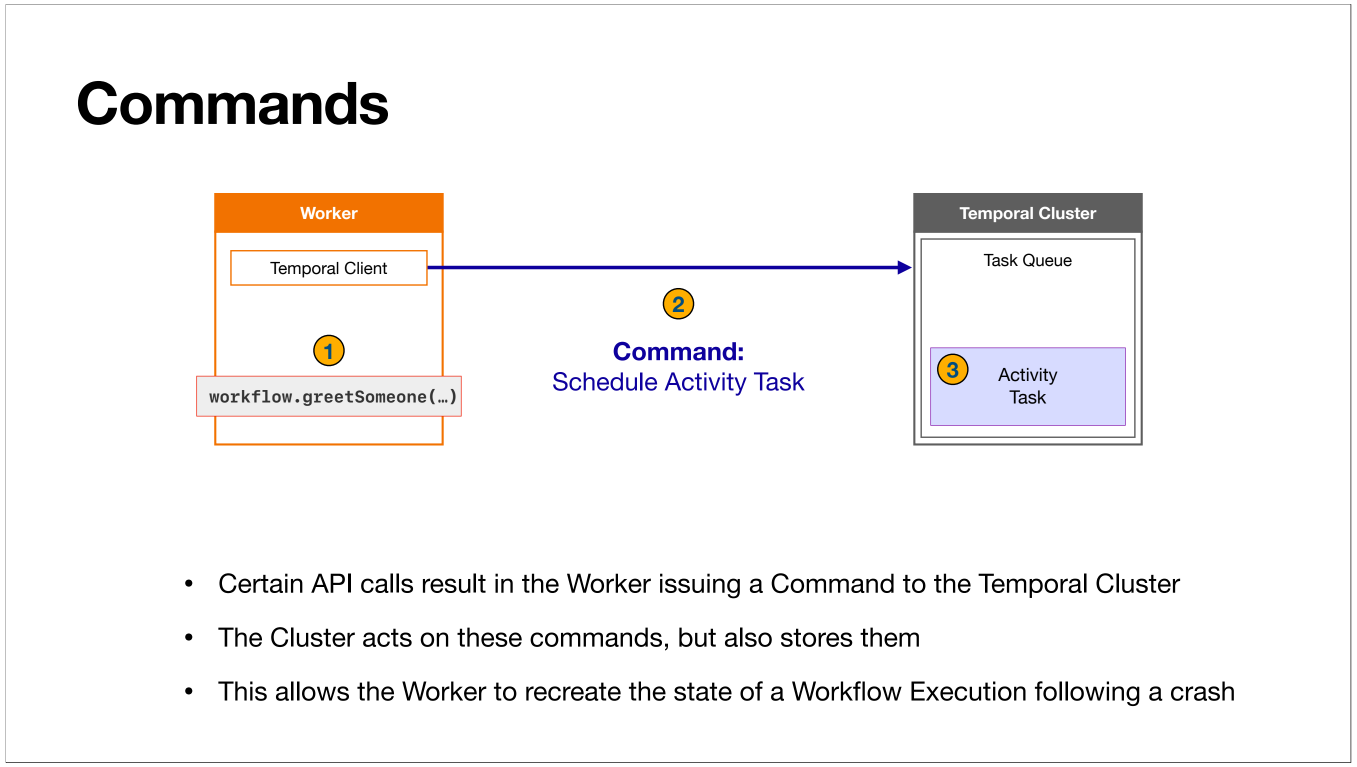
One example of something that a Workflow would await is Activity Execution. If the Workflow code uses the value returned by an Activity, then it must await the execution of that Activity for the value to become available. Another example would be a Timer. If the Workflow is blocked waiting on a Timer, then it cannot make progress until either that Timer fires or that Timer is canceled.

This cycle of progress and waiting continues throughout the Workflow Execution, only stopping when it enters the closed state.



How Workflow Code Maps to Commands

Now let’s look at how the Workflow code you write maps to commands that get sent to the Temporal Cluster.



Commands

Worker

Temporal Client

1

`workflow.greetSomeone(…)`

2

Command:
Schedule Activity Task

Temporal Cluster

Task Queue

3

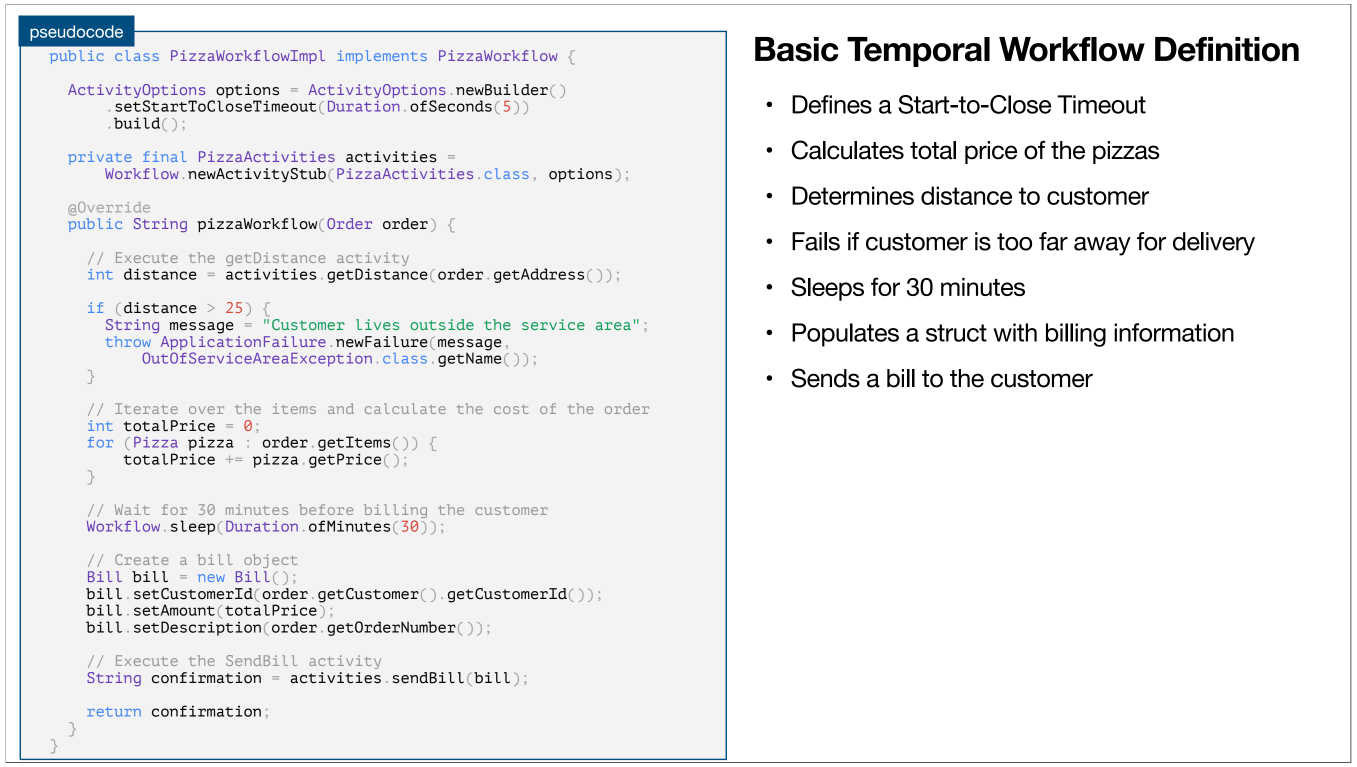
Activity
Task

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

Before jumping into a detailed demonstration, let’s review Commands.

When the Worker encounters certain API calls during Workflow Execution, such as a call to execute an Activity, it sends a Command to the Temporal Cluster. The cluster acts on these Commands, for example, by creating an Activity Task, but also stores them in case of failure. For example, if the Worker crashes, the Temporal cluster uses this information to recreate the state of the Workflow to what it was immediately before the crash and then resumes progress from that point. This allows you, as a developer, to code as if this type of failure wasn't even a possibility.

--



pseudocode

```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Basic Temporal Workflow Definition

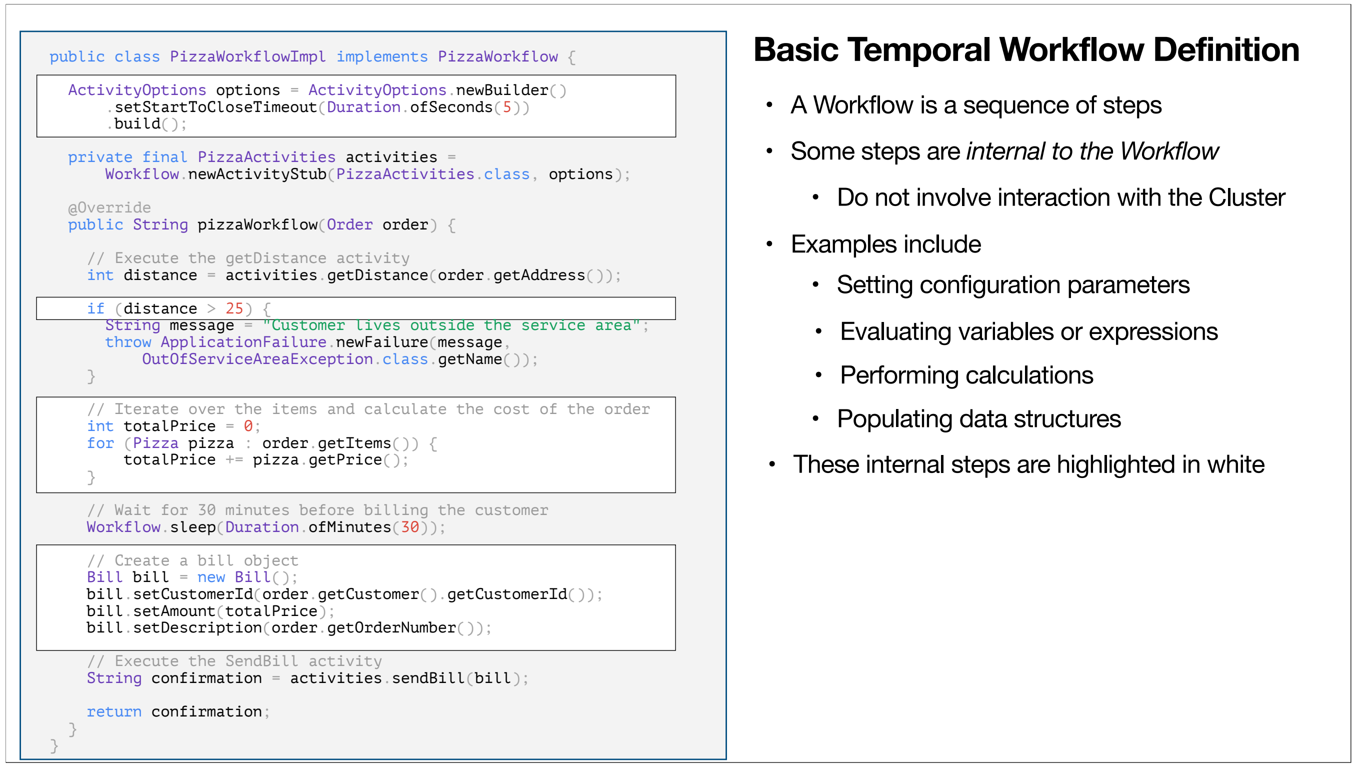
- Defines a Start-to-Close Timeout
- Calculates total price of the pizzas
- Determines distance to customer
- Fails if customer is too far away for delivery
- Sleeps for 30 minutes
- Populates a struct with billing information
- Sends a bill to the customer

Here is code for a basic Temporal Workflow Definition. It's actually pseudocode, because I simplified some aspects of the syntax and implementation to make it easier to follow and to better fit the limited space on the screen. For example, I have removed some error-handling code, logging statements, and am using a string as a return value rather then a struct, which would better reflect the recommended best practice. Although such details are important for production code, they distract from the points I want to make here. However, you can find a working example of the *actual* code in the code repository for this course.

I'll give a quick overview of what it does before explaining it in greater detail. It simulates the processing of a very simplistic pizza order. As with any Workflow that executes one or more Activities, it begins with a few lines that define the parameters of their execution; in this case, it sets a Start-to-Close Timeout of five seconds.

Next, it requests execution of an Activity that that returns the distance to the customer's location. If determined to be more than 25 kilometers away, it returns an error, failing the Workflow, because our business logic dictates that this is outside the service area.

It then goes on to iterate over the pizzas that make up this order, adding up the price of each one to calculate the total cost of the order. It then sleeps for 30 minutes, allowing time for the order to be cooked and delivered, and then requests execution of an Activity that will bill the customer.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

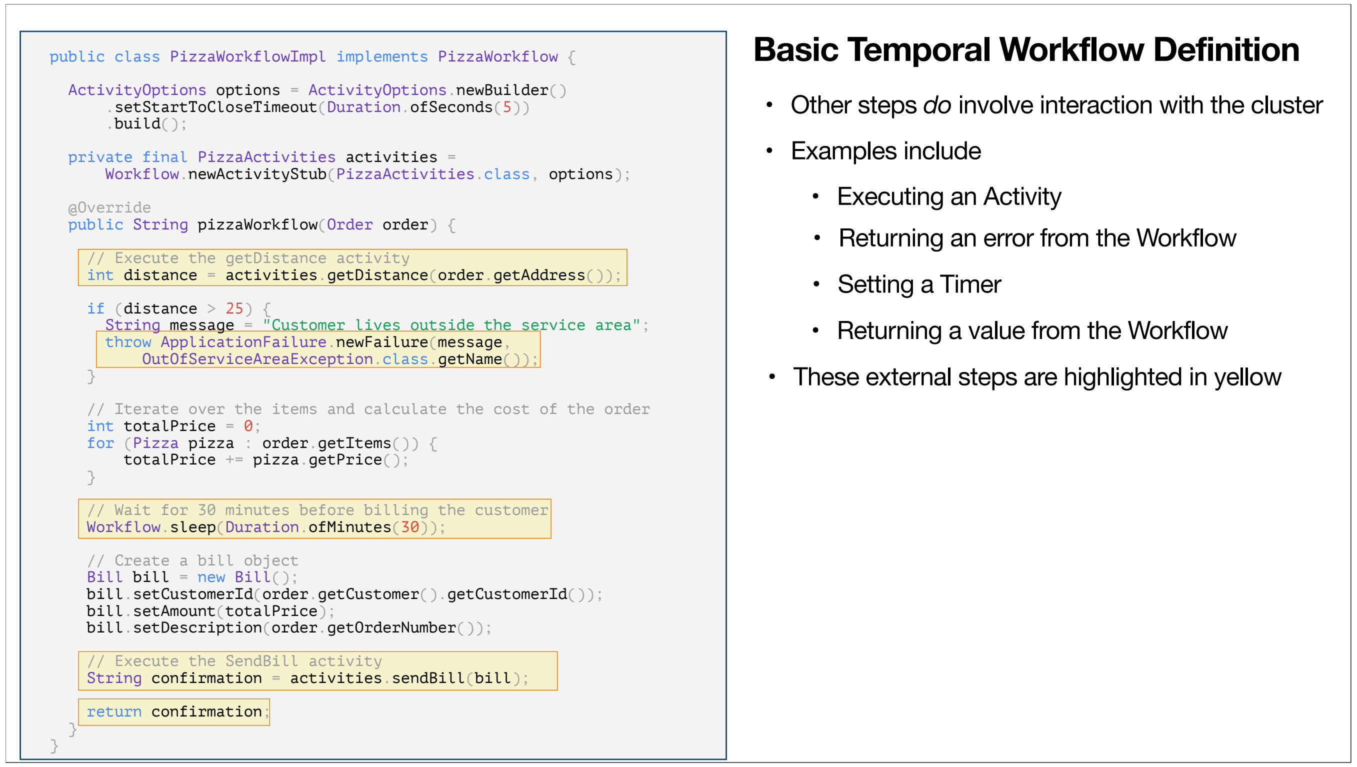
Basic Temporal Workflow Definition

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

A Workflow is a sequence of steps.

Some steps are internal and don't involve the cluster.

[advance]



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Basic Temporal Workflow Definition

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

In contrast, other steps within the Workflow do result in interaction with the Temporal Cluster. I'll highlight those in yellow as I step through the code.

[advance]

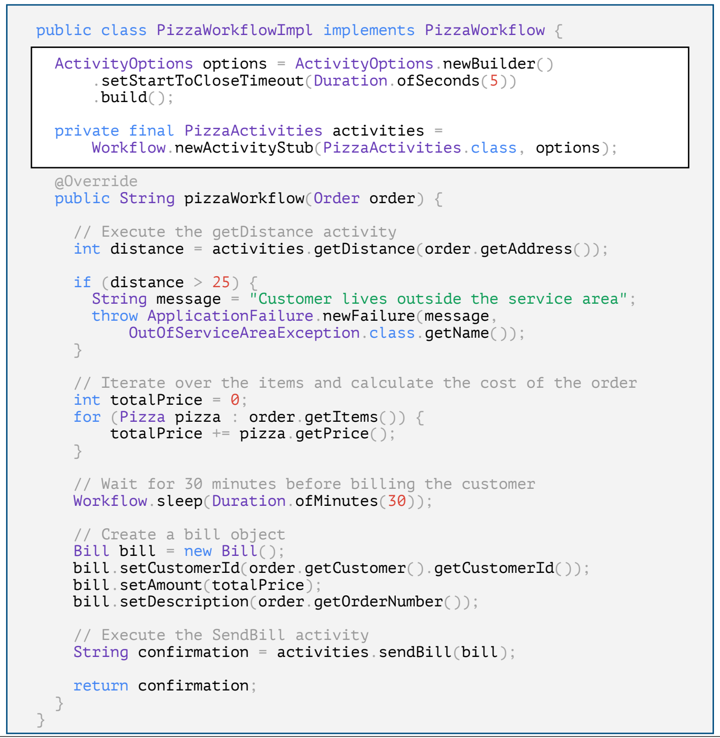
For example, requesting execution of an Activity results in the Temporal Cluster creating an Activity Task and adding it to a Task Queue.

[advance]

The `sleep` call here is another example, as it results in the Temporal Cluster starting a Timer with a duration of 30 minutes. Returning an error from the Workflow function causes the Temporal Cluster to mark the Workflow as failed,

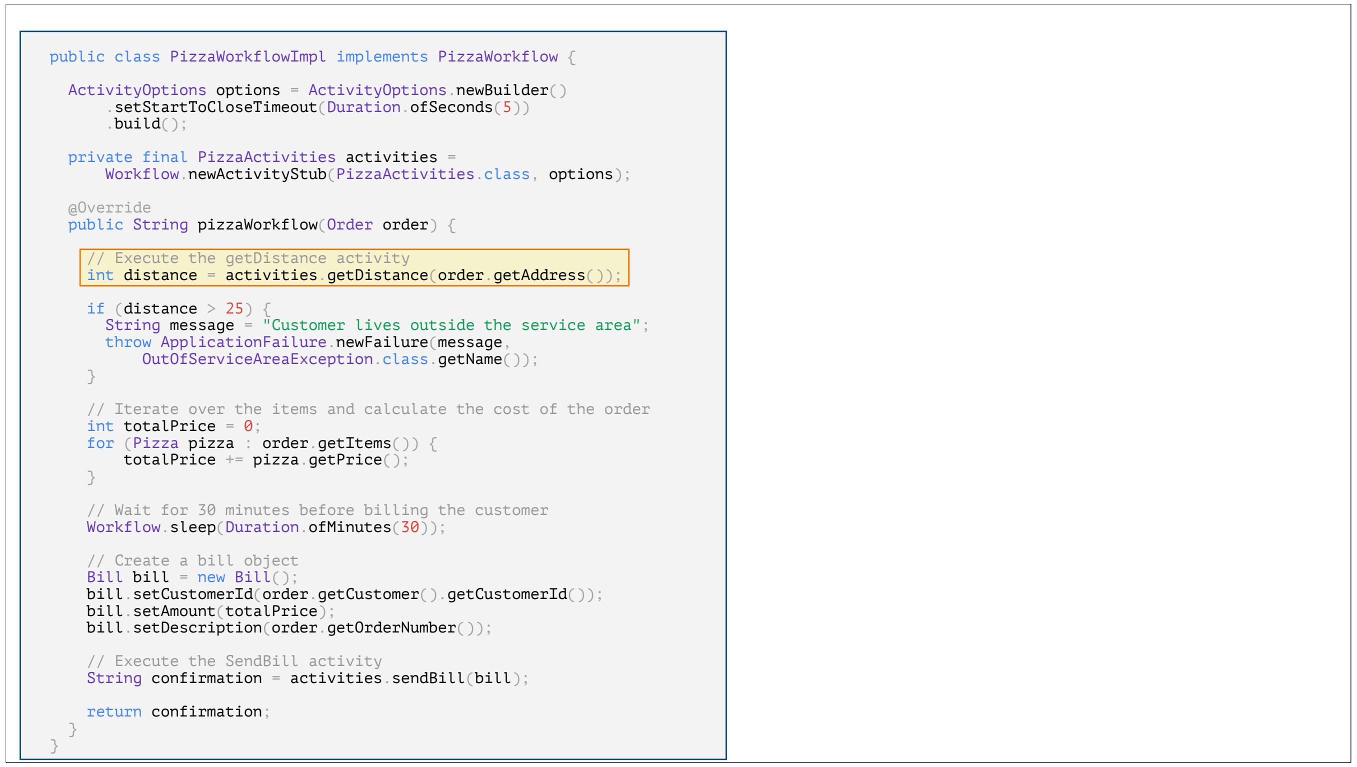
[advance]

while returning without an error causes the Temporal Cluster to mark the Workflow as completed.



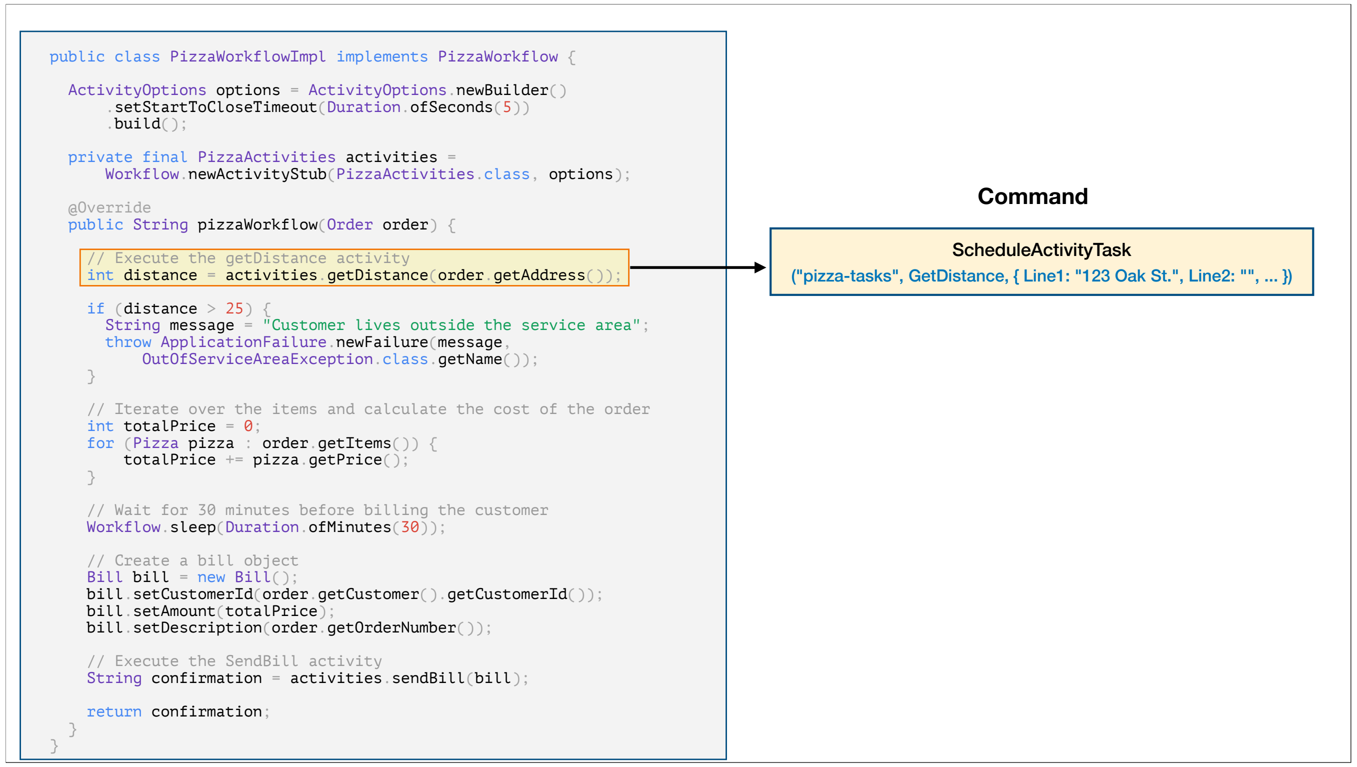
```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

In this Workflow Definition, the first several statements are internal to the Workflow. That is, they don't require any interaction with the Temporal Cluster. Their runtime behavior is the same as it would be in any other program.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

As execution continues, the Worker reaches a statement that does require interaction with the Temporal Cluster. In this case, it is a request to execute an Activity.



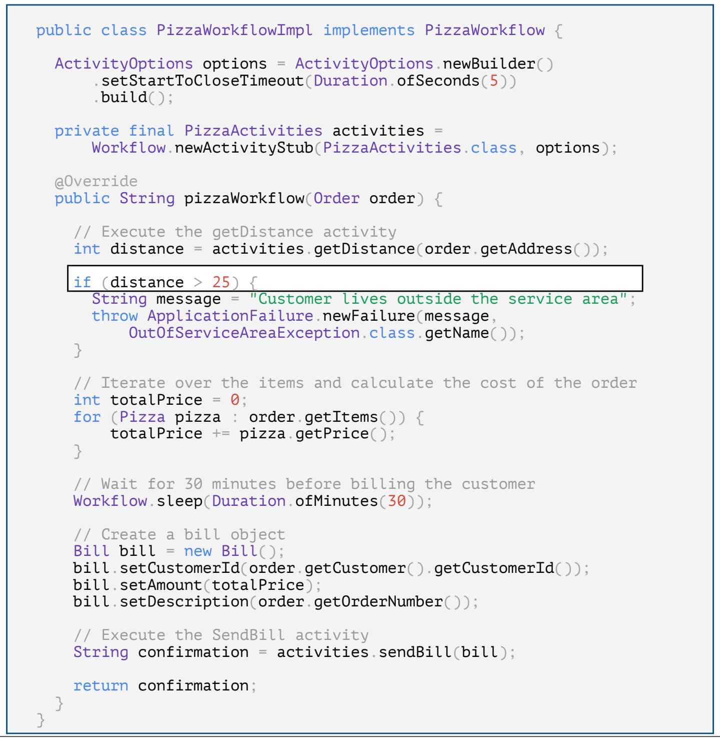
```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Command

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

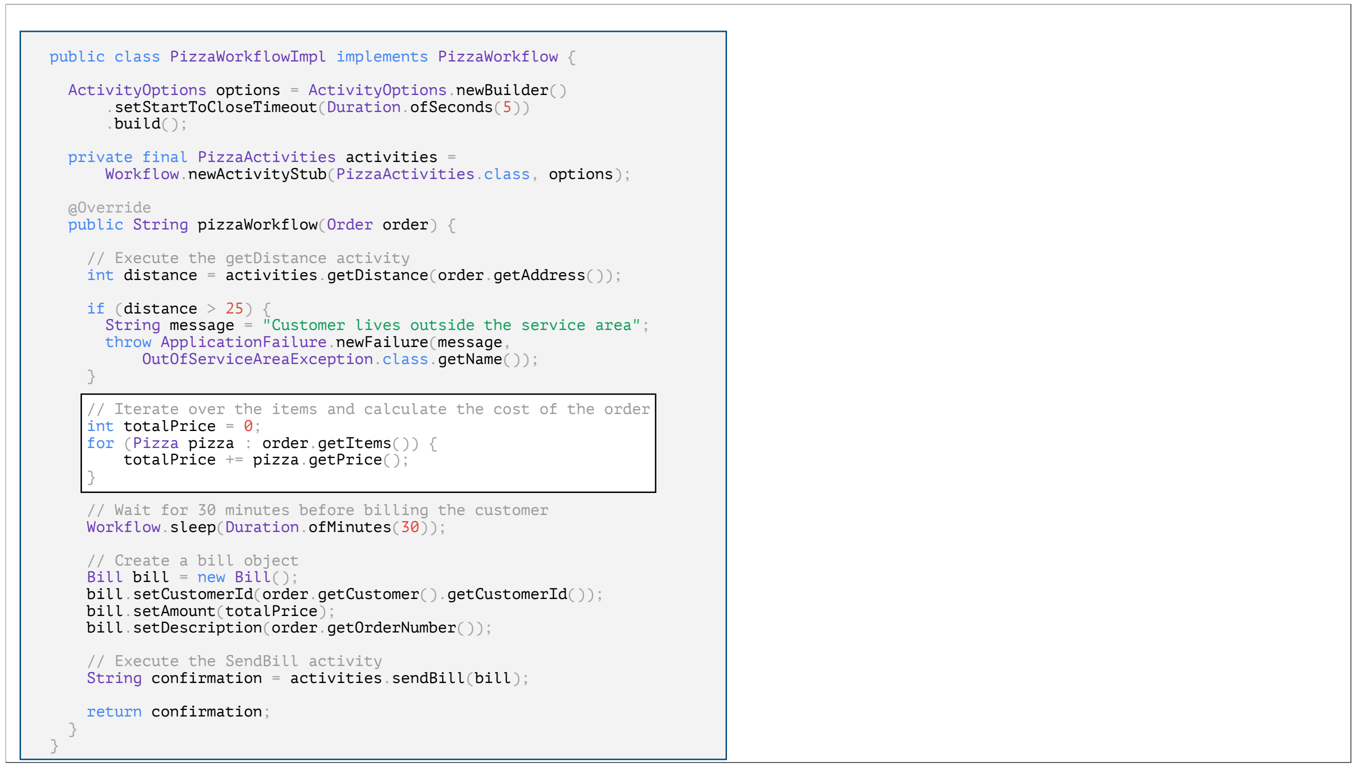
This causes the Worker to issue a Command to the Temporal Cluster, which requests the desired result and provides the details required to achieve it. For example, the `ScheduleActivityTask` Command contains details such as the Task Queue name, the Activity Type, and input parameter values. This Command is what initiates the scheduling of an Activity Task and the resulting execution of the code in the corresponding Activity Definition. I'll cover that in more detail in a moment, but I'd like to show a few more examples first.

Since the Workflow code is retrieving a result from the Activity Execution, it blocks until the Activity function returns.



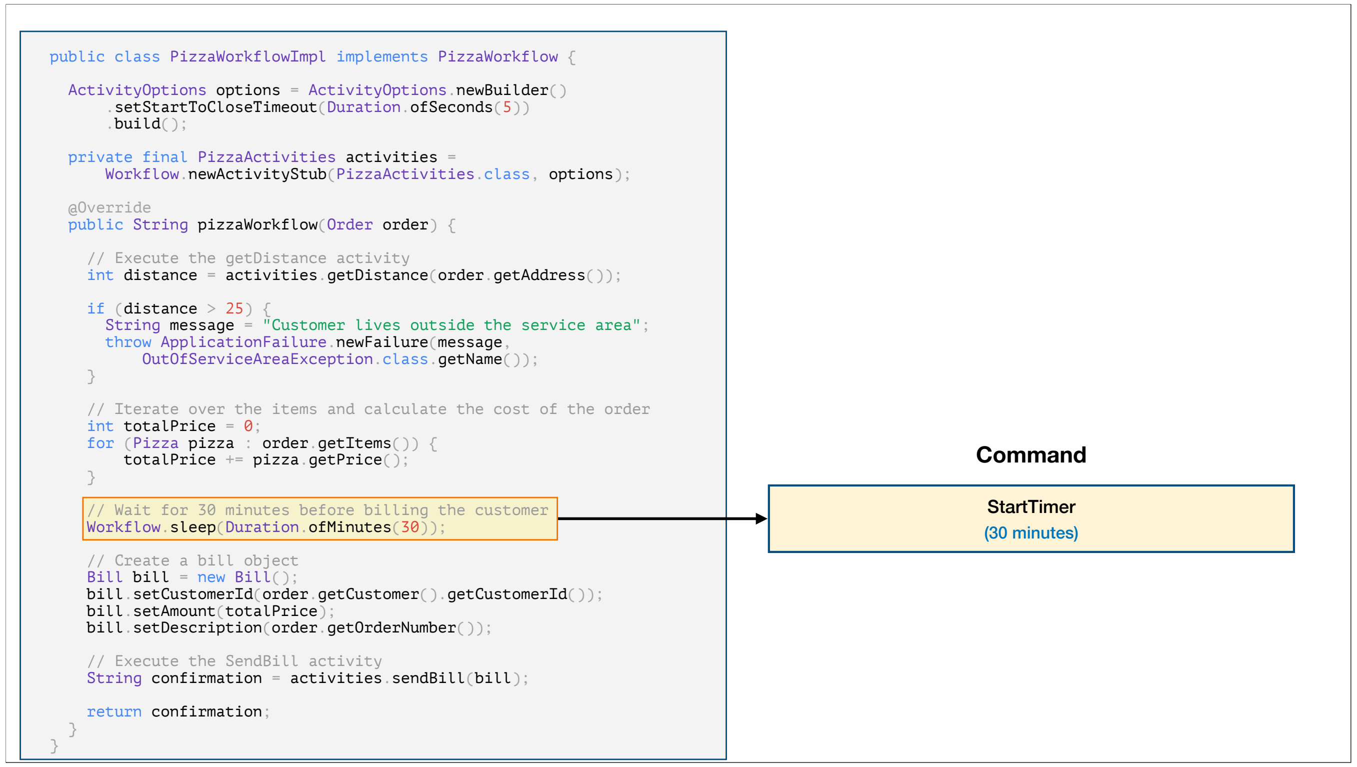
```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Afterwards, the Worker continues executing the Workflow code. The next few lines, highlighted here, evaluate a variable. Depending on the outcome, it may return an error, which would cause the Workflow to fail. However, let's be optimistic in this case and assume that this is a delivery for a nearby customer. The execution will continue.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

The next few lines iterate over the items in the order and calculate the total order price. This is another place where there’s no interaction with the Temporal Server. The execution happens locally.



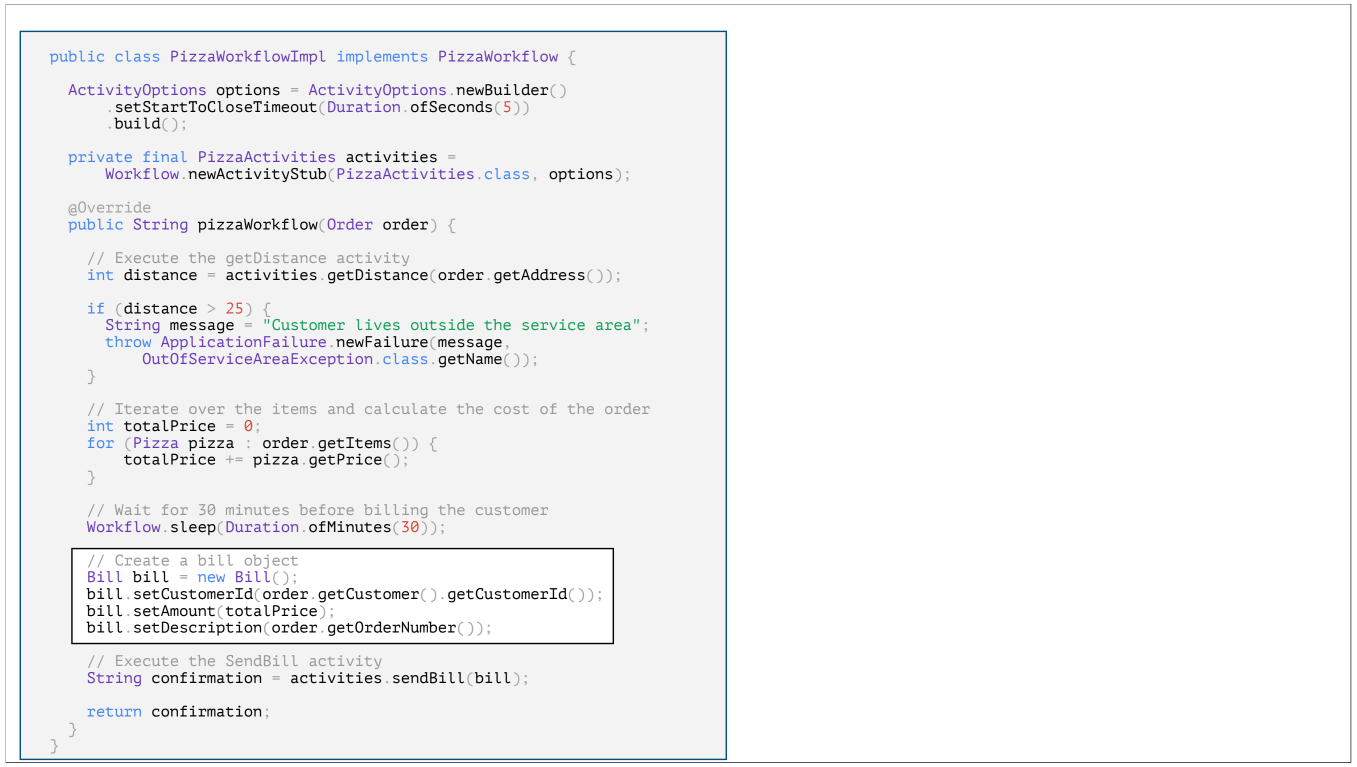
```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Command

StartTimer
(30 minutes)

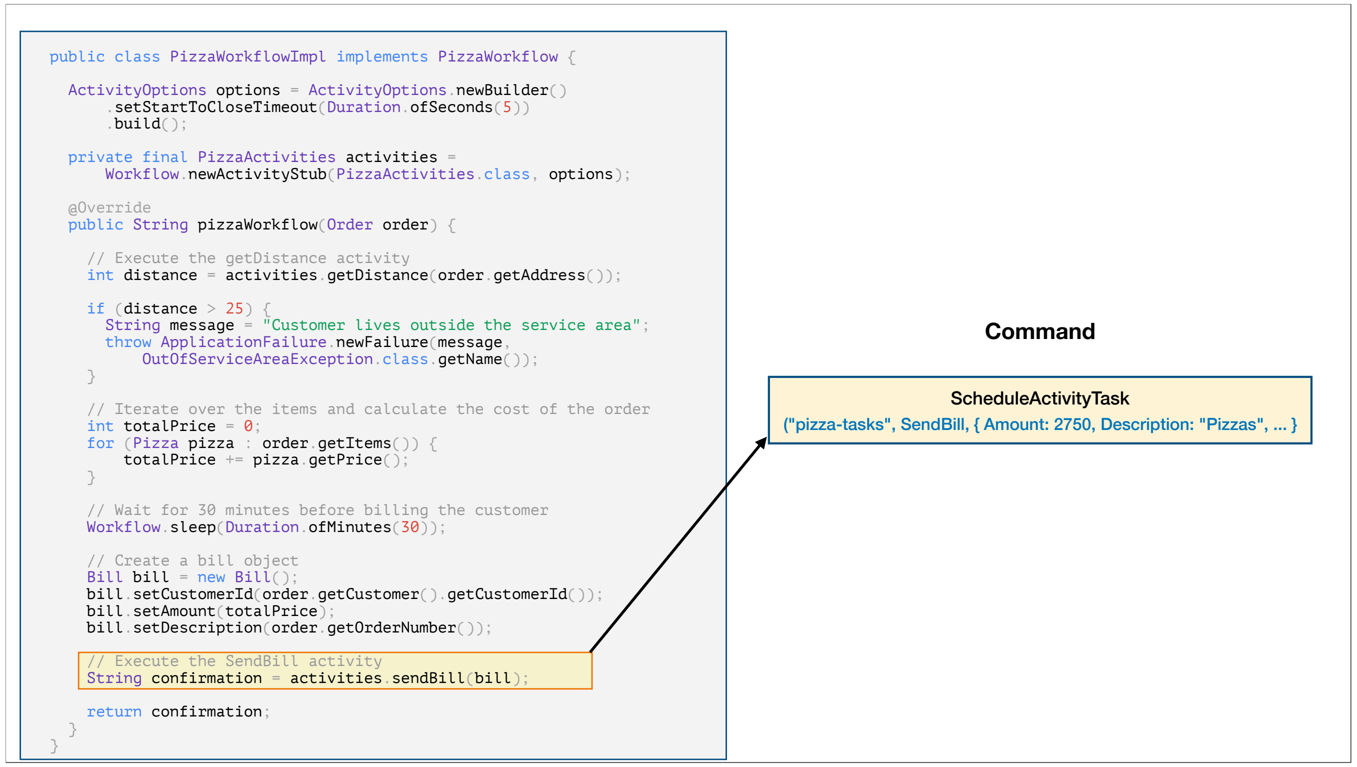
It now reaches the `Workflow.sleep` call, which is another statement that involves interaction with the Temporal Cluster. This causes the Worker to issue another Command, one which requests that it start a Timer. The duration is one of the details specified in this Command.

Further execution of this Workflow will now pause for 30 minutes until the Timer fires.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

The next few lines, highlighted here, create and populate a class that represents the input for the next Activity. While it is related to the Activity, it doesn't involve any interaction with the cluster.



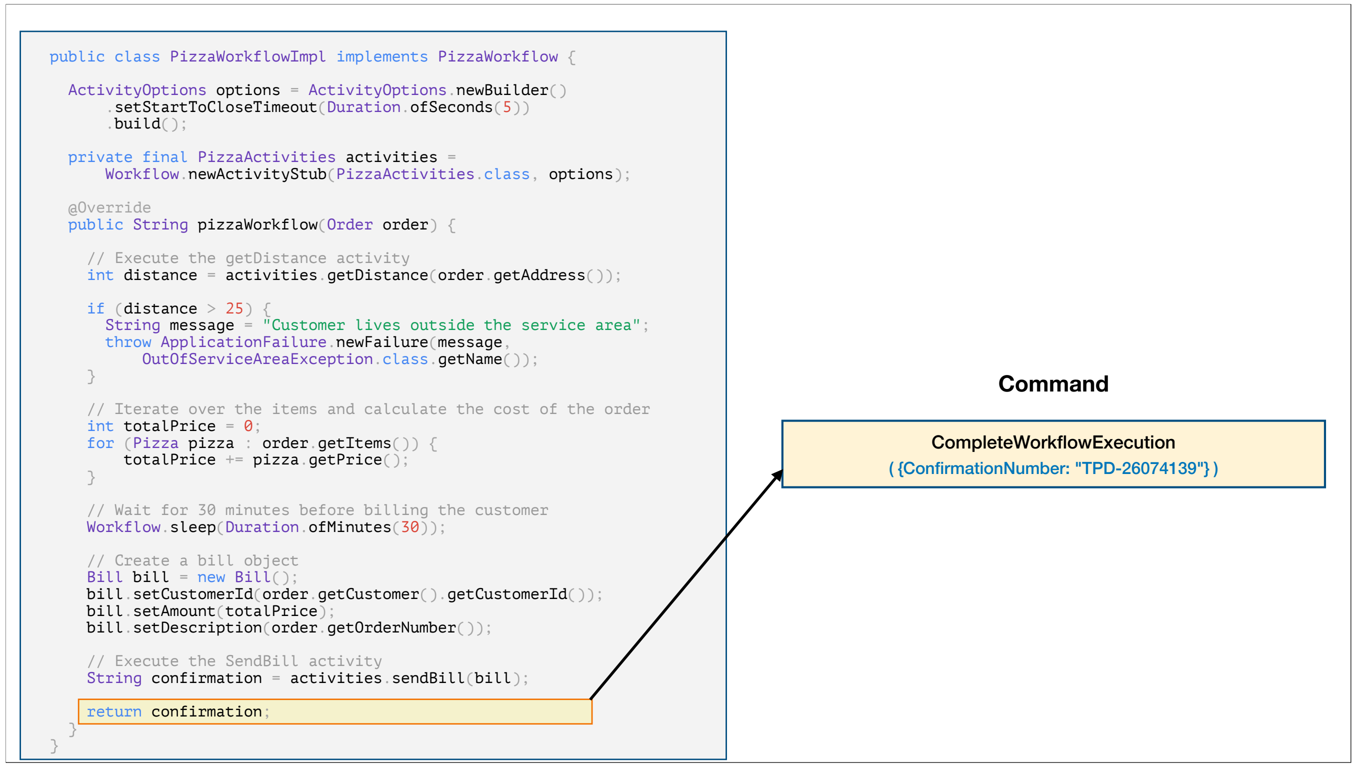
```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Command

ScheduleActivityTask

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

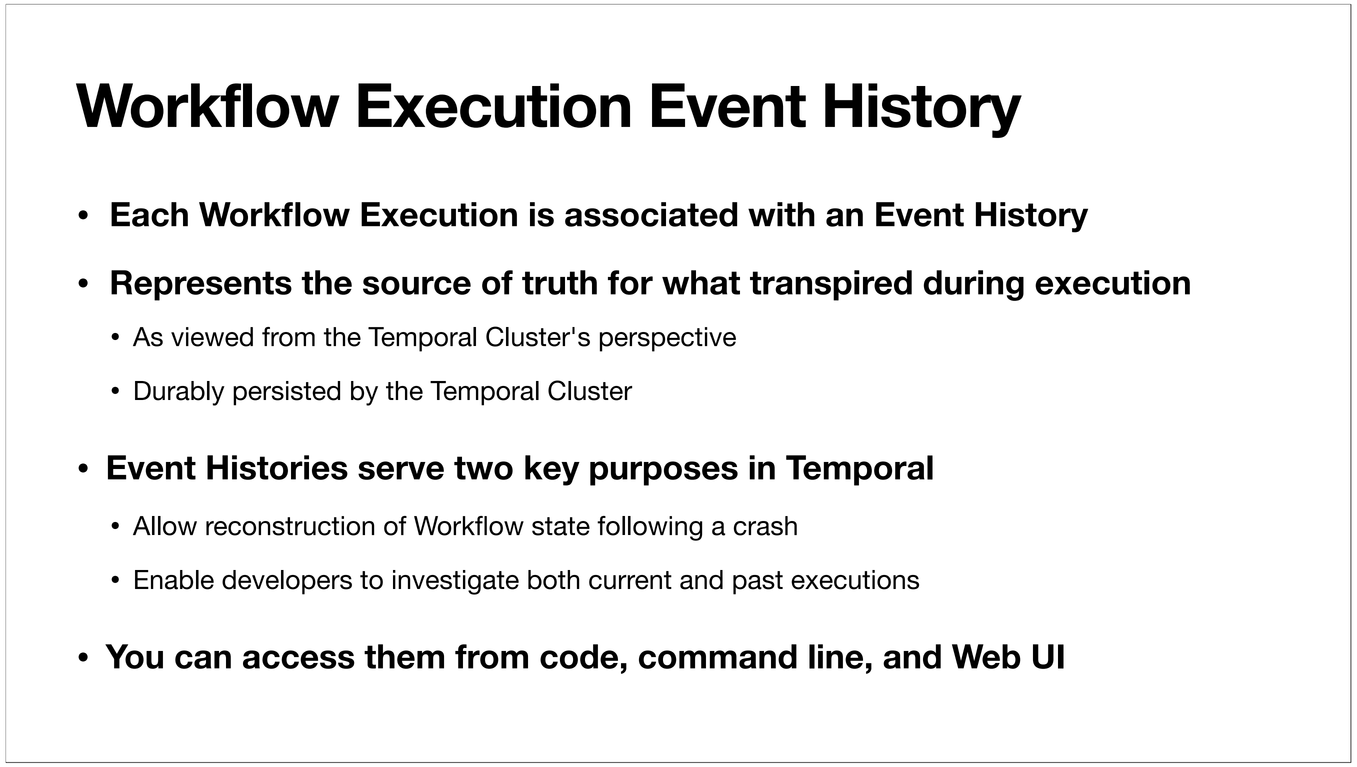
However, the next statement, does. It requests execution of an Activity, so the Worker issues another Command to the Temporal Cluster.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

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

Finally, returning from the Workflow function also results in a Command. In this case, we're returning a result and using `nil` for the error. The Worker identifies that as a successful completion of the Workflow Execution, so it issues a `CompleteWorkflowExecution` command to the Temporal Cluster, which includes the value we returned from the function.



Workflow Execution Event History

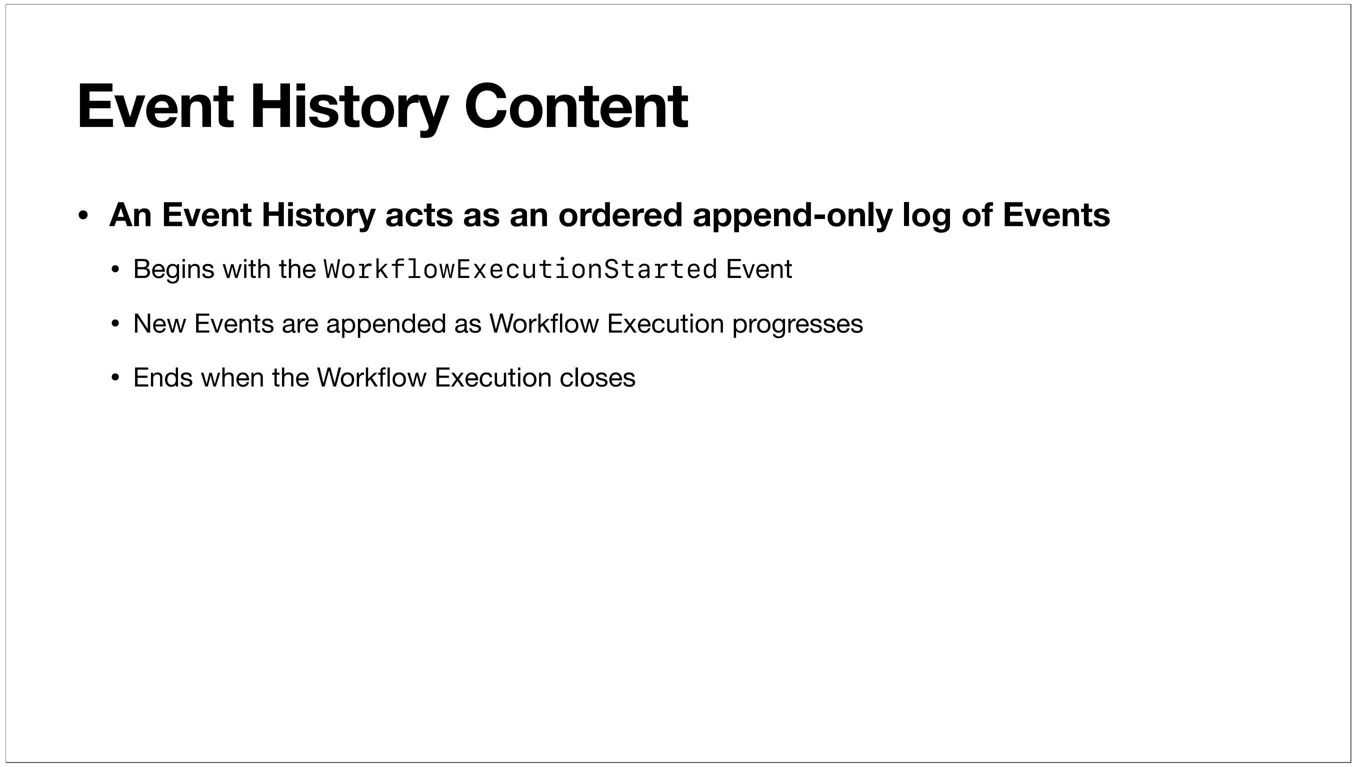
- **Each Workflow Execution is associated with an Event History**
- **Represents the source of truth for what transpired during execution**
  - As viewed from the Temporal Cluster's perspective
  - Durably persisted by the Temporal Cluster
- **Event Histories serve two key purposes in Temporal**
  - 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**

Each Workflow Execution is associated with an Event History

This history represents the source of truth for what happened during the execution.

Event histories allow reconstruction of Workflow state following a crash
They also enable developers to explore and investigate current and past executions.

You can access histories from code, command line, and the web ui.



Event History Content

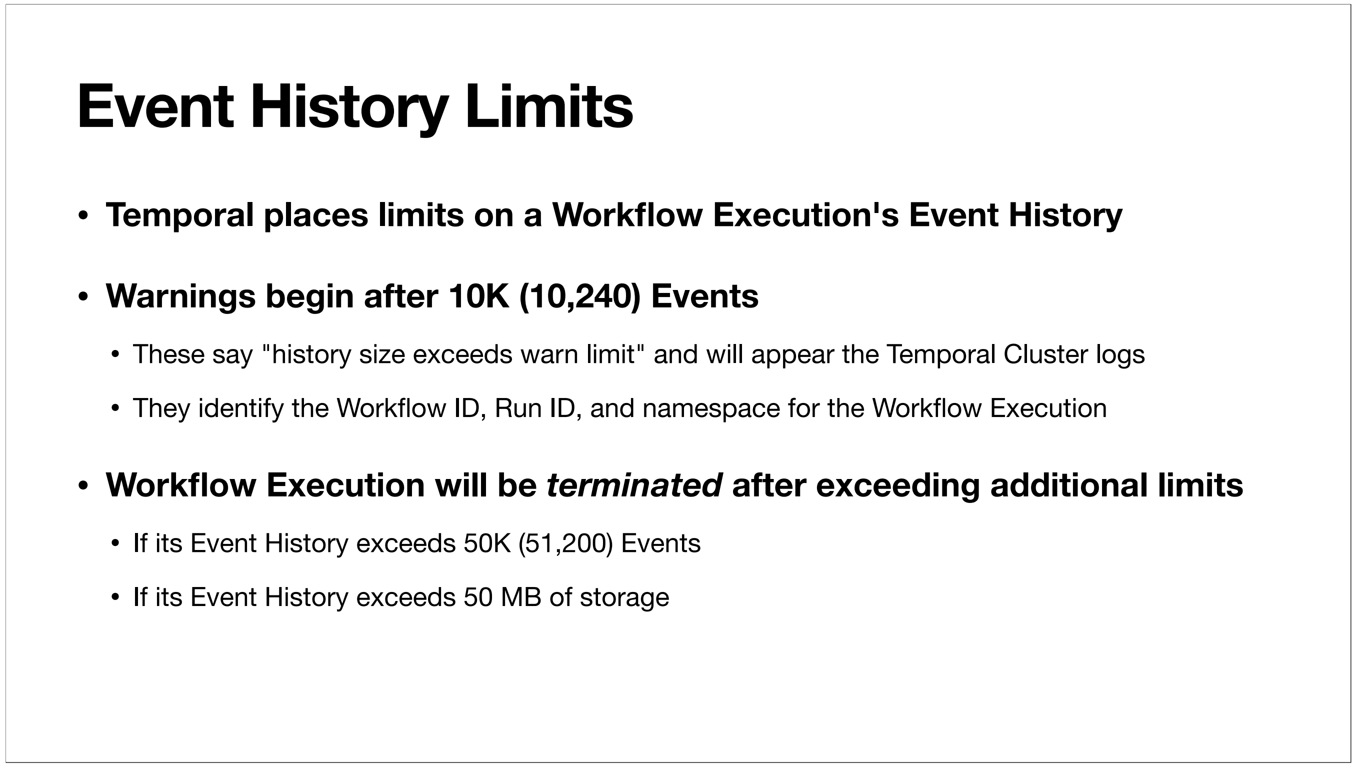
- **An Event History acts as an ordered append-only log of Events**
  - Begins with the `WorkflowExecutionStarted` Event
  - New Events are appended as Workflow Execution progresses
  - Ends when the Workflow Execution closes

An Event History acts as an ordered append-only log 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 Cluster 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

Temporal places limits on a Workflow Execution's Event History

Warnings begin after 10,240 Events

Workflow Execution will be terminated after exceeding additional limits

Event Structure and Characteristics

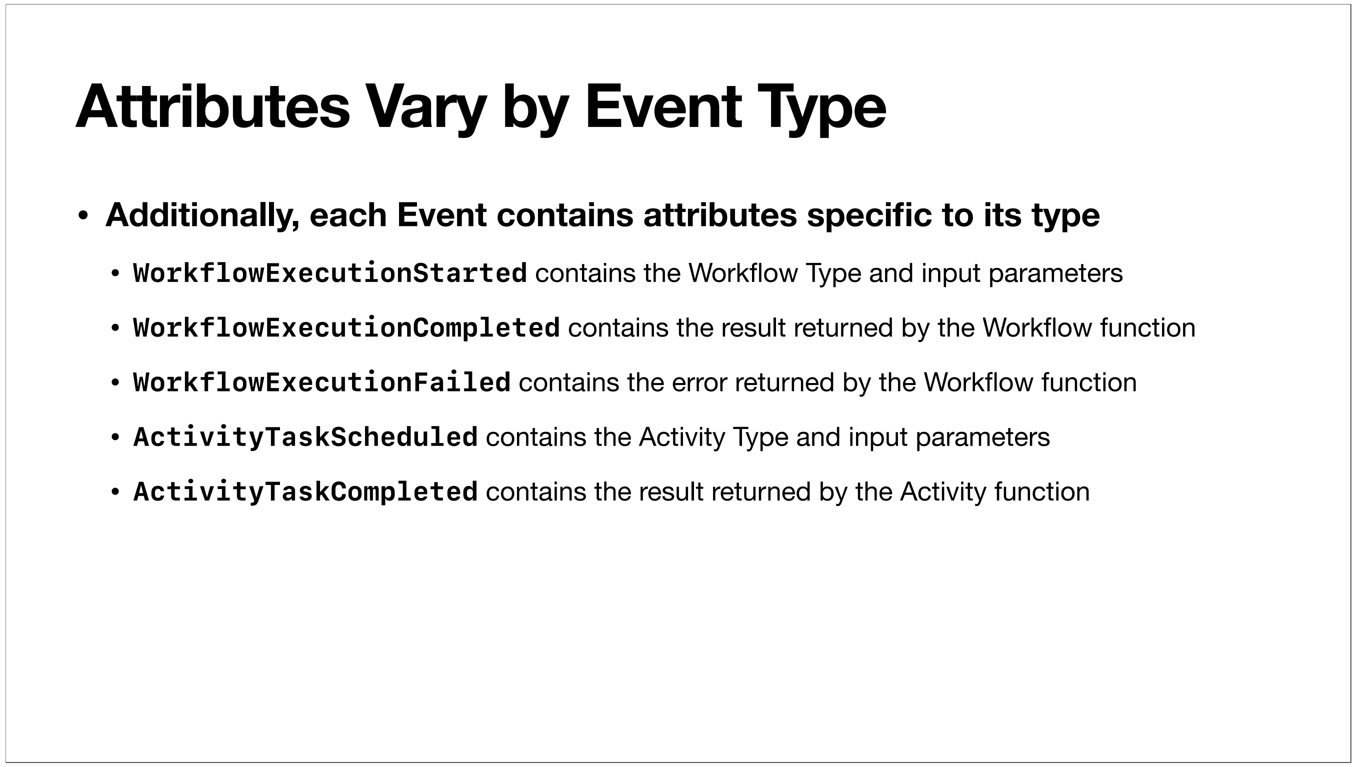
- **Every Event always contains the following three attributes**
  - ID (uniquely identifies this Event within the History)
  - Time (timestamp representing when the Event occurred)
  - Type (the kind of Event it is)

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)



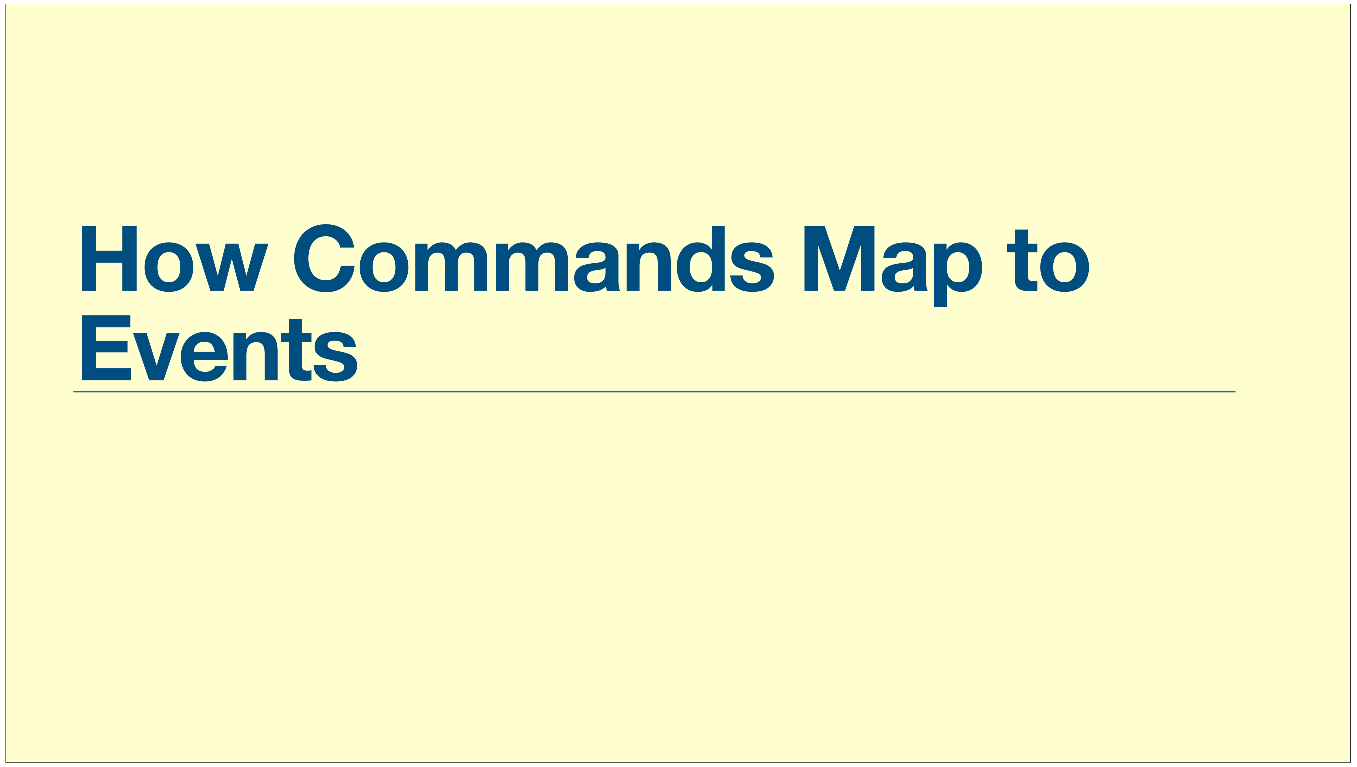
Attributes Vary by Event Type

- **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 function
  - **`WorkflowExecutionFailed`** contains the error returned by the Workflow function
  - **`ActivityTaskScheduled`** contains the Activity Type and input parameters
  - **`ActivityTaskCompleted`** contains the result returned by the Activity function

Attributes Vary by Event Type.

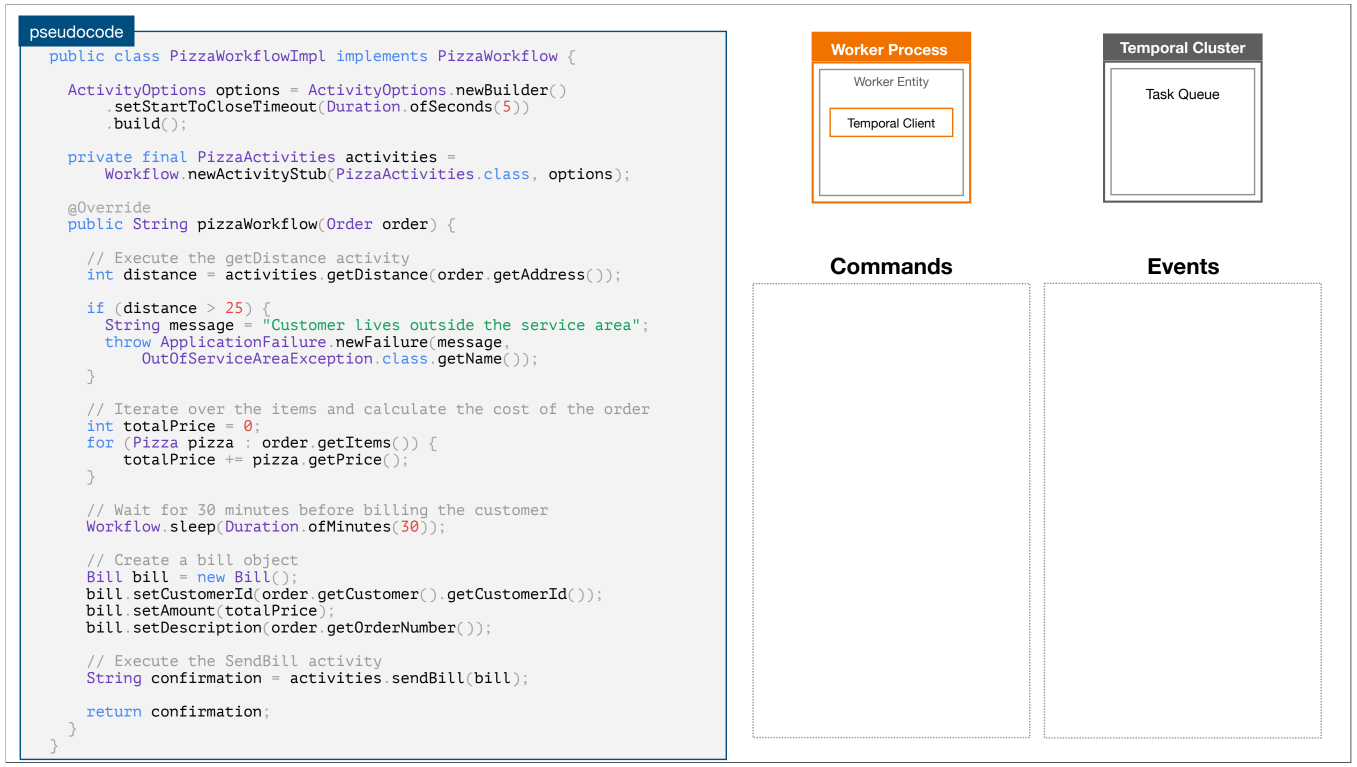
Additionally, each Event contains attributes specific to its type

WorkflowExecutionStarted contains the Workflow Type and input parameters
WorkflowExecutionCompleted contains the result returned by the Workflow function
WorkflowExecutionFailed contains the error returned by the Workflow function
ActivityTaskScheduled contains the Activity Type and input parameters
ActivityTaskCompleted contains the result returned by the Activity function



How Commands Map to Events

So let’s look at how those commands we send map to events.



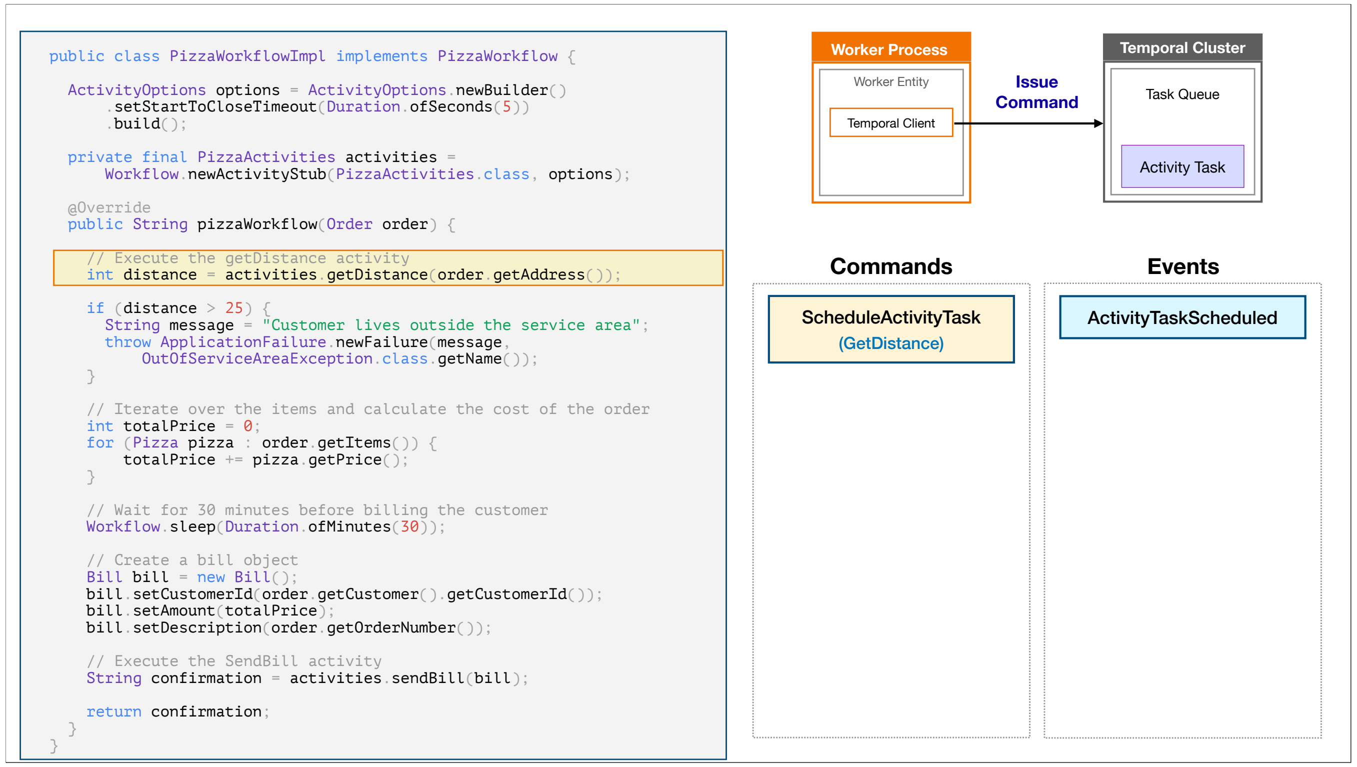
```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Let’s go back to the code you saw previously; the basic pizza order workflow.

I want to explain the layout you'll see as I proceed through the explanation. The workflow code is on the left, while the upper-right will illustrate the interaction between the Worker and the Temporal Cluster.

Below that, I show a running list of the Commands issued, with the corresponding Events just to the right of it.

Finally, since I am going to approach this from the perspective of Commands, I will only include the Events that relate to those Commands. Specifically, I will only show the ones related to the Activities and Timer in this Workflow, as this should be sufficient for you to see the pattern.



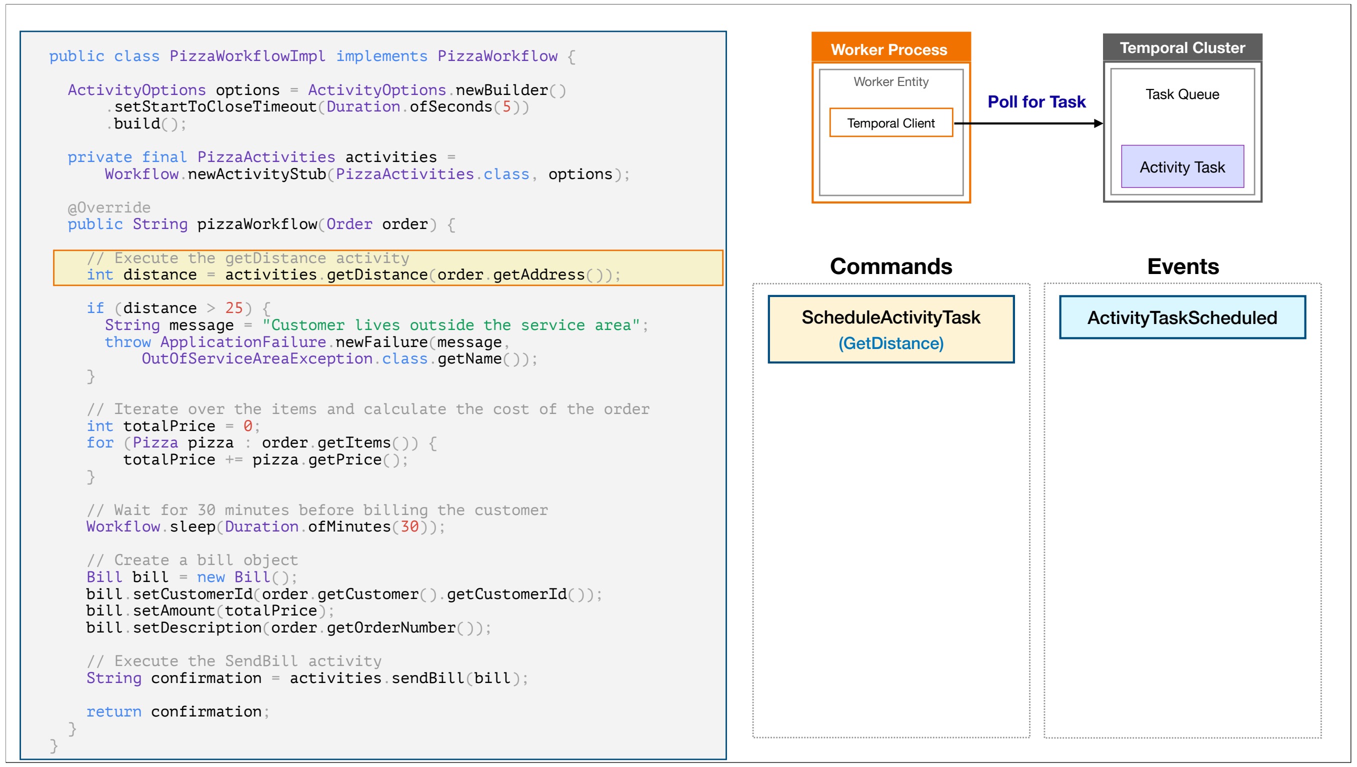
```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

The call to the getDistance activity is the first thing in the Workflow that causes a command to be issued.

In response to this Command, the Temporal Cluster creates an Activity Task, adds it to the Task Queue, and appends the `ActivityTaskScheduled` Event to this Workflow Execution's history.

I colored the rectangle for this Event light blue to indicate that it's the direct result of a Command.

By the way, the Event History is durably persisted to the database used by the Temporal Cluster, so it will survive even if the Temporal Cluster itself crashes.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

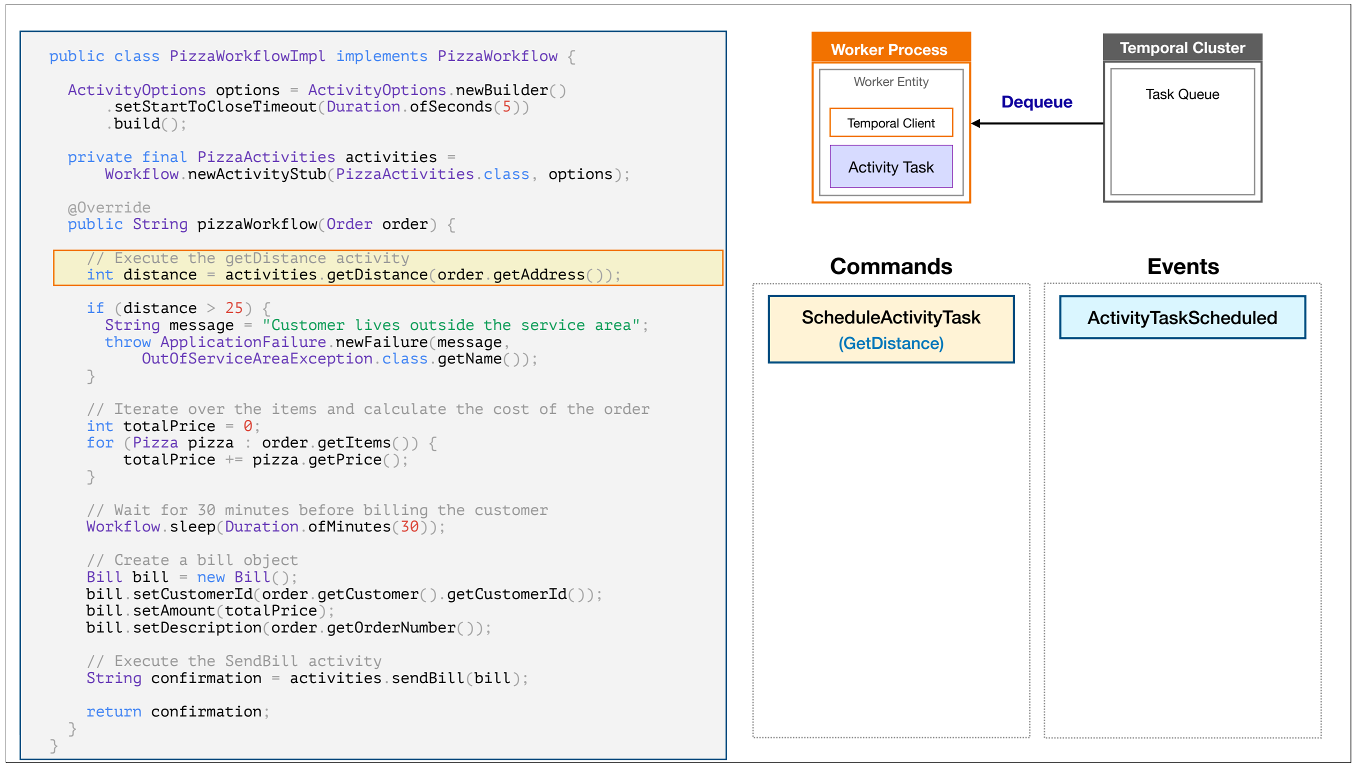
Commands

ScheduleActivityTask
(GetDistance)

Events

ActivityTaskScheduled

When a Worker has spare capacity to do some work, and that might be the same Worker that sent the Command or another Worker that's listening to the this Task Queue, it will poll that Task Queue on the Temporal Cluster.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
      totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Activity Task

Dequeue

Temporal Cluster

Task Queue

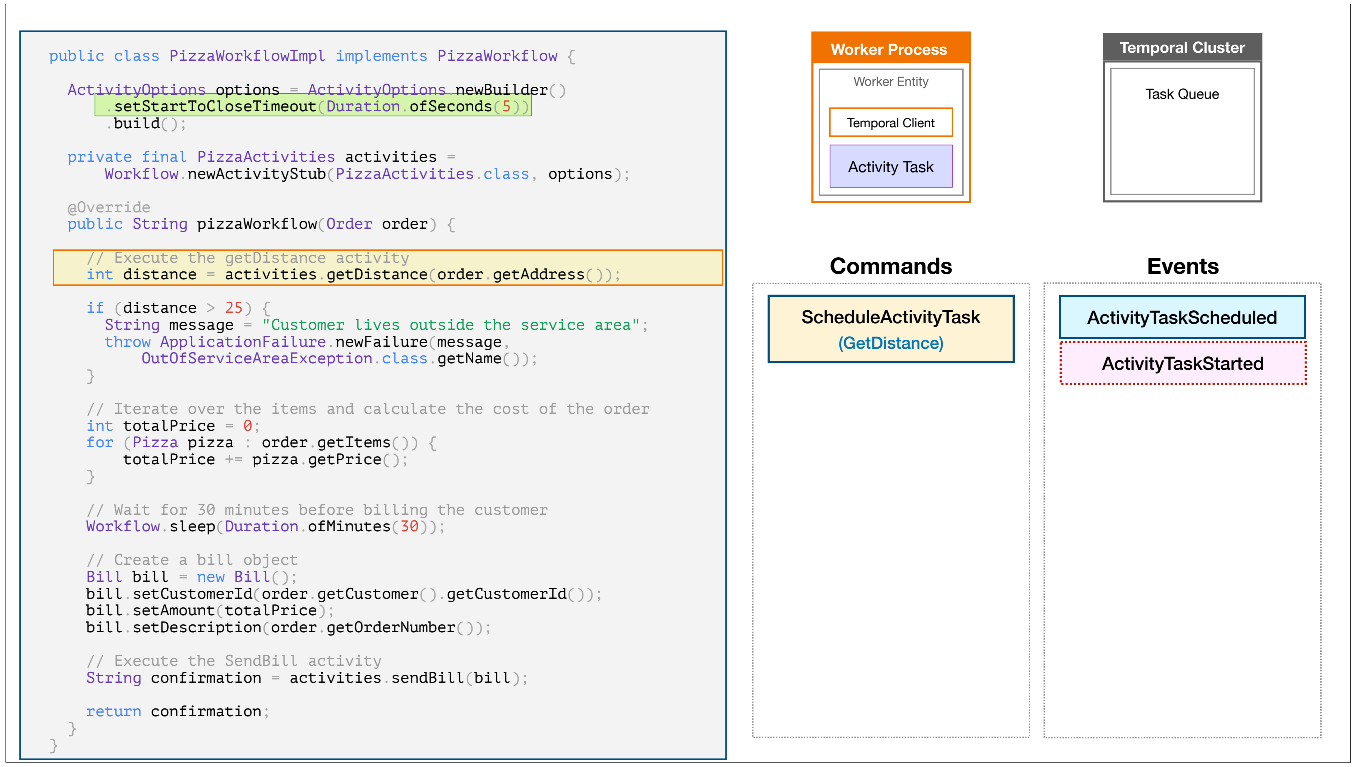
Commands

ScheduleActivityTask
(GetDistance)

Events

ActivityTaskScheduled

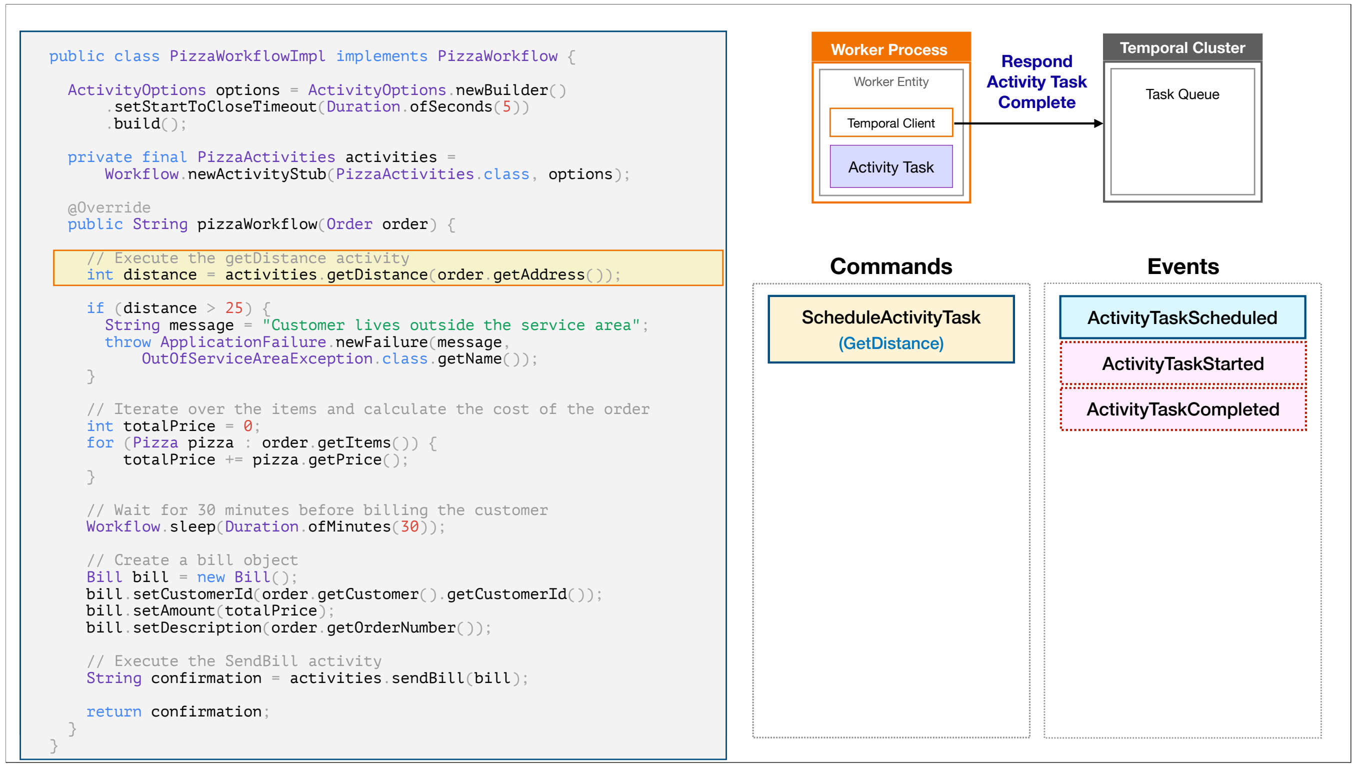
After the Temporal Cluster matches a Worker that's polling for a Task with a Task that's queued, the Worker will then begin executing the code needed to complete that Task.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

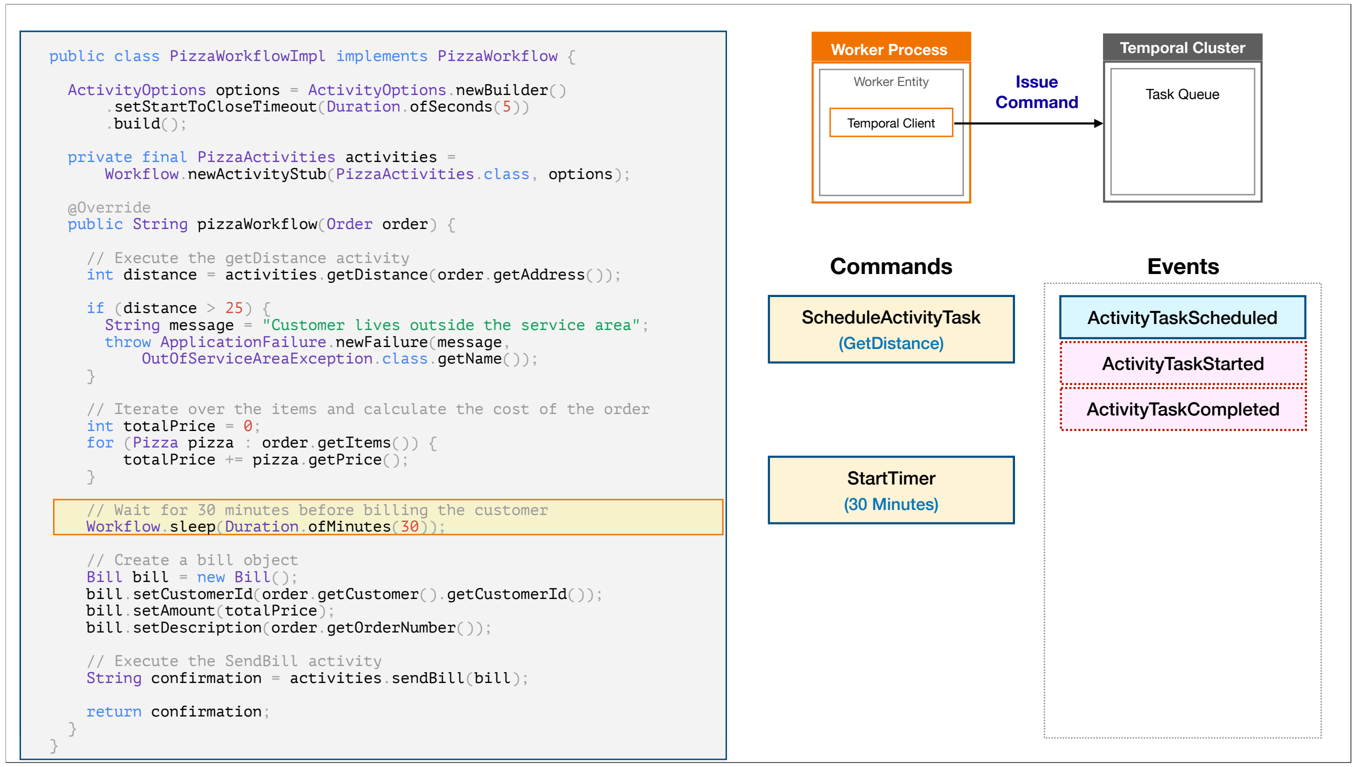
The Temporal Cluster logs another Event in response to the Worker accepting the Task: `ActivityTaskStarted`. I used a different color and border style for the rectangle depicting this event to indicate that it's an indirect result of the Command.

Temporal is designed to ensure that a Task is only ever given to a single Worker, although it will reschedule the Task if this Worker fails to complete it within the time constraints you've specified. In this case, the `ActivityOptions` at the top of the Workflow Definition sets a Start-to-Close Timeout of 5 seconds, which means that the Worker must complete this Task within 5 seconds.



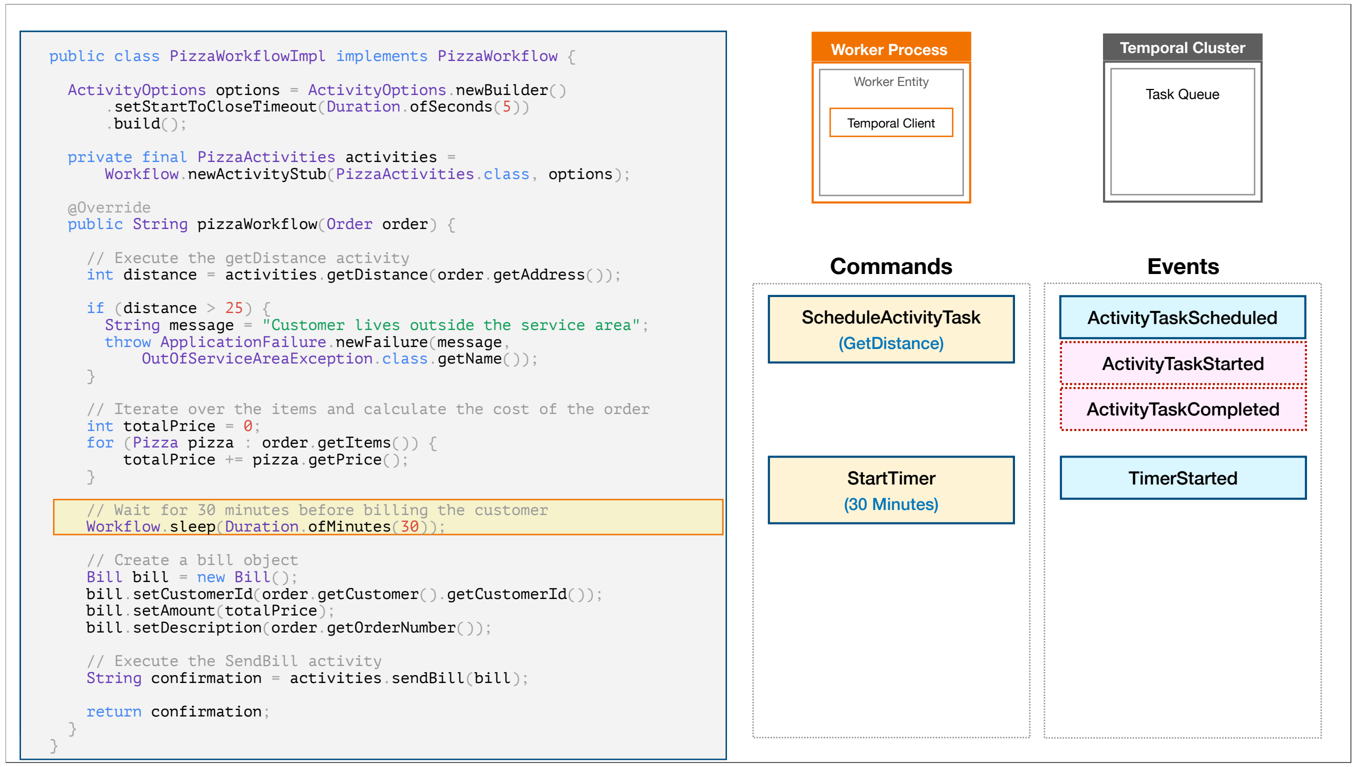
```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

The Worker executes the code within the Activity Definition, and when that function returns a result, the Worker sends a message to the Temporal Cluster, notifying it that the Task is complete. To be clear, this is just a notification, not a Command, because it's not requesting the Temporal Cluster to do something that will allow Workflow Execution to progress. In response to this notification, the Temporal Cluster logs another Event: `ActivityTaskCompleted`.



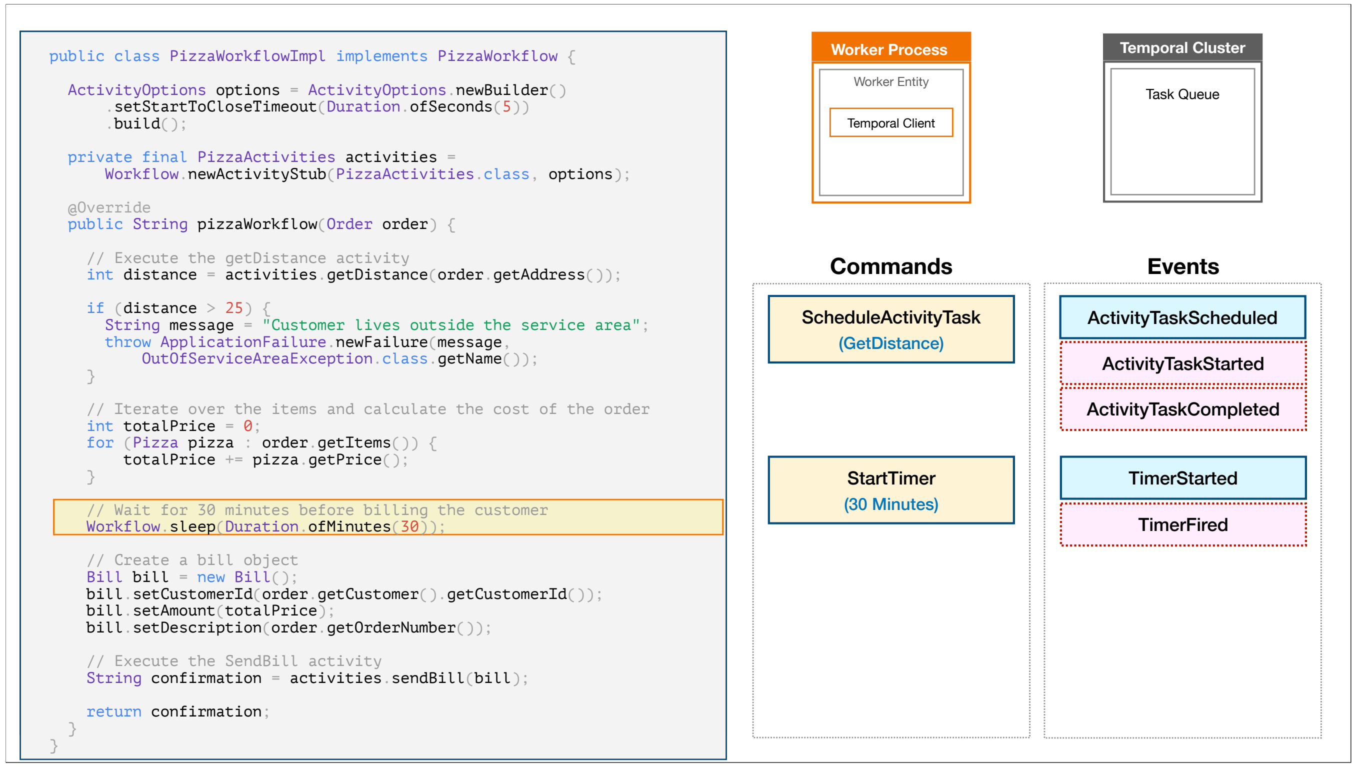
```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

The next statement that results in a Command is the call to `sleep`, which issues a `StartTimer` command.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

The Temporal Cluster responds by starting a Timer for 30 minutes and logging a `TimerStarted` Event to the history.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask (GetDistance)

StartTimer (30 Minutes)

Events

ActivityTaskScheduled

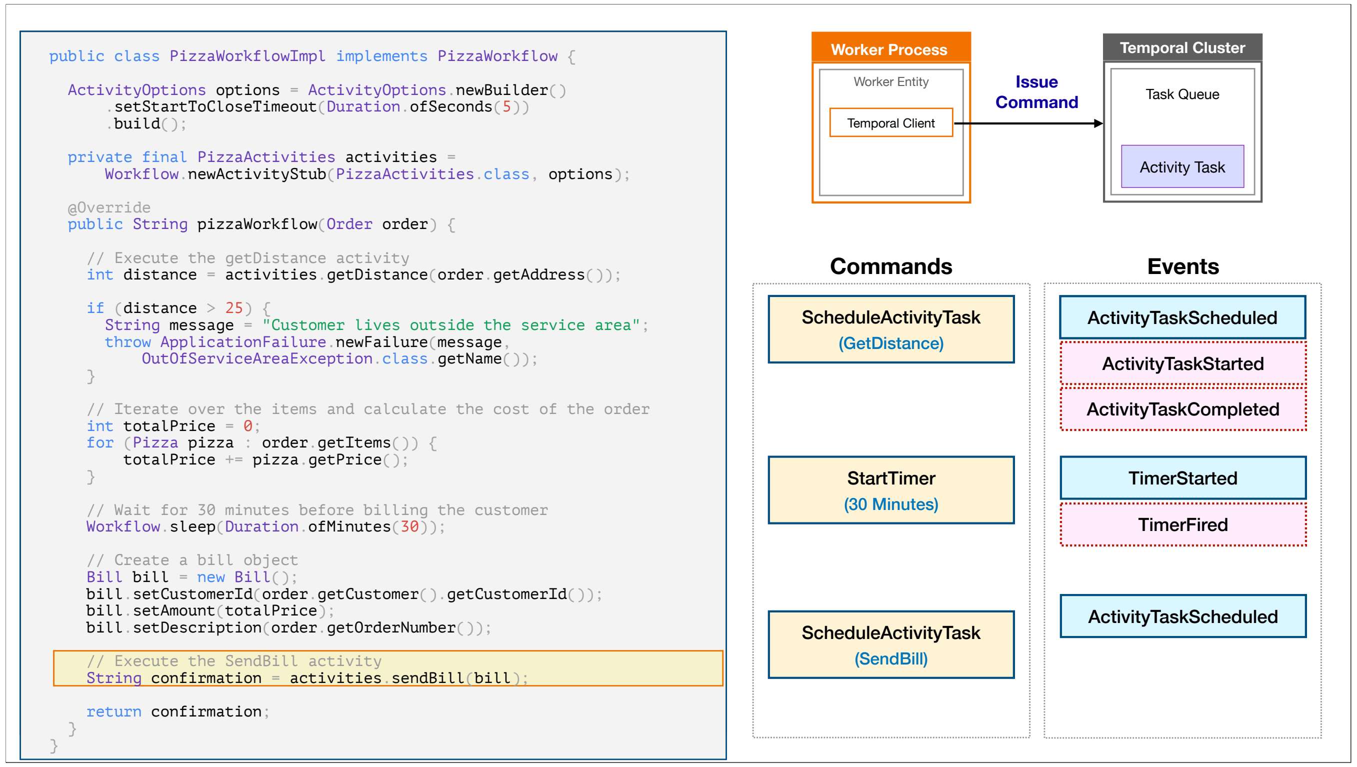
ActivityTaskStarted

ActivityTaskCompleted

TimerStarted

TimerFired

After 30 minutes has elapsed, the Timer is fired, and the Temporal Cluster logs the Event to the history. The Workflow Execution continues with the next statement, but this is an internal step.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Issue Command

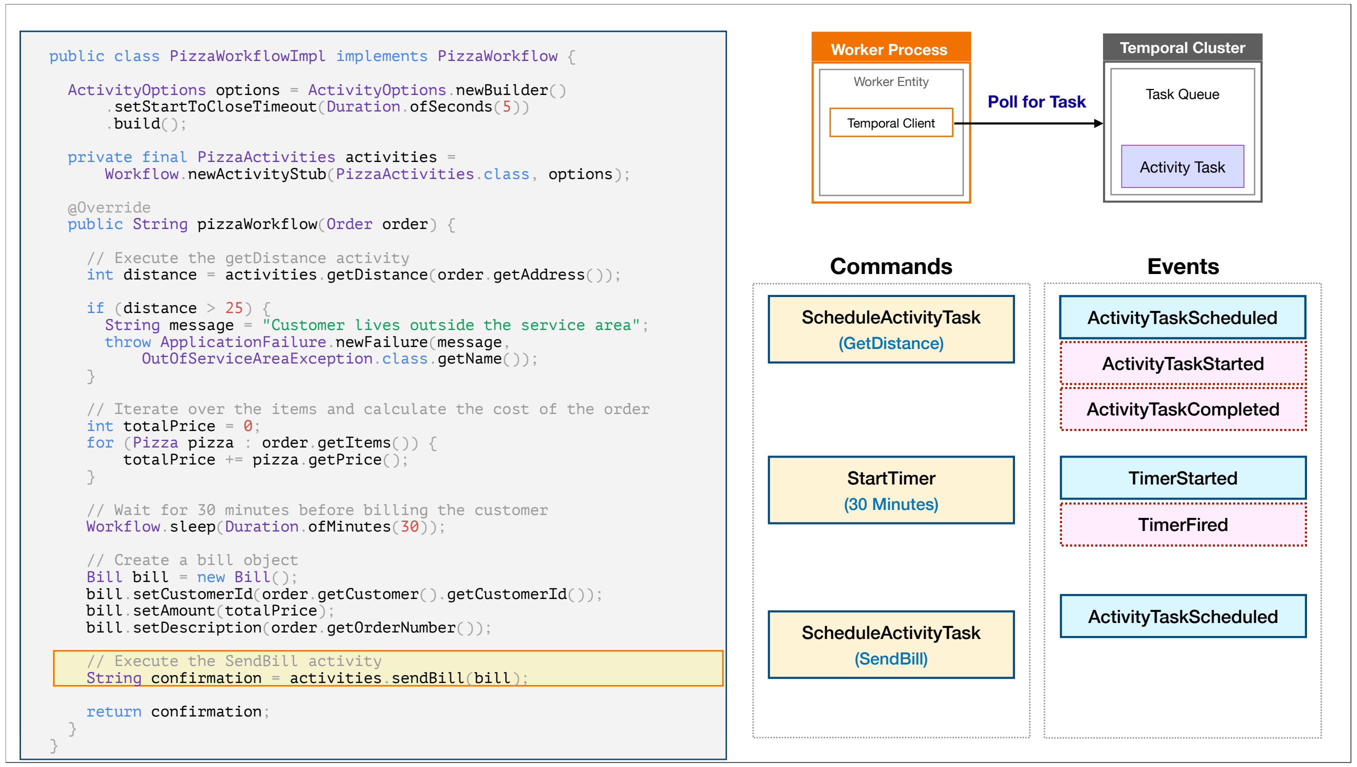
Temporal Cluster

Task Queue

Activity Task

Commands	Events
ScheduleActivityTask (GetDistance)	ActivityTaskScheduled ActivityTaskStarted ActivityTaskCompleted
StartTimer (30 Minutes)	TimerStarted TimerFired
ScheduleActivityTask (SendBill)	ActivityTaskScheduled

It then reaches the call to the sendBill Activity Method and issues another `ScheduleActivityTask` Command. The Temporal Cluster adds an Activity Task to the queue and logs an `ActivityTaskScheduled` Event to history.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask (GetDistance)

StartTimer (30 Minutes)

ScheduleActivityTask (SendBill)

Events

ActivityTaskScheduled

ActivityTaskStarted

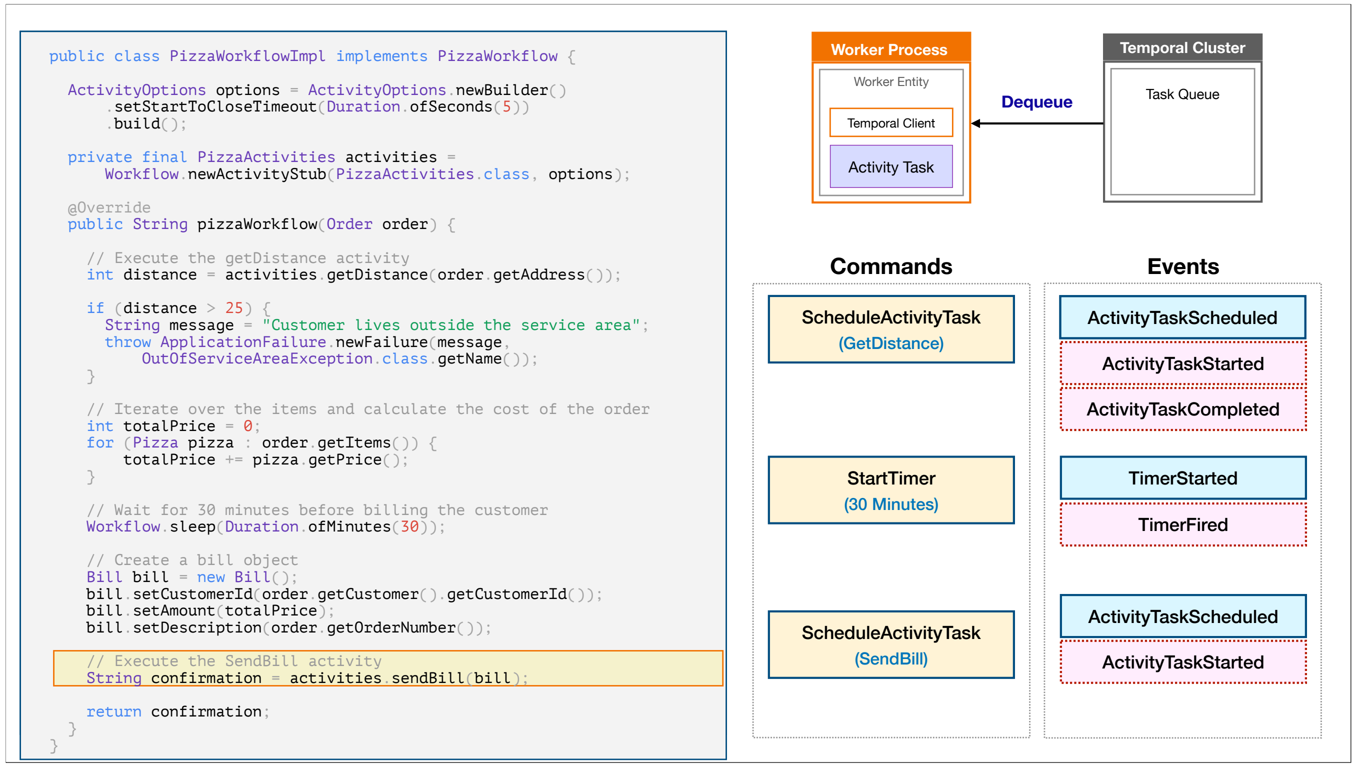
ActivityTaskCompleted

TimerStarted

TimerFired

ActivityTaskScheduled

When the Worker polls the Task Queue, it will be matched with this Task.

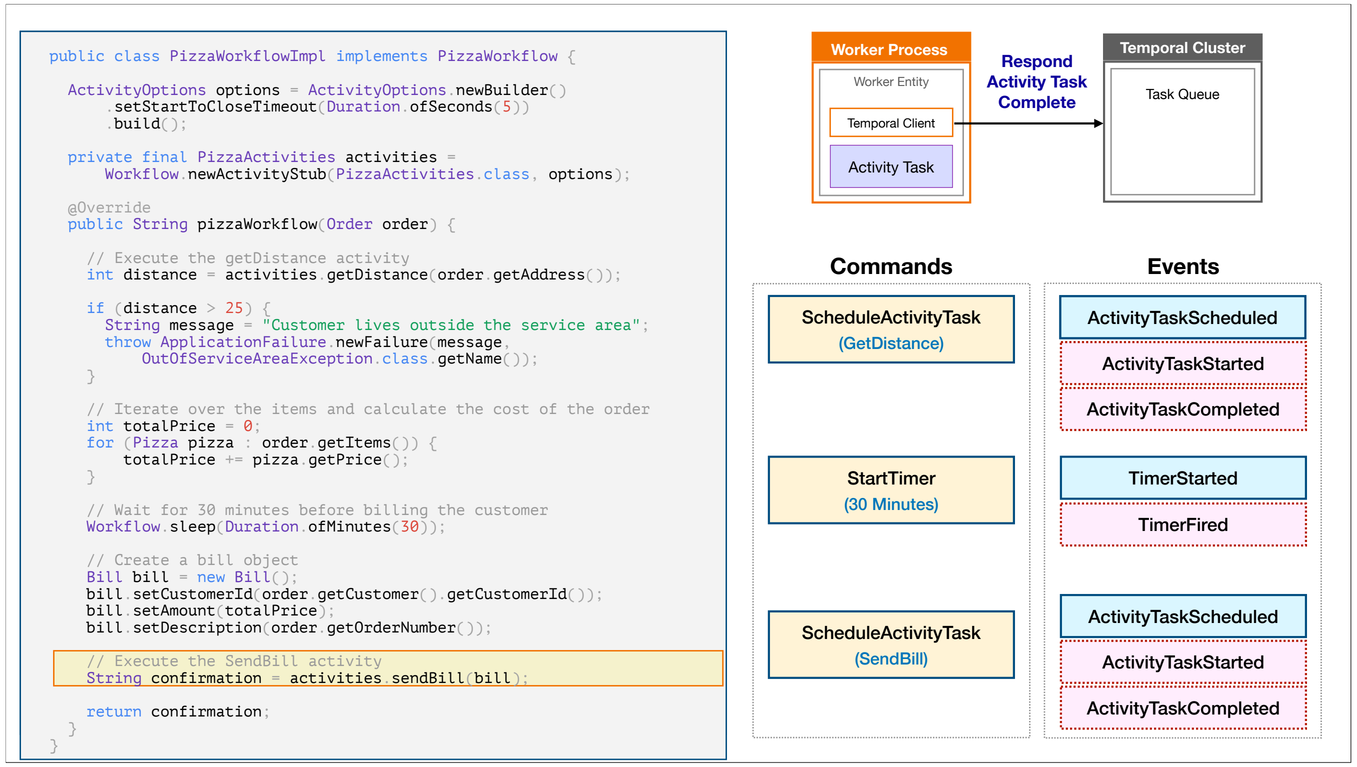


```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands	Events
ScheduleActivityTask (GetDistance)	ActivityTaskScheduled
	ActivityTaskStarted
	ActivityTaskCompleted
StartTimer (30 Minutes)	TimerStarted
	TimerFired
ScheduleActivityTask (SendBill)	ActivityTaskScheduled
	ActivityTaskStarted

The Worker removes it from the queue, and begins working on it.

The Temporal Cluster logs an ActivityTaskStarted event to the history, signifying that the Task has been dequeued.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Activity Task

Respond Activity Task Complete

Temporal Cluster

Task Queue

Commands

ScheduleActivityTask (GetDistance)

StartTimer (30 Minutes)

ScheduleActivityTask (SendBill)

Events

ActivityTaskScheduled

ActivityTaskStarted

ActivityTaskCompleted

TimerStarted

TimerFired

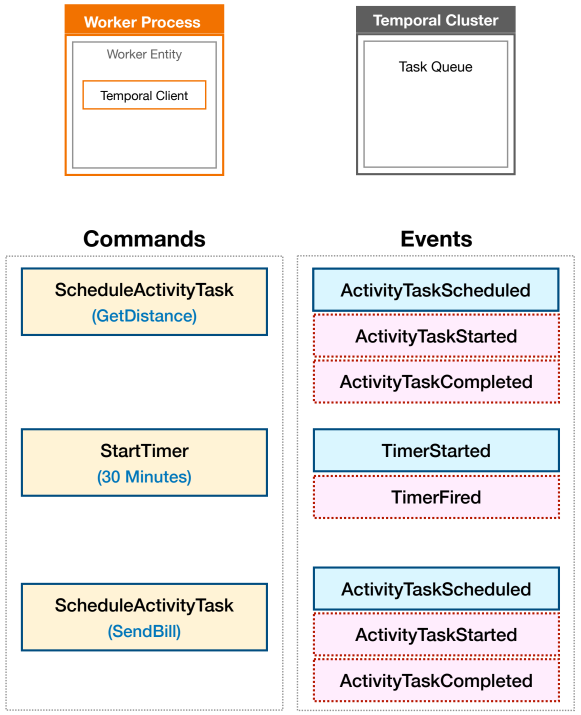
ActivityTaskScheduled

ActivityTaskStarted

ActivityTaskCompleted

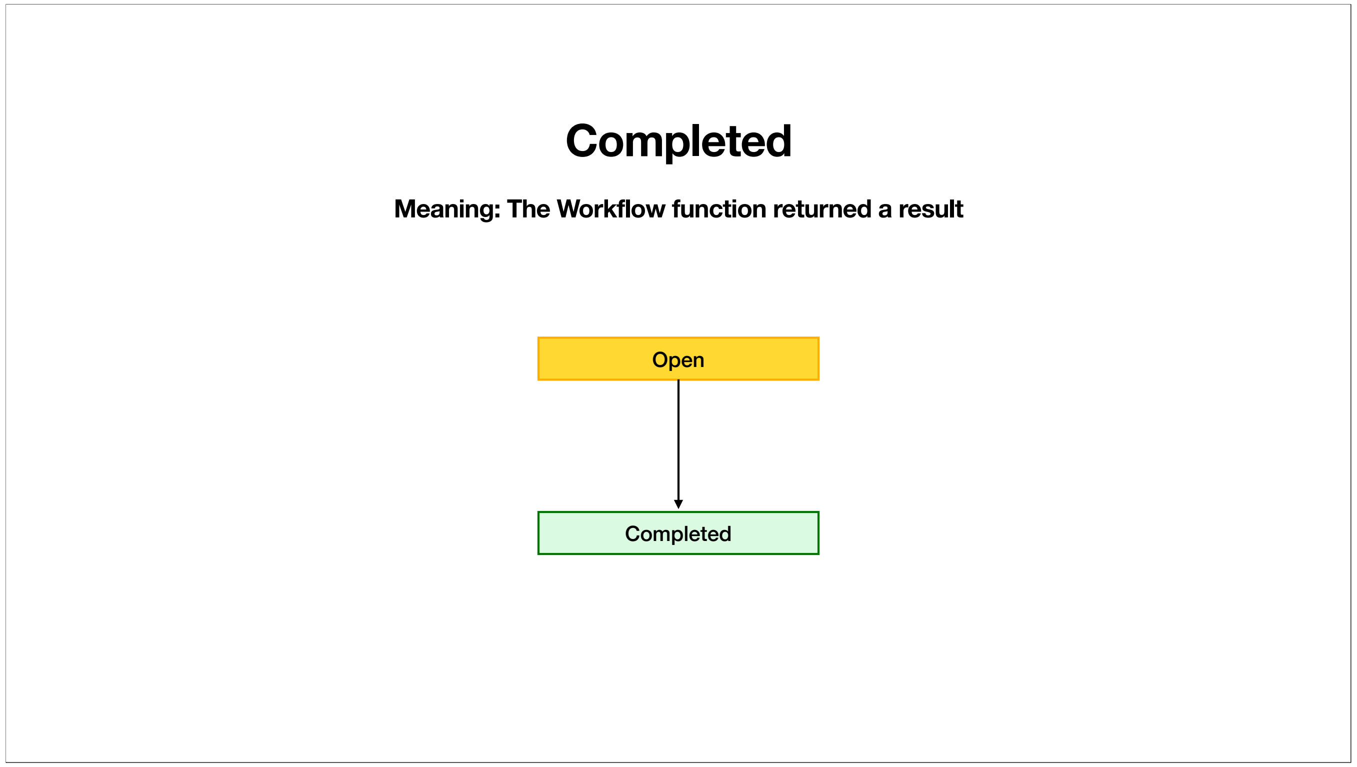
When the Activity function returns, the Task is complete, and the Worker notifies the Temporal Cluster. In response, the Temporal Cluster logs the `ActivityTaskCompleted` Event to the history.

```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  @Override
  public String pizzaWorkflow(Order order) {

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    int totalPrice = 0;
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```



10 minute break

Workflow Execution States



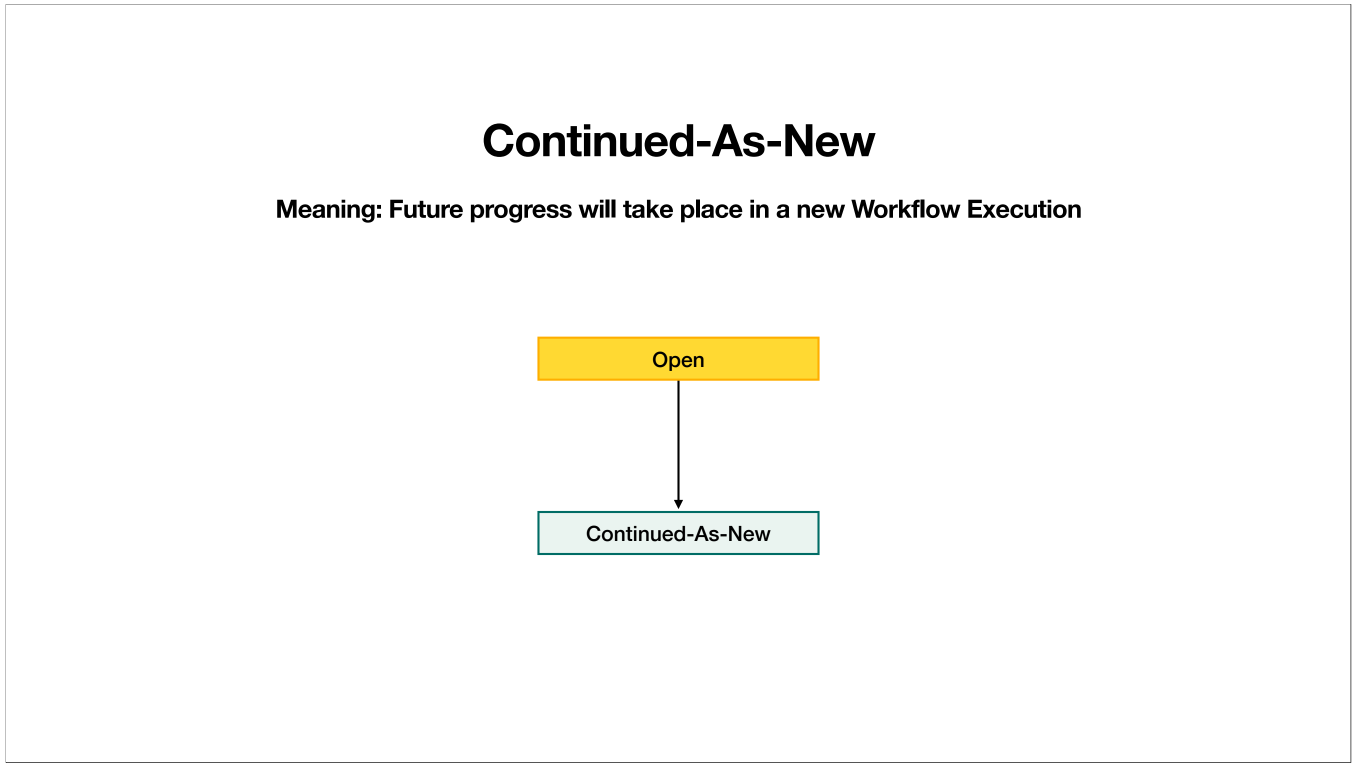
Completed

Meaning: The Workflow function returned a result

Open

Completed

Workflow Execution can enter the closed state for any one of several reasons. The most desirable reason is because the Workflow function returned a result, meaning that it completed successfully.



Continued-As-New

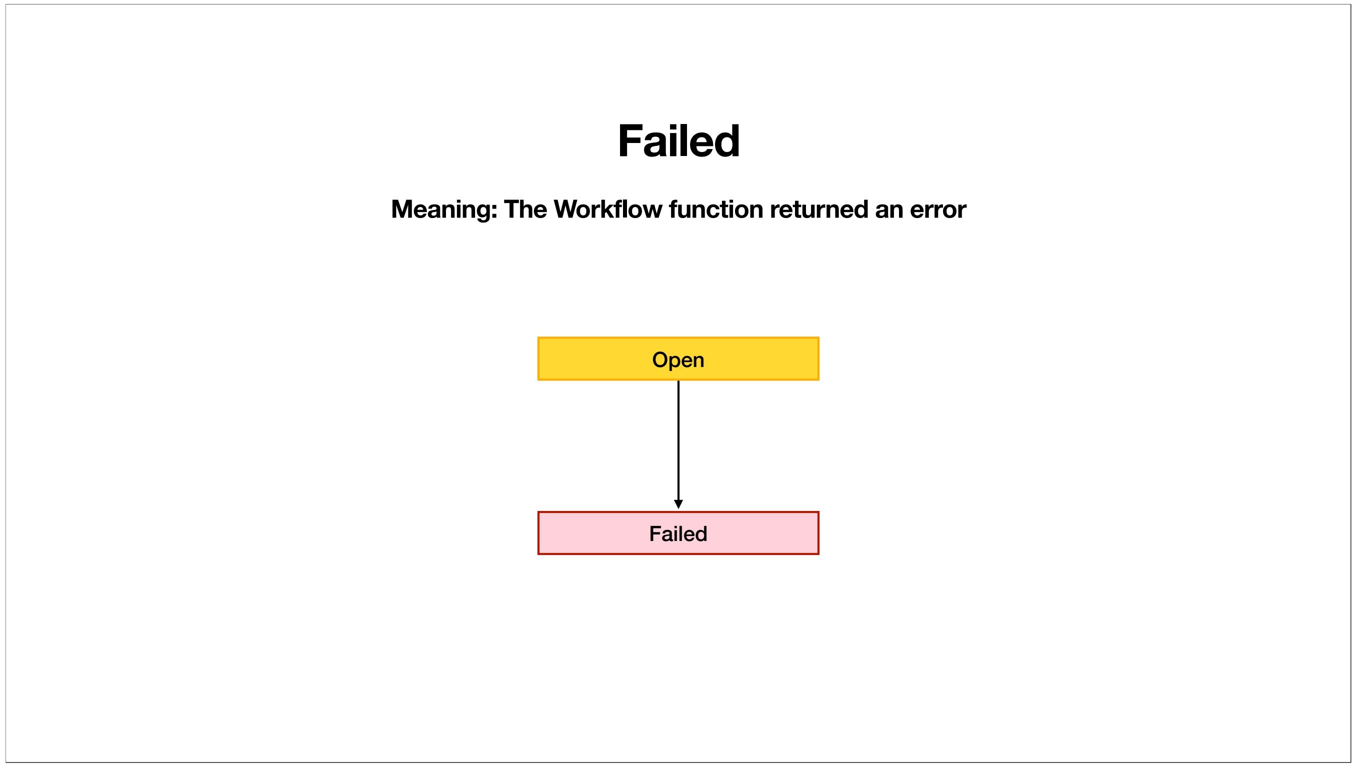
Meaning: Future progress will take place in a new Workflow Execution

Open

Continued-As-New

A variation on this is known as Continued-as-New, which means that the code is still running, but any future progress will take place in a new Workflow Execution and Event History.

Why would a Workflow do this? In order to maintain good performance, Temporal enforces a limit on both the size and number of events in the history associated with a Workflow Execution. You'll find the details in our documentation, but for reference, you're unlikely to reach these limits unless a single execution of your Workflow runs thousands of Activities during its execution. Continue-As-New is a technique designed to avoid reaching these limits.



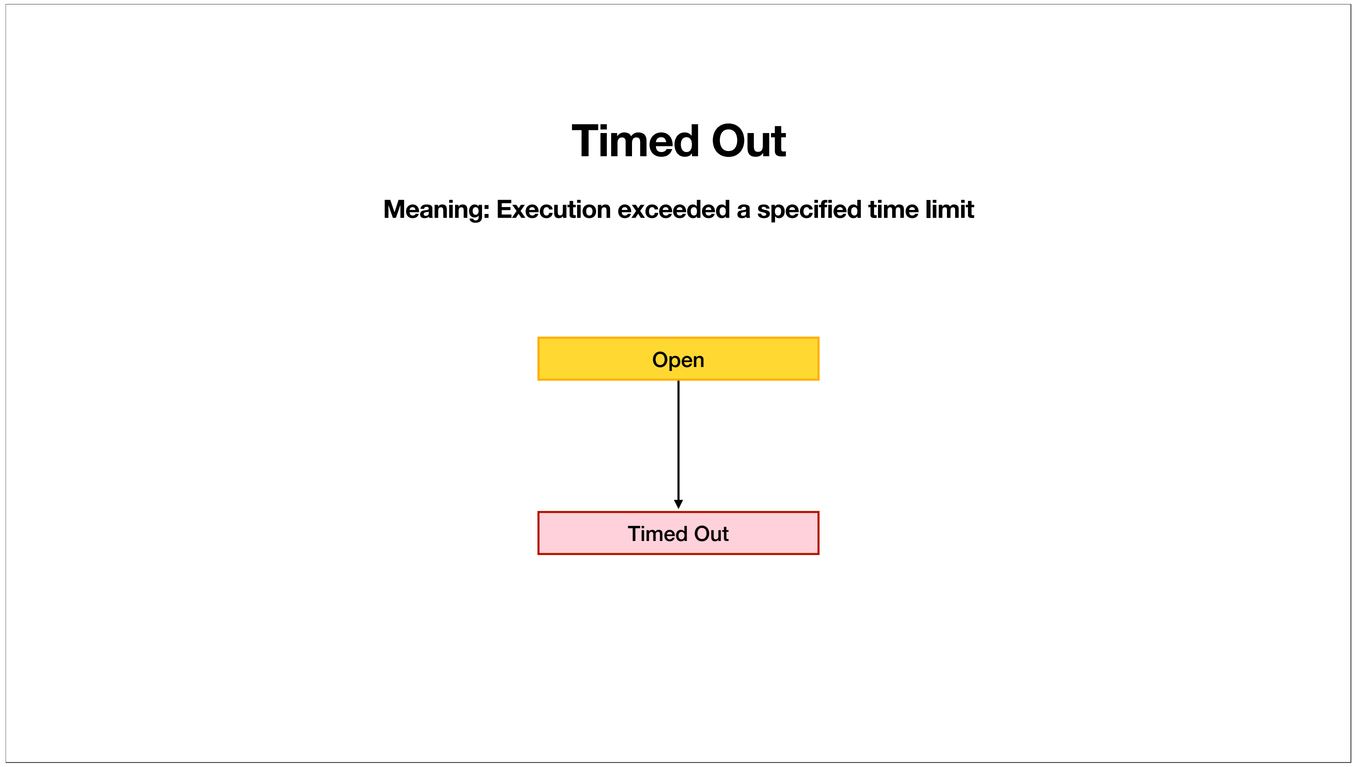
Failed

Meaning: The Workflow function returned an error

Open

Failed

However, Workflow Execution can close as a result of something undesirable happening. An example of this is a failed execution, which happens when the Workflow function returns an error instead of a result.



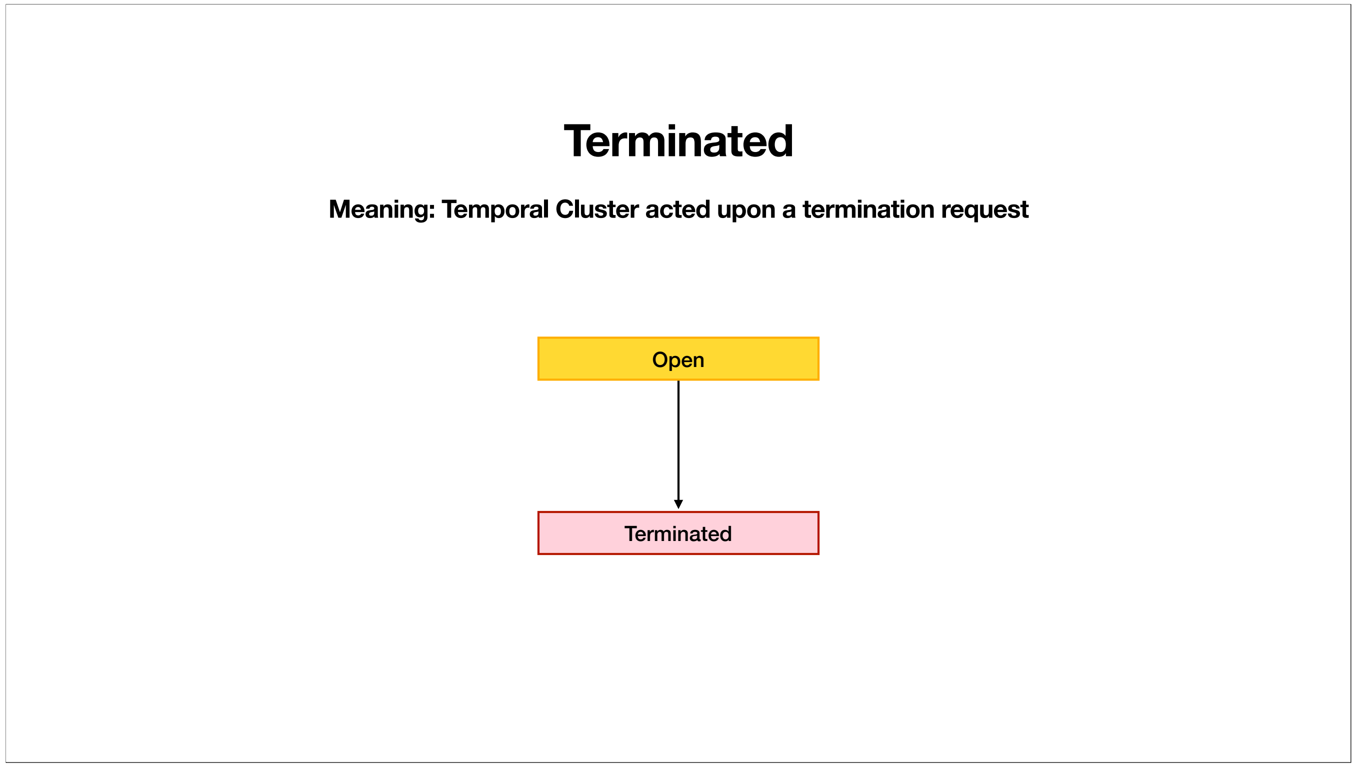
Timed Out

Meaning: Execution exceeded a specified time limit

Open

Timed Out

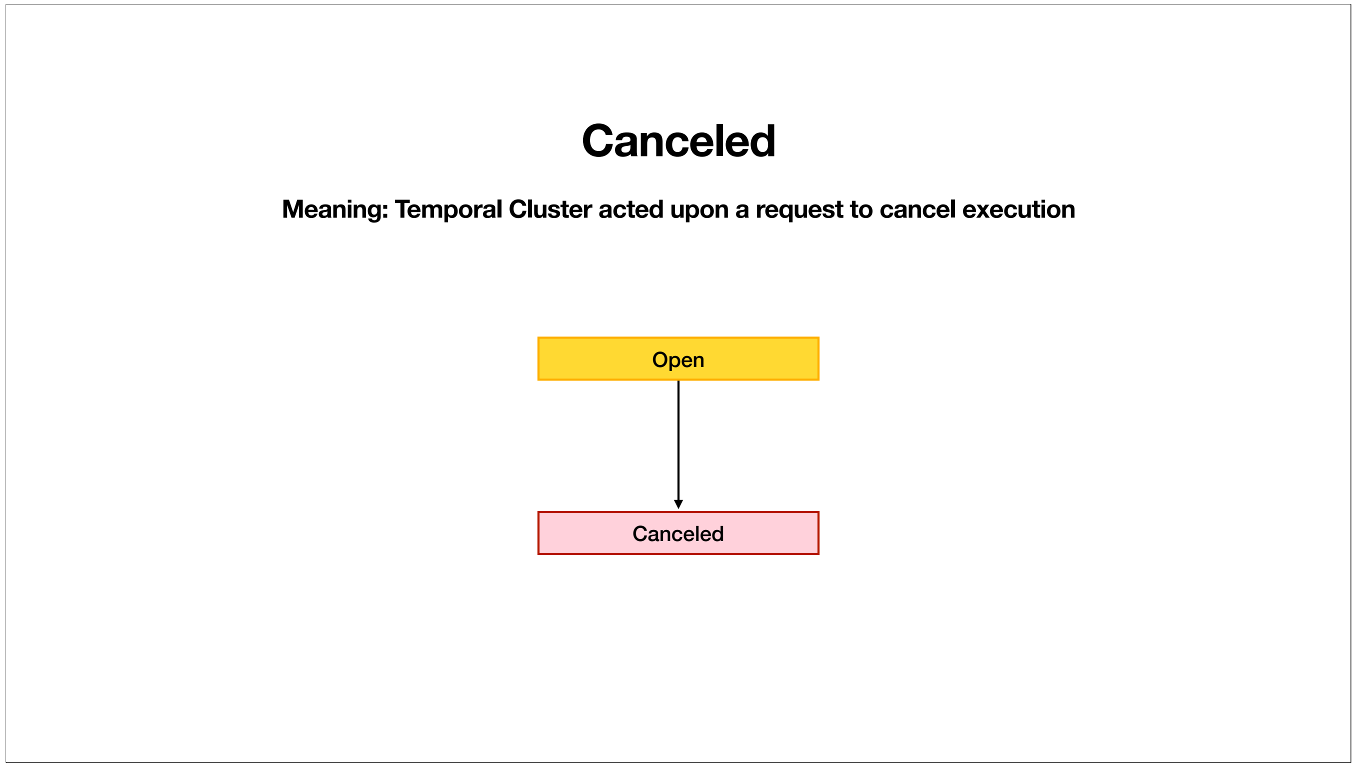
The Workflow Execution might time out, meaning that a time limit associated with the execution elapsed before the Workflow function returned either a result or an error.



Terminated

Meaning: Temporal Cluster acted upon a termination request

It might be terminated, whether from code, the command line, or the Web UI.



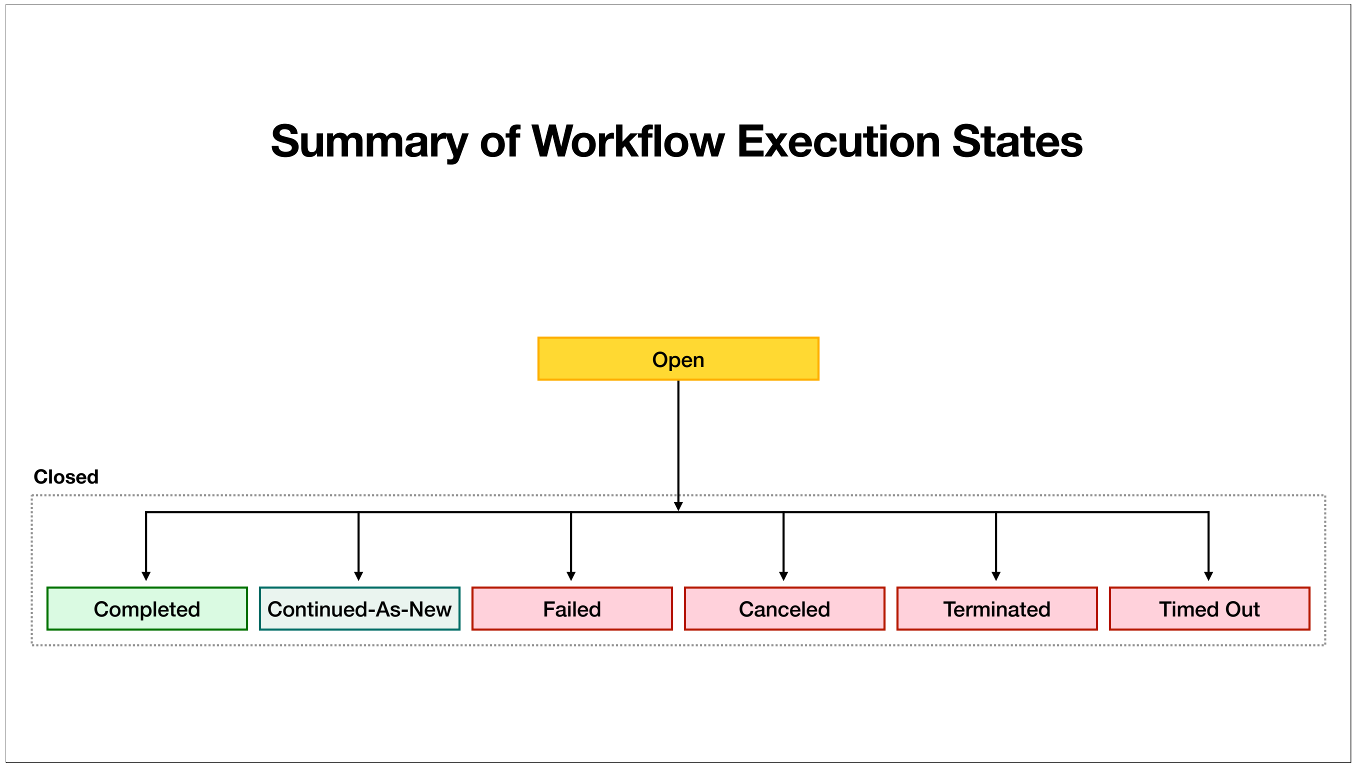
Canceled

Meaning: Temporal Cluster acted upon a request to cancel execution

Open

Canceled

Likewise, someone may have initiated cancellation of the Workflow using code, the command line, or the Web UI. Cancelation is similar to termination, but is a more graceful way of ending execution prematurely, since Workflows and Activities can be notified of cancelation and perform some cleanup before exiting.



Summary of Workflow Execution States

Open

Closed

Completed

Continued-As-New

Failed

Canceled

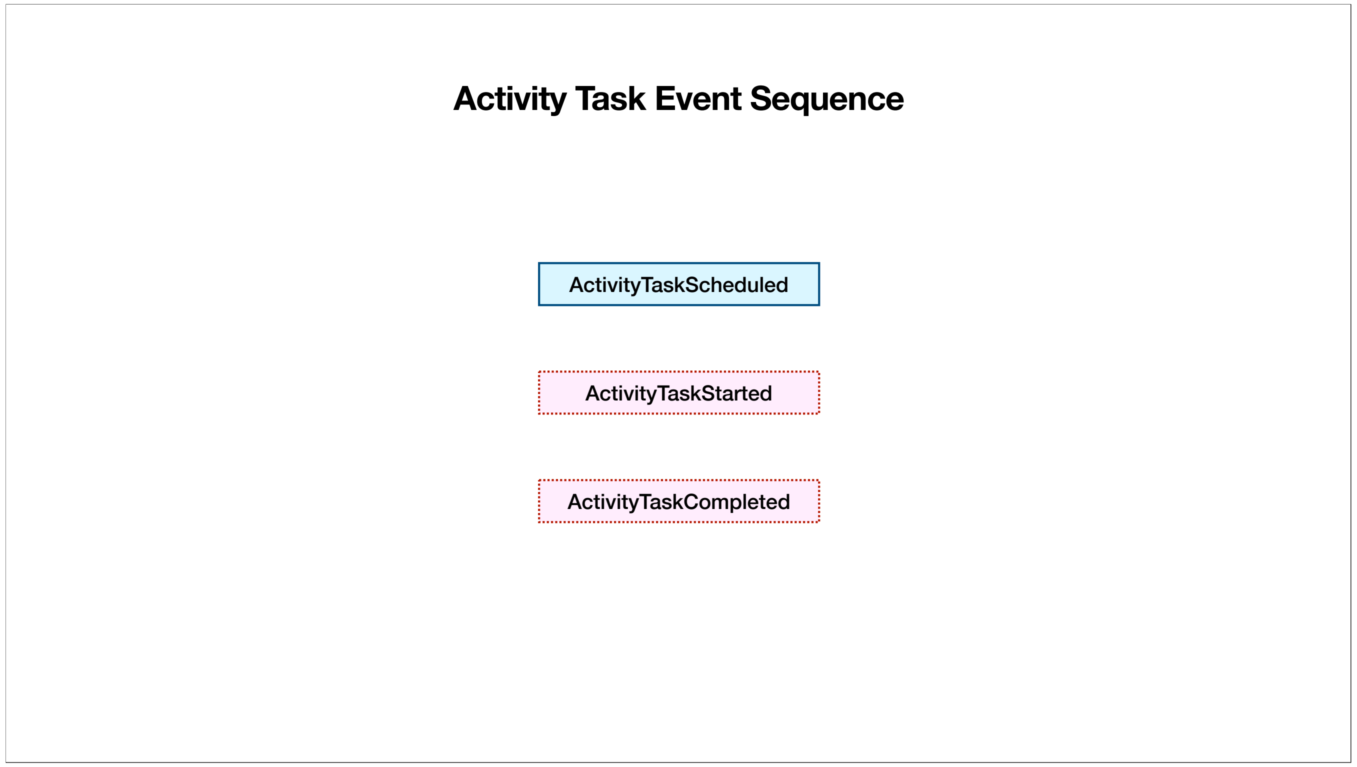
Terminated

Timed Out

In summary, an open Workflow Execution is one that is currently running. Eventually, every Workflow Execution will transition to the closed state, with a final status corresponding to one of the six items shown at the bottom.

Understanding the differences between them and what causes each to occur will help you interpret the Workflow Execution Event History. This, in turn, can help you to determine the source of a problem. For example, if the Workflow Execution ended with a status of failed, then you know that the Workflow function returned an error. In fact, the final Event in the history for that Workflow Execution will contain information about that error, which is conveniently shown in the Web UI.

Workflow and Activity Task States



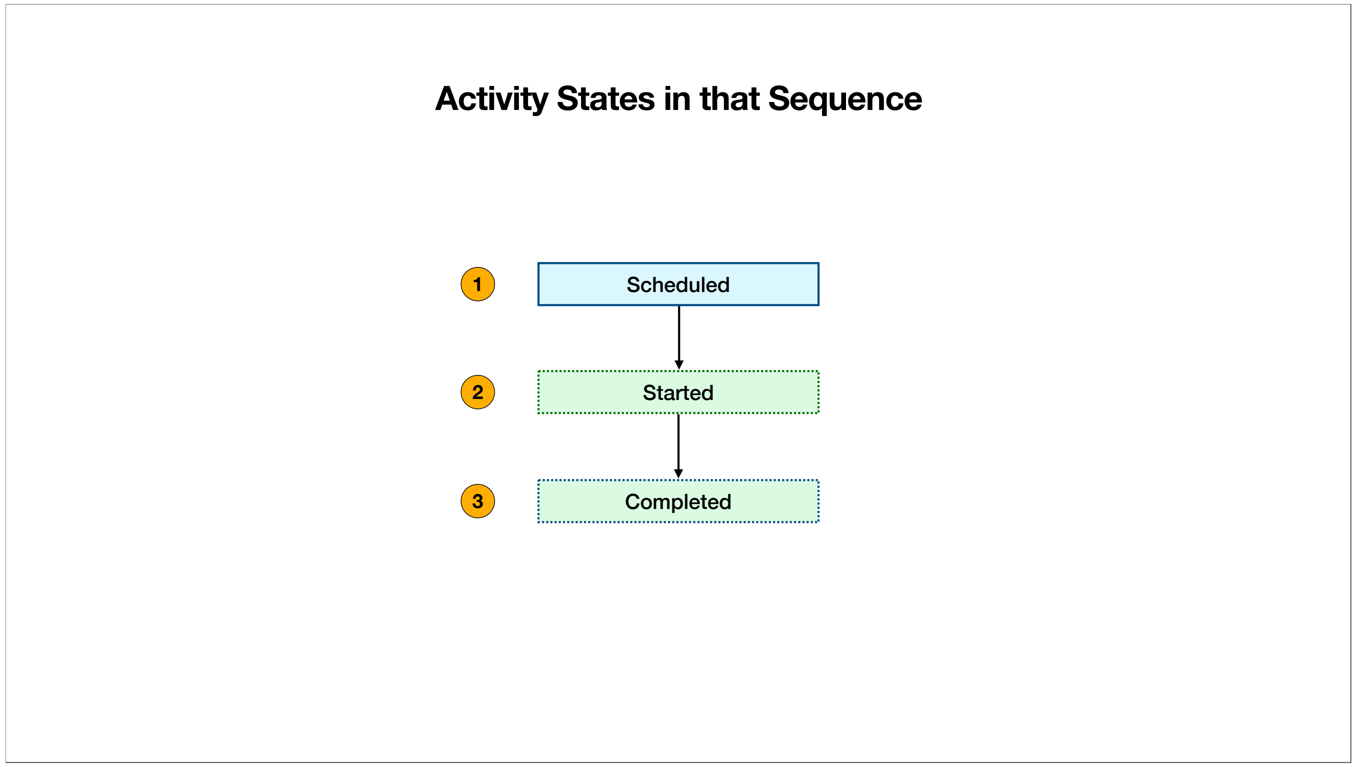
Activity Task Event Sequence

ActivityTaskScheduled

ActivityTaskStarted

ActivityTaskCompleted

Did you notice a pattern in the names of the Activity-related Events?



Activity States in that Sequence

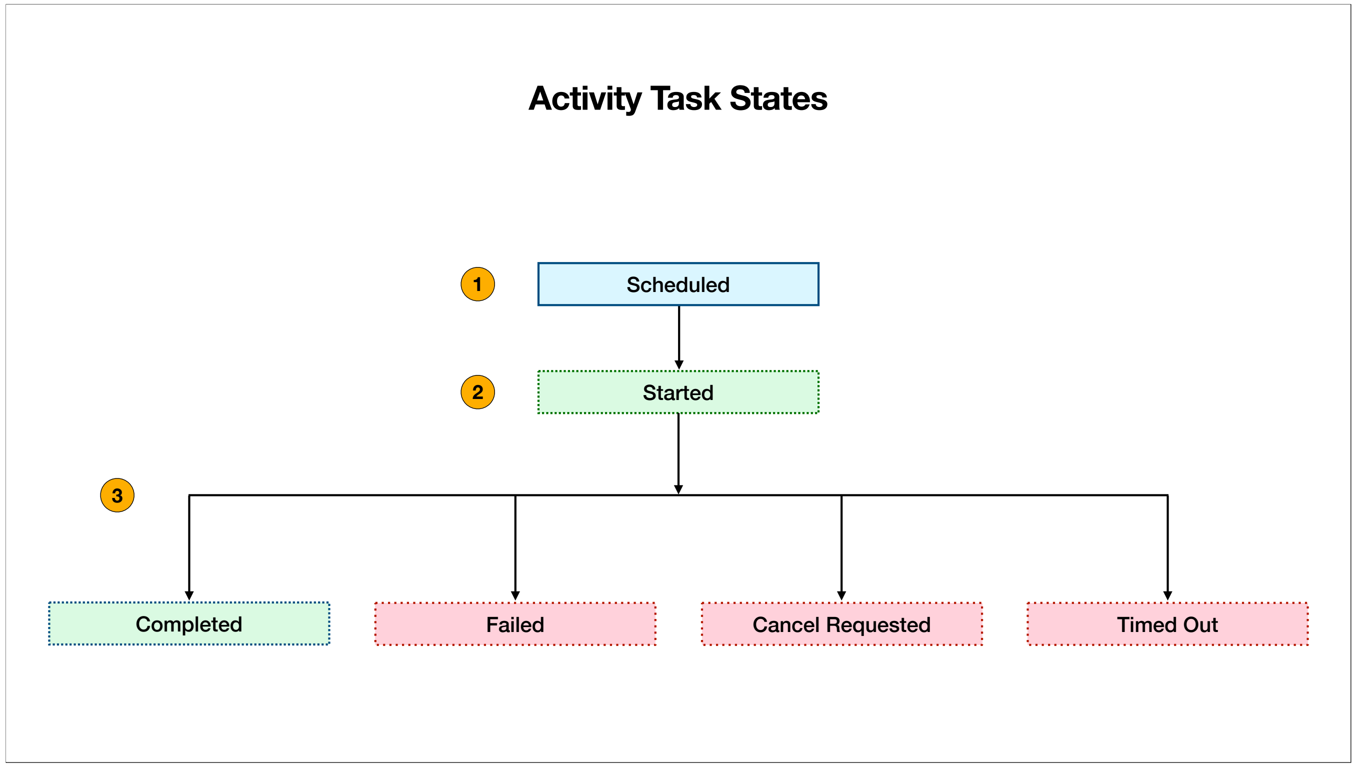
1 Scheduled

2 Started

3 Completed

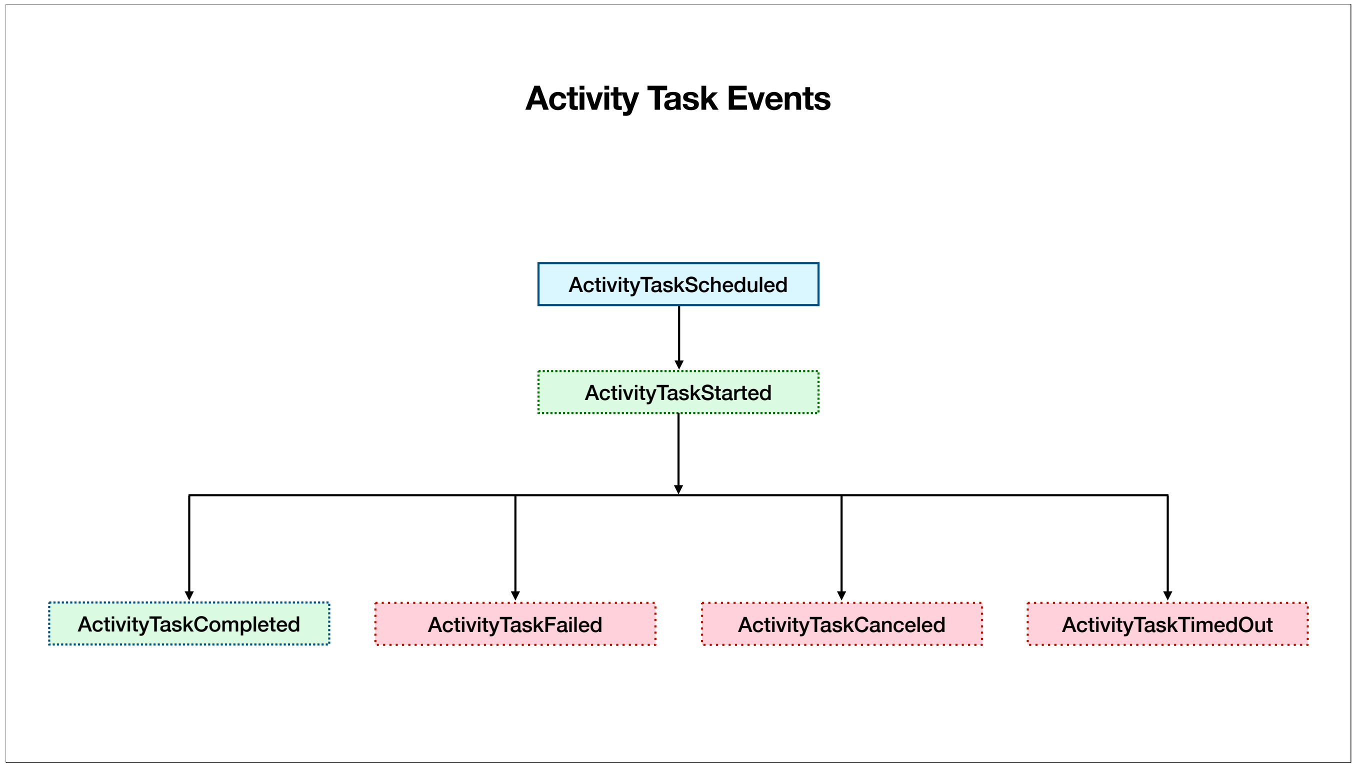
Removing "ActivityTask" from their names reveals the state of that Task at the time of the Event.

The ones that ended with the suffix of "Scheduled" indicate that a Task was added to the Task Queue, an action performed by the Cluster. This was always the first Event in that sequence and Events that represent subsequent actions performed by the Worker follow that. Tasks that end with "Started" represent the Worker dequeueing a Task, while those ending with "Completed" represents a Worker successfully finishing a Task.



However, just as with Workflow Executions, Activity Tasks have closed states that represent failure, as well as success, as you can see here. Recognizing this pattern will help you to understand the names of the Timeouts and what they represent.

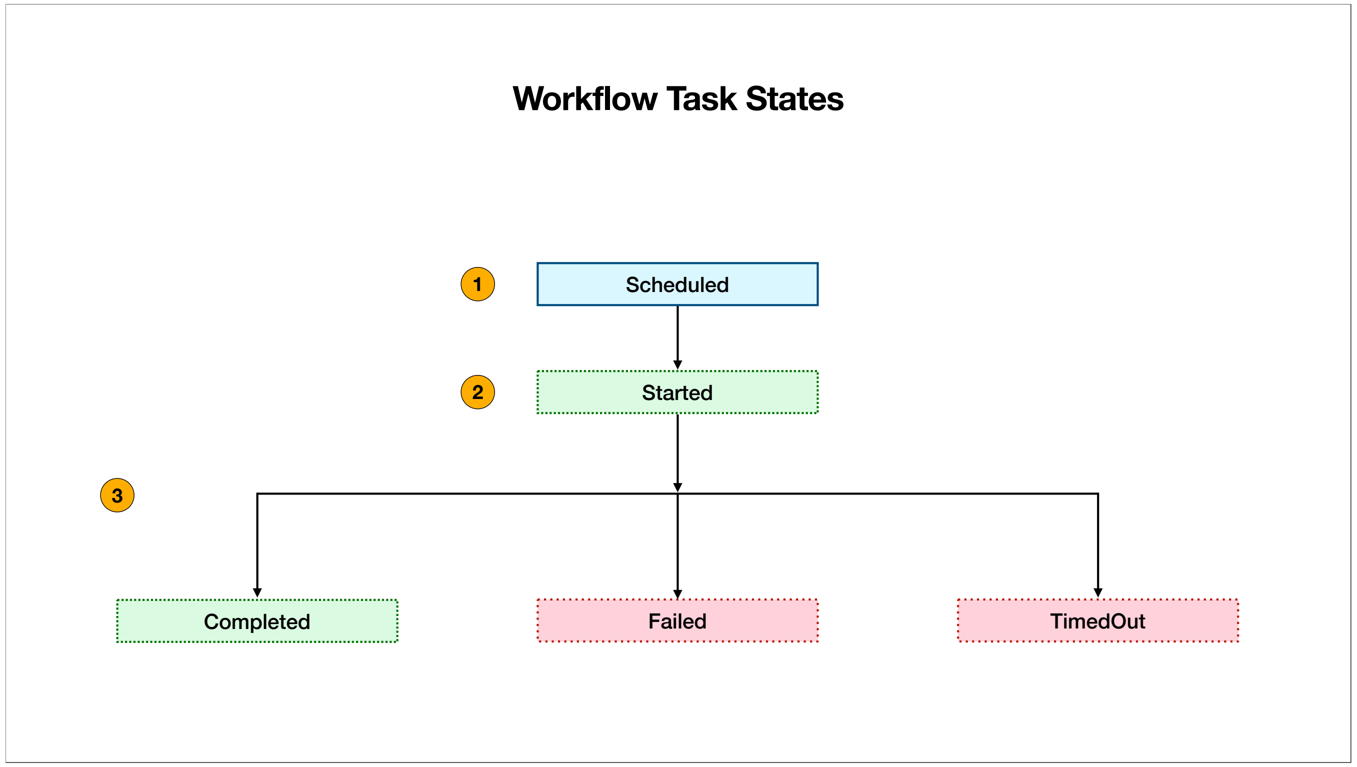
For example, Start-to-Close Timeout is the maximum amount of time allowed for between when the Worker starts working on a task and when that Task enters the closed state. In other words, it specifies the maximum duration between step 2, when the Worker begins execution, and step 3, when execution ends.



Let's now look at the Events corresponding to each of these states and the various ways that a Task can reach one of these closed states.

Here are the Events corresponding to each of those states. The Worker determines whether an Activity Task is completed or failed. If executing the code for that Task results in an error, then the Task is failed. If it runs to completion without an error, then the Task is completed.

However, it is the Temporal Cluster that determines whether the Task times out, based on whether or not the Worker notified the Cluster of the result before the time period allowed for execution elapsed. This is intuitive when you think about it, since a Worker crash is one reason that a Task might time out, and the Worker that had crashed wouldn't be able to report a time out.



Workflow Task States

1 Scheduled

2 Started

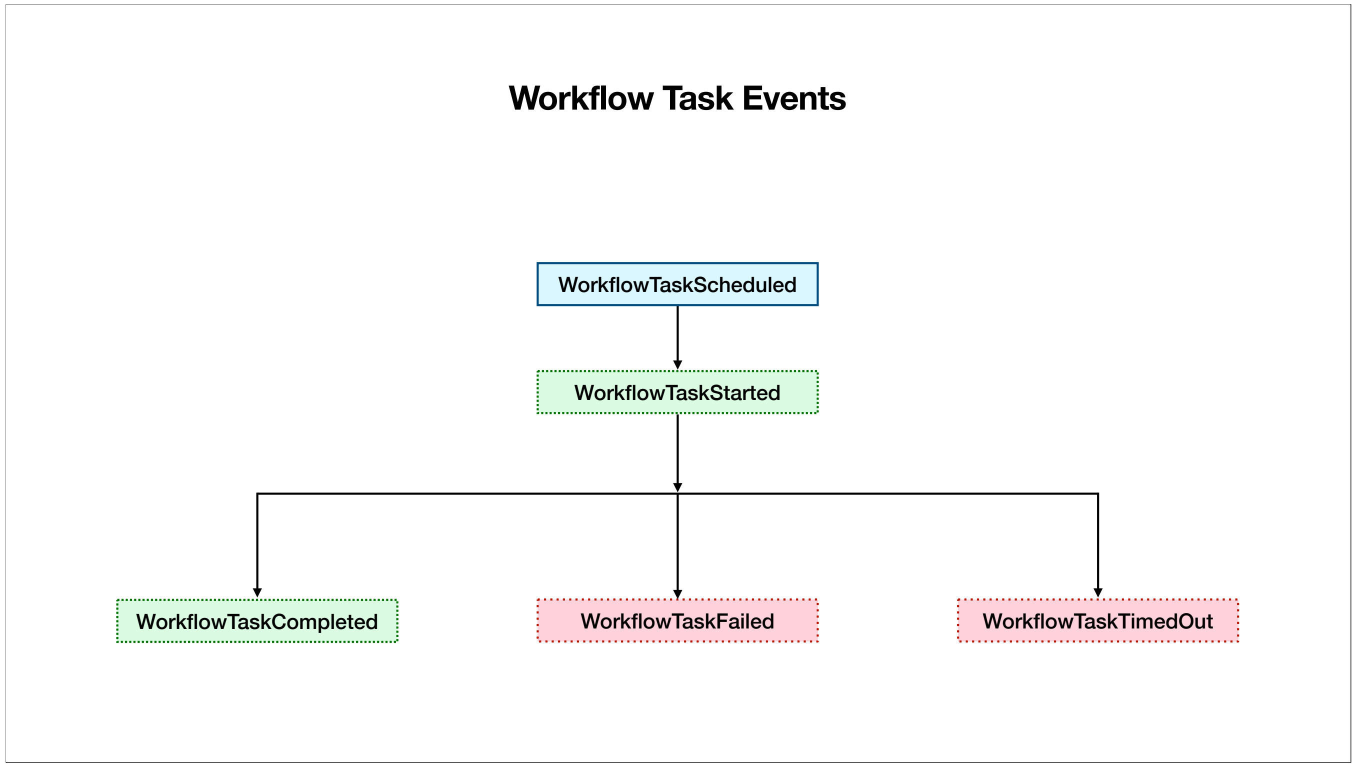
3

Completed

Failed

TimedOut

The pattern you saw applies to Workflow Tasks as well.



Workflow Task Events

WorkflowTaskScheduled

WorkflowTaskStarted

WorkflowTaskCompleted

WorkflowTaskFailed

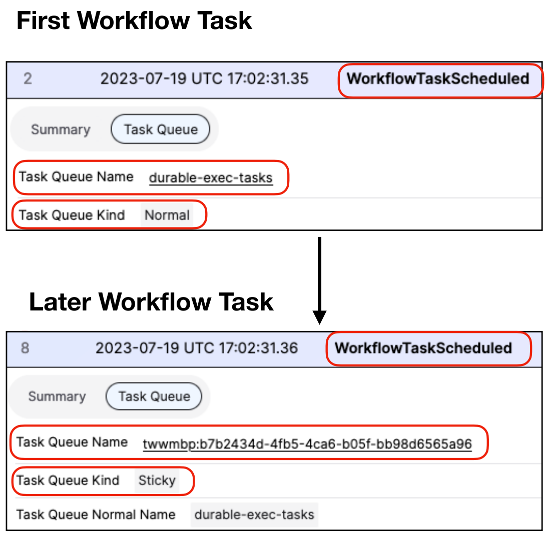
WorkflowTaskTimedOut

Here are the Events related to Workflow Tasks. As you can see, their names follow the same pattern you saw with Activity Tasks.

Now that you understand how Activity and Workflow Task events got their names, you'll be much more effective at interpreting the Event Histories in the Temporal Web UI.

Sticky Execution

- **To improve effectiveness of Worker's caching, Temporal use "sticky" execution for Workflow Tasks**
  - A Worker which completed the first Workflow Task is given preference for subsequent Workflow Tasks in the same execution via a Worker-specific Task Queue
- **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

Task Queue Name durable-exec-tasks

Task Queue Kind Normal

Later Workflow Task

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

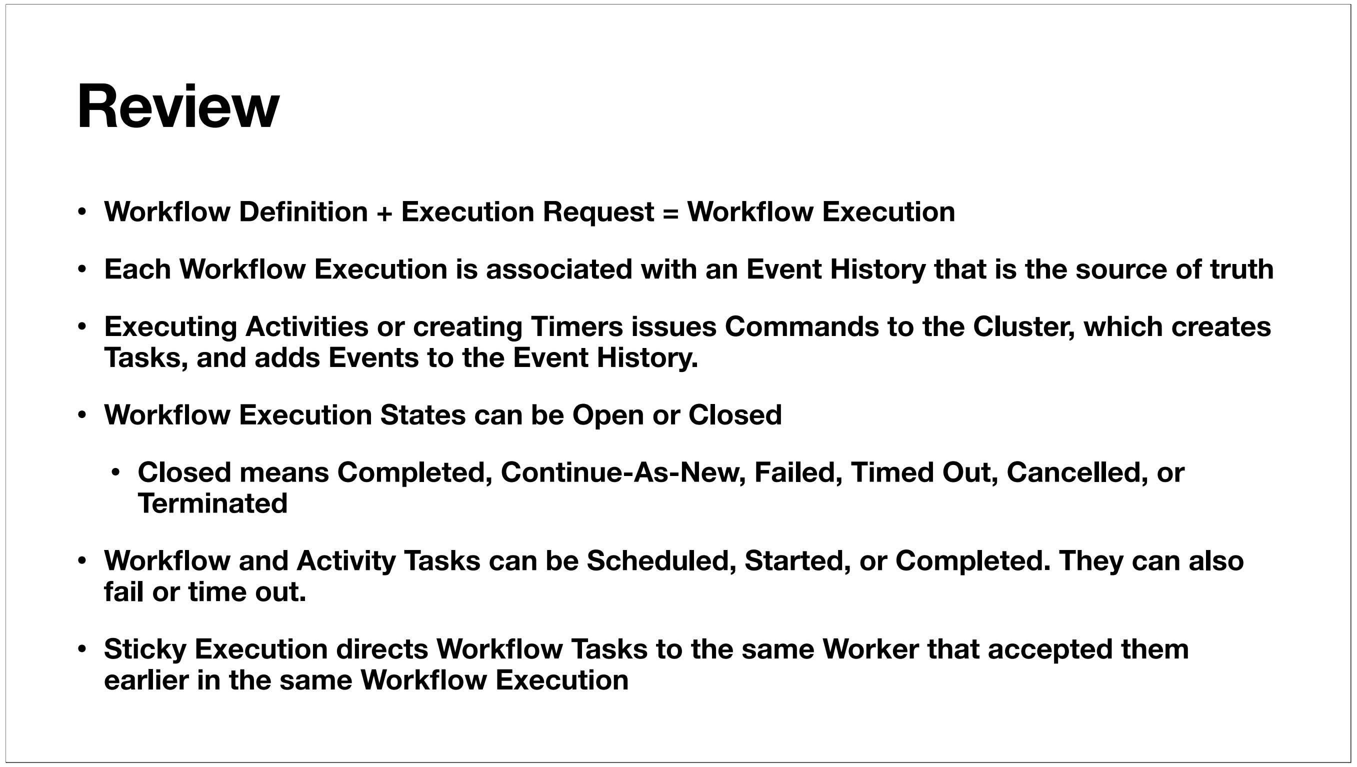
Task Queue Name twwmbp:b7b2434d-4fb5-4ca6-b05f-bb98d6565a96

Task Queue Kind Sticky

Task Queue Normal Name durable-exec-tasks

Workers cache the state of the Workflow functions they execute. To make this caching more effective, Temporal employs a performance optimization known as "Sticky Execution," which directs Workflow Tasks to the same Worker that accepted them earlier in the same Workflow Execution.

Note that Sticky Execution only applies to Workflow Tasks. Since Event History is associated with a Workflow, the concept of Sticky Execution is not relevant to Activity 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 Cluster, 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

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 Cluster, 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

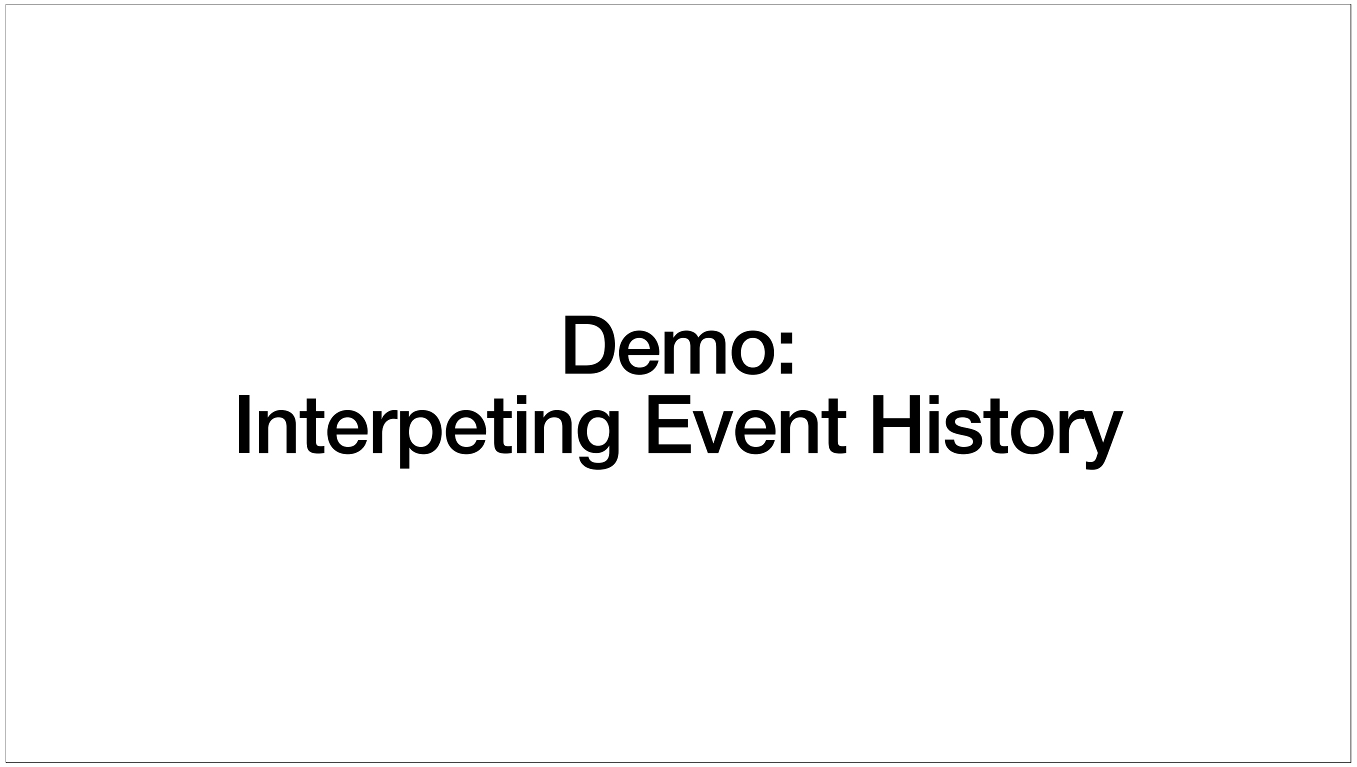
Temporal 102

00. About this Workshop
01. Understanding Key Concepts in Temporal
02. Improving Your Temporal Application Code
03. Using Timers in a Workflow Definition
04. Testing Your Temporal Application Code
05. Understanding Event History
▶ **06. Debugging Workflow History**
07. Deploying Your Application to Production
08. Understanding Workflow Determinism
09. Conclusion

Demo:
Debugging a Workflow that Doesn’t Progress

Demo
https://github.com/temporalio/edu-102-java-content/blob/chapters-6-and-7/debugging-workflow-execution/assets/video-production/demo-workflow-does-not-progress.md

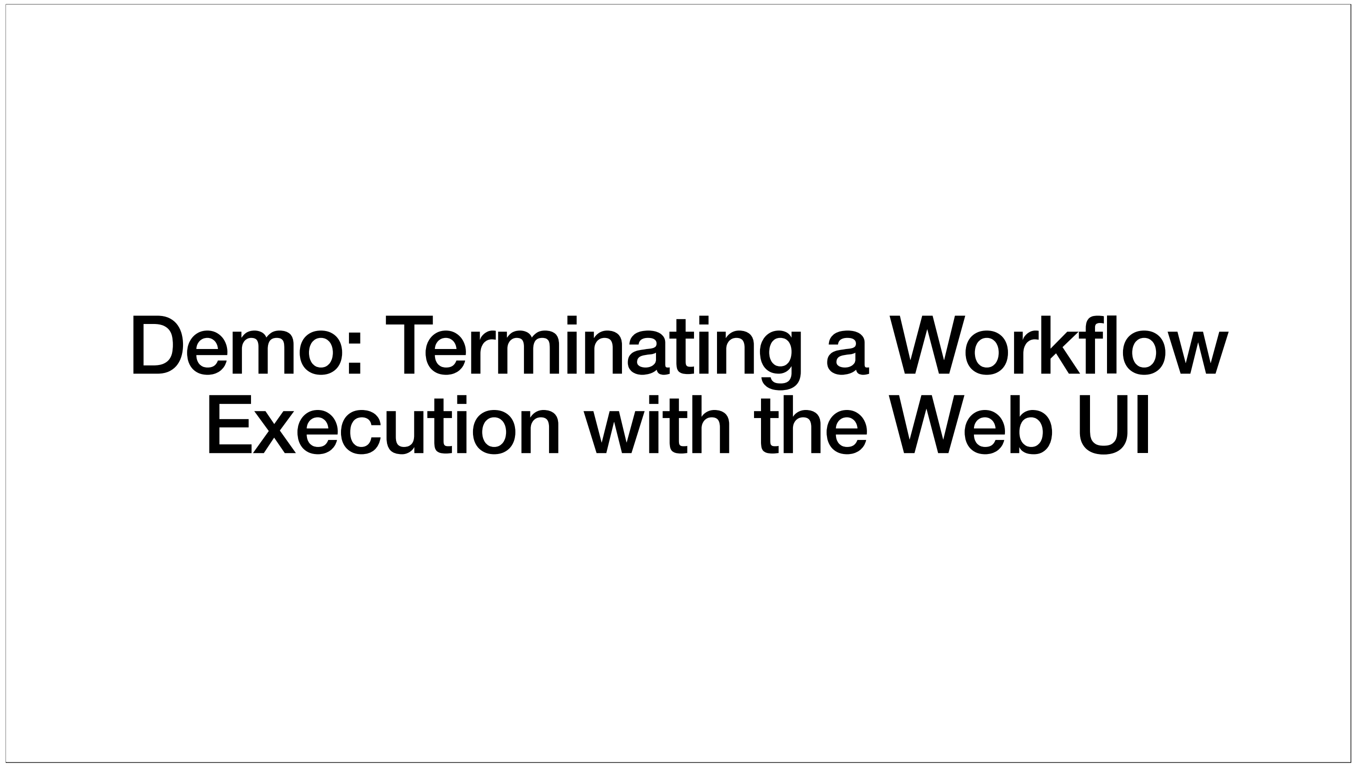
Scenario: Workflow started, but no Workers running
Example used: Translation Workflow (exercises/testing-code/solution)



Demo:
Interpeting Event History

Scenario: A tour of the Web UI and how to interpret Events
Example used: Translation Workflow (exercises/testing-code/solution)

https://github.com/temporalio/edu-102-java-content/blob/chapters-6-and-7/debugging-workflow-execution/assets/video-production/demo-interpret-event-history.md

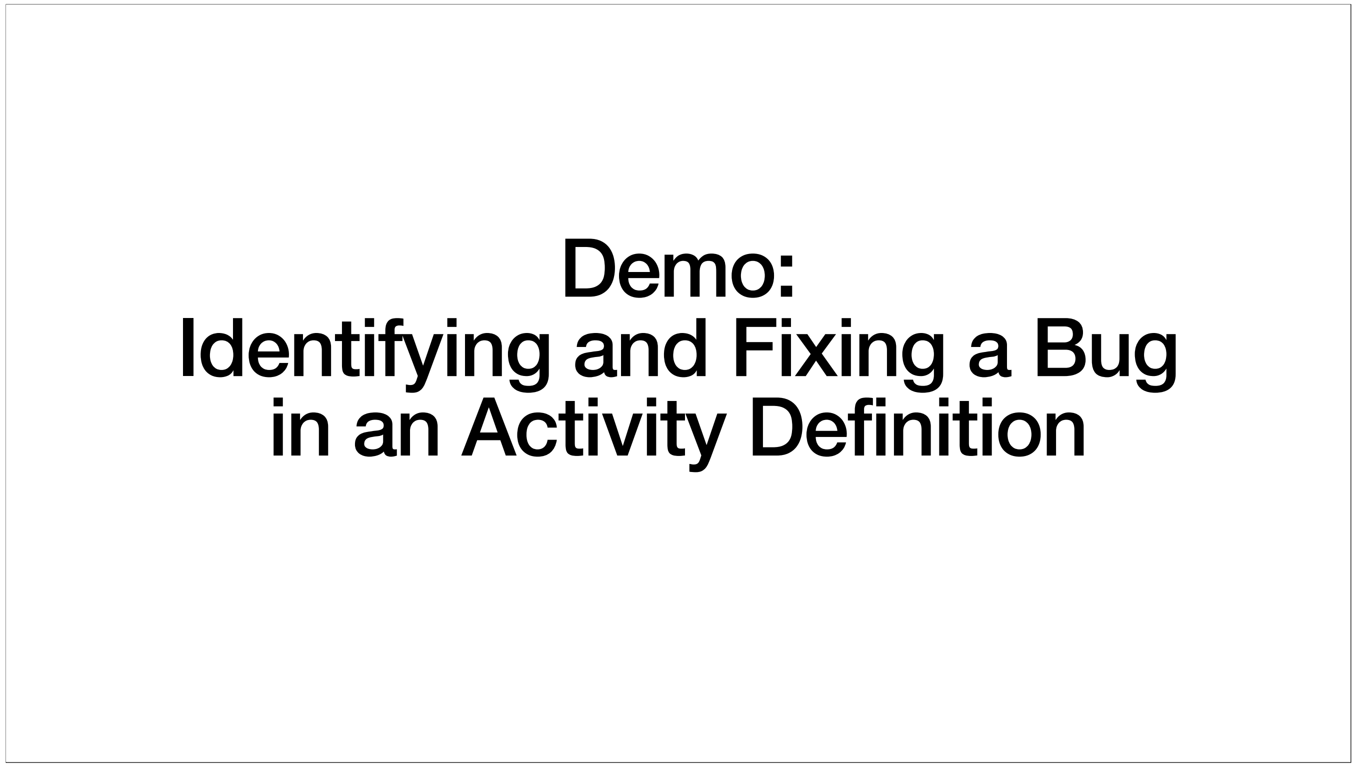


Demo: Terminating a Workflow Execution with the Web UI

* **Scenario**: Worker and translation microservice are running, but the Workflow has not yet been started. We’ll start the Workflow with invalid input that is passed to the Activity, thereby resulting in a perpetual of Activity execution attempt failures, so we terminate the Workflow, allowing us to run it again with the correct input.

* **Example used**: Translation Workflow (`exercises/testing-code/solution`)

https://github.com/temporalio/edu-102-java-content/blob/chapters-6-and-7/debugging-workflow-execution/assets/video-production/demo-terminate.md



Demo:
Identifying and Fixing a Bug in an Activity Definition

* **Scenario**: I make a small change to the Workflow Definition and am running it to see it in action. Unfortunately, someone on my team introduced a bug in the Activity definition, which I discover while running the Workflow.

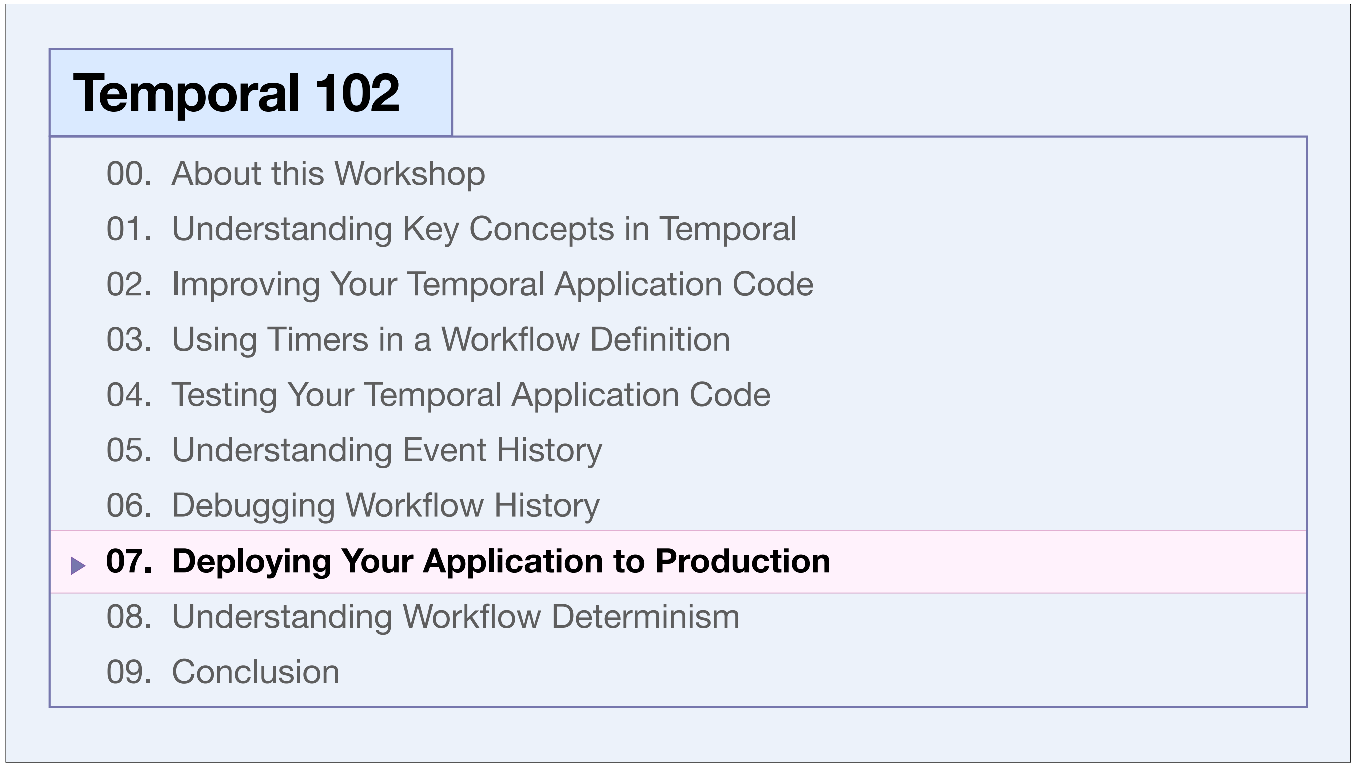
* **Example used**: Translation Workflow (`exercises/testing-code/solution`)

https://github.com/temporalio/edu-102-java-content/blob/chapters-6-and-7/debugging-workflow-execution/assets/video-production/demo-identify-fix-activity-bug.md

Exercise #4: 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

Exercise #4: Debugging and Fixing an Activity Failure SOLUTION



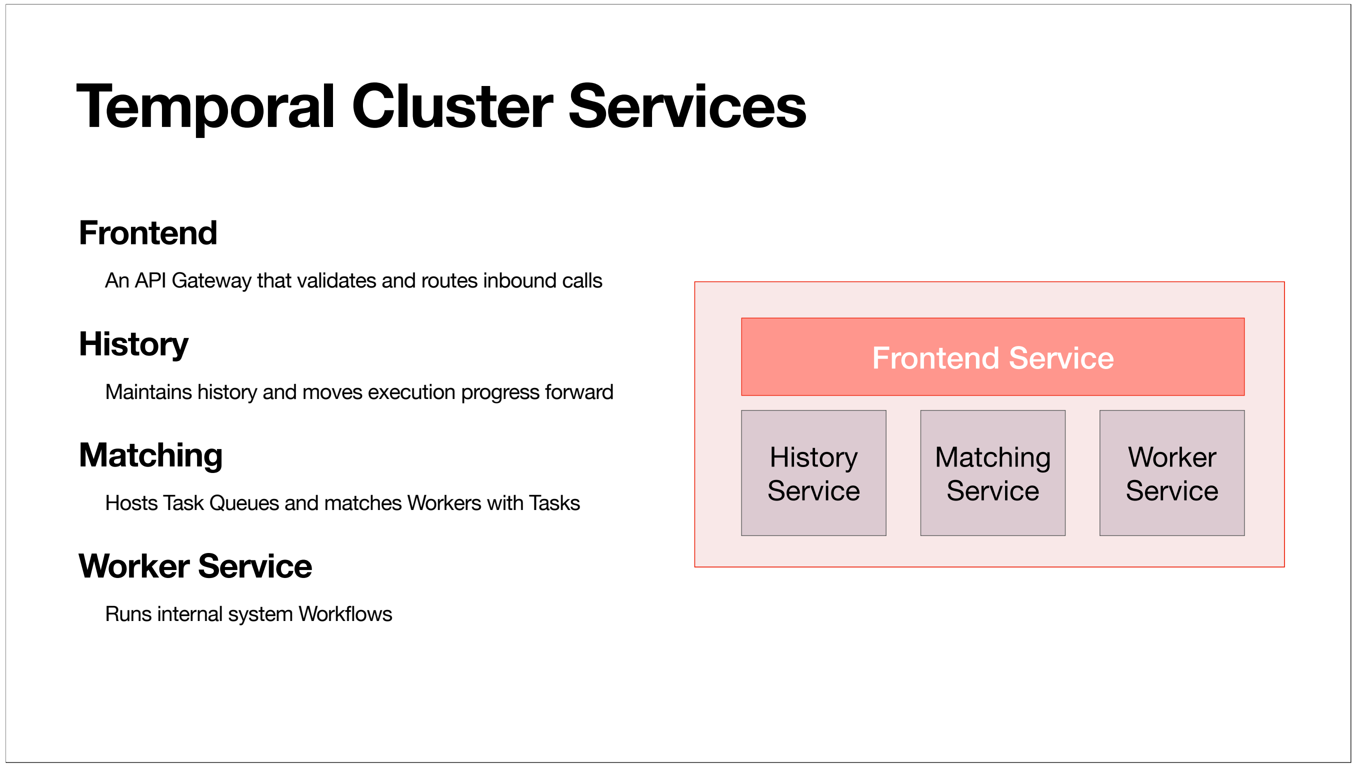
Temporal 102

00. About this Workshop
01. Understanding Key Concepts in Temporal
02. Improving Your Temporal Application Code
03. Using Timers in a Workflow Definition
04. Testing Your Temporal Application Code
05. Understanding Event History
06. Debugging Workflow History

▶ **07. Deploying Your Application to Production**

08. Understanding Workflow Determinism
09. Conclusion

We have so far depicted the Temporal Server as having a Frontend Service and a set of backend services. A developer doesn't usually require detailed knowledge of the server architecture, but it is important to understand how Temporal scales to support the needs of your applications in production.



Temporal Cluster Services

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

Worker Service

Runs internal system Workflows

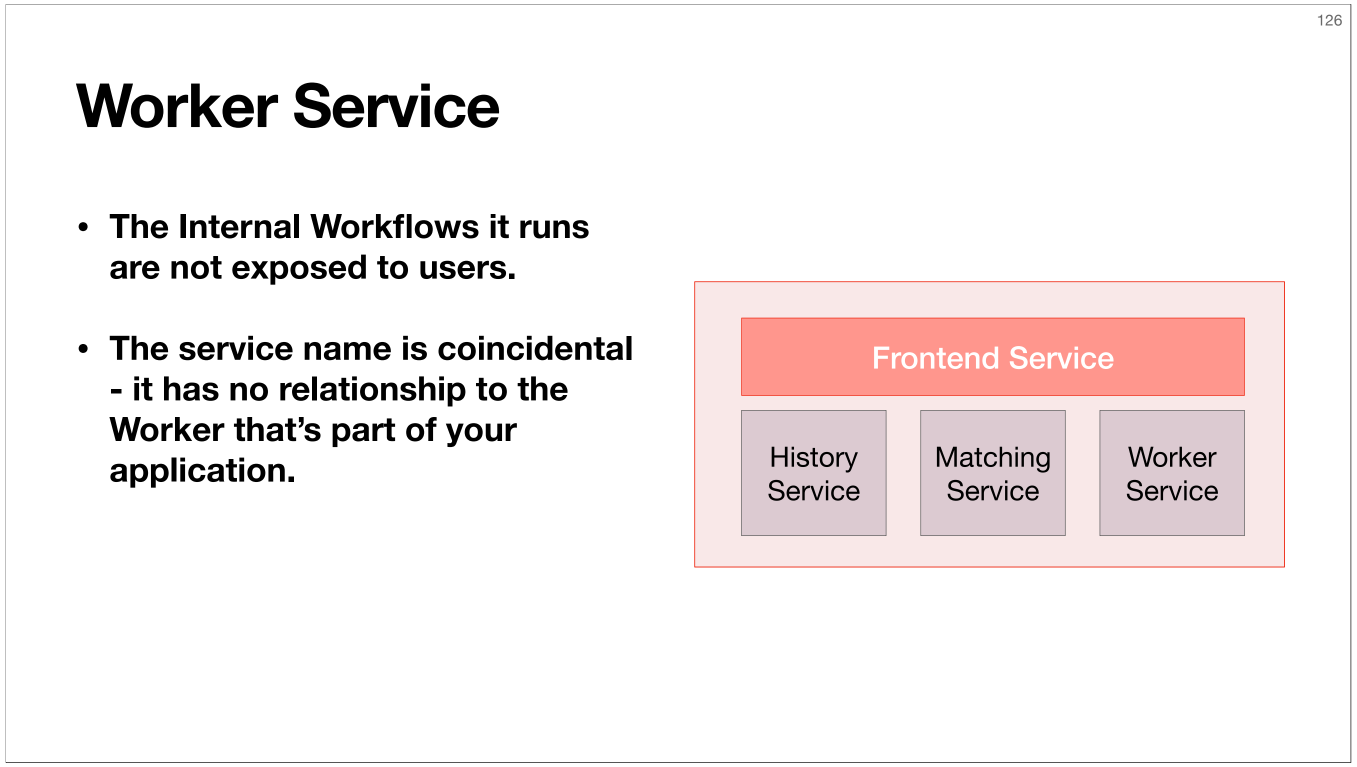
The cluster has a frontend API gateway that validates and routes inbound calls to other services.

What was previously labeled "Backend Services" is actually a set of three services.

The History Service, as the name implies, maintains the history of Workflow Executions by persisting their state. However, it is also the service responsible for moving the progress of Workflow Executions forward by initiating Workflow and Activity Tasks.

It works closely with the Matching Service, which hosts the Task Queue and matches tasks to polling Workers.

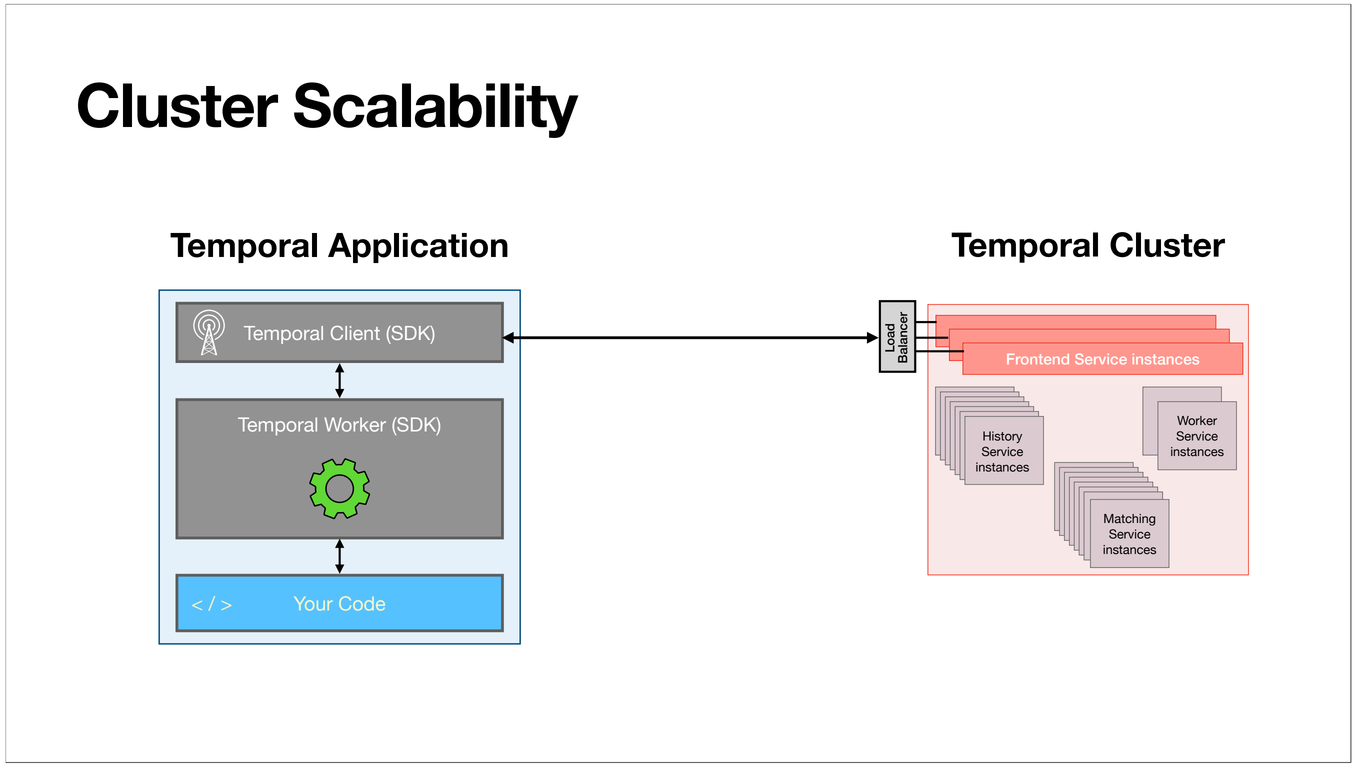
Finally, the Worker Service runs Workflow that are internal to the system, such as those used for replication or archiving old data.



Worker Service

- 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.

I want to make two important points regarding the Worker service. First, the internal Workflows that it runs are not exposed to users; you won't see them listed, for example, in the Web UI nor in the output of the Temporal commandline tool. Second, the service name suggests a relationship to the Worker that is part of your Temporal application, but this is coincidental, so take care not to confuse the two.

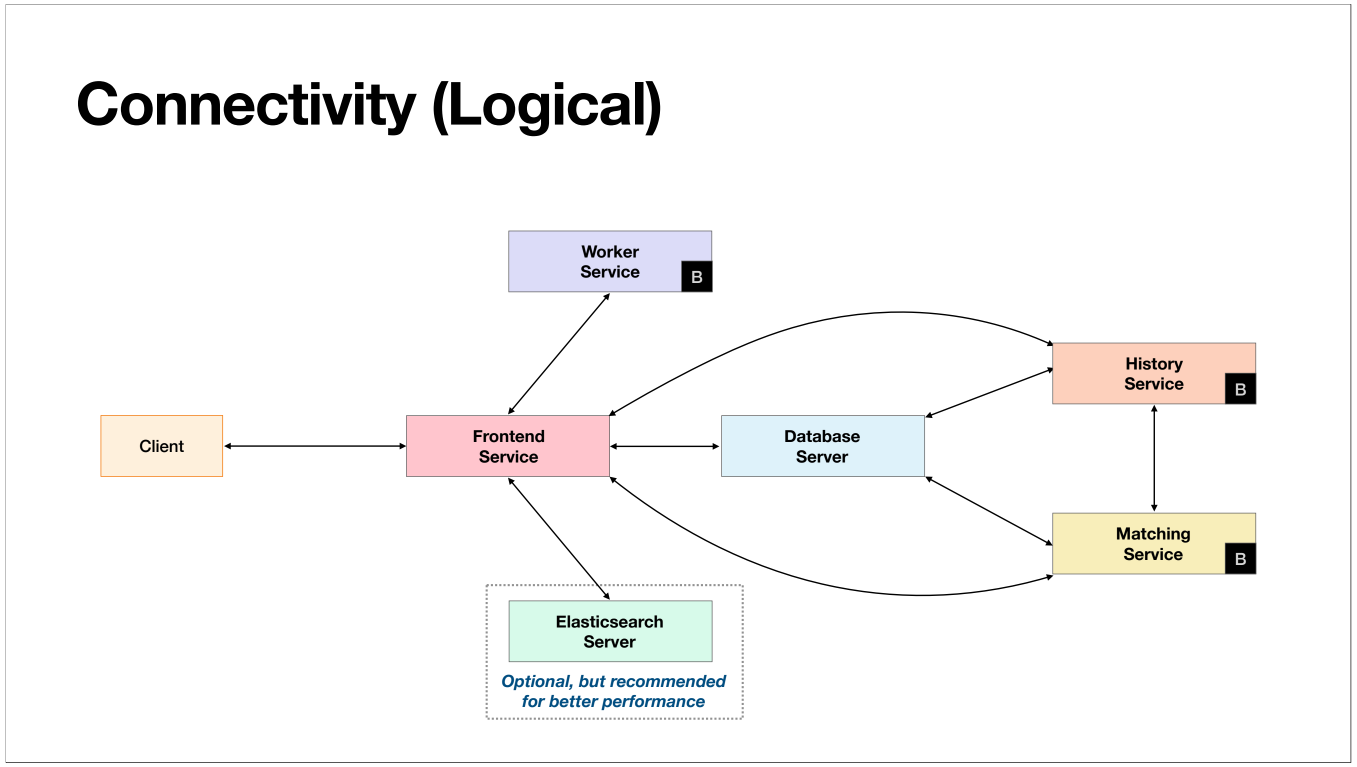


Cluster Scalability

However, a Temporal Cluster can scale well beyond that, with multiple instances of each service. Production clusters often have dozens or even hundreds of instances of these services running, which provides availability because the cluster can continue operations even as some instances fail.

When running multiple Frontend Services, it is typical to use a load balancer or network ingress to distribute inbound traffic among the various Frontend Service instances. This approach provides clients with a single address to use when contacting the Frontend Service, thus eliminating the need to know how many Frontend Services are deployed or the addresses of those individual instances.

Each of these four services scales independently of the others, which means that operations teams can direct resources precisely where they're needed.



Connectivity (Logical)

Putting it all together, we can see the communication and connectivity between the services.

[advance]

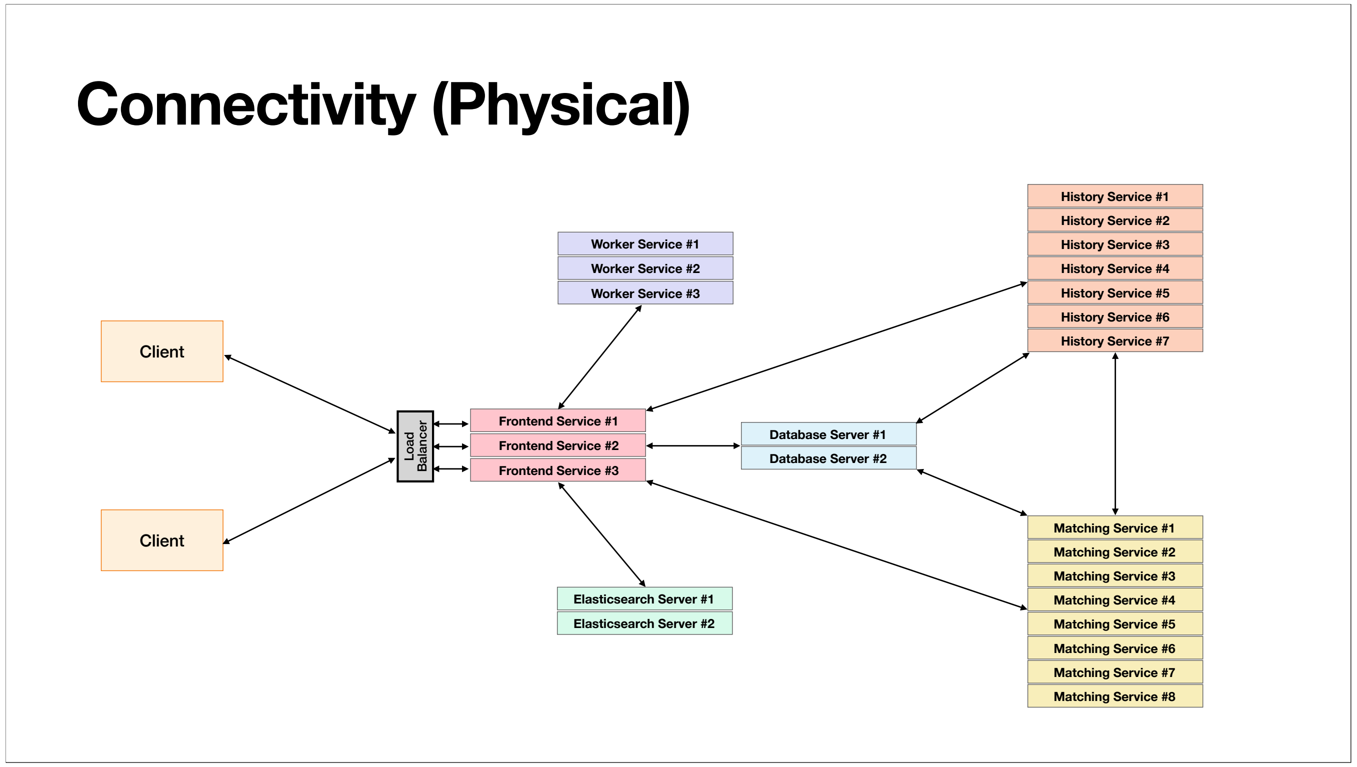
I've annotated the three backend services with a "B" and have also included the required database component and optional Elasticsearch server component.

Although Elasticsearch is optional, it is very much recommended for production clusters because it improves the performance of basic searches that help you to locate a specific Workflow Execution,

To the left of the Frontend Service is a client, such as the Temporal Client inside a Worker that executes your code or a Temporal Client in another part of your application that starts a Workflow and retrieves its result.

Clients do not not communicate with the backend services, nor do they access the cluster's other components, such as the database that it uses for persistence. This makes it easy to control access at the network level, since firewalls and other network hardware only need to pass inbound traffic to a single port.

You learned earlier that the communication between the Client and the Frontend Service uses gRPC, sending messages encoded using Protocol Buffers. This is also true of the communication between the Frontend and the backend services. As with communication between the Client and Frontend Service, this internode communication can also use TLS for enhanced security.

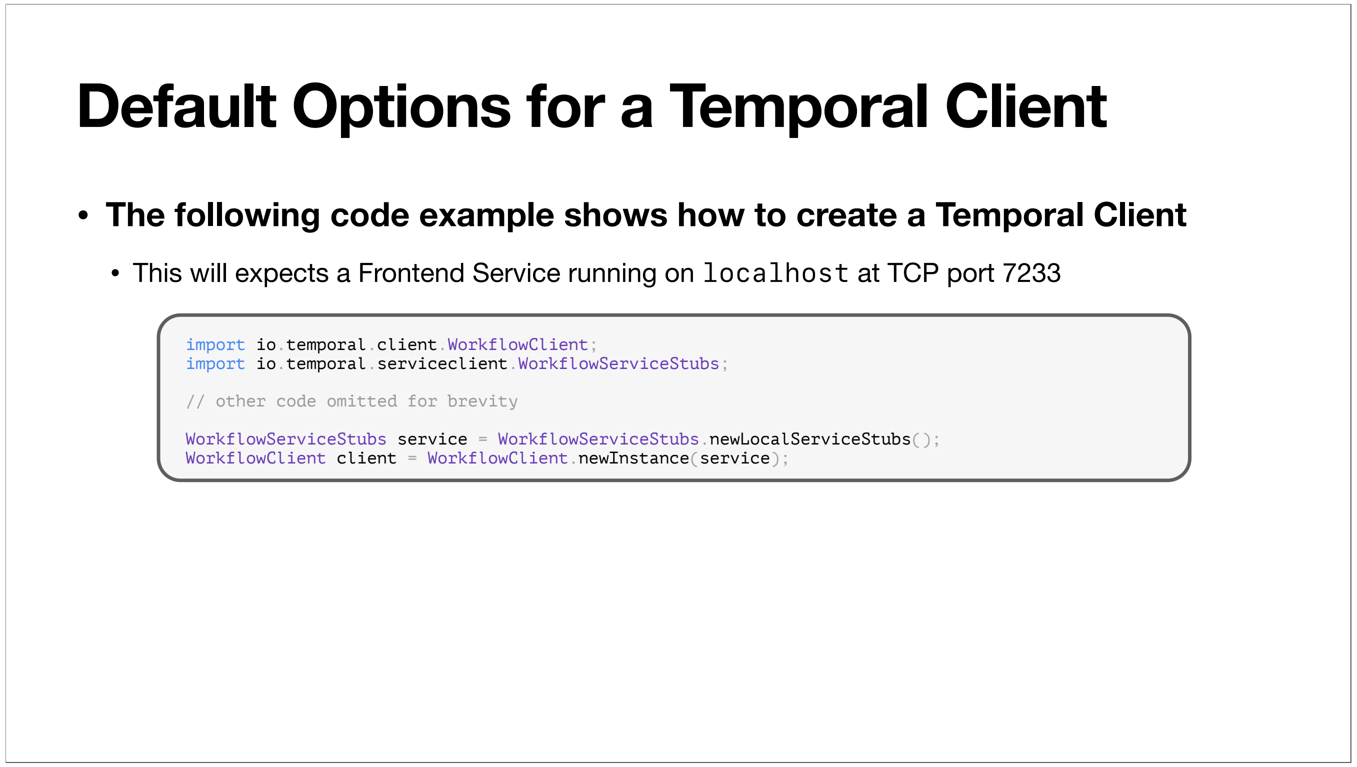


Connectivity (Physical)

Temporal Clusters used for production workloads can—and typically will—have multiple instances of each service. Here is an example of a production deployment, which illustrates the connectivity between different parts of the cluster as well as Temporal Clients that are external to the cluster.

As with the database, the load balancer isn't a component provided by Temporal. If you're running a self-hosted cluster on bare metal, you would likely use an actual piece of network hardware for load balancing. If you're deploying a self-hosted cluster on cloud infrastructure, then this will likely be part of the virtual network infrastructure, such as an ingress in Kubernetes.

Although many users choose to self-host the database server instances for their clusters, others prefer to use cloud provider's database hosting service, such as the AWS Relational Database Service (RDS).



Default Options for a Temporal Client

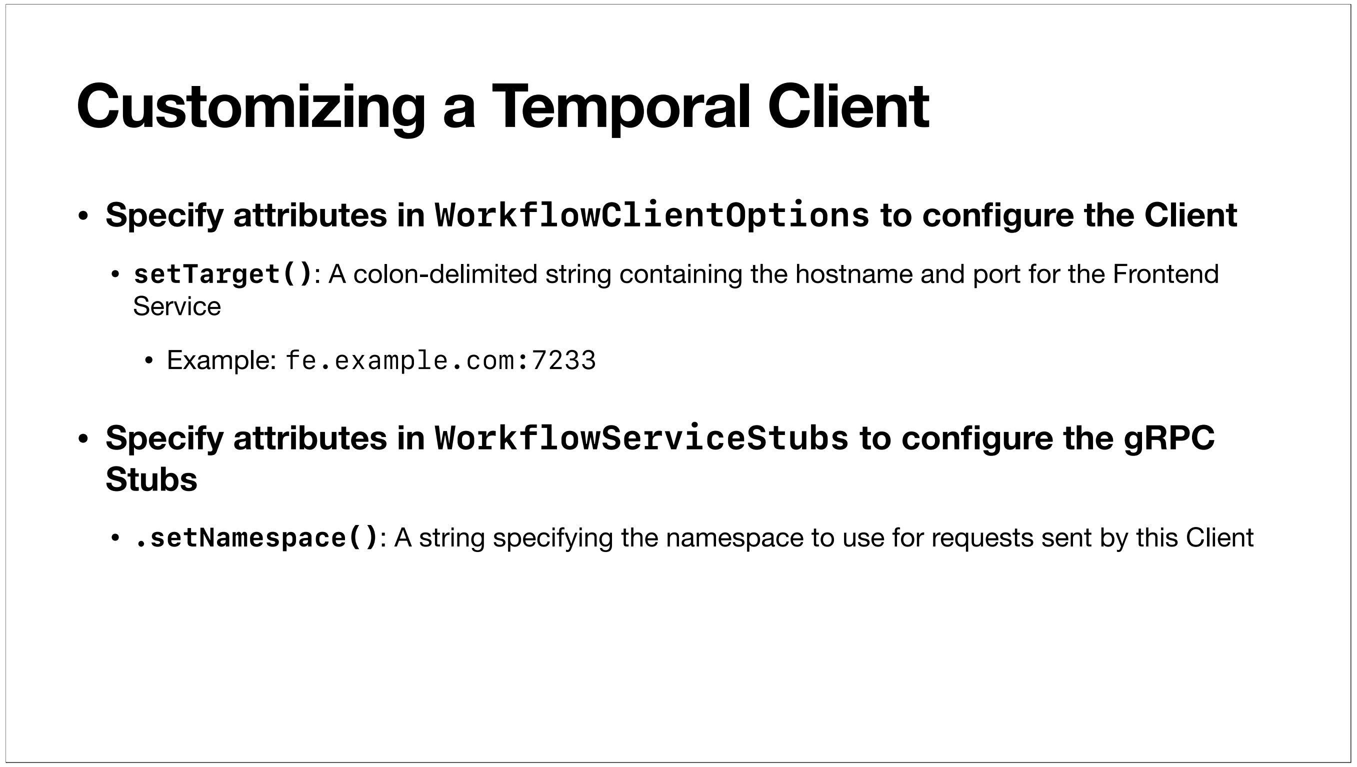
- **The following code example shows how to create a Temporal Client**
  - This will expects a Frontend Service running on `localhost` at TCP port 7233

```
import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;

// other code omitted for brevity

WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
WorkflowClient client = WorkflowClient.newInstance(service);
```

As you learned in a previous section, Temporal Clients communicate with the Temporal Cluster via gRPC. gRPC uses stubs, a client-side representation of a remote service, generated from a gRPC service definition, as configuration to be used by the Temporal Client.

Temporal provides the newLocalServiceStubs method as a helper method to create the gRPC stubs necessary to configure a connection to a Temporal Cluster running on localhost.

As you can see in the code above, you first create WorkflowServiceStubs object to store the gRPC stubs configuration, then pass that to the WorkflowClient. The Client is created with the default options and uses the default namespace. It also expects to find a Temporal Frontend Service running on localhost (unless otherwise specified) and accepting connections on the default port for that service (TCP port 7233).



Customizing a Temporal Client

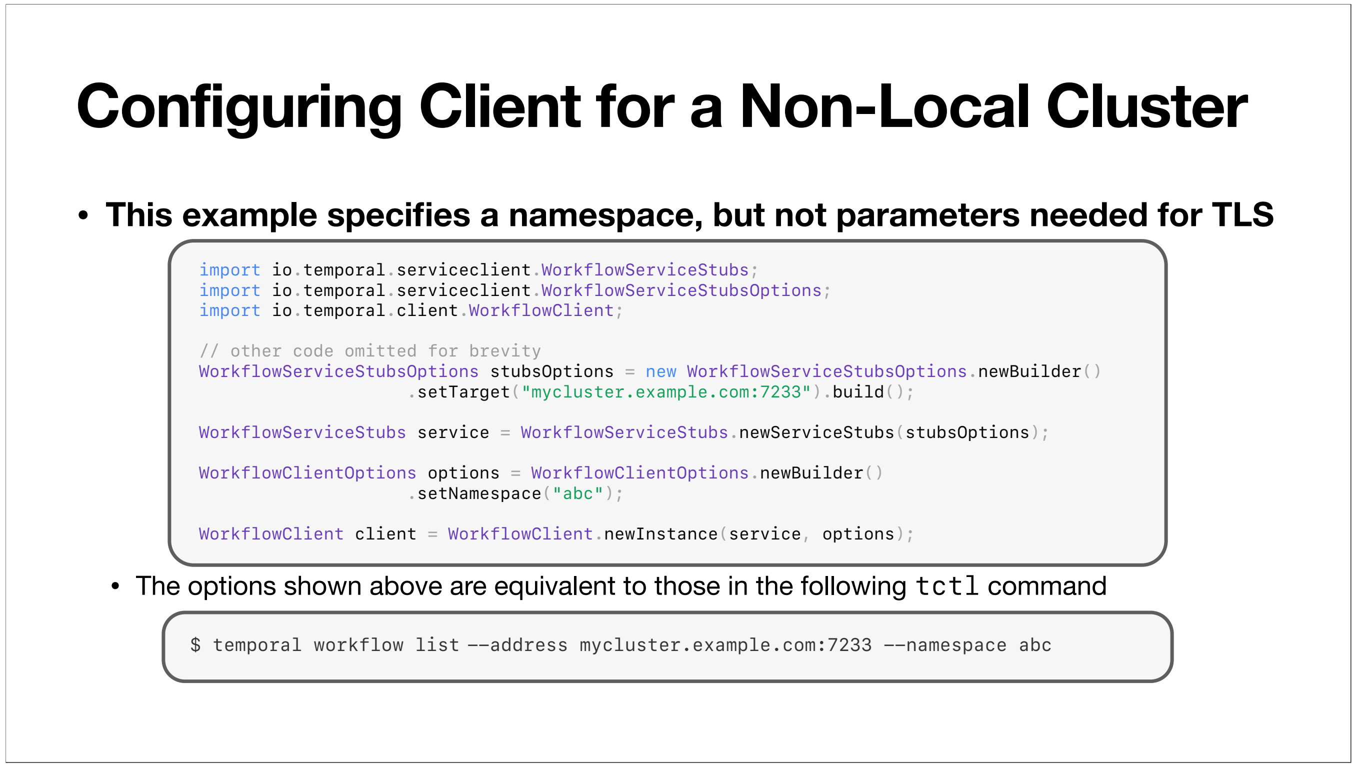
- **Specify attributes in `WorkflowClientOptions` to configure the Client**
  - **`setTarget()`**: A colon-delimited string containing the hostname and port for the Frontend Service
    - Example: `fe.example.com:7233`
- **Specify attributes in `WorkflowServiceStubs` to configure the gRPC Stubs**
  - **`.setNamespace()`**: A string specifying the namespace to use for requests sent by this Client

The newLocalServiceStubs configuration is often suitable for development, where it's common to run a Temporal server locally, but not appropriate for a production deployment.

In production, the Frontend Service will be running on a different system, perhaps a physical machine, virtual machine, or container, which you'll access using a hostname specific to that system. A cluster with multiple Frontend Services will typically by fronted by a load balancer that distributes incoming requests among them, while providing clients with a single IP address or hostname to use.

You can customize the default options by specifying various attributes related to the configuration of the Workflow Client using WorkflowClientOptions when creating the Client.

You can also customize various gRPC configurations that are in turn used when creating the Workflow Client by setting various WorkflowServiceStubs options when creating the Client.



Configuring Client for a Non-Local Cluster

- **This example specifies a namespace, but not parameters needed for TLS**

```
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.serviceclient.WorkflowServiceStubsOptions;
import io.temporal.client.WorkflowClient;

// other code omitted for brevity
WorkflowServiceStubsOptions stubsOptions = new WorkflowServiceStubsOptions.newBuilder()
                    .setTarget("mycluster.example.com:7233").build();

WorkflowServiceStubs service = WorkflowServiceStubs.newServiceStubs(stubsOptions);

WorkflowClientOptions options = WorkflowClientOptions.newBuilder()
                    .setNamespace("abc");

WorkflowClient client = WorkflowClient.newInstance(service, options);
```

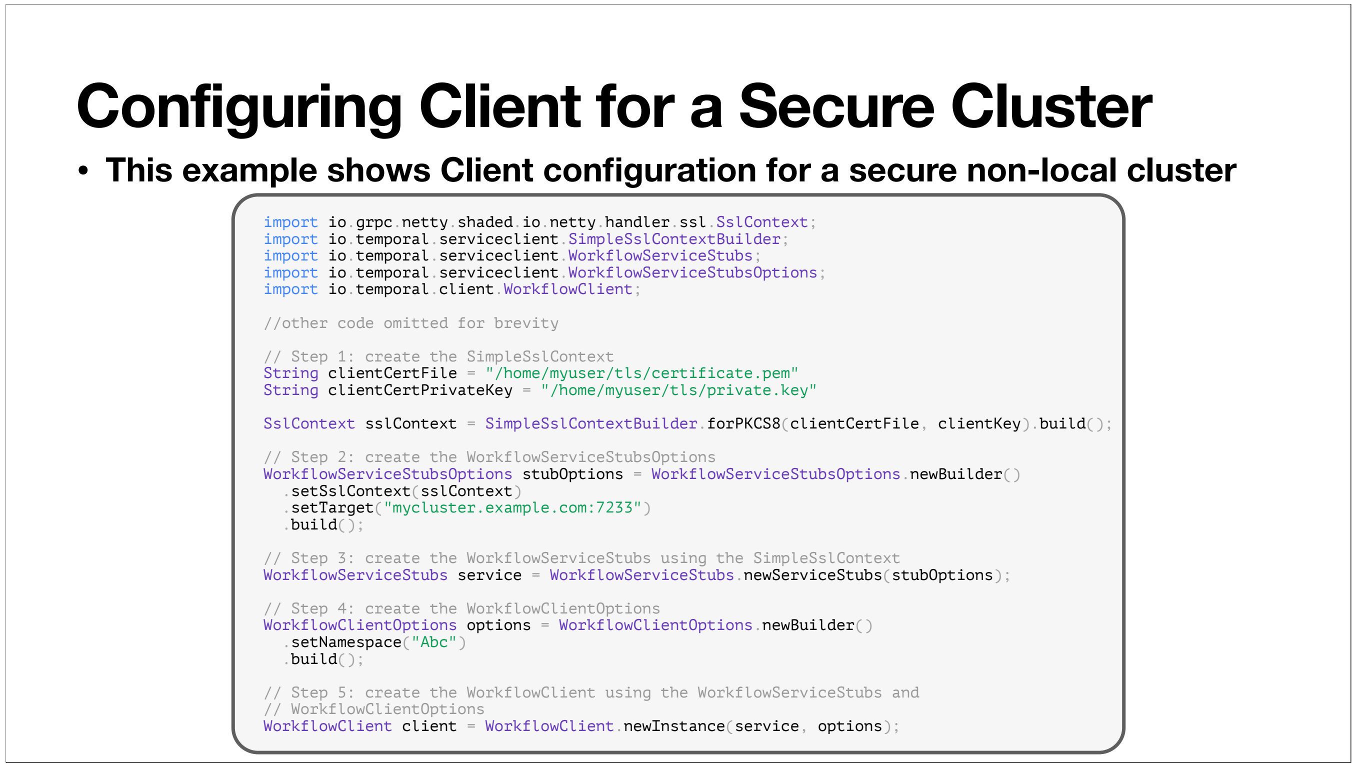
  - The options shown above are equivalent to those in the following `tctl` command

```
$ temporal workflow list --address mycluster.example.com:7233 --namespace abc
```

When connecting to a Temporal Cluster hosted remotely, whether it is self-hosted or Temporal Cloud, you will use the newServiceStubs method and pass in your configuration as a WorkflowServiceStubsOptions object.

The following example shows how to use WorkflowServiceStubsOptions and WorkflowClientOptions to customize the Client so that it will send its requests to a non-local cluster. This example also specifies a namespace to use for its requests, but does not specify parameters related to TLS, so it is not compatible with a cluster that's configured to require secure connections

Recall that the temporal command-line tool is another Temporal client and its default options also assume that it can contact the Frontend Service running on localhost at TCP port 7233. If you need to use temporal, you must configure it with the same options used to create a Client through code.



Configuring Client for a Secure Cluster

- **This example shows Client configuration for a secure non-local cluster**

```java
import io.grpc.netty.shaded.io.netty.handler.ssl.SslContext;
import io.temporal.serviceclient.SimpleSslContextBuilder;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.serviceclient.WorkflowServiceStubsOptions;
import io.temporal.client.WorkflowClient;

//other code omitted for brevity

// Step 1: create the SimpleSslContext
String clientCertFile = "/home/myuser/tls/certificate.pem"
String clientCertPrivateKey = "/home/myuser/tls/private.key"

SslContext sslContext = SimpleSslContextBuilder.forPKCS8(clientCertFile, clientKey).build();

// Step 2: create the WorkflowServiceStubsOptions
WorkflowServiceStubsOptions stubOptions = WorkflowServiceStubsOptions.newBuilder()
   .setSslContext(sslContext)
   .setTarget("mycluster.example.com:7233")
   .build();

// Step 3: create the WorkflowServiceStubs using the SimpleSslContext
WorkflowServiceStubs service = WorkflowServiceStubs.newServiceStubs(stubOptions);

// Step 4: create the WorkflowClientOptions
WorkflowClientOptions options = WorkflowClientOptions.newBuilder()
   .setNamespace("Abc")
   .build();

// Step 5: create the WorkflowClient using the WorkflowServiceStubs and
// WorkflowClientOptions
WorkflowClient client = WorkflowClient.newInstance(service, options);
```

Temporal Cloud, as well as most self-hosted clusters intended for production use, require the use of TLS for security.

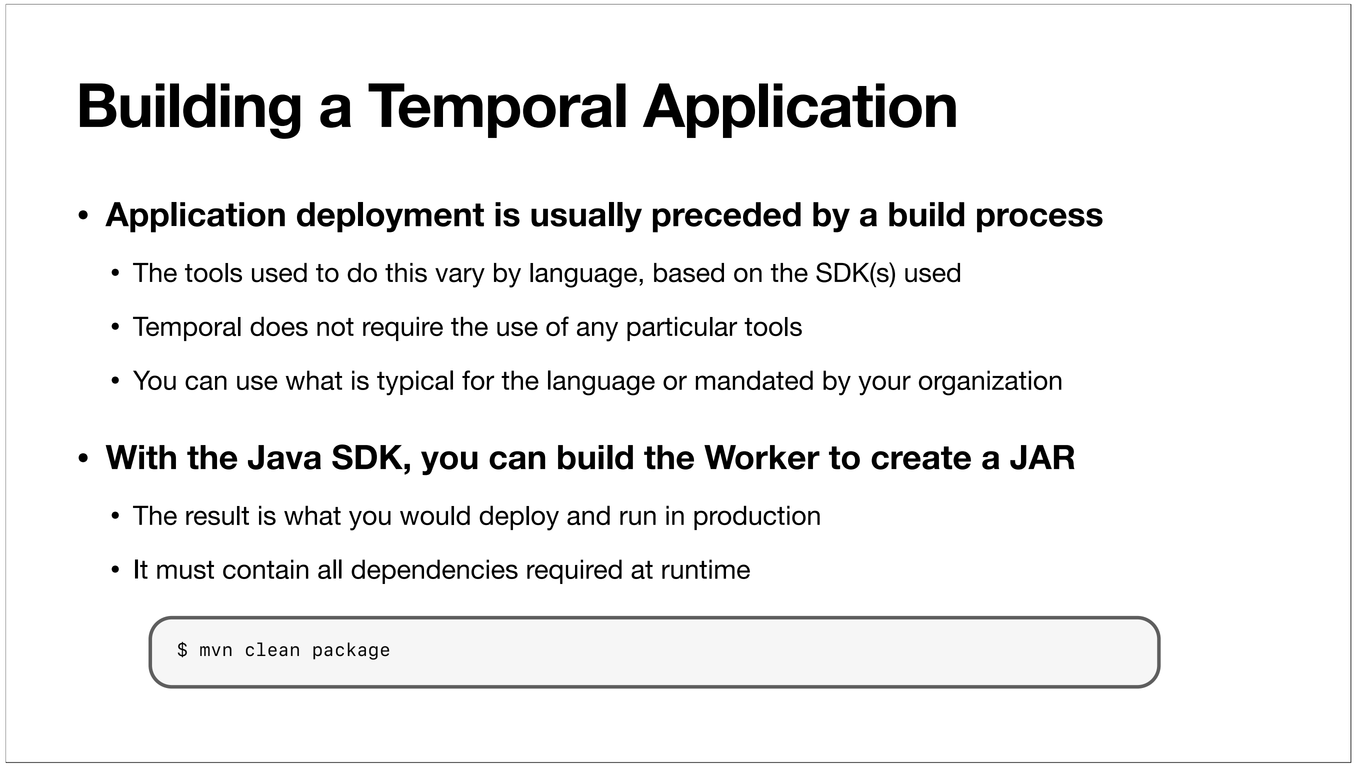
TLS, short for Transport Layer Security, is the modern version of a cryptographic protocol formerly known as SSL, which is widely known as the mechanism used to secure communication between Web browsers and Web servers. As with these e-commerce use cases, TLS protects communication between the Client and Frontend service from eavesdropping as it's transmitted across the network.

Because the certificates used in TLS connections represent the identity of the parties involved in that communication, these can also be used for authentication, controlling access based on the certificate. Temporal supports TLS with mutual authentication, often abbreviated as mTLS, which means that it can validate the identity of both the client and server ends of the connection.

Using TLS requires that you have a certificate and a corresponding private key. These will likely either be provided to you by your administrator or you will create them yourself using instructions provided by the administrator.

The Temporal Java SDK provides a class, SimpleSslContextBuilder, for handling the context around the SSL key pairs. This class builds an gRPC SslContext object from the io.grpc.netty package and is a dependency withing the Temporal package, so you will not need to manually include it in your pom.xml.

You then create the WorkflowServiceStubsOptions as before, but call the setSslContext method in the builder context, passing in the context created previously.



Building a Temporal Application

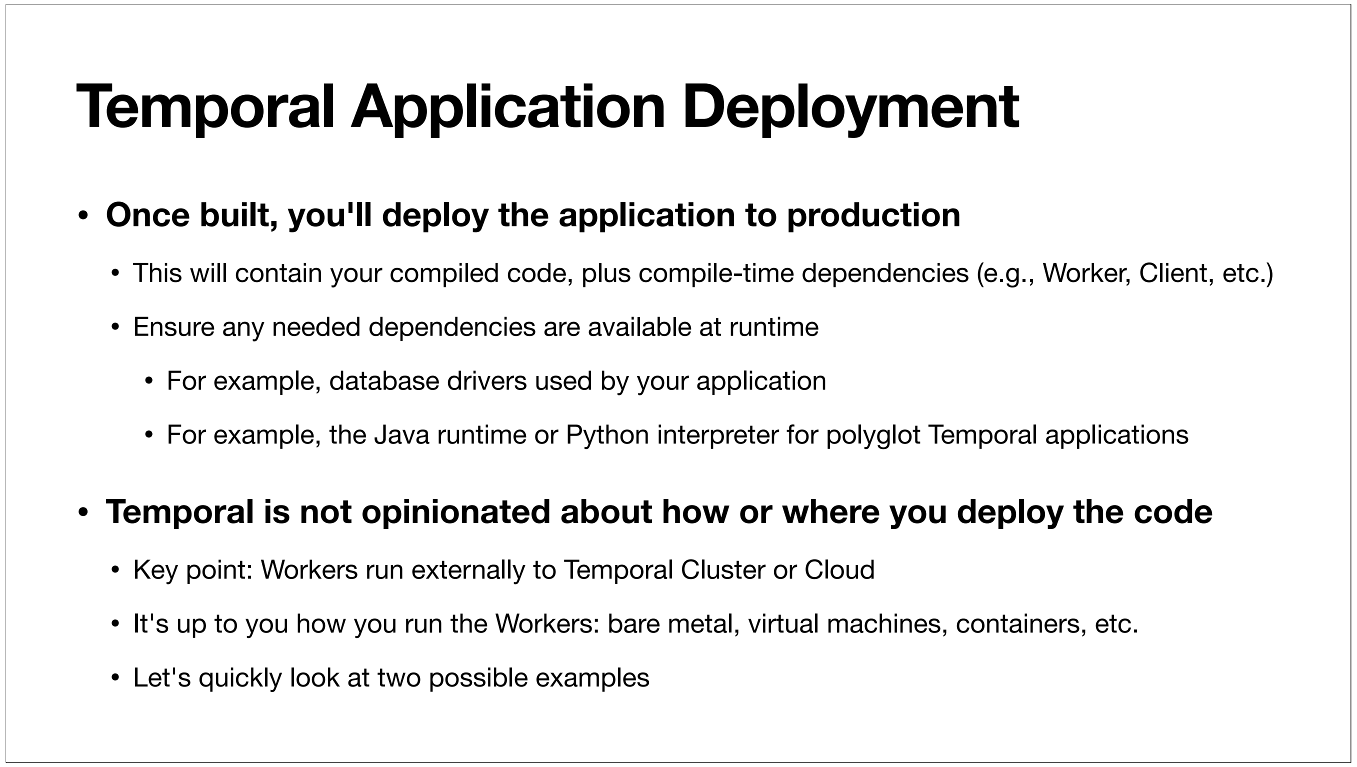
- **Application deployment is usually preceded by a build process**
  - The tools used to do this vary by language, based on the SDK(s) used
  - Temporal does not require the use of any particular tools
  - You can use what is typical for the language or mandated by your organization
- **With the Java SDK, you can build the Worker to create a JAR**
  - The result is what you would deploy and run in production
  - It must contain all dependencies required at runtime

```
$ mvn clean package
```

Temporal supports “polyglot” development, so different parts of a single application can be implemented in different languages. Your application must have at least one Worker to execute the Workflow and Activity code corresponding to each SDK you use. For example, you could use a Client provided by the TypeScript SDK to start a Workflow that is written in Go and which calls an Activity that’s written in Java. In this case, you would need at least two Workers, one to run the Workflow (Go) and another to run the Activity (Java). However, we recommend running multiple processes of each, potentially spread across multiple servers, to improve the scalability and availability of the application.

The methods used to build the application will vary based on the SDKs used. Temporal does not specify or require any particular tools for this, so developers are free to use whatever is commonly used for that language and/or is mandated by their organizations. As with other types of applications, many developers use Continuous Integration (CI) tools, such as Jenkins, BuildKit, or CircleCI, to automate building and testing their Temporal applications.

With Java, it's typical to compile the Worker into a jar file that you can deploy to production. Since Workflows and Activities are registered with the Worker, the executable will include those functions, and should also include any dependencies referenced in your code. The output from this build is what you'll deploy and run.



Temporal Application Deployment

- **Once built, you'll deploy the application to production**
  - This will contain your compiled code, plus compile-time dependencies (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 Cluster 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

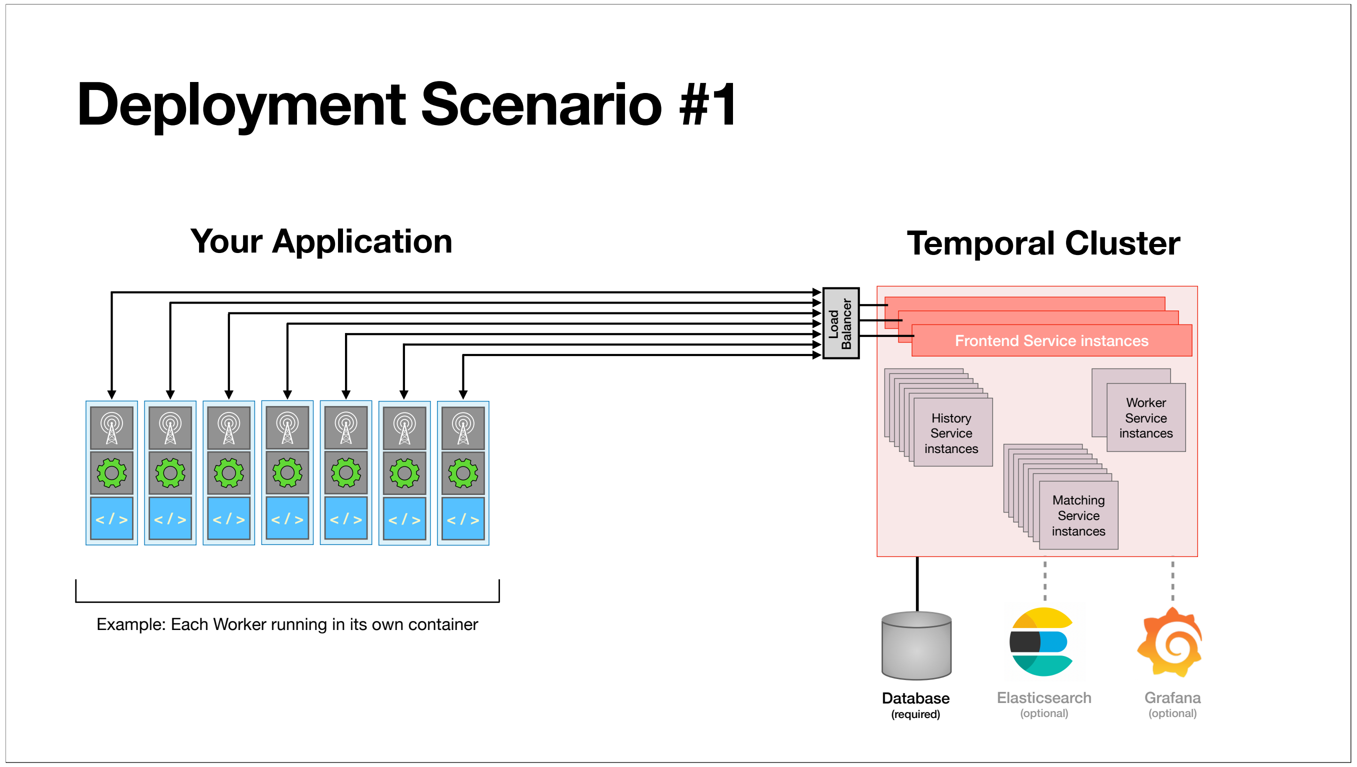
People often speak of "deploying Workers" to production, but in reality, what you'll deploy is everything that's necessary for running a Worker Process.

That will include artifacts compiled from code you write; for example, the Workflow Definition, Activity Definitions, and the Worker configuration, but also all of the dependencies used by that code. Those dependencies include the Temporal SDK, as well as any other libraries your code might use, which will vary from one app to the next, but might include things such as database drivers or clients for any services that your Activities might call.

Additionally, the system on which the application runs, which might be a physical server, virtual machine, or container, must have the software needed to run an application written in that language. For example, a Worker that executes Activities written in Java will require the Java Virtual Machine, while a Worker that executes Python code will require the Python interpreter.

Temporal is not opinionated about how or where you deploy the code. Workers run externally to Temporal Cluster 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

Your Application

Temporal Cluster

Load Balancer

Frontend Service instances

History Service instances

Worker Service instances

Matching Service instances

Example: Each Worker running in its own container

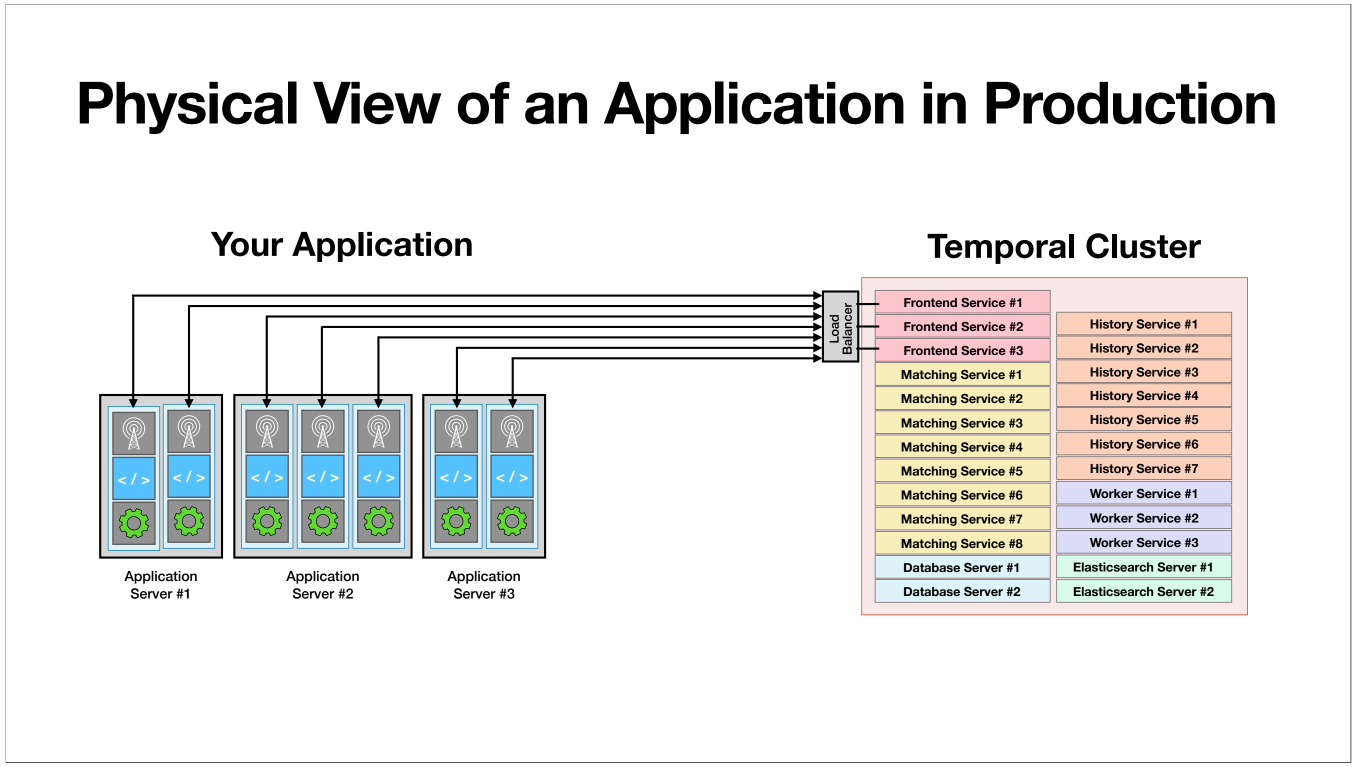
Database (required)

Elasticsearch (optional)

Grafana (optional)

Here's the logical view of an application in the context of a production system. There are multiple Worker Processes, which we collectively refer to as a *Worker fleet*. This runs on infrastructure you manage, which can take many forms, such as physical servers in your data center, virtual machines hosted by a cloud provider, or containers running inside of a Kubernetes cluster.

On the other side of the connection is the Temporal Cluster or Temporal Cloud.



Physical View of an Application in Production

Your Application

Temporal Cluster

Load Balancer

Frontend Service #1
Frontend Service #2
Frontend Service #3
Matching Service #1
Matching Service #2
Matching Service #3
Matching Service #4
Matching Service #5
Matching Service #6
Matching Service #7
Matching Service #8
Database Server #1
Database Server #2

History Service #1
History Service #2
History Service #3
History Service #4
History Service #5
History Service #6
History Service #7
Worker Service #1
Worker Service #2
Worker Service #3
Elasticsearch Server #1
Elasticsearch Server #2

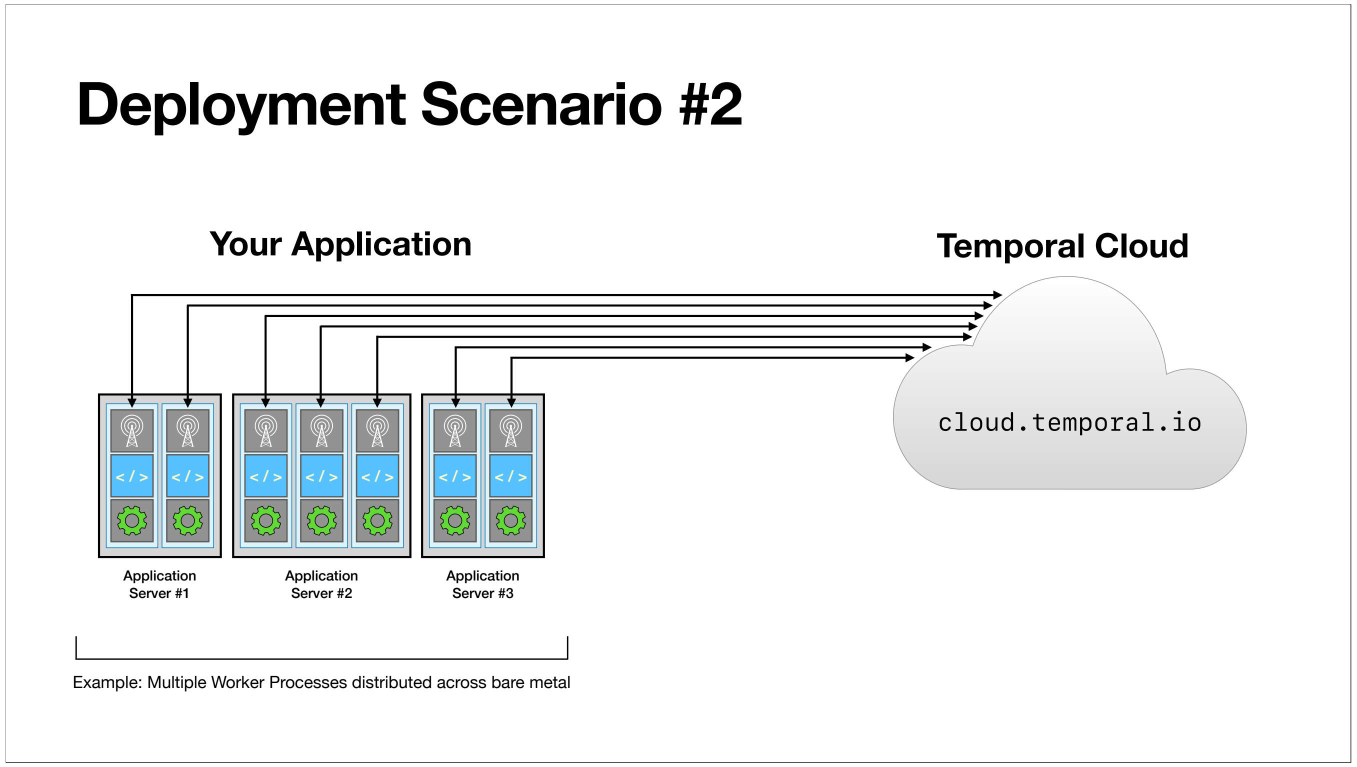
Application Server #1

Application Server #2

Application Server #3

Here's how that logical view maps to a physical deployment for a production system, which has multiple Worker Processes spread across an appropriate number of servers. Although I've labeled them “Application Servers" here, they might be physical machines, virtual machines, or containers.

Applications always run on servers you control and manage. They are external to the Temporal Cluster or Temporal Cloud service. If you're running a self-hosted cluster, you are responsible for the infrastructure needed for both the application and the cluster.



Deployment Scenario #2

Your Application

Temporal Cloud

cloud.temporal.io

Application Server #1

Application Server #2

Application Server #3

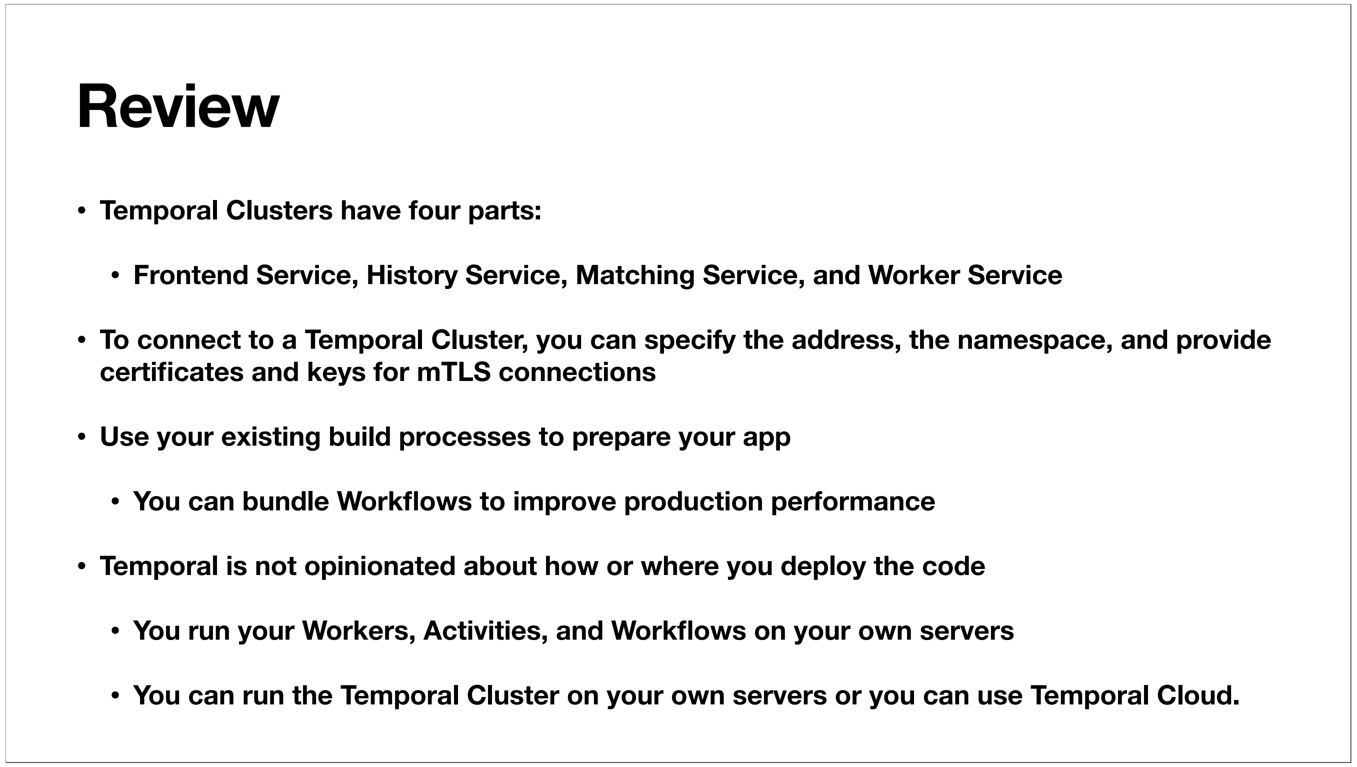
Example: Multiple Worker Processes distributed across bare metal

Temporal Cloud is an alternative to a self-hosted cluster. You're still responsible for running the application and the infrastructure it runs on, but we manage everything on the other end of the connection. With Temporal Cloud, your Clients have a single hostname and port to contact, representing a load-balanced Frontend Service.

When moving from a local development cluster to a self-hosted Cluster or Temporal Cloud, typically the only change you need to make is to the `ClientOptions` used to create your connection.

For example, it may specify a different hostname, a different namespace, and possibly some options related to security, such as the locations of a certificate and key.

The rest of your application code does not need to change as you move between these environments.



Review

- Temporal Clusters have four parts:
  - Frontend Service, History Service, Matching Service, and Worker Service
- To connect to a Temporal Cluster, 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 the Temporal Cluster on your own servers or you can use Temporal Cloud.

Temporal Clusters have four parts:
Frontend Service, History Service, Matching Service, and Worker Service
To connect to a Temporal Cluster, 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 the Temporal Cluster on your own servers or you can use Temporal Cloud.

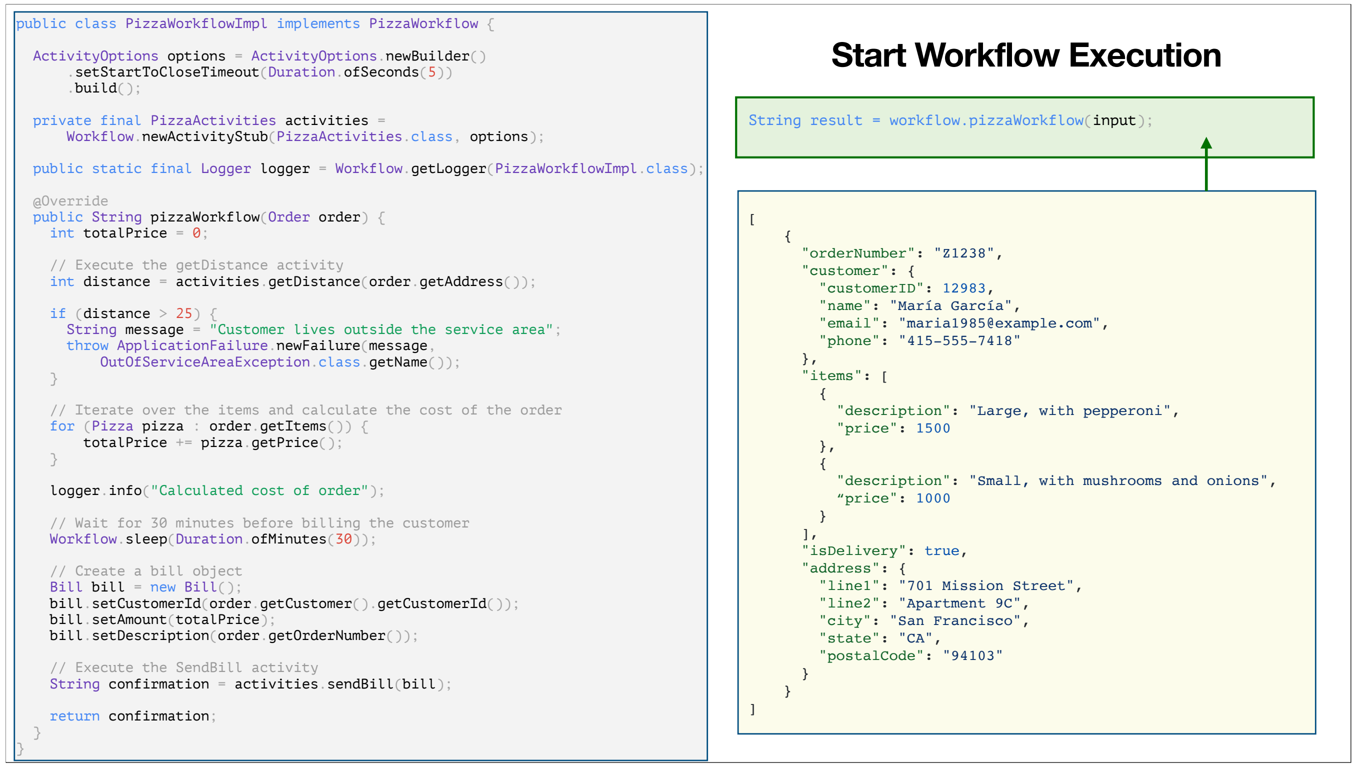
Temporal 102

00. About this Workshop
01. Understanding Key Concepts in Temporal
02. Improving Your Temporal Application Code
03. Using Timers in a Workflow Definition
04. Testing Your Temporal Application Code
05. Understanding Event History
06. Debugging Workflow History
07. Deploying Your Application to Production
▶ **08. Understanding Workflow Determinism**
09. Conclusion

History Replay:

How Temporal Provides Durable Execution

First, let’s look at how Temporal’s History Replay provides the durable execution we’ve been talking about.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Start Workflow Execution

```
String result = workflow.pizzaWorkflow(input);
```

```
[
    {
      "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"
      }
    }
]
```

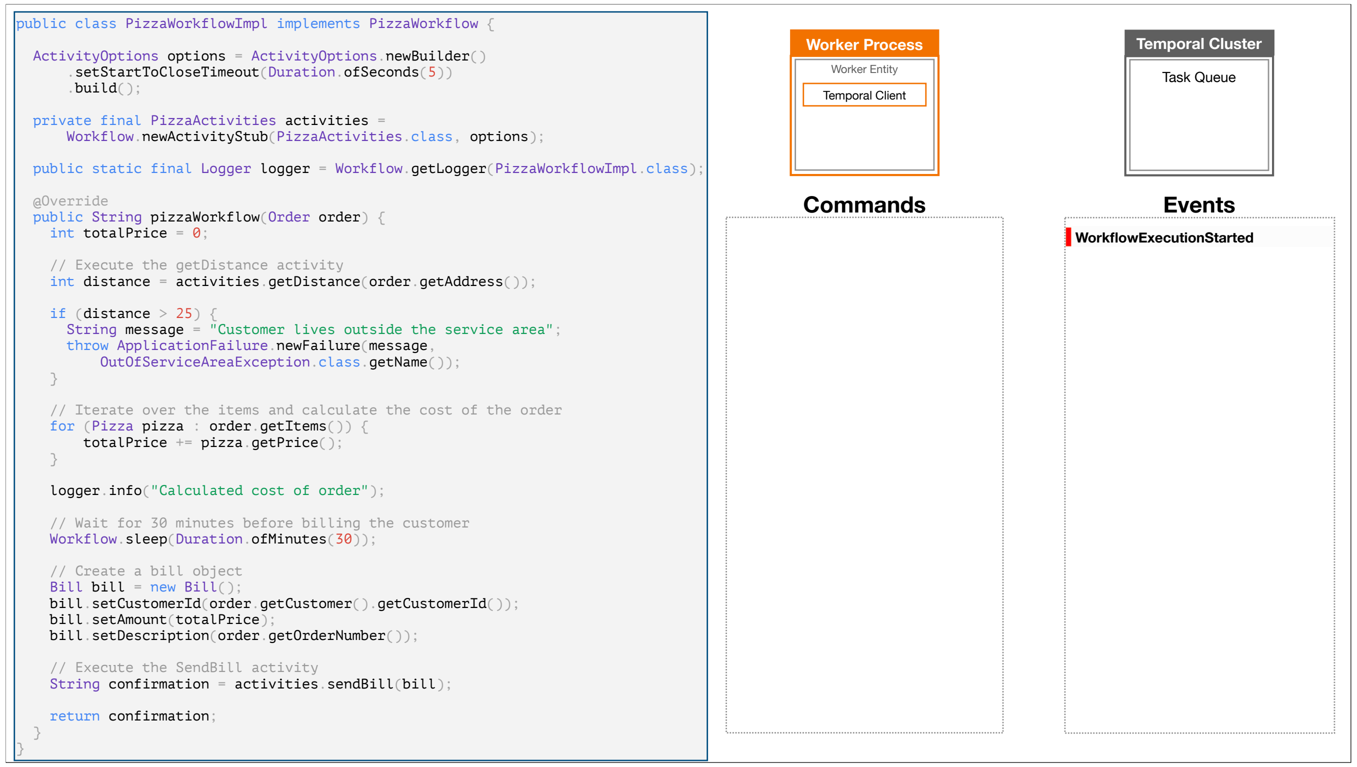
Since Commands and Events are essential to Workflow Replay, I'll explain the process using the pizza order Workflow that I used to cover those earlier. However, as before, I have omitted error handling code for the sake of brevity. Relatedly, I won't mention every Event that is written to the history, but I show a red rectangle to the left of each one when it first appears to help it stand out.

I'll also do this for Commands. As I step through the code, I use yellow highlighting to distinguish statements that result in Commands from code that does not, which is highlighted in white, just like before.

I'll start out by quickly walking through a Workflow Execution, showing a crash a little more than halfway through, and then explaining how Temporal uses Workflow Replay to recover the state, ultimately resulting in a completed execution that's identical to one that hadn't crashed.

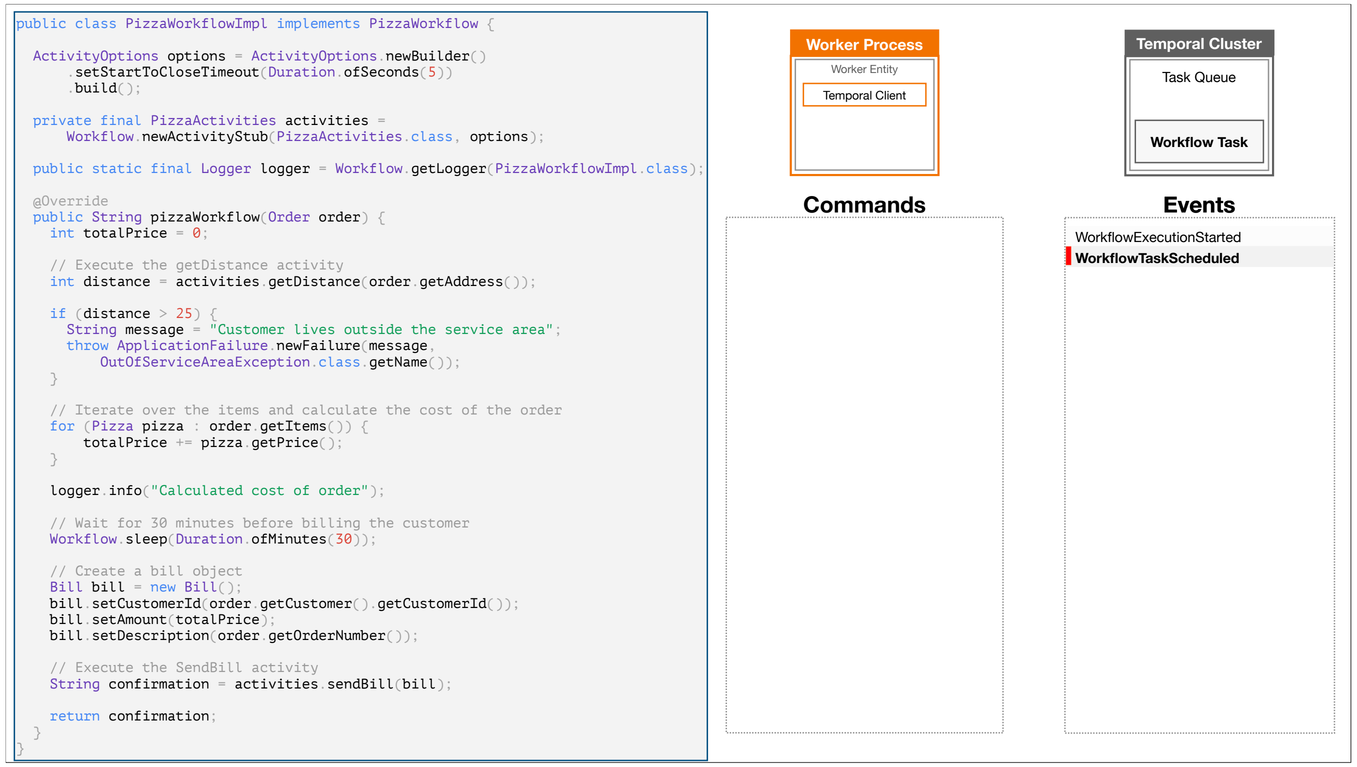
[advance]

It all begins by combining the code in the Workflow Definition with a request to execute it, passing in some input data. In this case, the input data contains information about the customer and the pizzas they ordered.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

This results in the Temporal Cluster logging a `WorkflowExecutionStarted` Event into the history,

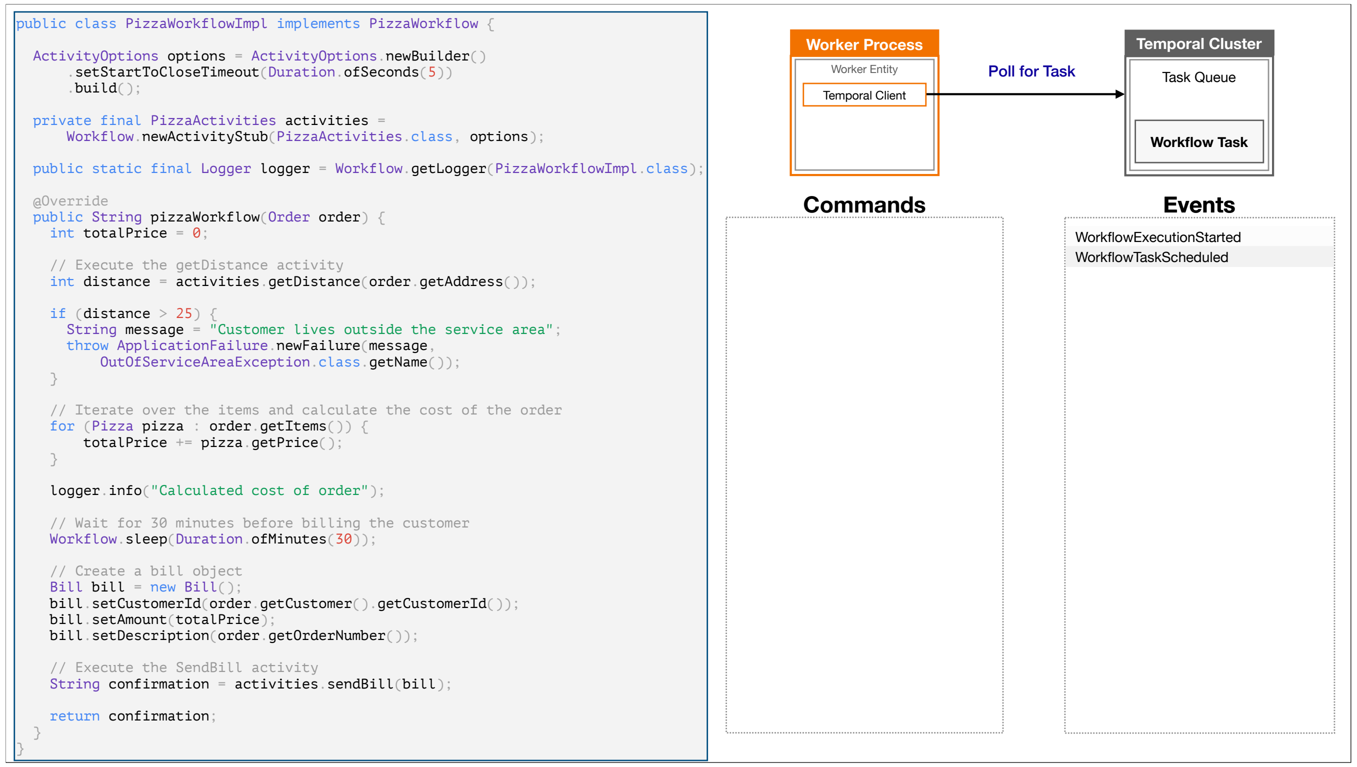


```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

WorkflowExecutionStarted
WorkflowTaskScheduled

adding a Workflow Task to the queue, and logging a `WorkflowTaskScheduled` Event.

Although I didn't indicate it here, due to limited space on the screen, the `WorkflowExecutionStarted` Event contains the input data provided to this Workflow Execution.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Poll for Task

Temporal Cluster

Task Queue

Workflow Task

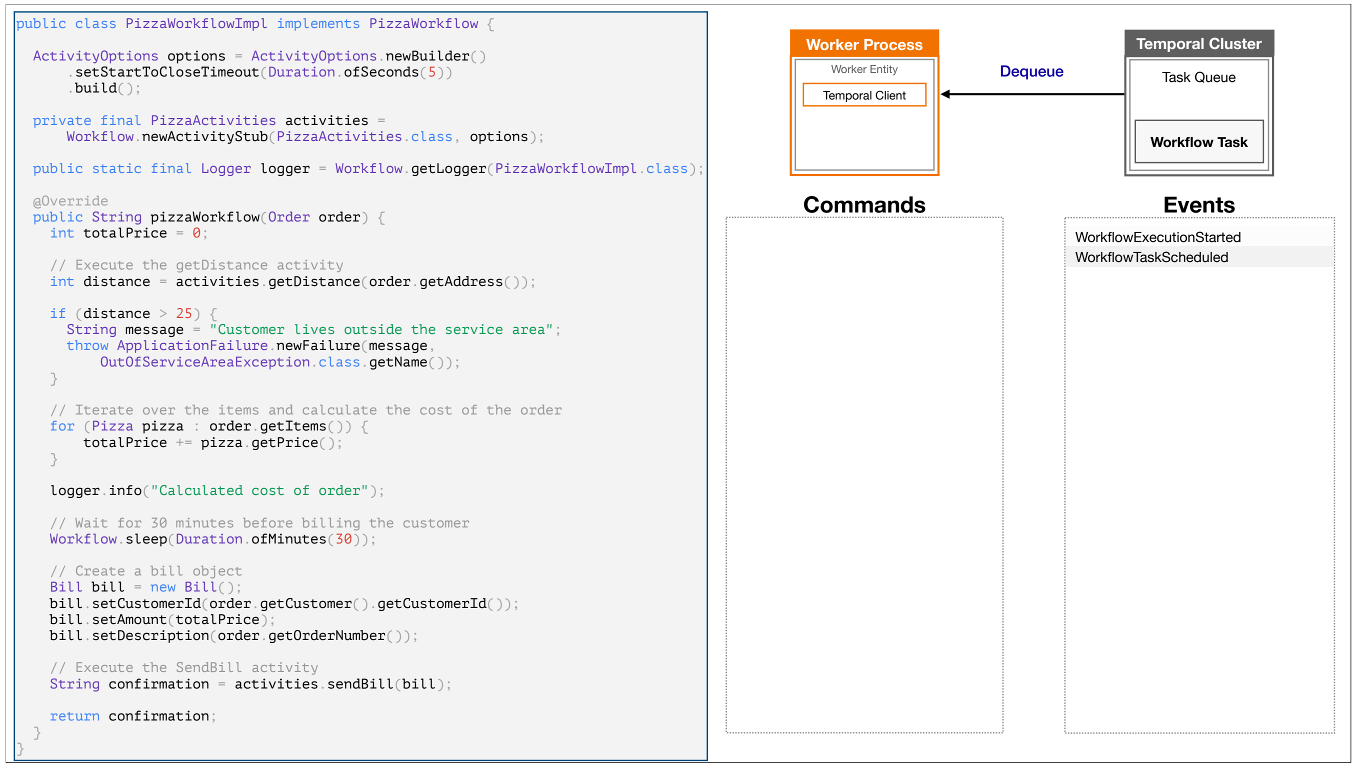
Commands

Events

WorkflowExecutionStarted

WorkflowTaskScheduled

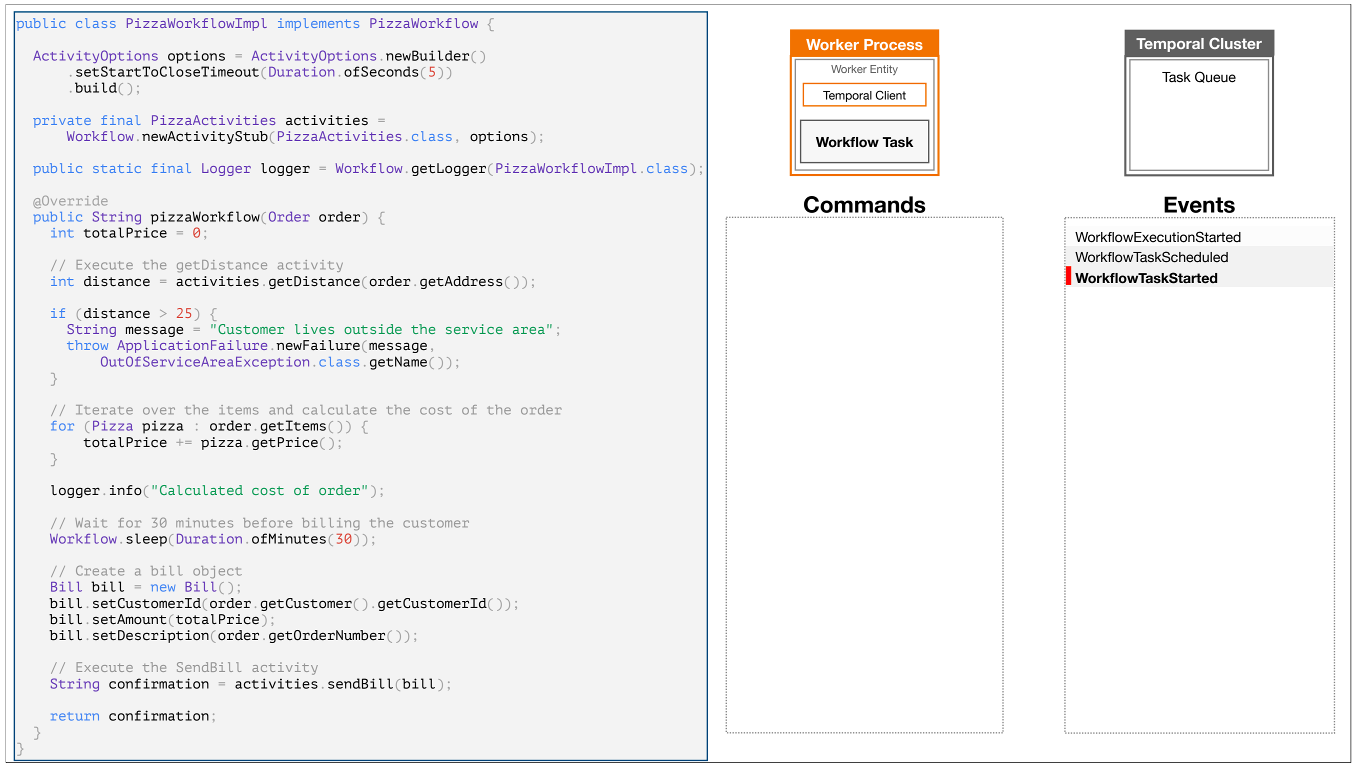
When a Worker polls the Task Queue



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

WorkflowExecutionStarted
WorkflowTaskScheduled

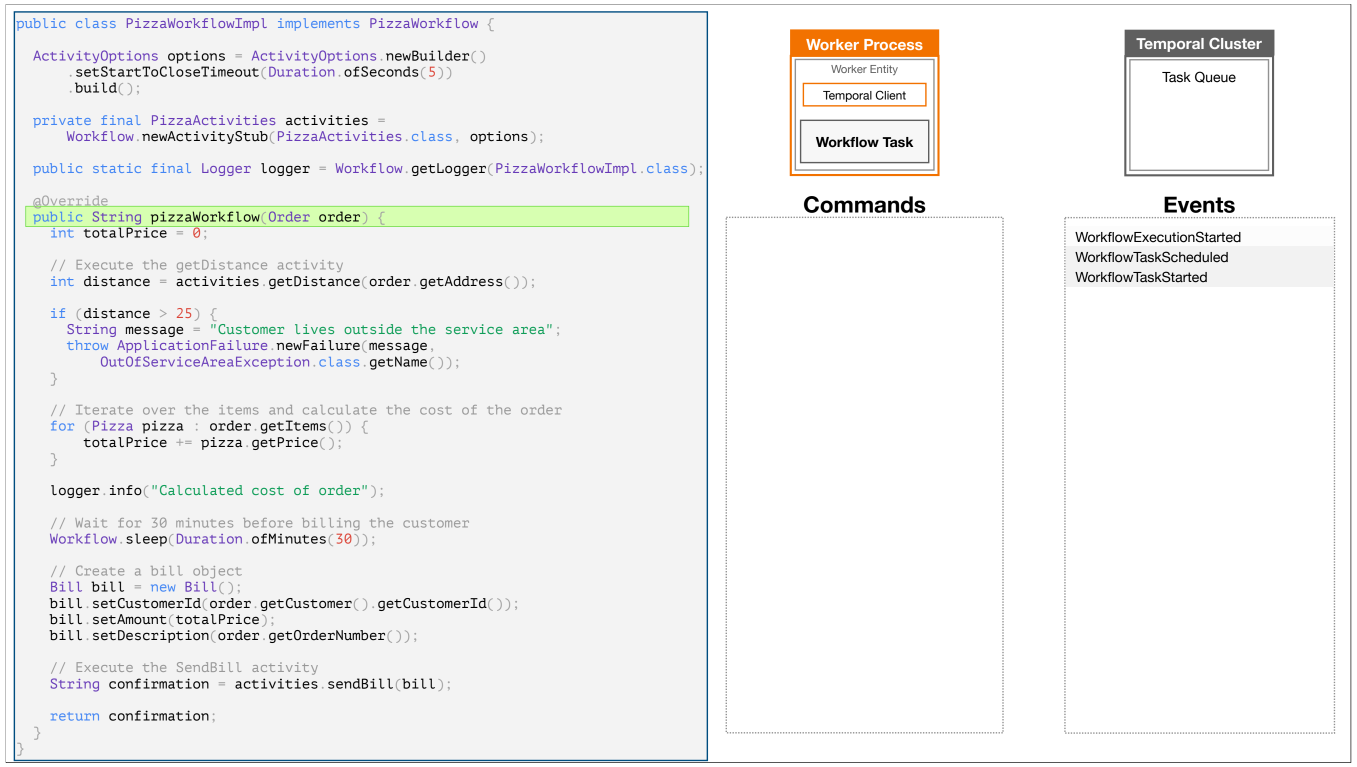
and accepts the Task,



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted

the cluster logs a `WorkflowTaskStarted` Event.



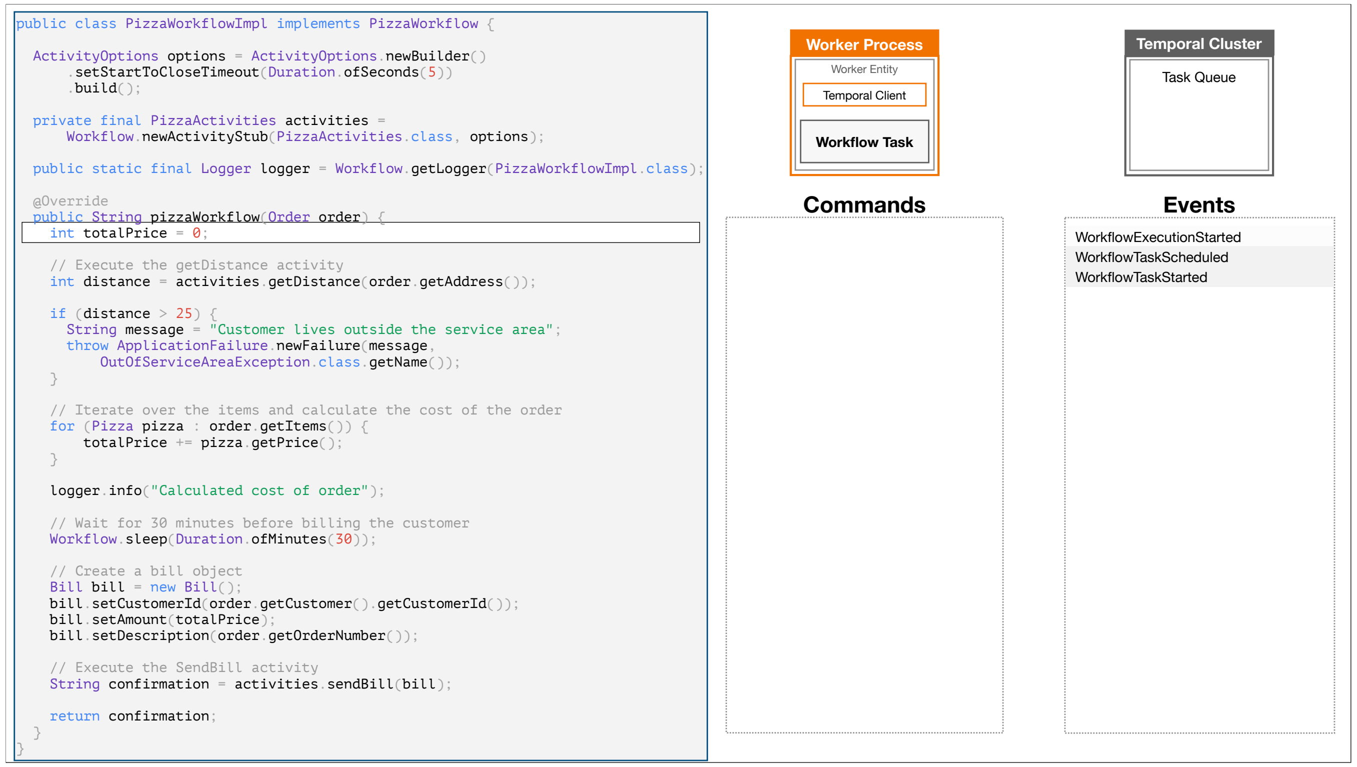
```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted

The Worker then invokes the Workflow function and runs the code within it, one statement at a time.



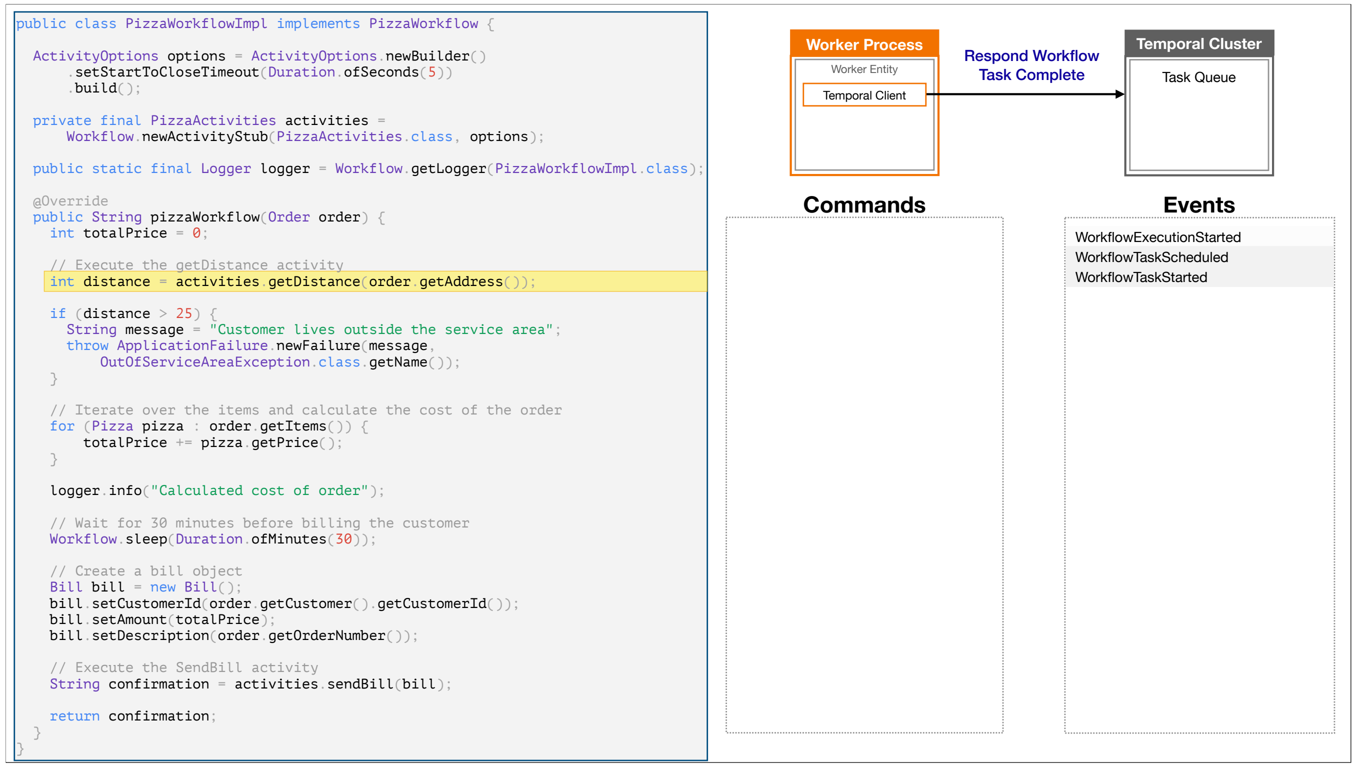
```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

Events

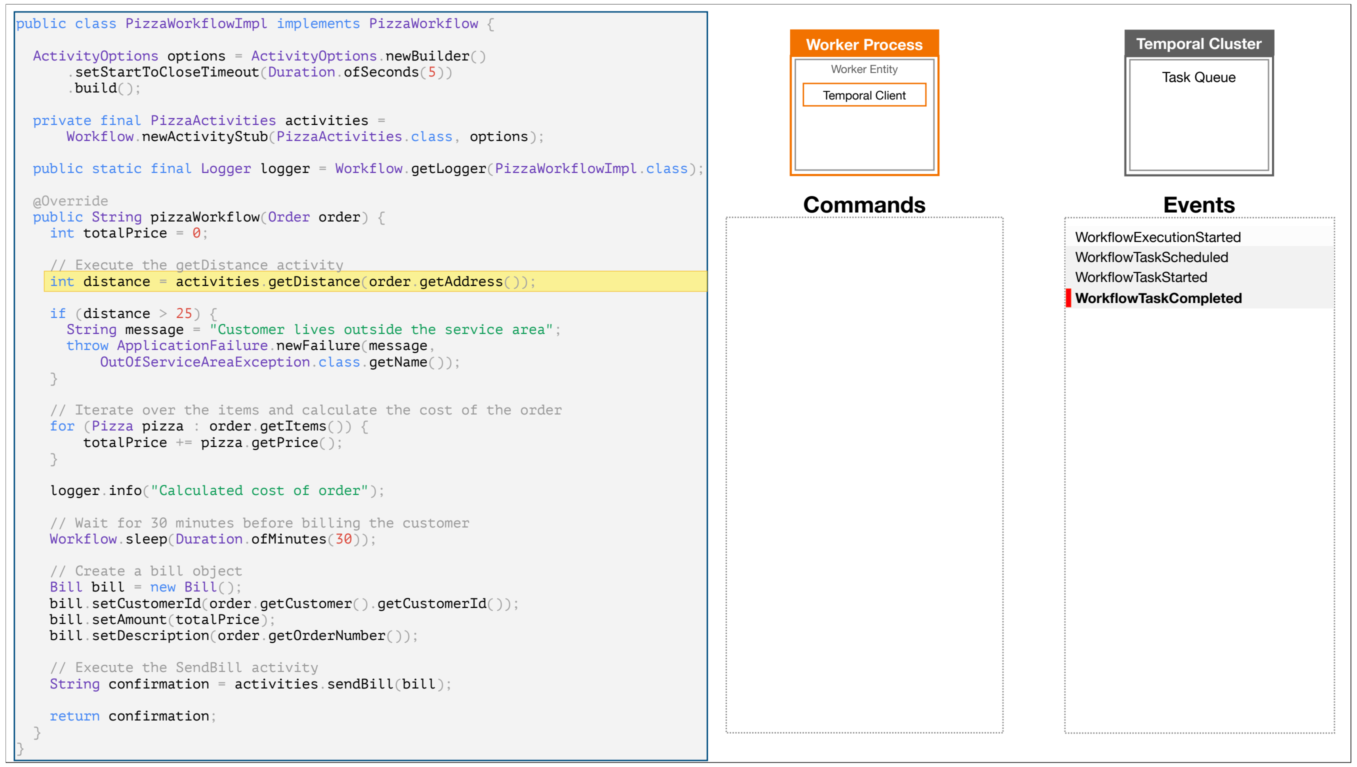
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted

The first statements do not result in any interaction with the cluster.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
       .setStartToCloseTimeout(Duration.ofSeconds(5))
       .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

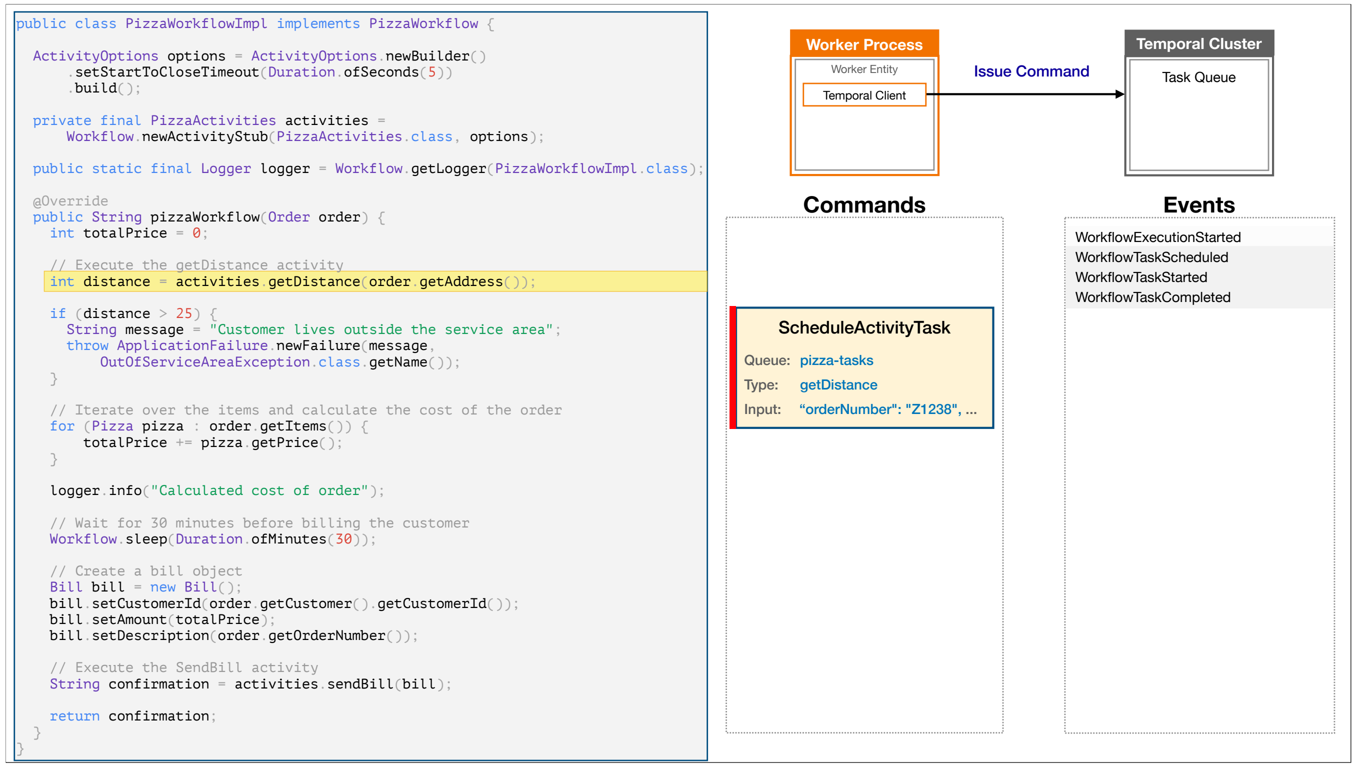
This statement requests the execution of an Activity,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted

so the Worker completes the current Workflow Task.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

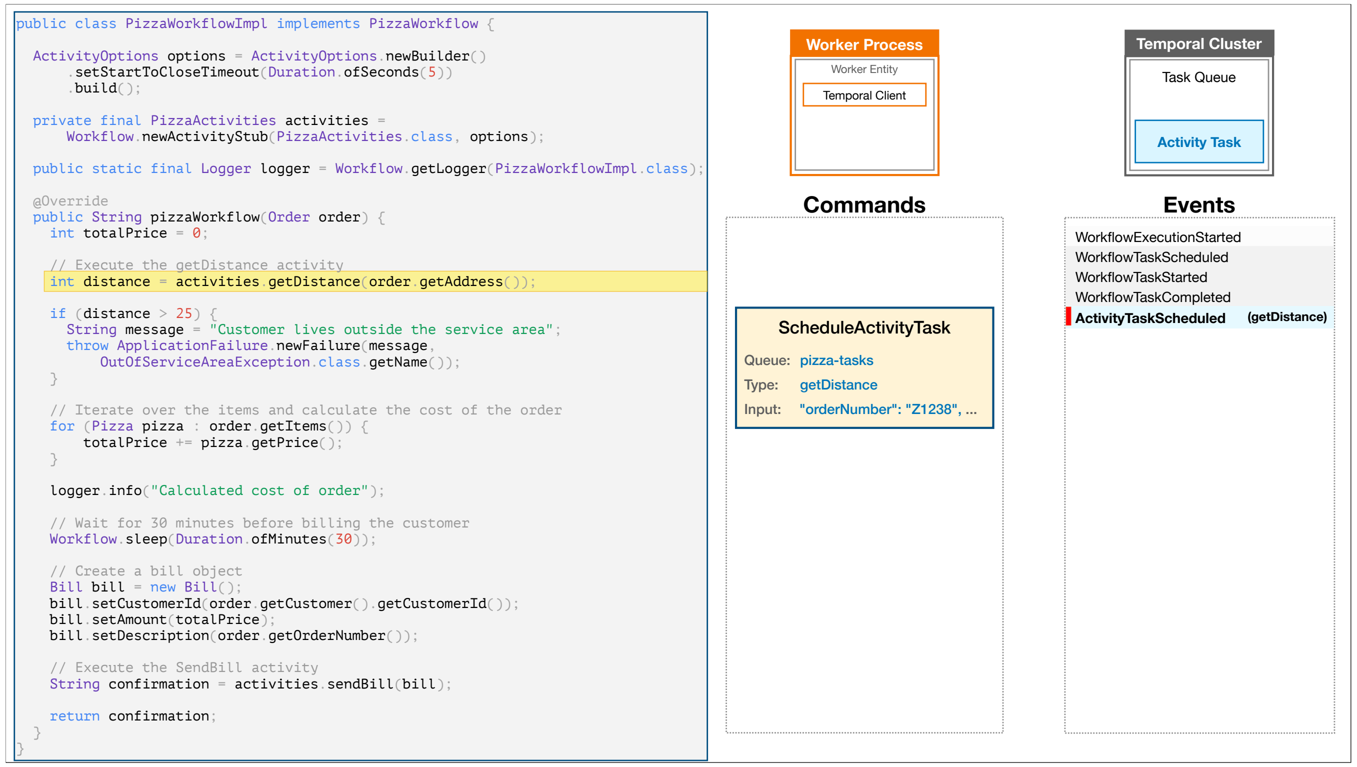
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted

The Worker issues a Command, which contains details about the Activity Execution, to the cluster



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks

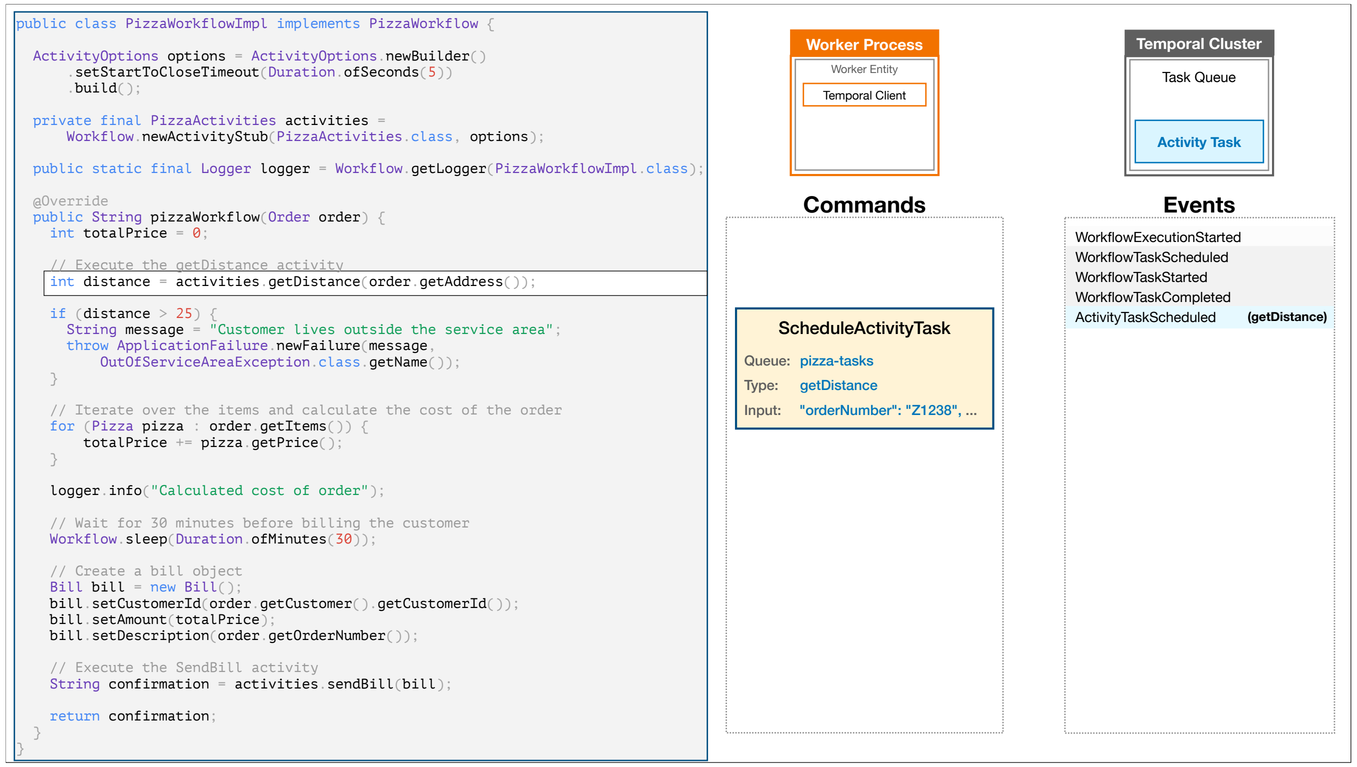
Type: getDistance

Input: "orderNumber": "Z1238", ...

Events

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

In response, the cluster queues an Activity Task and logs an `ActivityTaskScheduled` Event. I have shown this in blue to indicate that it is the direct result of the Command.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Temporal Cluster

Task Queue

Activity Task

Commands

ScheduleActivityTask

Queue: pizza-tasks

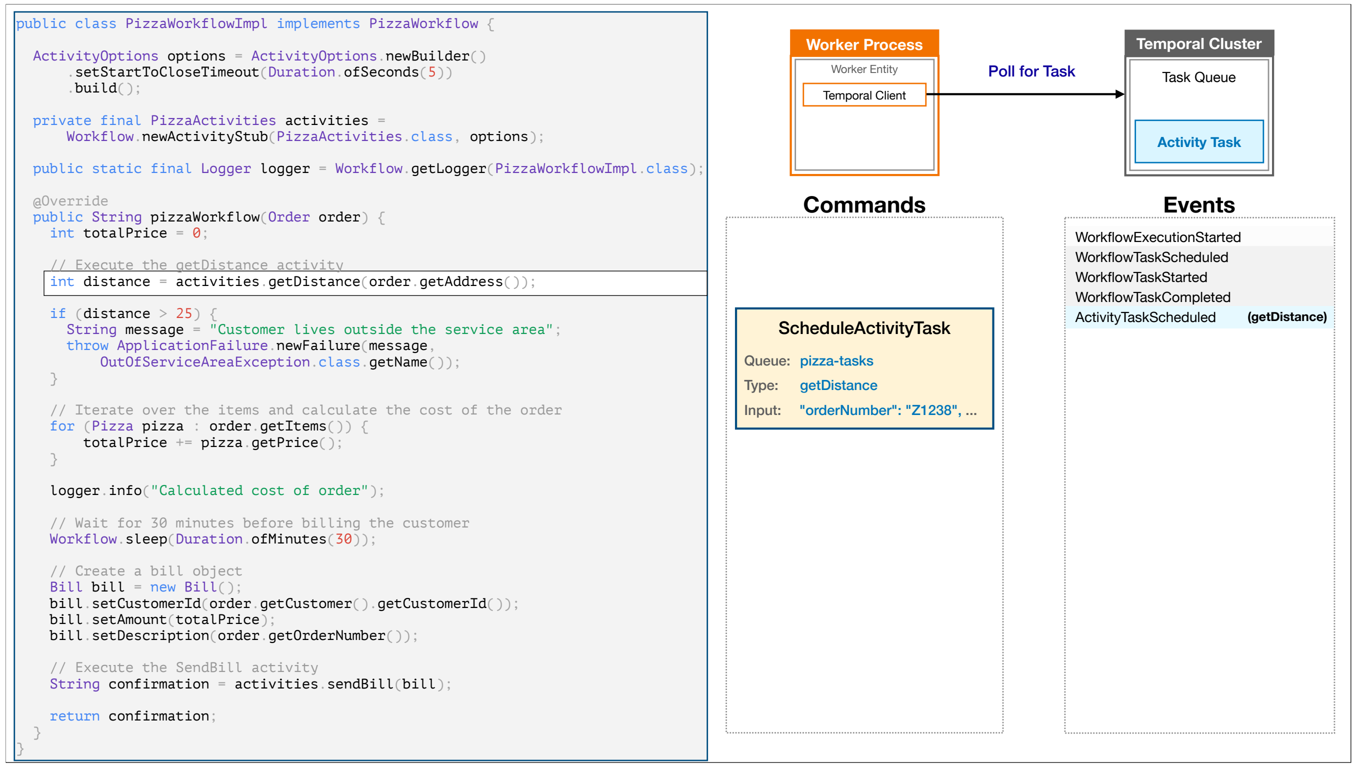
Type: getDistance

Input: "orderNumber": "Z1238", ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)

The Worker now awaits the result from Activity Execution.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

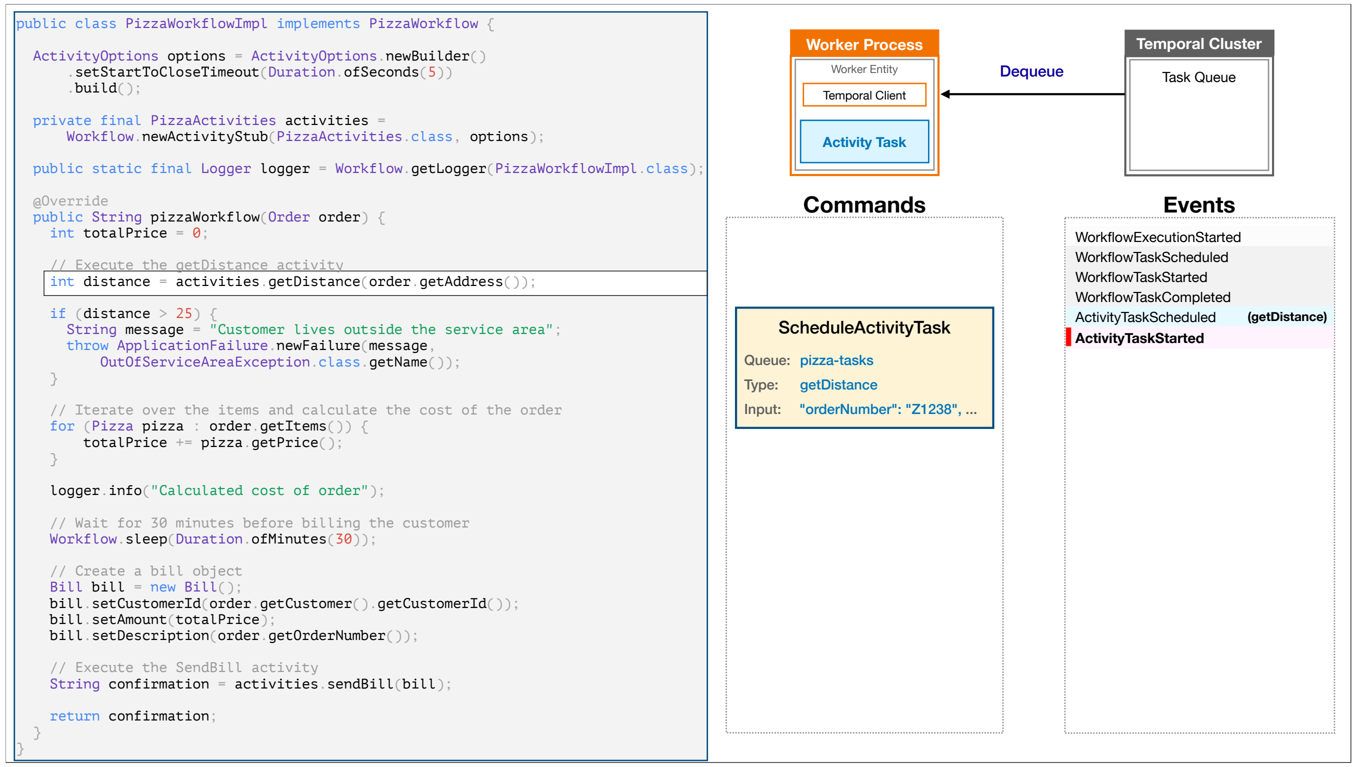
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)

When this Worker—or another one—has spare capacity to do some work, it polls and is matched with the Activity Task, which it accepts.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

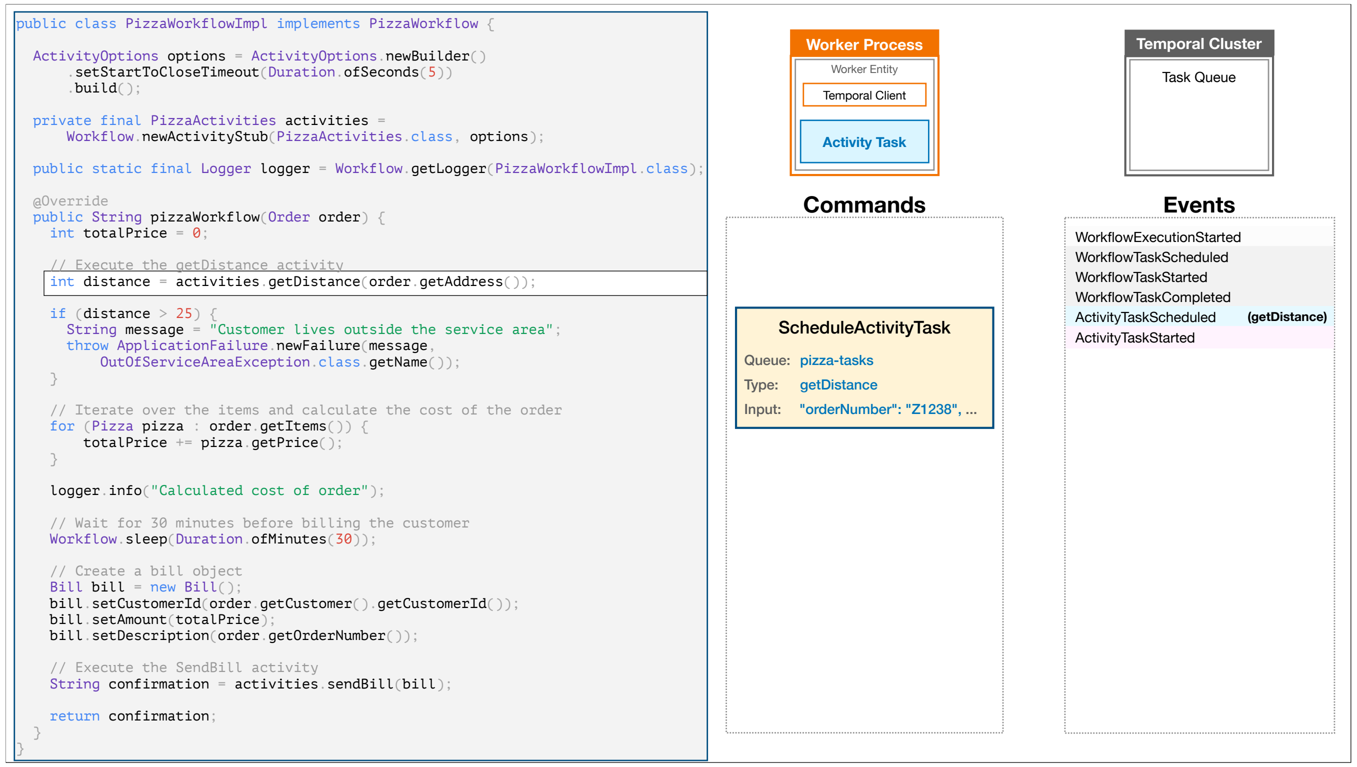
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

The cluster logs an Event to signify that the Worker has started the Activity Task. I've shown this in pink to indicate that it's the indirect result of the Command.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

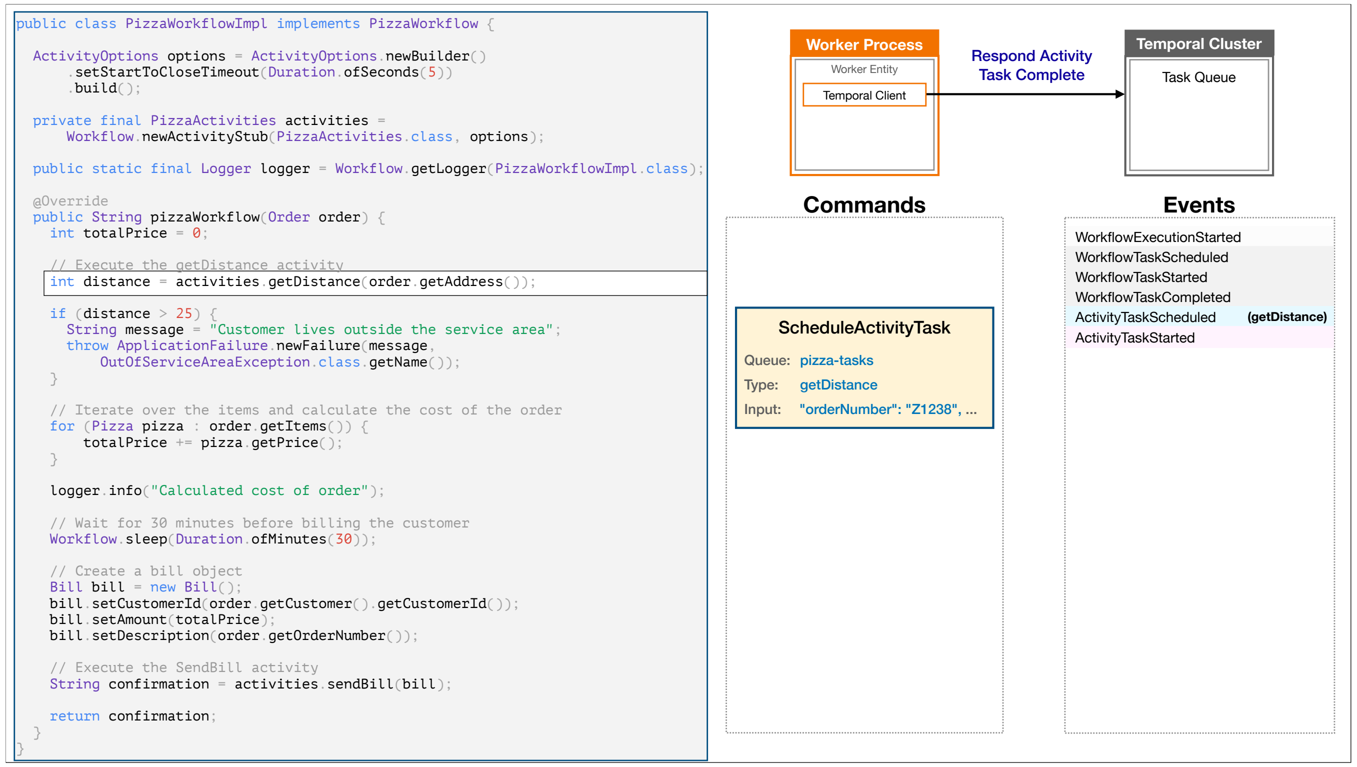
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted

The Worker invokes the Activity Definition. In this case, that's the `getDistance` function.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

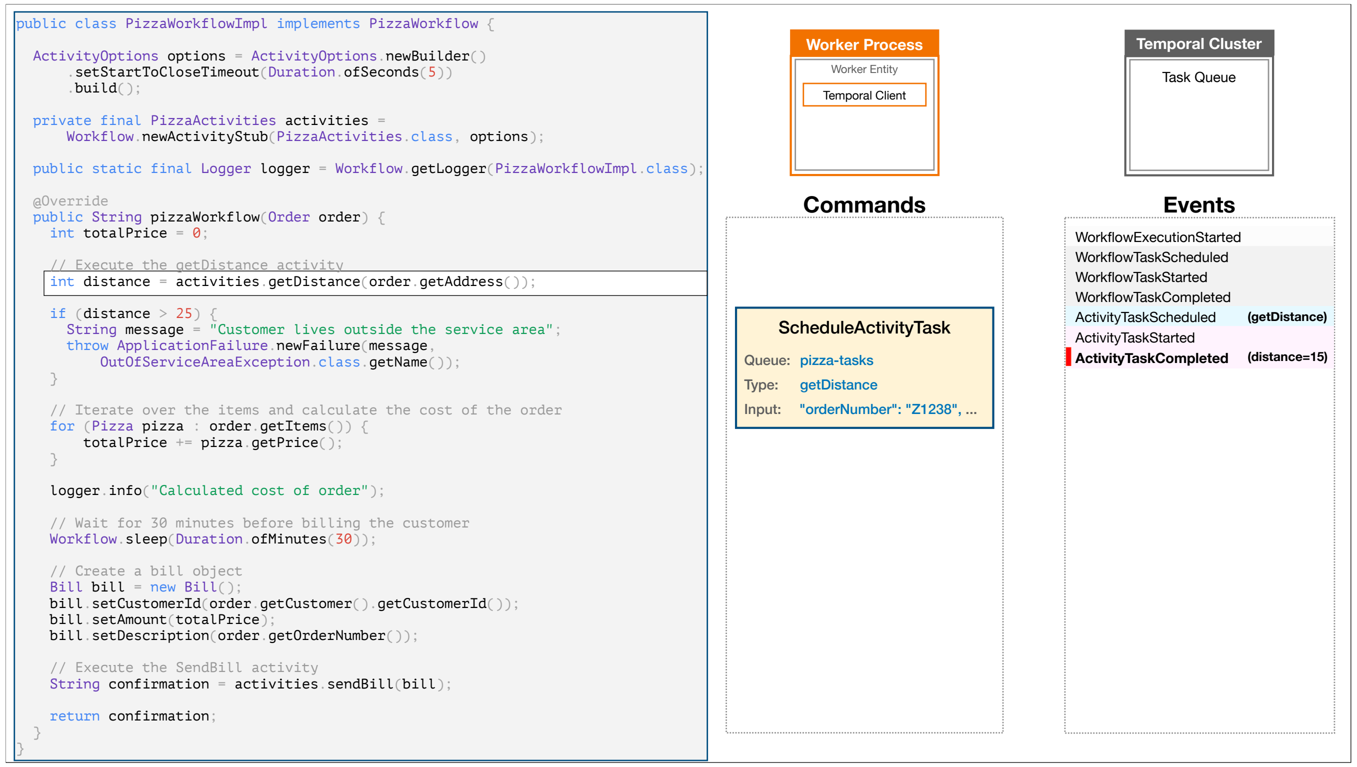
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted

Let’s suppose that it ultimately returns a value of `15`. When the function returns, the Worker notifies that cluster that the Activity Execution is complete.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

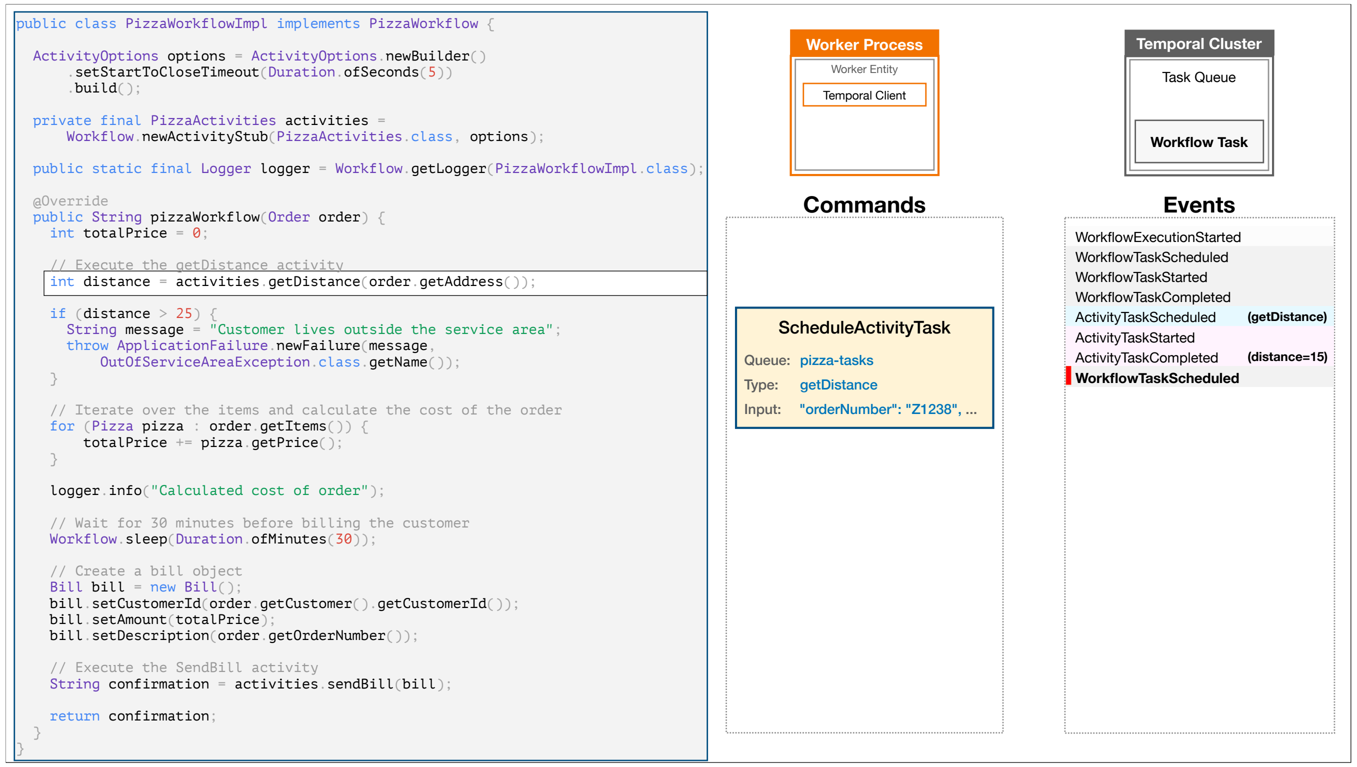
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

The cluster logs an `ActivityTaskCompleted` Event, which contains this result.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Temporal Cluster

Task Queue

Workflow Task

Commands

ScheduleActivityTask

Queue: pizza-tasks

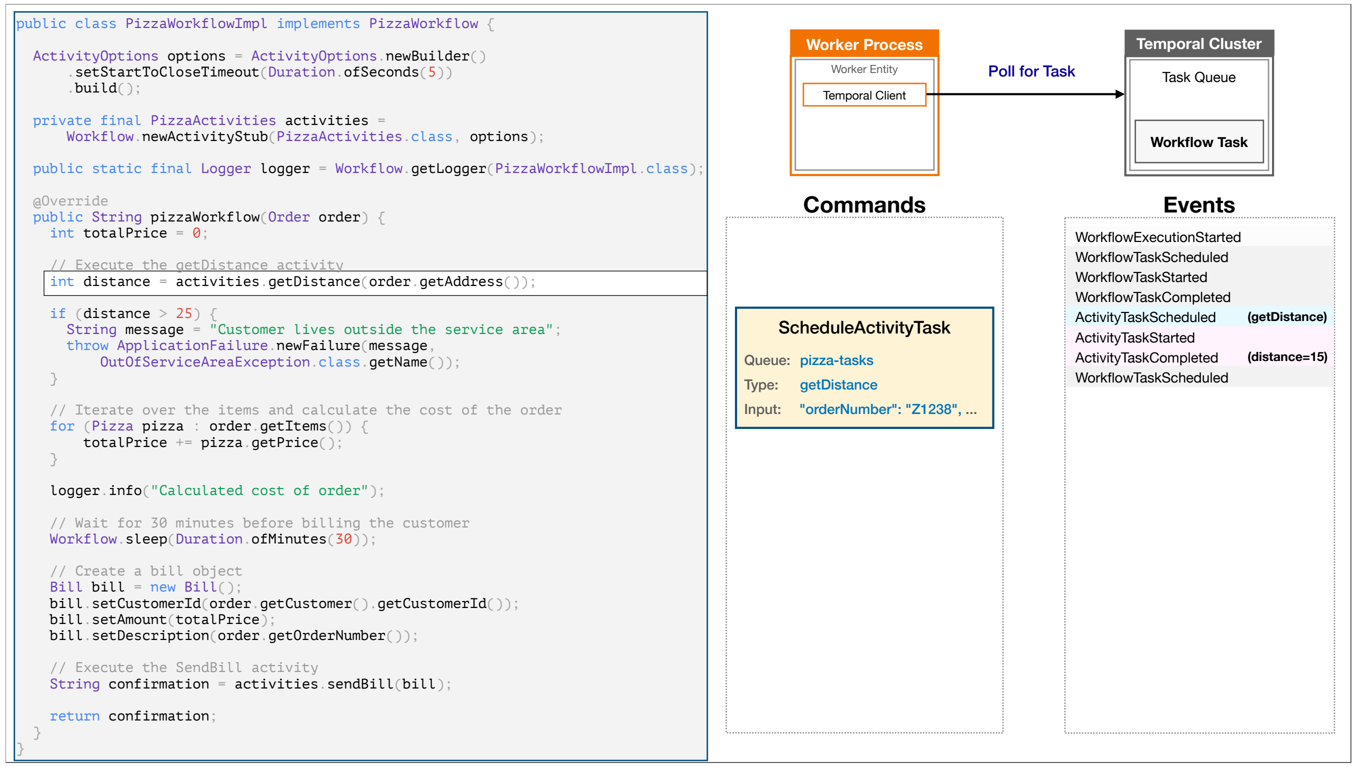
Type: getDistance

Input: "orderNumber": "Z1238", ...

Events

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

The cluster then adds another Workflow Task to drive the Workflow Execution forward.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

ScheduleActivityTask

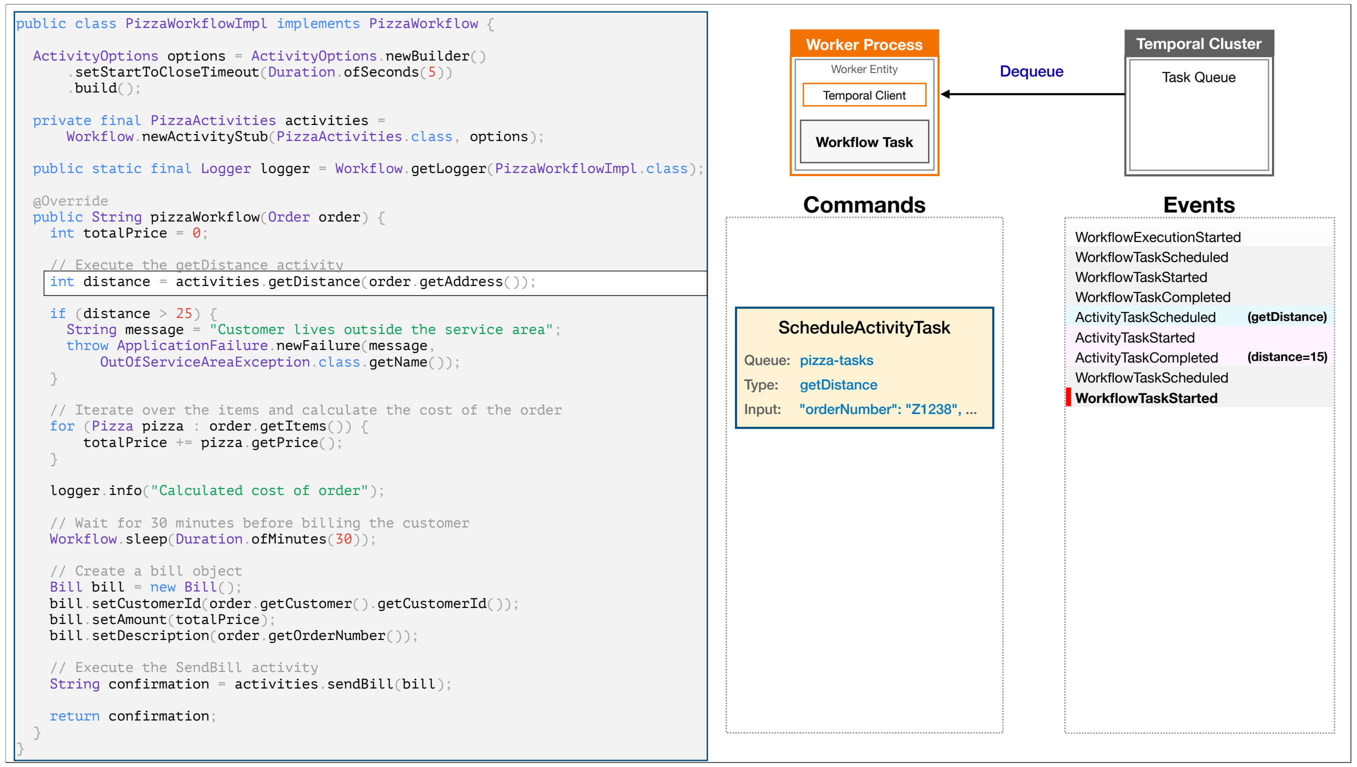
Queue: pizza-tasks

Type: getDistance

Input: "orderNumber": "Z1238", ...

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

When the Worker polls, it's matched with this task,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks

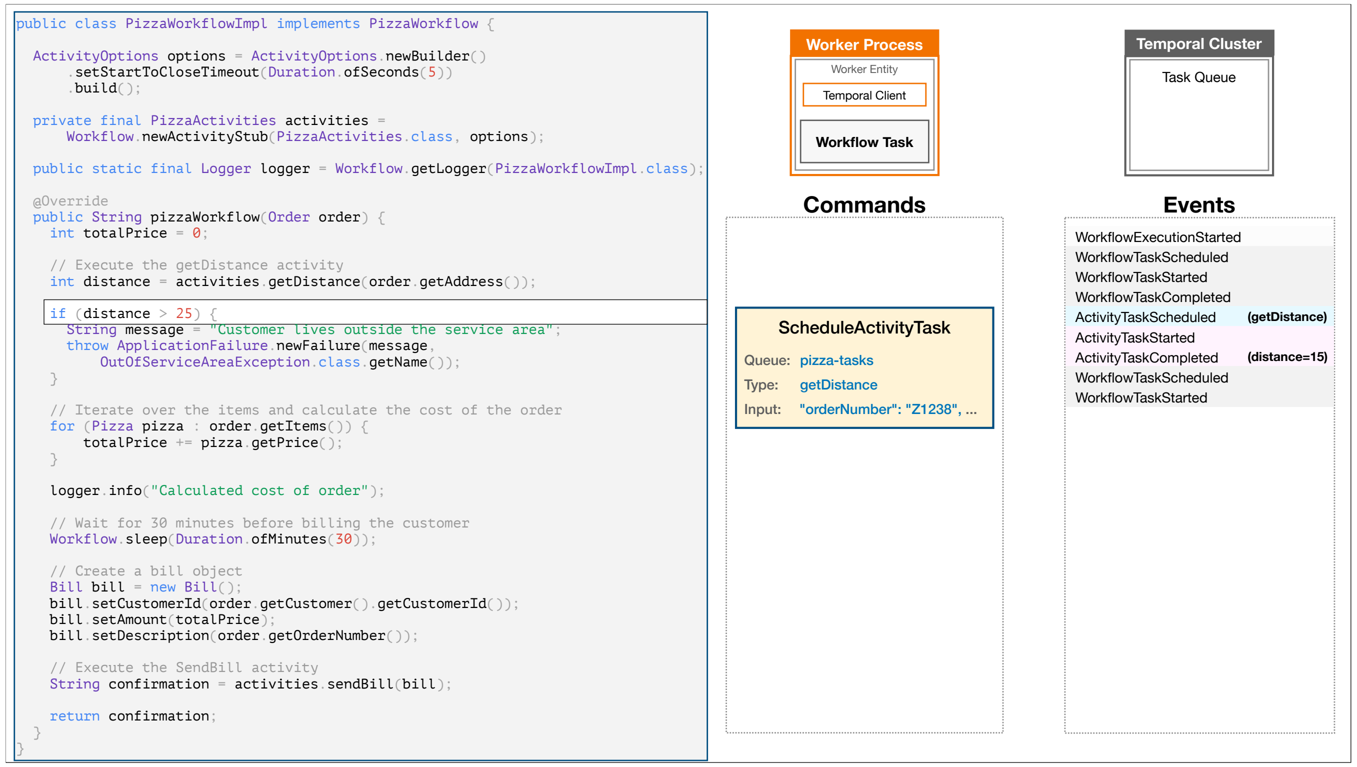
Type: getDistance

Input: "orderNumber": "Z1238", ...

Events

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

dequeues it,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

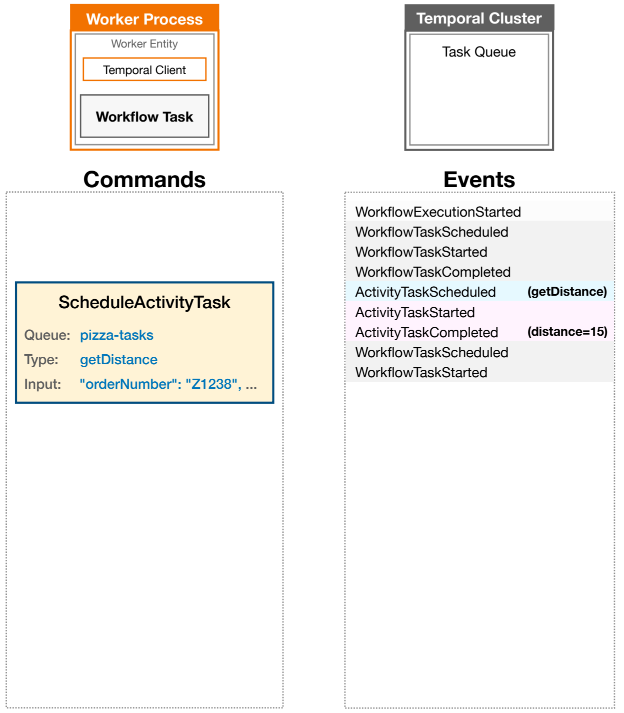
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

and resumes execution of the Workflow code.

```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```



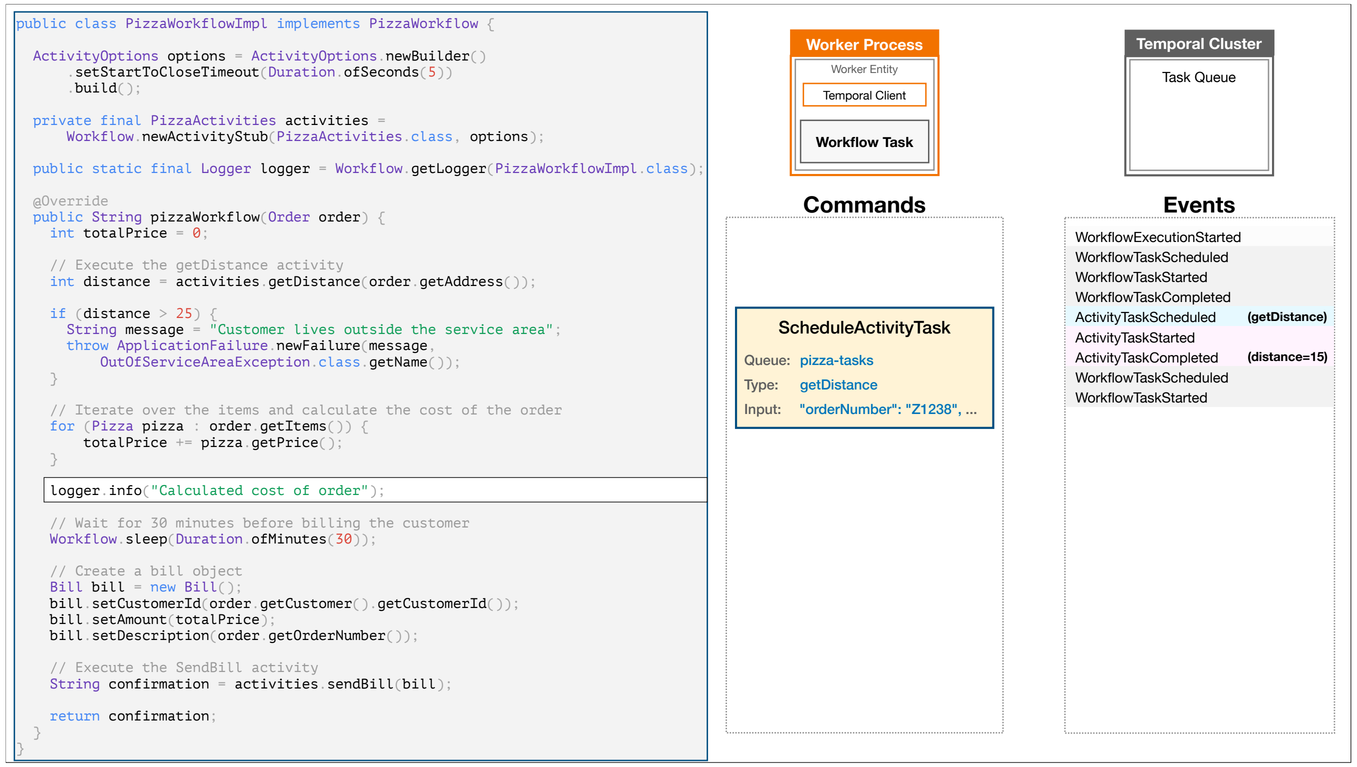
Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

The price is totaled up, which happens in the Workflow. Once again, there’s no communication with the Temporal Cluster.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks

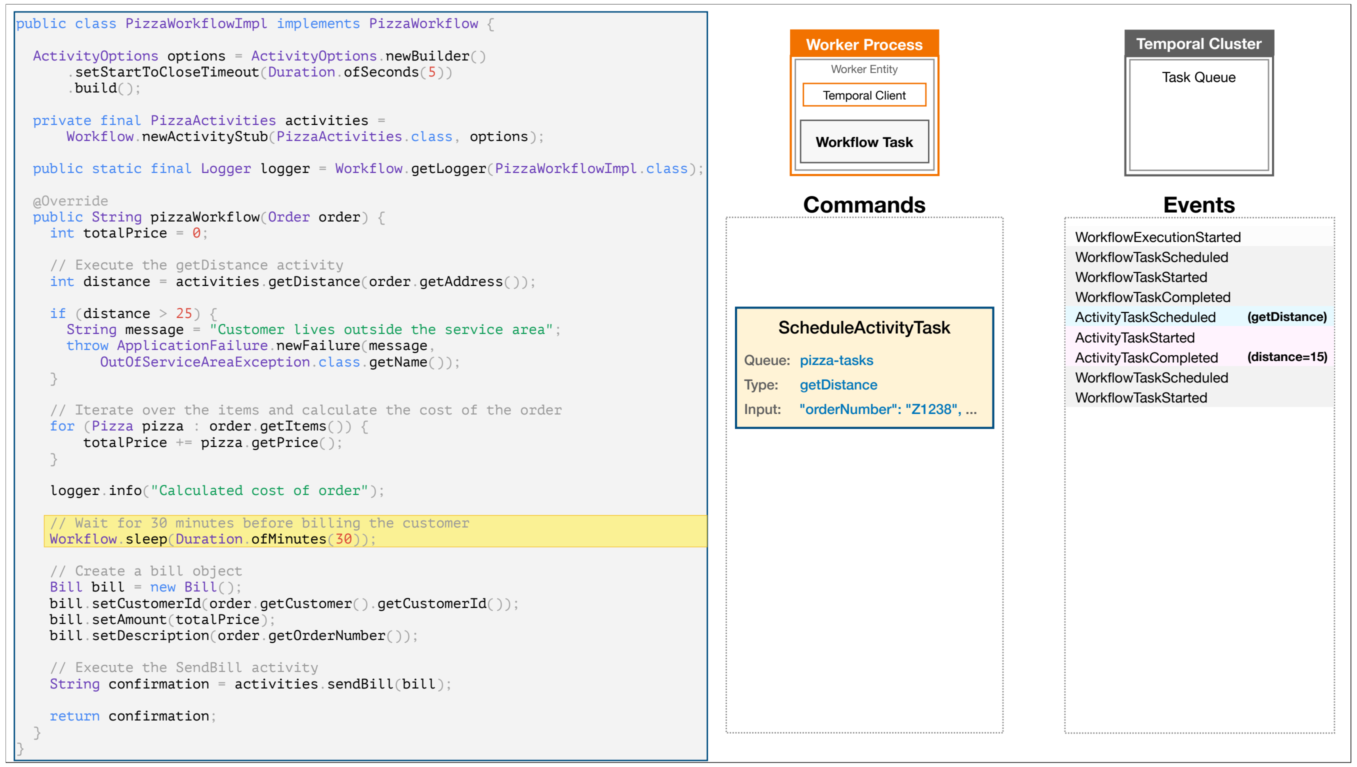
Type: getDistance

Input: "orderNumber": "Z1238", ...

Events

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

This log statement also happens locally.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

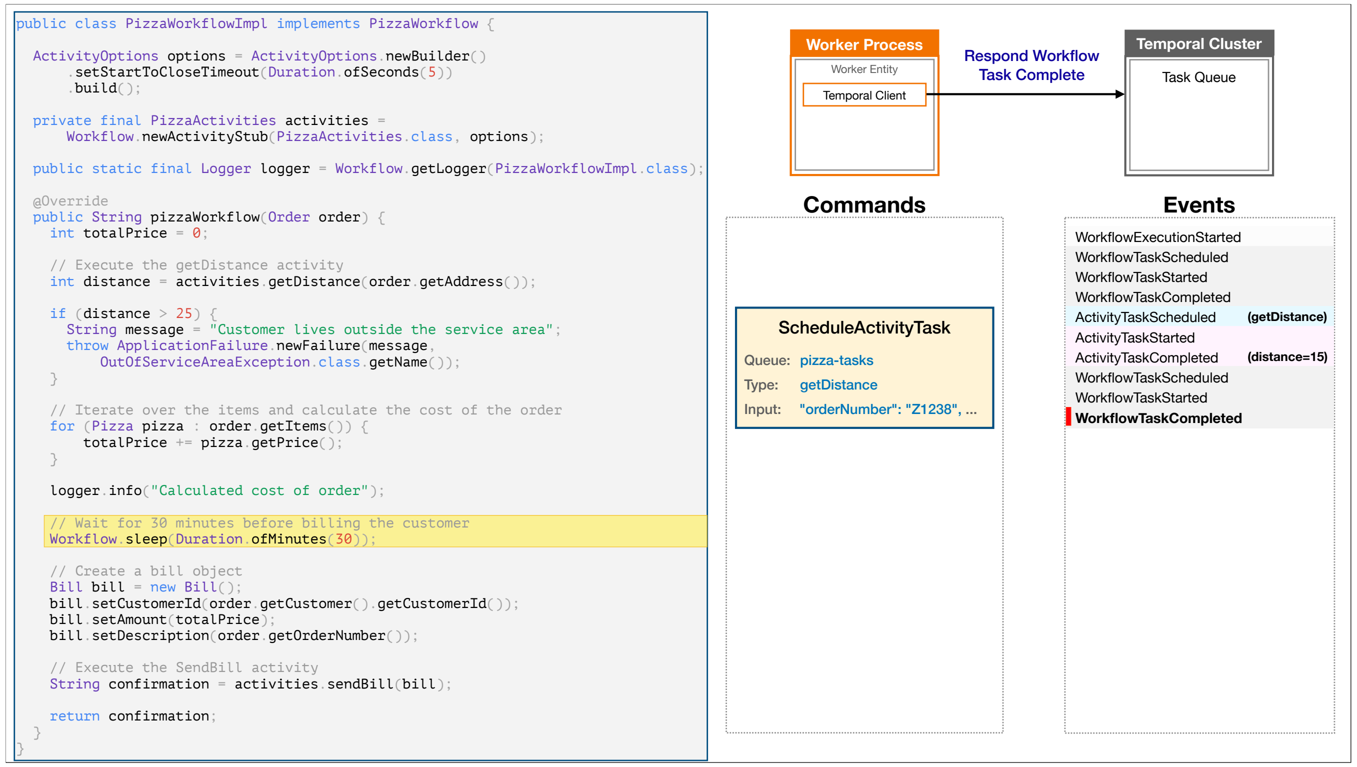
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

When it hits the `Workflow.sleep` call…



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

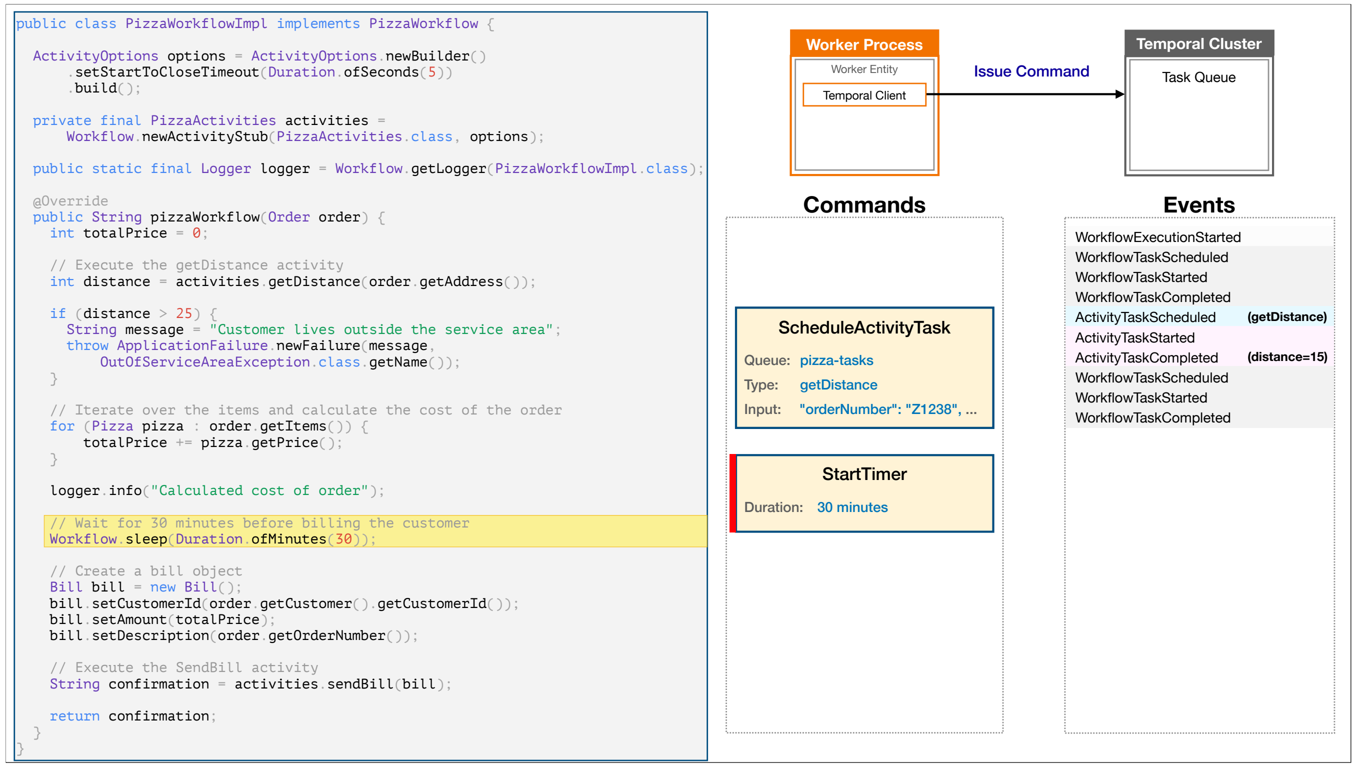
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

it completes the current Workflow Task…



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

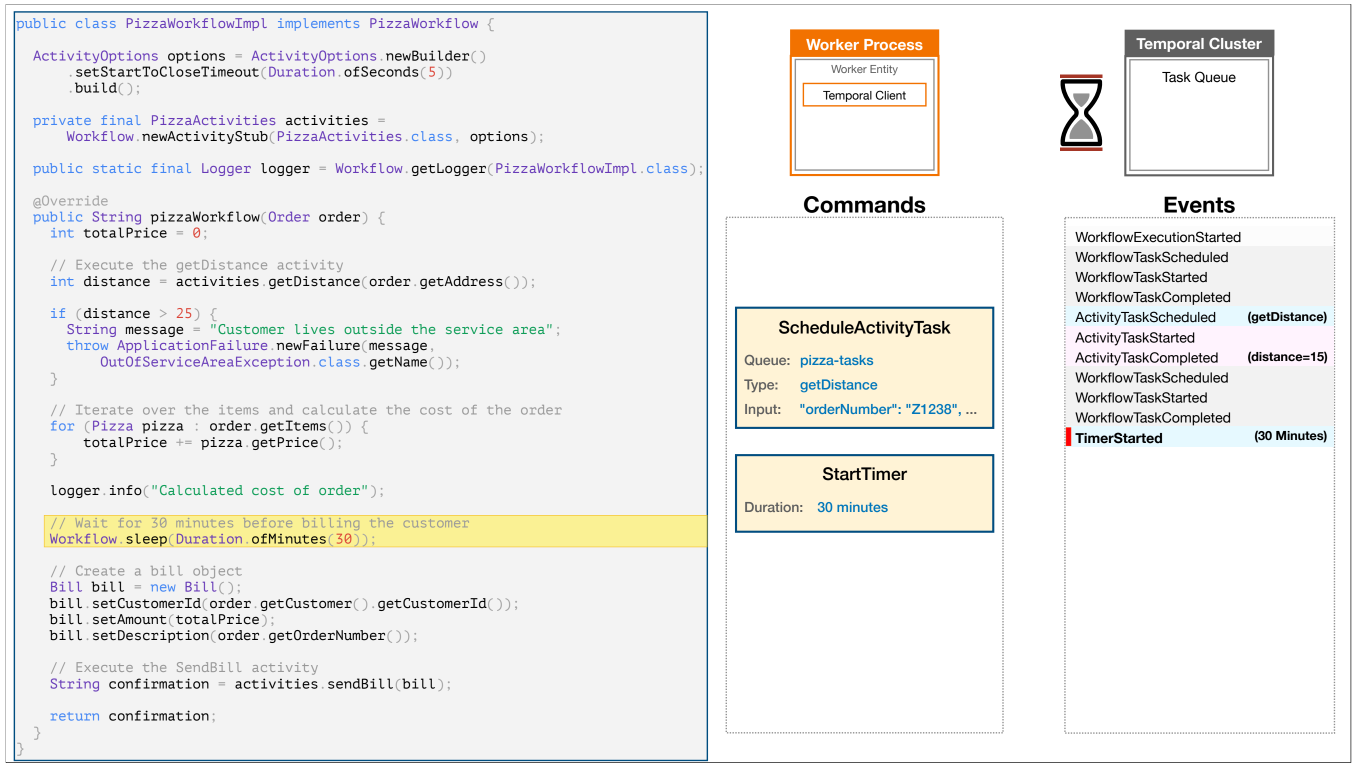
ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
Duration: 30 minutes

Events

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

and issues a Command to the cluster, requesting it to set a Timer for 30 minutes.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

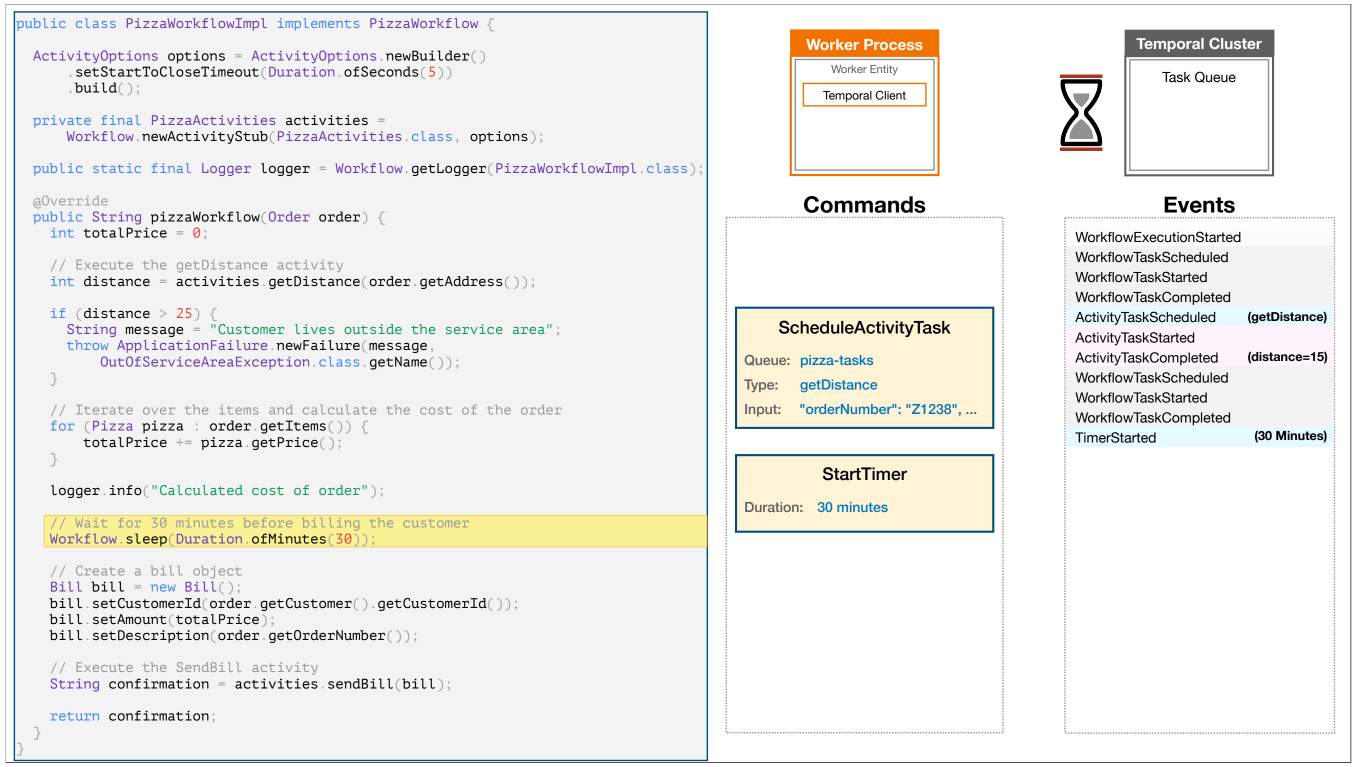
StartTimer

Duration: 30 minutes

Events

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

The cluster logs a `TimerStarted` Event in response. The Workflow cannot progress until that Timer fires, so the cluster does not queue any new Tasks for this Workflow Execution until that happens.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

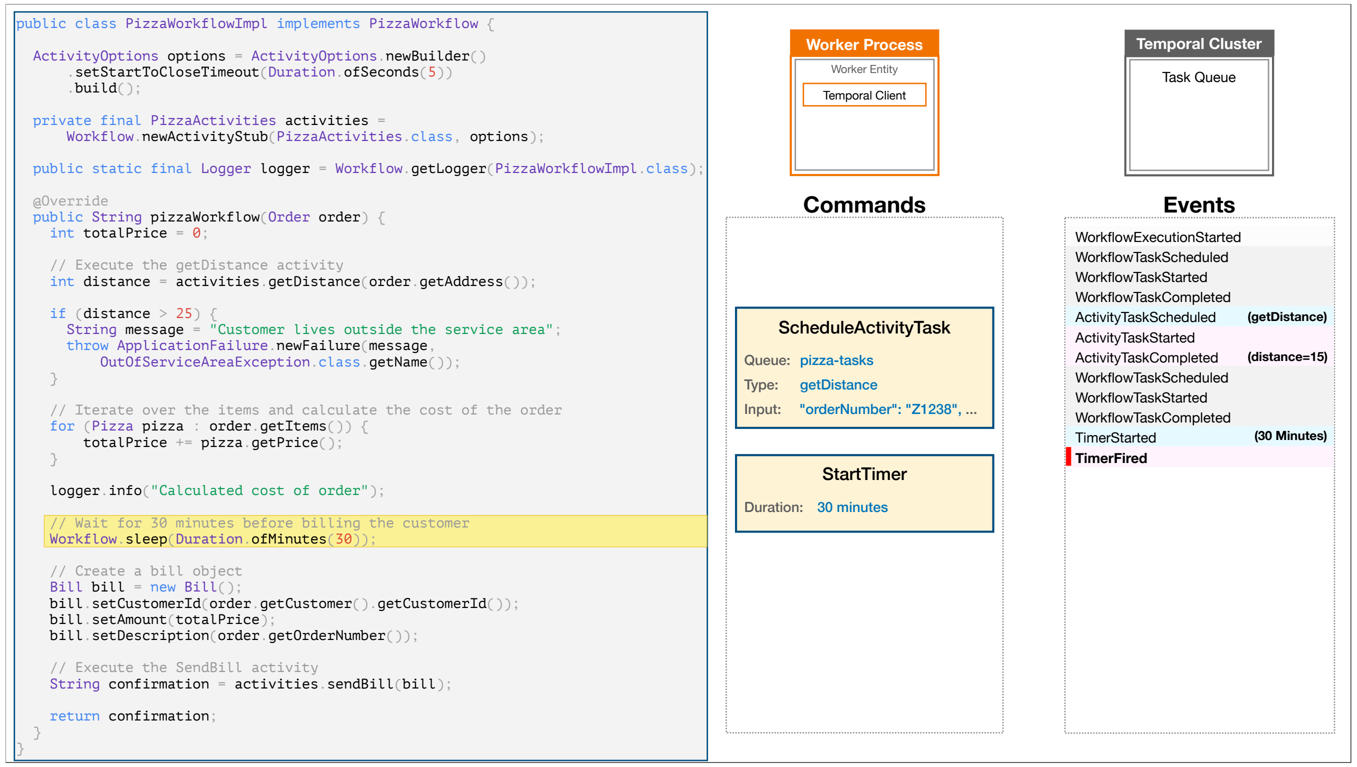
ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
Duration: 30 minutes

Events

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

The Worker may continue polling during this time, but since there are no Tasks related to this Workflow Execution, it won't perform any work. Therefore, setting a Timer in a Temporal Workflow, even one that lasts for several years, does not waste resources.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

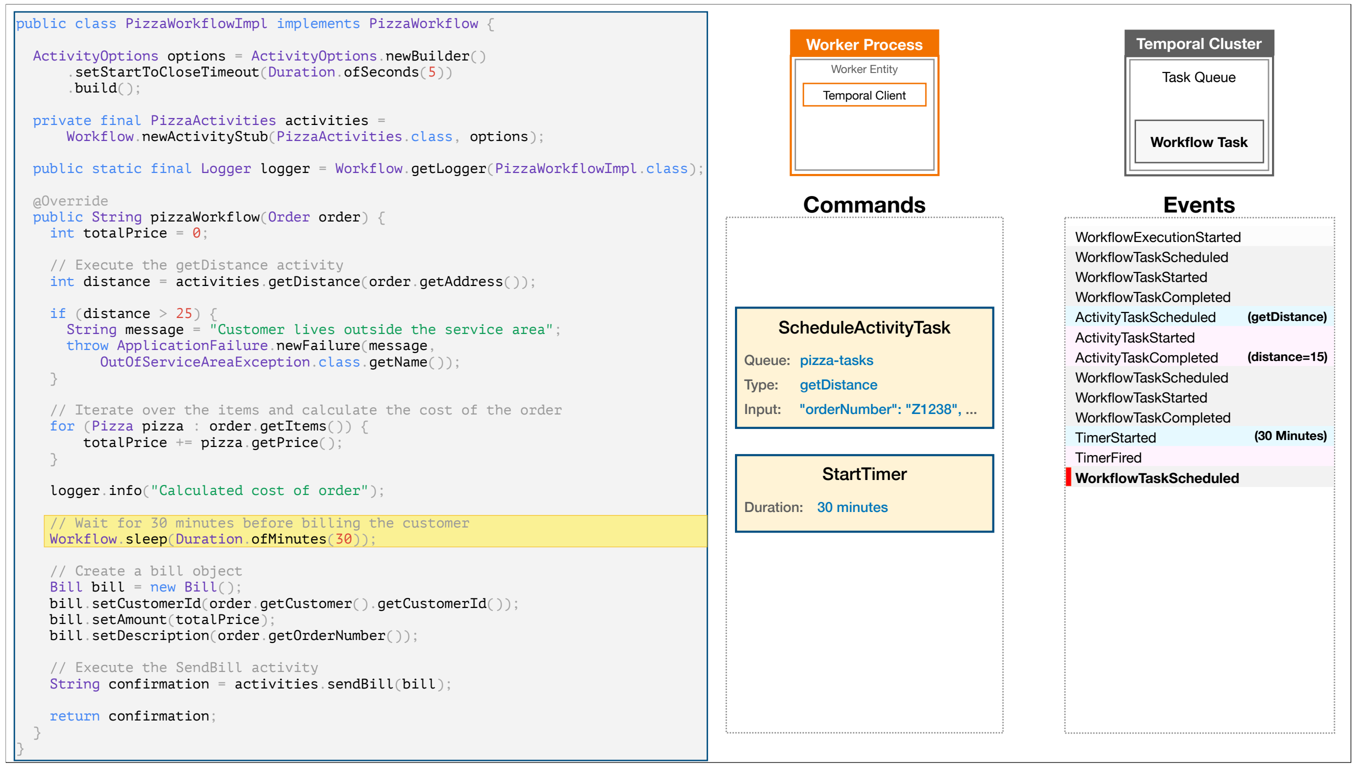
ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
Duration: 30 minutes

Events

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

After 30 minutes has elapsed, the Timer fires, and the cluster logs a`TimerFired` Event. How does the Worker know that it can continue running the Workflow code now?



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

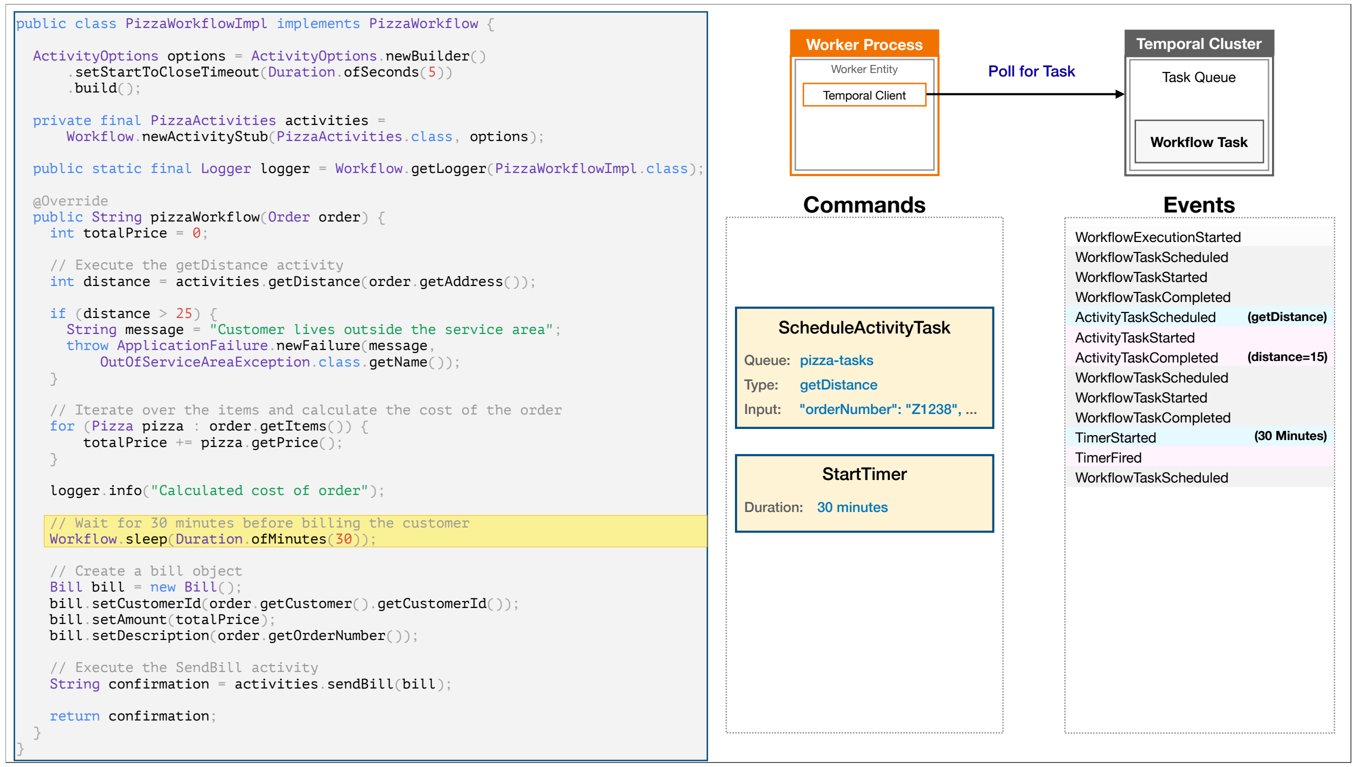
StartTimer

Duration: 30 minutes

Events

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

It's because the cluster now adds a new Workflow Task to the queue,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

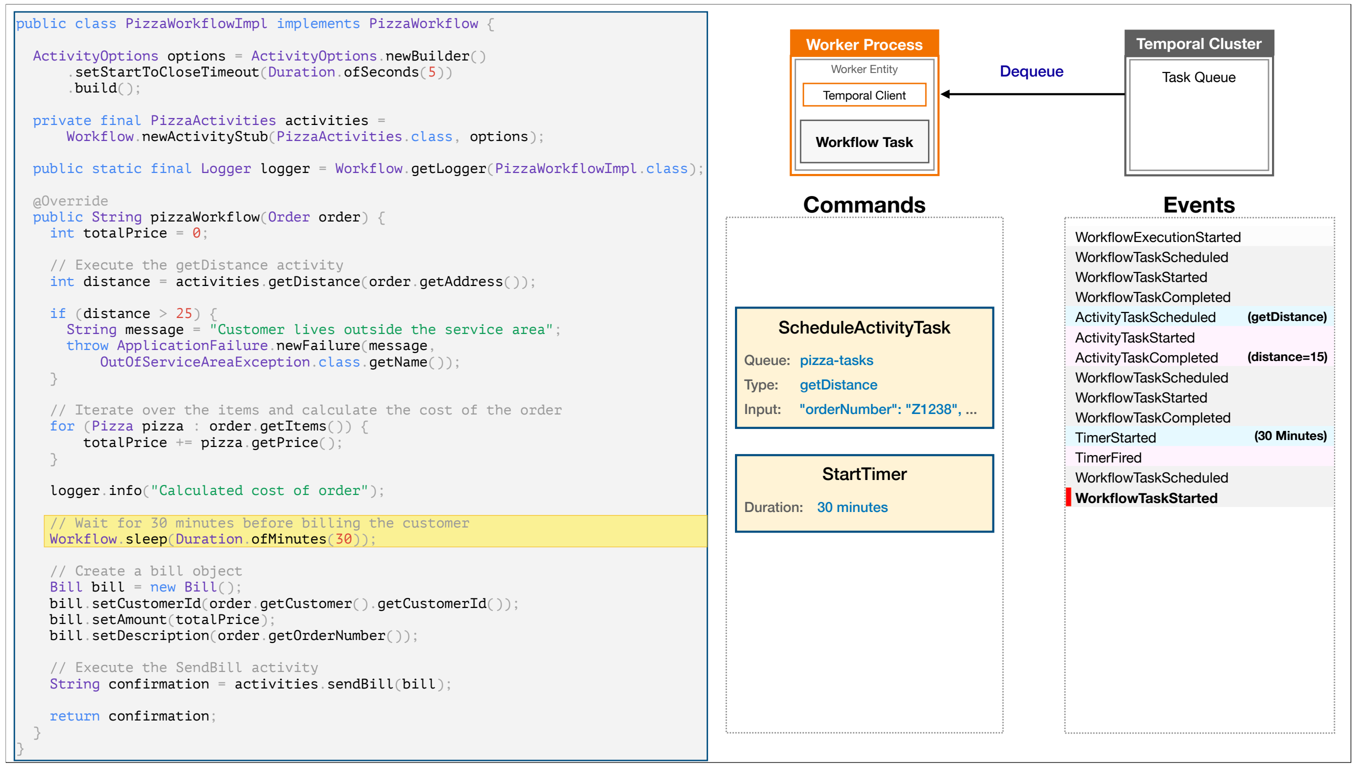
StartTimer

Duration: 30 minutes

Events

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

which the Worker will find when it polls again.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

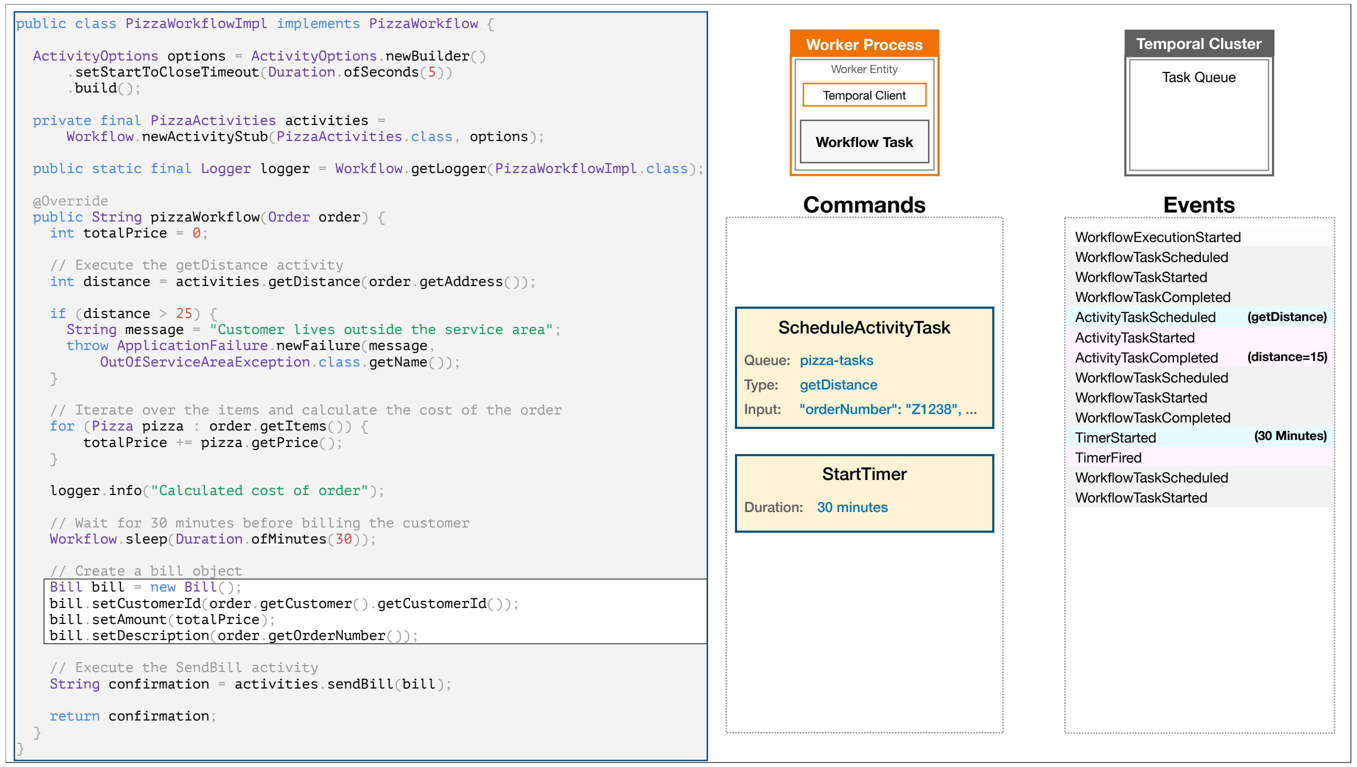
ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
Duration: 30 minutes

Events

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

After it accepts this Task,



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

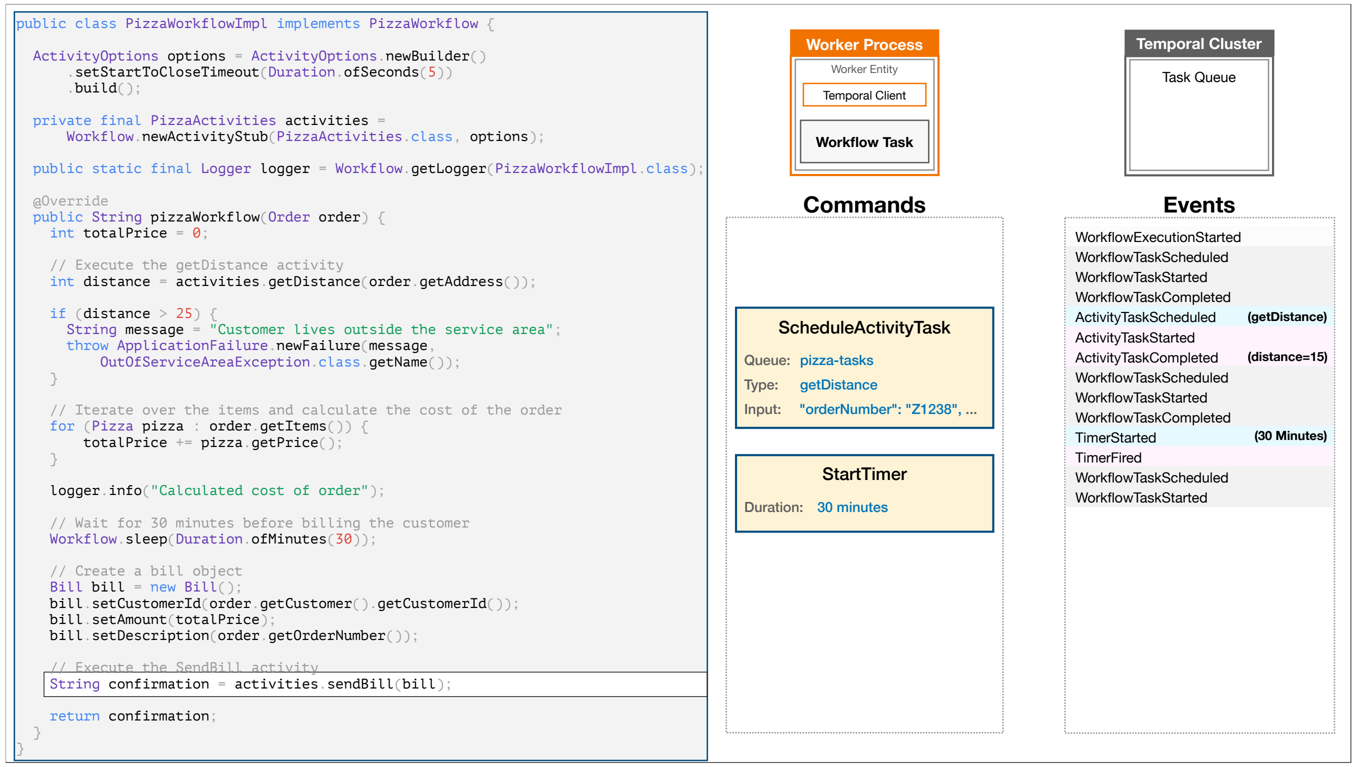
StartTimer

Duration: 30 minutes

Events

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

it continues execution of the Workflow code.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

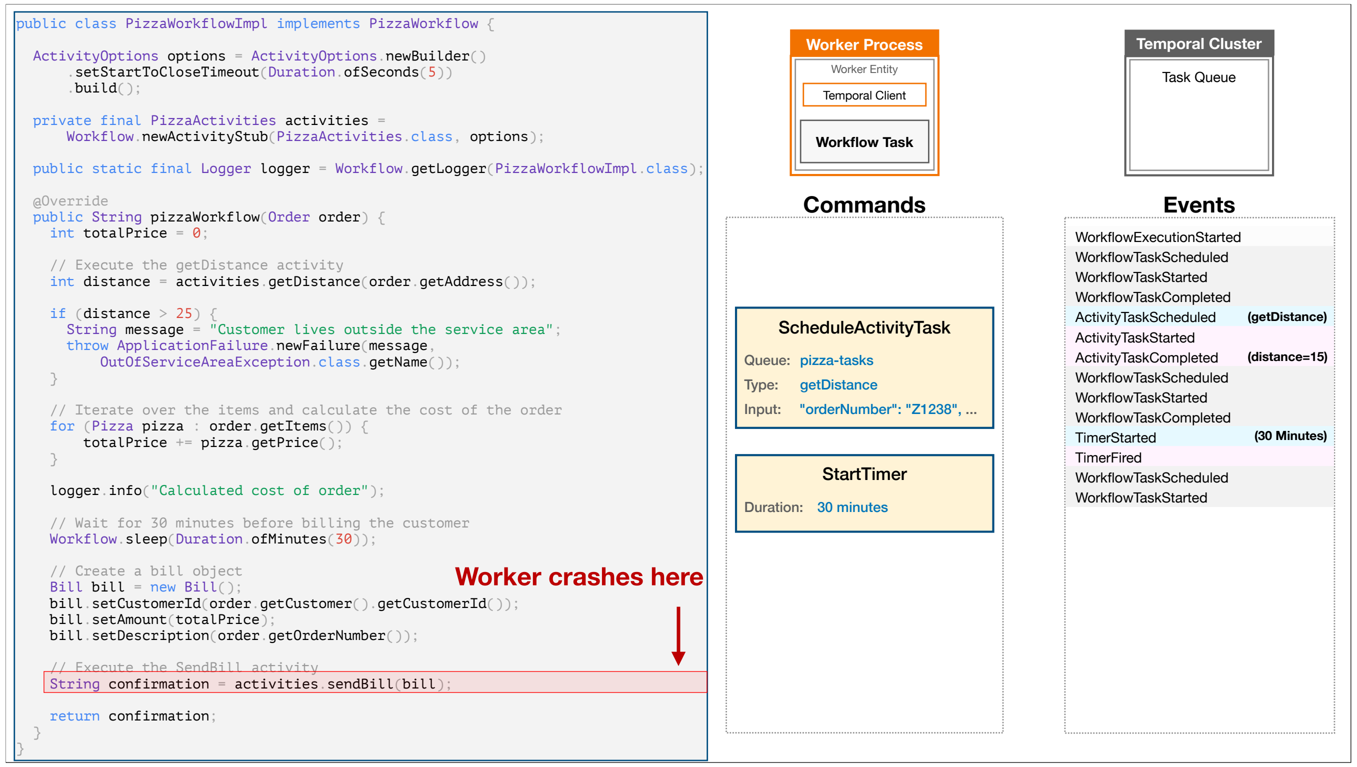
StartTimer

Duration: 30 minutes

Events

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

it continues execution of the Workflow code.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker crashes here

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

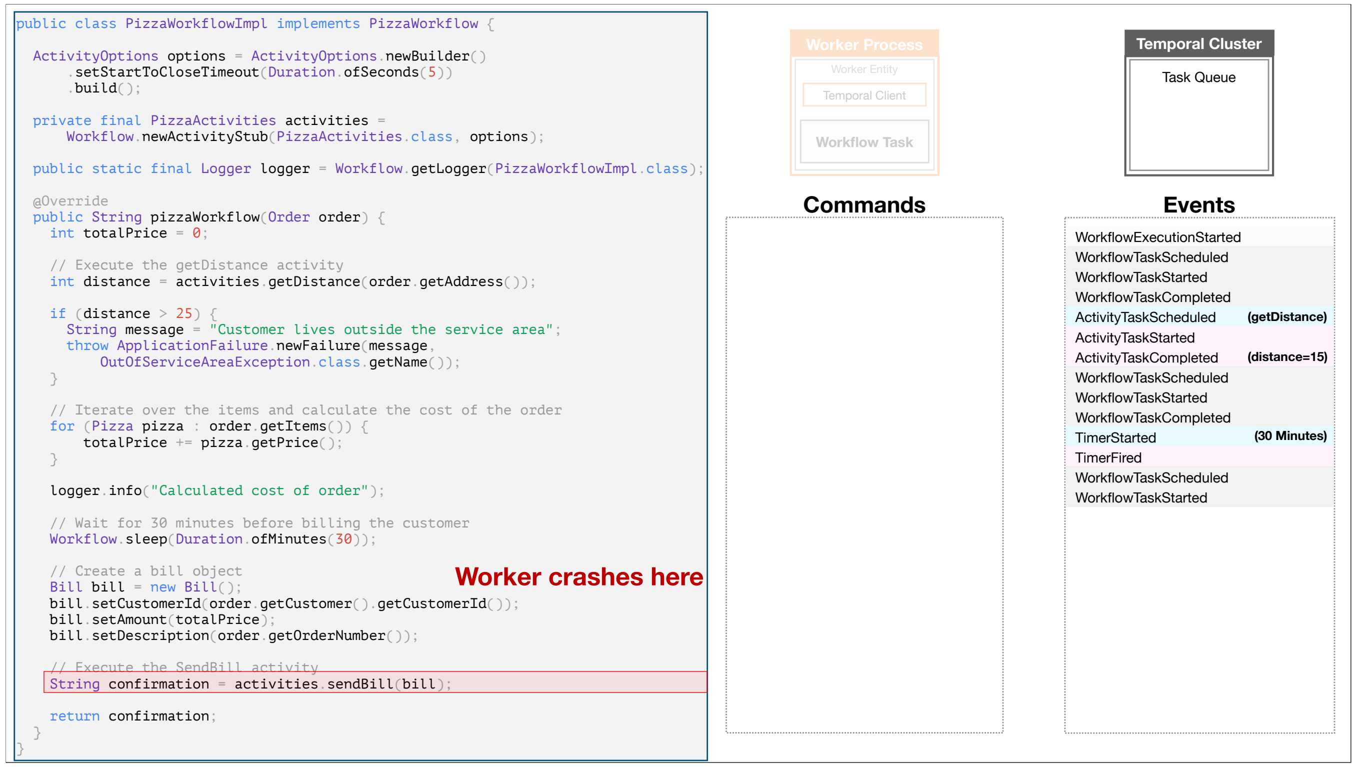
StartTimer

Duration: 30 minutes

Events

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

However, let’s suppose that the Worker happens to crash right here. How does Temporal recover the state of the Workflow?



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

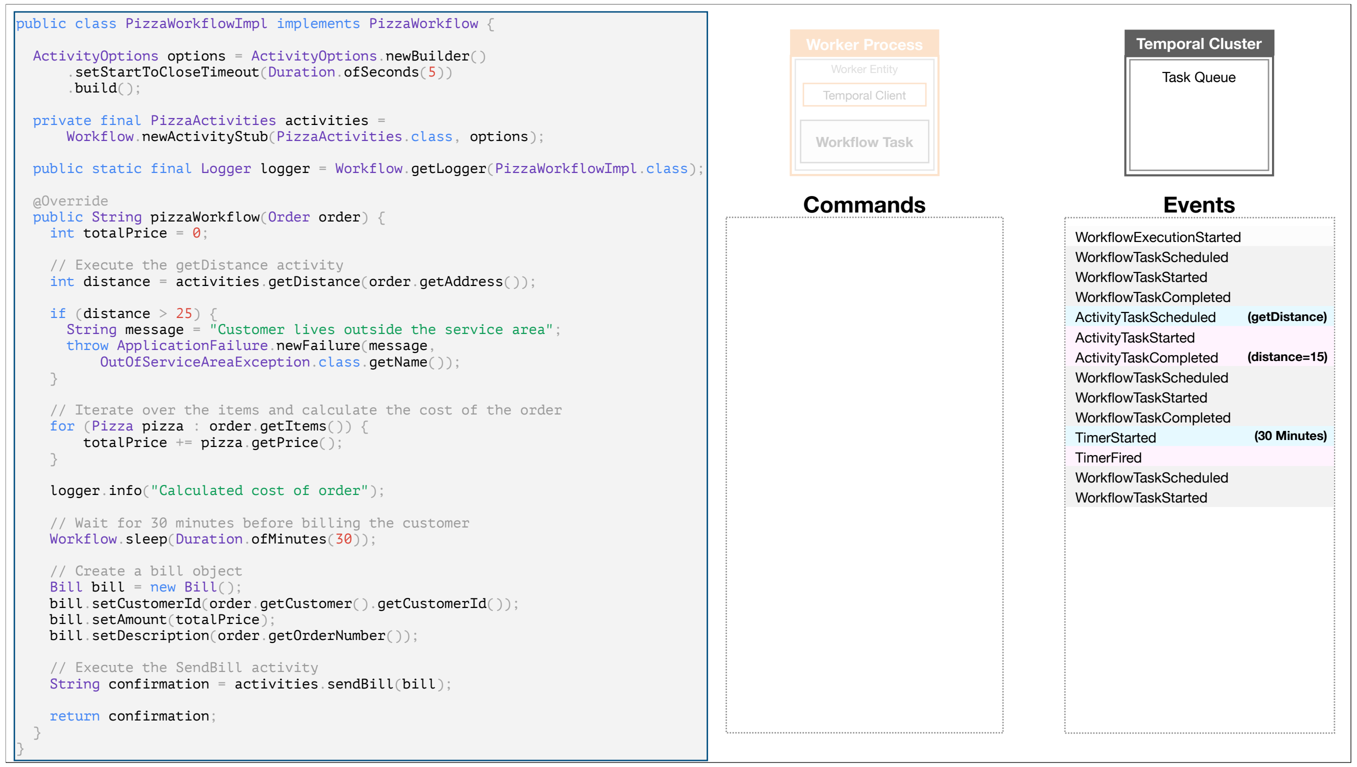
Worker crashes here

Commands

Events

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

Workers are external to the cluster and operate with autonomy. They long-poll the Task Queue, but only when seeking work to perform. They retrieve tasks from the queue, rather than being assigned tasks directly by the cluster.



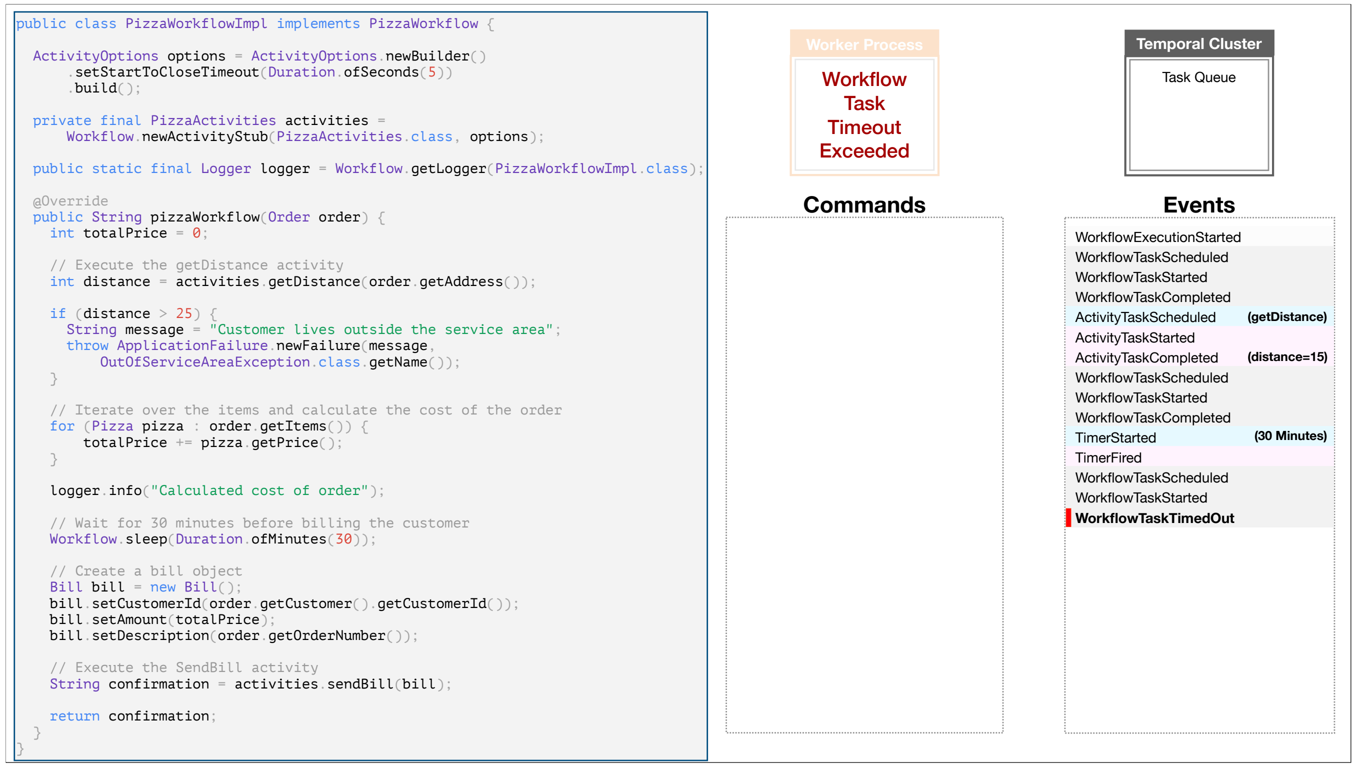
```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

Events

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

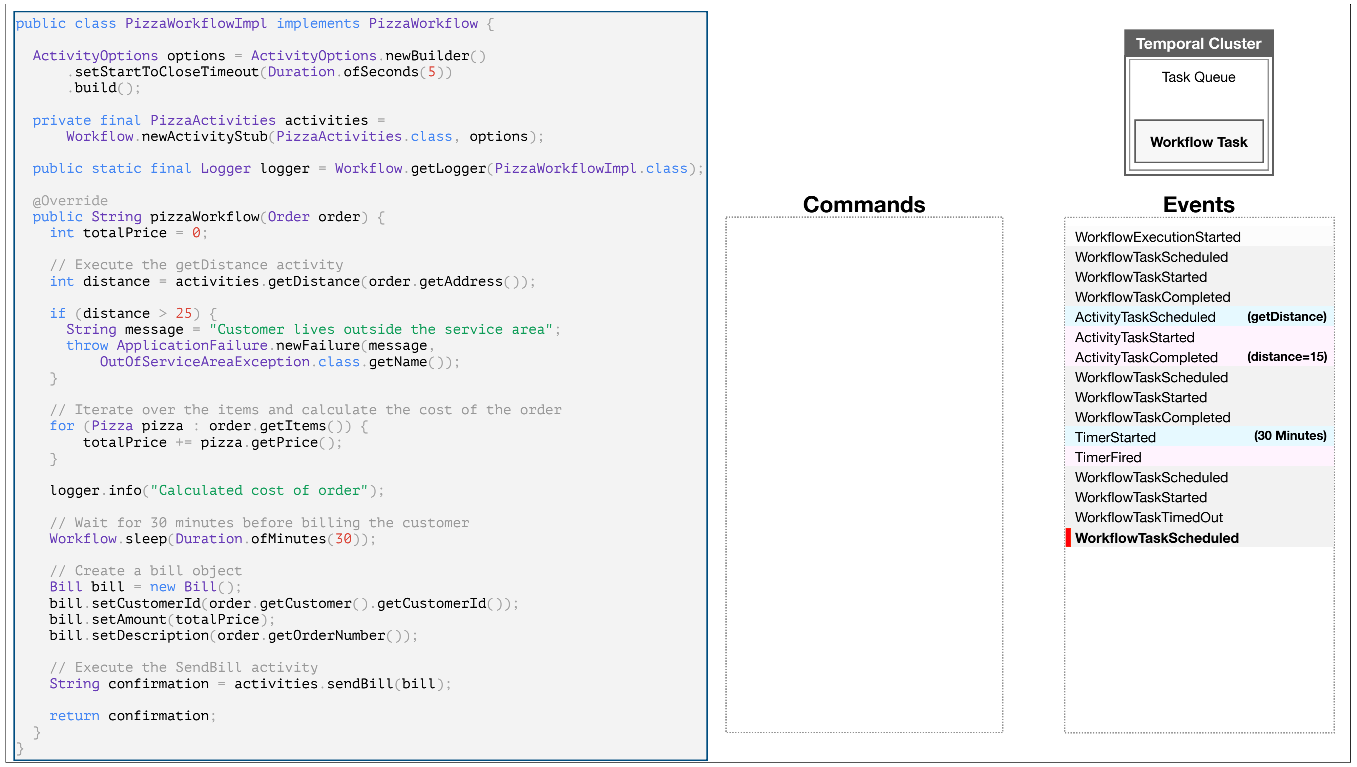
However, once a Worker has accepted a Task, it is expected to complete that task within a predefined duration, known as a Timeout.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

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

There are several types of Timeouts in Temporal, but since the Worker had a Workflow Task at the time of the crash, the relevant one in this case is the Workflow Task Timeout, which has a default value of 10 seconds.

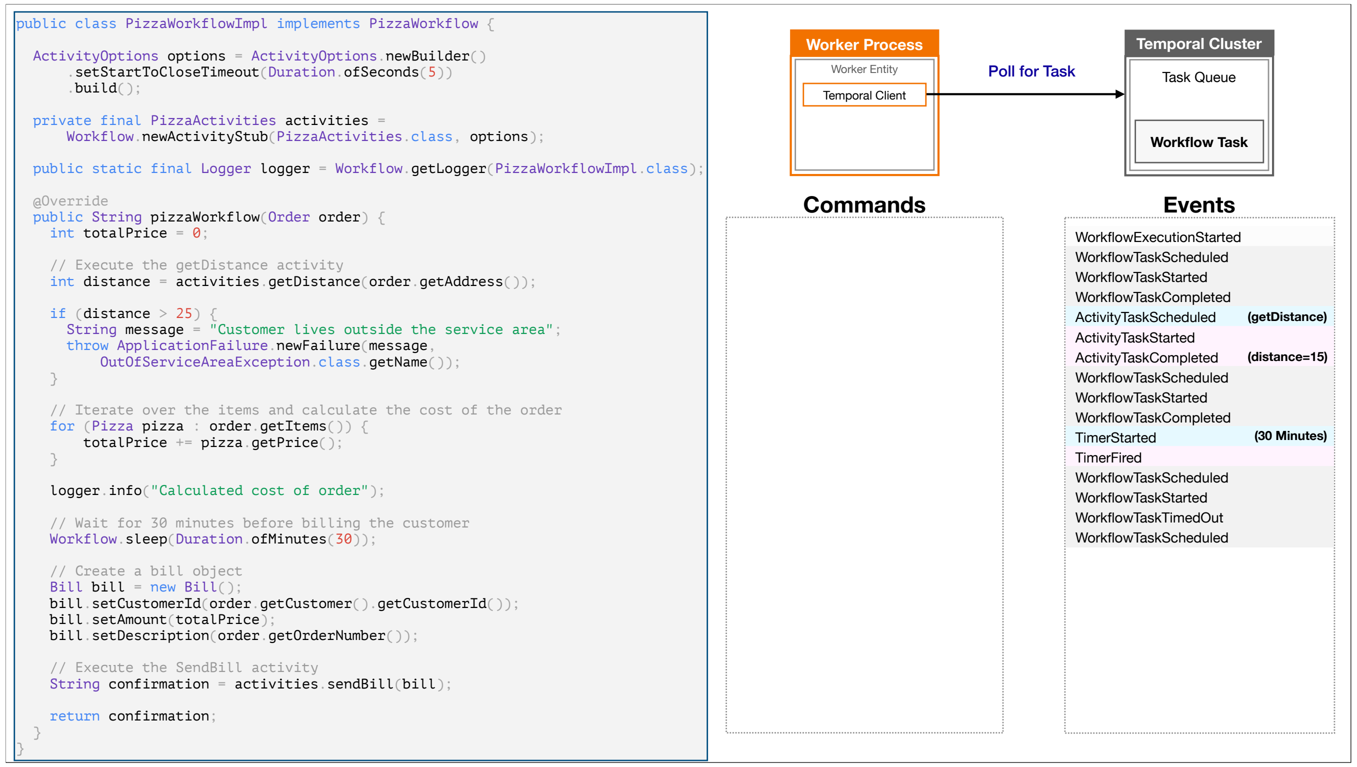


```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

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

Therefore, if the Worker failed to complete this Workflow Task within that time, the cluster assumes that the Worker has gone down, and will schedule a new Workflow Task.

Unlike the **original** Workflow Task, this one is not added to the "sticky queue" used to favor the Worker that previously accepted Workflow Tasks for this execution. Instead, it's placed into the Task Queue specified when Workflow Execution began, which means that it will be available to any available Worker polling this Task Queue.



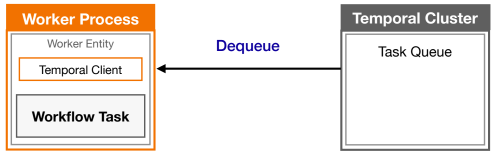
```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Events

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

This Task will remain in the queue until another Worker polls and accepts it. That might be done by another one that's already running in the Worker fleet or by a new Worker Process created by restarting the one that crashed.

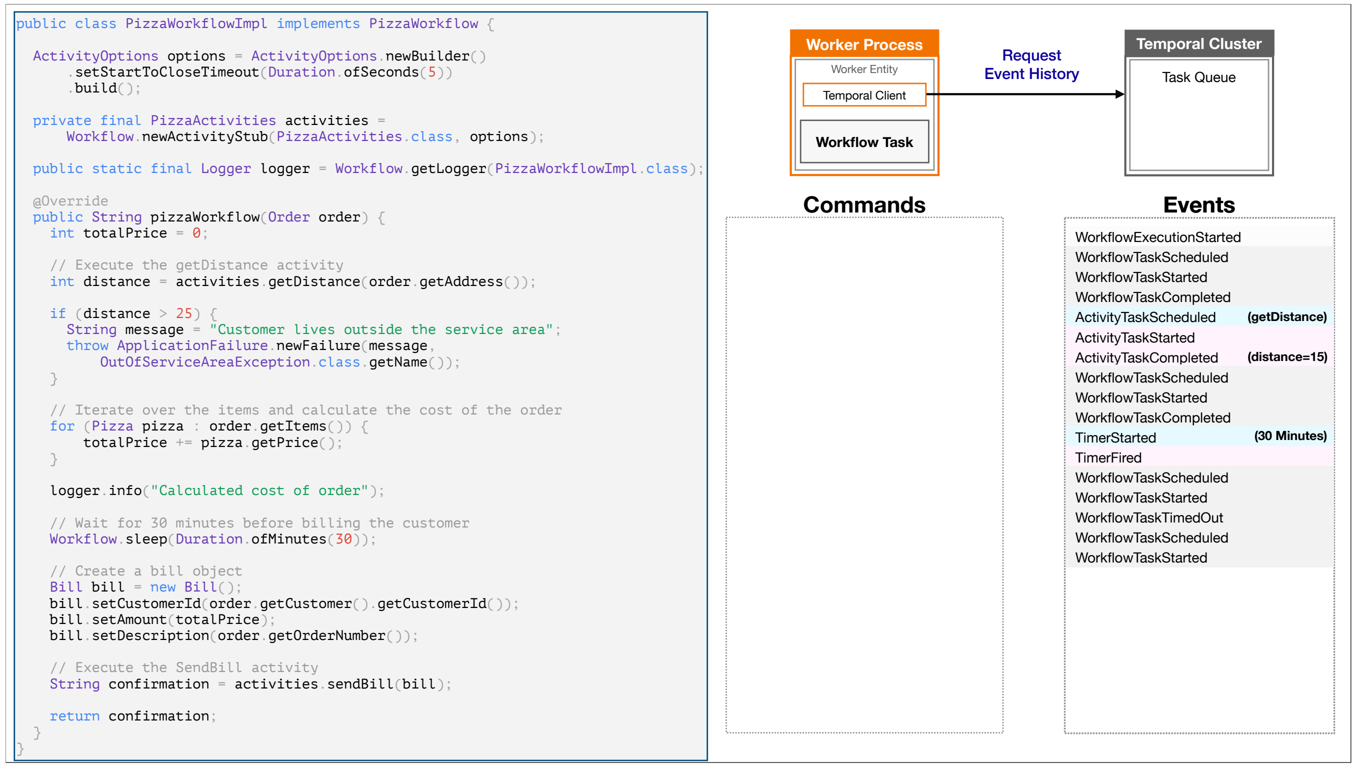
```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```



Commands

Events

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



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Workflow Task

Request
Event History

Temporal Cluster

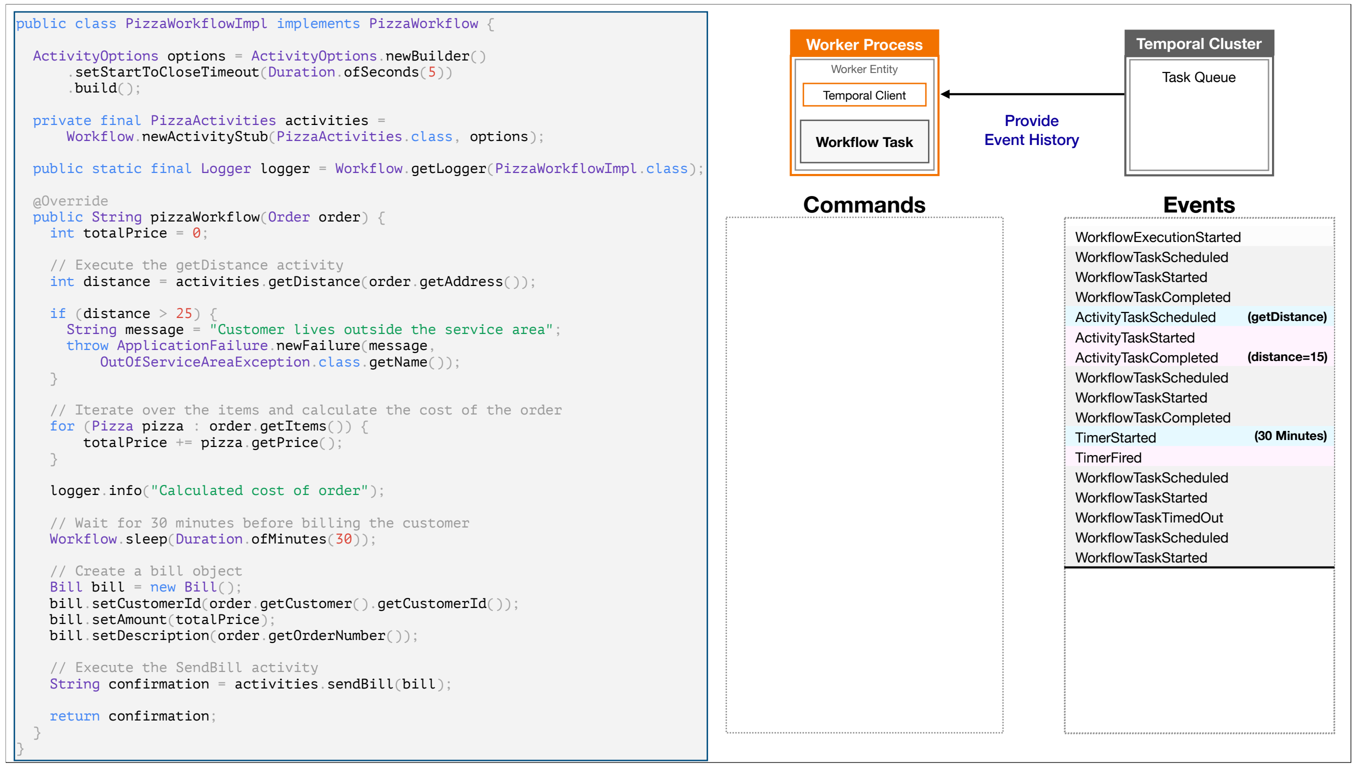
Task Queue

Commands

Events

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

In either case, the Worker will need the current Event History for this execution, so it requests it from the cluster.

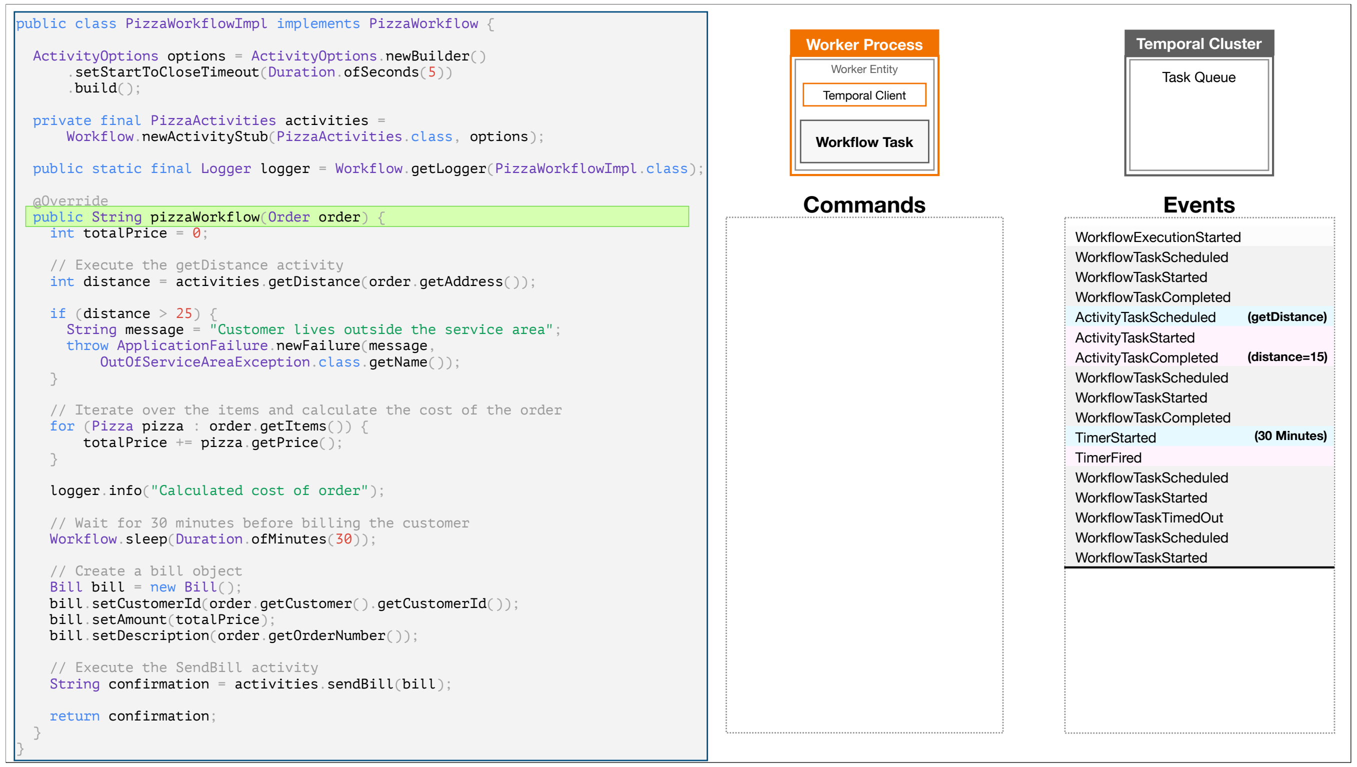


```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
     Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

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

The Worker streams the Event History from the cluster.

I've added a black horizontal line in the column on the right to indicate the final Event in the History at the time of the Worker crash.

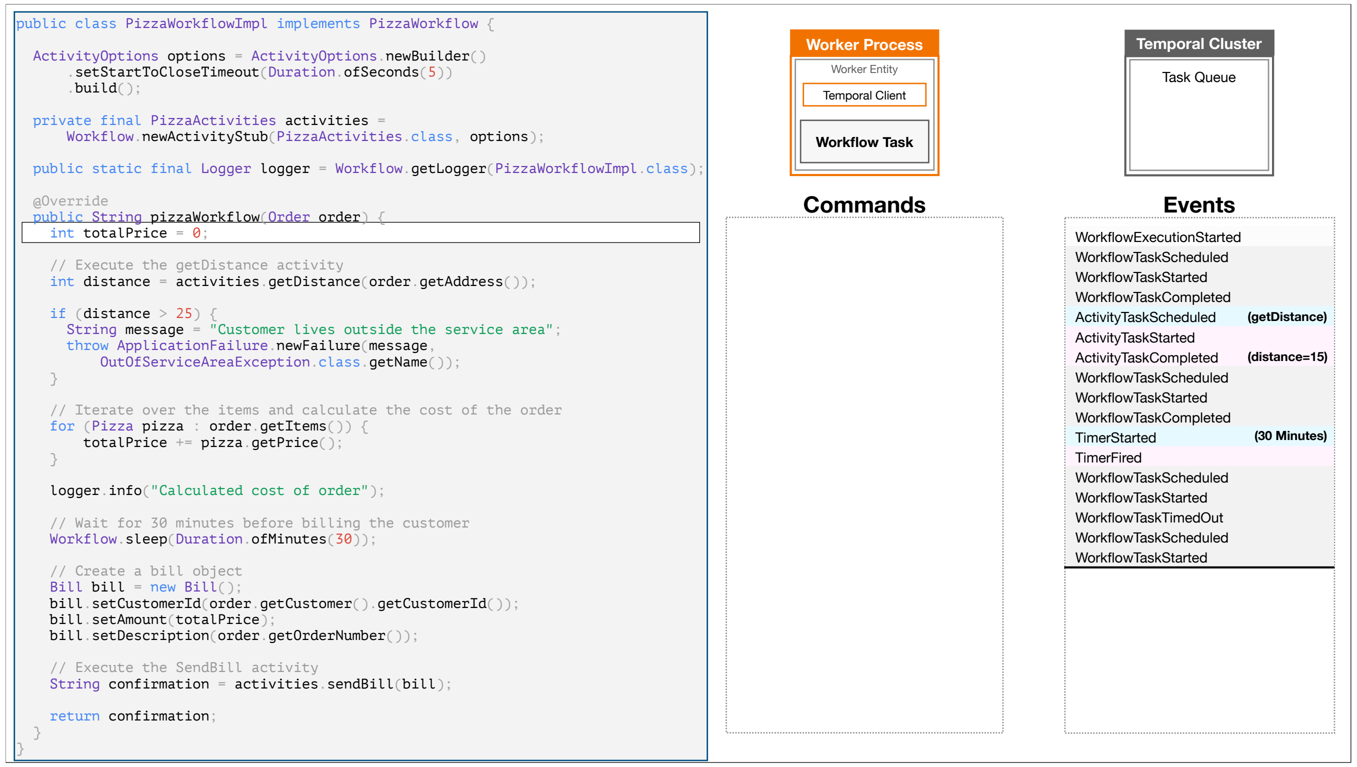


```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

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

The Worker then begins a re-execution of the code, using the same input, which was stored in the `WorkflowExecutionStarted` Event.

With a couple of exceptions that I'll point out along the way, it does the same thing as when the previous Worker executed the code before the crash.

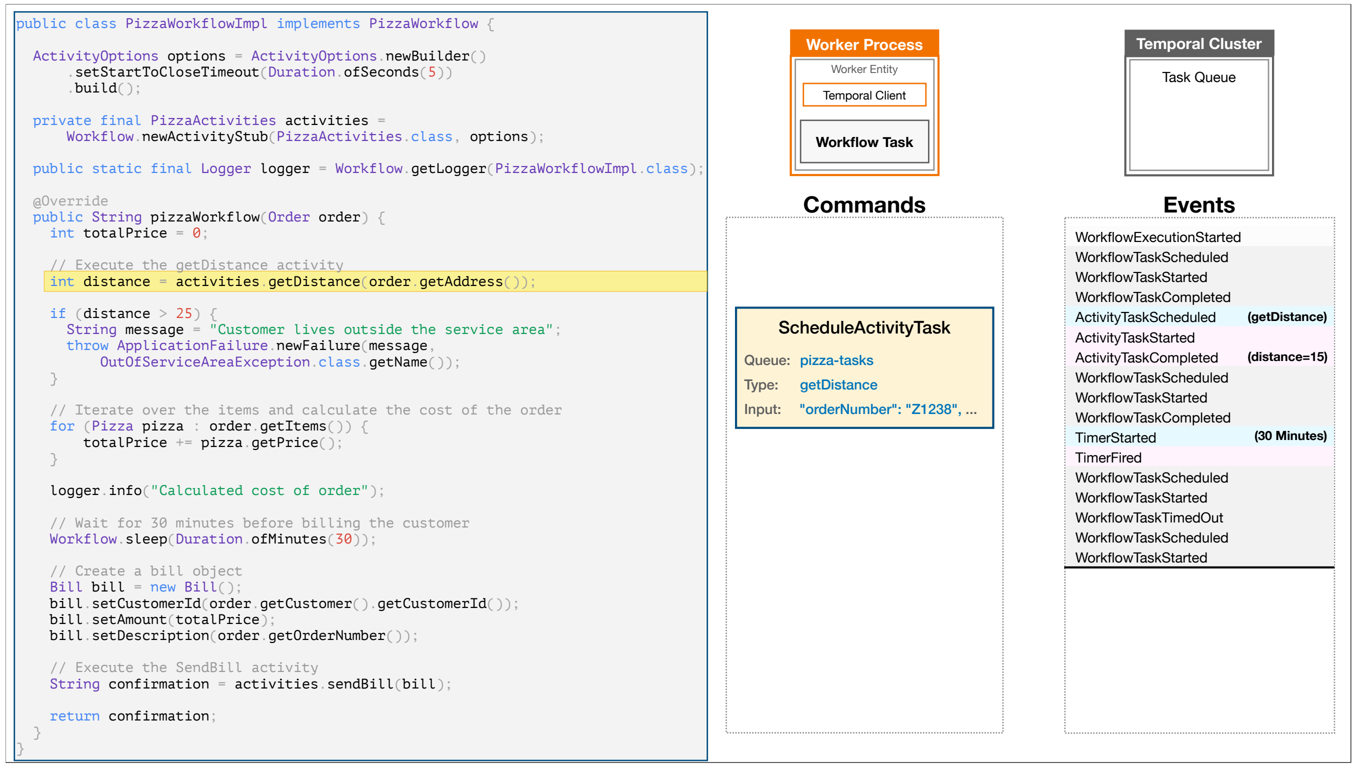


```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Events

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

By the way, because the Workflow code is deterministic, the state of all variables encountered so far is identical to what it was before the crash.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

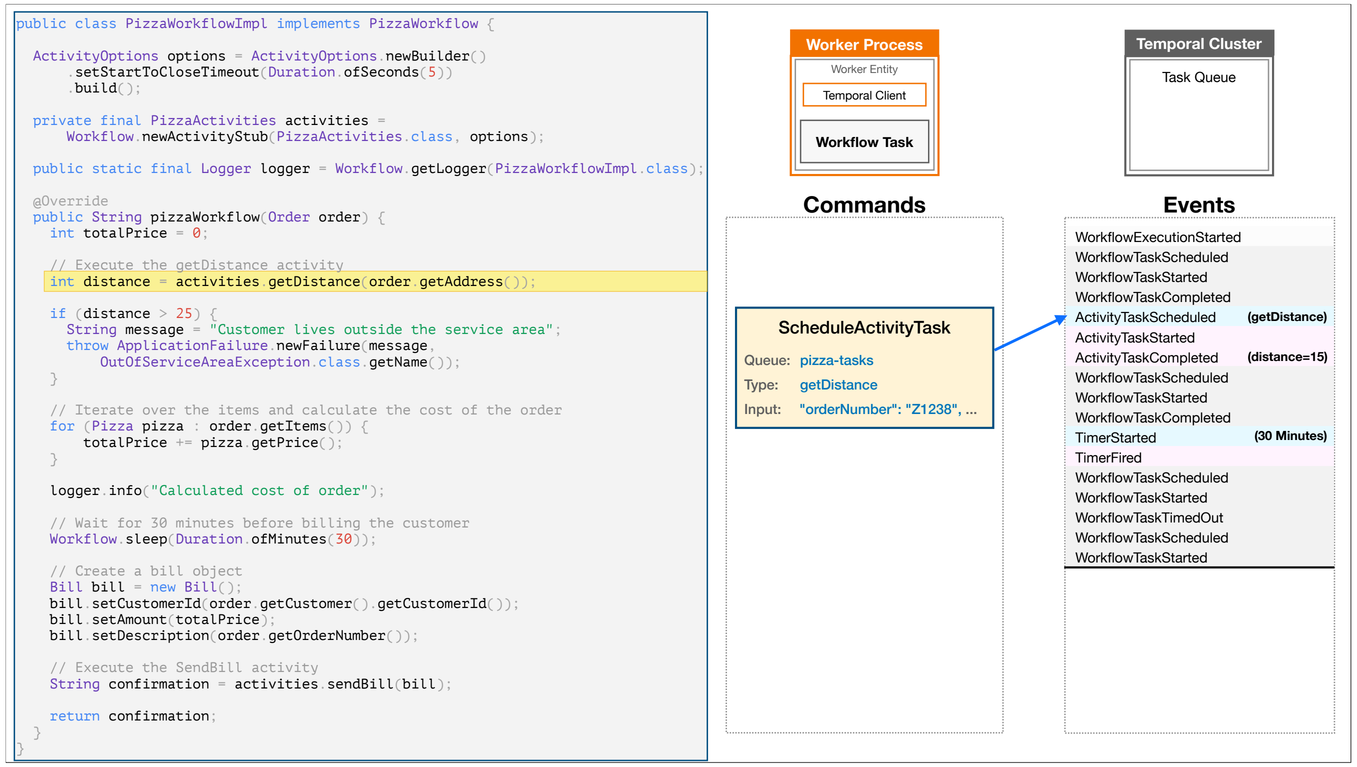
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

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

When it reaches the call to schedule the Activity, it creates a Command, but does not issue it to the cluster.

Instead, it inspects the Event History and finds three Events related to this Activity.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Workflow Task

Temporal Cluster

Task Queue

Commands

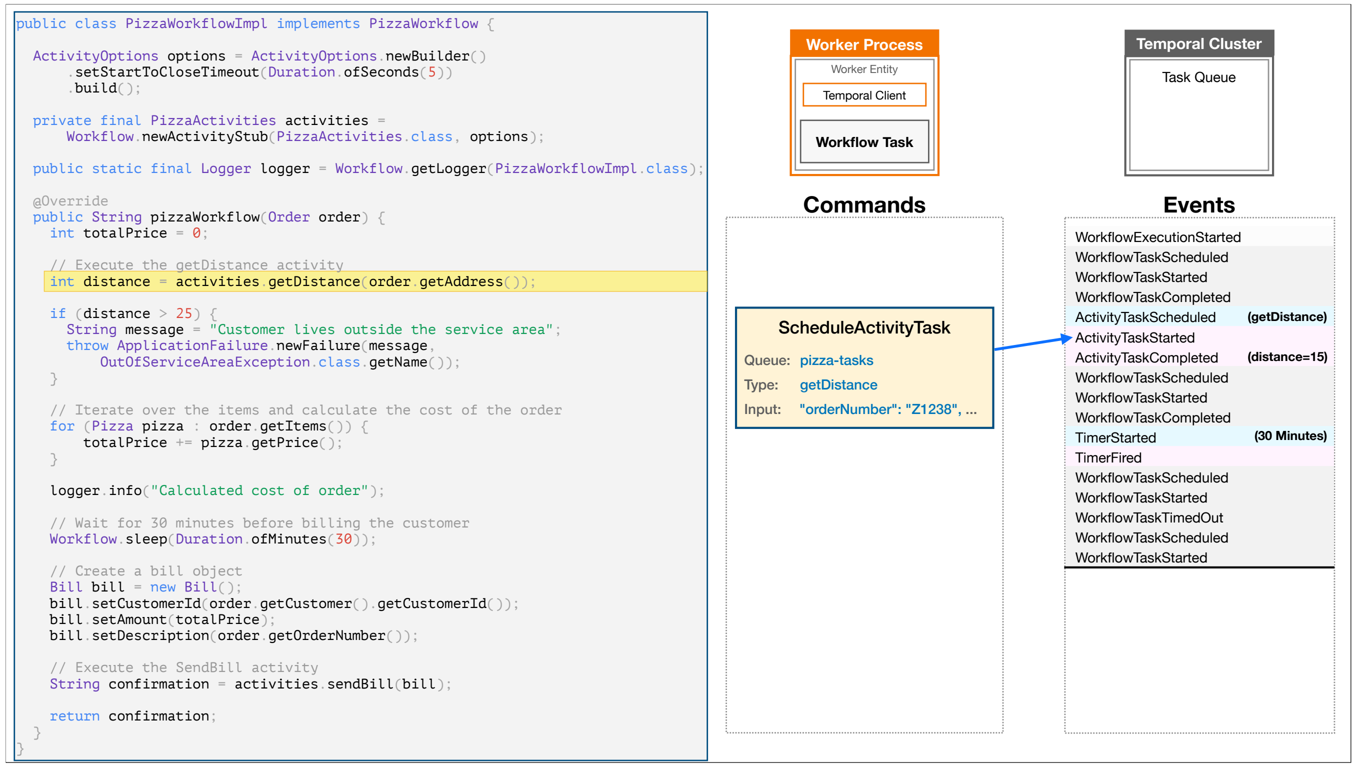
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

The first one indicates that the Task was previously scheduled by the cluster,



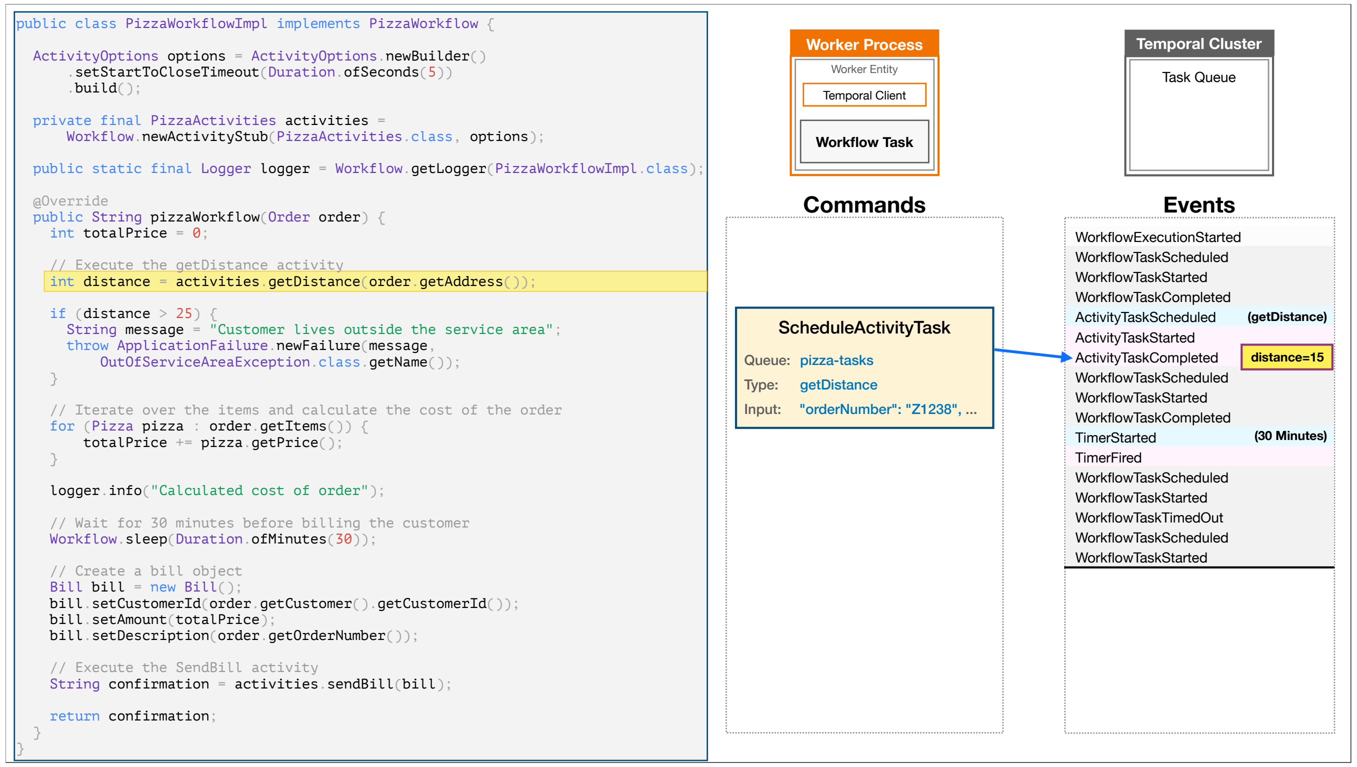
```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

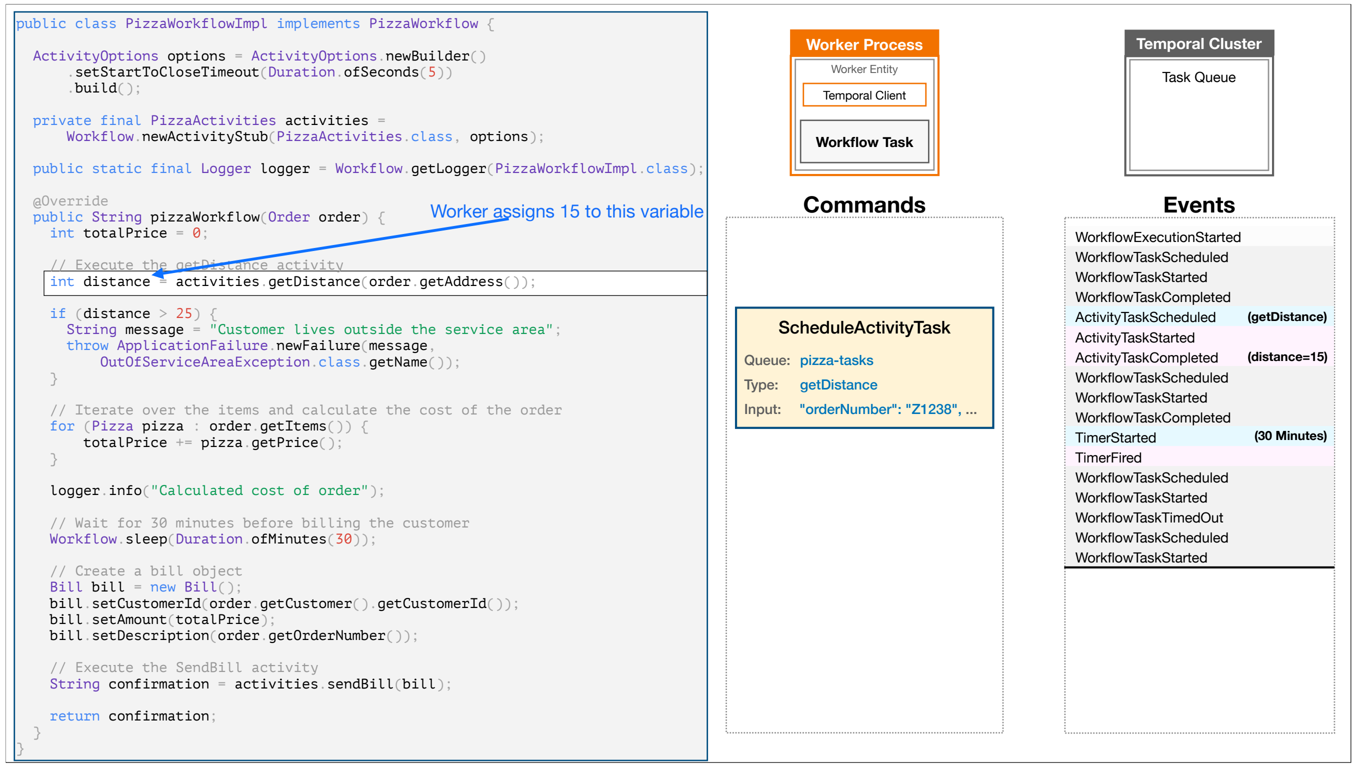
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted
ActivityTaskCompleted (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted (30 Minutes)
TimerFired
WorkflowTaskScheduled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

The second indicates that a Worker dequeued the Task,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

and the third indicates that the Worker successfully completed the Task for the `getDistance` Activity, having returned a value of `15`.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker assigns 15 to this variable

Commands

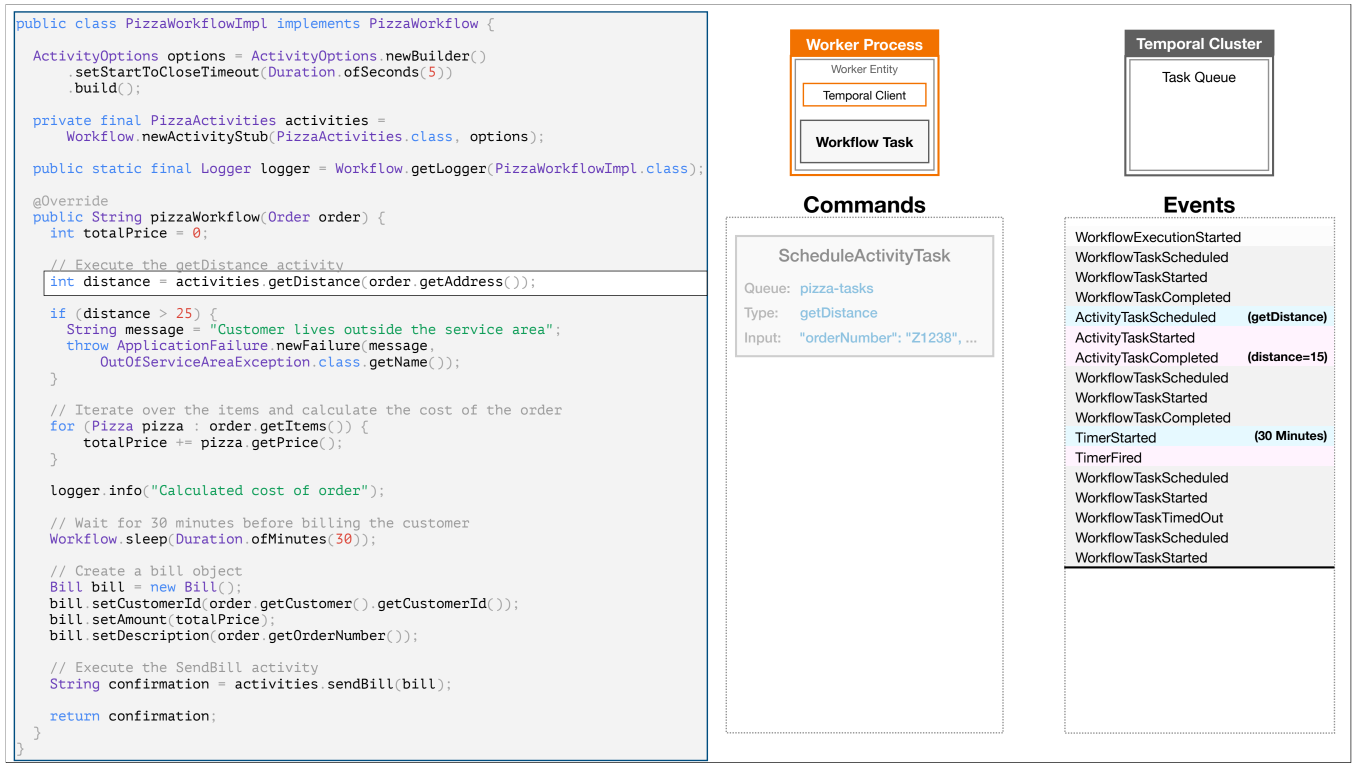
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

Instead of executing the Activity again during replay, the Worker assigns the value returned by the __previous__ execution.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

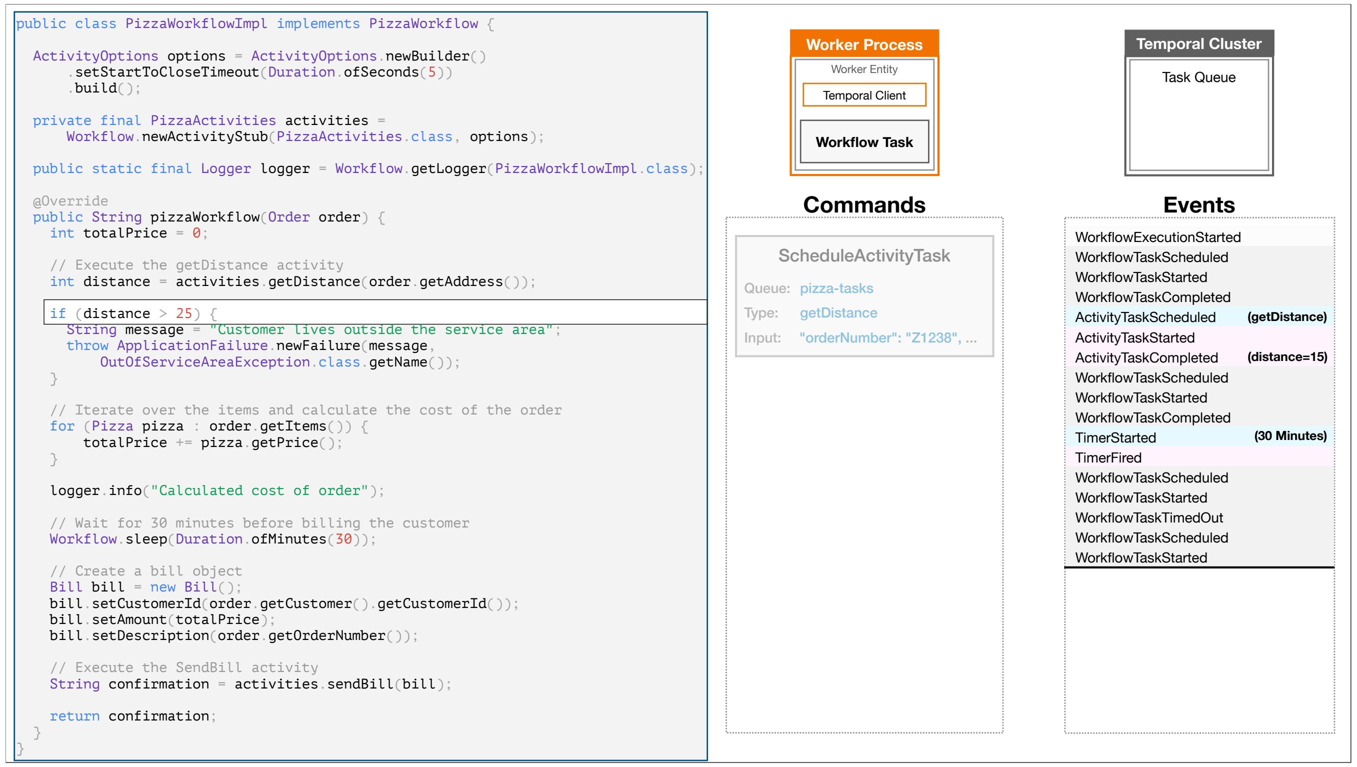
Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

This also means that the Worker has no need to issue the Command to theTemporal Cluster. While Activity code isn't required to be deterministic, the fact that the Worker reuses the result stored in the `ActivityTaskCompleted` Event from the original execution eliminates the possibility that an Activity could behave differently during History Replay than it did originally.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

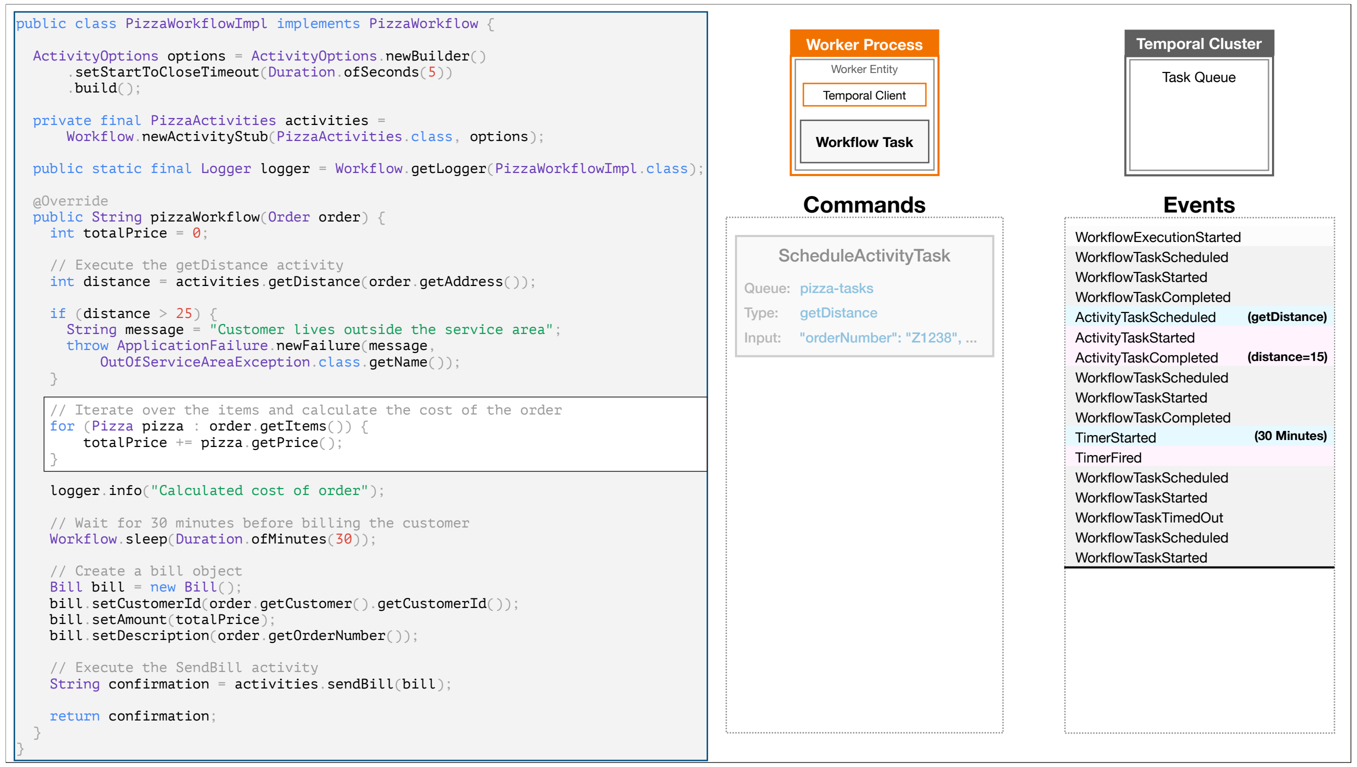
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

The execution of each statement helps to restore the previous state of the Workflow.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

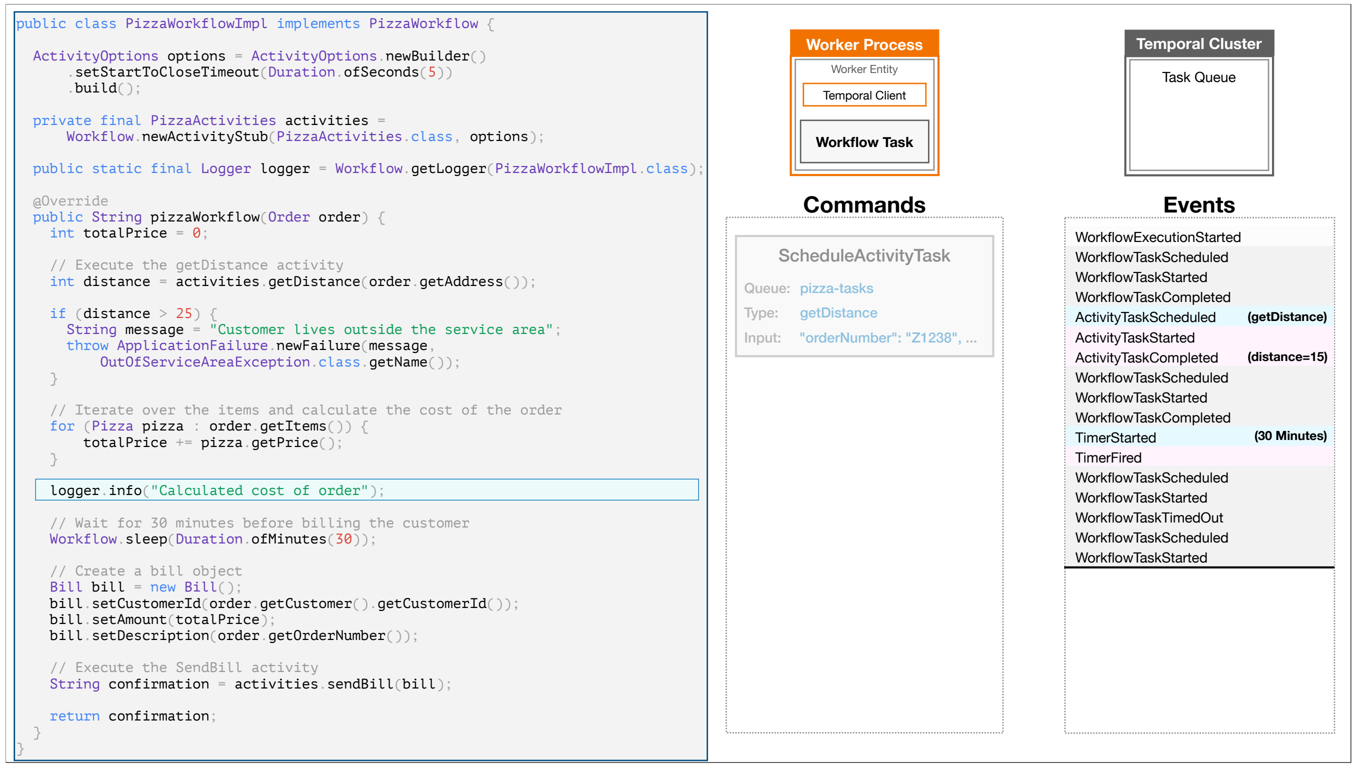
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

For example, the total gets recomputed.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

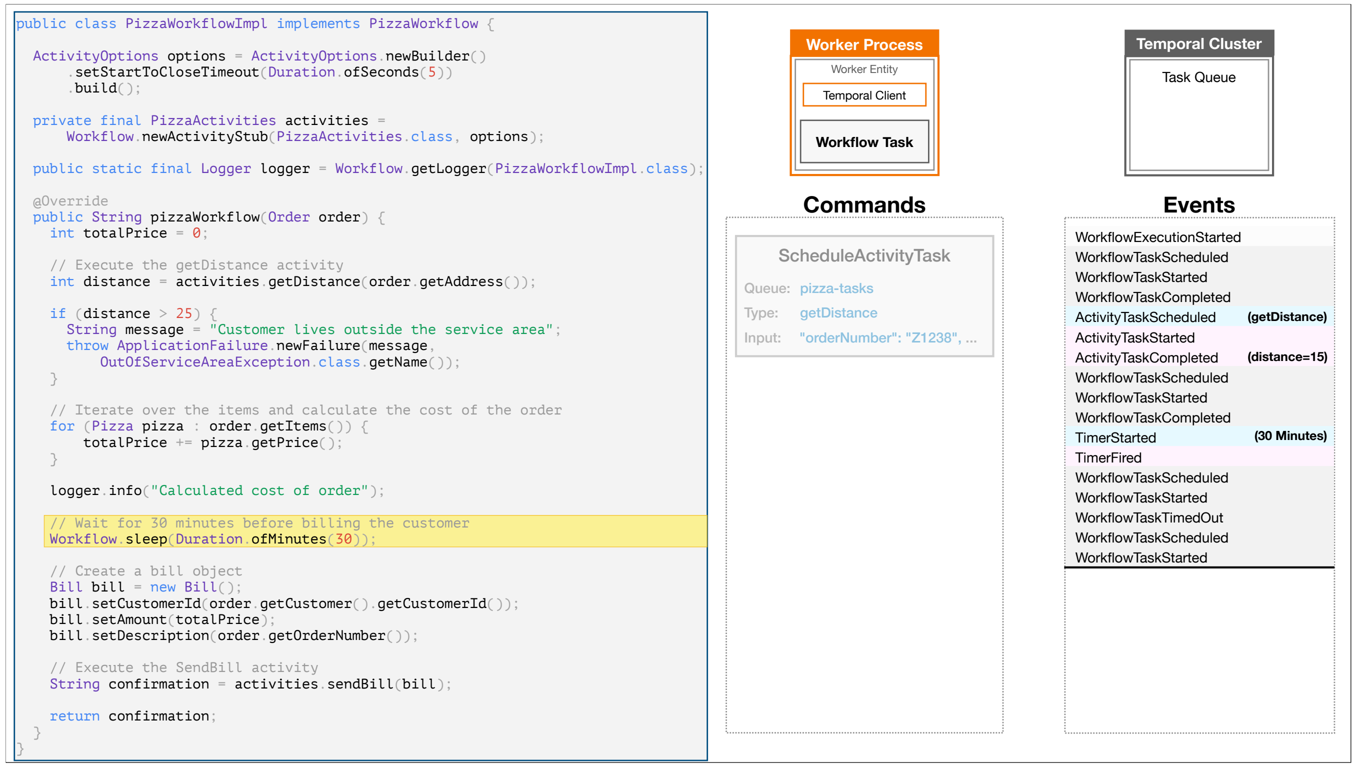
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

This logging statement is one of the things that behaves differently during replay. Temporal's logger is replay-aware, so it suppresses output during replay so that the logs won't show duplicate messages.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

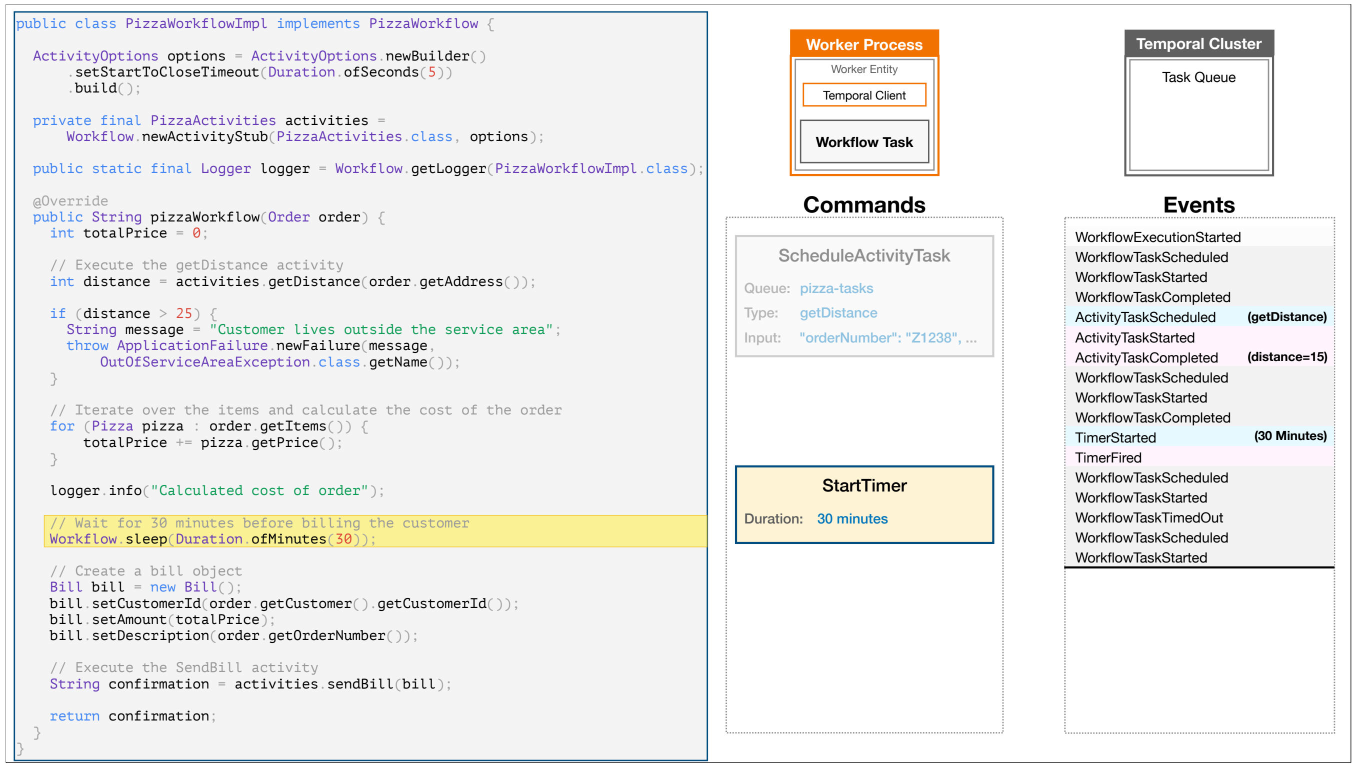
ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

Events

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

When the Worker reaches the `workflow.Sleep` statement, it evaluates the Event History as it did with the Activity.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks

Type: getDistance

Input: "orderNumber": "Z1238", ...

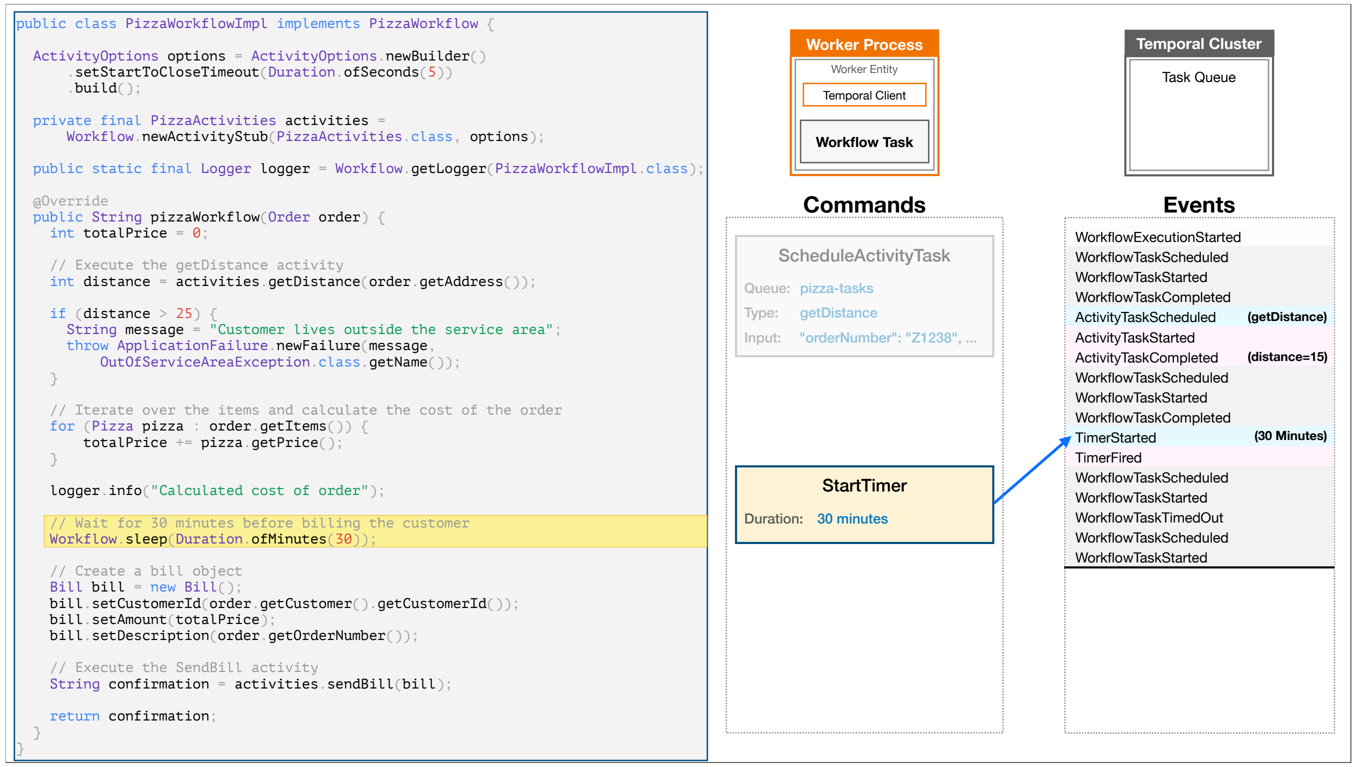
StartTimer

Duration: 30 minutes

Events

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

It creates a Command



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

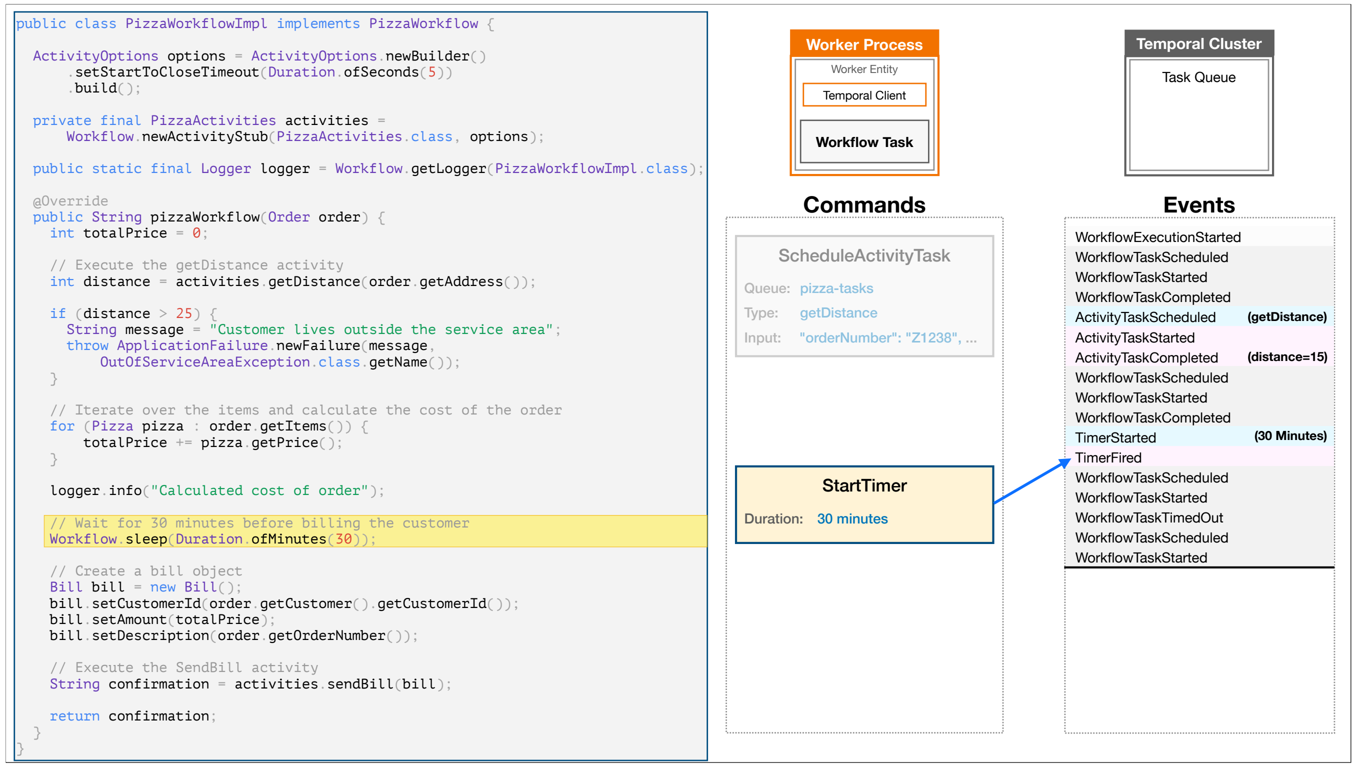
StartTimer

Duration: 30 minutes

Events

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

and then checks the Event History to see whether the Timer was started



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Workflow Task

Temporal Cluster

Task Queue

Commands

ScheduleActivityTask

Queue: pizza-tasks

Type: getDistance

Input: "orderNumber": "Z1238", ...

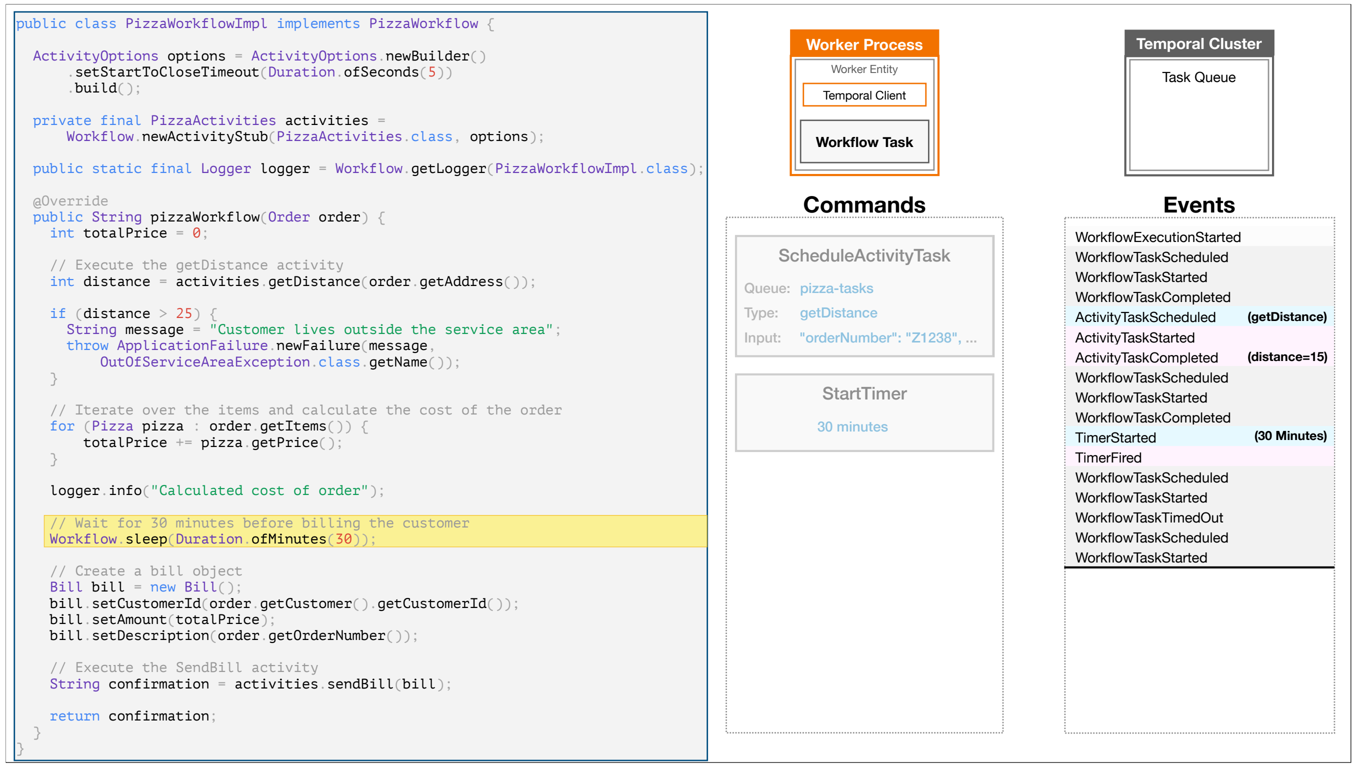
StartTimer

Duration: 30 minutes

Events

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

and fired during the previous execution.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

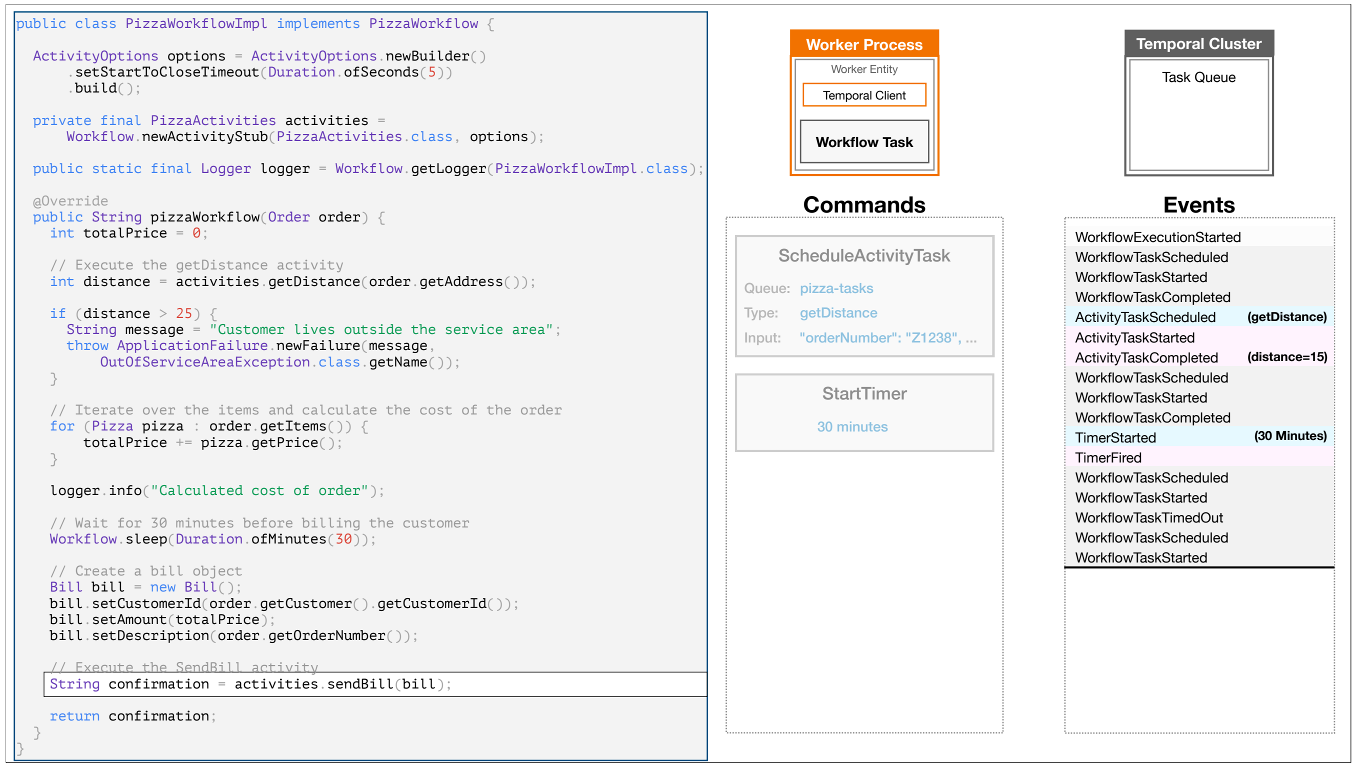
ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
30 minutes

Events

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

Since the history indicates that both of these things happened, the Worker does not issue the Command to the cluster.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

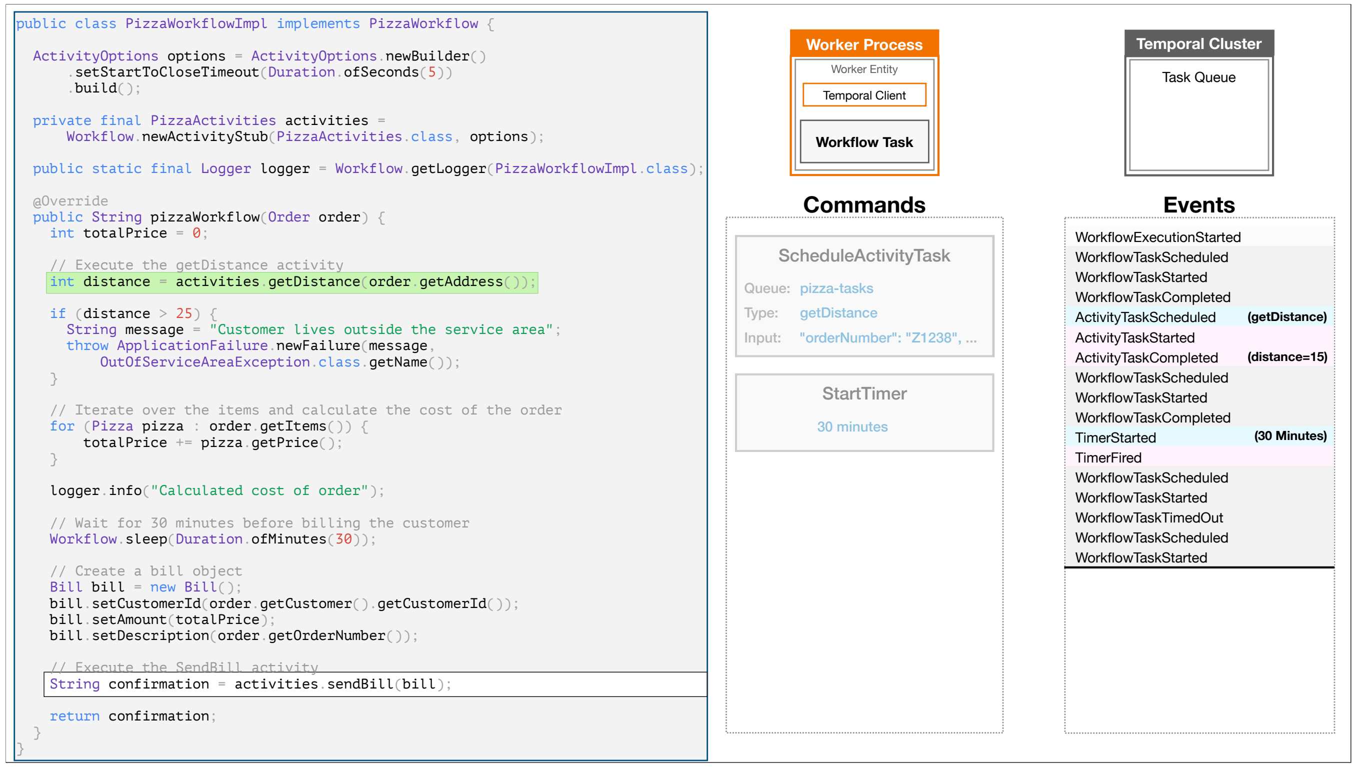
ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
30 minutes

Events

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

At this point, the Worker has reached the point where the crash occurred, and replaying the code has completely restored the state of the Workflow prior to the crash



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

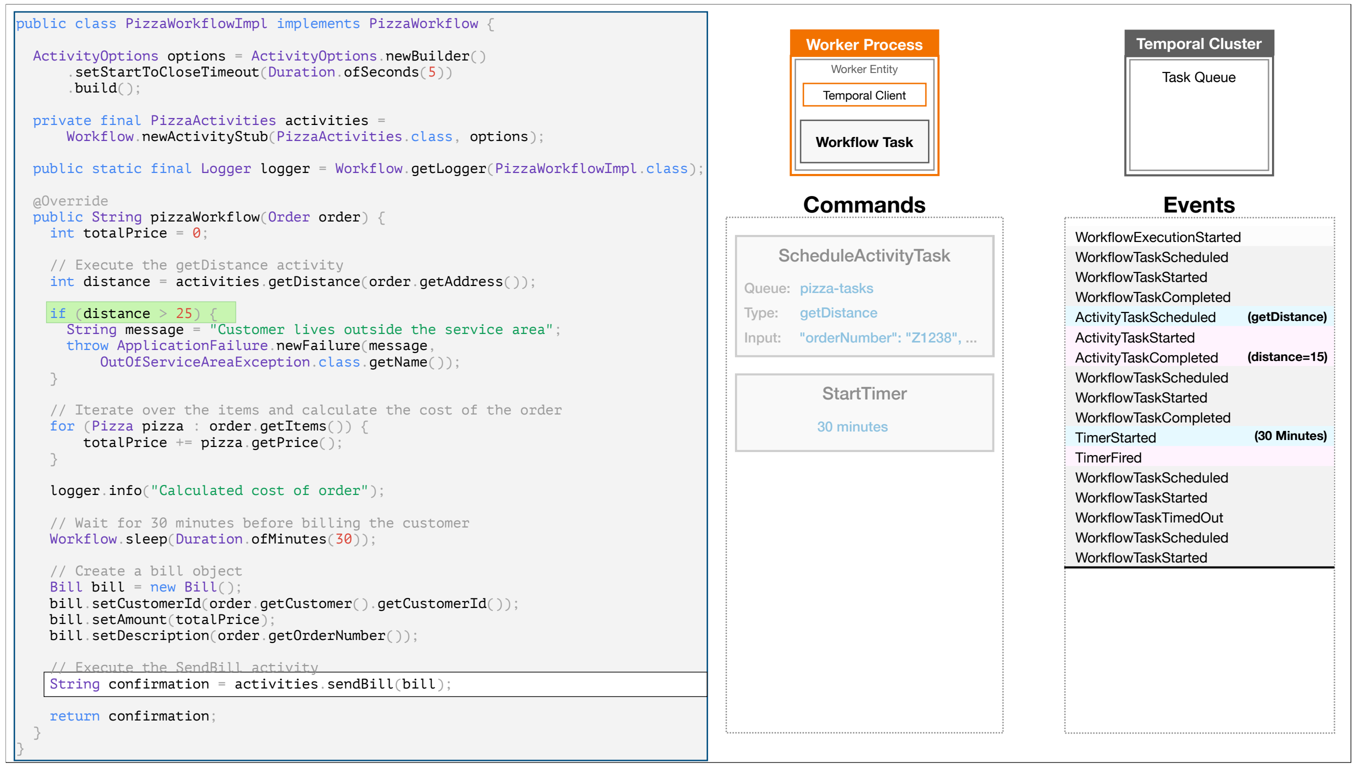
ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
30 minutes

Events

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

For example, the distance variable has the same value it did prior to the crash, which was originally returned by executing the `getDistance` Activity.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

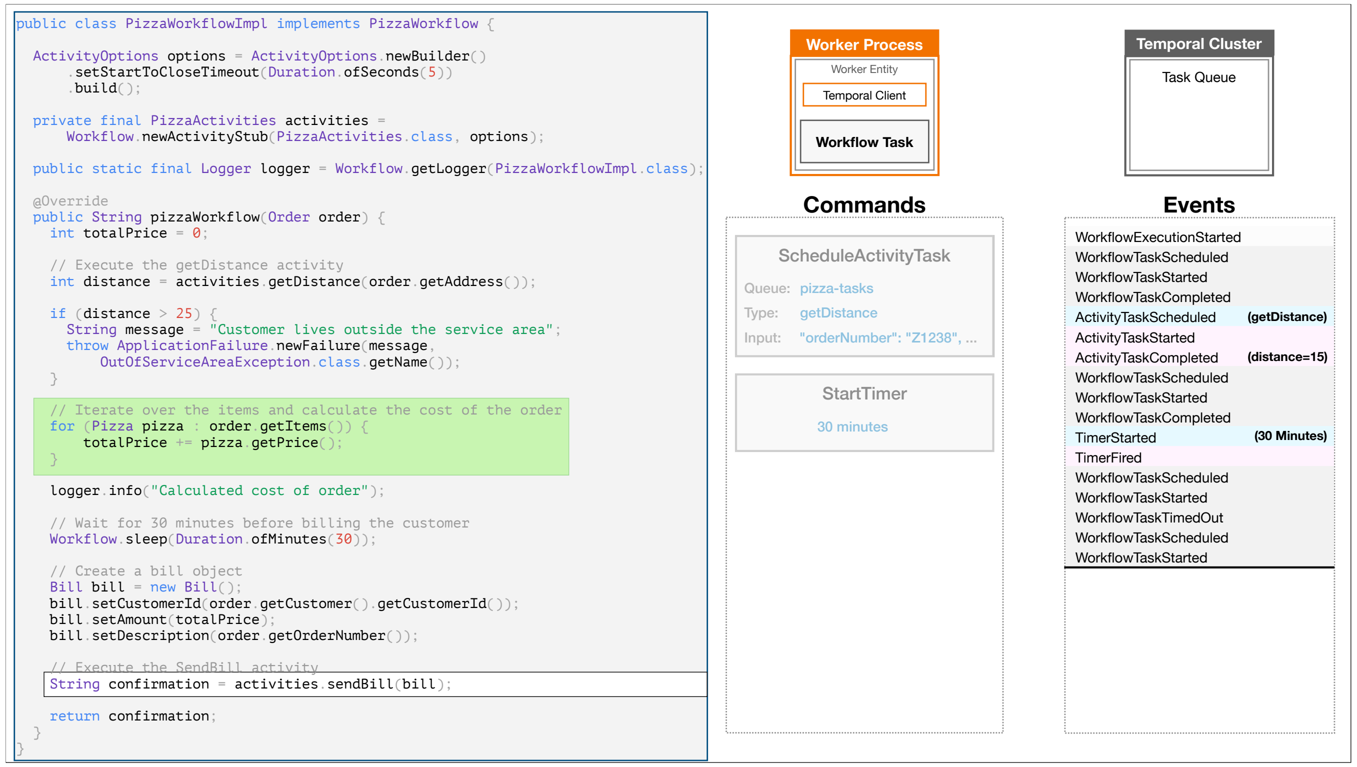
ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
30 minutes

Events

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

Since replay uses the same input data as before, this also means that the conditional statement on line 20 evaluates to `false`, just like it did before.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

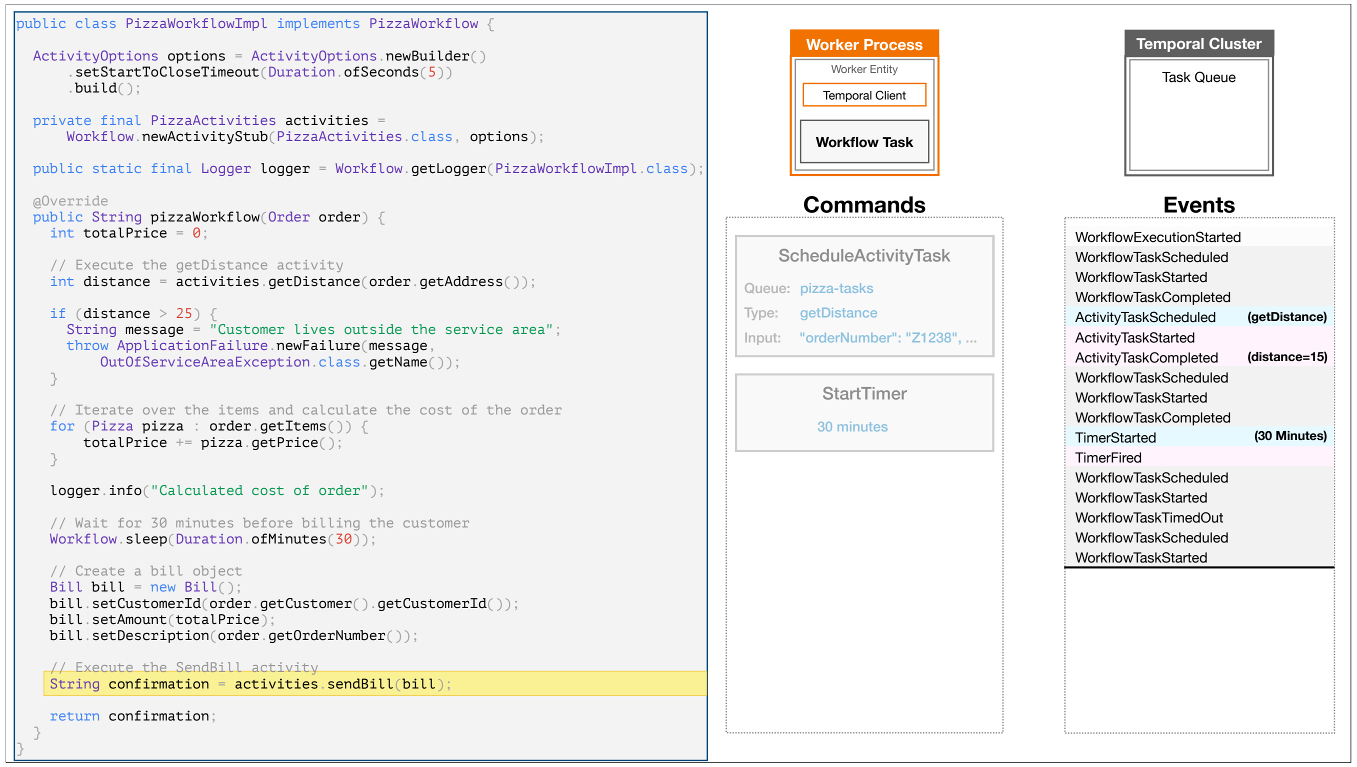
StartTimer

30 minutes

Events

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

It has the same value it did as well.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
      totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

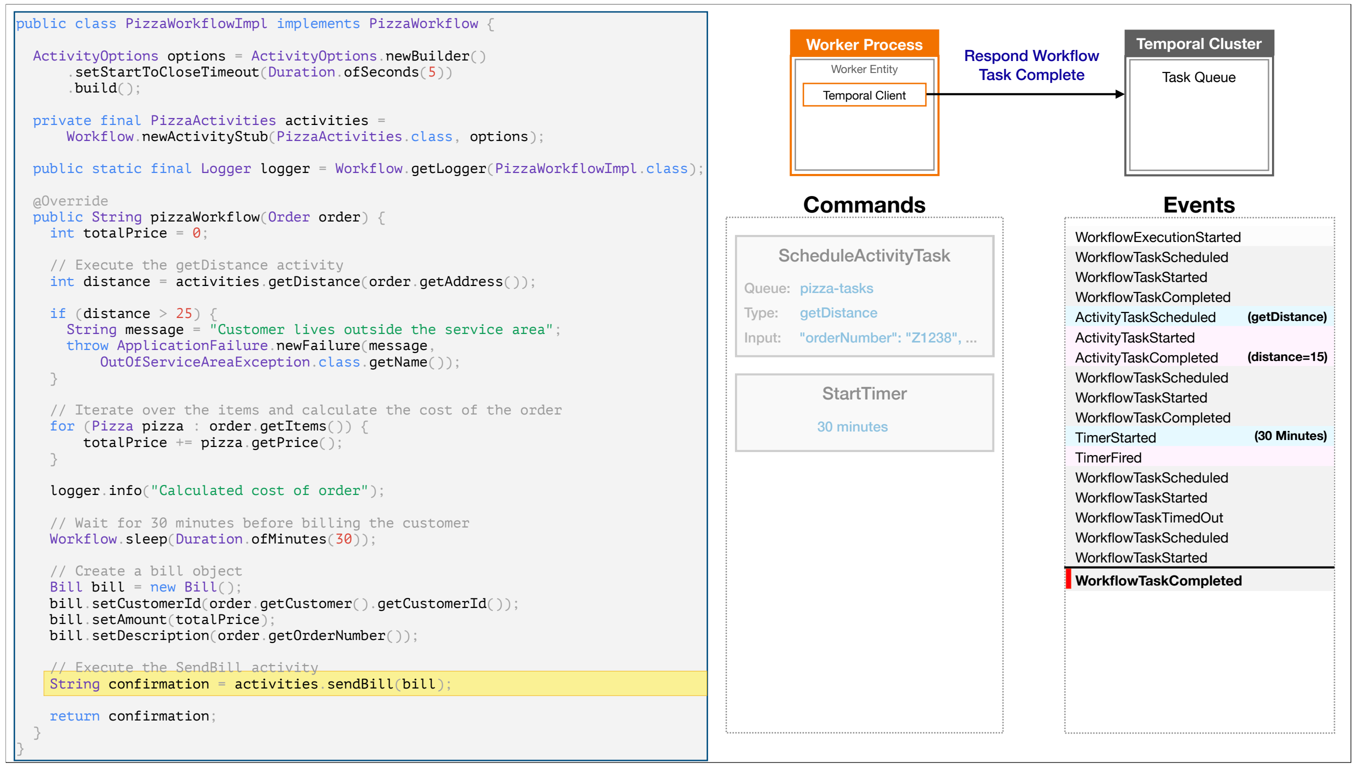
StartTimer

30 minutes

Events

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

The Worker has now reached a statement __beyond__ where the crash occurred, which is evident because the Event History doesn't contain any Events related to this Activity. Further execution of this Workflow continues on as is the crash had never happened.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask

Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

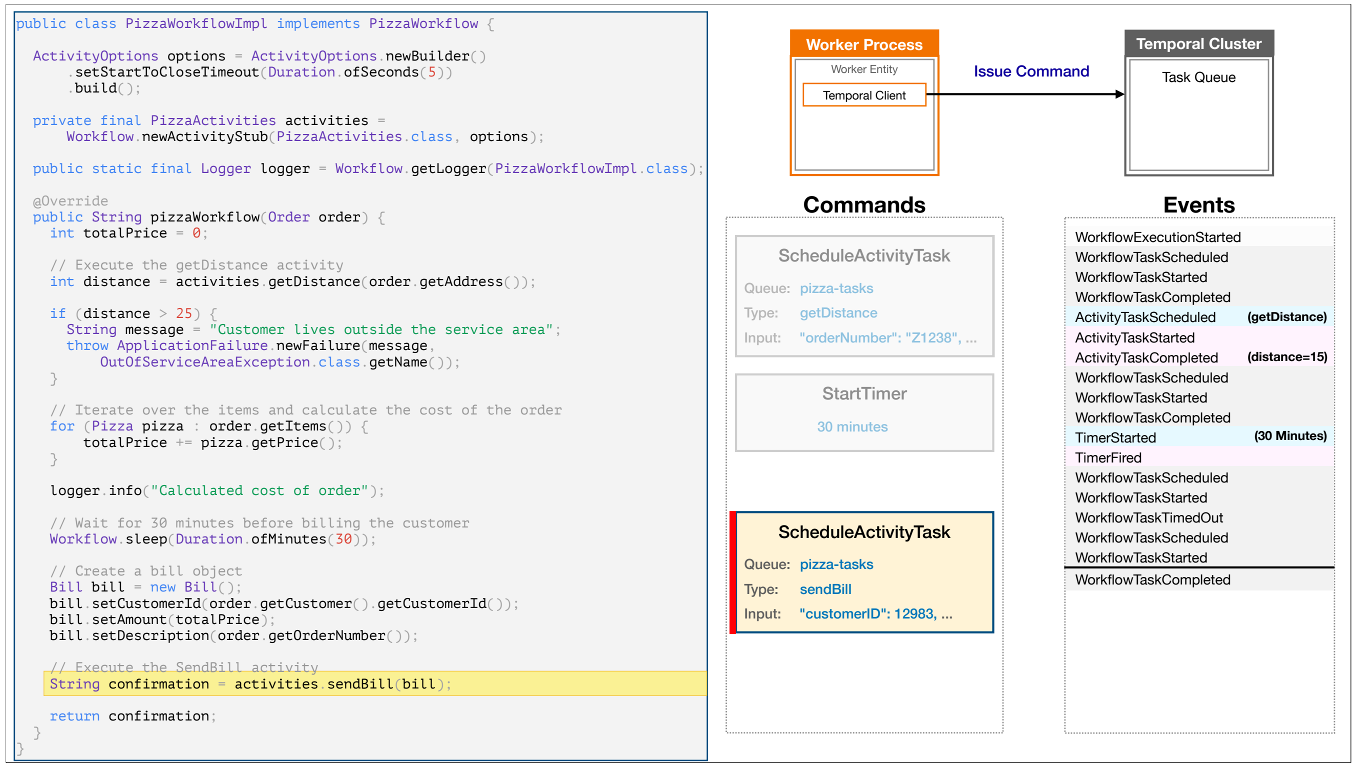
StartTimer

30 minutes

Events

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

Because it has encountered an Activity Method, the Worker completes the current Workflow Task,



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Worker Process

Worker Entity

Temporal Client

Issue Command

Temporal Cluster

Task Queue

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "orderNumber": "Z1238", ...

StartTimer
30 minutes

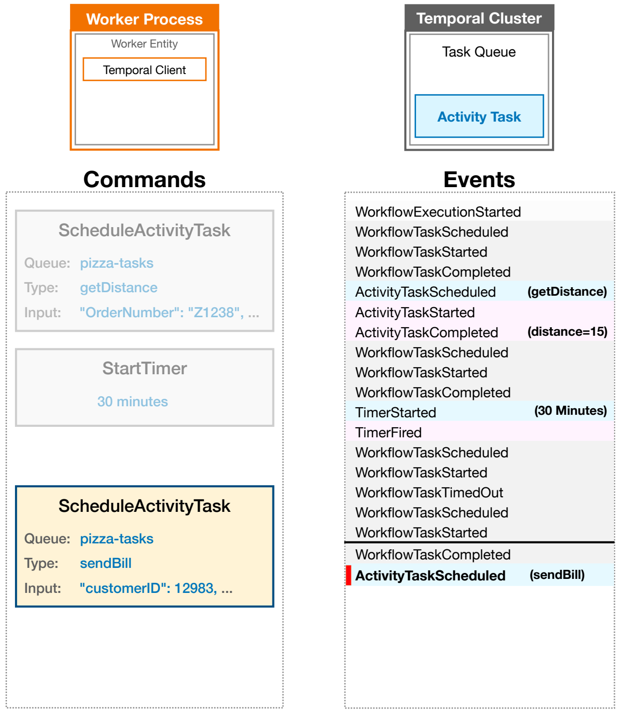
ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

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

and issues a Command to the cluster, requesting execution of this Activity.

```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```



Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

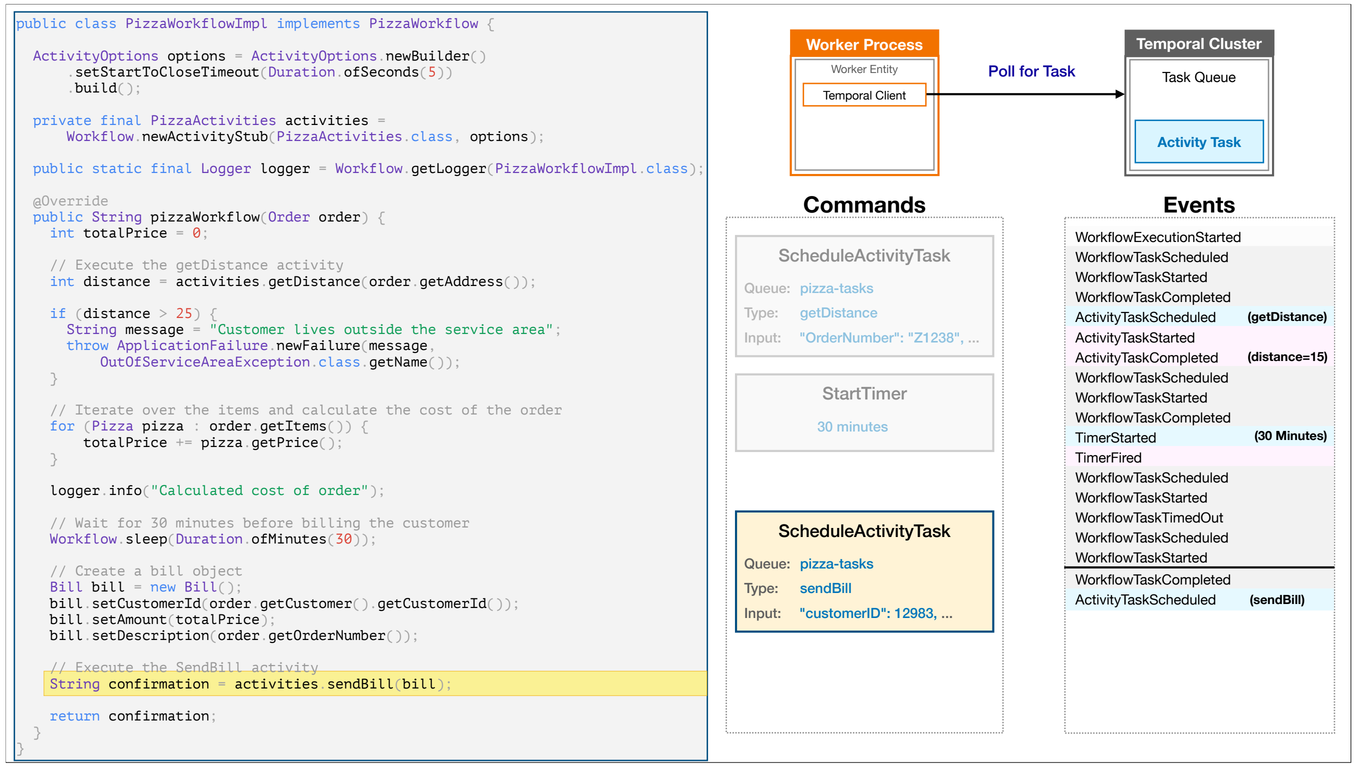
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

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

The cluster schedules an Activity Task.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

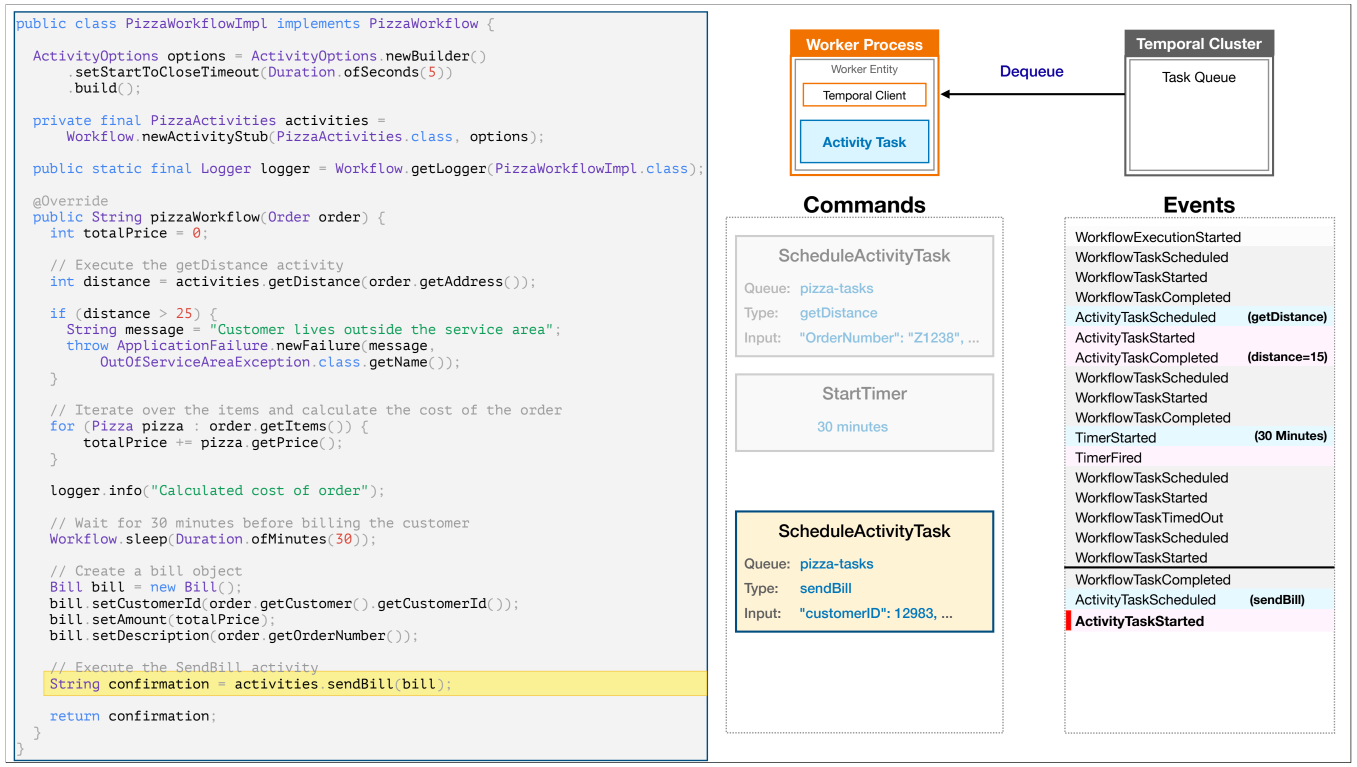
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

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

When the Worker polls,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

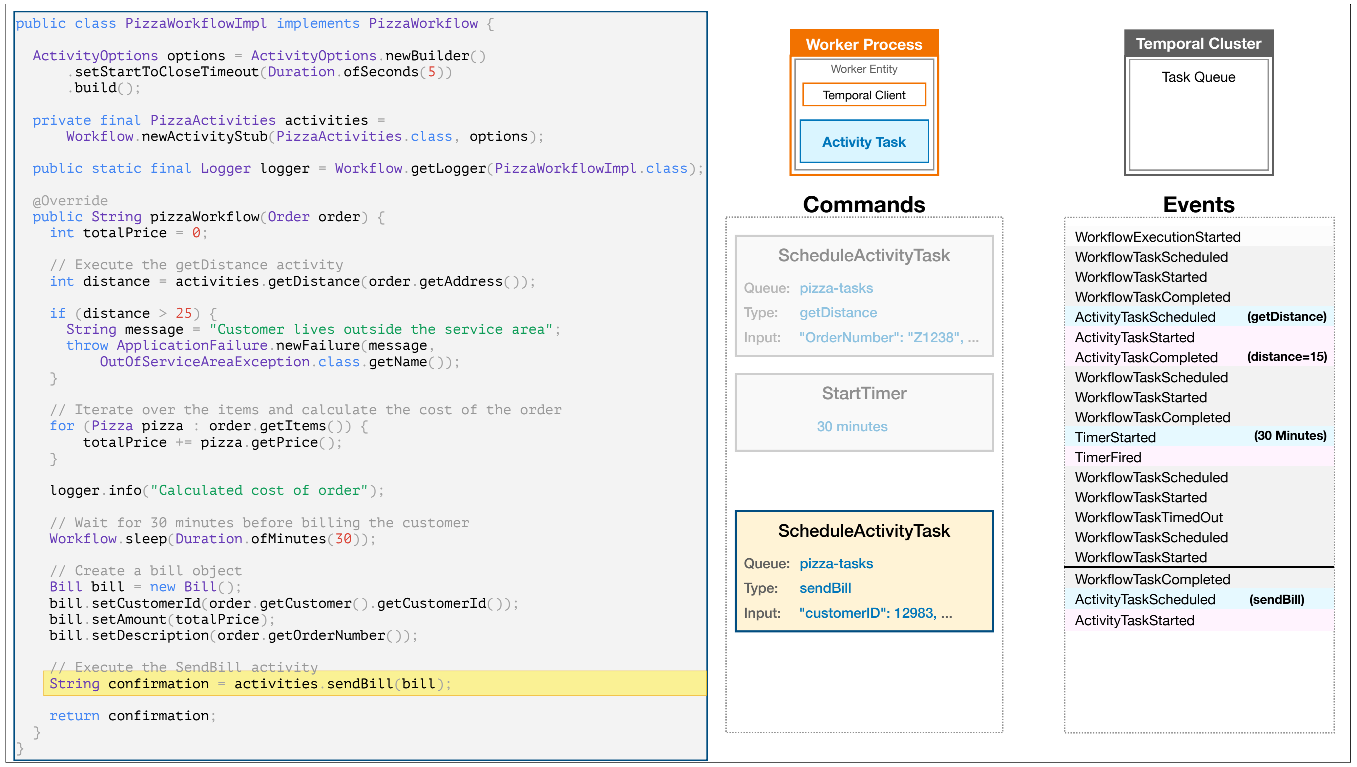
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

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

it accepts the Activity Task



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

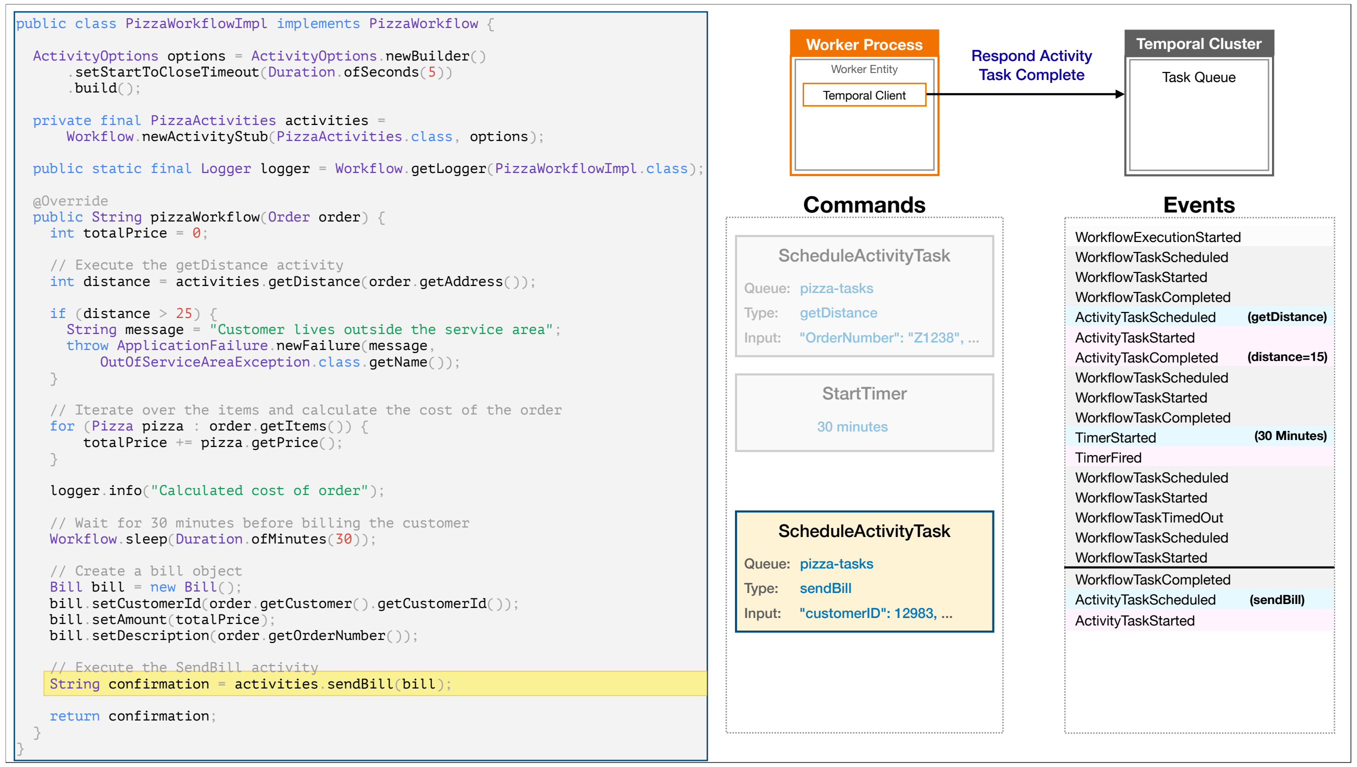
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

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

and executes the code for this Activity.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

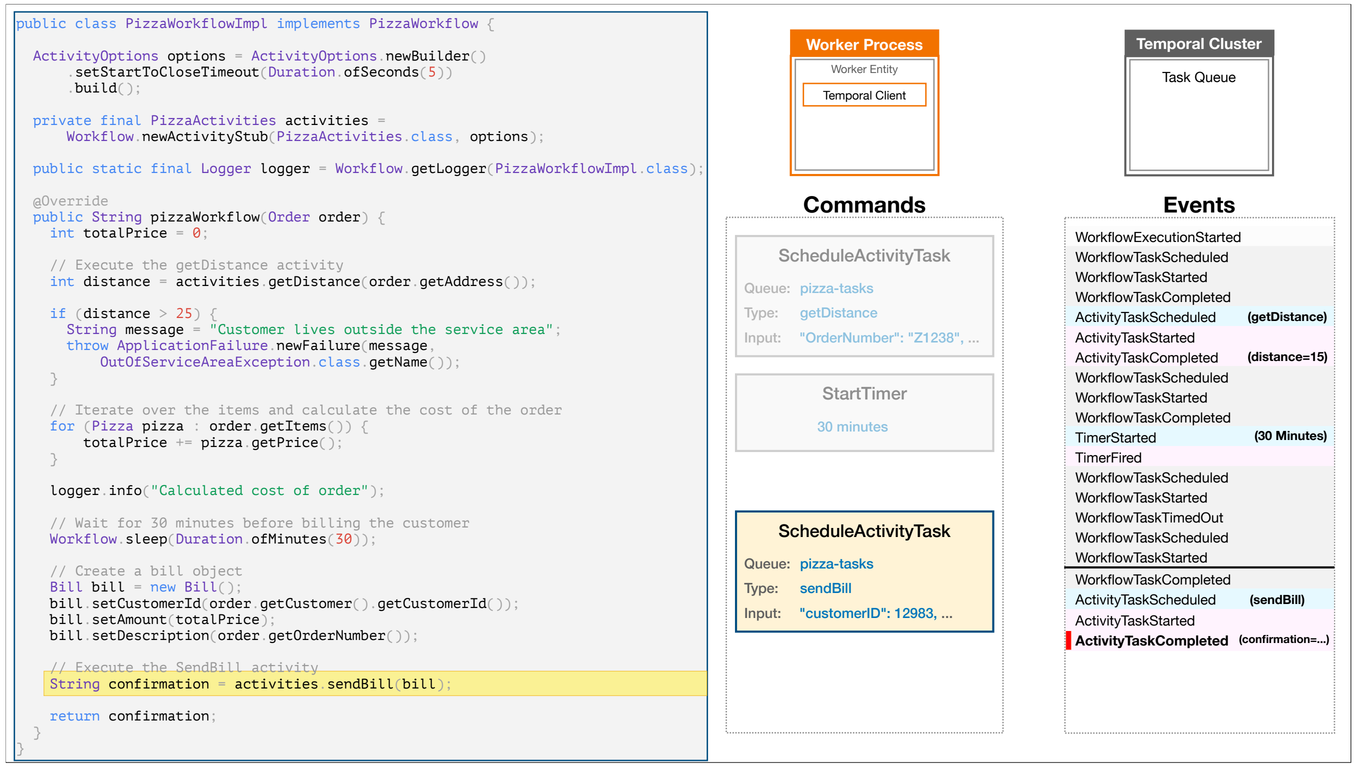
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

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

When the Activity function returns a result, the Worker notifies the cluster,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

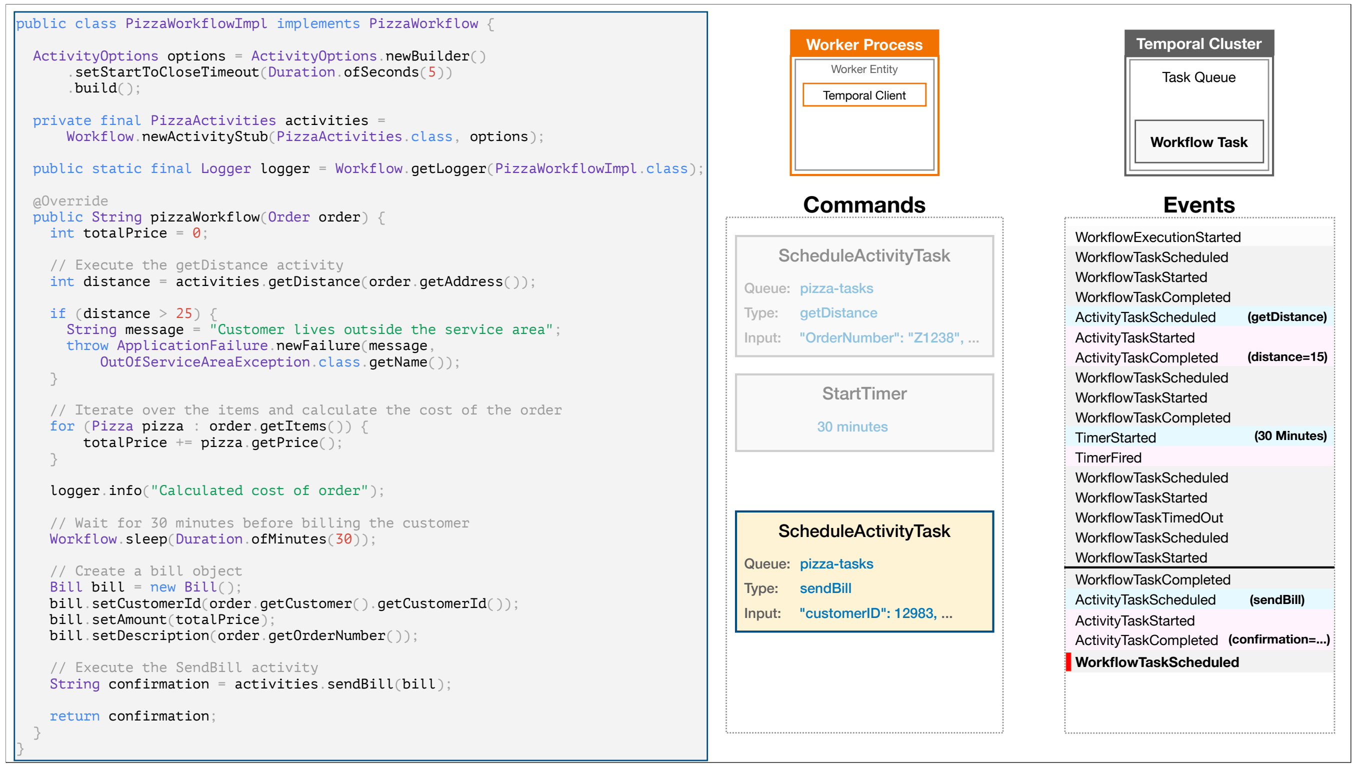
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted
ActivityTaskCompleted (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (sendBill)
ActivityTaskStarted
ActivityTaskCompleted (confirmation=...)

which logs an `ActivityTaskCompleted` Event.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

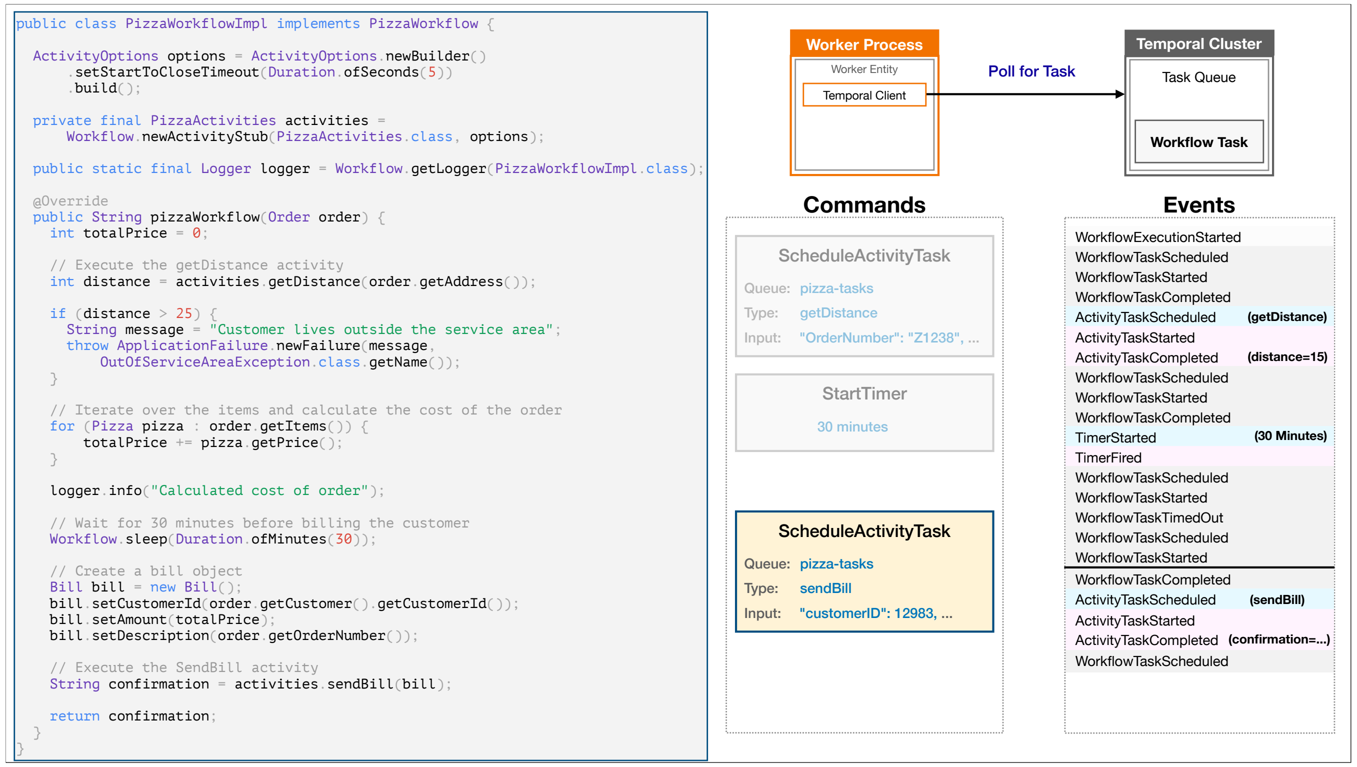
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted
ActivityTaskCompleted (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (sendBill)
ActivityTaskStarted
ActivityTaskCompleted (confirmation=...)
WorkflowTaskScheduled

But since the cluster hasn't yet received a Command that says the Workflow Execution has completed or failed, the cluster schedules another Workflow Task to continue progress of this execution.



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

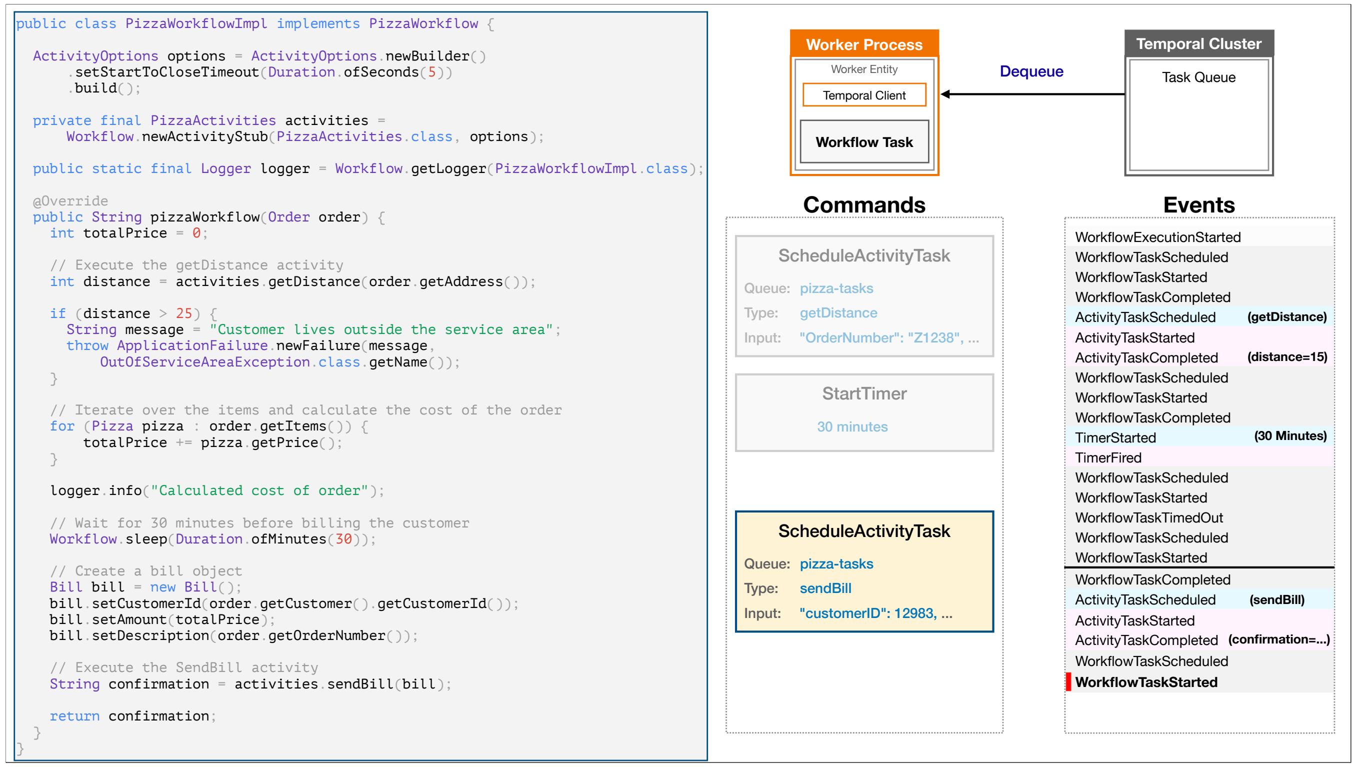
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted
ActivityTaskCompleted (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (sendBill)
ActivityTaskStarted
ActivityTaskCompleted (confirmation=...)
WorkflowTaskScheduled

The Worker polls the Task Queue,



```
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

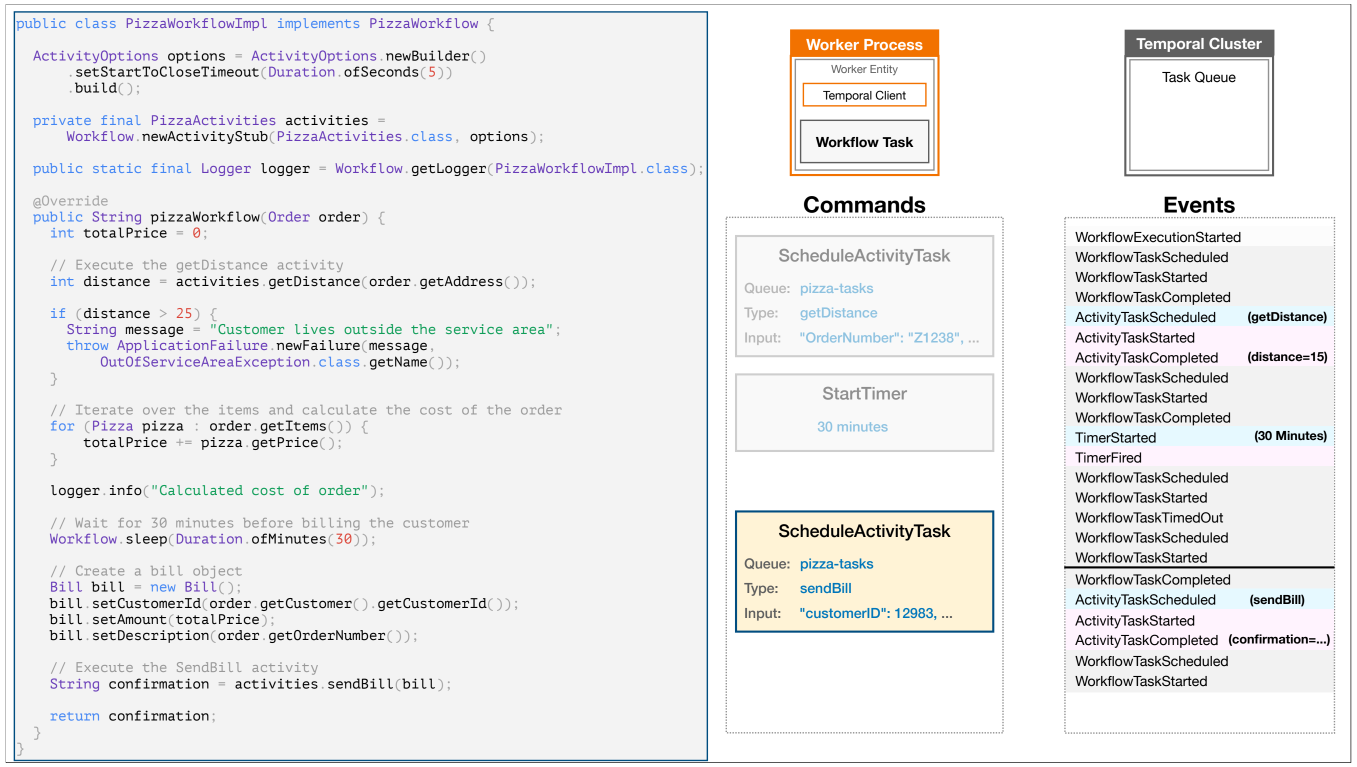
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted
ActivityTaskCompleted (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (sendBill)
ActivityTaskStarted
ActivityTaskCompleted (confirmation=...)
WorkflowTaskScheduled
WorkflowTaskStarted

accepts the Workflow Task,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

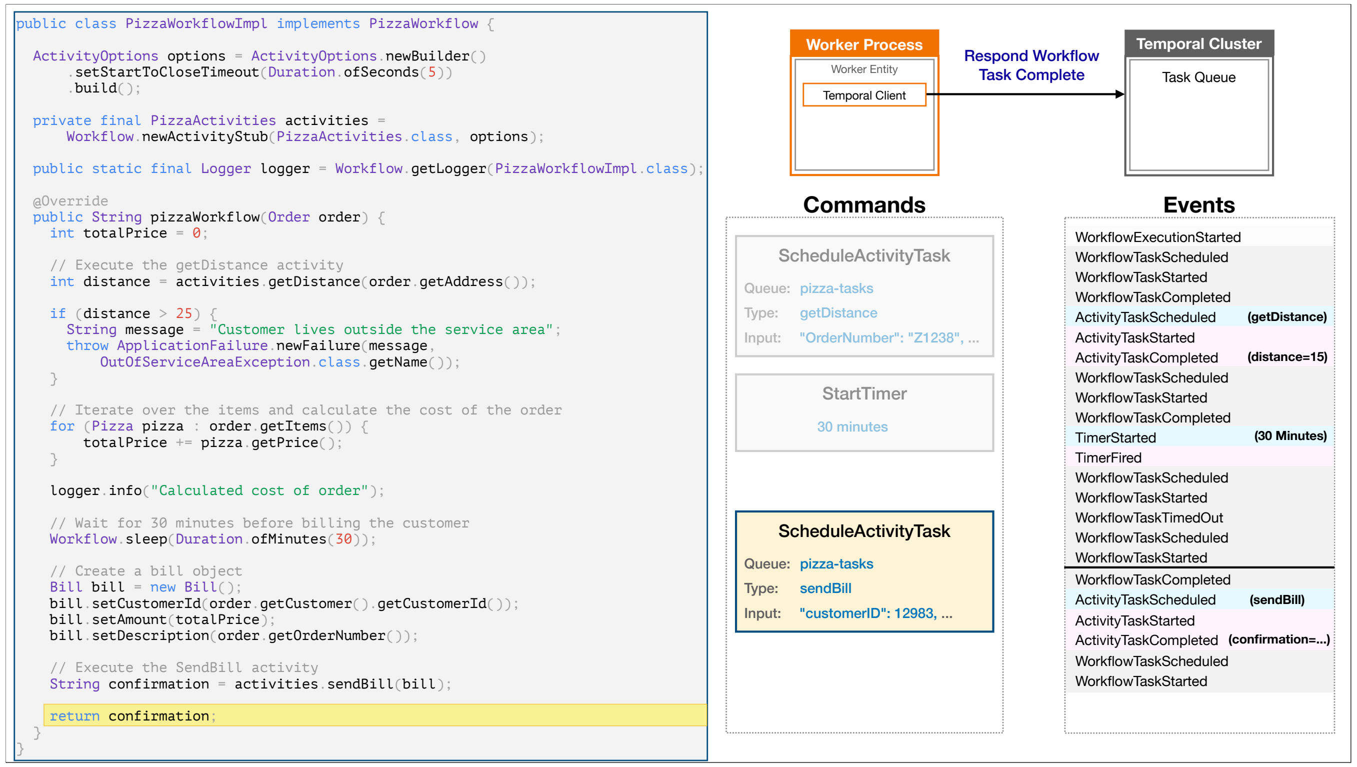
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled        (getDistance)
ActivityTaskStarted
ActivityTaskCompleted        (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted                 (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled        (sendBill)
ActivityTaskStarted
ActivityTaskCompleted        (confirmation=...)
WorkflowTaskScheduled
WorkflowTaskStarted
```

and resumes execution of the remaining Workflow code.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

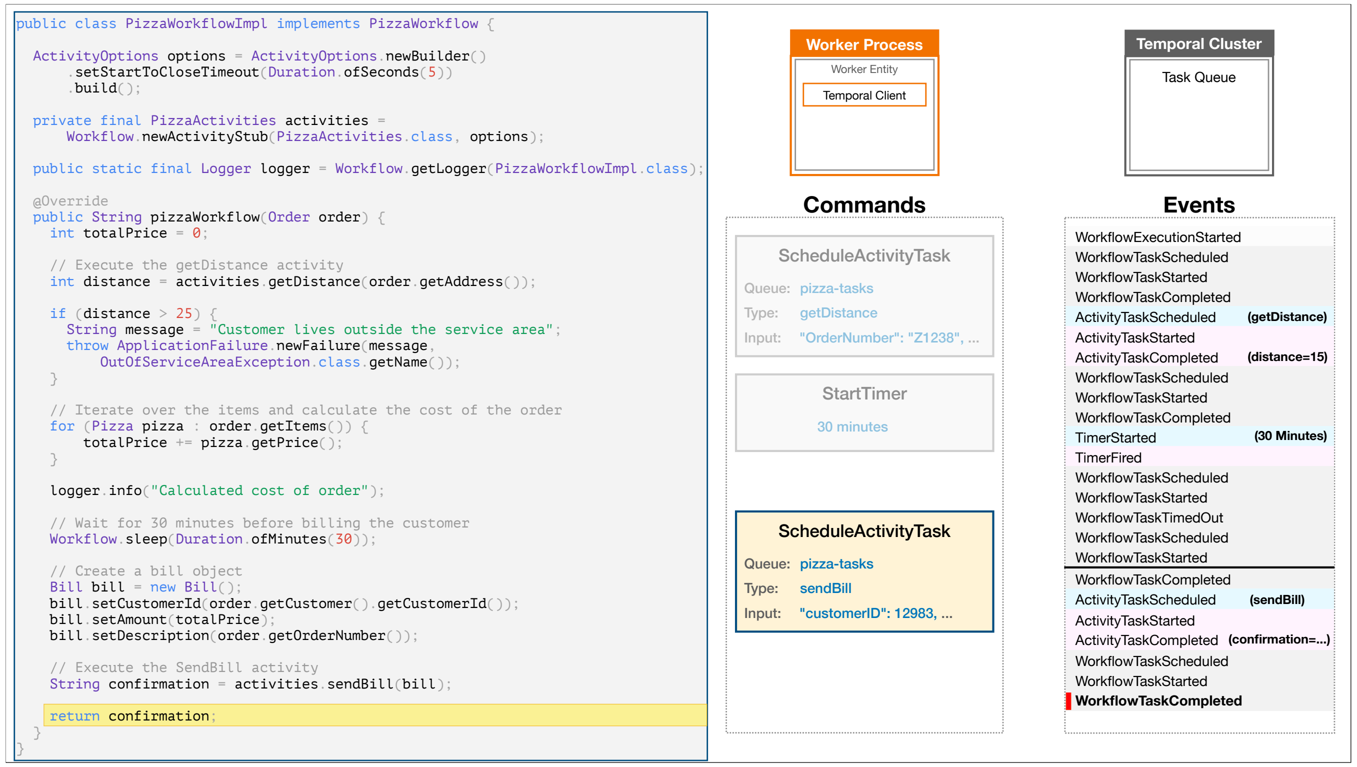
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted
ActivityTaskCompleted (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (sendBill)
ActivityTaskStarted
ActivityTaskCompleted (confirmation=...)
WorkflowTaskScheduled
WorkflowTaskStarted

When it reaches the end,



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

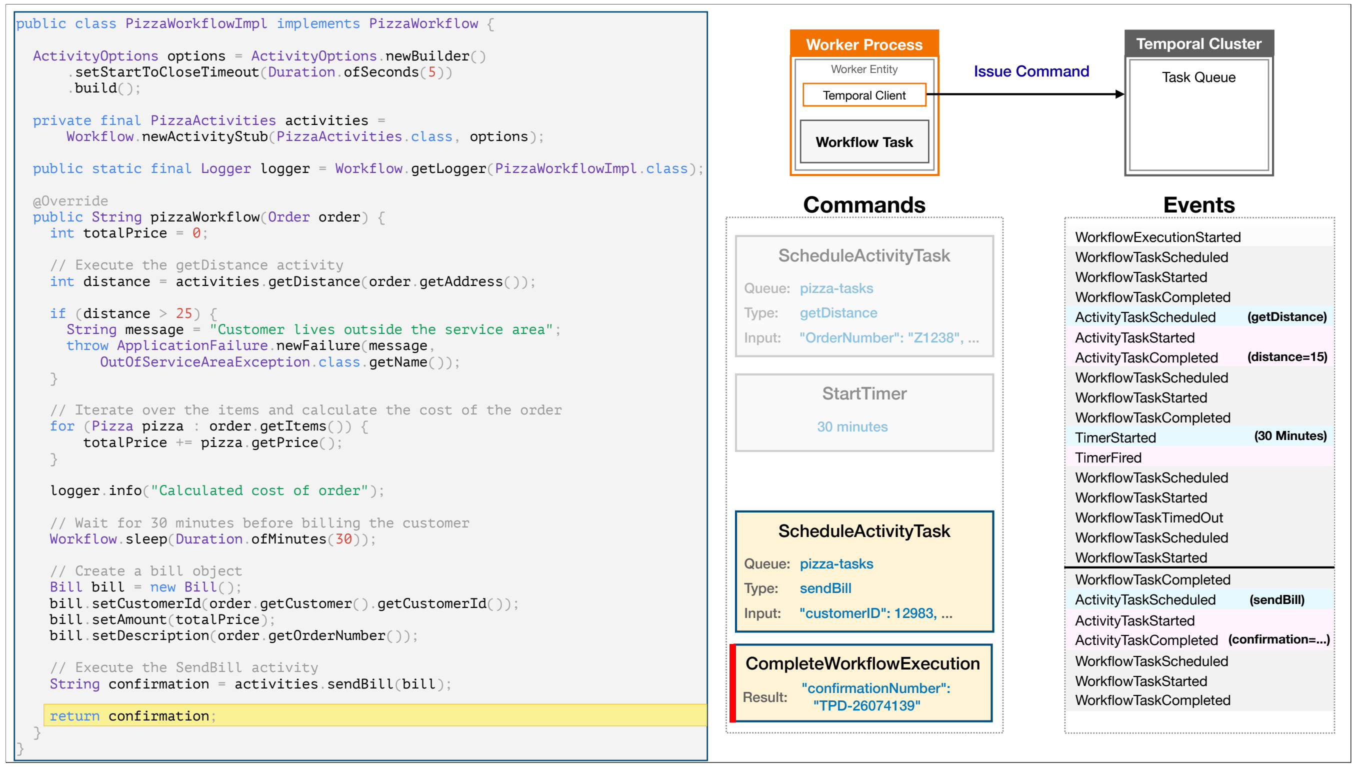
StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

Events

```
WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled        (getDistance)
ActivityTaskStarted
ActivityTaskCompleted        (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted                 (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled        (sendBill)
ActivityTaskStarted
ActivityTaskCompleted   (confirmation=...)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
```

it notifies that the cluster that the current Workflow Task is complete, and the cluster logs an Event to reflect this.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

StartTimer
30 minutes

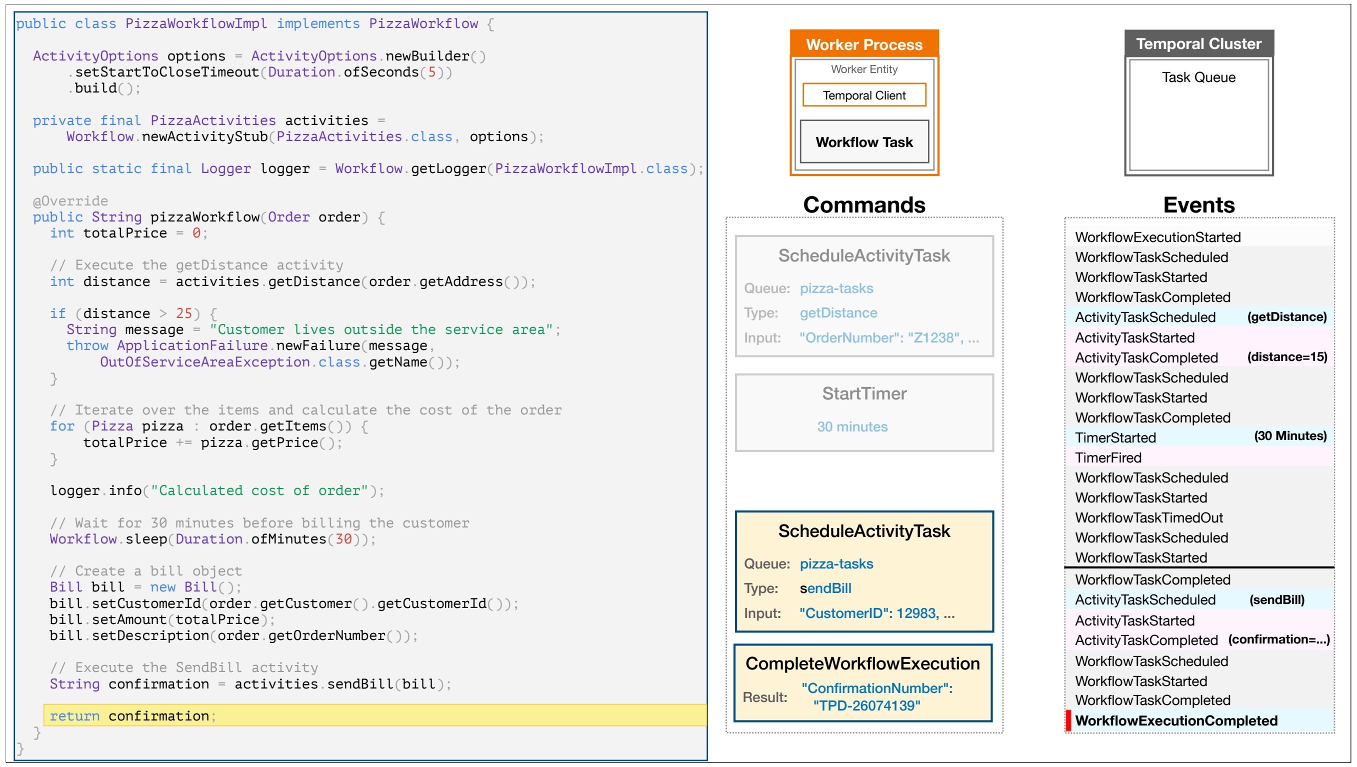
ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "customerID": 12983, ...

CompleteWorkflowExecution
Result: "confirmationNumber": "TPD-26074139"

Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted
ActivityTaskCompleted (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (sendBill)
ActivityTaskStarted
ActivityTaskCompleted (confirmation=...)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted

Since the Worker has now successfully completed the execution of the Workflow function, it issues a `CompleteWorkflowExecution` Command to the cluster, which contains the result returned by this function.



```java
public class PizzaWorkflowImpl implements PizzaWorkflow {

  ActivityOptions options = ActivityOptions.newBuilder()
      .setStartToCloseTimeout(Duration.ofSeconds(5))
      .build();

  private final PizzaActivities activities =
      Workflow.newActivityStub(PizzaActivities.class, options);

  public static final Logger logger = Workflow.getLogger(PizzaWorkflowImpl.class);

  @Override
  public String pizzaWorkflow(Order order) {
    int totalPrice = 0;

    // Execute the getDistance activity
    int distance = activities.getDistance(order.getAddress());

    if (distance > 25) {
      String message = "Customer lives outside the service area";
      throw ApplicationFailure.newFailure(message,
          OutOfServiceAreaException.class.getName());
    }

    // Iterate over the items and calculate the cost of the order
    for (Pizza pizza : order.getItems()) {
        totalPrice += pizza.getPrice();
    }

    logger.info("Calculated cost of order");

    // Wait for 30 minutes before billing the customer
    Workflow.sleep(Duration.ofMinutes(30));

    // Create a bill object
    Bill bill = new Bill();
    bill.setCustomerId(order.getCustomer().getCustomerId());
    bill.setAmount(totalPrice);
    bill.setDescription(order.getOrderNumber());

    // Execute the SendBill activity
    String confirmation = activities.sendBill(bill);

    return confirmation;
  }
}
```

Commands

ScheduleActivityTask
Queue: pizza-tasks
Type: getDistance
Input: "OrderNumber": "Z1238", ...

StartTimer
30 minutes

ScheduleActivityTask
Queue: pizza-tasks
Type: sendBill
Input: "CustomerID": 12983, ...

CompleteWorkflowExecution
Result: "ConfirmationNumber": "TPD-26074139"

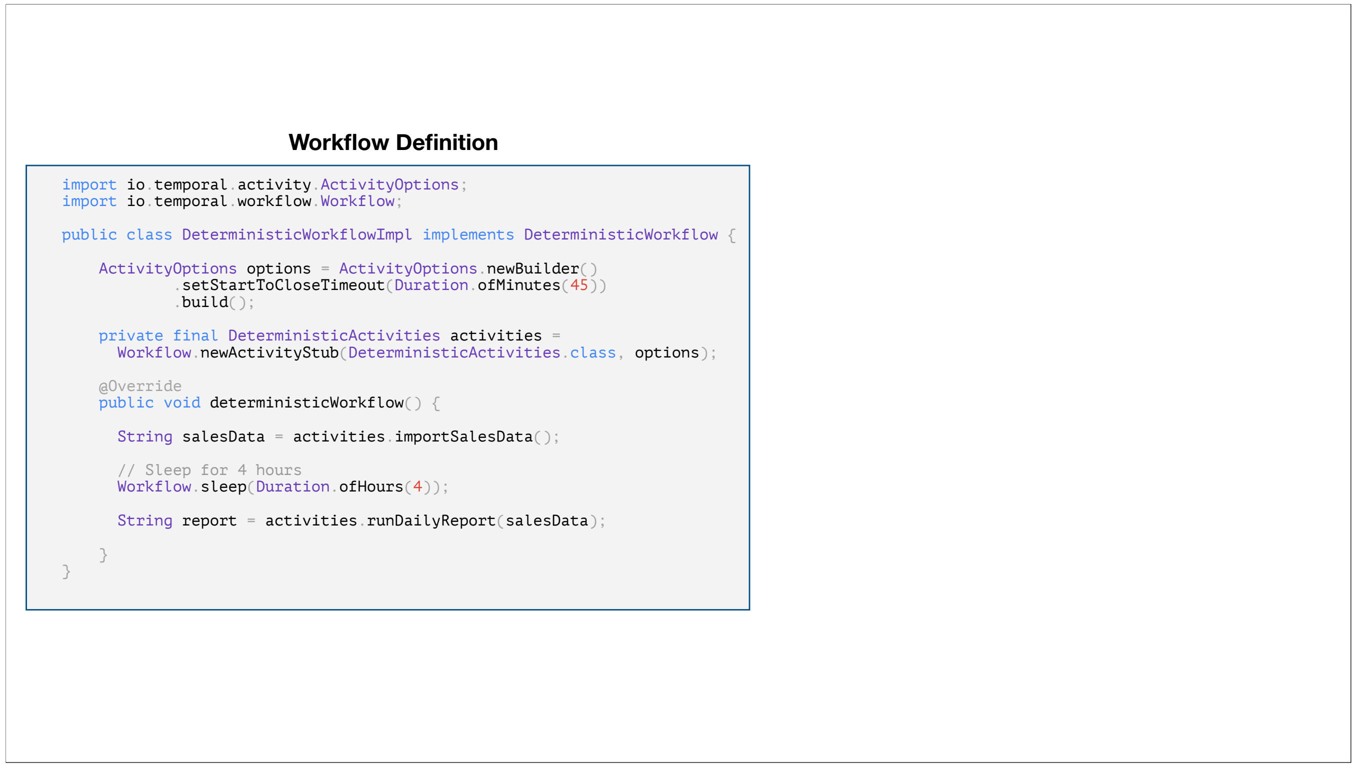
Events

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (getDistance)
ActivityTaskStarted
ActivityTaskCompleted (distance=15)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
TimerStarted (30 Minutes)
TimerFired
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskTimedOut
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
ActivityTaskScheduled (sendBill)
ActivityTaskStarted
ActivityTaskCompleted (confirmation=...)
WorkflowTaskScheduled
WorkflowTaskStarted
WorkflowTaskCompleted
WorkflowExecutionCompleted

The cluster then logs `WorkflowExecutionCompleted` as the final Event in the history. The Workflow Execution has now closed.

Why Temporal Requires Determinism for Workflows

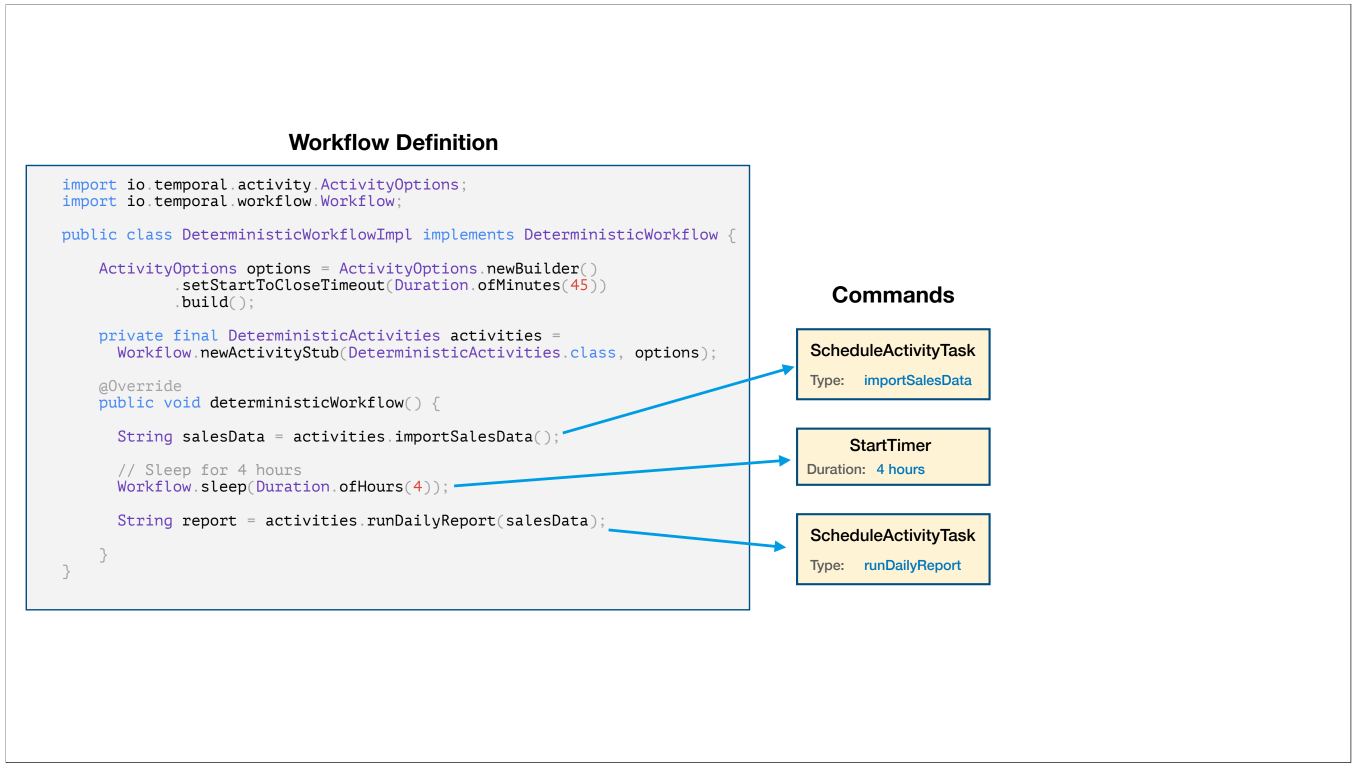
In Temporal 101, we mentioned that Workflow code must be deterministic. I'm going to explain not only what that means, but also why it's important, but first I want to reiterate a few important details about Workflow Execution to provide some context for this explanation.



Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      // Sleep for 4 hours
      Workflow.sleep(Duration.ofHours(4));

      String report = activities.runDailyReport(salesData);

    }
}
```

Let’s take a look at this workflow definition.



Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      // Sleep for 4 hours
      Workflow.sleep(Duration.ofHours(4));

      String report = activities.runDailyReport(salesData);

    }
}
```

Commands

ScheduleActivityTask
Type: importSalesData

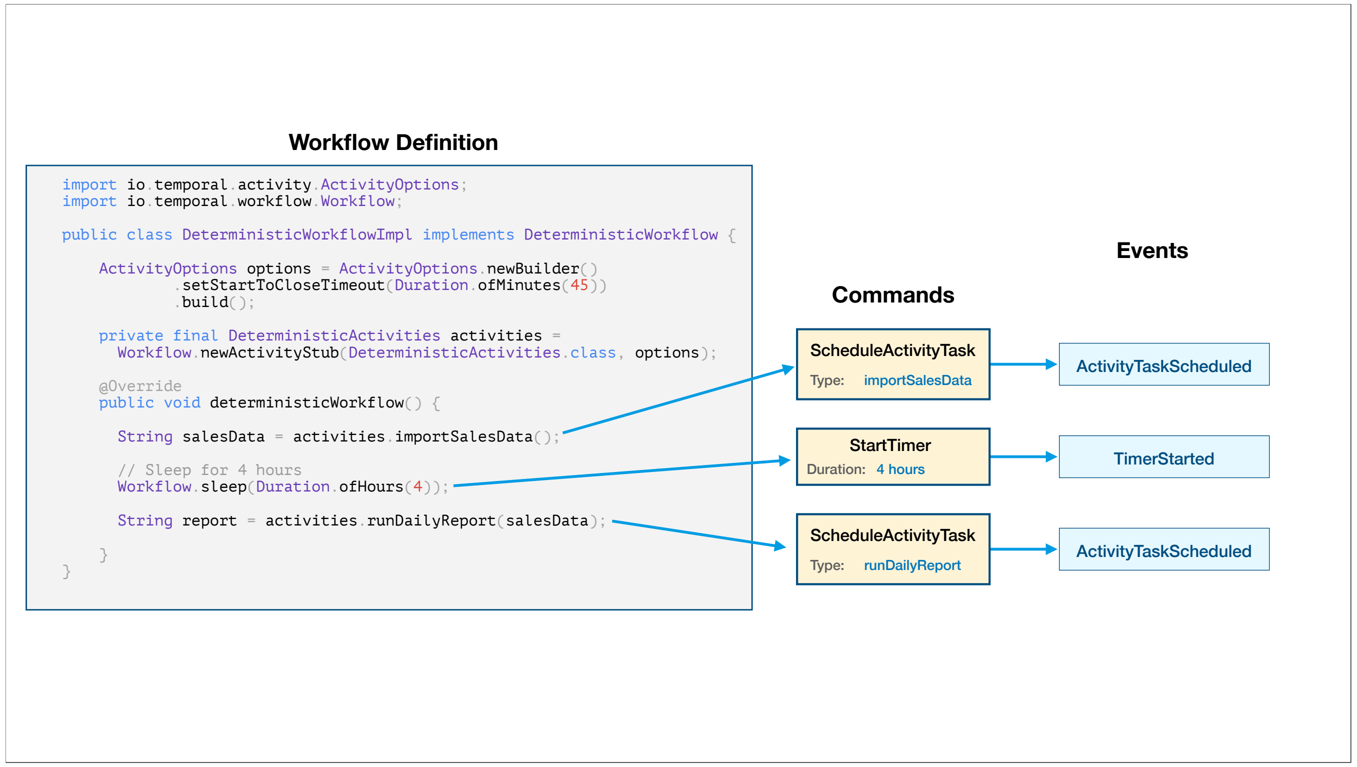
StartTimer
Duration: 4 hours

ScheduleActivityTask
Type: runDailyReport

As a Worker executes the code in your Workflow Definition, it creates Commands and issues them to a Temporal Cluster to request various operations, such as the execution of an Activity or the starting of a Timer.

The cluster maintains the Event History of each Workflow Execution.

Certain Events in the history are a direct result of a particular Command issued by a Worker.



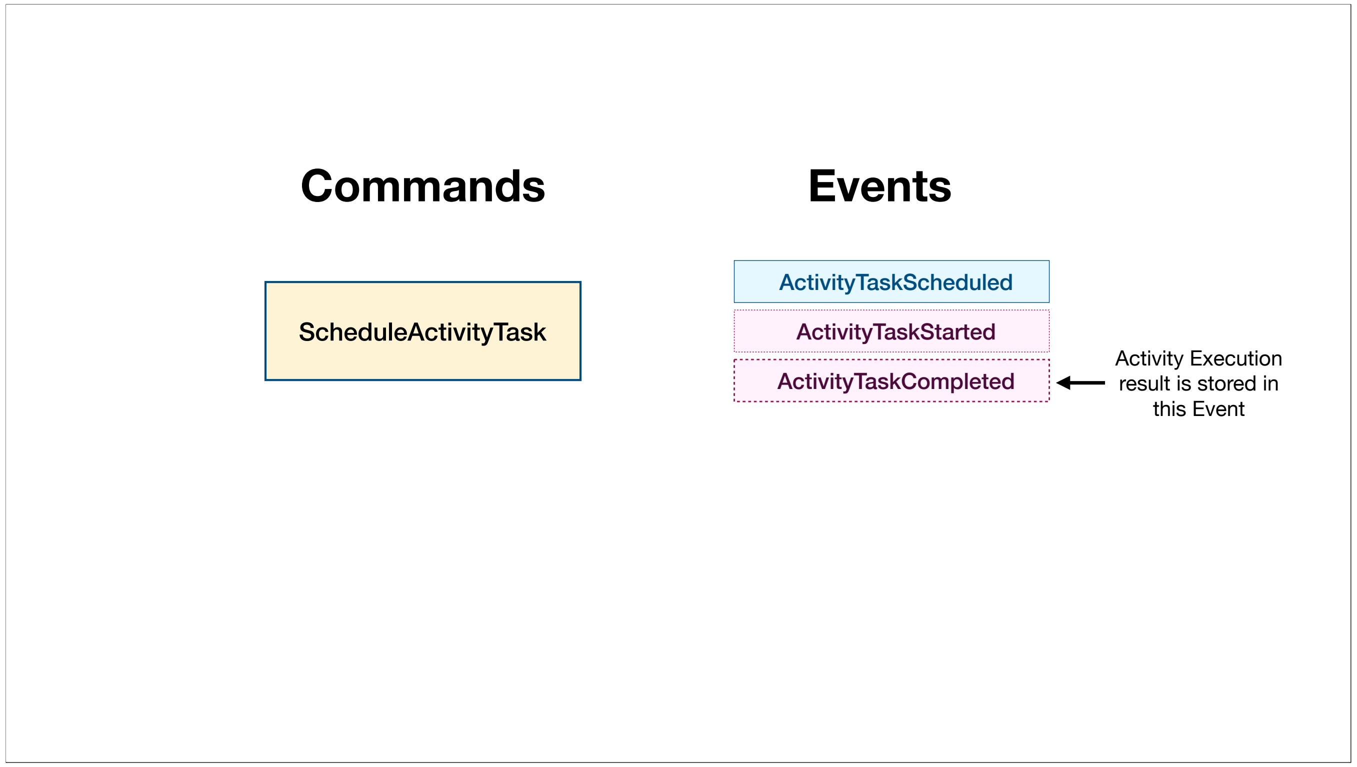
Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      // Sleep for 4 hours
      Workflow.sleep(Duration.ofHours(4));

      String report = activities.runDailyReport(salesData);

    }
}
```

For example, the `ScheduleActivityTask` Command results in an `ActivityTaskScheduled` Event

while the `StartTimer` Command results in a `TimerStarted` Event.

During Workflow Replay, the Worker uses this information to recover the state of the previous execution.



Commands

Events

ScheduleActivityTask

ActivityTaskScheduled

ActivityTaskStarted

ActivityTaskCompleted

Activity Execution result is stored in this Event

For example, if the `ScheduleActivityTask` Command has a corresponding

[advance]

`ActivityTaskScheduled` Event in the history, and this is followed by

[advance]

`ActivityTaskStarted` and

[advance]

`ActivityTaskCompleted`

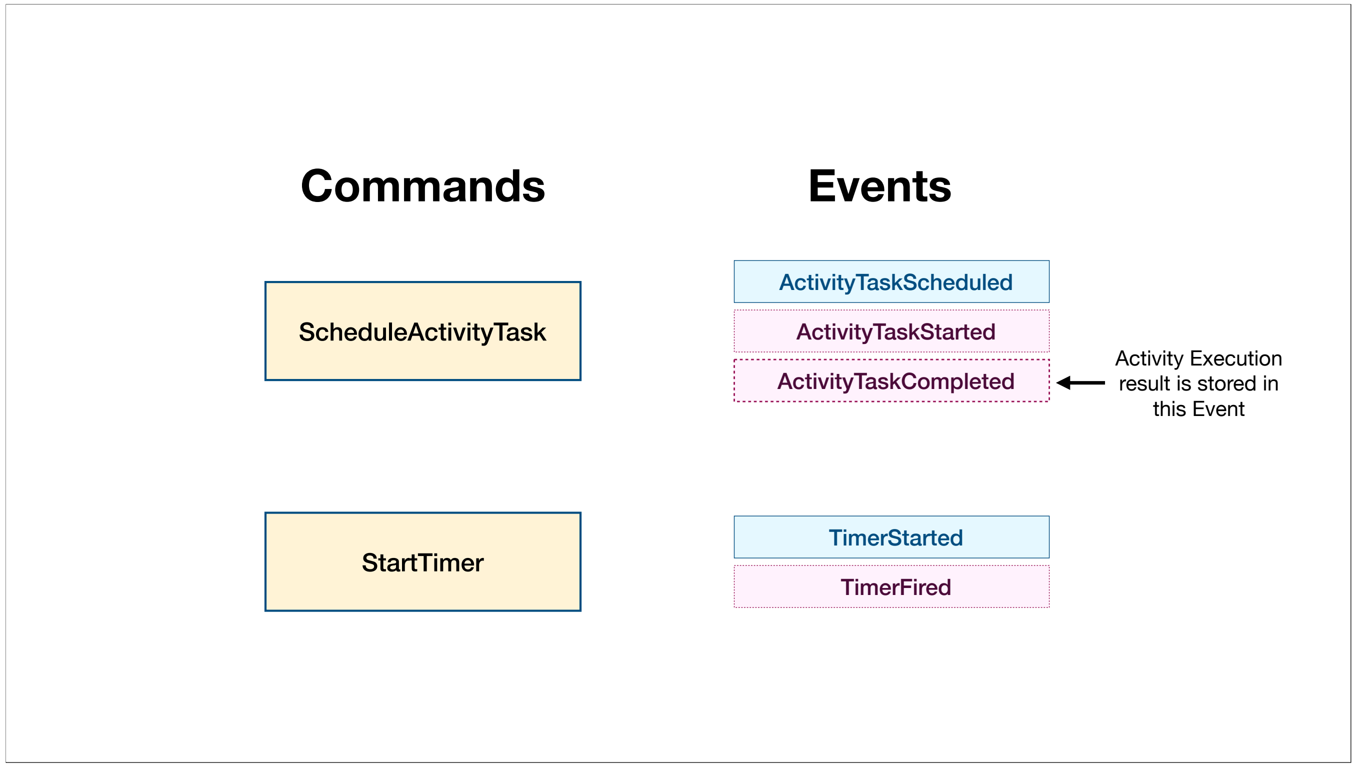
Events for that same
Activity Type, it's clear that this Activity already ran successfully. In this case, the Worker does not issue the Command to the cluster requesting a new execution of the Activity.

[advance]

Instead, it assigns the
result of the previous Activity Execution, which is stored in the Event
History.

Temporal requires that the code in your Workflow Definition behaves deterministically when executed.

In Temporal 101, we explained this by saying that it must produce the same output each time, given the same input. This explanation was sufficient for that point in your journey to learn Temporal, but you now know enough about Workflow Execution to understand the precise definition.



Commands

Events

ScheduleActivityTask

ActivityTaskScheduled

ActivityTaskStarted

ActivityTaskCompleted

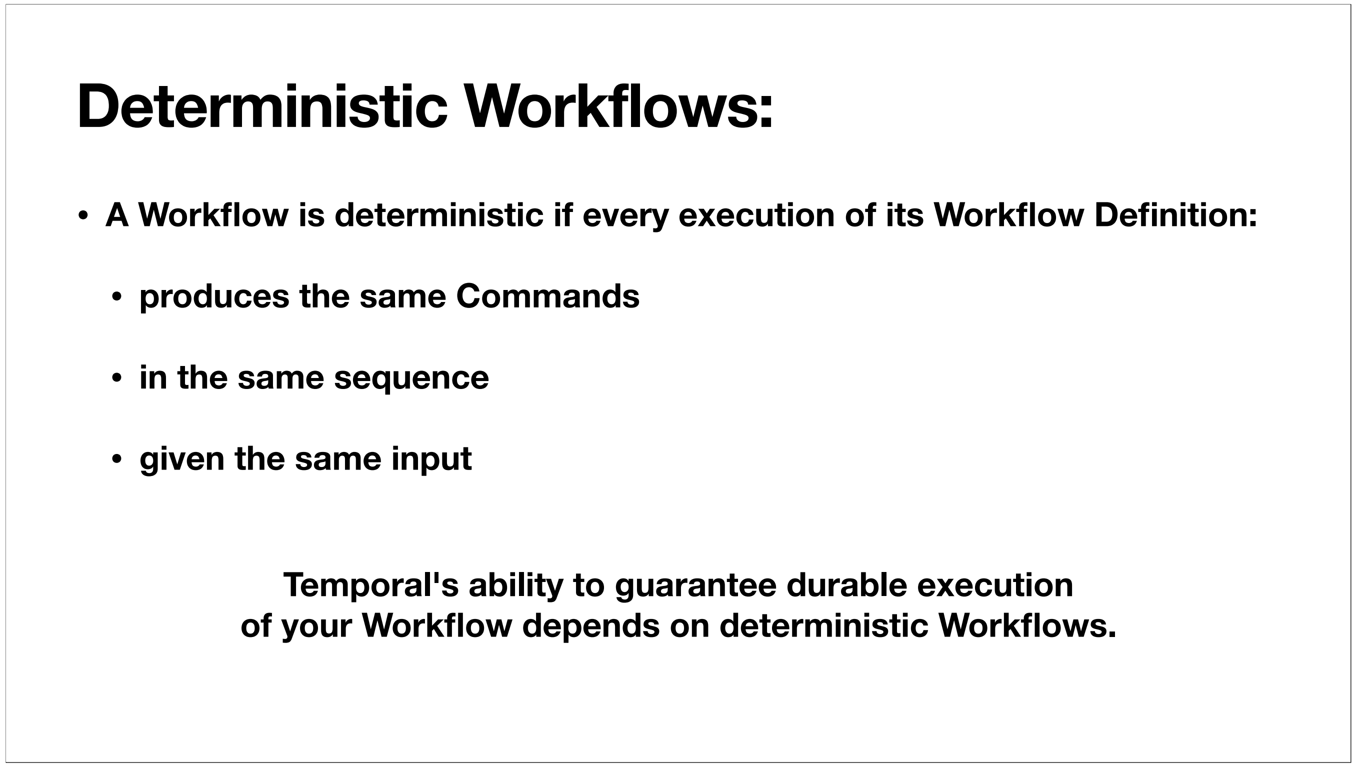
Activity Execution result is stored in this Event

StartTimer

TimerStarted

TimerFired

The same thing is true for timers.



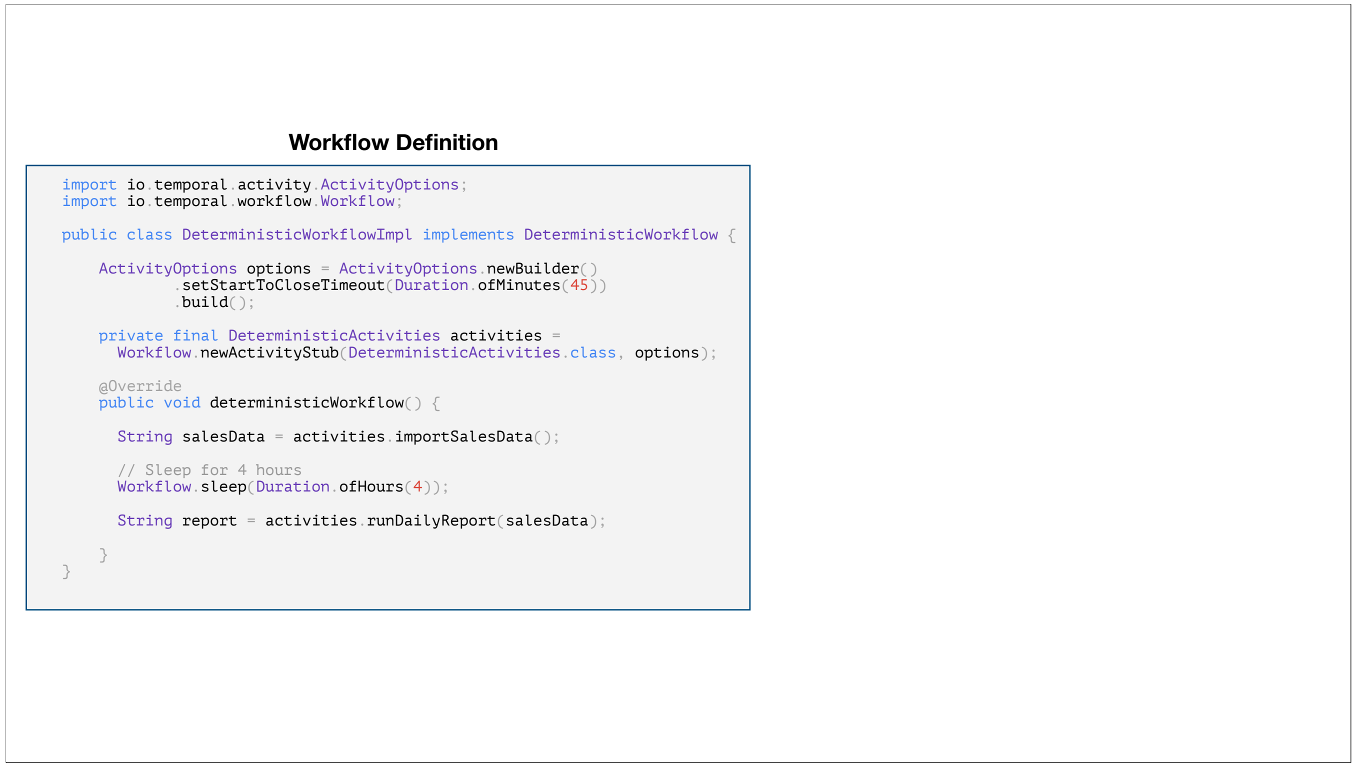
Deterministic Workflows:

- 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.**

A Workflow is deterministic if every execution of its Workflow Definition:
produces the same Commands
in the same sequence
given the same input

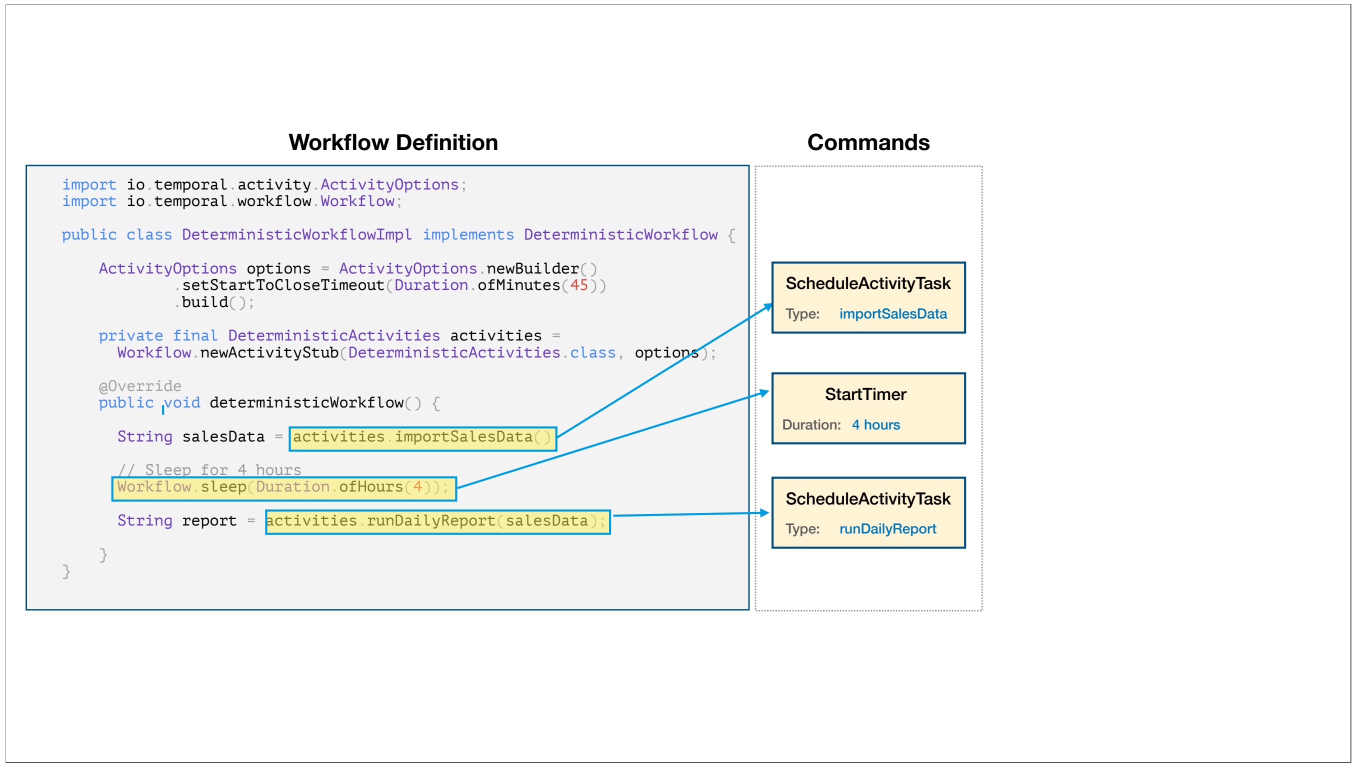
Temporal's ability to guarantee durable execution of your Workflow depends on deterministic Workflows.



Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      // Sleep for 4 hours
      Workflow.sleep(Duration.ofHours(4));

      String report = activities.runDailyReport(salesData);

    }
}
```

You've already learned that Temporal uses Workflow Replay to recover the state of a Workflow Execution. It does this if the Worker crashes, but it may also do this at other, less predictable times,

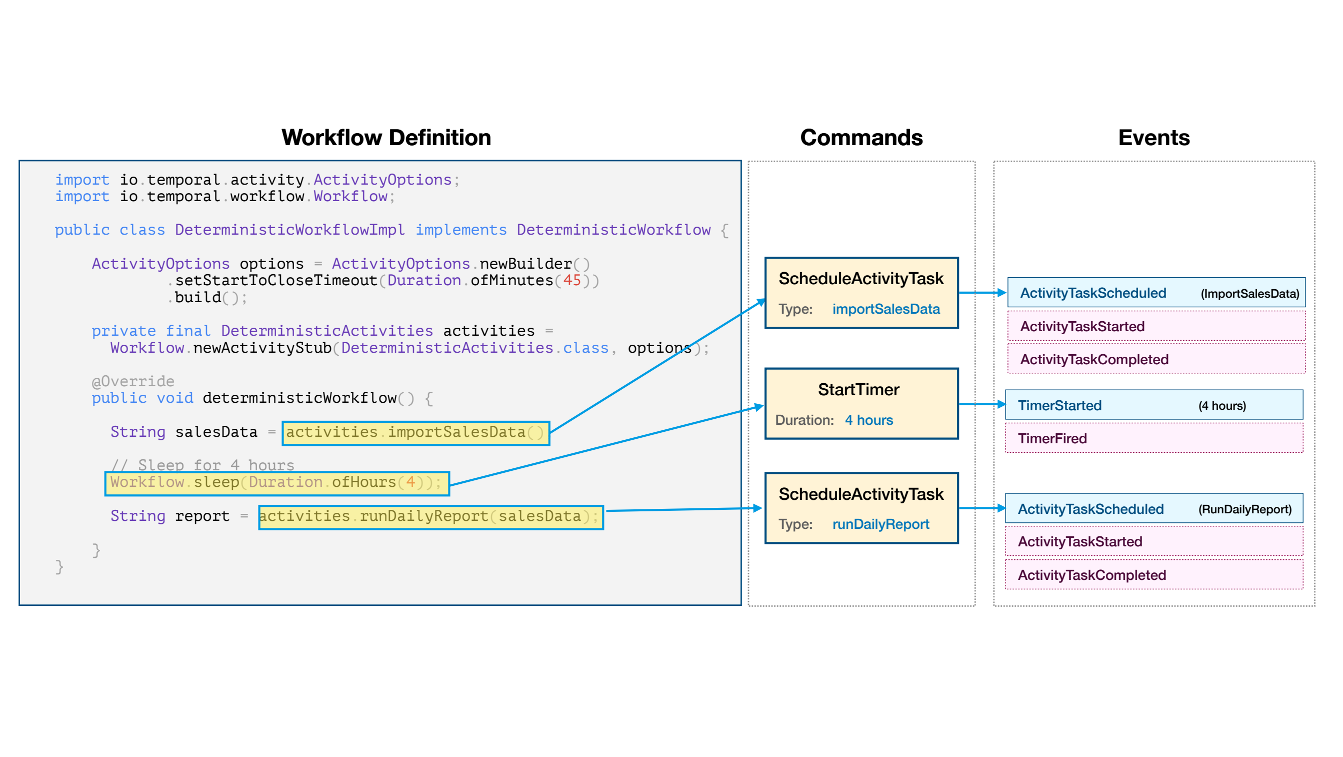


Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData()

      // Sleep for 4 hours
      Workflow.sleep(Duration.ofHours(4));

      String report = activities.runDailyReport(salesData);

    }
}
```

Commands

As you've seen, the Worker checks whether a Command created by replaying of the code…



Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData()

      // Sleep for 4 hours
      Workflow.sleep(Duration.ofHours(4));

      String report = activities.runDailyReport(salesData);

    }
}
```

Commands

ScheduleActivityTask
Type: importSalesData

StartTimer
Duration: 4 hours

ScheduleActivityTask
Type: runDailyReport

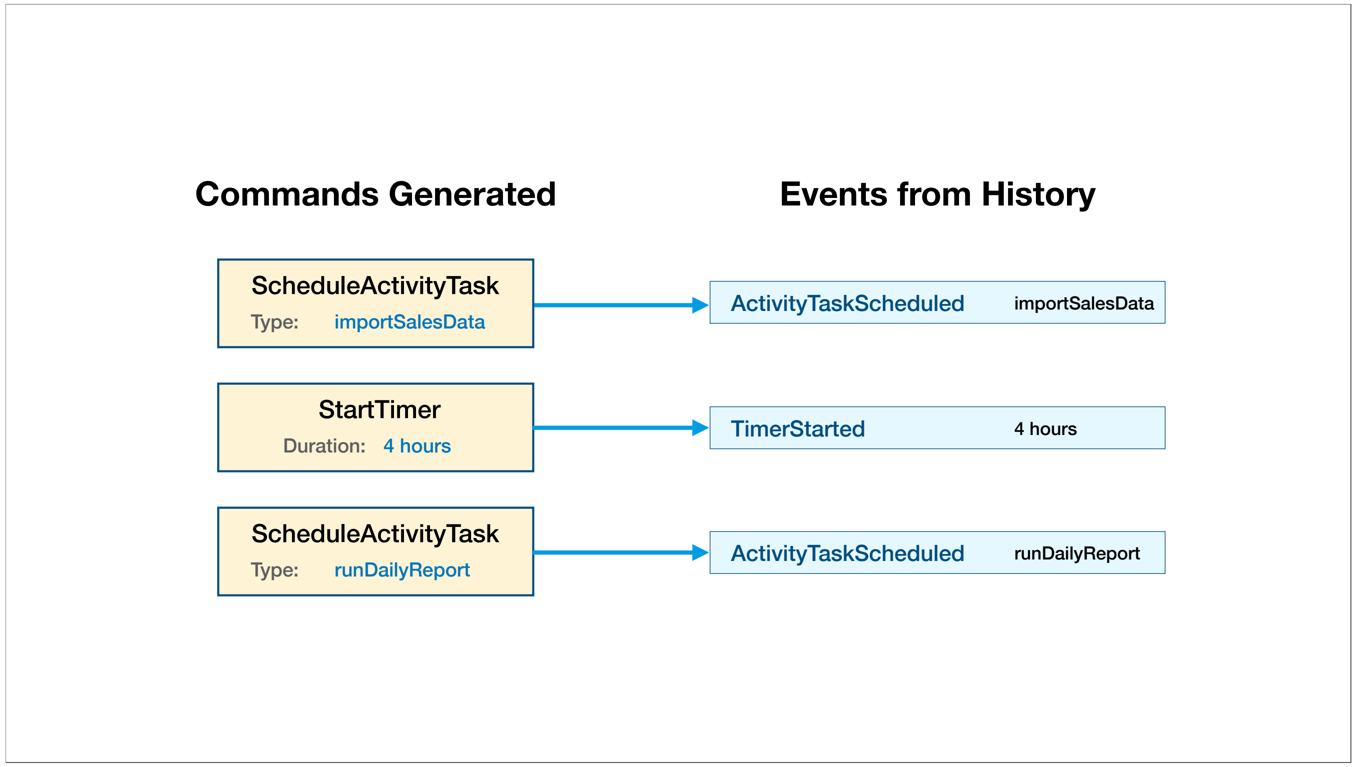
Events

ActivityTaskScheduled (ImportSalesData)
ActivityTaskStarted
ActivityTaskCompleted

TimerStarted (4 hours)
TimerFired

ActivityTaskScheduled (RunDailyReport)
ActivityTaskStarted
ActivityTaskCompleted

has a corresponding Event in the history, which it uses to determine whether the previous execution reached this point in the code.



Commands Generated

Events from History

ScheduleActivityTask
Type: importSalesData

ActivityTaskScheduled importSalesData

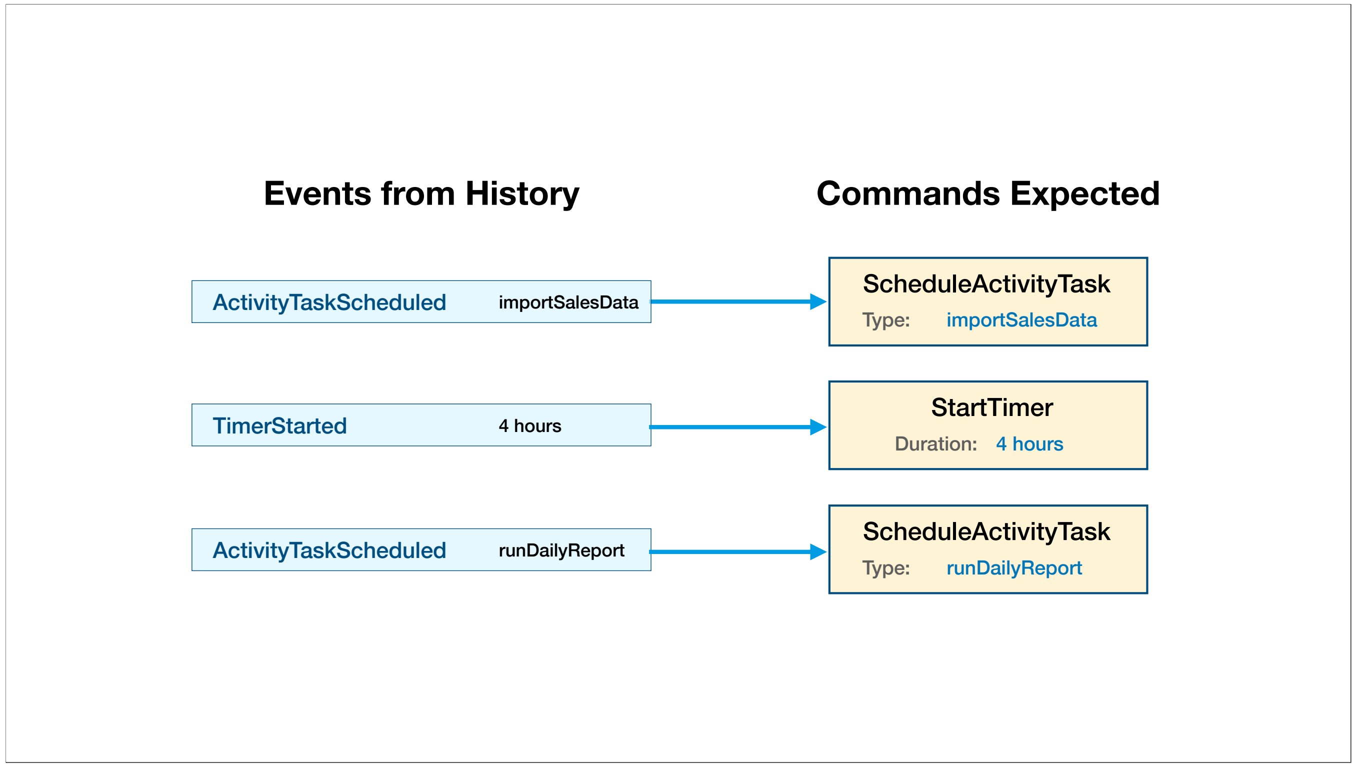
StartTimer
Duration: 4 hours

TimerStarted 4 hours

ScheduleActivityTask
Type: runDailyReport

ActivityTaskScheduled runDailyReport

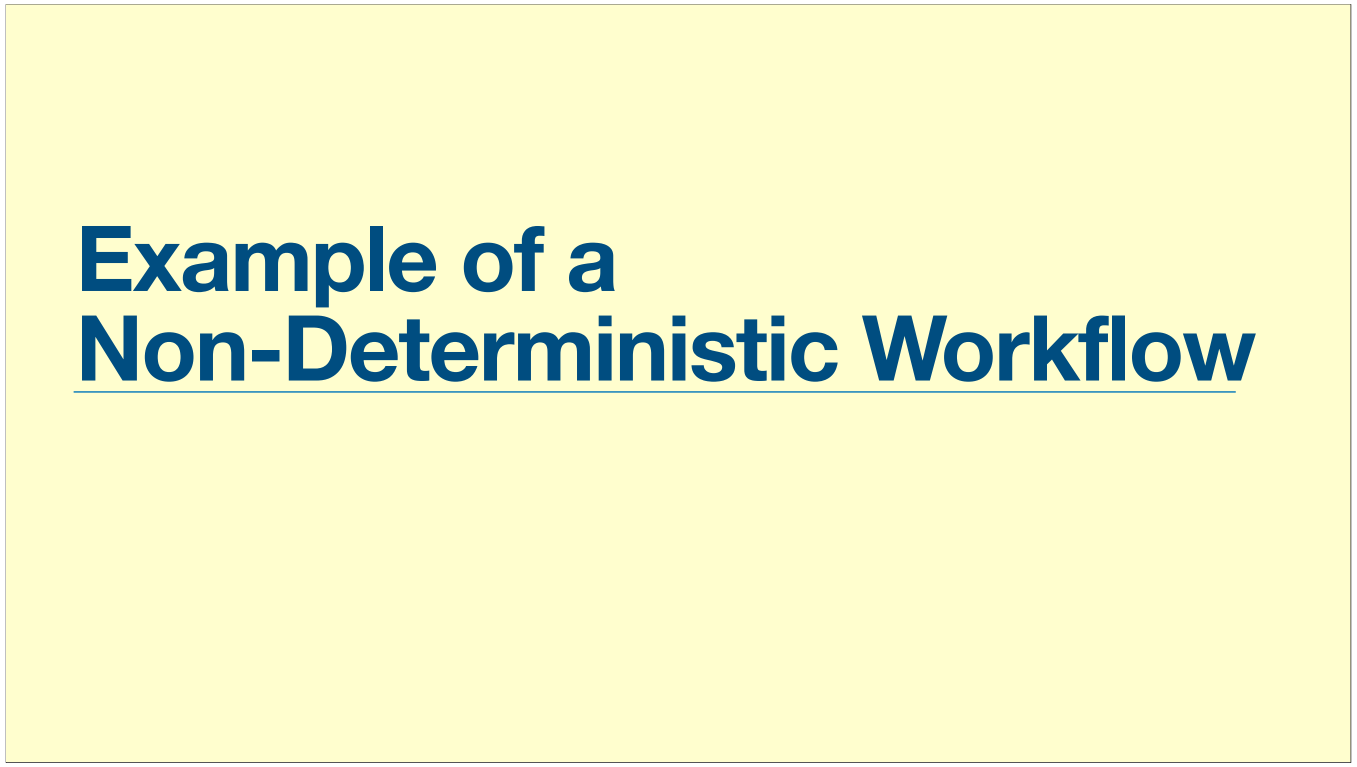
If a specific Command results in a specific type of Event, then it follows that the reverse is also true. In other words, given one of these Events, you can determine which Command led to that Event.



The Worker uses this logic to evaluate the Event History during replay and validate that it can reliably recover the Workflow Execution.

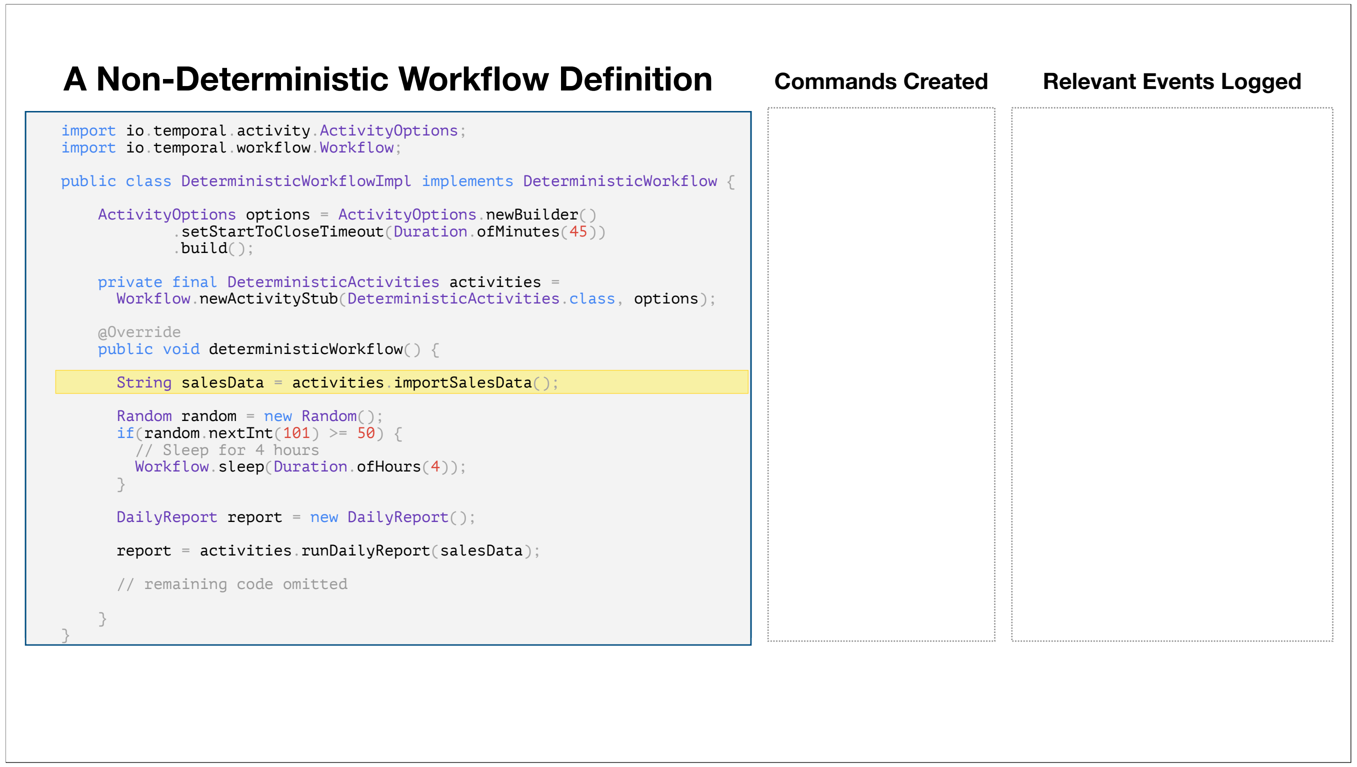
Events that are the direct result of Commands, shown here on the left, are used to create a list of Commands expected during replay.

A mismatch between the Commands that the Worker expected, based on the Event History, and those created, based on actually executing the code, results in a non-deterministic error. This error means that the Worker cannot accurately restore the state of the Workflow Execution.



Example of a Non-Deterministic Workflow

To better understand the determinism requirement, it's helpful to look at a Workflow Definition that violates it. In this case, we'll use one that uses a random number generator.



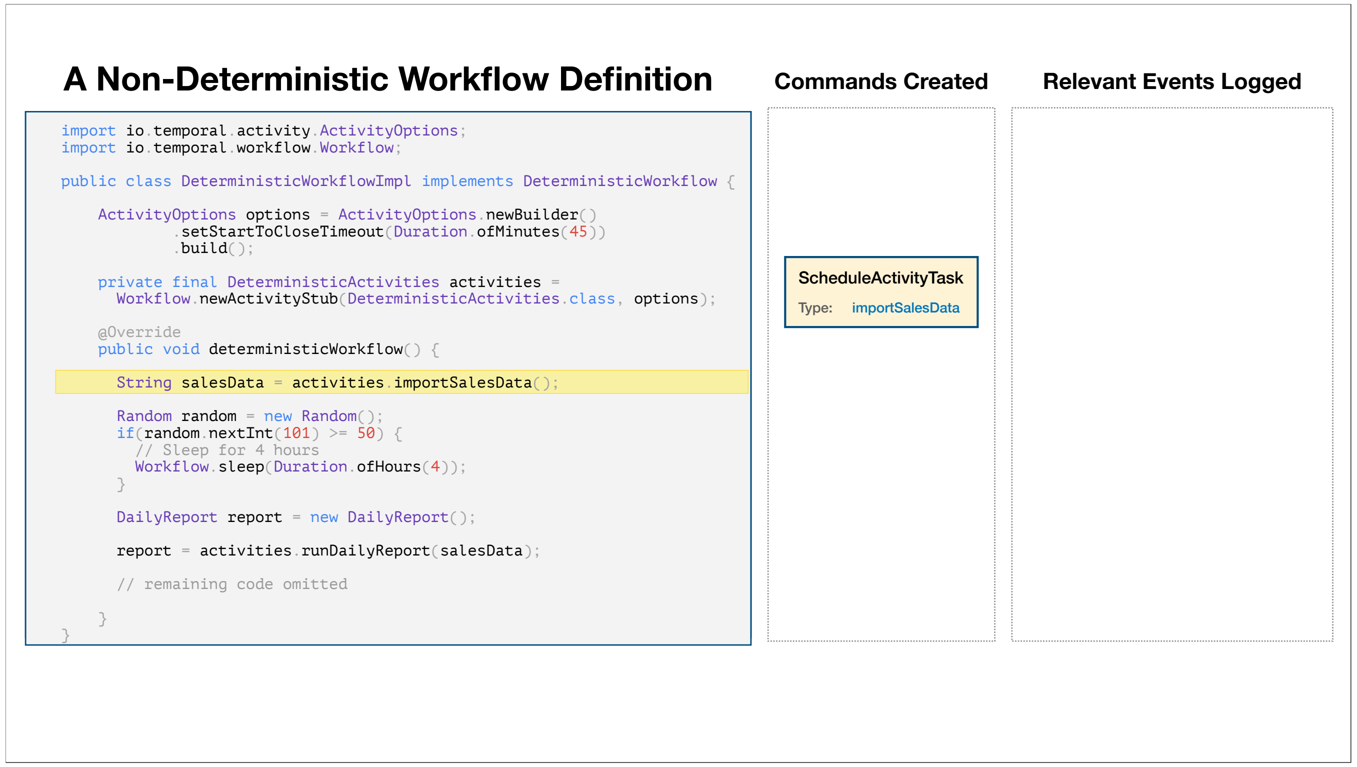
A Non-Deterministic Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Commands Created

Relevant Events Logged

Imagine that the following Workflow Definition is being executed. The first part behaves deterministically, because the lines above the highlighted one don't result in any Commands. The highlighted line does, but it should result in exactly the same Command generated each time it's executed.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

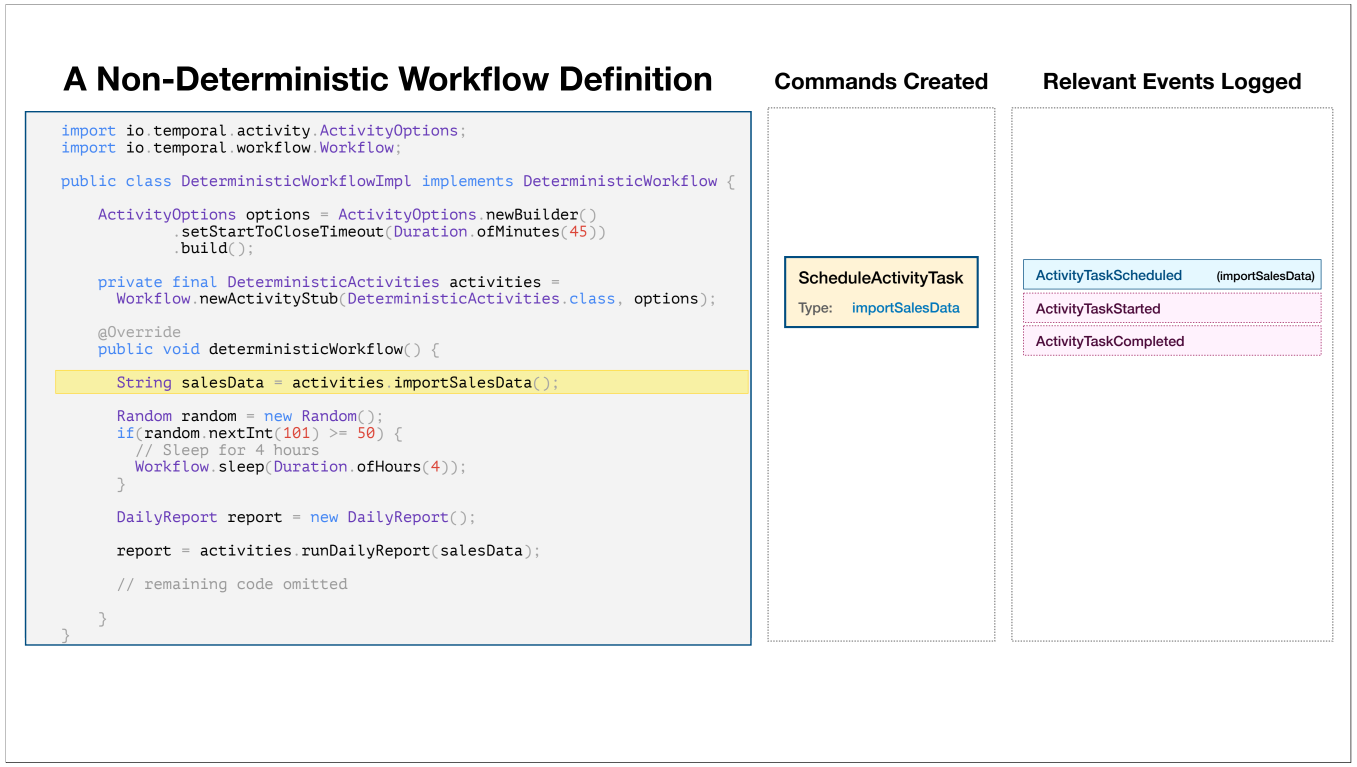
Commands Created

ScheduleActivityTask

Type: importSalesData

Relevant Events Logged

In this case, let's say that execution is successful,



A Non-Deterministic Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

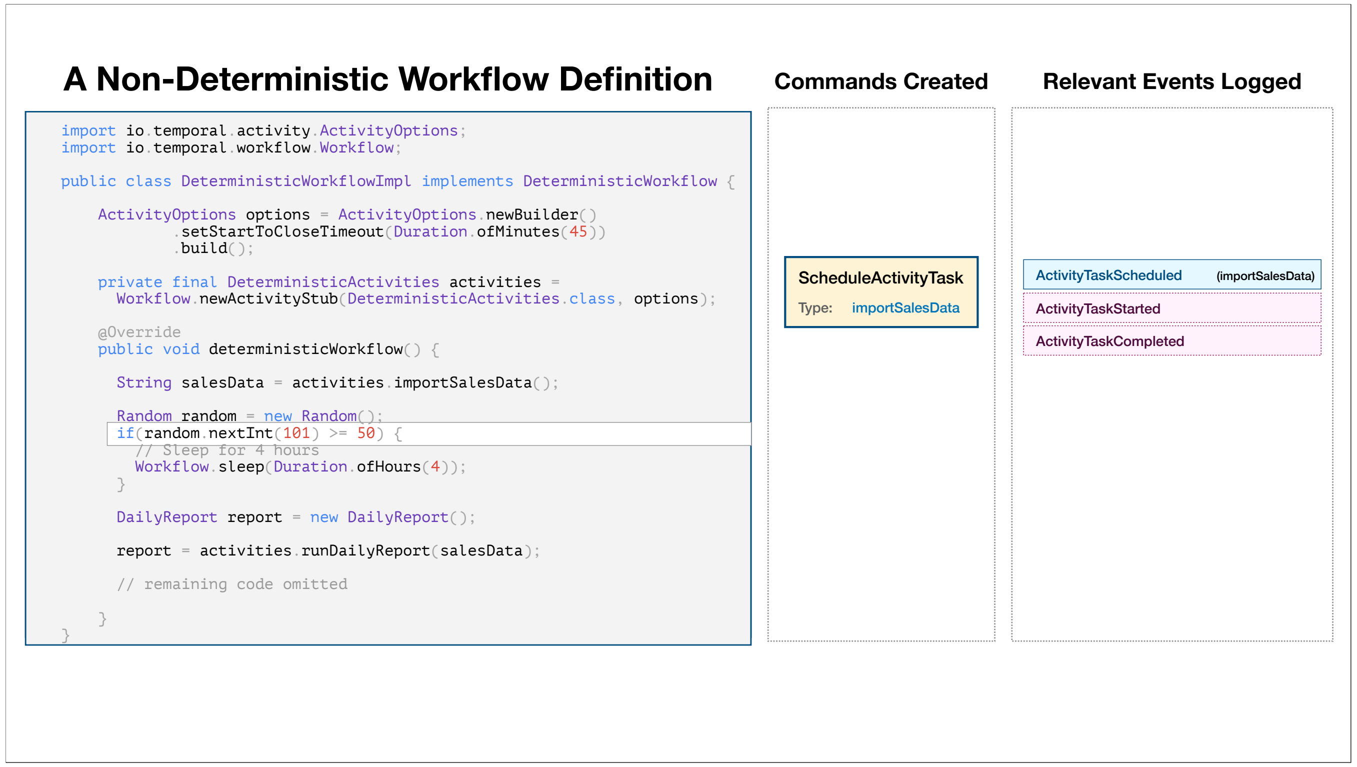
Commands Created

ScheduleActivityTask
Type: importSalesData

Relevant Events Logged

ActivityTaskScheduled (importSalesData)
ActivityTaskStarted
ActivityTaskCompleted

so the cluster logs the three Events shown into the history.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

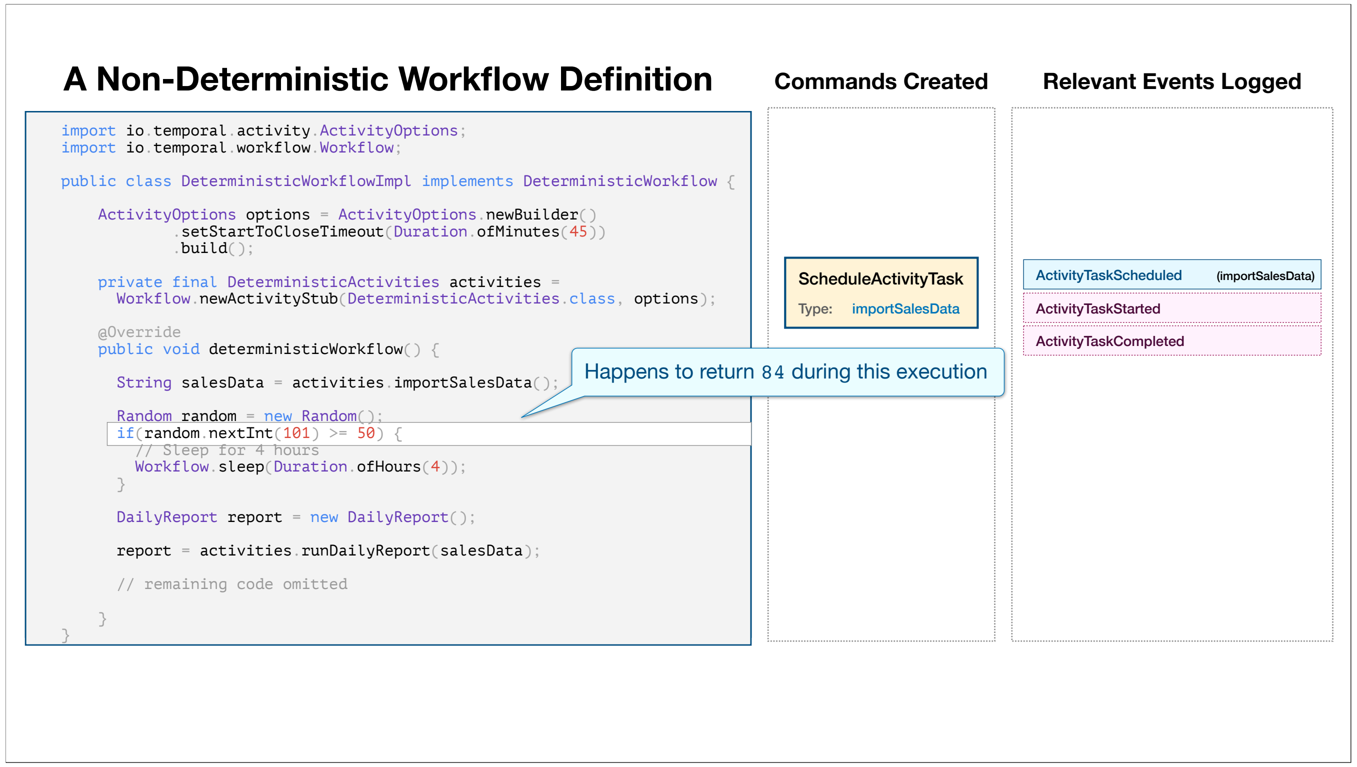
Commands Created

ScheduleActivityTask
Type: importSalesData

Relevant Events Logged

ActivityTaskScheduled (importSalesData)
ActivityTaskStarted
ActivityTaskCompleted

Next, there’s a conditional statement, which evaluates the value of a randomly-generated number.



A Non-Deterministic Workflow Definition

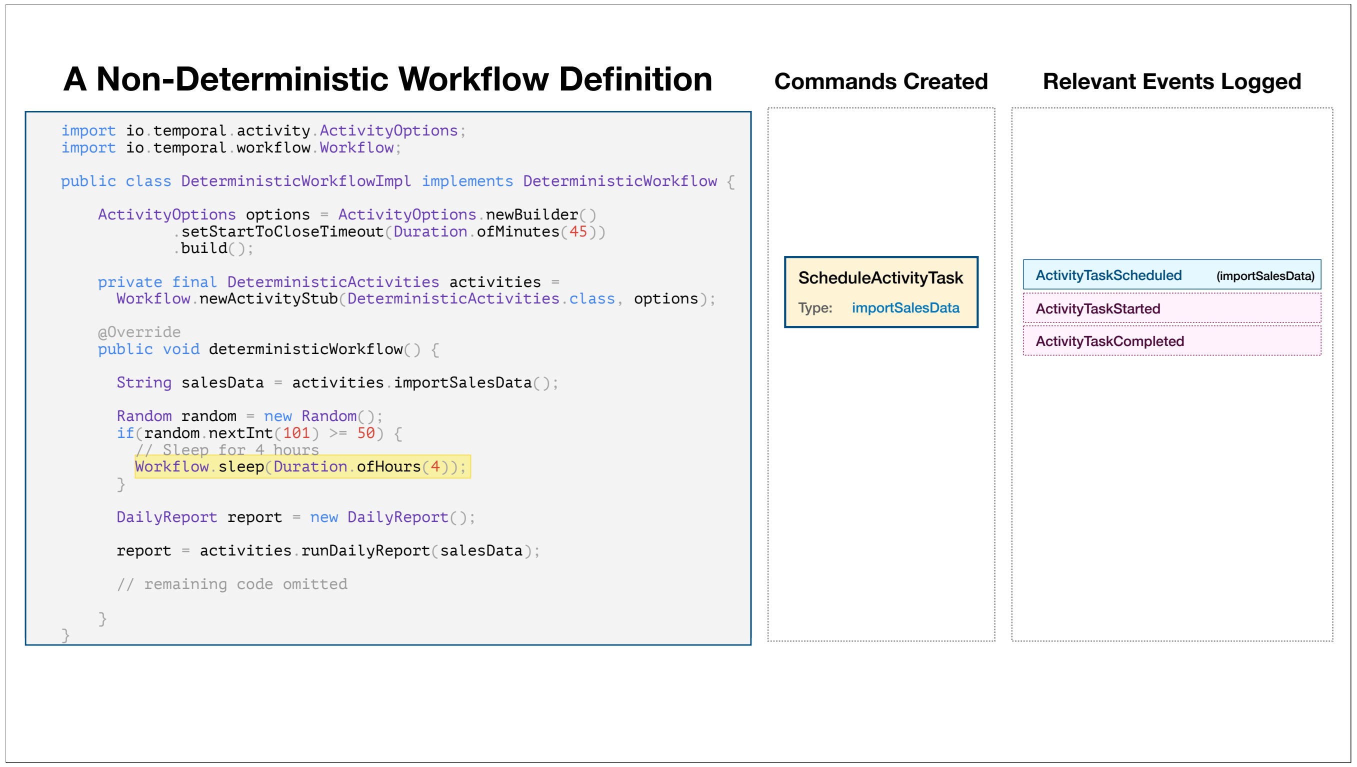
```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Happens to return 84 during this execution

Commands Created

Relevant Events Logged

Let's say that the random number generator happens to return the value 84 during this execution. Since the expression evaluates to true, execution continues with the next line.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

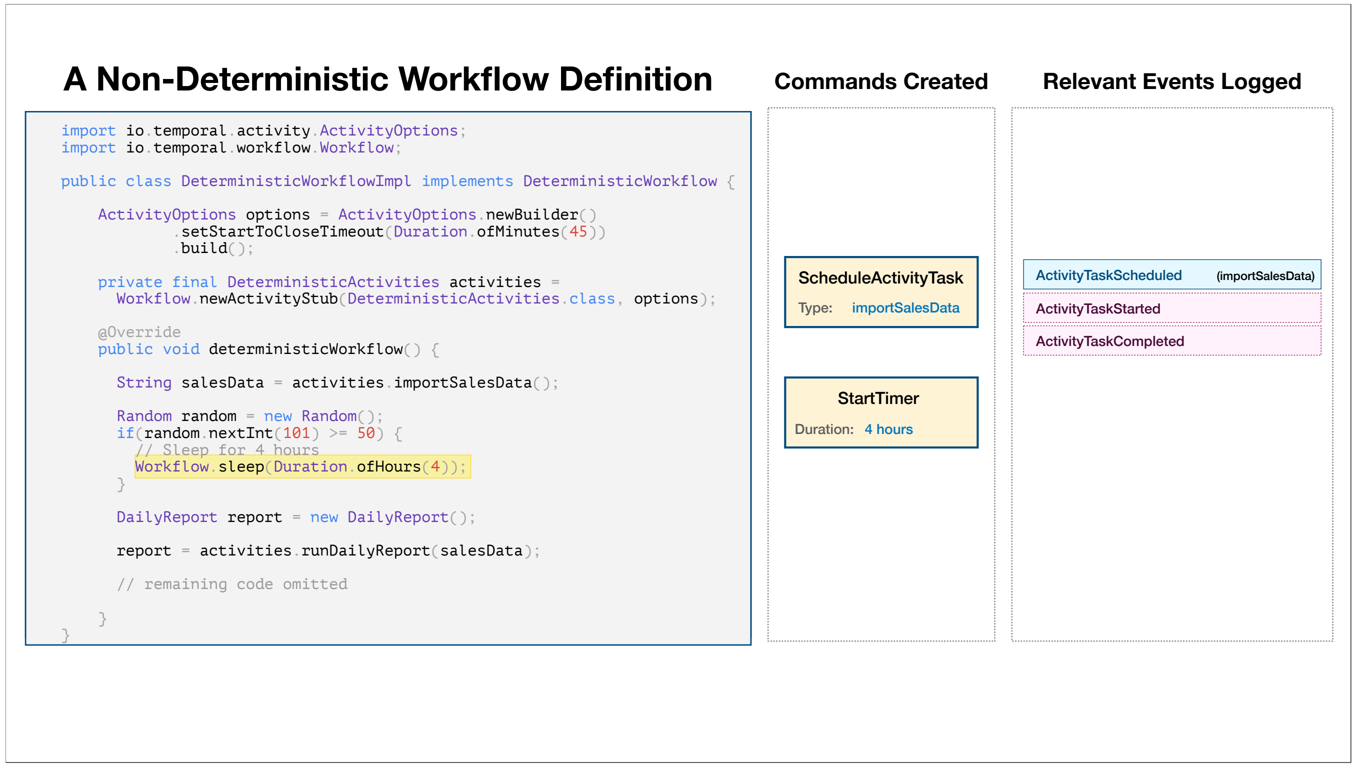
Commands Created

ScheduleActivityTask
Type: importSalesData

Relevant Events Logged

ActivityTaskScheduled (importSalesData)
ActivityTaskStarted
ActivityTaskCompleted

This contains a `sleep` statement,



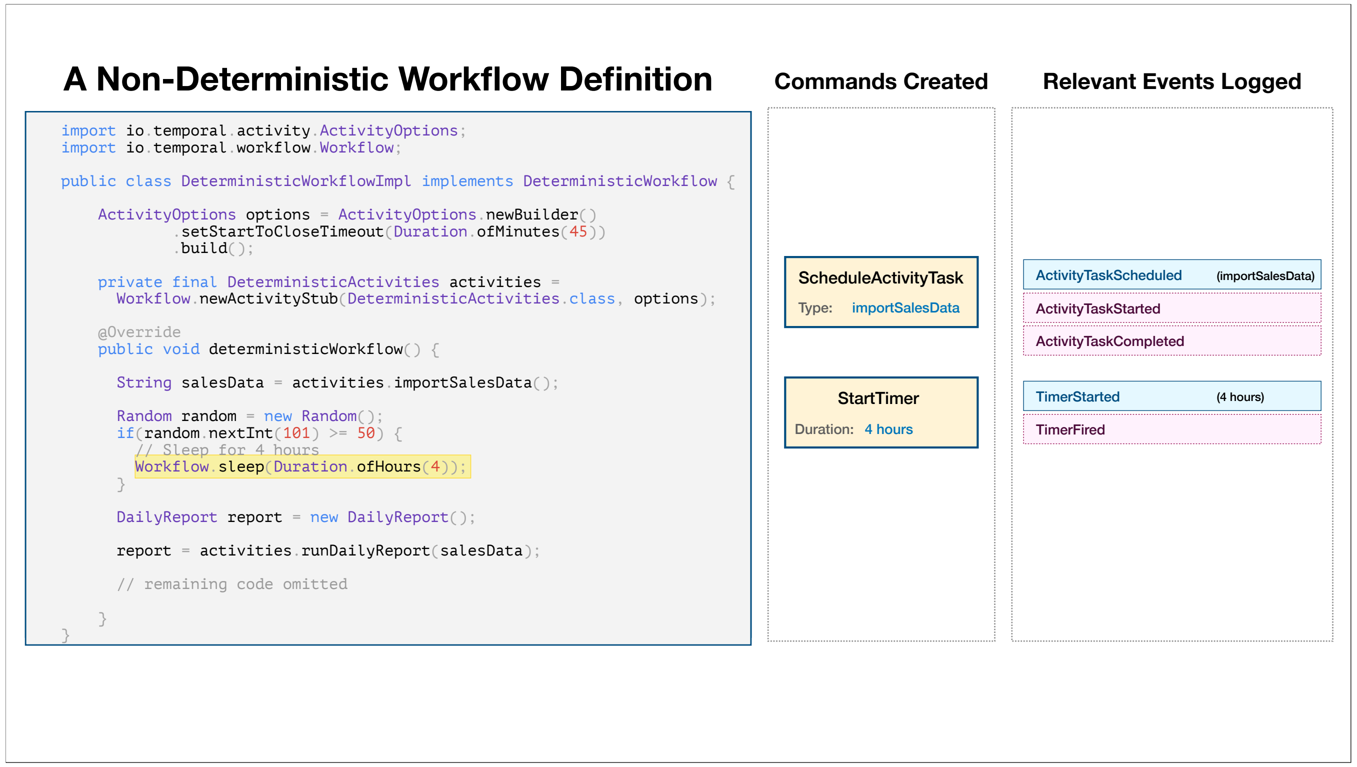
A Non-Deterministic Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Commands Created

Relevant Events Logged

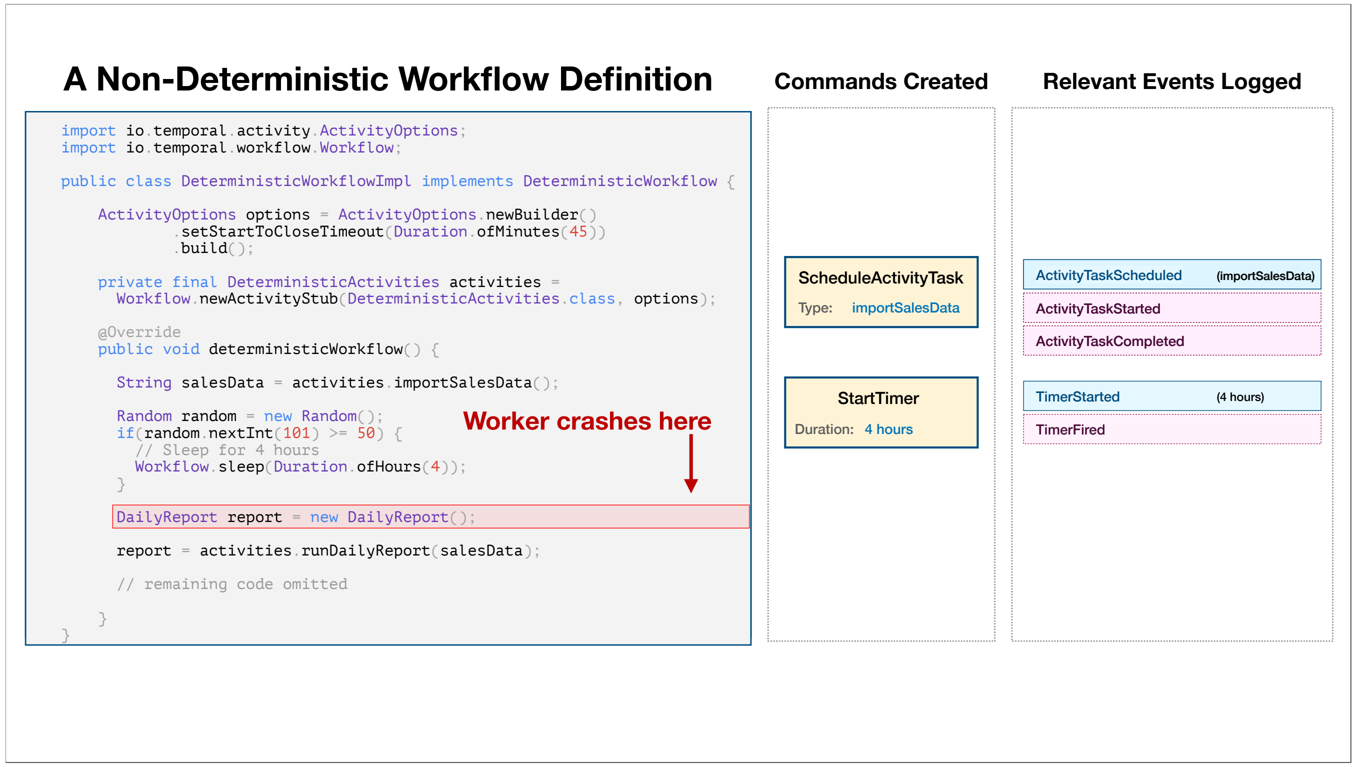
so the Worker issues a Command to the cluster, requesting that it starts a Timer.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

The cluster starts the Timer, logs an Event, and then logs another Event when the Timer fires.

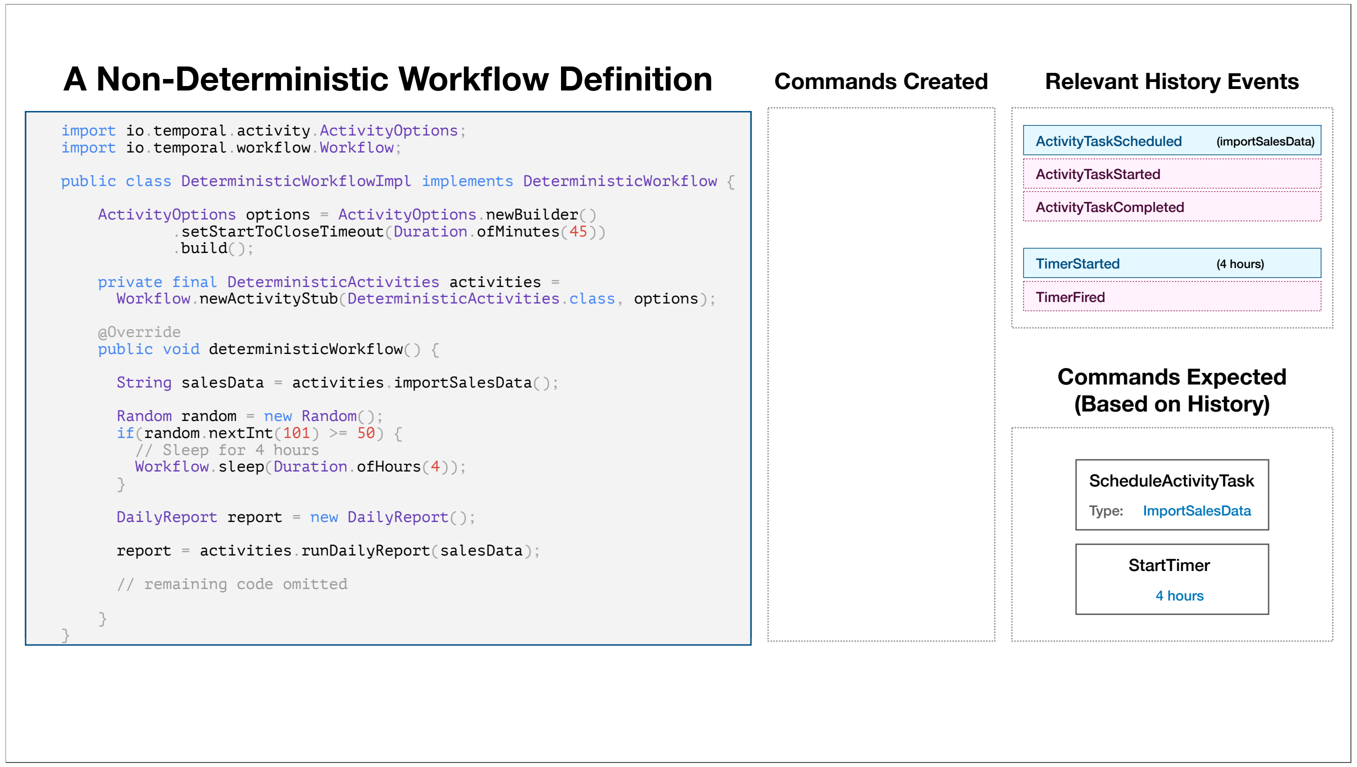


A Non-Deterministic Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Worker crashes here

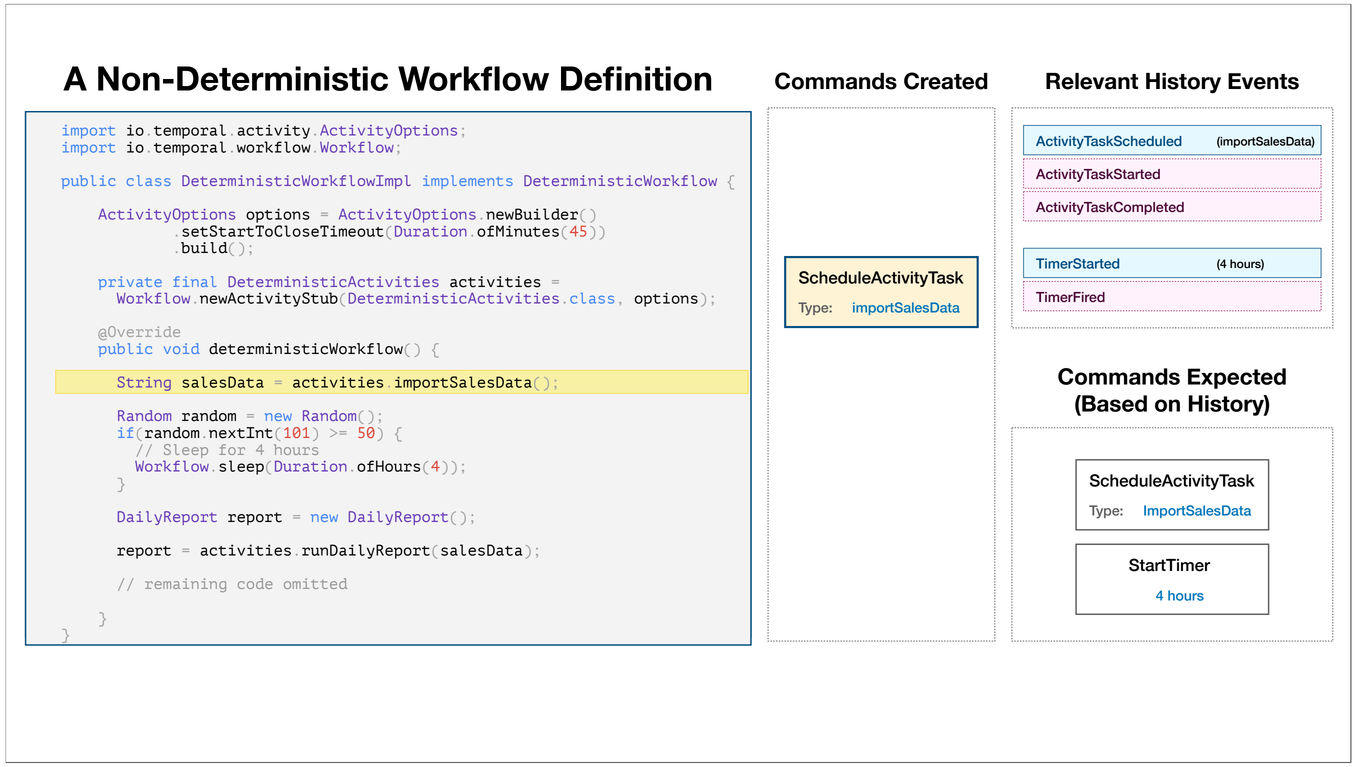
Now imagine that the Worker happens to crash once it reaches the next line, so another Worker takes over, using replay to restore the current state before continuing execution of the lines that follow.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

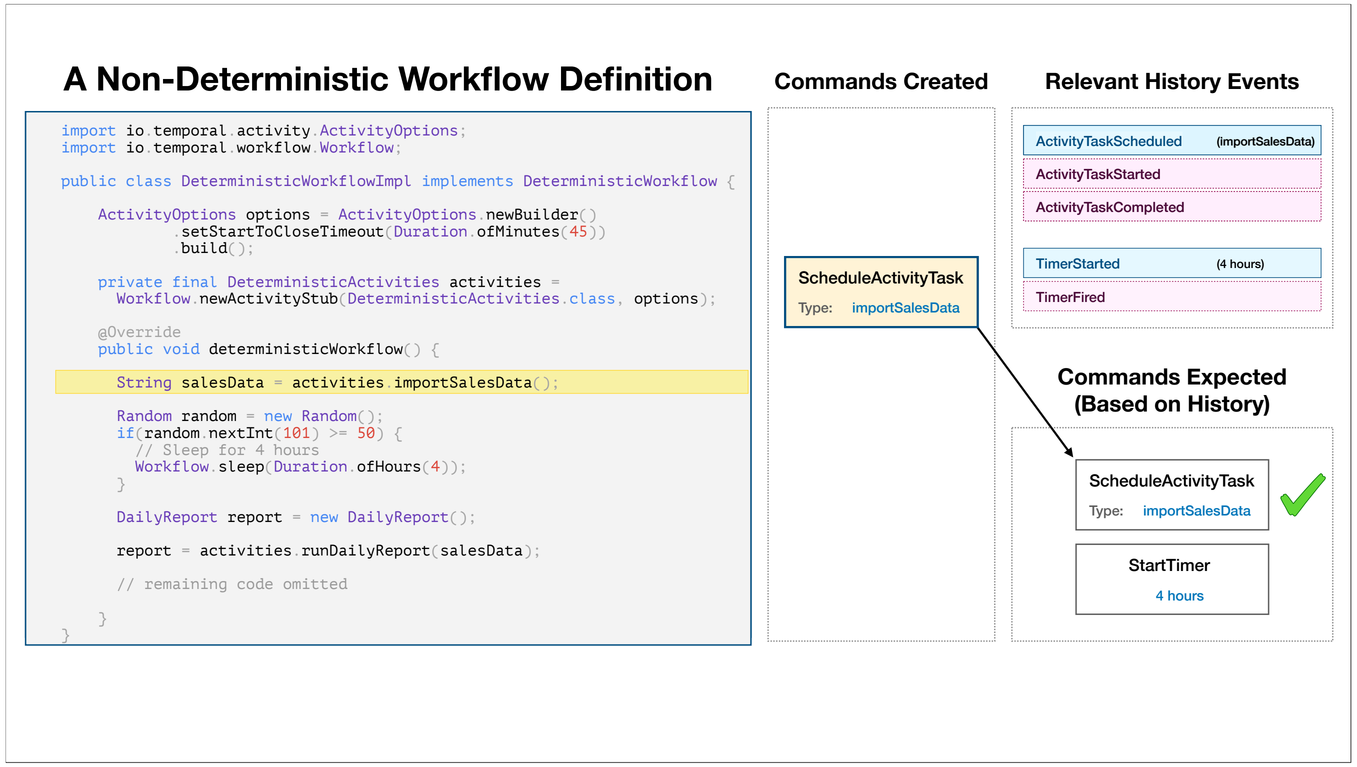
By evaluating the Event History, the Worker determines the expected sequence of Commands needed to restore the current state.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

As it executes the code during replay, it reaches the first `activity` call, and creates a `ScheduleActivityTask` Command.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
             .setStartToCloseTimeout(Duration.ofMinutes(45))
             .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Commands Created

ScheduleActivityTask
Type: importSalesData

Relevant History Events

ActivityTaskScheduled (importSalesData)
ActivityTaskStarted
ActivityTaskCompleted

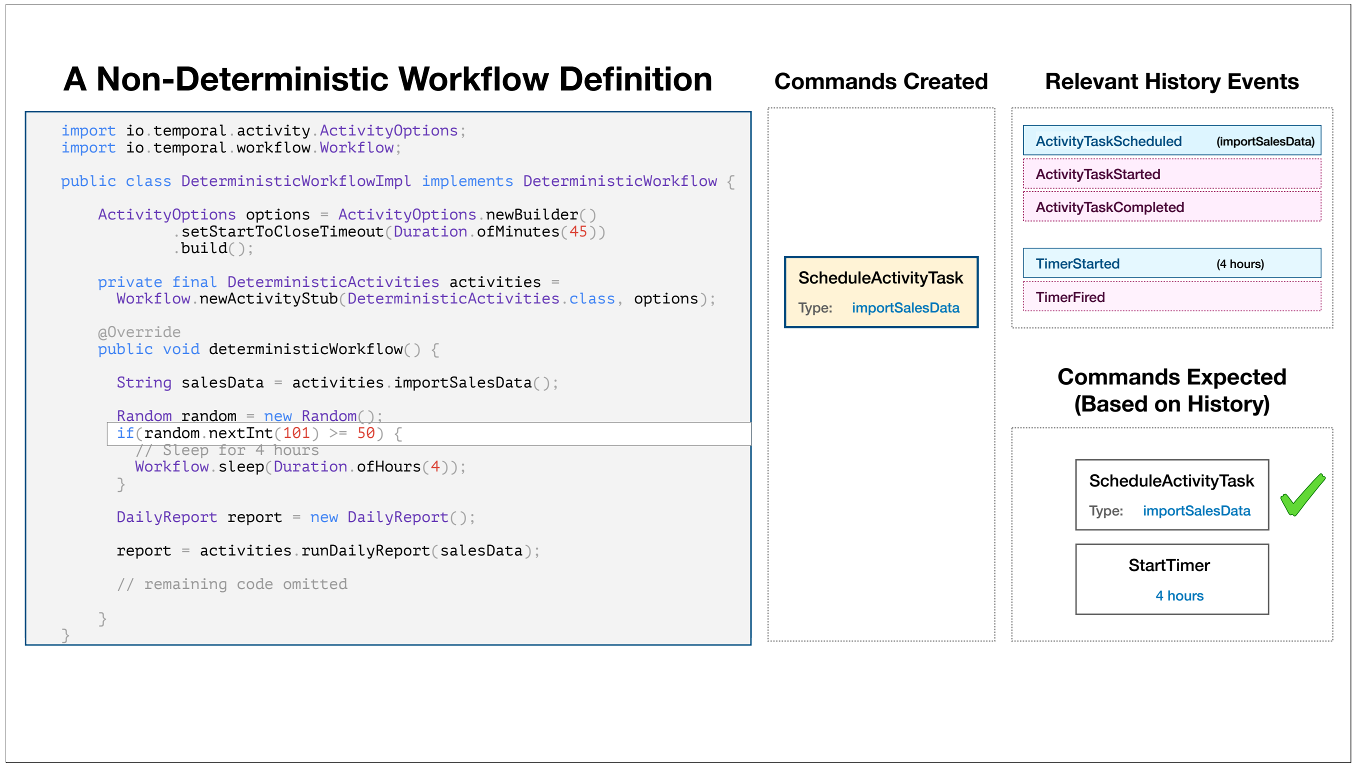
TimerStarted (4 hours)
TimerFired

Commands Expected (Based on History)

ScheduleActivityTask
Type: importSalesData ✓

StartTimer
4 hours

This Command matches the one expected based on the Event History. It's not only the right type of Command, with the same details, but it also occurs at the right position in the sequence of expected Commands. Therefore, the replay proceeds.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Commands Created

ScheduleActivityTask
Type: importSalesData

Relevant History Events

ActivityTaskScheduled (importSalesData)
ActivityTaskStarted
ActivityTaskCompleted

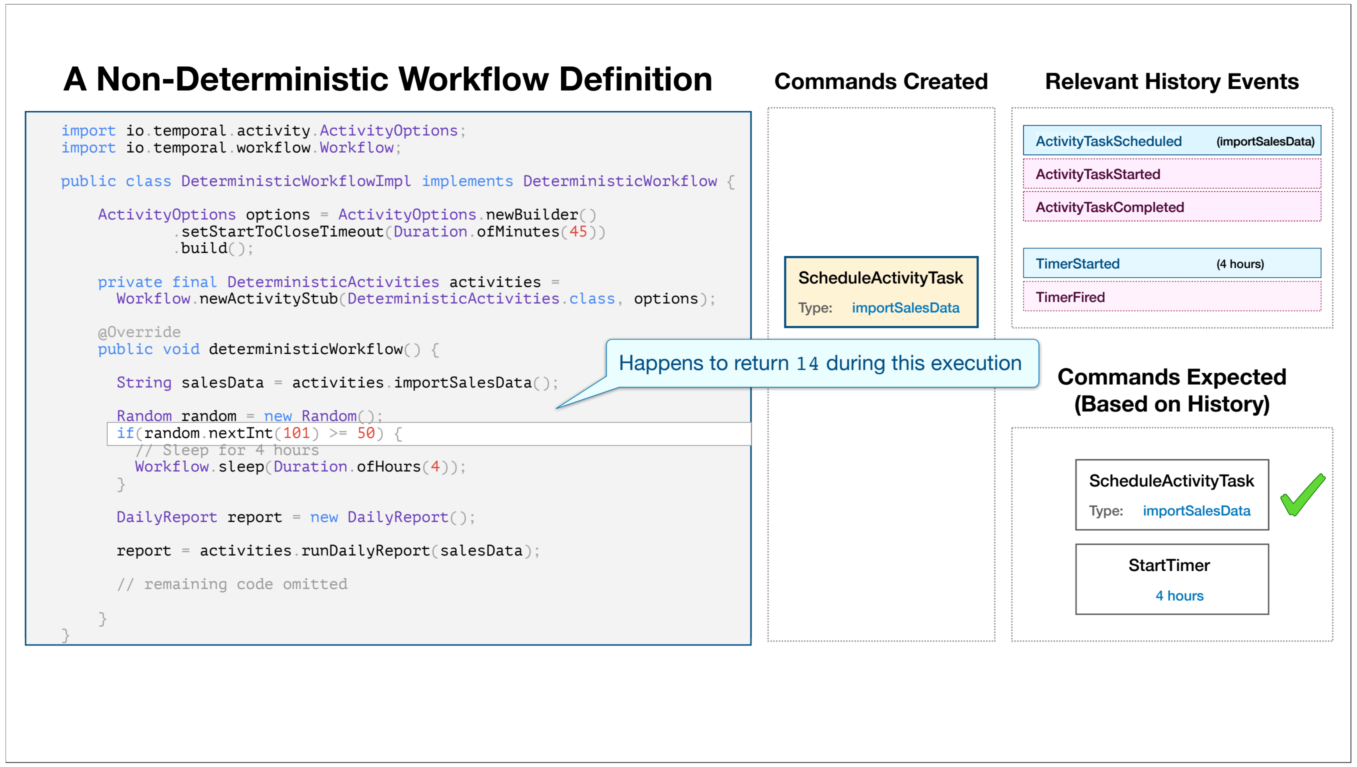
TimerStarted (4 hours)
TimerFired

Commands Expected (Based on History)

ScheduleActivityTask ✓
Type: importSalesData

StartTimer
4 hours

It now reaches the conditional statement with the random number generator.

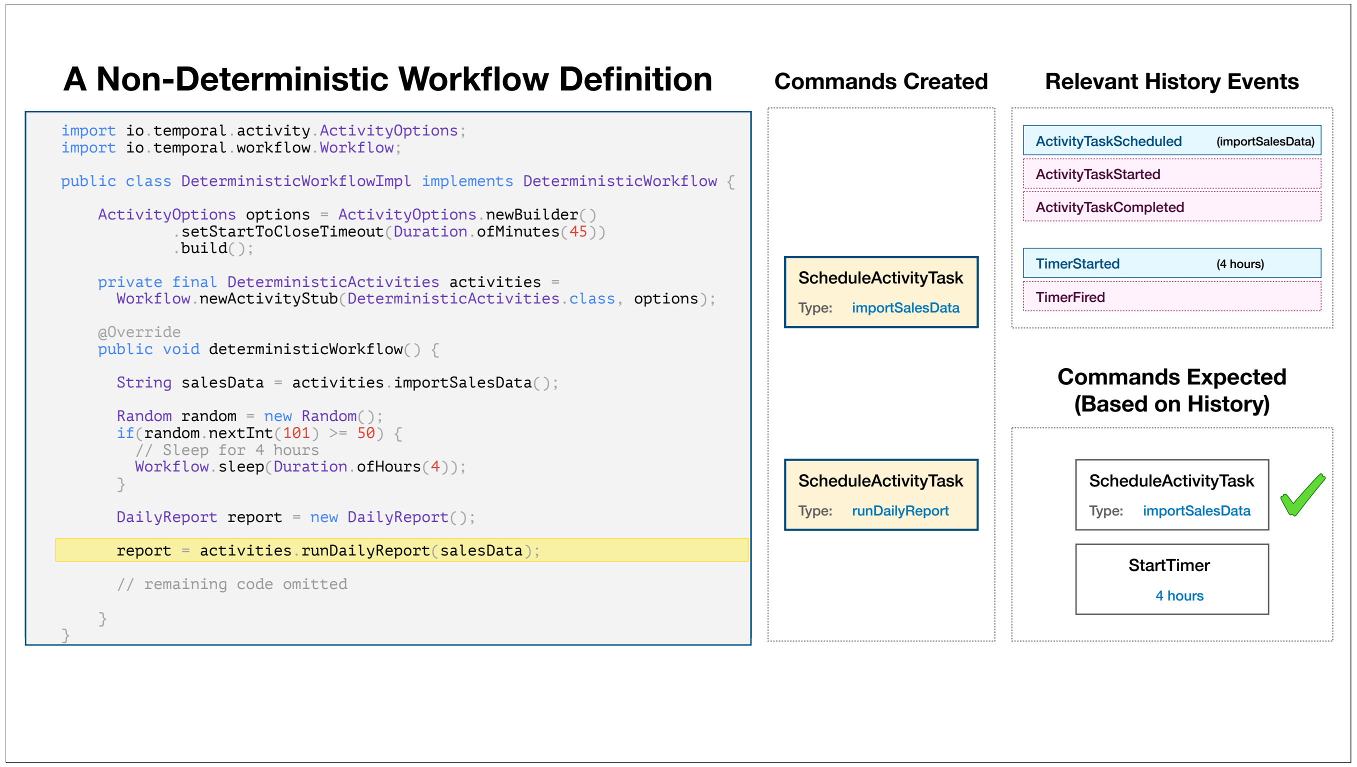


A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Happens to return 14 during this execution

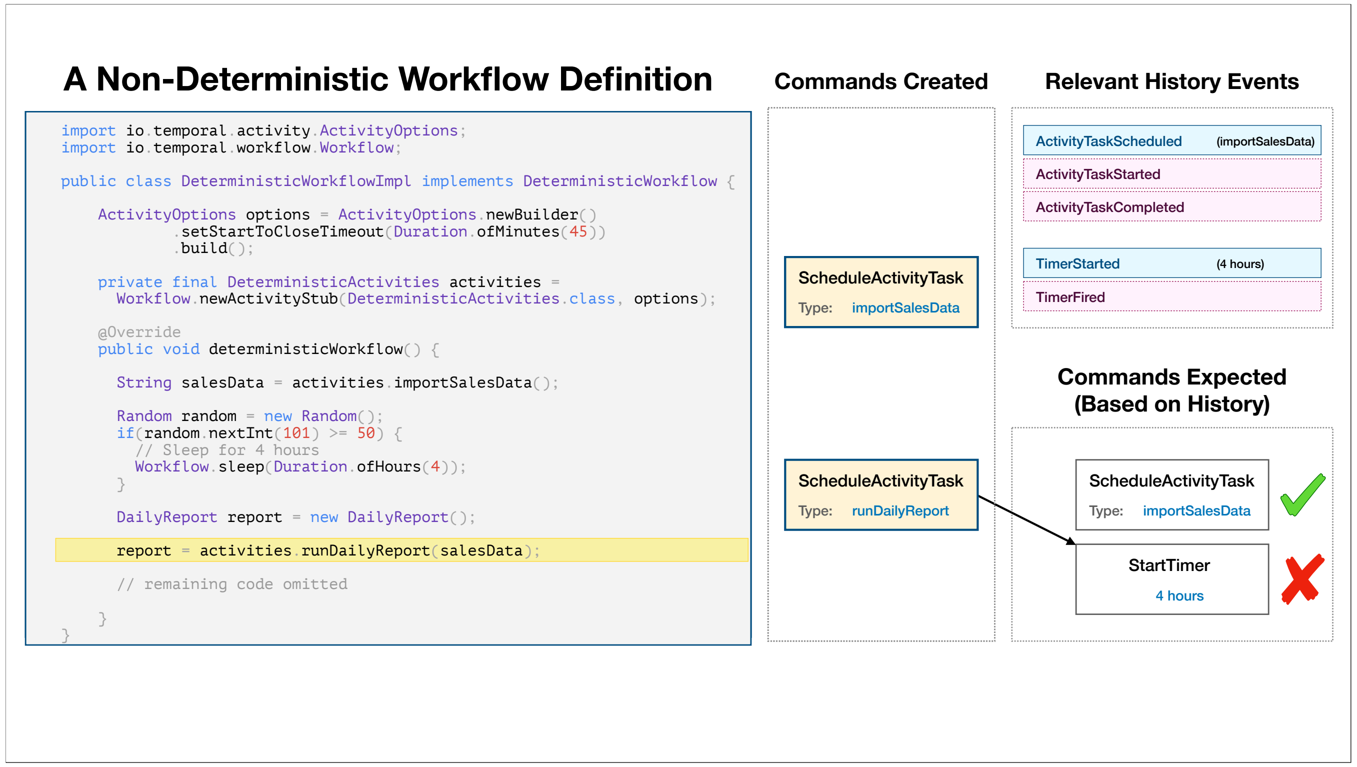
The random number generator happens to return 14 in this case, so the conditional expression evaluates to false, and execution skips over the next line.



A Non-Deterministic Workflow Definition

```
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Eventually, the Workflow requests execution of another Activity, so the Worker creates another `ScheduleActivityTask` Command.



A Non-Deterministic Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Commands Created

ScheduleActivityTask
Type: importSalesData

ScheduleActivityTask
Type: runDailyReport

Relevant History Events

ActivityTaskScheduled (importSalesData)
ActivityTaskStarted
ActivityTaskCompleted

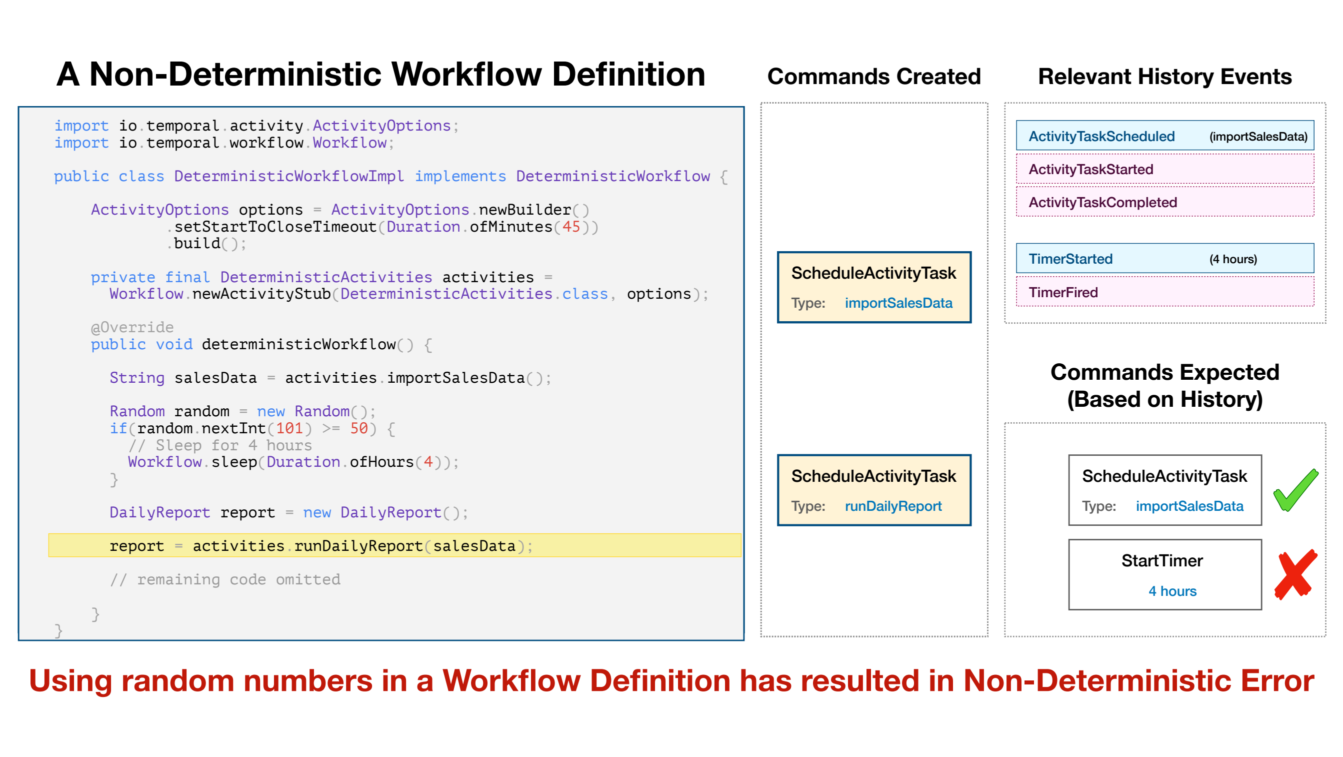
TimerStarted (4 hours)
TimerFired

Commands Expected (Based on History)

ScheduleActivityTask
Type: importSalesData ✓

StartTimer
4 hours ✗

However, this is a different Command than it expected to find at this position in the history.

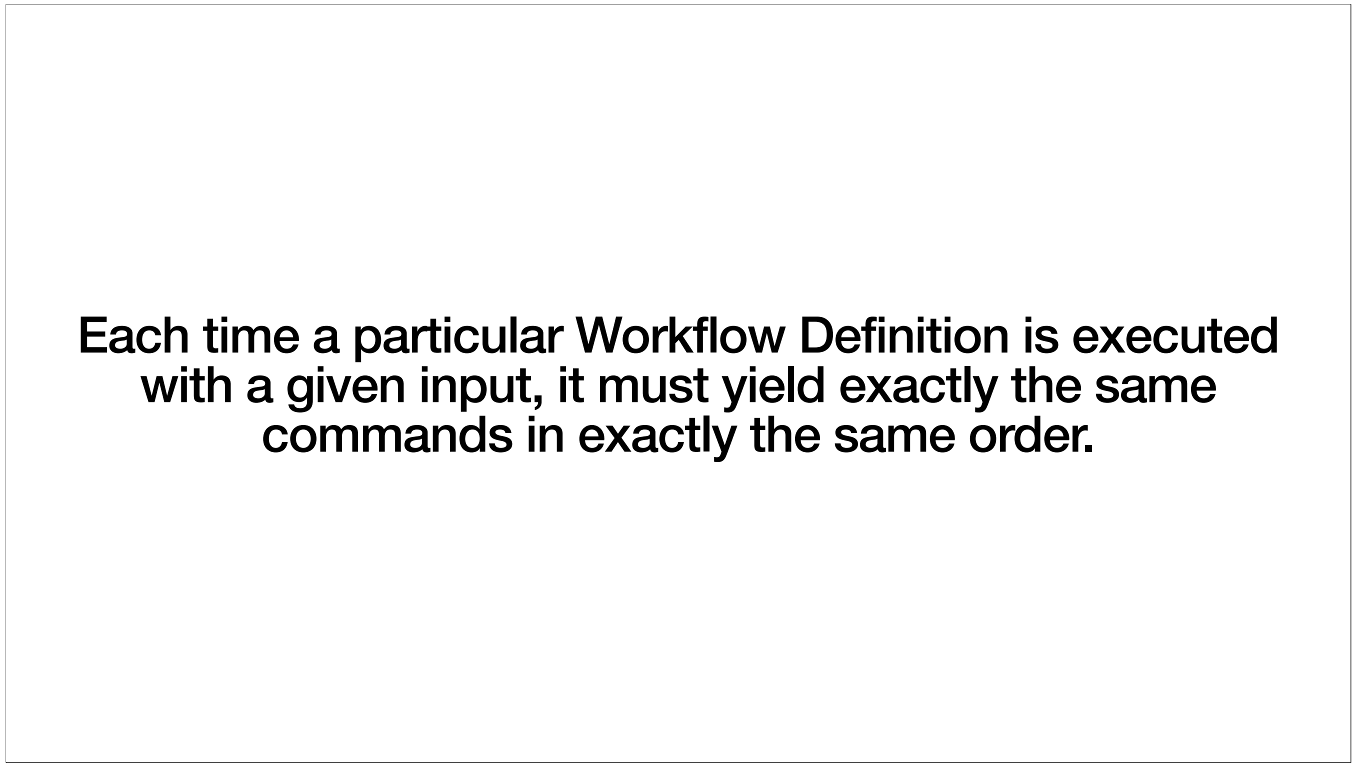


A Non-Deterministic Workflow Definition

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

public class DeterministicWorkflowImpl implements DeterministicWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(45))
            .build();

    private final DeterministicActivities activities =
      Workflow.newActivityStub(DeterministicActivities.class, options);

    @Override
    public void deterministicWorkflow() {

      String salesData = activities.importSalesData();

      Random random = new Random();
      if(random.nextInt(101) >= 50) {
        // Sleep for 4 hours
        Workflow.sleep(Duration.ofHours(4));
      }

      DailyReport report = new DailyReport();

      report = activities.runDailyReport(salesData);

      // remaining code omitted

    }
}
```

Using random numbers in a Workflow Definition has resulted in Non-Deterministic Error

Since the Workflow Definition produced a different sequence of Commands during replay than it did prior to the crash, the Worker is unable to restore the previous state, so the use of random numbers in the Workflow code has resulted in a 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.

As a developer, it's important to understand that the Workflow code you write can not catch or handle non-deterministic errors. Instead, you must recognize and avoid the problems that cause them.

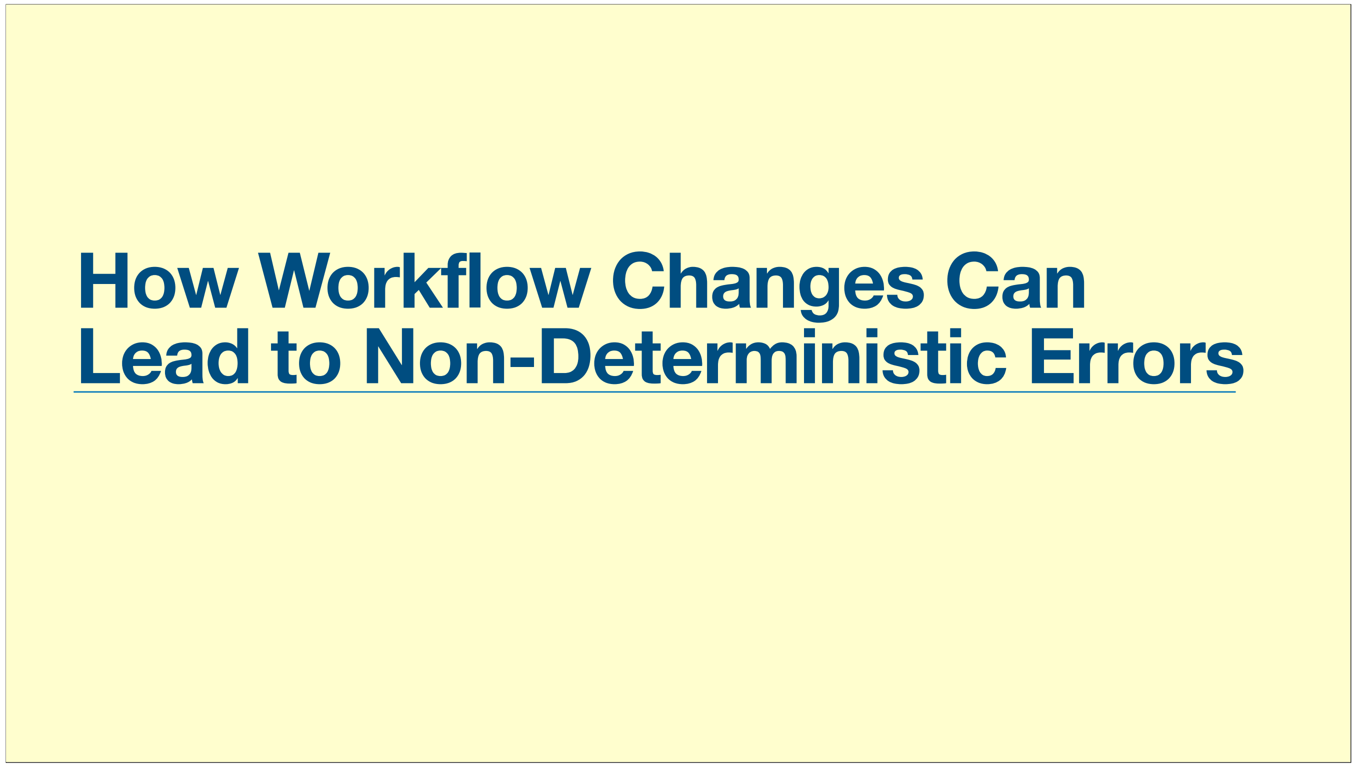
Common Sources of Non-Determinism

Let’s look at some of the most common sources of non determinism.

Let’s look at some of the most common sources of non determinism.

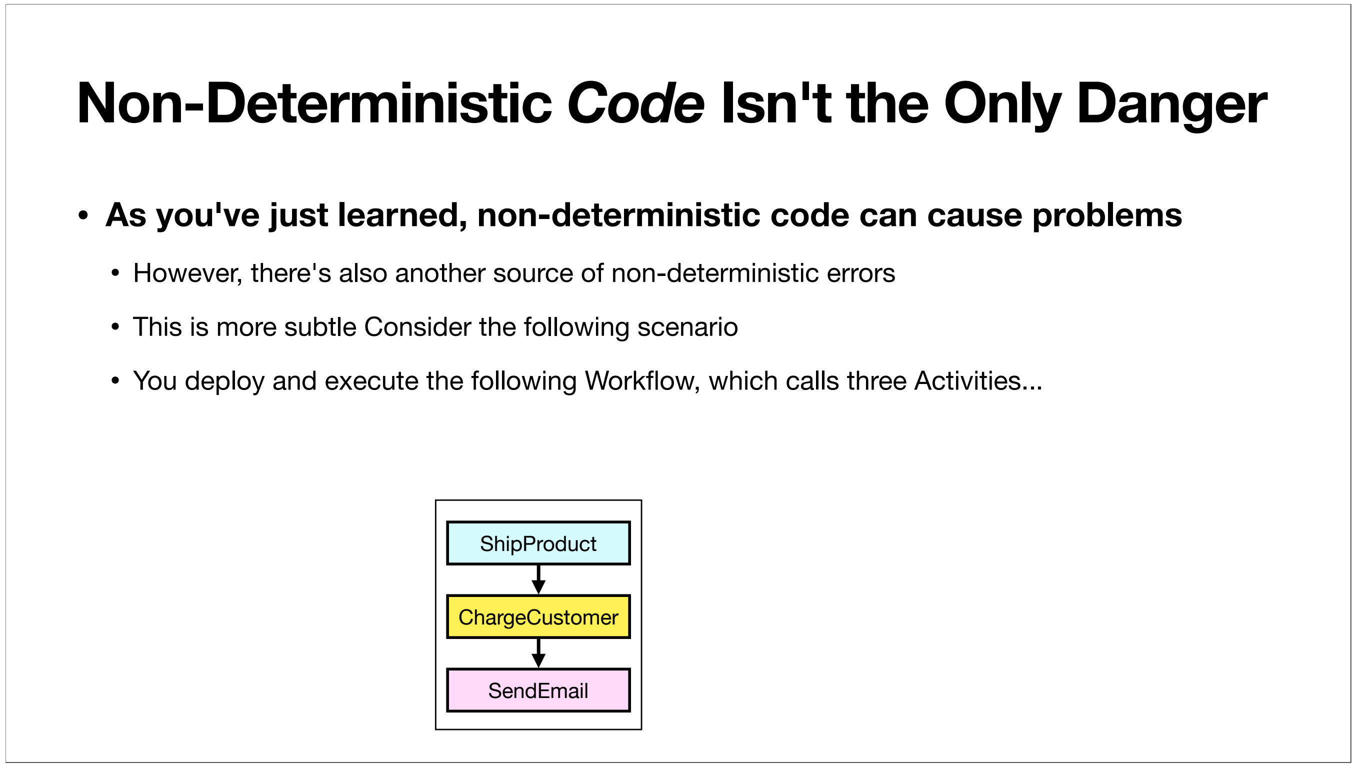
Things to Avoid in a Workflow Definition

- **Accessing external systems, such as databases or network services**
  - Instead, use Activities to perform these operations
- **Writing business logic or calling functions that rely on system time**
  - Instead, use Workflow-safe functions such as `Workflow.currentTimeMillis` and `Workflow.sleep`
- **Working directly with threads**
- **Do not iterate over data structures with unknown ordering**



How Workflow Changes Can Lead to Non-Deterministic Errors

There are other kinds of changes that can lead to non deterministic errors.

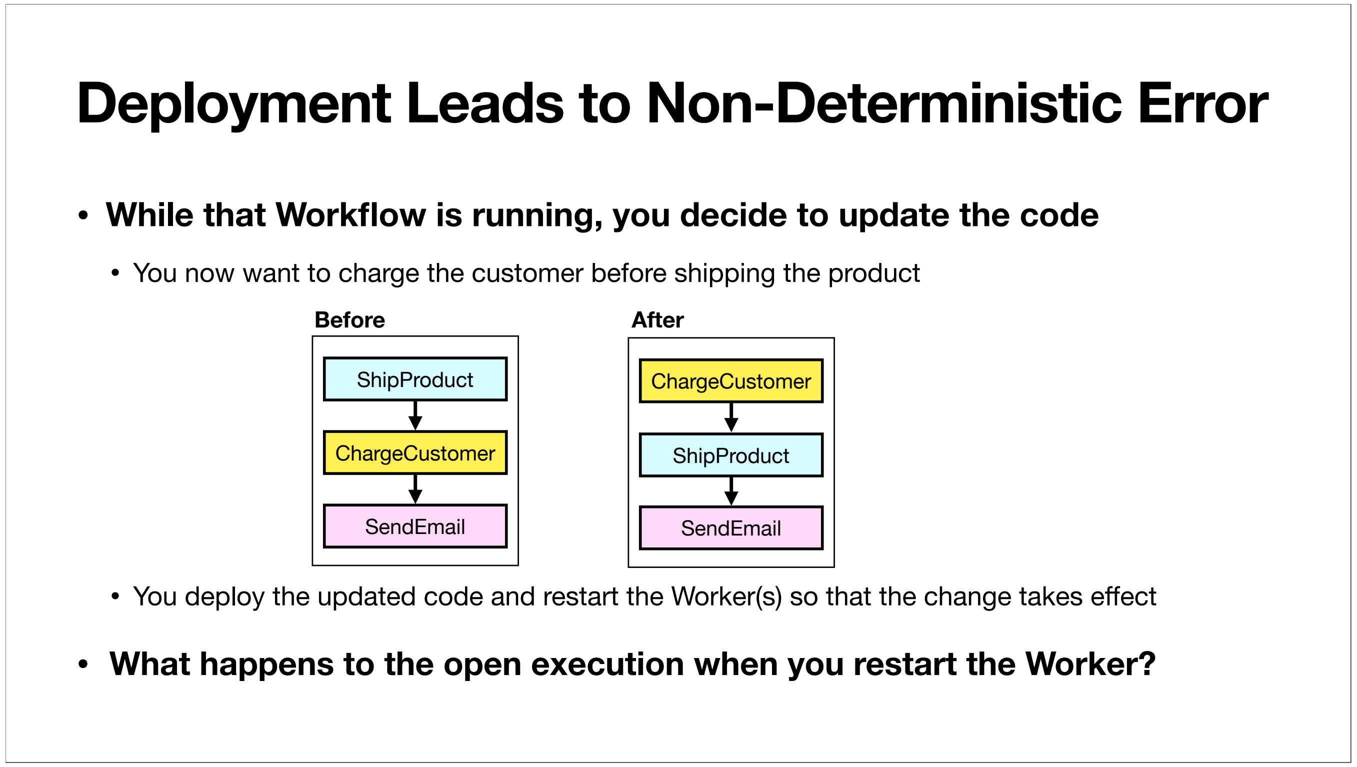


Non-Deterministic *Code* Isn't the Only Danger

- **As you've just learned, non-deterministic code can cause problems**
  - However, there's also another source of non-deterministic errors
  - This is more subtle Consider the following scenario
  - You deploy and execute the following Workflow, which calls three Activities...

ShipProduct
ChargeCustomer
SendEmail

This is part one of the scenario: part 2 is on the next slide.

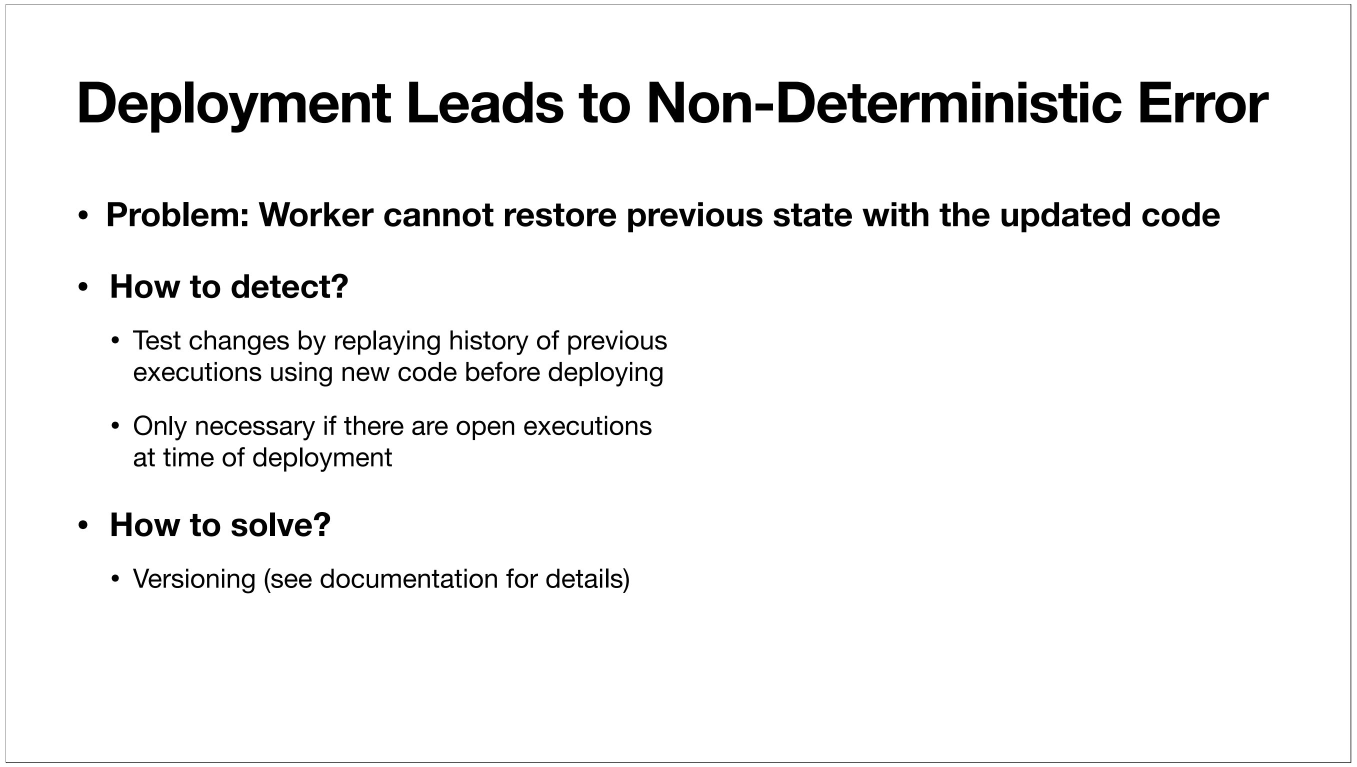


Deployment Leads to Non-Deterministic Error

- **While that Workflow is running, you decide to update the code**
  - You now want to charge the customer before shipping the product

  - 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?**

Suggestion: Ask the learners out loud and see if anyone can reason through it, based on what they've learned, to come up with the correct answer.



Deployment Leads to Non-Deterministic Error

- **Problem: Worker cannot restore previous state with the updated code**
- **How to detect?**
  - Test changes by replaying history of previous executions using new code before deploying
  - Only necessary if there are open executions at time of deployment
- **How to solve?**
  - Versioning (see documentation for details)

What happens is that the Worker uses History Replay to reconstruct the state of the open execution from just prior to the restart. However, if the Event History indicates that its first Activity (ShipProduct) has already been started, this will result in a non-deterministic error. Why? Because the updated code produced a different sequence of Commands than the original code did, so it's impossible to recreate the original state with the new version of the code.

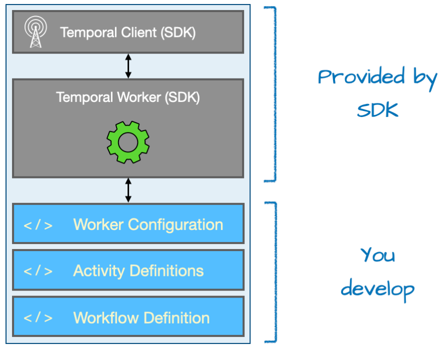
You can solve this problem by using Versioning, and although we don't have time to cover that during the live version of Temporal 102, you can read about it in our documentation (or in the online version of Temporal, if available for your SDK).

Temporal 102

00. About this Workshop
01. Understanding Key Concepts in Temporal
02. Improving Your Temporal Application Code
03. Using Timers in a Workflow Definition
04. Testing Your Temporal Application Code
05. Understanding Event History
06. Debugging Workflow History
07. Deploying Your Application to Production
08. Understanding Workflow Determinism
▶ **09. Conclusion**

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 Cluster / 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 Cluster / Cloud
- **As Workers run your code, they send Commands to Temporal Cluster/Cloud**
  - For example, when encountering calls to Activity Methods or `Workflow.sleep`
    or when returning a result from the Workflow Definition
- **Commands sent by the Worker lead to Events logged by Temporal Cluster / Cloud**

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 (but it should be idempotent)
    - Activities are automatically retried upon failure, according to a configurable Retry Policy

Essential Points (6)

- **Recommended best practices for Temporal app development**
  - Use classes (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, ideally integrating with a 3rd-party logging package

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)

Thank You