



Temporal 102



Replay 2023

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**
- **Our Helpers**
- **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)

<https://t.mp/replay-102-typescript>



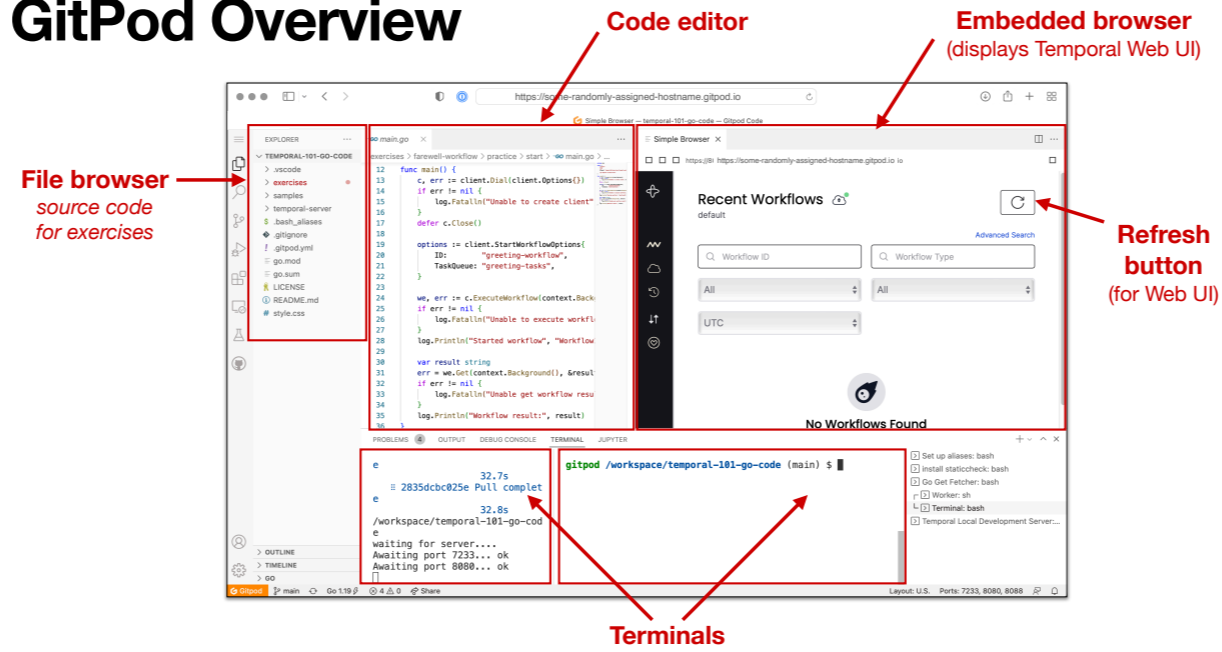
In this course you'll use an exercise environment powered by GitPod. You won't need to install anything on your personal machine to do the exercises. This environment will include a text editor, running terminals, and a Temporal development cluster.

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.

--

TODO: Show the URL in the course material here: you need to set up a short URL and show it on this screen so that people can enter it in their own browser. Make sure that they do that before you proceed. Having them launch GitPod now ensures that it will be initialized and ready for use by the time you get to the first exercise (although it may go to sleep between now and then, it's much faster to wake it back up than to initialize it from scratch).

GitPod Overview



--

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



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 TypeScript SDK, you define a Temporal Workflow as an exportable function that returns a Promise
- Temporal requires that Workflows are *deterministic*

 Workflow Definition



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 written in a general-purpose programming language such as 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 TypeScript SDK, you define Activities as exportable functions that return Promises
 - 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

</> Activity Definitions

</> Workflow Definition



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 .

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

</> Worker Configuration

</> Activity Definitions

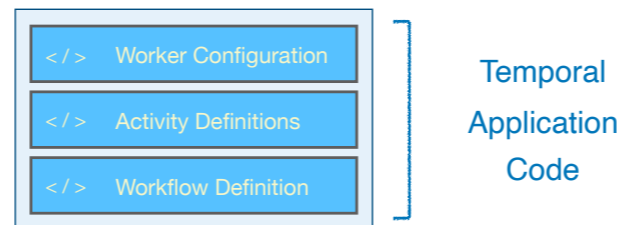
</> Workflow Definition



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.

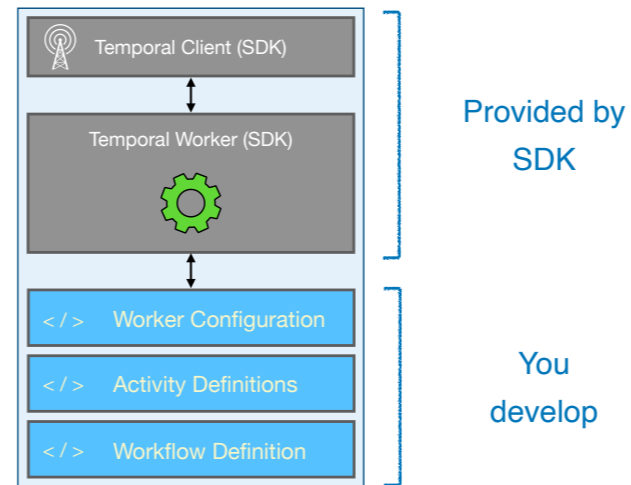
The Worker implementation is provided by the Temporal SDK, so you don't need to write it. You will need to configure it, though, by specifying the Task Queue name and registering your Workflow and Activity functions. You'll also write code to start the Worker.

Code You Develop



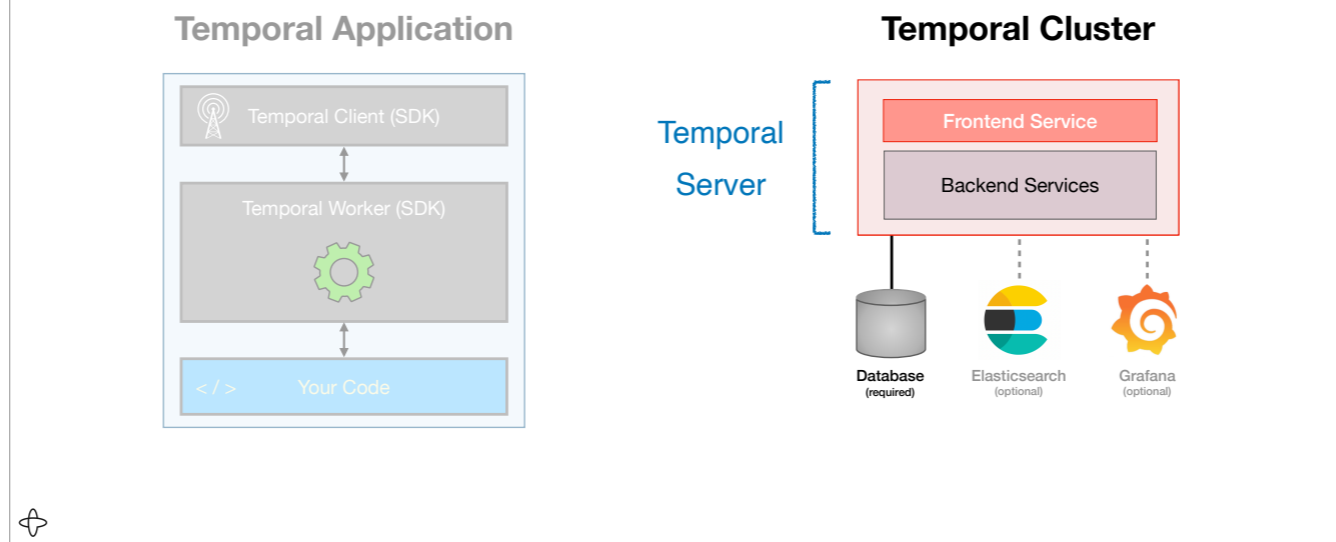
In summary, a Temporal application developer is responsible for writing Workflow Definitions, Activity Definitions, and the code to configure and start the Workers that coordinate with a Temporal Cluster to carry execution forward. Since these represent the code that you, the developer, are responsible 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



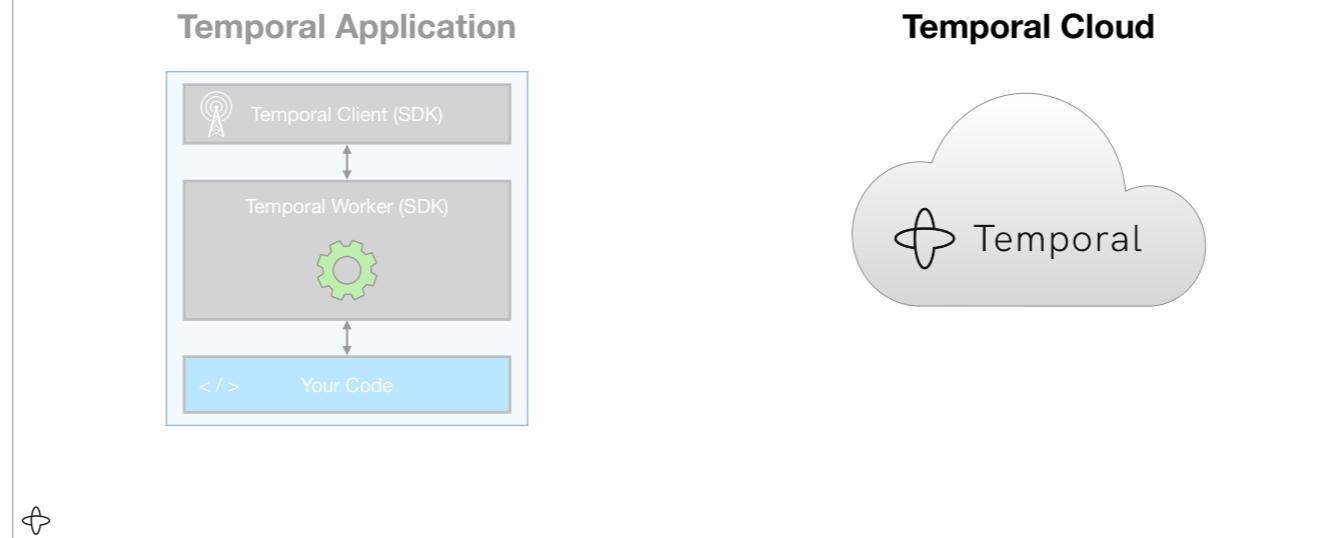
I think it's helpful to think of a Temporal application as having two parts: one that you develop and another part provided by the SDK.

The Role of Temporal Cluster



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.

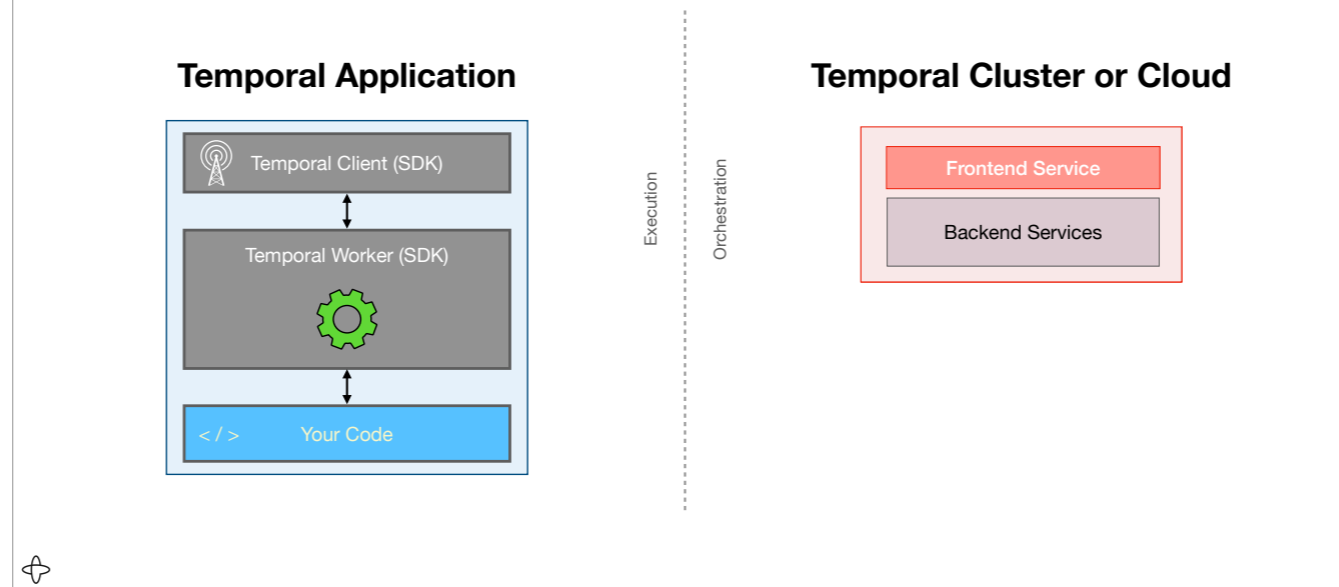
The Role of Temporal Cloud



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.

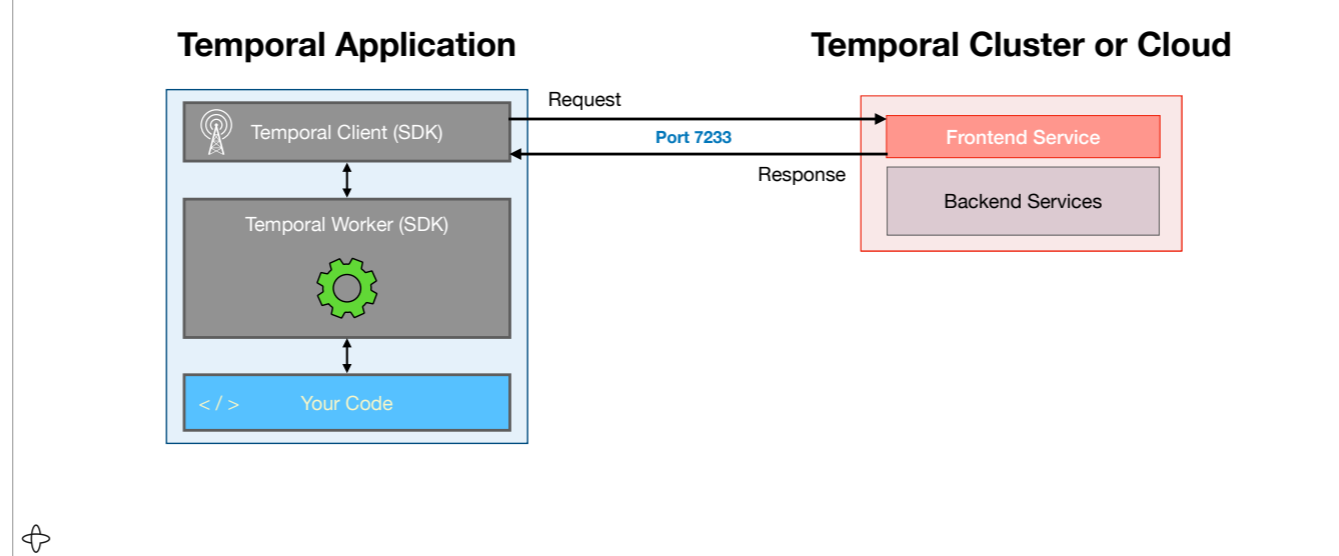
Applications Are External to the Cluster



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**
 - Changes to the composition of this object does not affect the function signature
- **This is also the recommended approach for return values**
 - Using object 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 an Object in an Activity (1)

- Imagine that you have the following Activity

```
// This Activity returns a customized greeting in Spanish, using the provided name
export async function getSpanishGreeting(name: string): Promise<string> {
  // implementation omitted for brevity
```

↑
input

↑
output

- You later need to update it to support other languages, such as German.
 - Changing what is passed into or returned from the function changes its signature
 - Changes to the input value's composition don't affect the signature of the functions 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 function 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 function can change the function's signature.

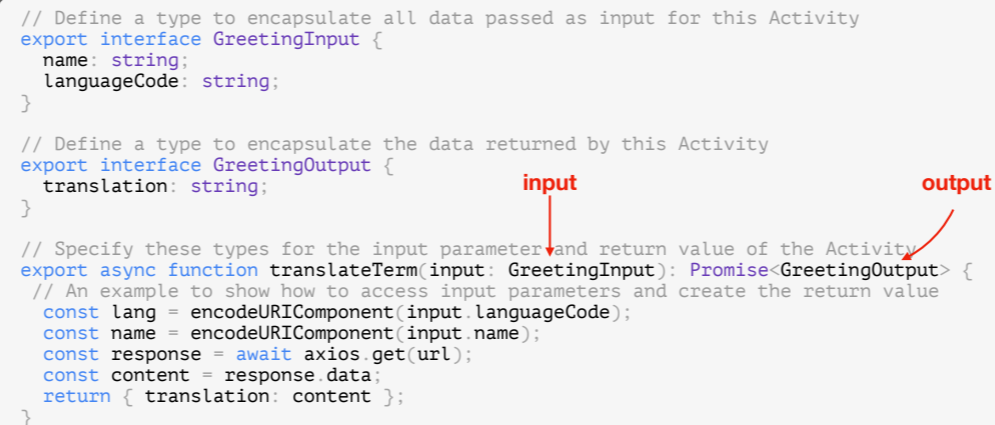
Example: Using an Object in an Activity (2)

- The following code sample illustrates how you could support this

```
// Define a type to encapsulate all data passed as input for this Activity
export interface GreetingInput {
  name: string;
  languageCode: string;
}

// Define a type to encapsulate the data returned by this Activity
export interface GreetingOutput {
  translation: string;
}

// Specify these types for the input parameter and return value of the Activity
export async function translateTerm(input: GreetingInput): Promise<GreetingOutput> {
  // An example to show how to access input parameters and create the return value
  const lang = encodeURIComponent(input.languageCode);
  const name = encodeURIComponent(input.name);
  const response = await axios.get(url);
  const content = response.data;
  return { translation: content };
}
```



This code example illustrates a better design for this Activity. It begins by defining a interface that represents input to this Activity, which includes the original name but also the new language code field. It then defines another interface, which represents the output returned by the Activity function. It just contains the translated greeting for now, but you could update this as requirements change in the future. Finally, the Activity function 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 Objects for Data

- **During this exercise, you will**
 - Examine how the Workflow uses objects for input parameters and return values
 - Define types to represent input and output of an Activity Definition
 - Update the code to use the objects and types 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.

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.

Specifying the Task Queue

- **Client**

```
await client.workflow.start(OrderProcessingWorkflow, {  
  args: [order],  
  taskQueue: TASK_QUEUE_NAME,  
  workflowId: `workflow-order-${order.id}`,  
})
```

- **Worker**

```
const worker = await Worker.create({  
  taskQueue: TASK_QUEUE_NAME,  
  connection,  
  workflowsPath: require.resolve('./workflows'),  
  activities,  
});
```



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!

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
await client.workflow.start(OrderProcessingWorkflow, {
  args: [order],
  taskQueue: TASK_QUEUE_NAME,
  workflowId: `workflow-order-${order.id}`,
})
```

- **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



Another area to consider is the Workflow ID you specify.

When you start a Workflow Execution you specify a Workflow ID. This must be unique across all running workflow executions in the namespace. We recommend you specify a workflow ID that is meaningful to your business logic. You can use an order number or another identifier that's easier for you to track and look up later.

How Errors Affect Workflow Execution

- **An Activity that throws an error is considered as failed**
 - It may or may not be 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
- **A Workflow that throws an error is also considered as failed if the error is a Temporal failure.**
 - Cancellation, Activity errors that bubble up, or **ApplicationFailure**.
 - **Other errors raised result in Workflows being retried**
 - Failing an Activity is common, but failing a Workflow is considered unusual
 - It is considered a better practice to fix the Workflow



When you're writing Temporal code, you also need to be mindful about how you handle errors.

Activities that throw errors are failed, and a Workflow will retry these activities based on its retry policy. The default policy results in retrying until execution succeeds or is canceled

A Workflow that throws an error is also considered as failed if the error is a Temporal failure. This is a Cancellation, an explicit `ApplicationFailure` error, or an Activity error that bubbles up.

Other errors raised result in the Workflow being retried.

How to Return Errors in Application Code

- You can throw errors as necessary in Activities

```
try {
  const response = await axios.get(url);
  const content = response.data;
  return { translation: content };
} catch (error: any) {
  if (error.response) {
    throw new Error(`HTTP Error ${error.response.status}: ${error.response.data}`);
  }
  else if (error.request) {
    throw new Error(`Request error: ${error.request}`);
  }
  throw new Error('Something else failed during translation.');
```

- Developers are not *required* to use a Temporal-specific API for errors
 - Application errors are automatically converted into a language neutral format



You can throw errors in your activities just like you would in any other TypeScript application. Any error you throw gets converted into a language neutral format.

Logging in Temporal Applications

- The recommended way of logging is via the interface in the TypeScript SDK
- This interface defines four log levels, 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 Interface in Workflows

- Log statements can include any number of key-value pairs

```
import { log } from '@temporalio/workflow';  
  
export async function sayHelloGoodbyeWorkflow(input: string): Promise<string> {  
  log.info('SayHelloGoodbye Workflow Invoked', { name: input.name });  
  
  // other code  
}
```



In Workflows, you import the log function from the workflow package.

You can specify a message as well as a key/value object with additional data.

Using the Logger in Activities

- Accessing and using the Activity logger is similar

```
import * as activity from '@temporalio/activity';

export async function translateTerm(input: GreetingInput): Promise<GreetingOutput> {
  const context = activity.Context.current();
  context.log.info('Translating term:', { LanguageCode: input.languageCode, Term: input.term });
  // other code
}
```



In Activities, you access the log through the activity context. Once you have access to the log function, it works the same way. You specify a message and a key/value object with additional info.

Customizing the Logger

```
import { DefaultLogger, Runtime } from '@temporalio/worker';  
  
const logger = new DefaultLogger('WARN', ({ level, message }) => {  
  console.log(`Custom logger: ${level} - ${message}`);  
});  
Runtime.install({ logger });
```



You can customize the default logger. You can set its default message level and how it logs. In this example, we change the default level and customize how it prints values.

Long-Running Executions

- **Temporal Workflows may have executions that span several years**
 - Activities may also run for long periods of time
 - We'll only focus on long-running Workflows in this course!



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.

Workflow and Activity Executions are async operations

- The following call submits a *Workflow execution request* to the cluster.
- Nothing happens until a Worker with a matching Workflow or Activity Type picks up the task

```
// Use a client to request Workflow execution
const handle = await client.workflow.start(sayHelloGoodbyeWorkflow, {
  args: [input],
  taskQueue: TASK_QUEUE_NAME,
  workflowId: 'translation-workflow-' + nanoid(),
});
```



What you should know is that Workflow and Activity Executions are async operations.

For example, this code submits a workflow execution request to the cluster.

Nothing happens until a Worker with a matching Workflow or Activity Type picks up the task.

Waiting on Workflow Execution Results

- The TypeScript SDK uses Promises to provide access to results from asynchronous executions
- Use `await client.workflow.start` to get a handle once the Temporal Cluster has accepted the receipt
- Use `await handle.result()` to get the result of the Workflow Execution

```
// Start the Workflow Execution
const handle = await client.workflow.start(sayHelloGoodbyeWorkflow, {
  args: [input],
  taskQueue: TASK_QUEUE_NAME,
  workflowId: 'translation-workflow-' + nanoid(),
});

// Get the result of the Workflow Execution
const output = await handle.result();
```



The TypeScript SDK uses Promises to provide access to results from asynchronous executions. When executing a workflow, you first get a handle to the workflow execution, but only once the Temporal Cluster has accepted the receipt of the request.

Use `handle.result()` to get the result of the workflow.

Waiting on Activity Execution Results

- The TypeScript SDK uses Promises to provide access to results from Activity Executions.
- Use **await** to wait for the result:

```
// use await to wait for the result.  
const helloResult = await translateTerm(helloInput);
```



In the TypeScript SDK, An Activity is just an async function that returns a promise. To wait for the result of the Activity Execution, await the result of the call.

Deferring Access to Activity 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

```
// Request execution of multiple Activities: these calls do not block
const promiseA = activityA(inputA);
const promiseB = activityB(inputB);
const promiseC = activityC(inputC);

// This will block until all promises have resolved
const [resultA, resultB, resultC] = await Promise.all([promiseA, promiseB, promiseC]);
```



Sometimes you may want to defer the results. For example, if you need to call unrelated activities or have Activities execute in parallel. To invoke Activity Executions without blocking, just resolve the promises later.

The Activity will still be scheduled and executed when you execute the call.

Review

- **Use objects for input and outputs to Workflows and Activities**
- **Use constants when defining Task Queue names to ensure consistency**
- **Use Workflow IDs that are meaningful**
- **Handle errors appropriately, especially errors from Workflows.**
- **Use logging in your Workflows and Activities**
- **Remember that Workflows and Activities can run for long periods of time**



Use objects for input and outputs to Workflows and Activities

Use constants when defining Task Queue names to ensure consistency

Use Workflow IDs that are meaningful

Handle errors appropriately, especially errors from Workflows.

Use logging in your Workflows and Activities

Remember that Workflows and Activities can run for long periods of time.

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



Timers are used to introduce delays into a Workflow Execution.

Code that awaits the Timer pauses execution for the specified duration

The duration is fixed and may range from seconds to years

Once the time has elapsed, the Timer fires, and execution continues

Use Cases for Timers

- **Execute an Activity multiple times at predefined intervals**
 - 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



Use timers to

Execute an Activity multiple times at predefined intervals

Execute an Activity multiple times at dynamically-calculated intervals

Allow time for offline steps to complete

Pausing Workflow Execution for a Specified Duration

- Use the SDK-provided `sleep` function for this
 - This is an alternative to TypeScript's `sleep` function
 - Use `await` to block until the Timer is fired (or is canceled)

```
// use sleep from temporal's TypeScript SDK
import { sleep } from '@temporalio/workflow';

// This will pause Workflow Execution for 10 seconds
await sleep("10 seconds");
```



You use the `sleep` function in your TypeScript workflows to create a Timer. There's a nice string interface where you can specify the time duration.

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



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

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 `@temporalio/testing` package provides what you need.
 - Use it with a suitable testing library. We recommend `mocha`.
 - It downloads a test server.
 - It provides `TestWorkflowEnvironment`, which you use to connect the Client and Worker to the test server and interact with the test server.
 - You can “skip time” so you can test long-running Workflows without waiting.



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

The `@temporalio/testing` package provides what you need to test Temporal applications.

The `TestWorkflowEnvironment` type provides a runtime environment used to test a Workflow. You'll use this to register your Workflow Type and access information about the Workflow Execution under test, such as whether it completed successfully and the result or error it returned.

Test setup with Mocha

- Add mocha and @temporalio/testing to your package.json file, along with types:

```
"devDependencies": {  
  "@temporalio/testing": "~1.8.0",  
  "@types/mocha": "8.x",  
  "mocha": "8.x",  
  ...  
}
```



Add mocha and @temporalio/testing to your package.json file, along with types as development dependencies:

Testing Activities - the estimateAge Activity

```
import axios from 'axios';
import { URLSearchParams } from 'url';

export async function estimateAge(name: string): Promise<number> {
  const base = 'https://api.agify.io/?';
  const url = base + new URLSearchParams({ name });

  const response = await axios.get(url);
  const responseBody = response.data;

  interface EstimatorResponse {
    age: number;
    count: number;
    name: string;
  }

  const parsedResponse: EstimatorResponse = responseBody as EstimatorResponse;
  return parsedResponse.age;
}
```



Let's start by looking at how to test Activities.

To do that, we'll use this estimateAge activity as an example. This Activity accepts a name and connects to a web service to guess someone's age based on that name. It's not a complex Activity, but it does call an external service, so we should write tests for it.

The Activity Test

- The `@temporalio/testing` package provides `MockActivityEnvironment`, so you can test Activities in isolation

```
import { MockActivityEnvironment } from '@temporalio/testing';
import { describe, it } from 'mocha';
import * as activities from '../activities';
import assert from 'assert';

describe('estimateAge activity', async () => {

  it('runs estimateAgeWorkflow with activity call', async () => {
    const env = new MockActivityEnvironment();
    const res = await env.run(activities.estimateAge, 'Betty');
    assert.equal(res, 76);
  });
});
```



The `@temporalio/testing` package provides `MockActivityEnvironment`, so you can test Activities in isolation.

You import `MockActivityEnvironment`, along with your activities, and the various pieces from Mocha, and then use the mock activity environment to run the activity. You can use regular assertions to test the result.

Testing Workflows - estimateAgeWorkflow

```
import { proxyActivities } from '@temporalio/workflow';
import type * as activities from './activities';

const { estimateAge } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function estimateAgeWorkflow(name: string): Promise<string> {
  const age = await estimateAge(name);
  return `${name} has an estimated age of ${age}`;
}
```



Here's the workflow for estimating the age.

It takes in the name, executes the activity, and retrieves the result.

The Workflow Test (1)

```
import { TestWorkflowEnvironment } from '@temporalio/testing';
import { after, before, it } from 'mocha';
import { Worker } from '@temporalio/worker';
import { estimateAgeWorkflow } from '../workflows';
import * as activities from '../activities';
import assert from 'assert';

describe('estimateAge workflow', async () => {
  let testEnv: TestWorkflowEnvironment;

  before(async () => {
    testEnv = await TestWorkflowEnvironment.createTimeSkipping();
  });

  after(async () => {
    await testEnv?.teardown();
  });

  // tests
});
```



Testing a Workflow is a little more involved. You need to import the TestWorkflow Environment and the pieces from Mocha that you need, but you also need to import the Worker, your Workflow, and the activities.

The Workflow Test (2)

```
import { TestWorkflowEnvironment } from '@temporalio/testing';
import { after, before, it } from 'mocha';
import { Worker } from '@temporalio/worker';
import { estimateAgeWorkflow } from '../workflows';
import * as activities from '../activities';
import assert from 'assert';

describe('estimateAge workflow', async () => {
  let testEnv: TestWorkflowEnvironment;

  before(async () => {
    testEnv = await TestWorkflowEnvironment.createTimeSkipping();
  });

  after(async () => {
    await testEnv?.teardown();
  });

  // tests
});
```



The TestWorkflow environment should be shared across tests, so you'll set it up before all of your tests, and tear it down after your tests are done.

This is boilerplate you can use for most of your Workflow test cases.

The Workflow Test (3)

```
describe('estimateAge workflow', async () => {
  // before, after omitted

  it('runs estimateAgeWorkflow with activity call', async () => {
    const { client, nativeConnection } = testEnv;
    const worker = await Worker.create({
      connection: nativeConnection,
      taskQueue: 'test',
      workflowsPath: require.resolve('../workflows'),
      activities,
    });

    const result = await worker.runUntil(
      client.workflow.execute(estimateAgeWorkflow, {
        args: ['Betty'],
        workflowId: 'test',
        taskQueue: 'test',
      })
    );

    assert.equal(result, 'Betty has an estimated age of 76');
  });
});
```



To test the Workflow itself, you set up a Worker and connect it to the test environment.

[advance]

The Workflow Test (4)

```
describe('estimateAge workflow', async () => {
  // before, after omitted

  it('runs estimateAgeWorkflow with activity call', async () => {
    const { client, nativeConnection } = testEnv;
    const worker = await Worker.create({
      connection: nativeConnection,
      taskQueue: 'test',
      workflowsPath: require.resolve('../workflows'),
      activities,
    });

    const result = await worker.runUntil(
      client.workflow.execute(estimateAgeWorkflow, {
        args: ['Betty'],
        workflowId: 'test',
        taskQueue: 'test',
      })
    );

    assert.equal(result, 'Betty has an estimated age of 76');
  });
});
```



You then use that worker to execute the Workflow.

[advance]

The Workflow Test (5)

```
describe('estimateAge workflow', async () => {
  // before, after omitted

  it('runs estimateAgeWorkflow with activity call', async () => {
    const { client, nativeConnection } = testEnv;
    const worker = await Worker.create({
      connection: nativeConnection,
      taskQueue: 'test',
      workflowsPath: require.resolve('../workflows'),
      activities,
    });

    const result = await worker.runUntil(
      client.workflow.execute(estimateAgeWorkflow, {
        args: ['Betty'],
        workflowId: 'test',
        taskQueue: 'test',
      })
    );

    assert.equal(result, 'Betty has an estimated age of 76');
  });
});
```



You then assert the result like you would any other test.

[advance]

Mocking Activities in Workflow Tests

- **The Workflow test we wrote is an Integration Test**
 - It invokes an Activity
 - That Activity calls a real web service
 - It's tightly coupled to both
- **Unit test Workflows by mocking Activities**
 - Define new replacement Activities
 - Use the **sinon** library to create mocks



The Workflow test we wrote is an Integration Test! It's tightly coupled to the activity which is coupled to the service it's calling.

We can fix this by mocking out the activity calls in the workflow.

Mock the Activity Directly

```
it('runs estimateAgeWorkflow with mocked activity call', async () => {
  const { client, nativeConnection } = testEnv;
  const worker = await Worker.create({
    connection: nativeConnection,
    taskQueue: 'test',
    workflowsPath: require.resolve('../workflows'),
    activities: {
      estimateAge: async () => 76,
    },
  });

  const result = await worker.runUntil(
    client.workflow.execute(estimateAgeWorkflow, {
      args: ['Betty'],
      workflowId: 'test',
      taskQueue: 'test',
    })
  );

  assert.equal(result, 'Betty has an estimated age of 76');
});
```



When you assign the activities to the Worker, you can define a new Activity!

Mock the Activity with sinon

```
it('runs estimateAgeWorkflow with mocked activity call', async () => {
  const { client, nativeConnection } = testEnv;

  const estimateAgeMock = sinon.stub();
  estimateAgeMock.withArgs("Betty").resolves(76);

  const worker = await Worker.create({
    connection: nativeConnection,
    taskQueue: 'test',
    workflowsPath: require.resolve('../workflows'),
    activities: { estimateAge: estimateAgeMock },
  });

  const result = await worker.runUntil(
    client.workflow.execute(estimateAgeWorkflow, {
      args: ['Betty'],
      workflowId: 'test',
      taskQueue: 'test',
    })
  );

  assert.equal(result, 'Betty has an estimated age of 76');
});
```



Alternatively, you can use the sinon library to create mocks that resolve the way you need them to resolve. This is helpful in situations where a workflow might invoke the same activity with different inputs and outputs.

Running Tests

```
$ mocha \  
  --require ts-node/register \  
  --require source-map-support/register \  
  src/mocha/*.test.ts
```



To run the tests, you use the mocha command and require a couple of libraries and point the test runner to the appropriate folder containing the tests.

Running Tests

- Add a test script command to package.json:

```
"scripts": {  
  ...  
  "test": "mocha --require ts-node/register --require source-map-support/register src/mocha/*.test.ts"  
  ...  
},
```

- Run the command

```
$ npm test
```



Of course, that's too much typing. So you should add that command to the scripts section of the package.json file. That way you can run

```
npm test
```

to run your tests.

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



Review

- **Temporal's TypeScript SDK provides support for testing Workflows and Activities with the Mocha library**
- **You can test Activities in isolation**
- **You can test Workflows quickly, even if they have Timers**
- **You can mock Activities in Workflow tests, either directly or by using Sinon**



Temporal's TypeScript SDK provides support for testing Workflows and Activities with the Mocha library.

You can test Activities in isolation

You can test Workflows quickly, even if they have Timers

You can mock Activities in Workflow tests, either directly or by using Sinon.

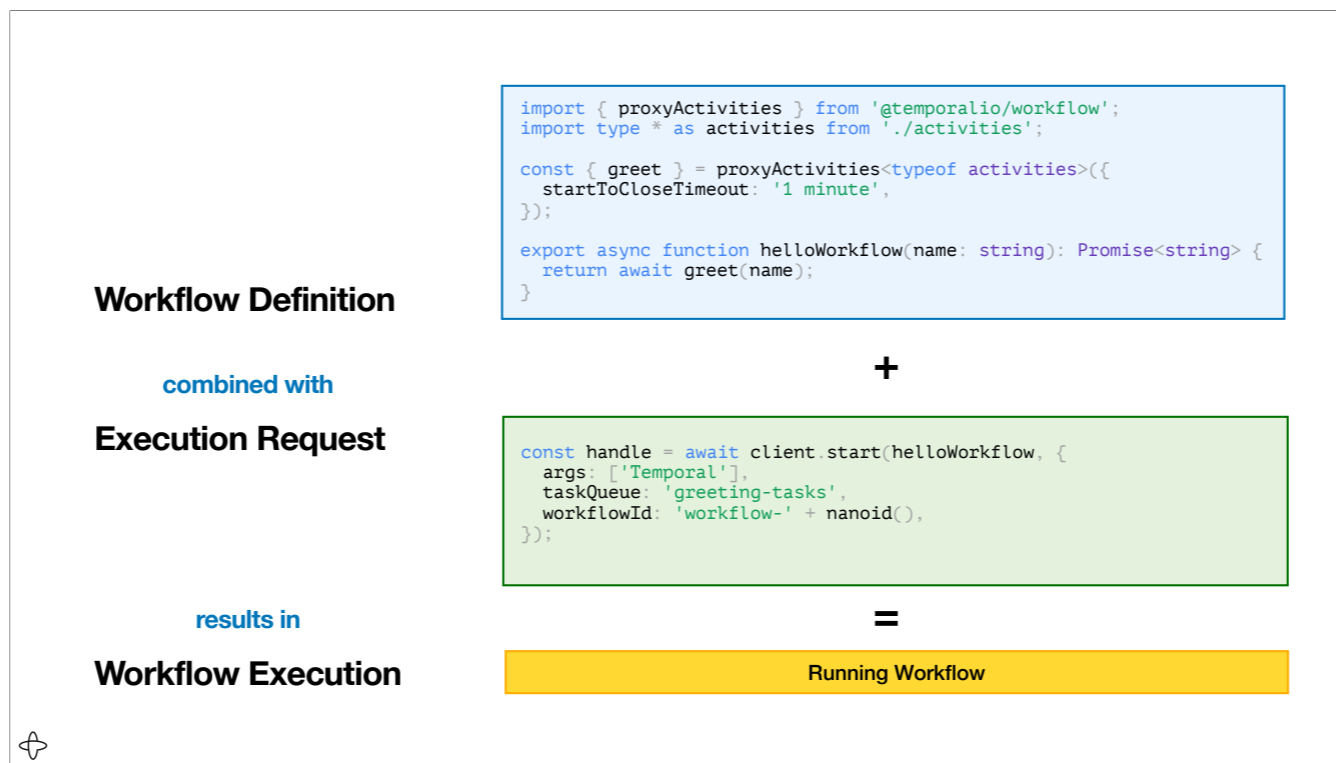
Temporal 102

- 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





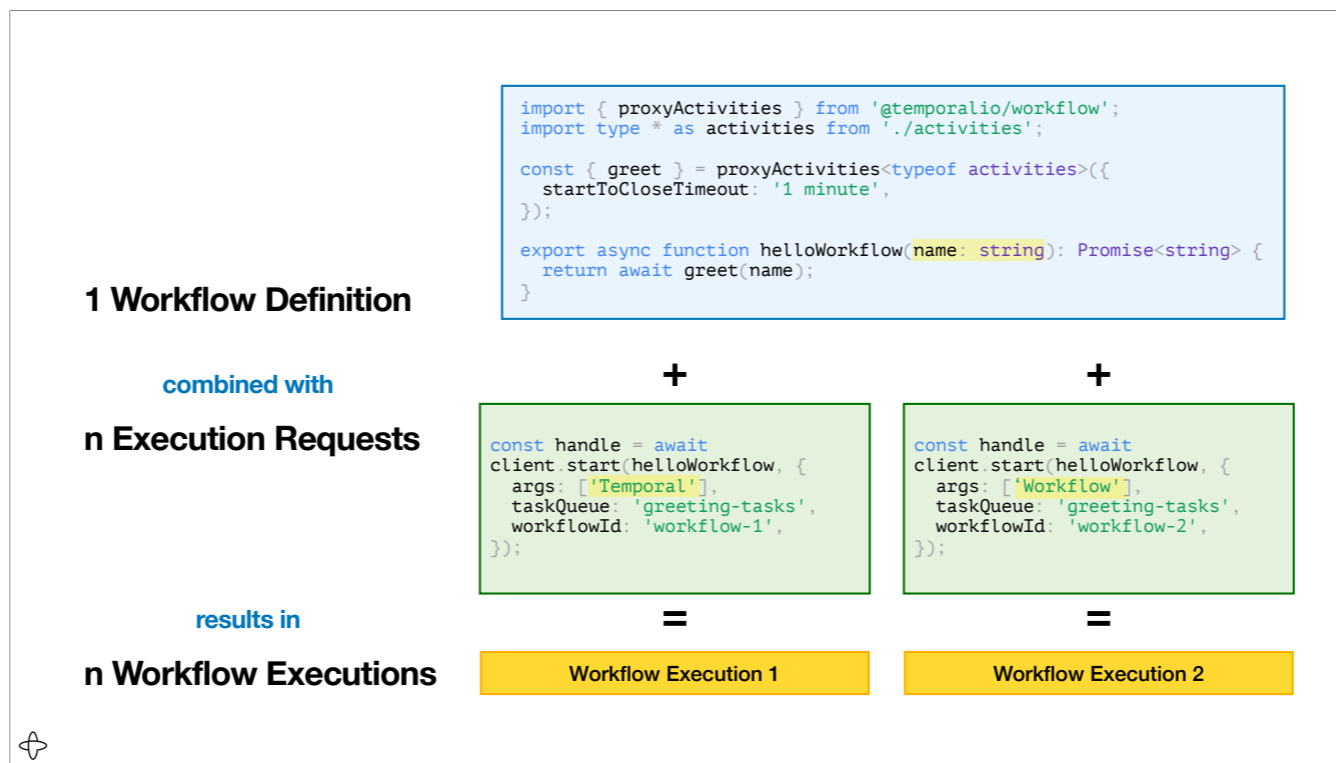
In Temporal, the code that defines your main business logic is implemented in a function referred to as a Workflow Definition. As with any other code you write, it doesn't actually do anything until you execute it. You do this by using a Client to initiate an execution request,

[advance]

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.

[advance]

In Temporal, we refer to this as a Workflow Execution.

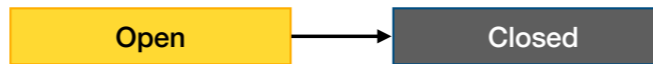


A single Workflow Definition can be executed any number of times. Each results in a new Workflow Execution. For example, you might run the same Workflow Definition each morning to generate some type of daily report.

[advance]

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

Every Workflow Execution has a unique RunID



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.

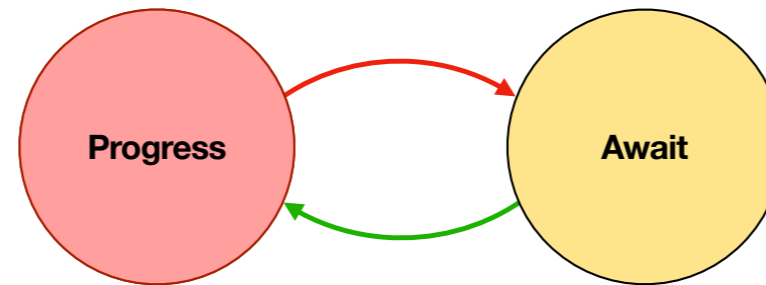
The state of a Workflow Execution can, and eventually will, change from open to closed, but this is a one-way transition. Once a Workflow Execution enters the closed state, it remains there.

Although you may run the same Workflow Definition again, even using the same input, this will result in a new Workflow Execution.

[advance]

Workflow Executions are uniquely identified by a value, called a Run ID, automatically generated when it's launched. Therefore, each new Workflow Execution will have a different Run ID than any previous ones.

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

[advance]

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.

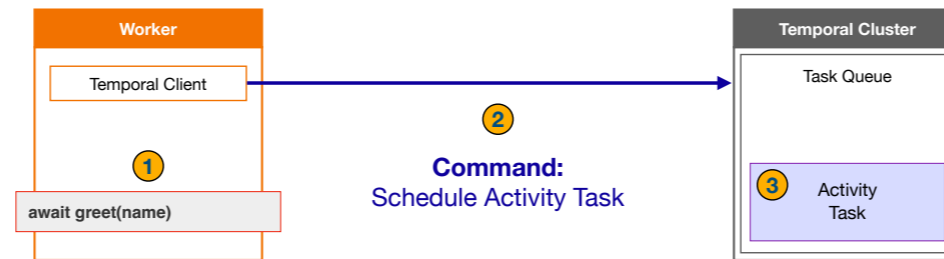
This cycle of progress and waiting continues throughout the Workflow Execution, only stopping when it enters the closed state.

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.

Review of Commands



- 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.

--

```

const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: Order): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  const bill = {
    customerID: order.customer.customerID,
    orderNumber: order.orderNumber,
    amount: totalPrice,
    description: 'Pizza',
  };

  const confirmation = await sendBill(bill);

  return(confirmation);
}

```

Basic Temporal Workflow Definition

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

So let's walk through some code and explore how the code causes the Worker to send commands to the cluster.

Here is code for a basic Temporal Workflow Definition. 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 and am using a string as a return value rather than an object, 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 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.


```

const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: Order): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  const bill = {
    customerID: order.customer.customerID,
    orderNumber: order.orderNumber,
    amount: totalPrice,
    description: 'Pizza',
  };

  const confirmation = await 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]

```

const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: PizzaOrder): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  const bill = {
    customerID: order.customer.customerID,
    orderNumber: order.orderNumber,
    amount: totalPrice,
    description: 'Pizza',
  };

  const confirmation = await sendBill(bill);

  return(confirmation);
}

```

Basic Temporal Workflow Definition

- Other steps *do* involve interaction with the cluster
- Examples include
 - Executing an Activity
 - Setting a Timer
 - Returning an error from the Workflow
 - 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.

[advance]

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.

```
const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: PizzaOrder): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  const bill = {
    customerID: order.customer.customerID,
    orderNumber: order.orderNumber,
    amount: totalPrice,
    description: 'Pizza',
  };

  const confirmation = await 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.

```
const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: PizzaOrder): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  const bill = {
    customerID: order.customer.customerID,
    orderNumber: order.orderNumber,
    amount: totalPrice,
    description: 'Pizza',
  };

  const confirmation = await 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.



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.

```
const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: PizzaOrder): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  const bill = {
    customerID: order.customer.customerID,
    orderNumber: order.orderNumber,
    amount: totalPrice,
    description: 'Pizza',
  };

  const confirmation = await sendBill(bill);

  return(confirmation);
}
```

Afterwards, the Worker continues executing the Workflow code. The next line, highlighted here, evaluates 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.

```
const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: PizzaOrder): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

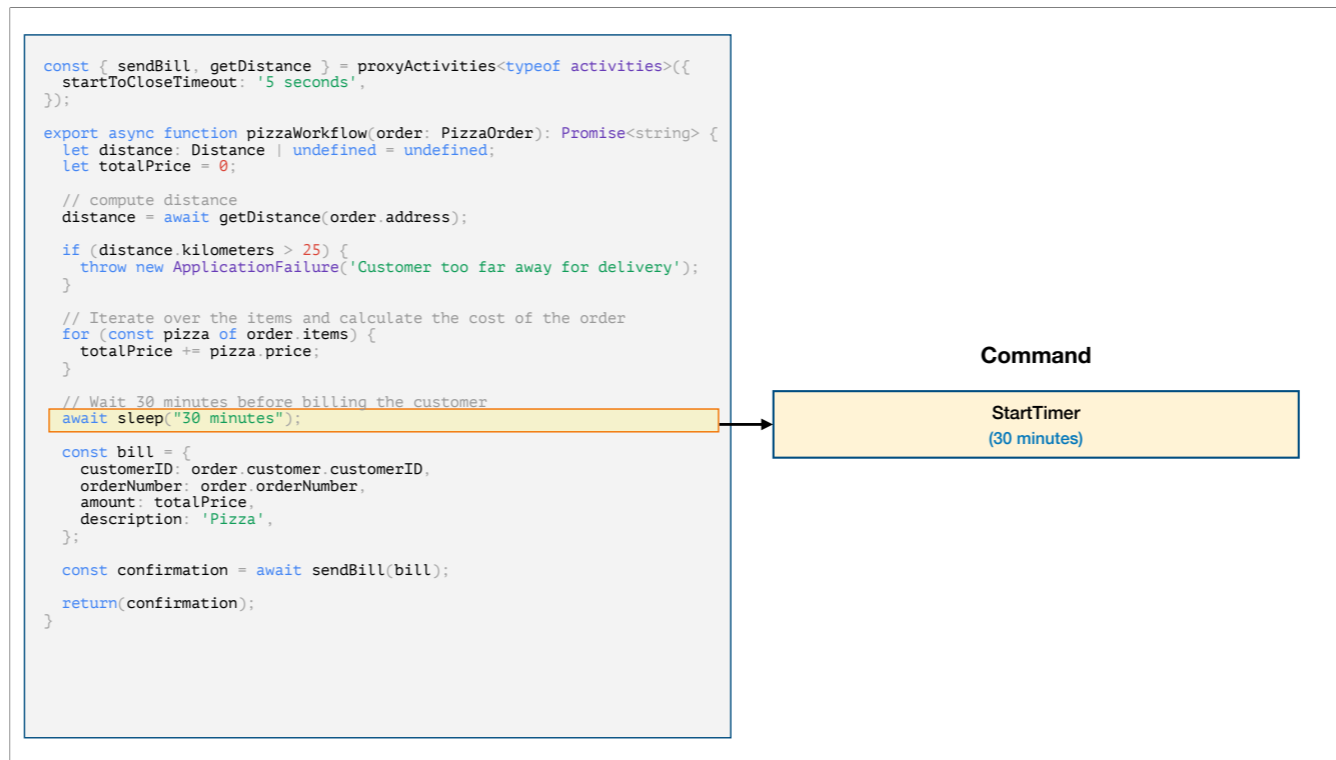
  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  const bill = {
    customerID: order.customer.customerID,
    orderNumber: order.orderNumber,
    amount: totalPrice,
    description: 'Pizza',
  };

  const confirmation = await 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.



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.


```
const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: PizzaOrder): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

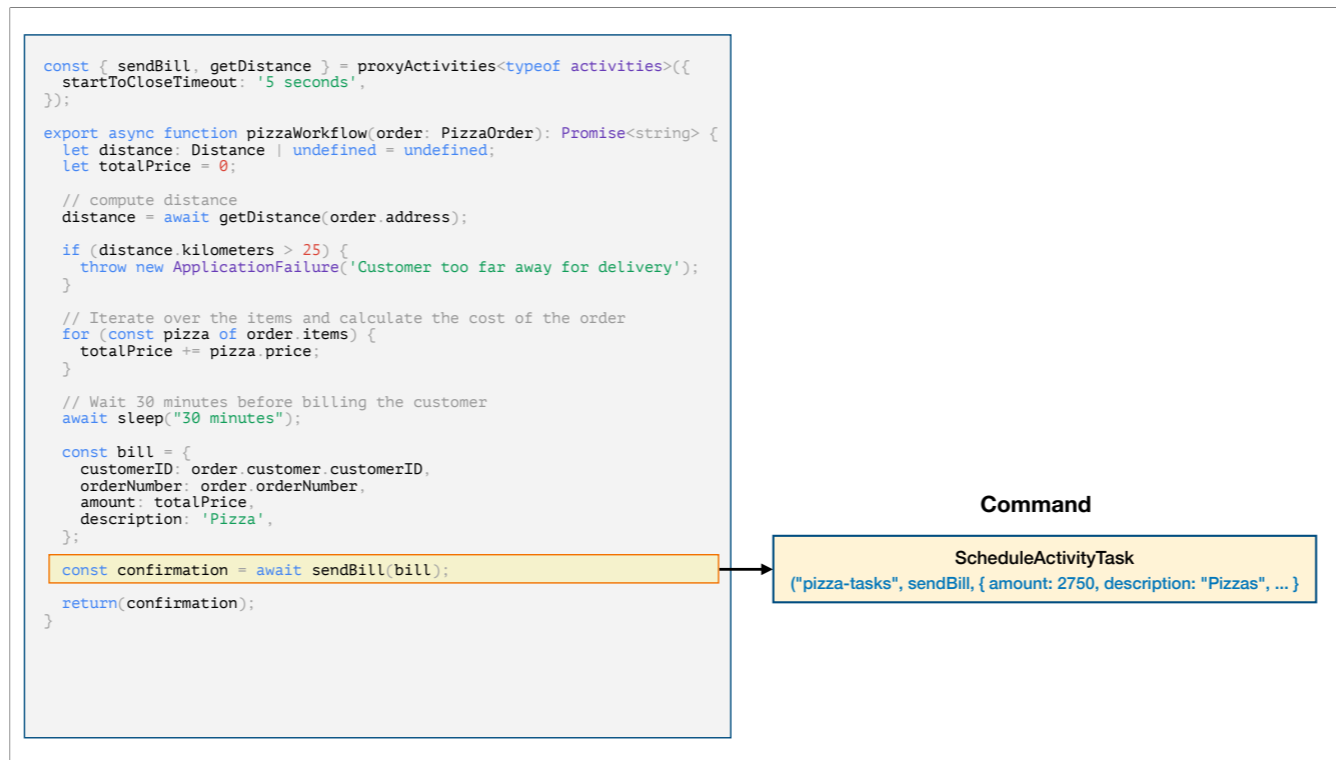
  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  const bill = {
    customerID: order.customer.customerID,
    orderNumber: order.orderNumber,
    amount: totalPrice,
    description: 'Pizza',
  };

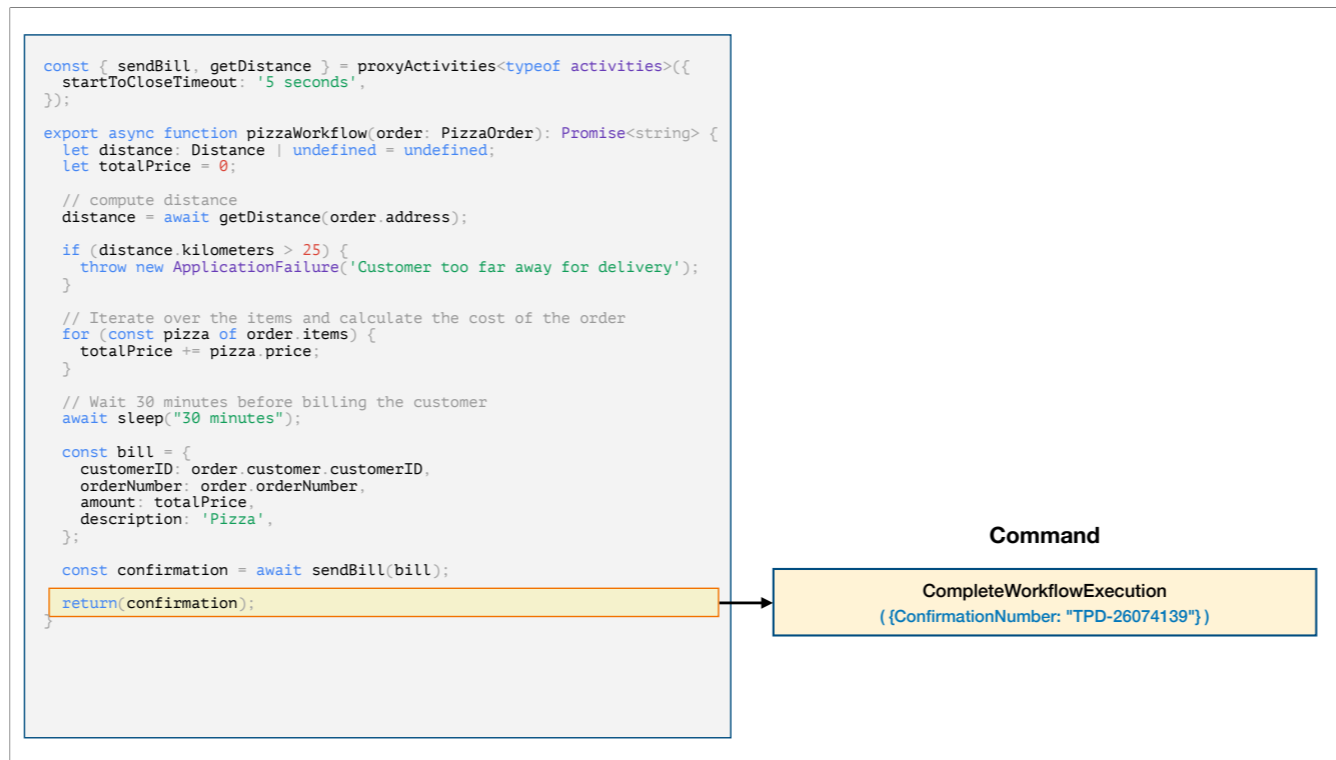
  const confirmation = await sendBill(bill);

  return(confirmation);
}
```

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



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



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

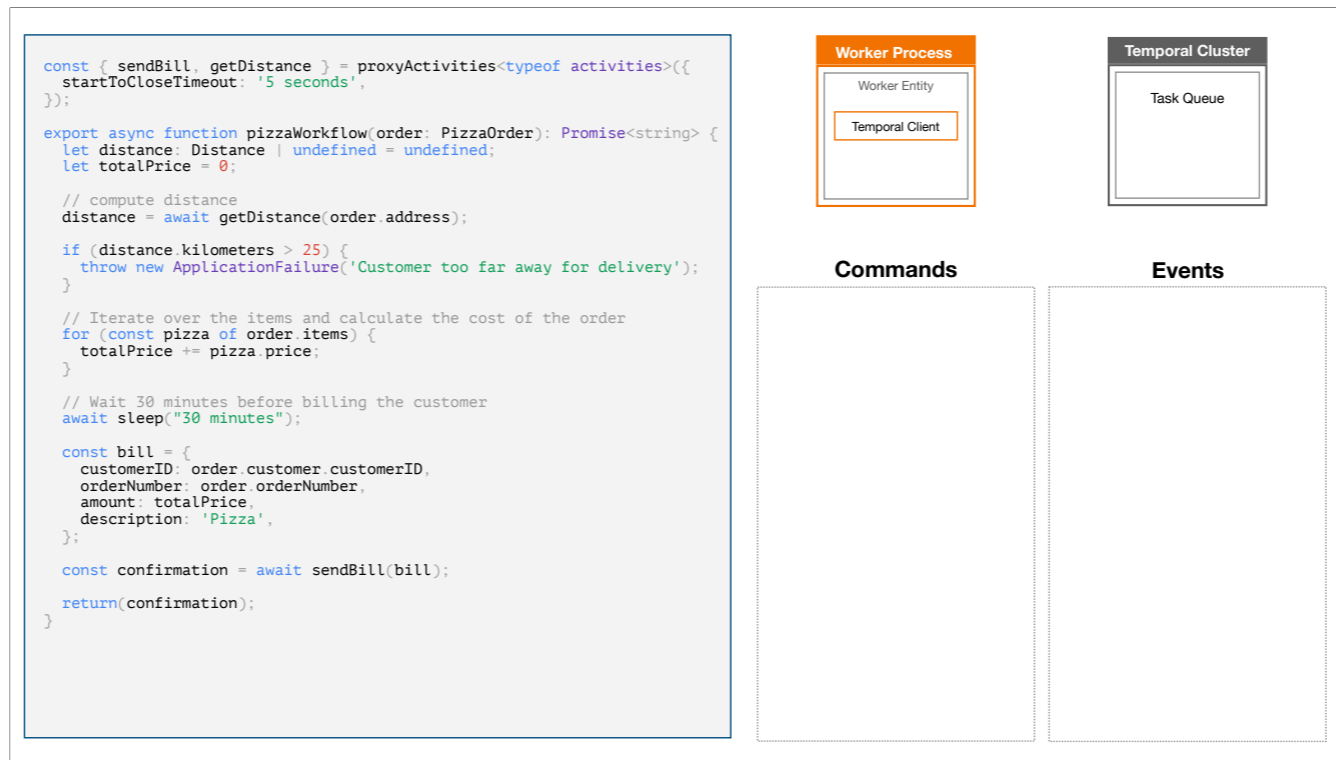
ActivityTaskScheduled contains the Activity Type and input parameters

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.



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,

[advance]

while the upper-right will illustrate the interaction between the Worker

[advance]

and the Temporal Cluster.

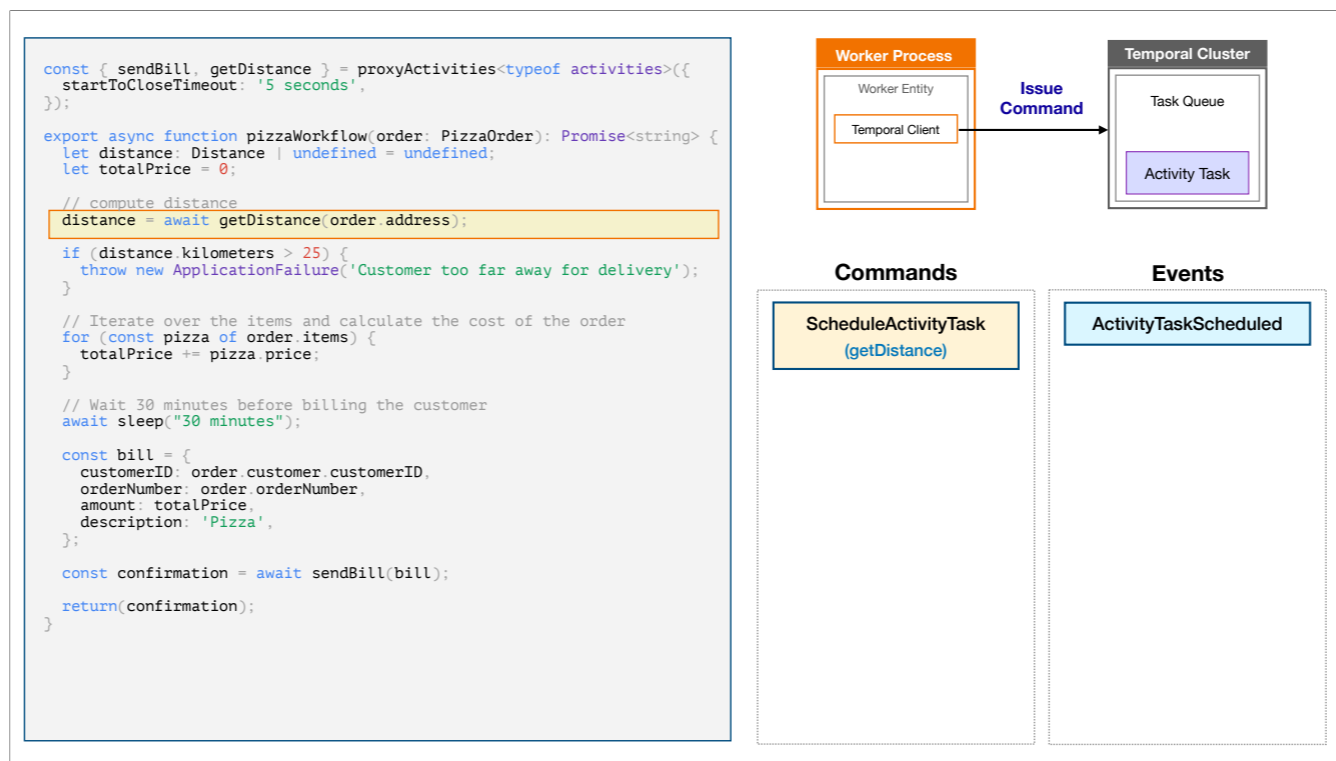
[advance]

Below that, I show a running list of the Commands issued,

[advance]

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.

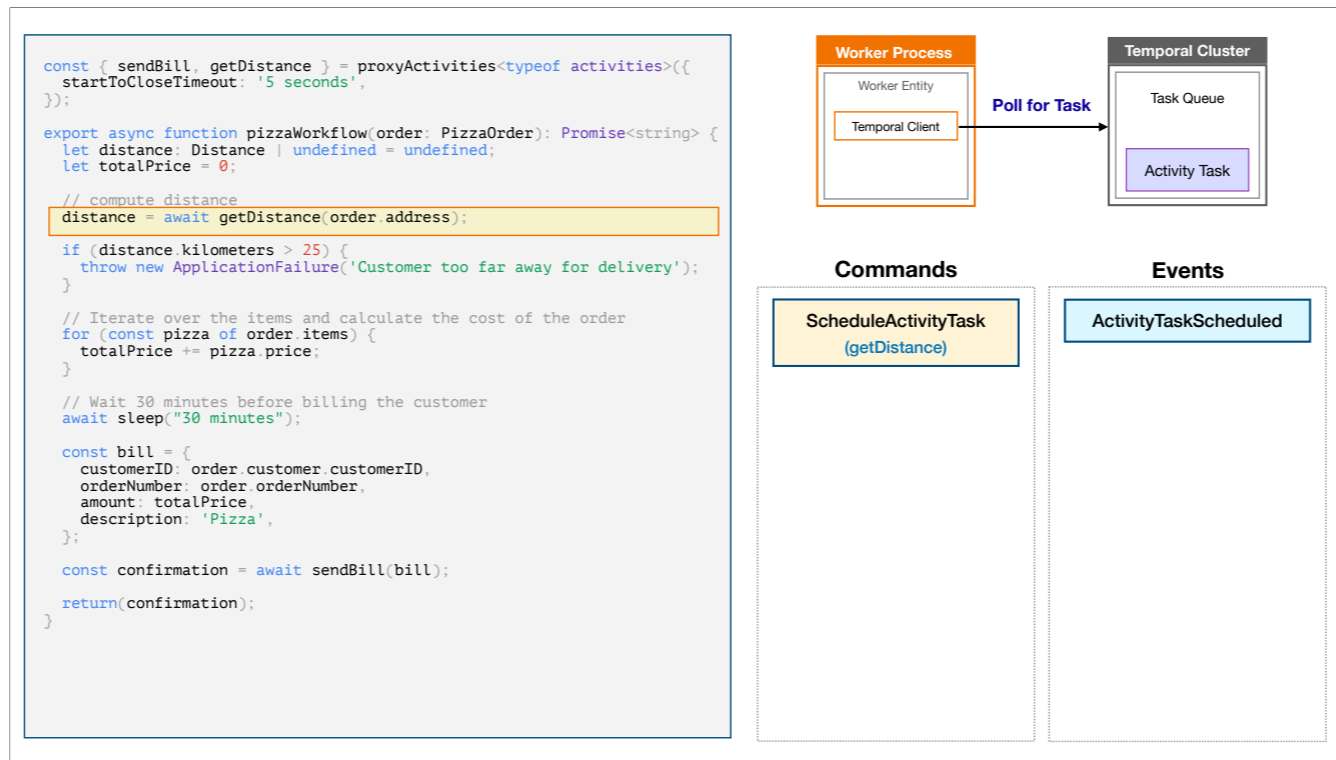


The call to the `getDistance` activity is the first thing in the Workflow that causes a command to be issued.

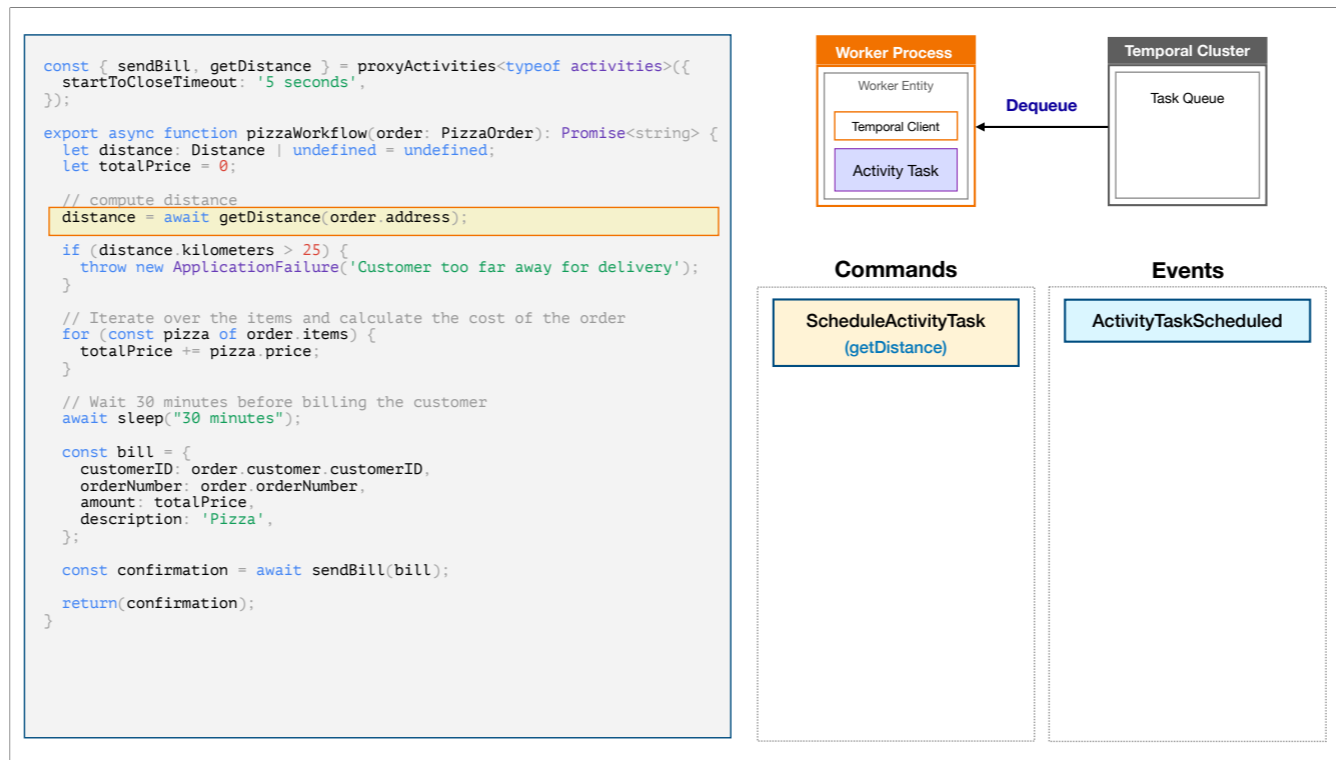
In response to this Command, the Temporal Cluster creates an Activity Task, adds it to the Task Queue, and appends the `ActivityTaskScheduled` Event to this Workflow Execution's history.

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

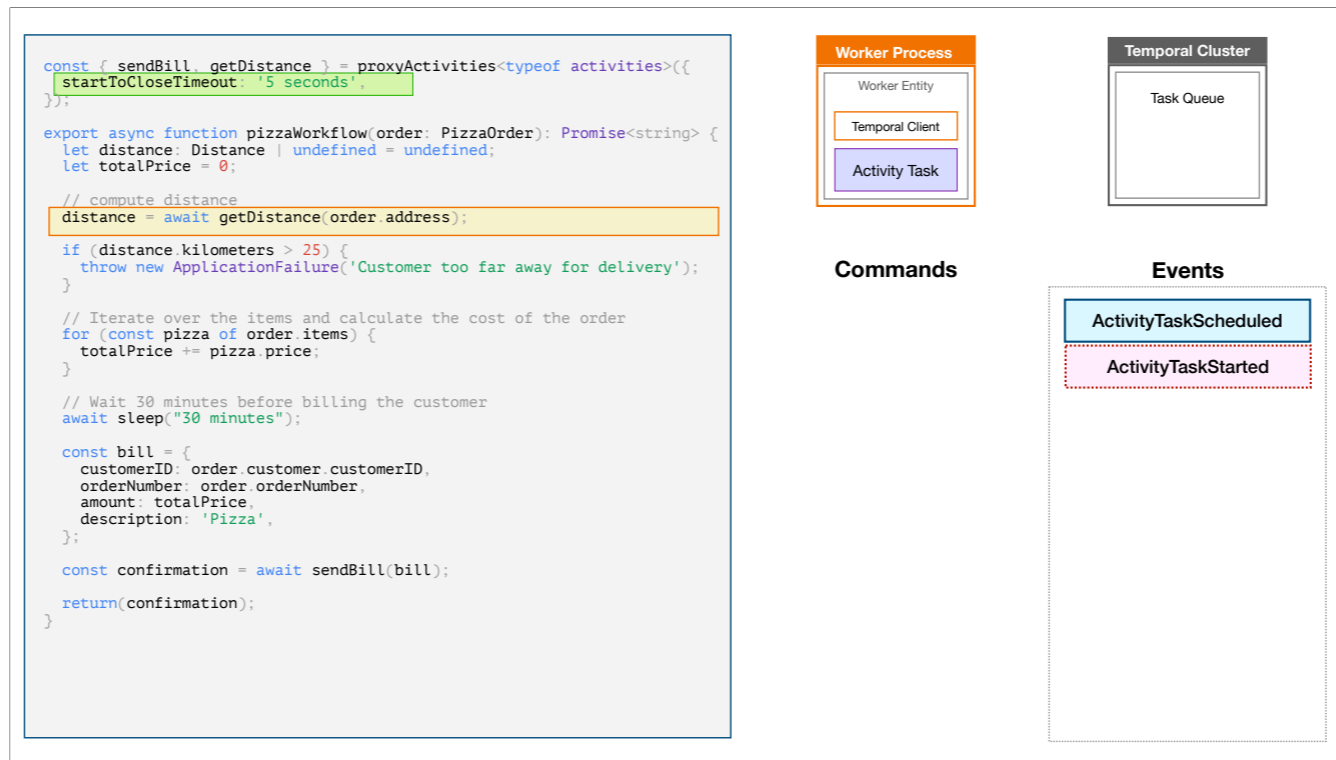
By the way, the Event History is durably persisted to the database used by the Temporal Cluster, so it will survive even if the Temporal Cluster itself crashes.



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.

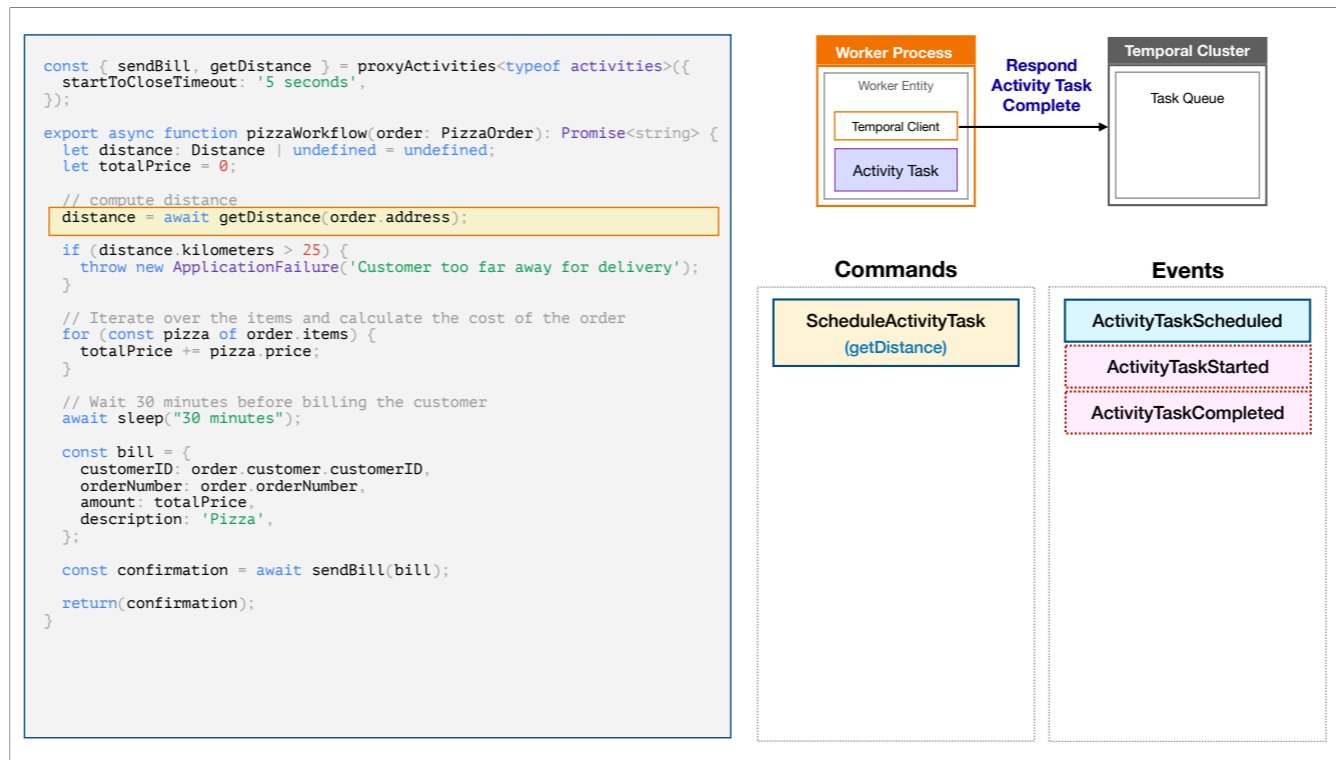


After the Temporal Cluster matches a Worker that's polling for a Task with a Task that's queued, the Worker will then begin executing the code needed to complete that Task.

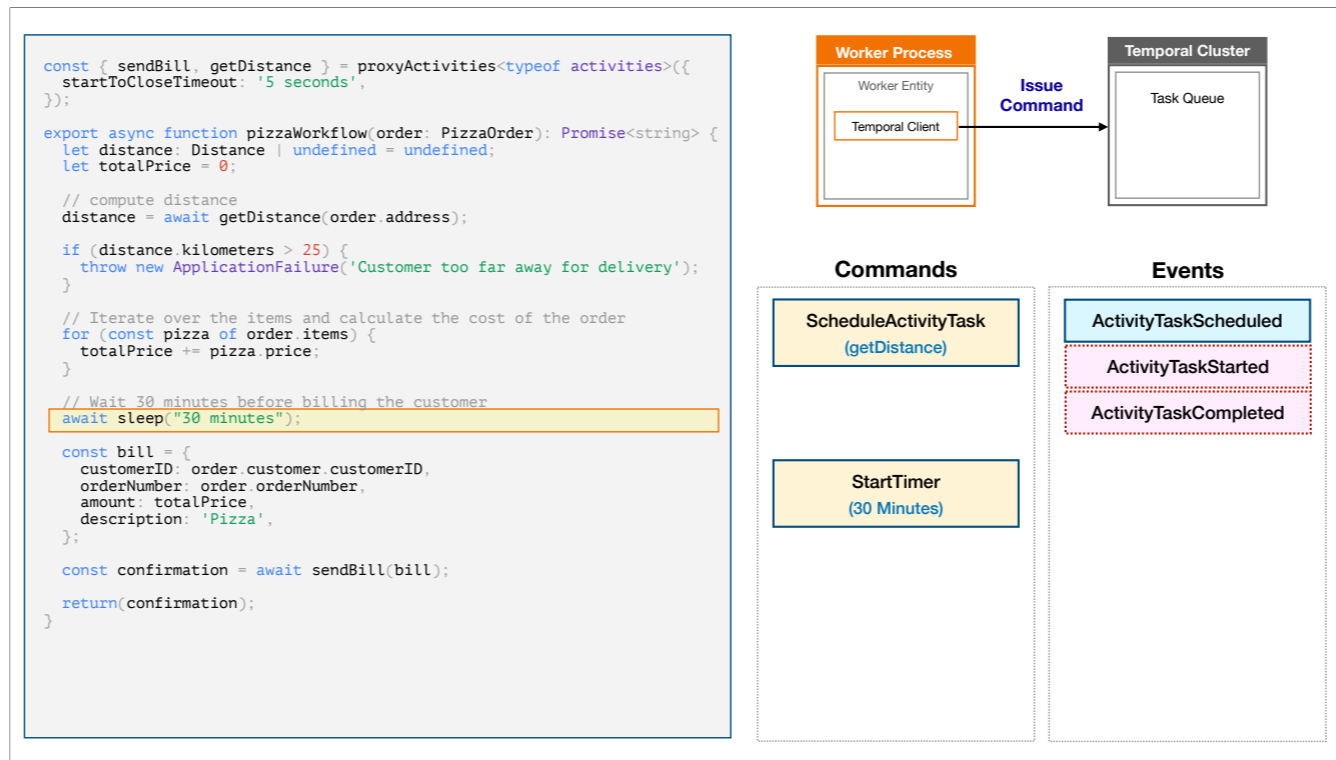


The Temporal Cluster logs another Event in response to the Worker accepting the Task: `ActivityTaskStarted`. I used a different color and border style for the rectangle depicting this event to indicate that it's an indirect result of the Command.

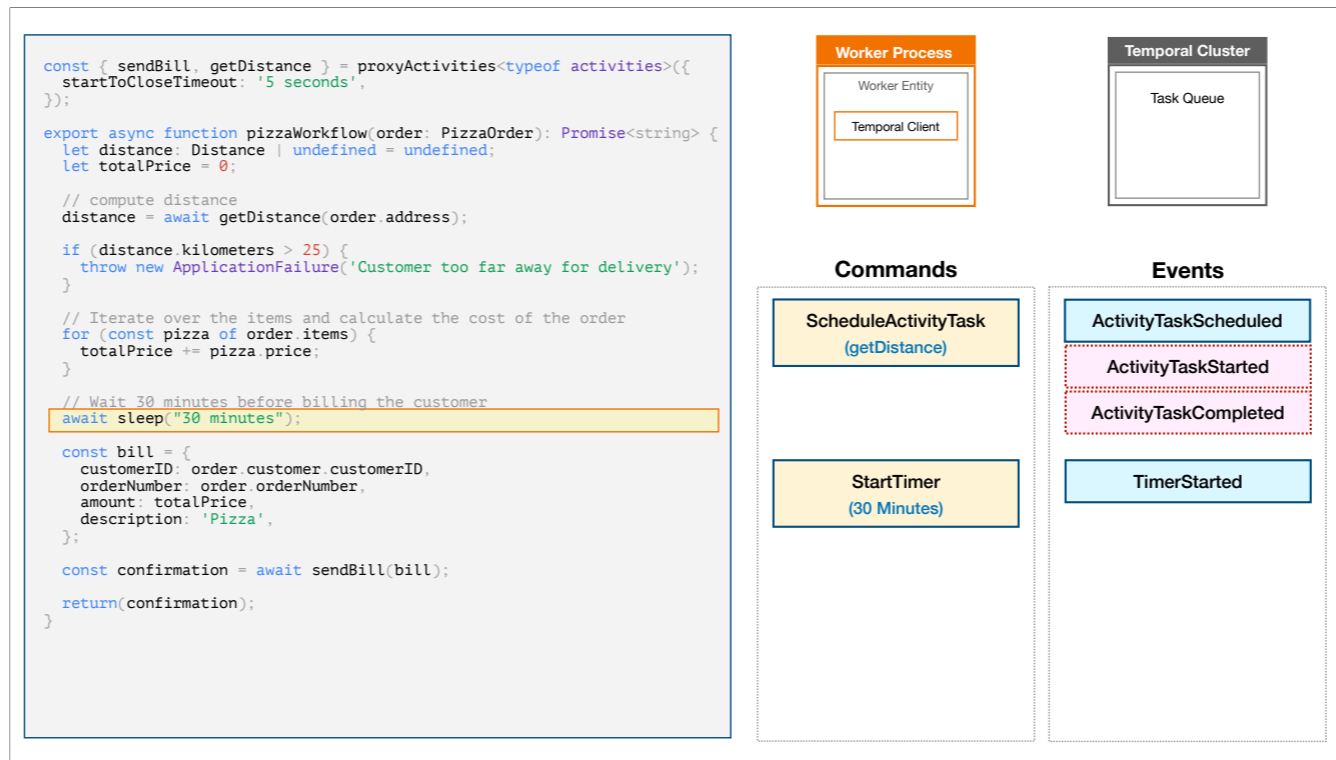
Temporal is designed to ensure that a Task is only ever given to a single Worker, although it will reschedule the Task if this Worker fails to complete it within the time constraints you've specified. In this case, the `ActivityOptions` at the top of the Workflow Definition sets a Start-to-Close Timeout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



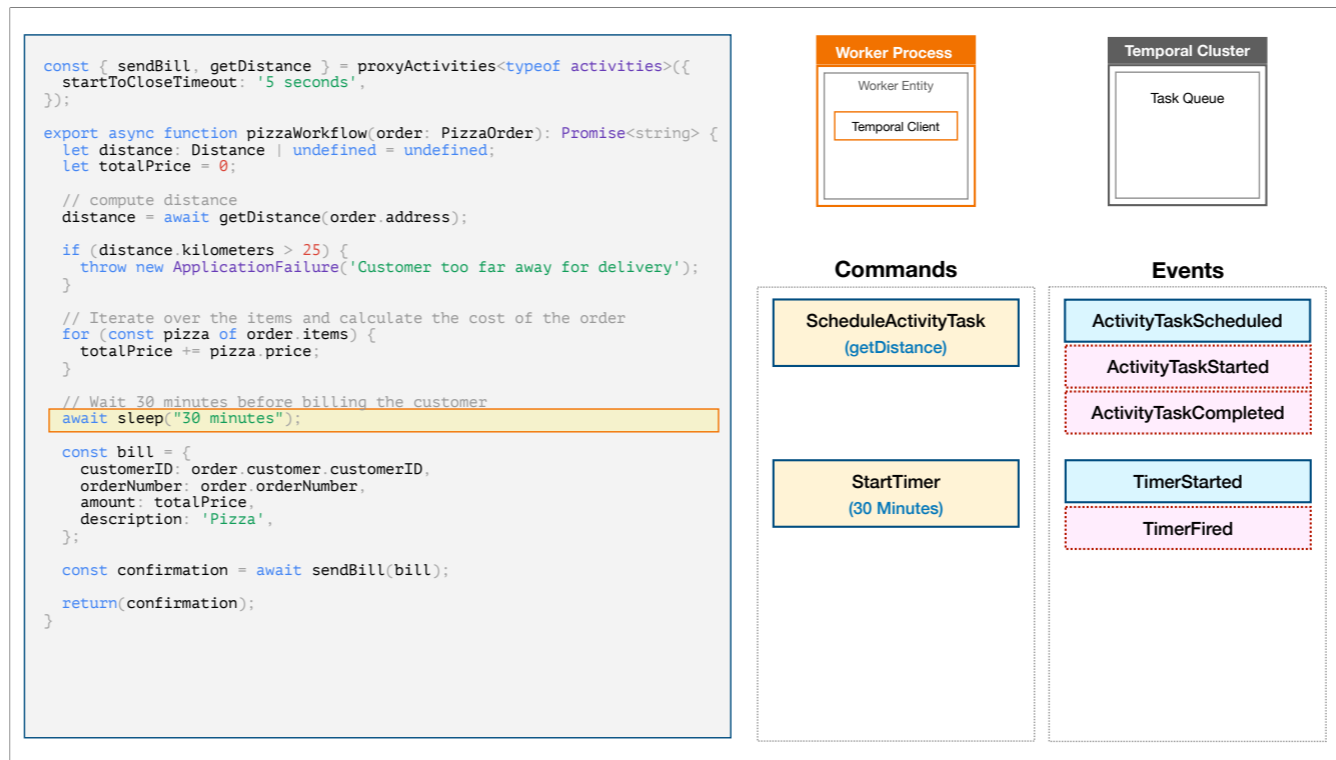
The Worker executes the code within the Activity Definition, and when that function returns a result, the Worker sends a message to the Temporal Cluster, notifying it that the Task is complete. To be clear, this is just a notification, not a Command, because it's not requesting the Temporal Cluster to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



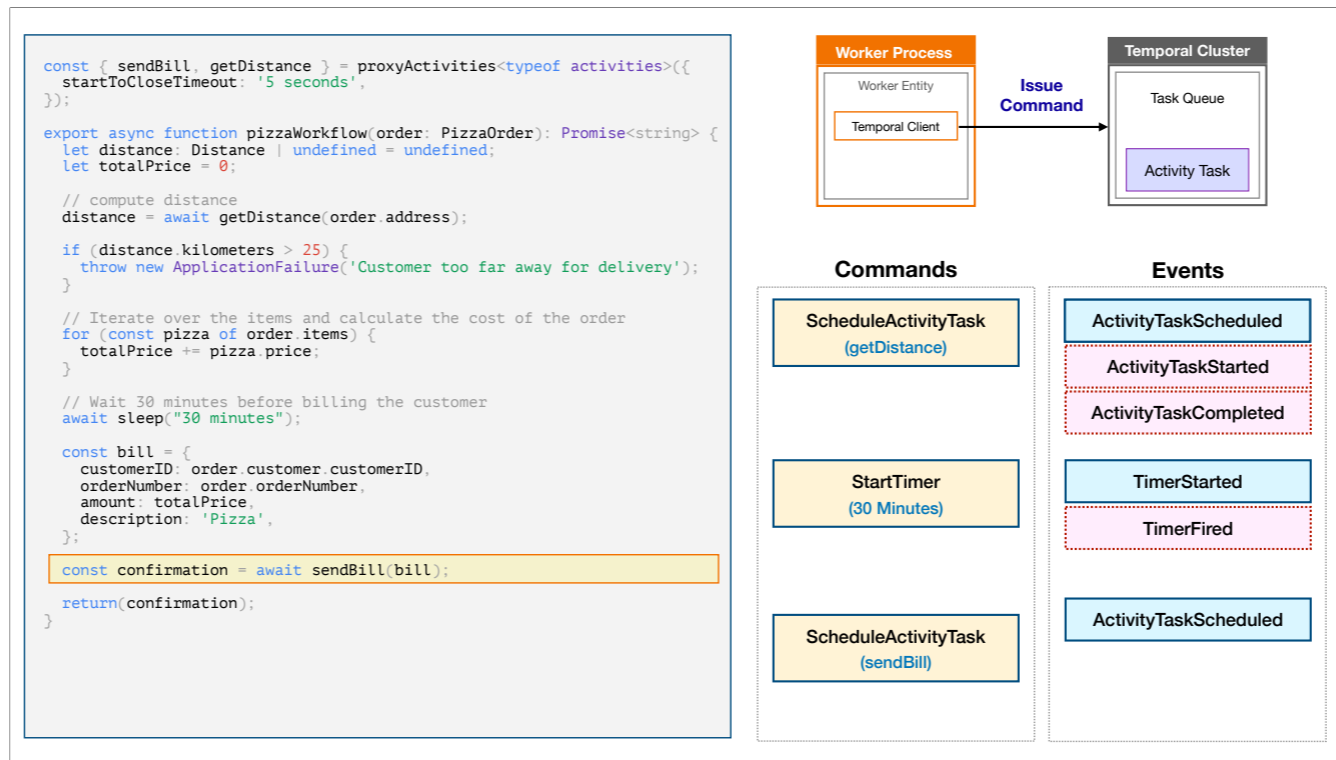
The next statement that results in a Command is the call to `sleep`, which issues a `StartTimer` command.



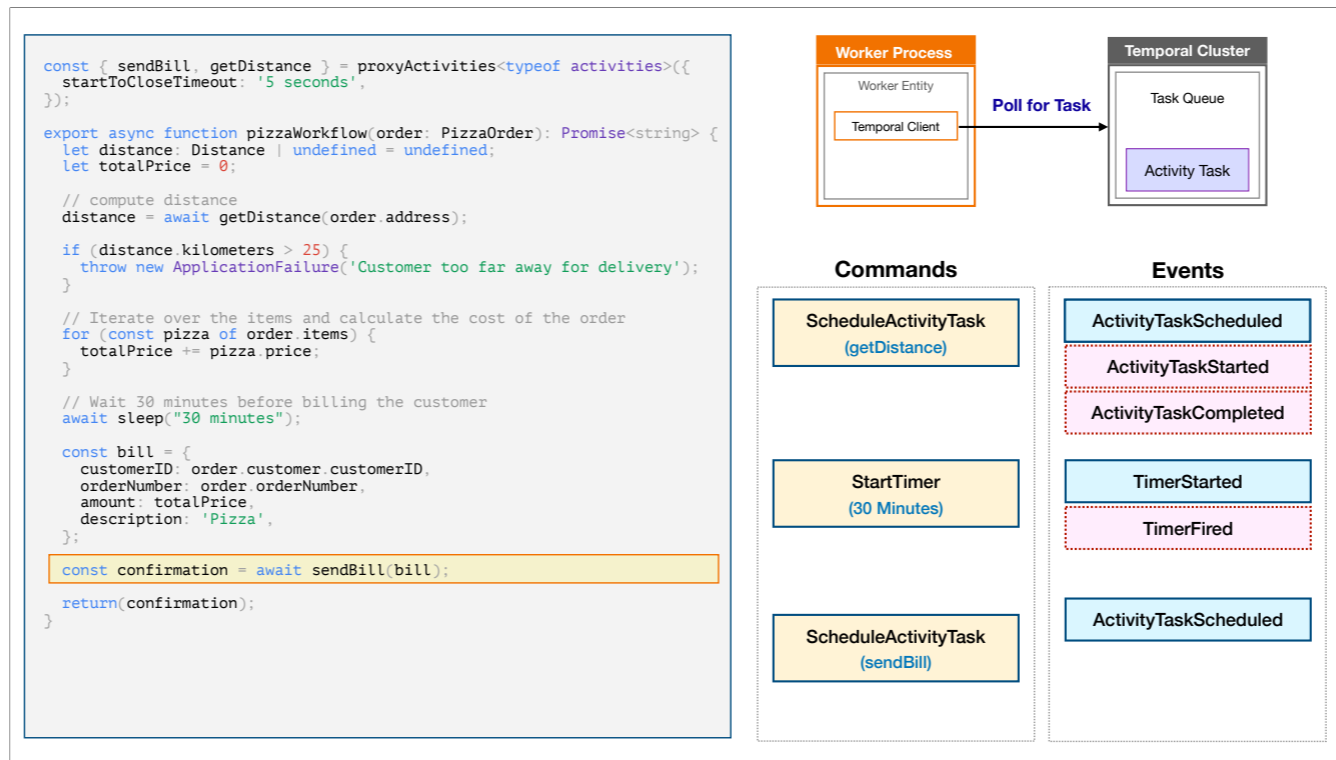
The Temporal Cluster responds by starting a Timer for 30 minutes and logging a `TimerStarted` Event to the history.



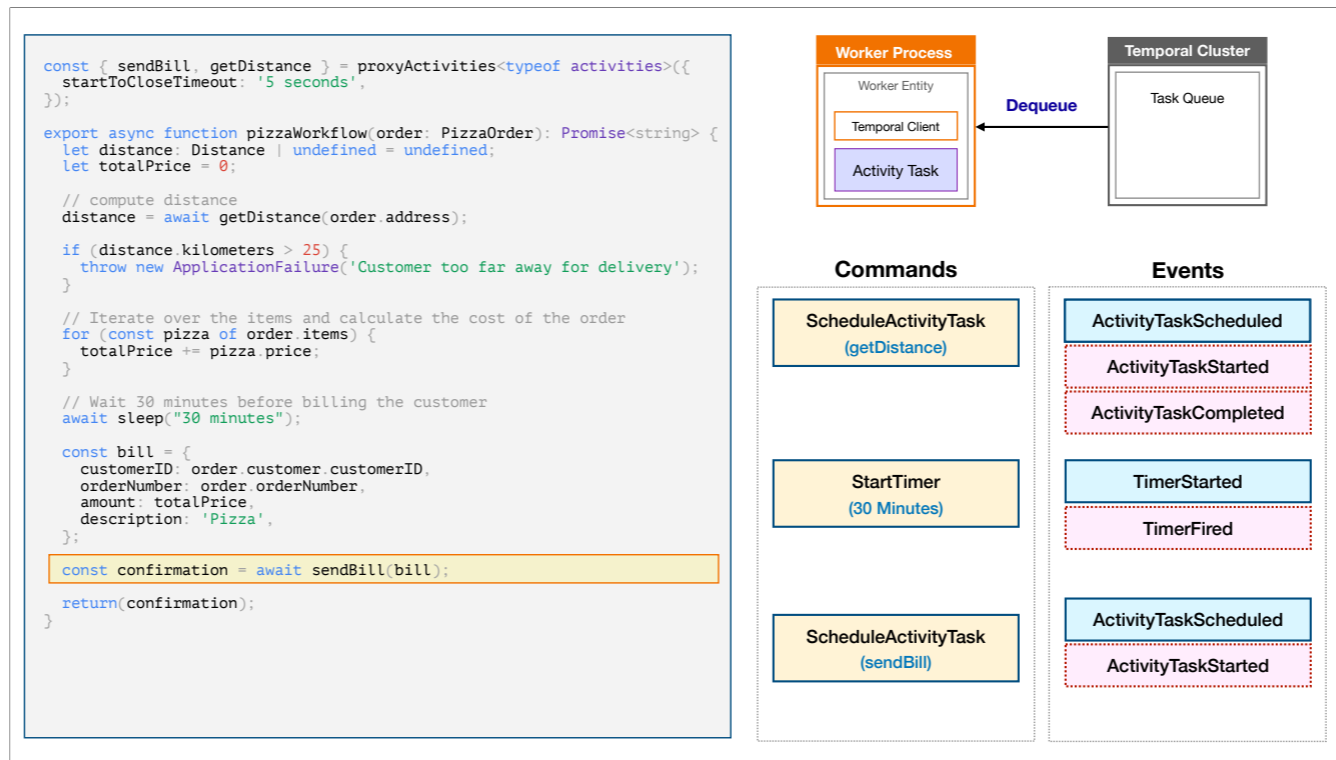
After 30 minutes has elapsed, the Timer is fired, and the Temporal Cluster logs the Event to the history. The Workflow Execution continues with the next statement, but this is an internal step.



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

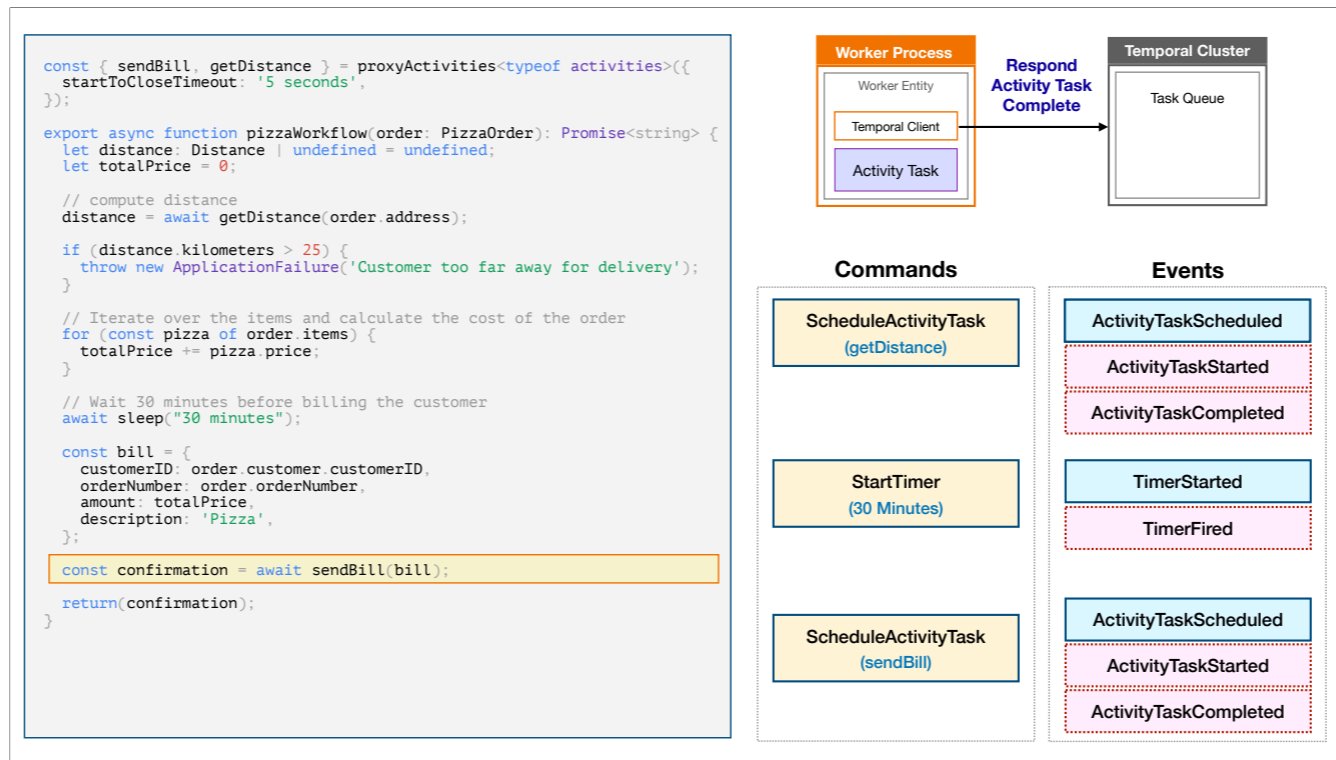


When the Worker polls the Task Queue, it will be matched with this Task.



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.



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.

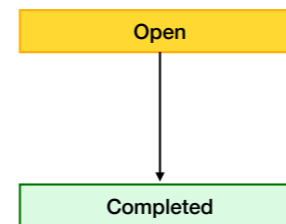
Workflow Execution States



You already know that Workflows can be Open or Closed. But there are different types of Closed states.

Completed

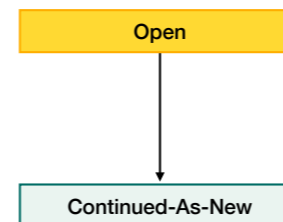
Meaning: The Workflow function returned a result



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

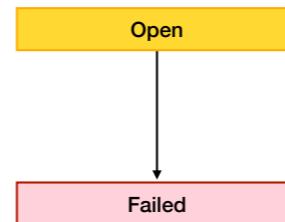


A variation on this is known as Continued-as-New, which means that the code is still running, but any future progress will take place in a new Workflow Execution and Event History.

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

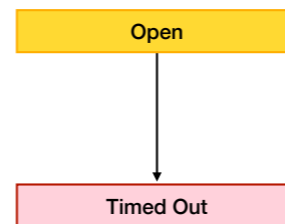
Meaning: The Workflow function returned an error



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

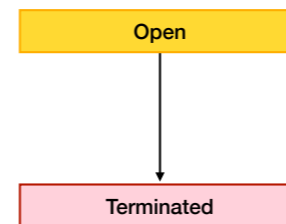
Meaning: Execution exceeded a specified time limit



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

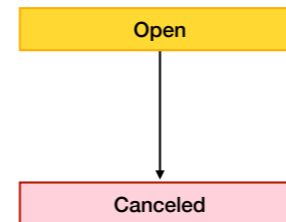
Meaning: Temporal Cluster acted upon a termination request



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

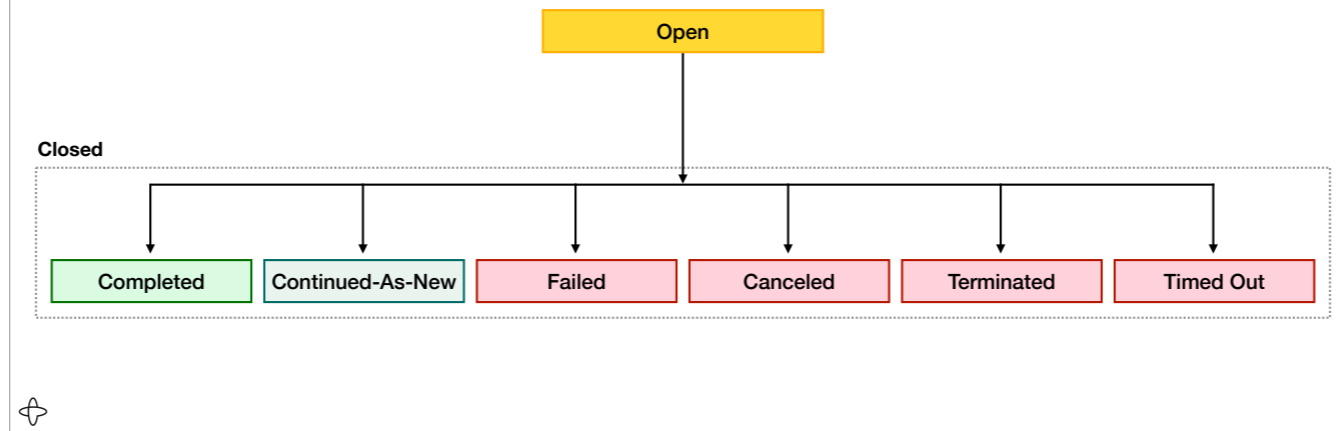
Canceled

Meaning: Temporal Cluster acted upon a request to cancel execution



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

Summary of Workflow Execution States



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



Now let's look at the states of Tasks.

Activity Task Event Sequence

ActivityTaskScheduled

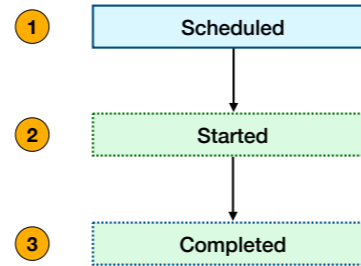
ActivityTaskStarted

ActivityTaskCompleted



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

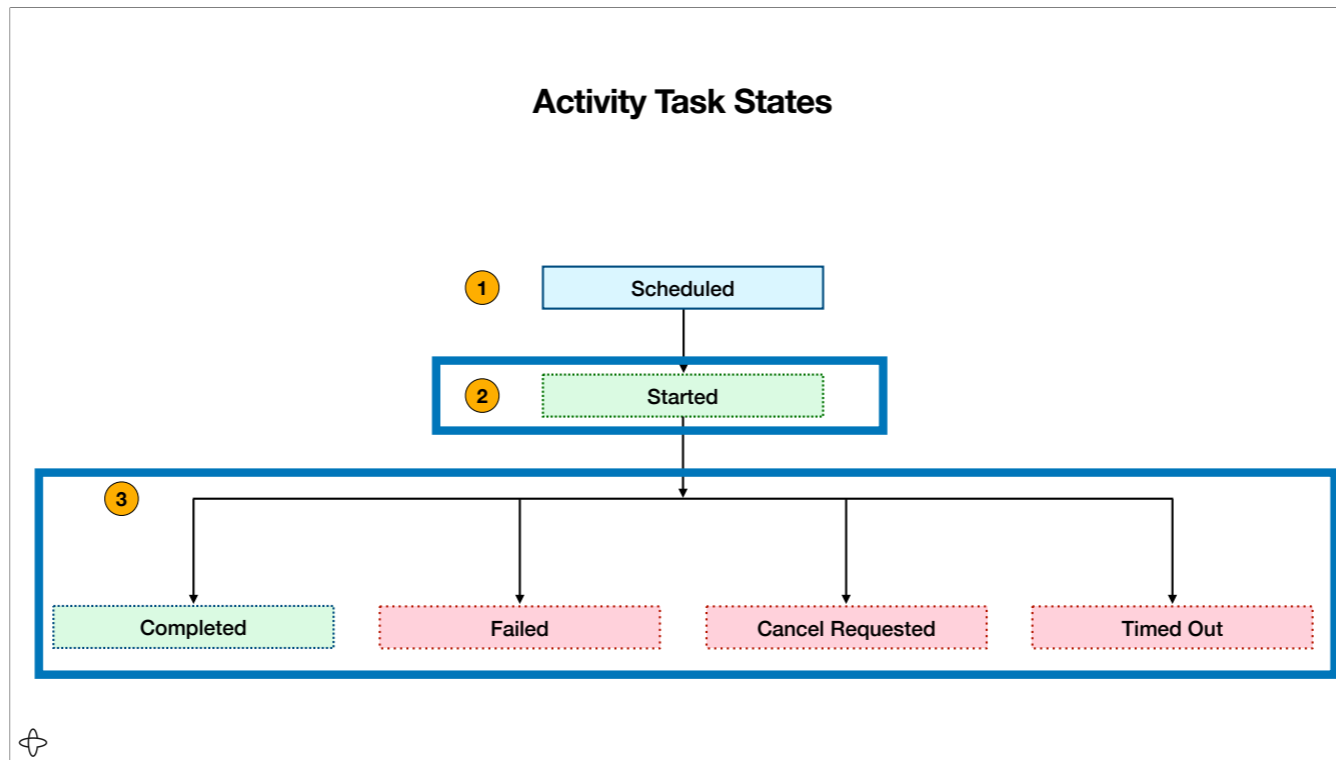
Activity States in that Sequence



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 dequeuing 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.

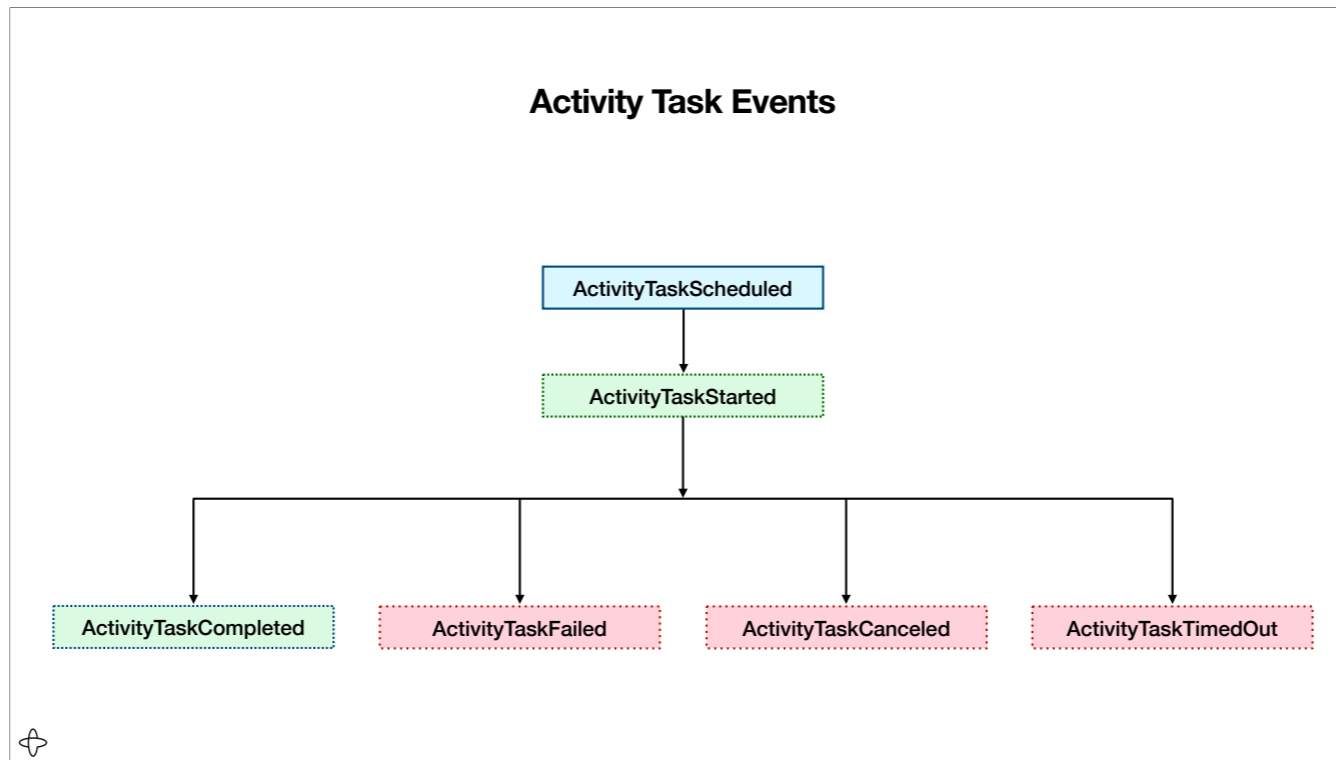
For example, Start-to-Close Timeout is the maximum amount of time allowed for between when the Worker starts working on a task and when that Task enters the closed state. In other words, it specifies the maximum duration between step 2,

[advance]

when the Worker begins execution, and step 3,

[advance]

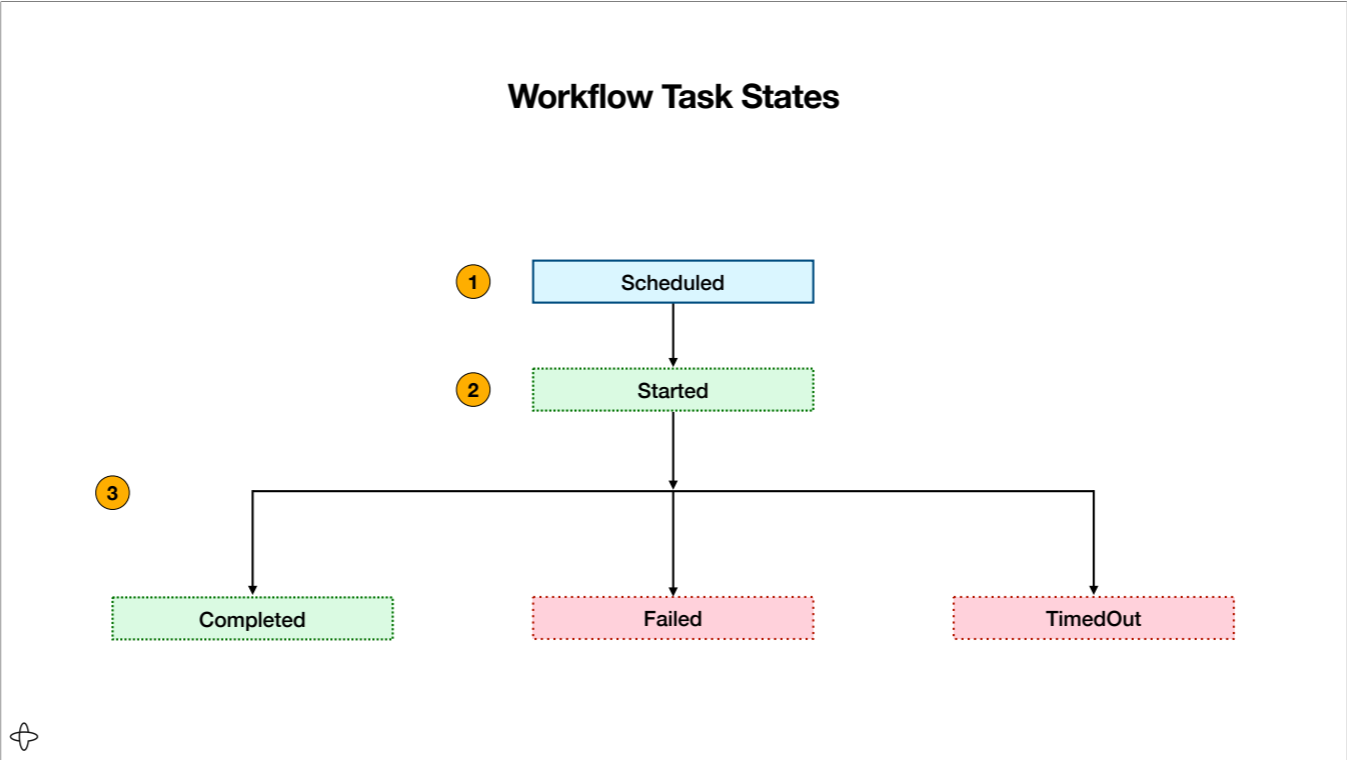
when execution ends.



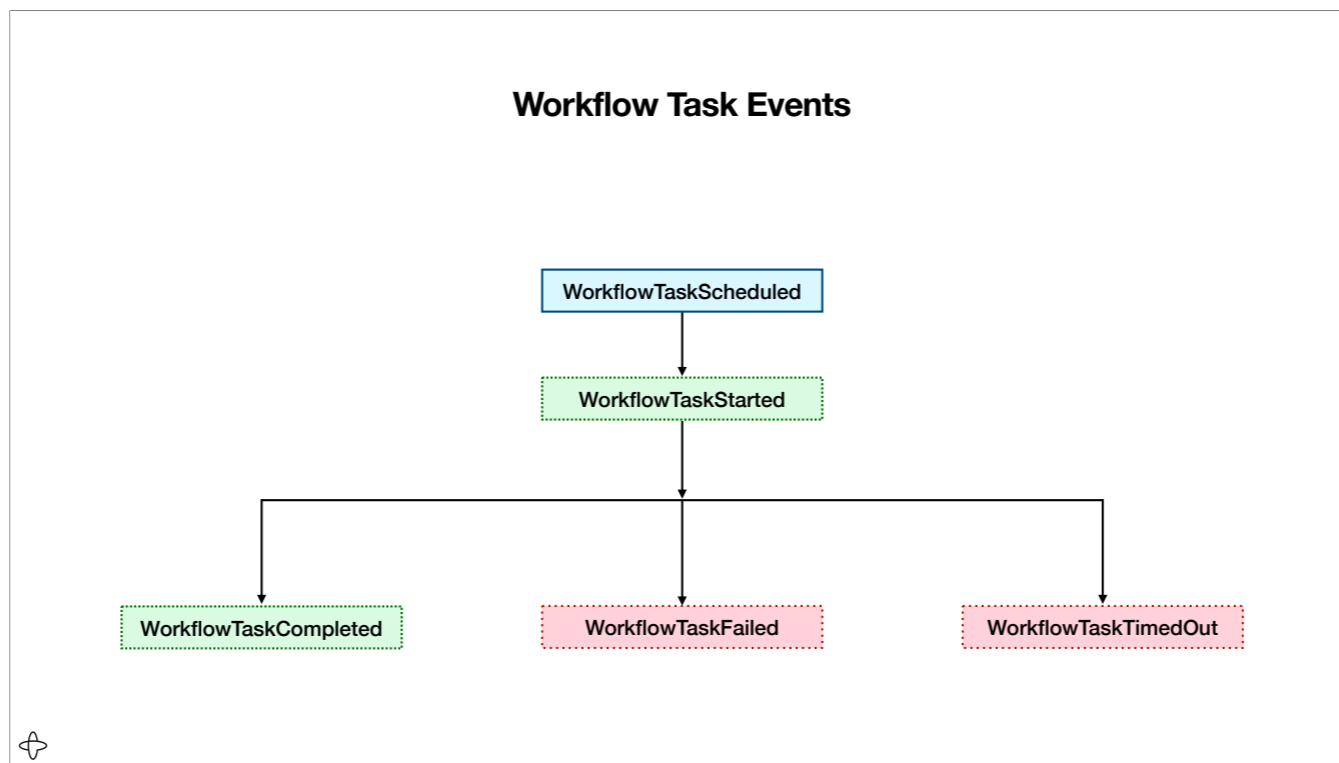
Let's now look at the Events corresponding to each of these states and the various ways that a Task can reach one of these closed states.

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.



The pattern you saw applies to Workflow Tasks as well.



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

Later Workflow Task

8	2023-07-19 UTC 17:02:31.36	WorkflowTaskScheduled
Summary Task Queue		
Task Queue Name	twwmbp:b7b2434d-4fb5-4ca6-b05f-bb98d6565a96	
Task Queue Kind	Sticky	
Task Queue Normal Name	durable-exec-tasks	

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



Let's get into debugging Workflow History next. And to do that, I'll go through a series of demonstrations rather than slides.

Demos

- **Debugging a Workflow that Doesn't Progress**
- **Interpeting Event History Workflow Execution States can be Open or Closed**
- **Terminating a Workflow Execution with the Web UI**
- **Identifying and Fixing a Bug in an Activity Definition**



Specifically, we'll go through these four demonstrations:

Debugging a Workflow that Doesn't Progress

Interpeting Event History Workflow Execution States can be Open or Closed

Terminating a Workflow Execution with the Web UI

Identifying and Fixing a Bug in an Activity Definition

Demo: Debugging a Workflow that Doesn't Progress



Demo

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)

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](#))

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](#))

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



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

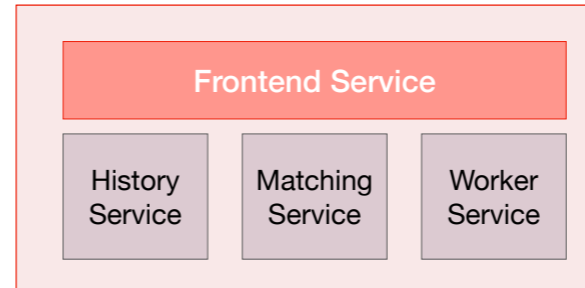
Maintains history and moves execution progress forward

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.

[advance]

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.

[advance]

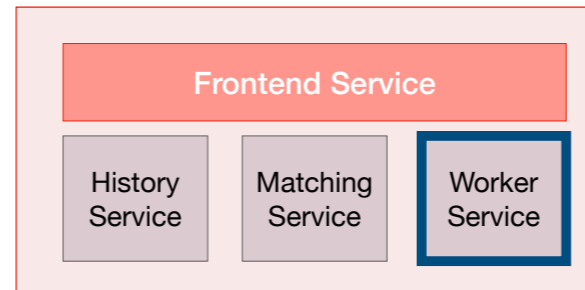
It works closely with the Matching Service, which hosts the Task Queue and matches tasks to polling Workers.

[advance]

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

The Worker Service

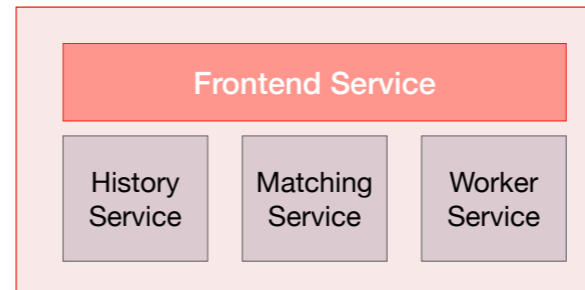
- **The Internal Workflows it runs are not exposed to users**
- **The service name is coincidental - it has no relationship to the Worker that's part of your application**



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.

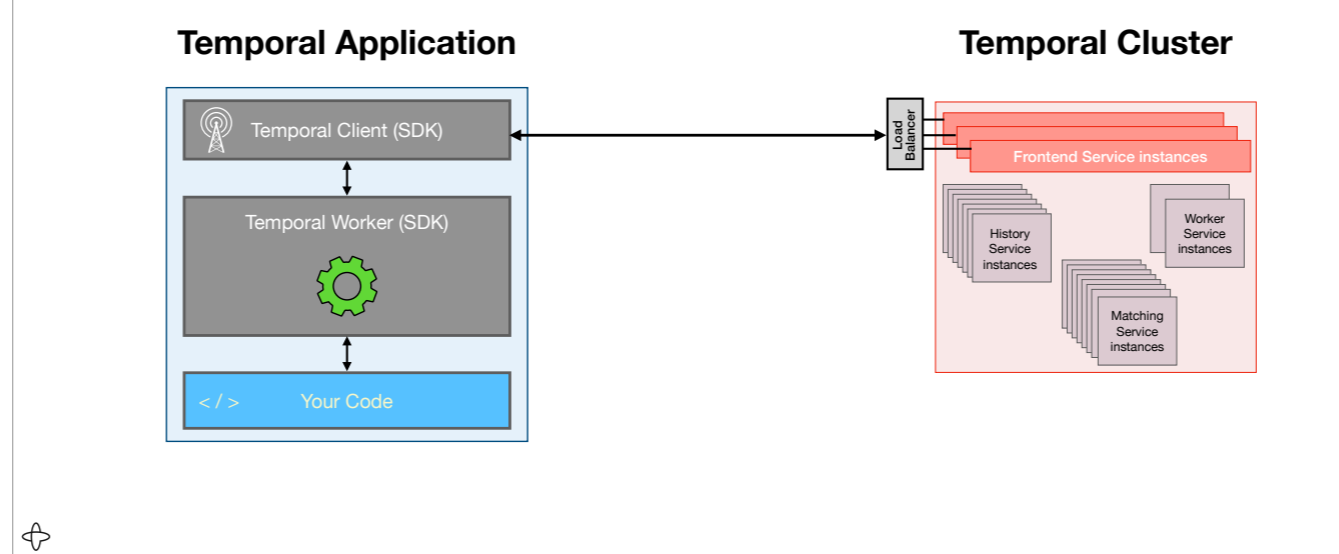
A Temporal Cluster with One Instance of Each Service

- What you get when running Temporal with Docker Compose or Temporal CLI
- Good for development on a small scale



Here you see a Temporal Cluster with one instance of each service, similar to what you might have for small-scale development, perhaps by deploying a cluster on your laptop with Docker Compose.

Cluster Scalability

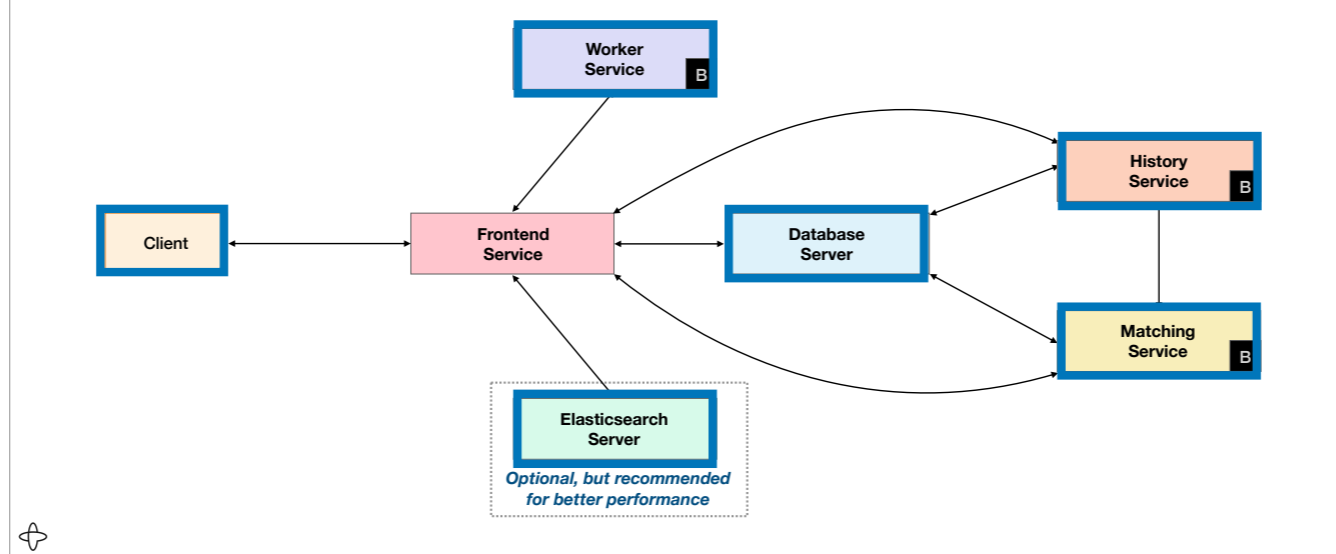


However, a Temporal Cluster can scale well beyond that, with multiple instances of each service. Production clusters often have dozens or even hundreds of instances of these services running, which provides availability because the cluster can continue operations even as some instances fail.

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

[advance]

have also included the required database component

[advance]

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,

[advance]

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

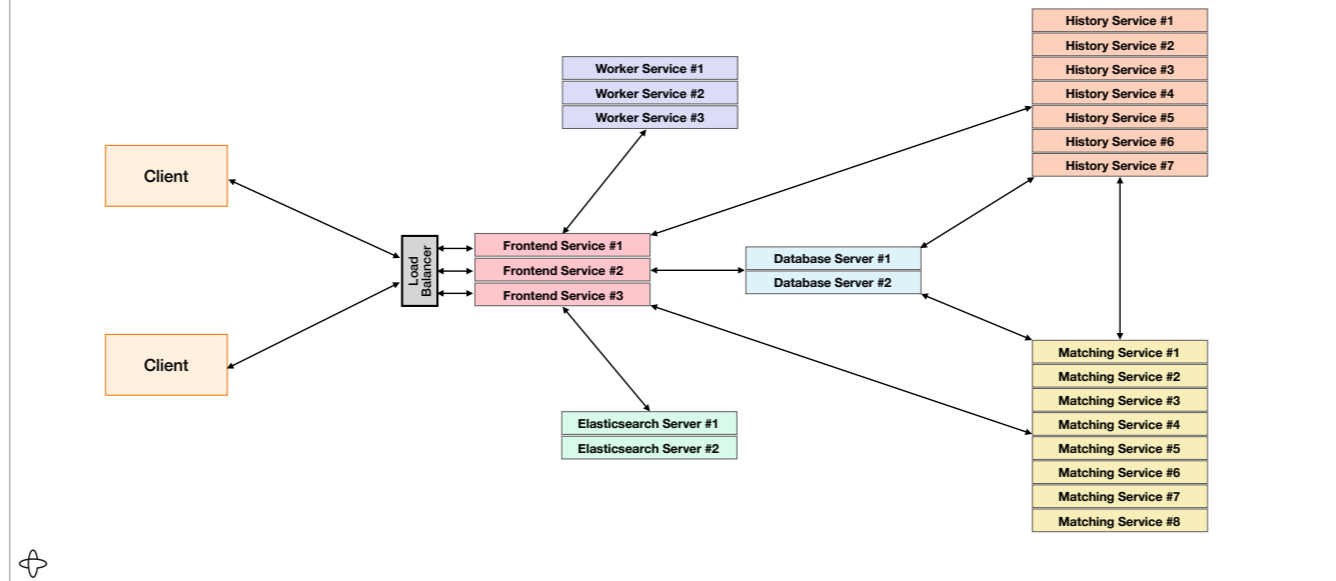
that starts a Workflow and retrieves its result.

[advance]

Clients do 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).

Creating a Temporal Client to a Local Cluster

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

```
// create connection details
const connection = await Connection.connect({ address: 'localhost:7233' });

// create the connection
const client = new Client({
  connection,
});
```



You create a connection and specify the address and port of the server and then use that connection with the Client. In this example, the client connects to a local Temporal server.

Creating a Temporal Client to a Local Cluster.

- If you pass no options to the client, you'll also connect to a local cluster:

```
// create the local connection  
const client = new Client();
```

- We recommend being explicit and specifying the options for the connection.



You may see a more compact version for connecting to local clusters. While this works, we recommend you explicitly set the connection options, as you'll eventually move to production.

Customizing a Temporal Client's Options

- **Specify attributes to configure the Connection:**
 - **address:** A colon-delimited string containing the hostname and port for the Frontend Service
 - Example: `fe.example.com:7233`
 - **tls:** Details about your key and certificate when making an mTLS connection
- **Specify attributes to create the Client:**
 - **connection:** The connection options you defined
 - **namespace:** A string specifying the namespace to use for requests sent by this Client



You configure the client's options in two places with the TypeScript SDK. You create a connection object and specify the address and port and the details about the key and certificate you use to make a secure connection.

Then you configure the client by passing the connection you created along with the namespace you want to connect to.

Configuring Client for a Non-Local Cluster

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

```
// create connection details
const connection = await Connection.connect({ address: 'mycluster.example.com:7233' });

// create the connection
const client = new Client({
  connection,
  namespace: 'your-namespace',
});
```

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

```
$ tctl --address mycluster.example.com:7233 \
  --namespace your-namespace \
  workflow list
```



Here's an example of a client connection that connects to a local server using a specific namespace.

Configuring Client for a Secure Cluster

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

```
// Cert and key
const cert = await fs.readFile('./path-to/your.pem');
const key = await fs.readFile('./path-to/your.key');

// Connection options
const connectionOptions = {
  address: 'your-namespace.tmprl.cloud:7233',
  tls: {
    clientCertPair: {
      crt: cert,
      key,
    },
  },
};

// create connection details
const connection = await Connection.connect(connectionOptions);

// create the connection
const client = new Client({
  connection,
  namespace: 'your-namespace',
});
```



Here's a more complex example where you're connecting to a remote Temporal cluster using mTLS. You read in the key and certificate, specify those when creating the connection, and then use that with the client. This example uses Temporal Cloud, but it works for any secure Temporal Cluster.

Preparing a Temporal Application for Deployment

- **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
- **TypeScript Temporal Applications are just Node apps.**
 - Your process can be the same process you use with any other Node.js application that uses TypeScript.



Application deployment is usually preceded by a build process. The tools used to do this vary by language, based on the SDK(s) used. You can use what is typical for the language or mandated by your organization; Temporal doesn't require you to use anything specific.

TypeScript Temporal apps are just Node apps, so you would use the same process to deploy your apps as you would with your other apps. You could deploy your app directly to servers, or build images and deploy using containers.

Bundle Workflows for Performance

- This improves performance since there are fewer imports.
- Activities stay separate.

```
const workflowOption = () =>
  process.env.NODE_ENV === 'production'
    ? {
      workflowBundle: {
        codePath: require.resolve('../workflow-bundle.js'),
      },
    }
    : { workflowsPath: require.resolve('./workflows') };

async function run() {
  const worker = await Worker.create({
    ...workflowOption(),
    activities,
    taskQueue: 'production-sample',
  });
}
```



With the TypeScript SDK, you can create a bundle of your Workflows. This will combine all of your workflow definitions into a single file which will improve the startup time of your Workers.

Here's an example that uses a Workflow Bundle in production, but loads the Workflows directly in development mode.

Creating the Bundle

- Create a `bundle.ts` script as part of your build process that uses the `bundleWorkflowCode` function:

```
import { bundleWorkflowCode } from '@temporalio/worker';
import { writeFile } from 'fs/promises';
import path from 'path';

async function bundle() {
  const { code } = await bundleWorkflowCode({
    workflowsPath: require.resolve('./workflows'),
  });
  const codePath = path.join(__dirname, '../workflow-bundle.js');

  await writeFile(codePath, code);
  console.log(`Bundle written to ${codePath}`);
}

bundle().catch((err) => {
  console.error(err);
  process.exit(1);
});
```



The bundle itself is something you can create with a script you include as part of your build process, using the `bundleWorkflowCode` function. This is a script from our samples that shows how to do this.

Temporal Application Deployment

- **Once ready, you'll deploy the application to production**
 - Deploy your code, plus runtime-time dependencies (e.g., Worker, Client, other libraries.)
 - 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.



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.

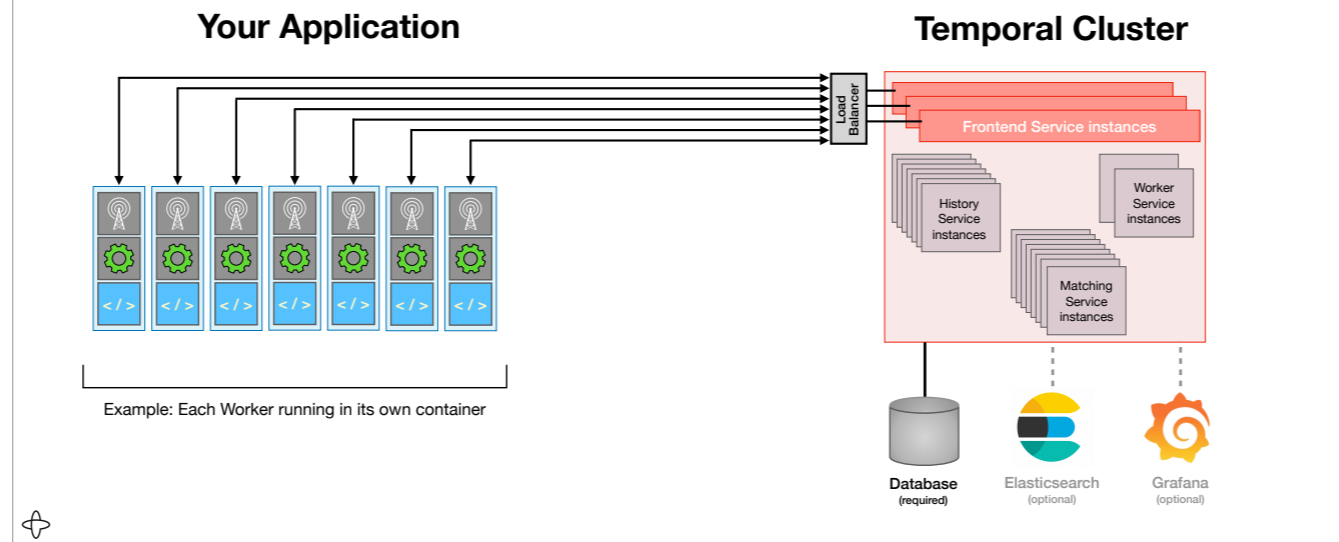
That will include artifacts compiled from code you write; for example, the Workflow Definition, Activity Definitions, and the Worker configuration, but also all of the dependencies used by that code. Those dependencies include the Temporal SDK, as well as any other libraries your code might use, which 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

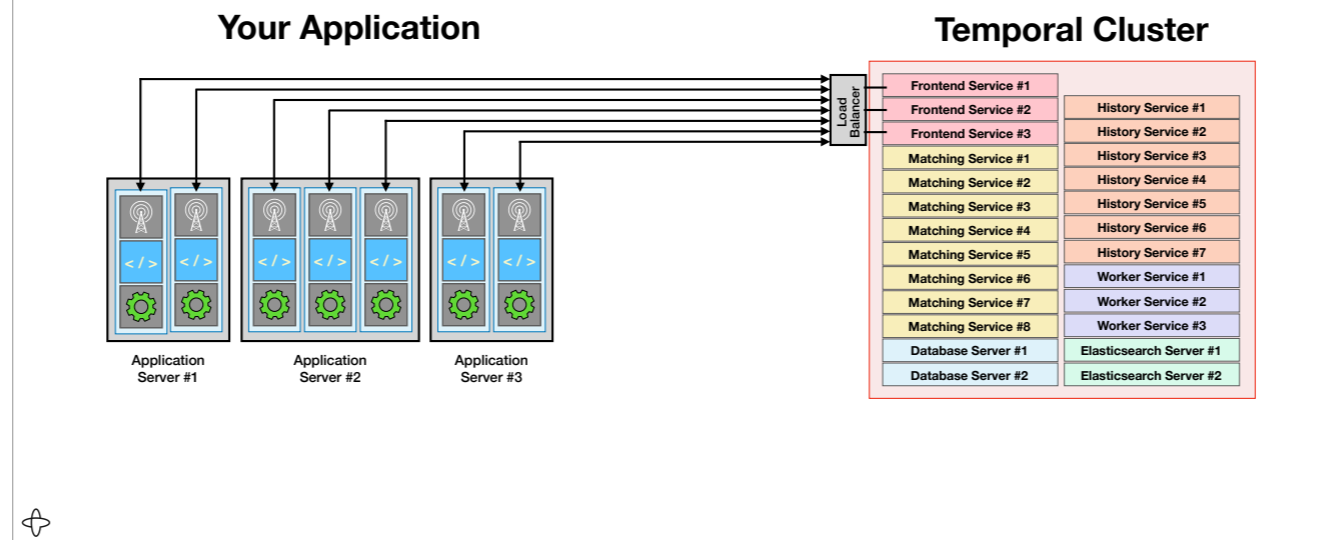
Deployment Scenario #1



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, along with the required database backend and other optional services.

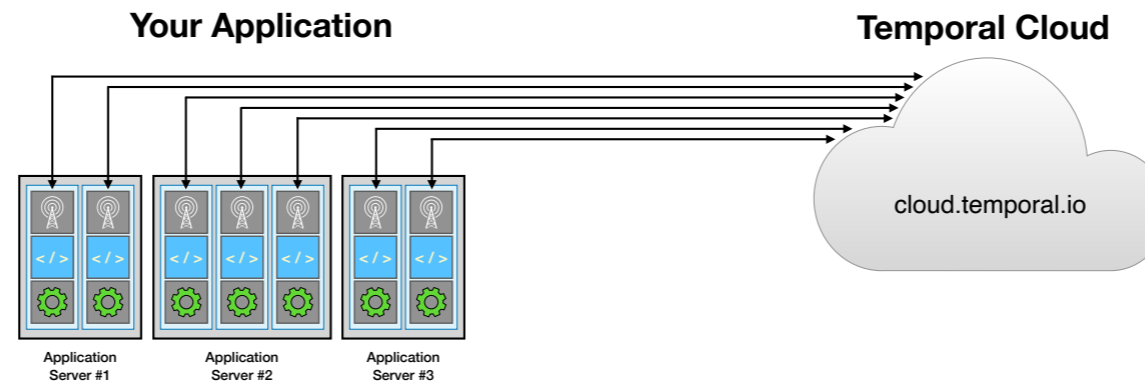
Physical View of an Application in Production



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



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



So we've gone over a lot of topics so far. There's one more thing we need to discuss, and that's determinism with Workflows, and why it's important.

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.

Start Workflow Execution

```
const { sendBill, getDistance } = proxyActivities<typeof activities>({
  startToCloseTimeout: '5 seconds',
});

export async function pizzaWorkflow(order: PizzaOrder): Promise<string> {
  let distance: Distance | undefined = undefined;
  let totalPrice = 0;

  // compute distance
  distance = await getDistance(order.address);

  if (distance.kilometers > 25) {
    throw new ApplicationFailure('Customer too far away for delivery!');
  }

  // Iterate over the items and calculate the cost of the order
  for (const pizza of order.items) {
    totalPrice += pizza.price;
  }

  log.info('Calculated cost of order', {});

  // Wait 30 minutes before billing the customer
  await sleep("30 minutes");

  // call a local function to create the input passed to next Activity
  const bill = createBill(order, totalPrice);

  const confirmation = await sendBill(bill);

  return(confirmation);
}
```

```
await client.workflow.start(pizzaWorkflow, {args[input],...})
```

```
{
  "orderNumber": "21238",
  "customer": {
    "customerID": 12983,
    "name": "María García",
    "email": "marial985@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"
  }
}
```

Commands and Events are essential to Workflow Replay, so 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'll use yellow highlighting to distinguish statements that result in Commands from code that does not, which will be highlighted in white, just like before.

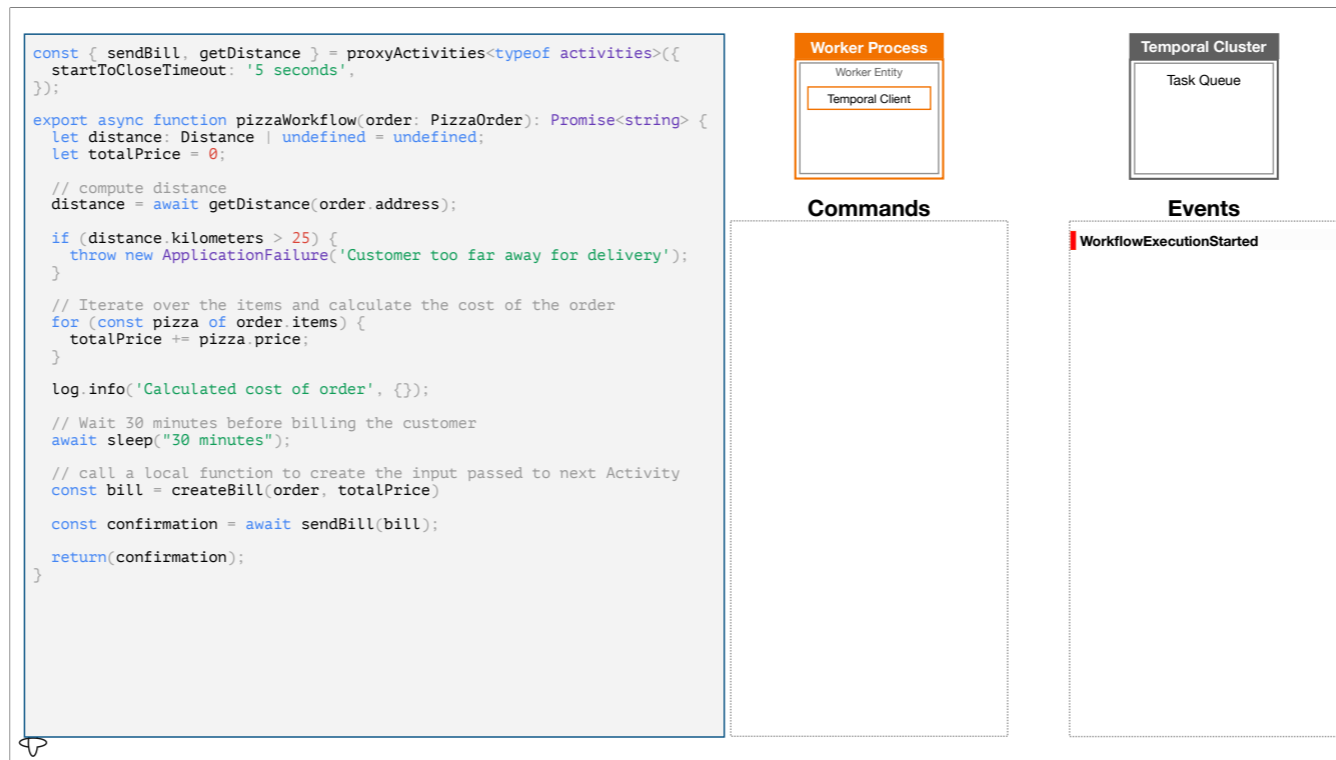
I'll start out by quickly walking through a Workflow Execution, showing a crash a little more than halfway through, and then explaining how Temporal uses Workflow Replay to recover the state, ultimately resulting in a completed execution that's identical to one that hadn't crashed.

[advance]

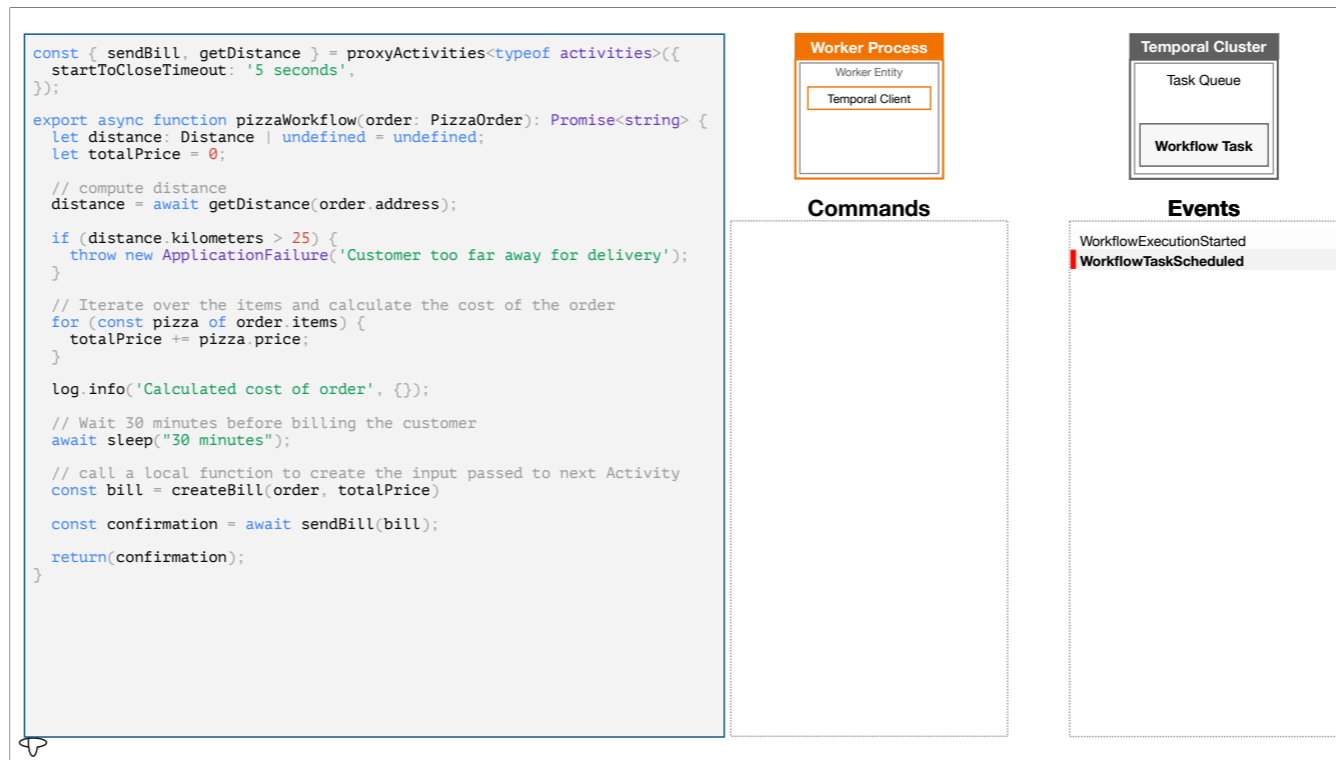
It all begins by combining the code in the Workflow Definition with a request to execute it, passing in some input data.

[advance]

In this case, the input data contains information about the customer and the pizzas they ordered.

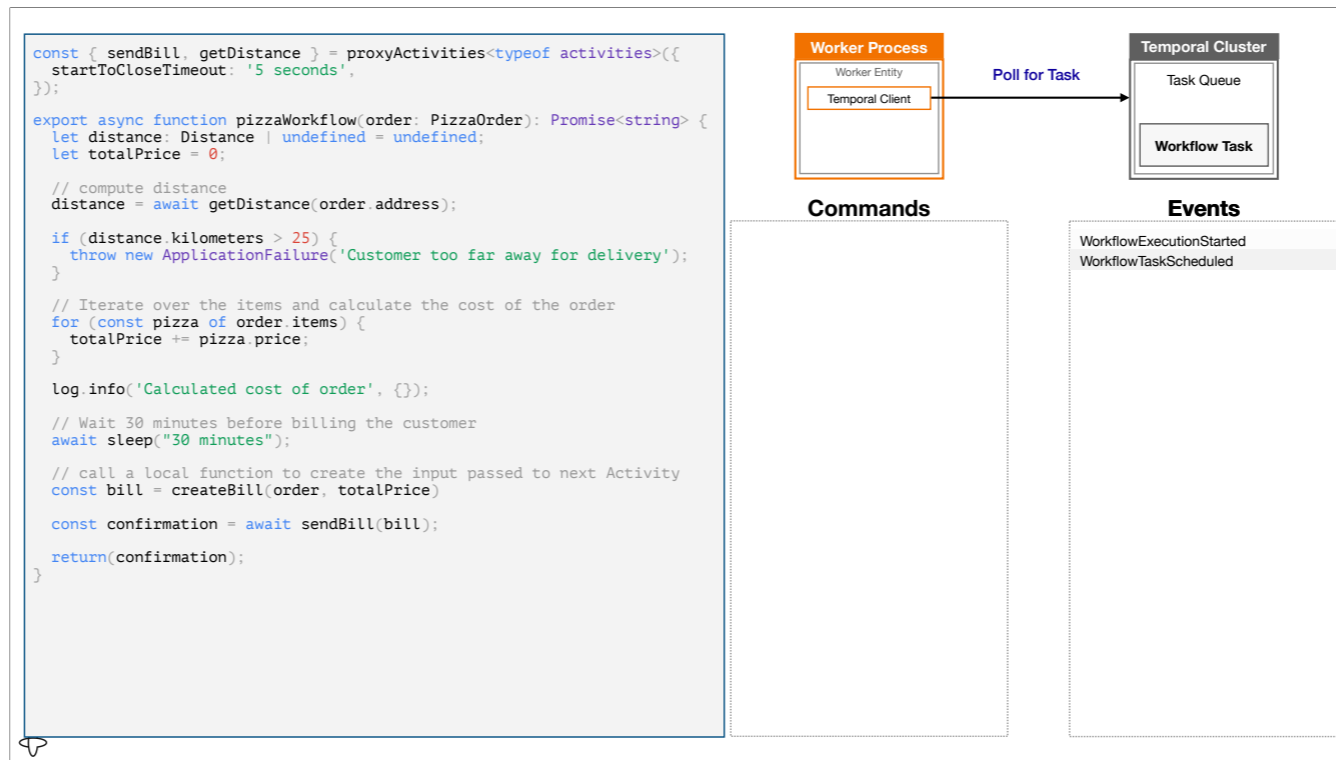


This results in the Temporal Cluster logging a `WorkflowExecutionStarted` Event into the history,

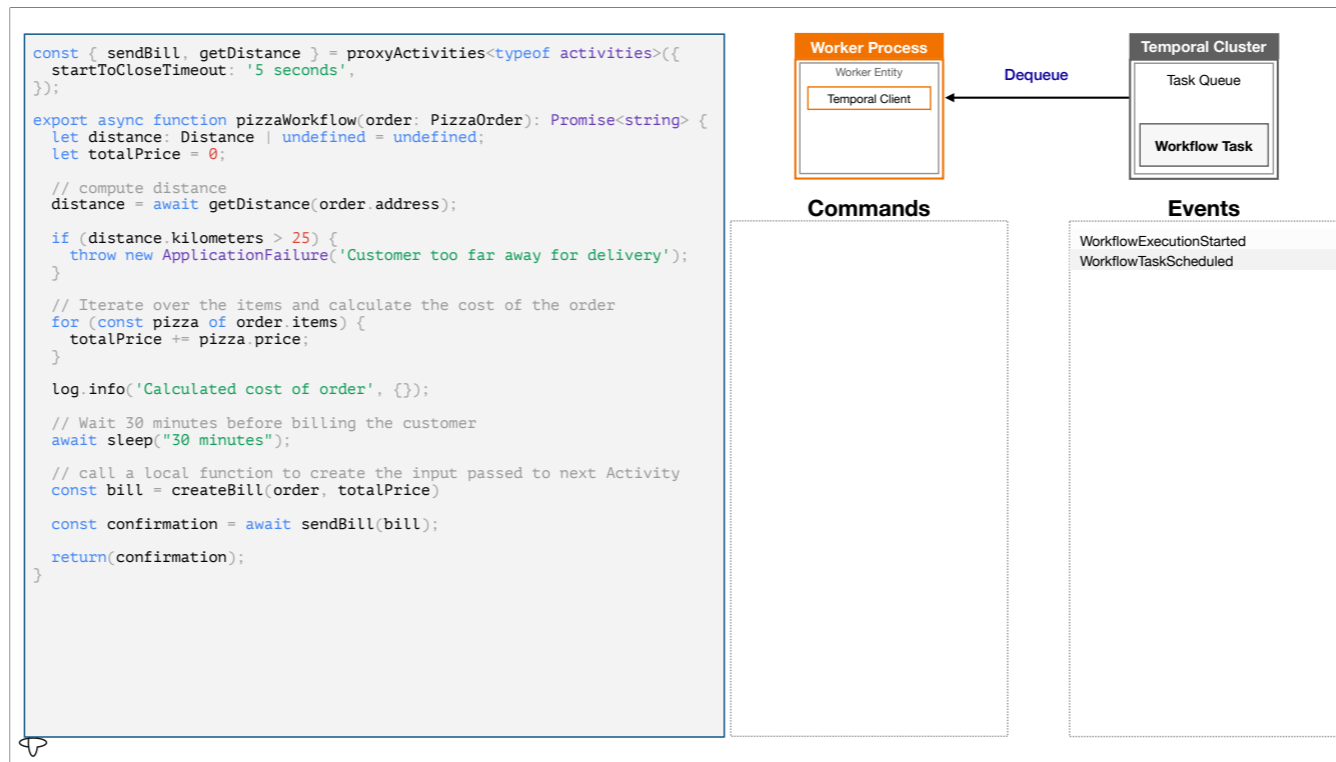


adding a Workflow Task to the queue, and logging a `WorkflowTaskScheduled` Event.

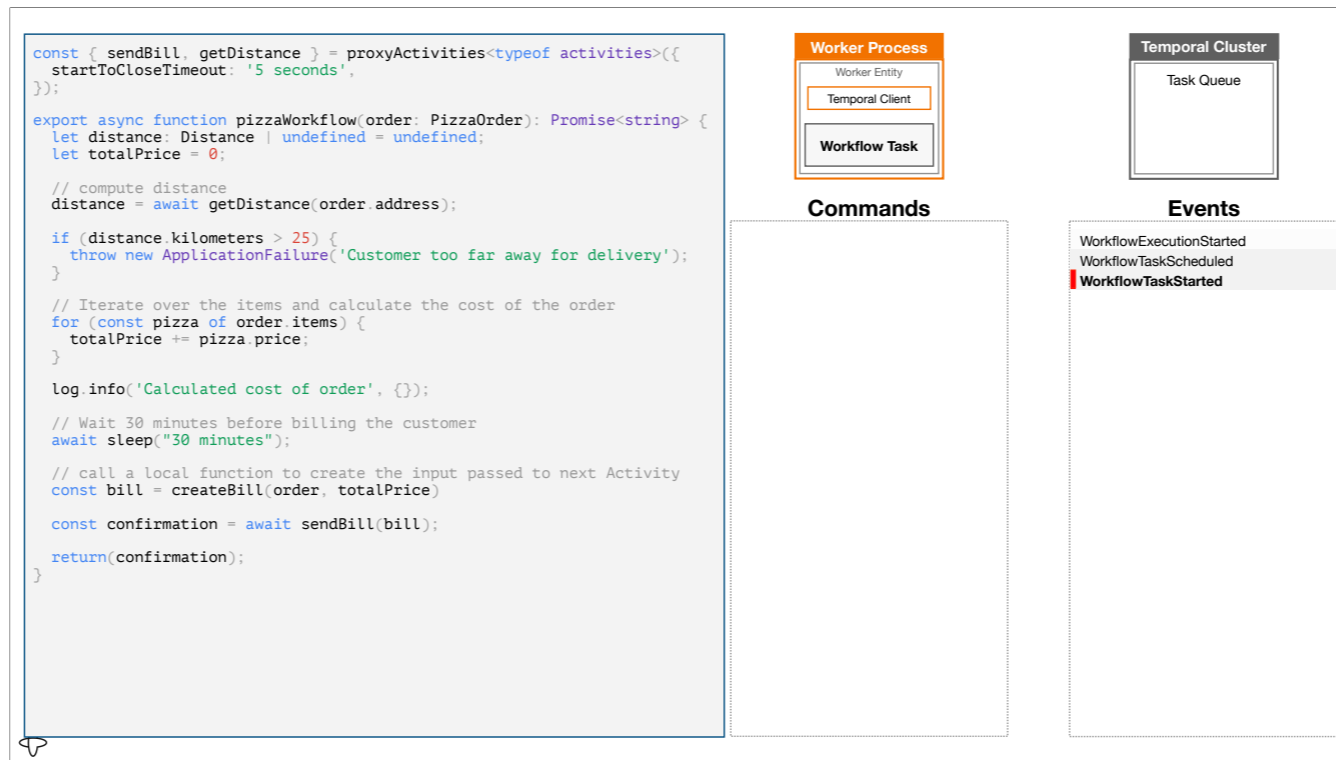
Although I didn't indicate it here, due to limited space on the screen, the `WorkflowExecutionStarted` Event contains the input data provided to this Workflow Execution.



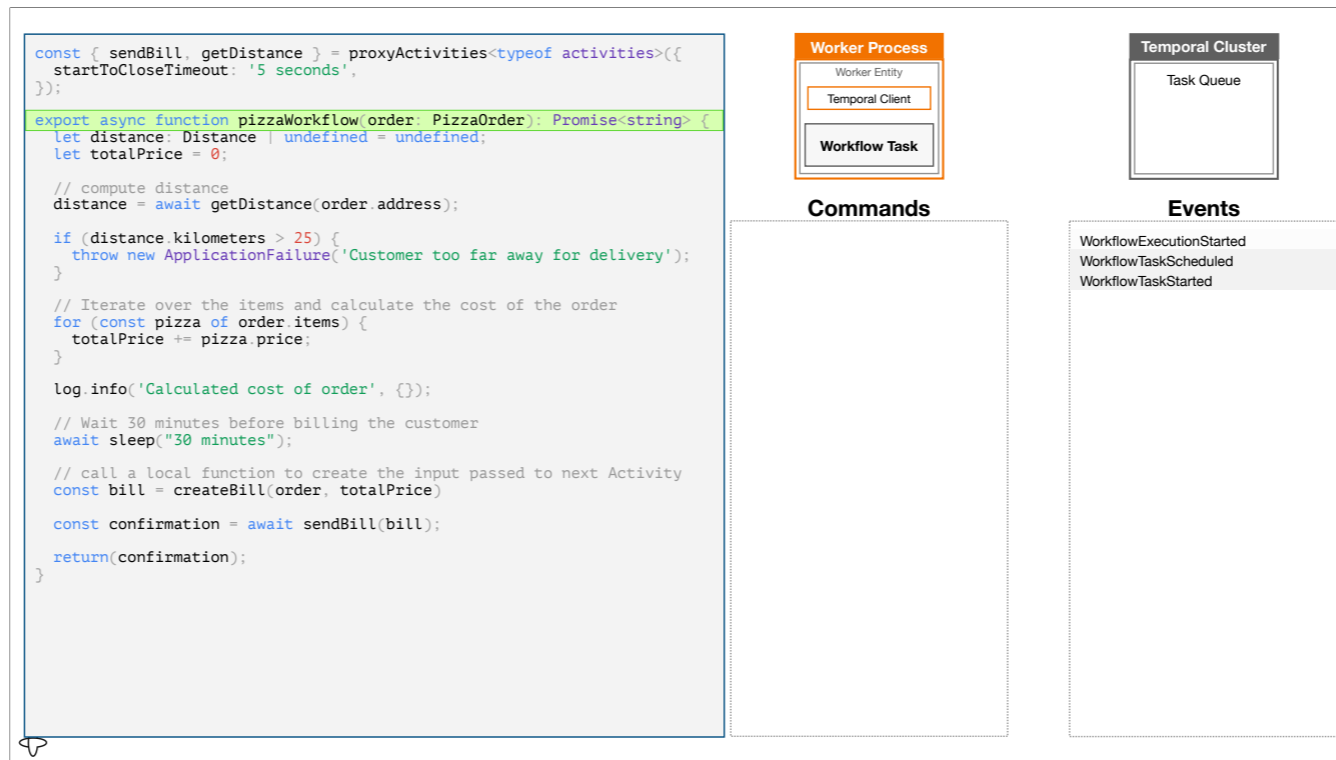
When a Worker polls the Task Queue



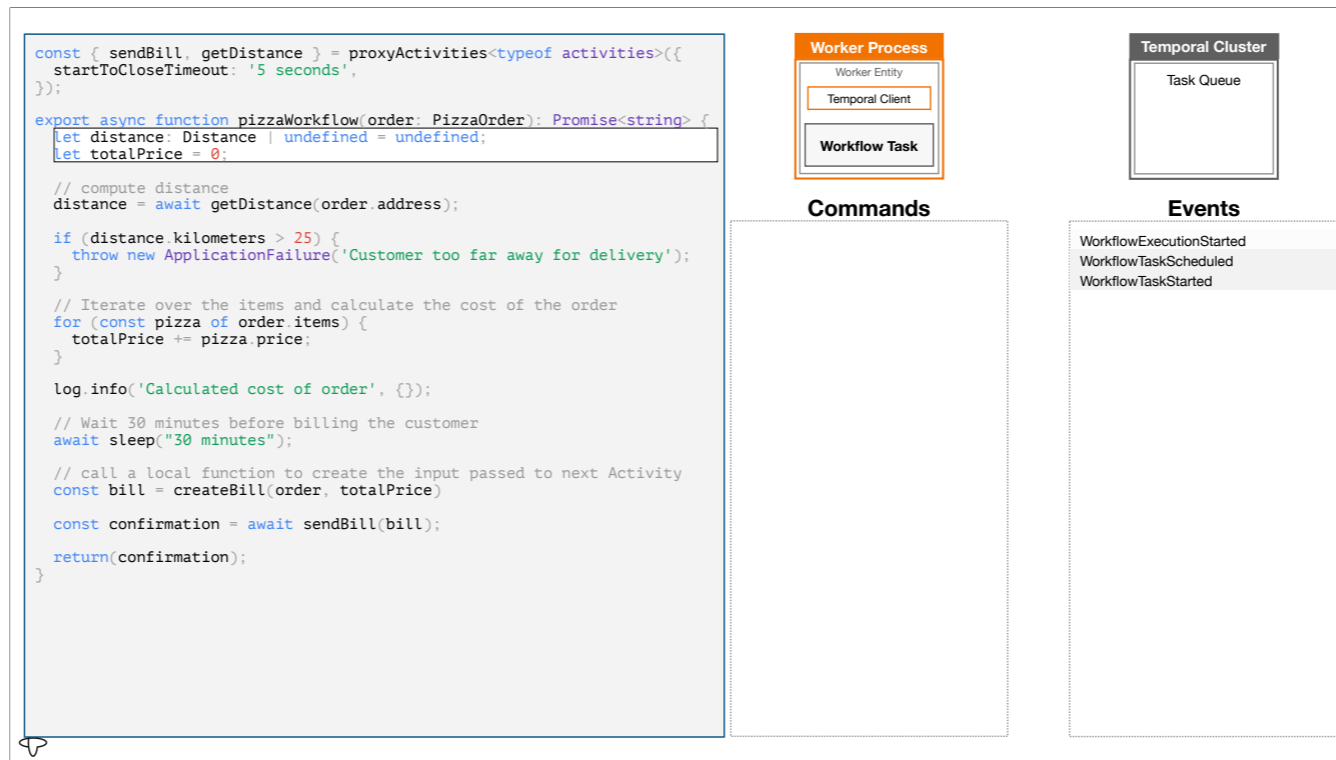
and accepts the Task,



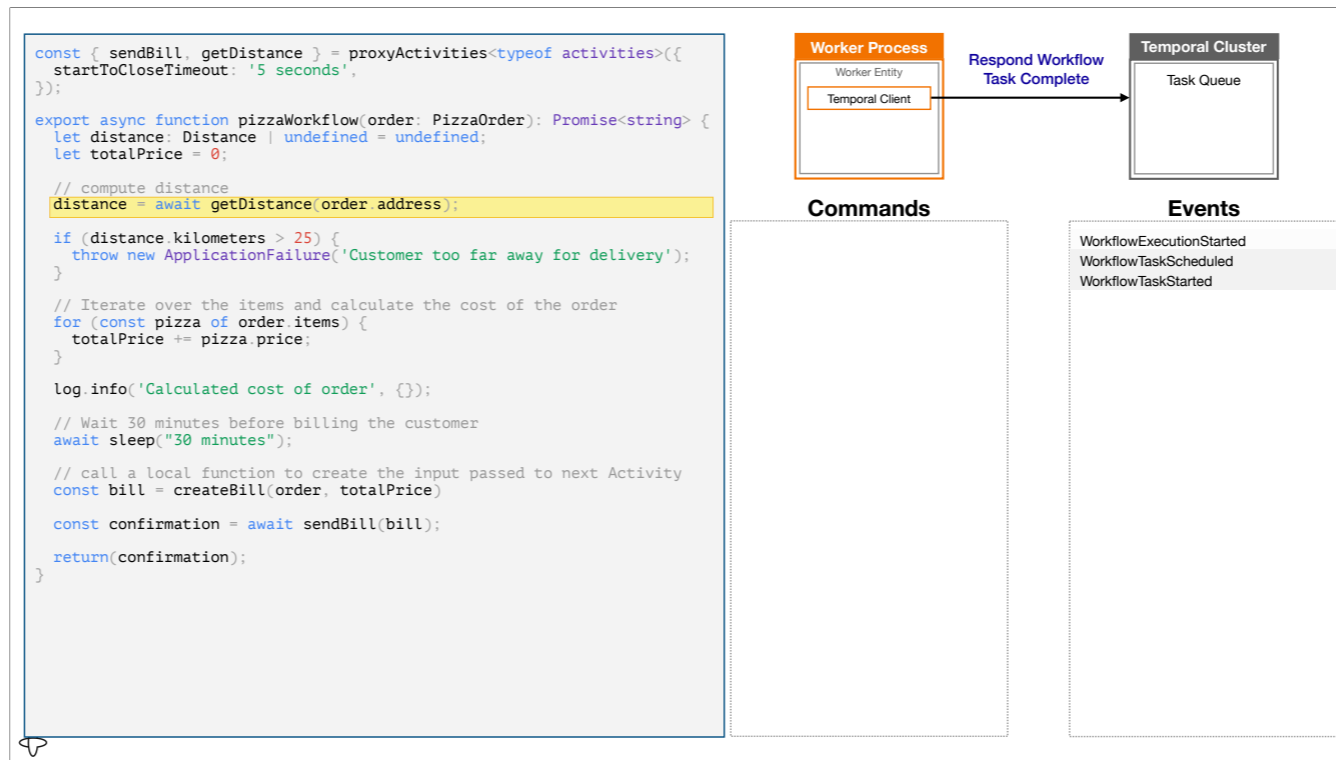
the cluster logs a `WorkflowTaskStarted` Event.



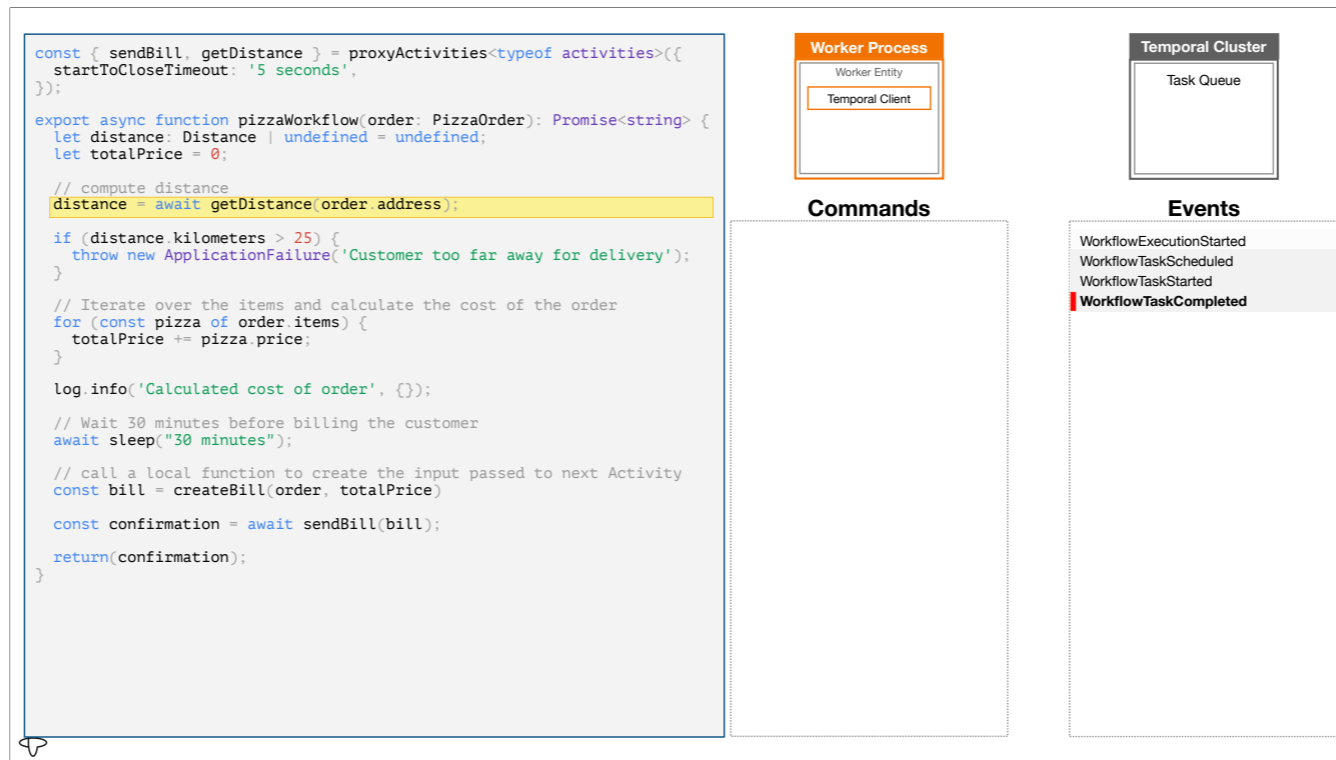
The Worker then invokes the Workflow function and runs the code within it, one statement at a time.



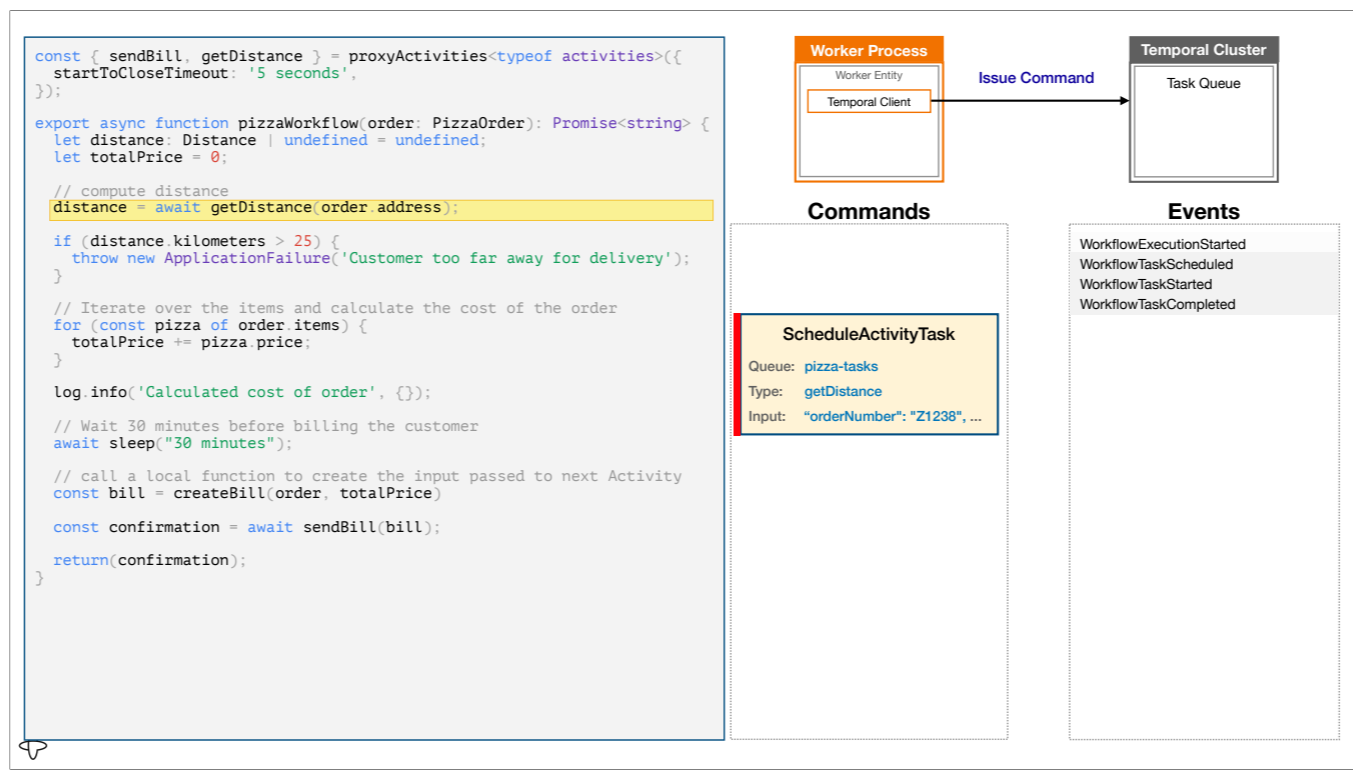
The first statements do not result in any interaction with the cluster.



This statement requests the execution of an Activity,



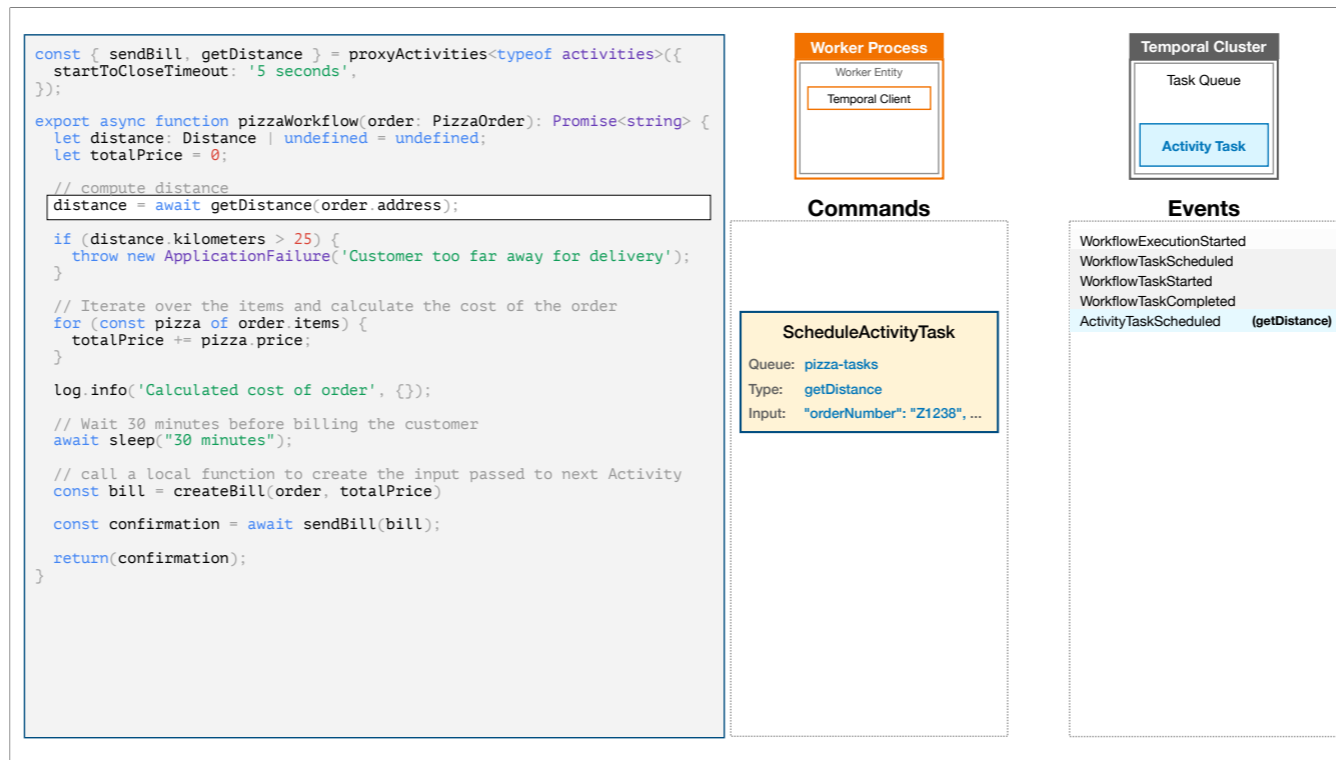
so the Worker completes the current Workflow Task.



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



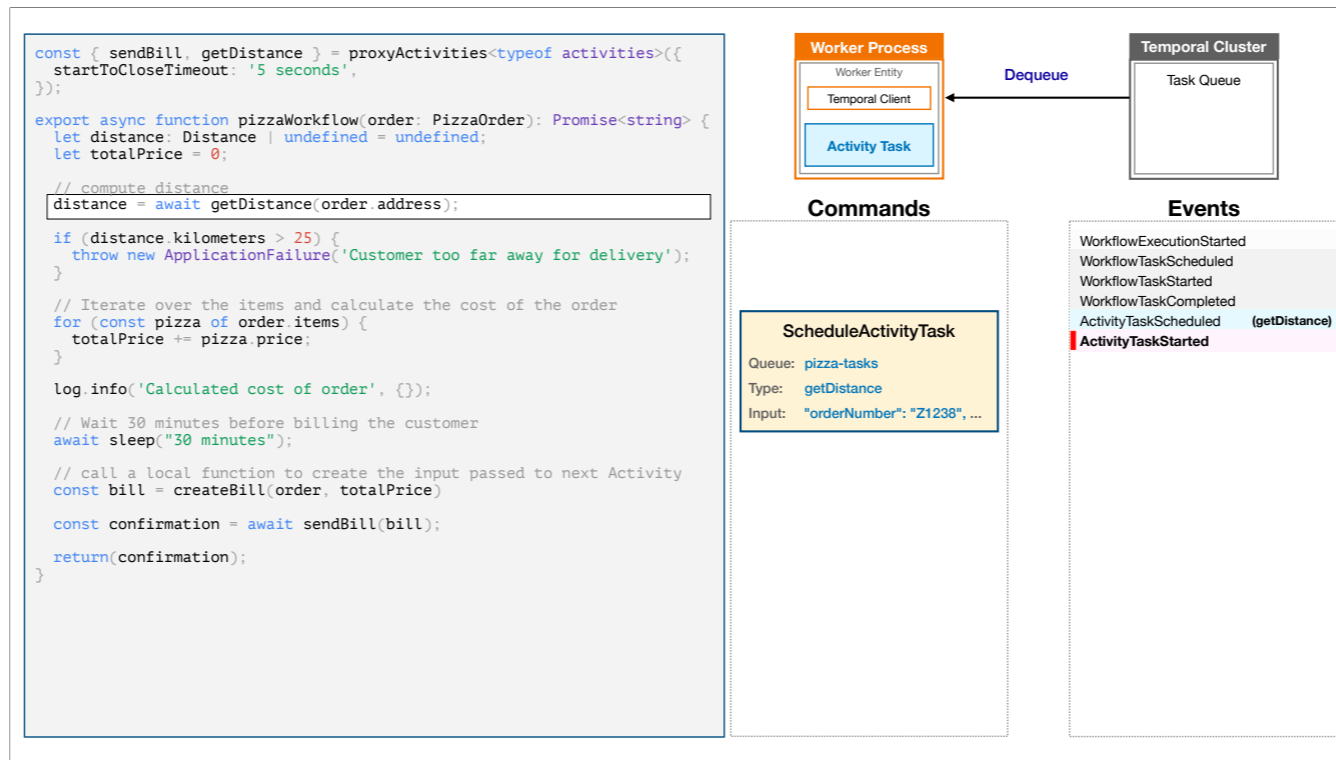
In response, the cluster queues an Activity Task and logs an `ActivityTaskScheduled` Event. I have shown this in blue to indicate that it is the direct result of the Command.



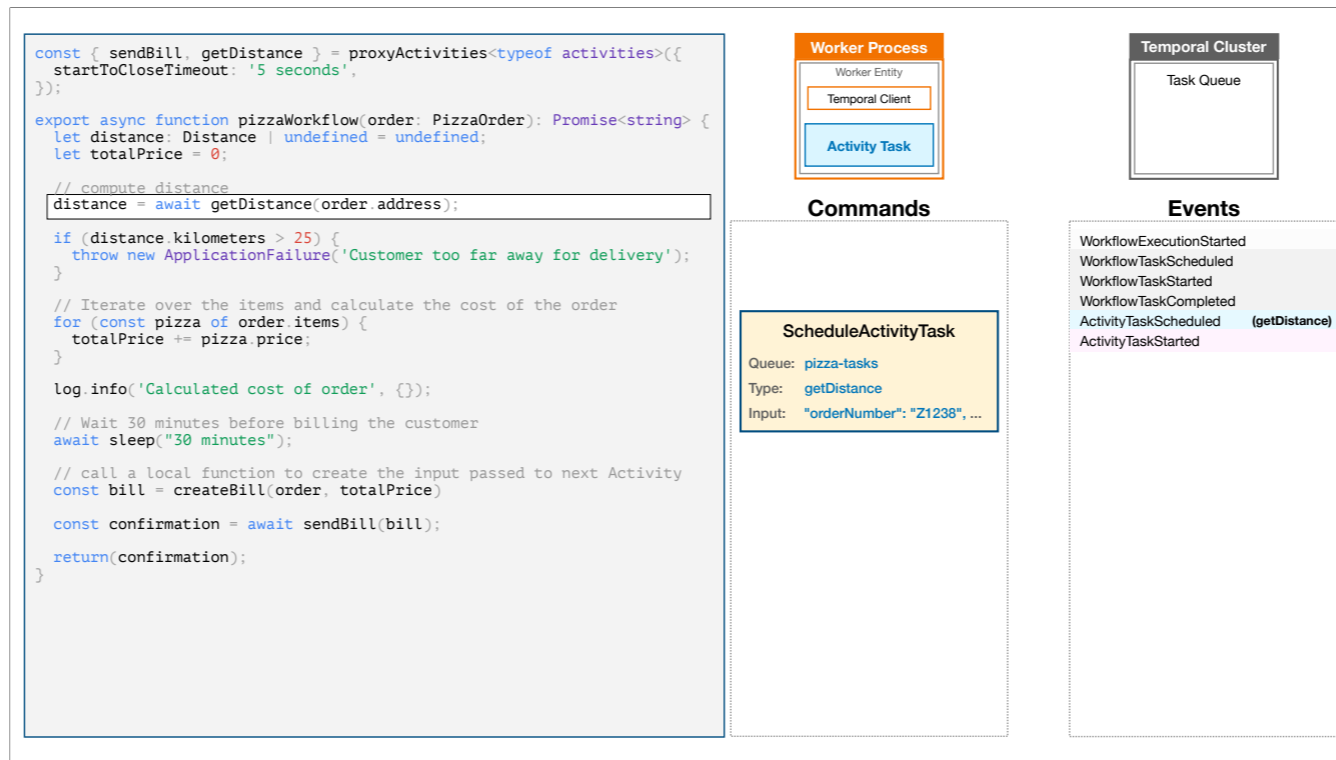
The Worker now awaits the result from Activity Execution.



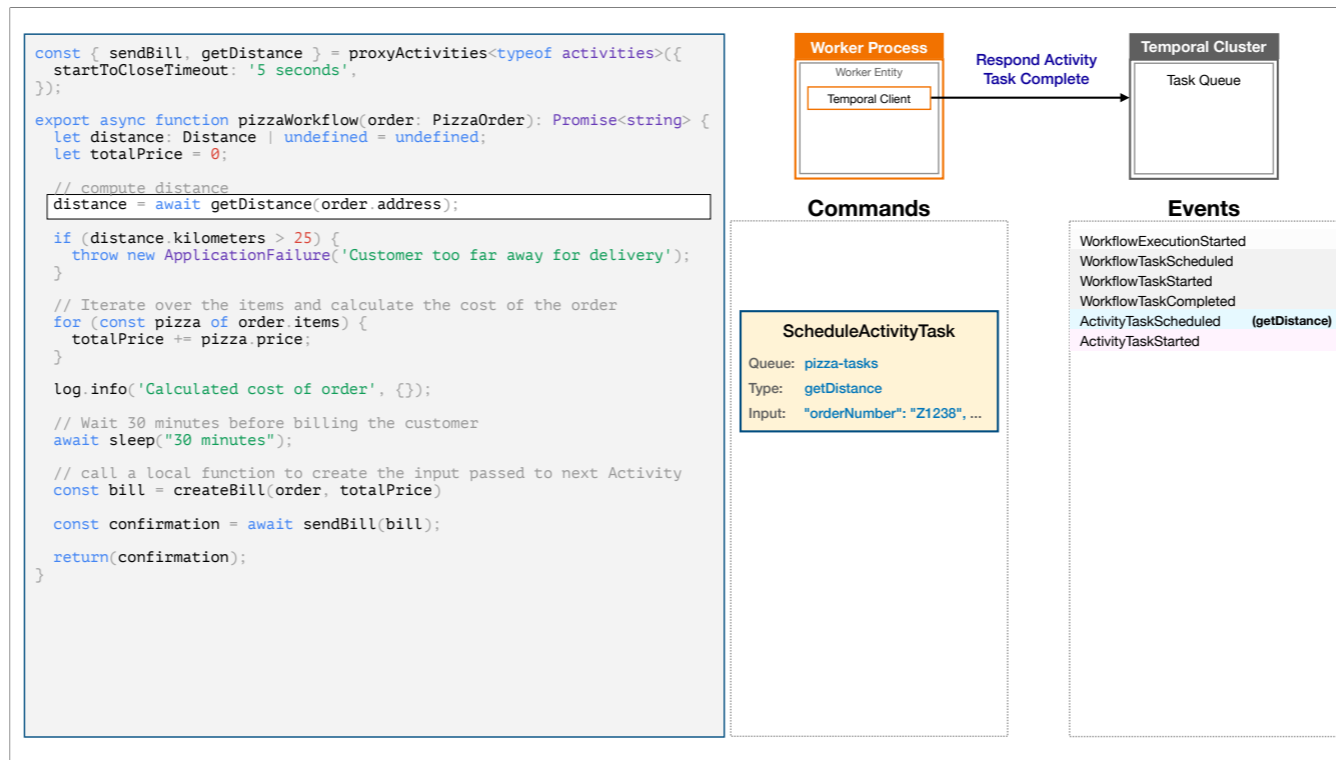
When this Worker—or another one—has spare capacity to do some work, it polls and is matched with the Activity Task, which it accepts.



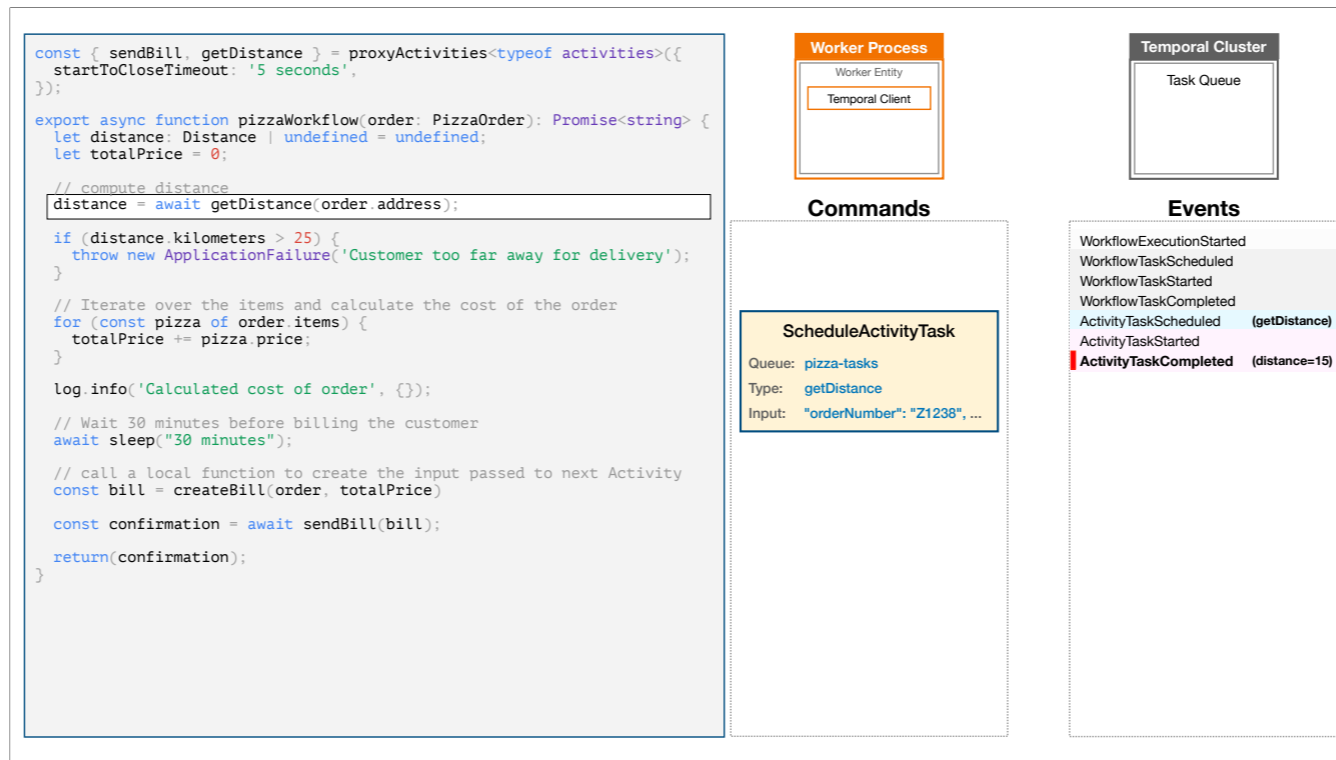
The cluster logs an Event to signify that the Worker has started the Activity Task. I've shown this in pink to indicate that it's the indirect result of the Command.



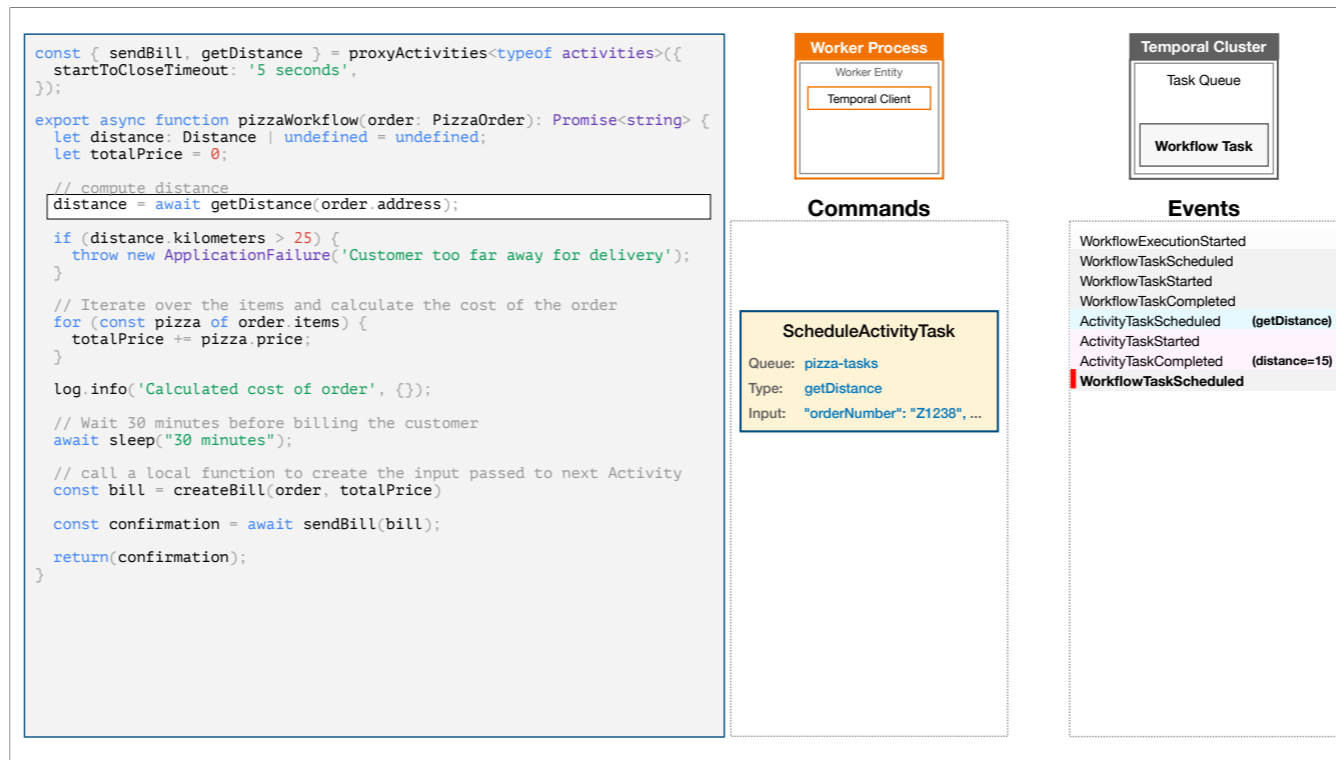
The Worker invokes the Activity Definition. In this case, that's the `getDistance` function.



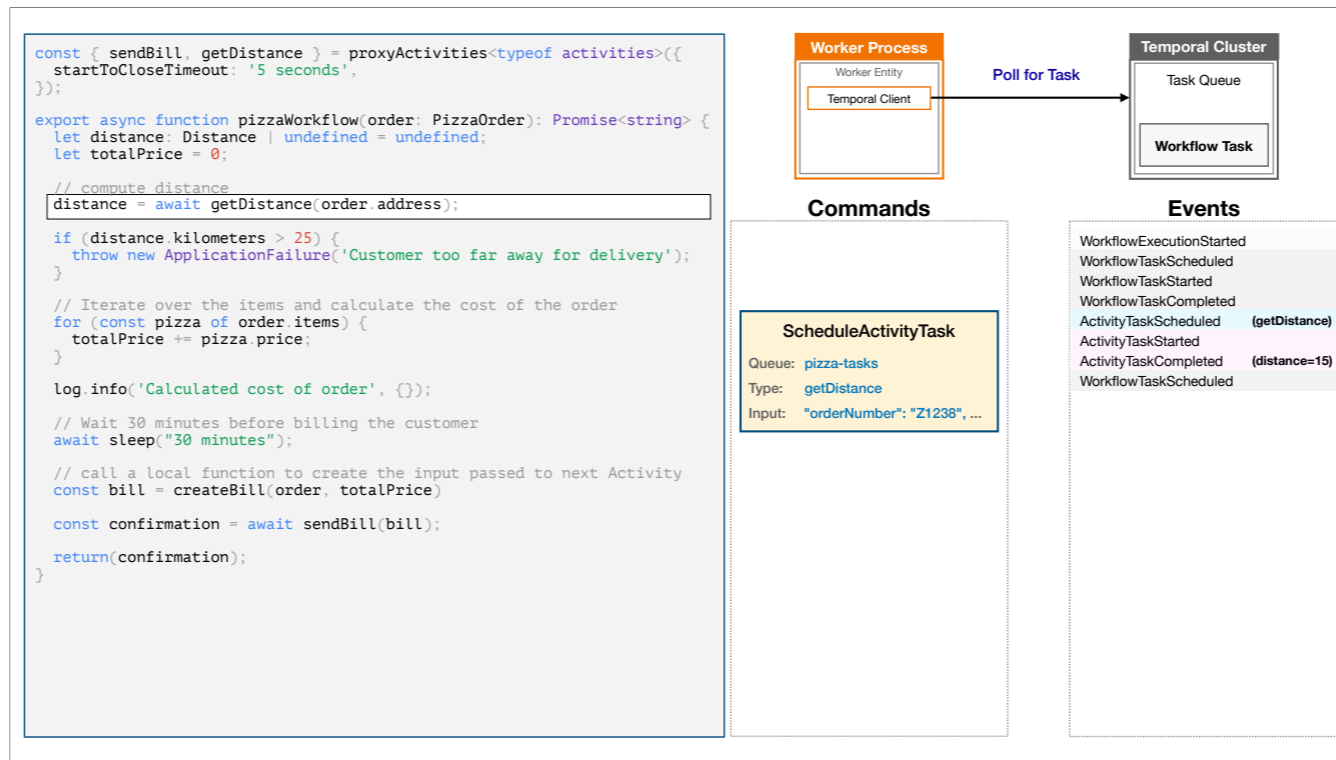
Let's suppose that it ultimately returns a value of `15`. When the function returns, the Worker notifies that cluster that the Activity Execution is complete.



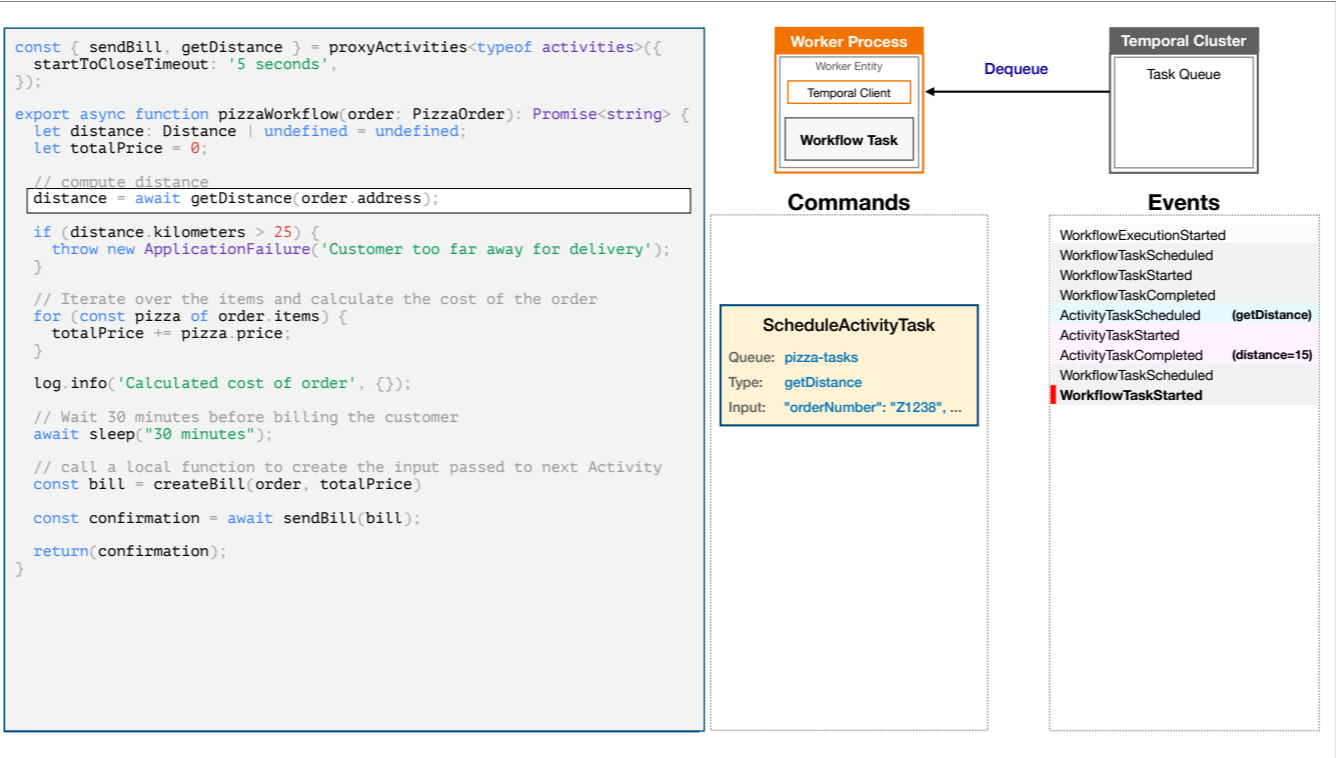
The cluster logs an `ActivityTaskCompleted` Event, which contains this result.



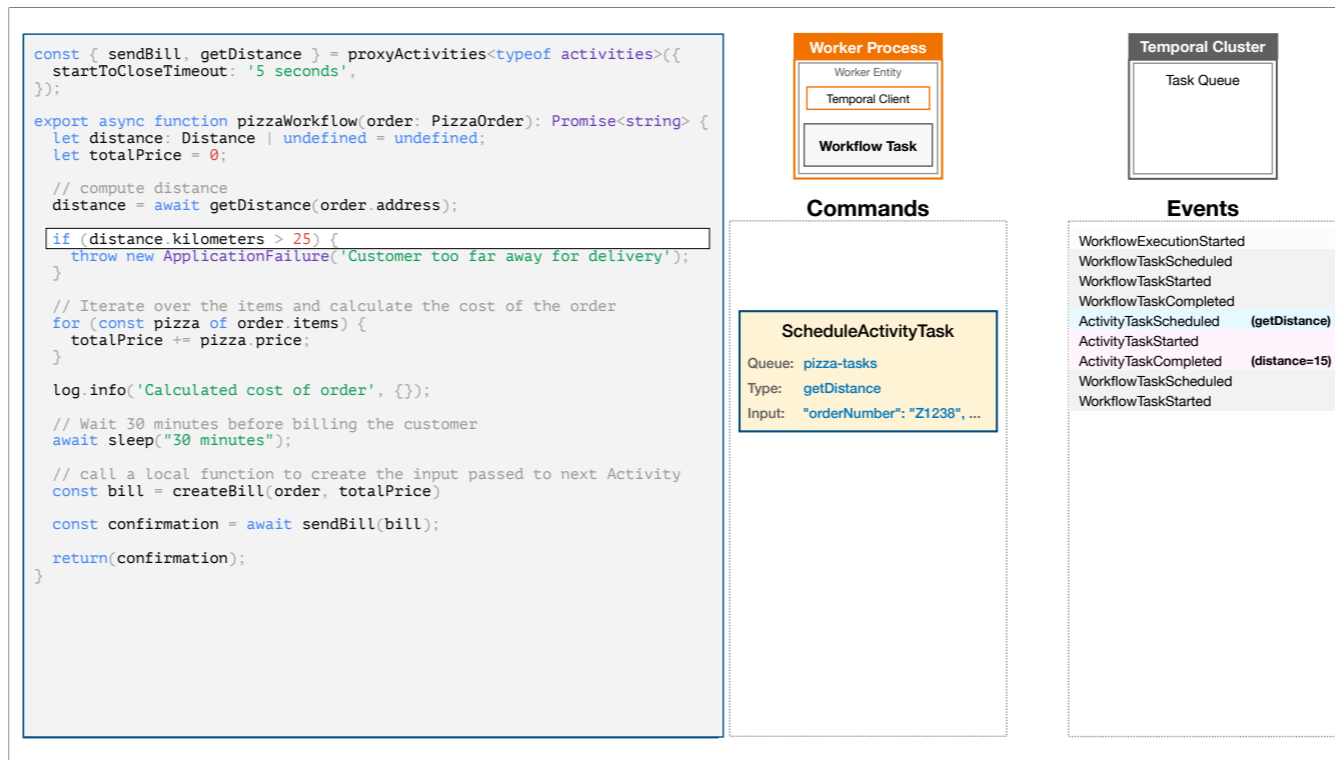
The cluster then adds another Workflow Task to drive the Workflow Execution forward.



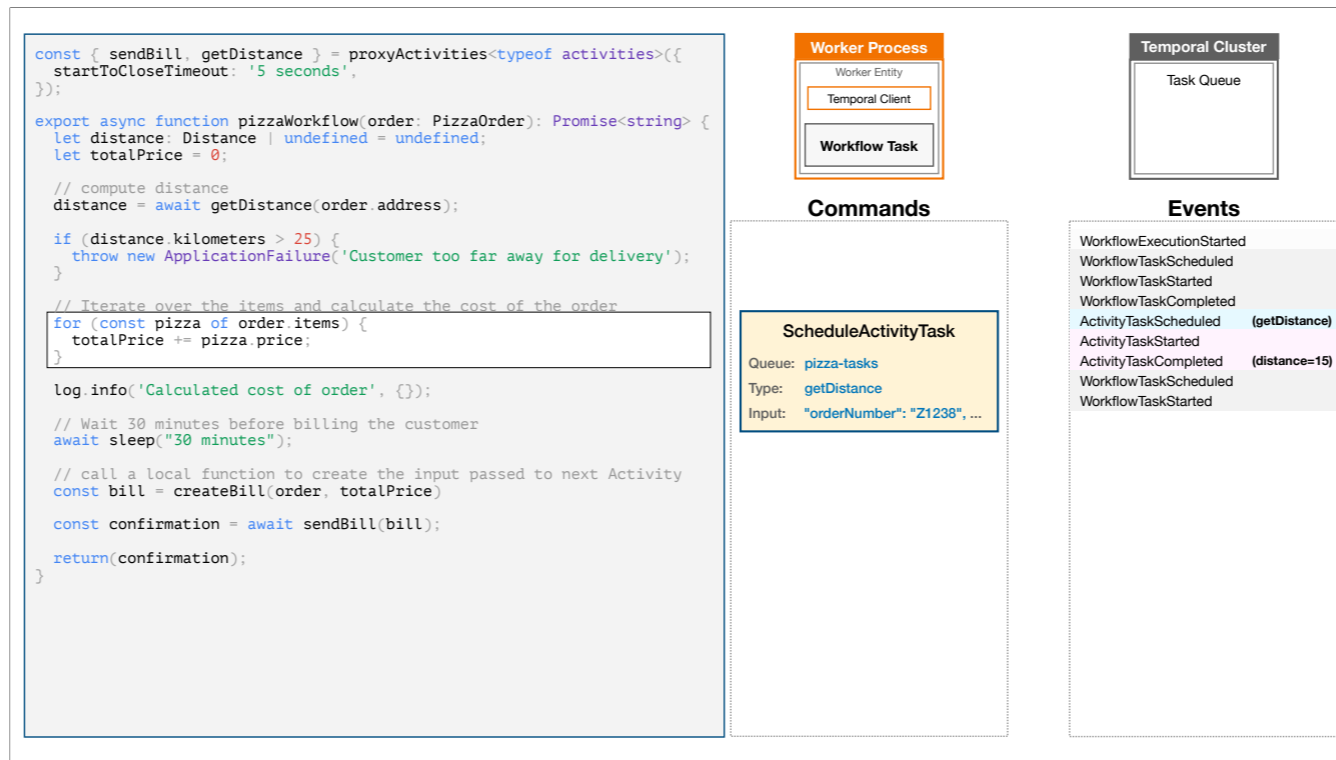
When the Worker polls, it's matched with this task,



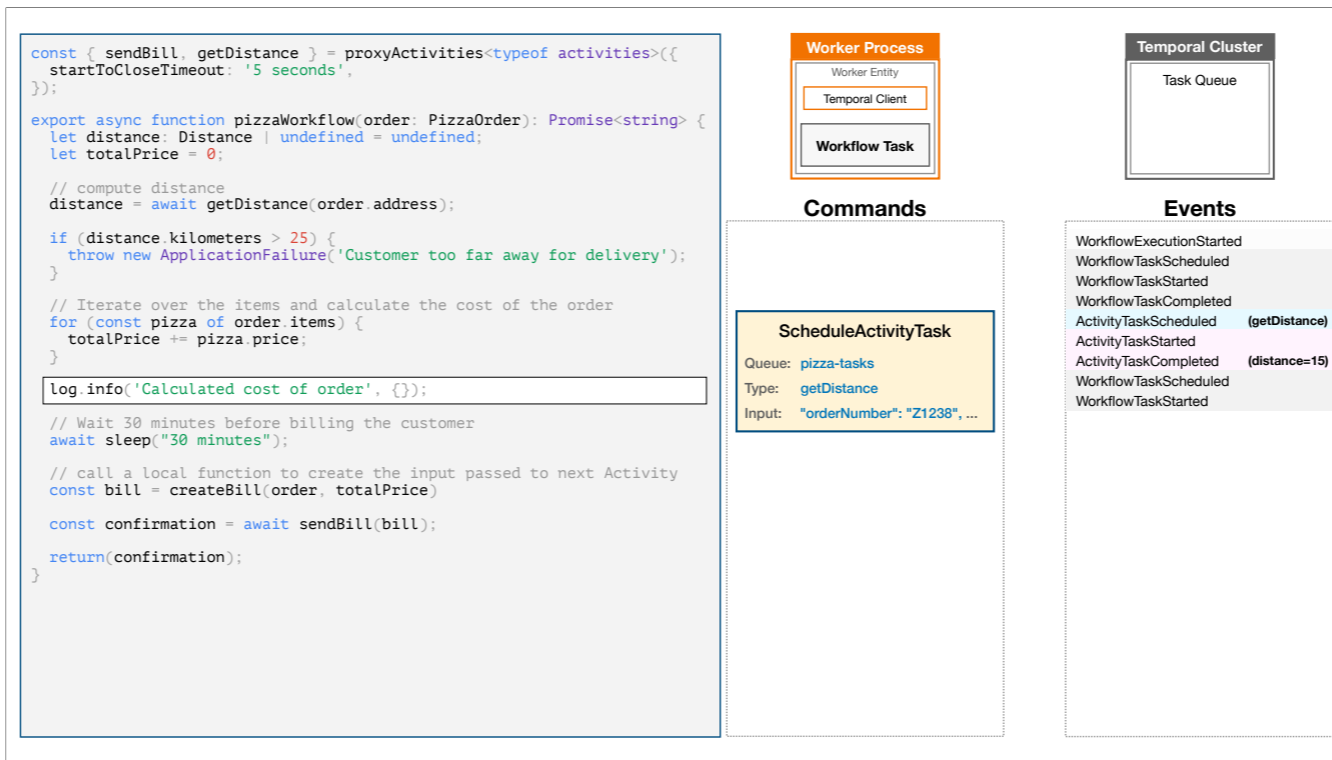
dequeues it,



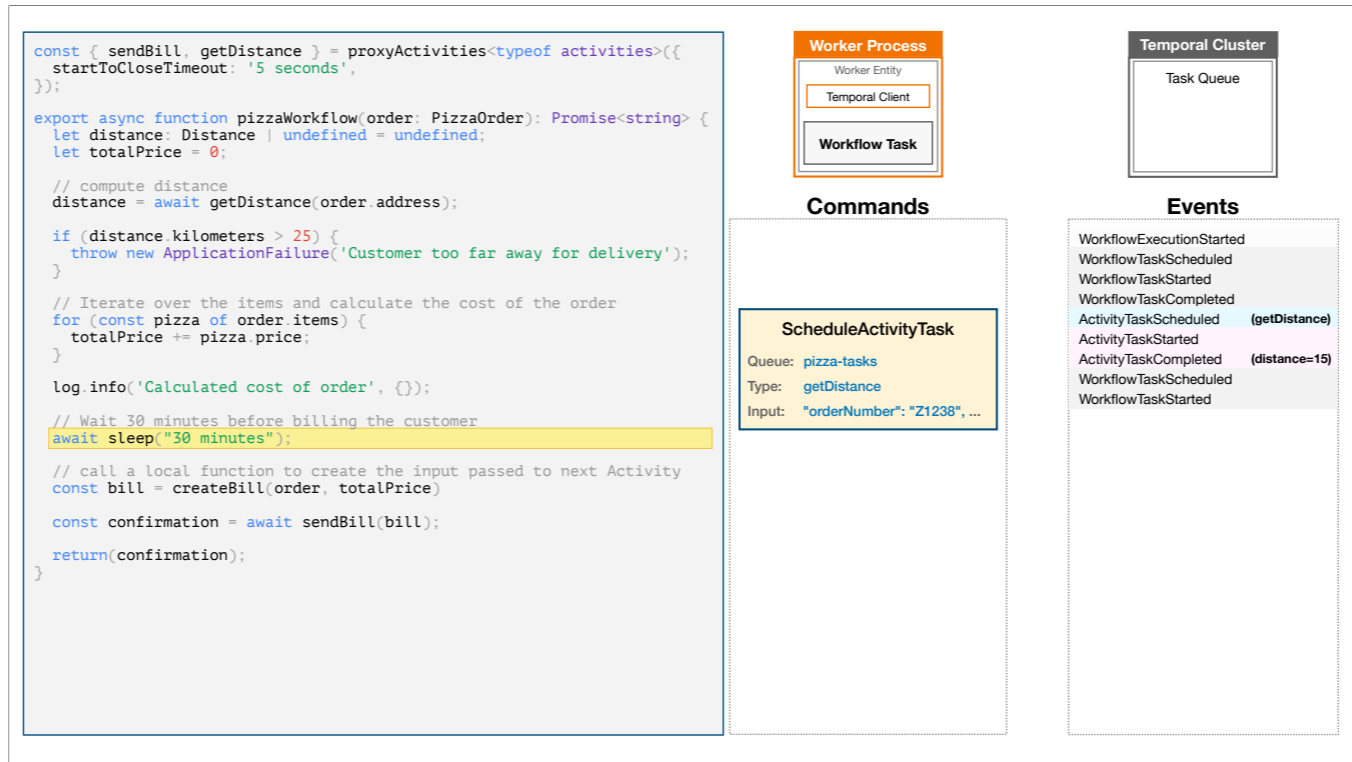
and resumes execution of the Workflow code.



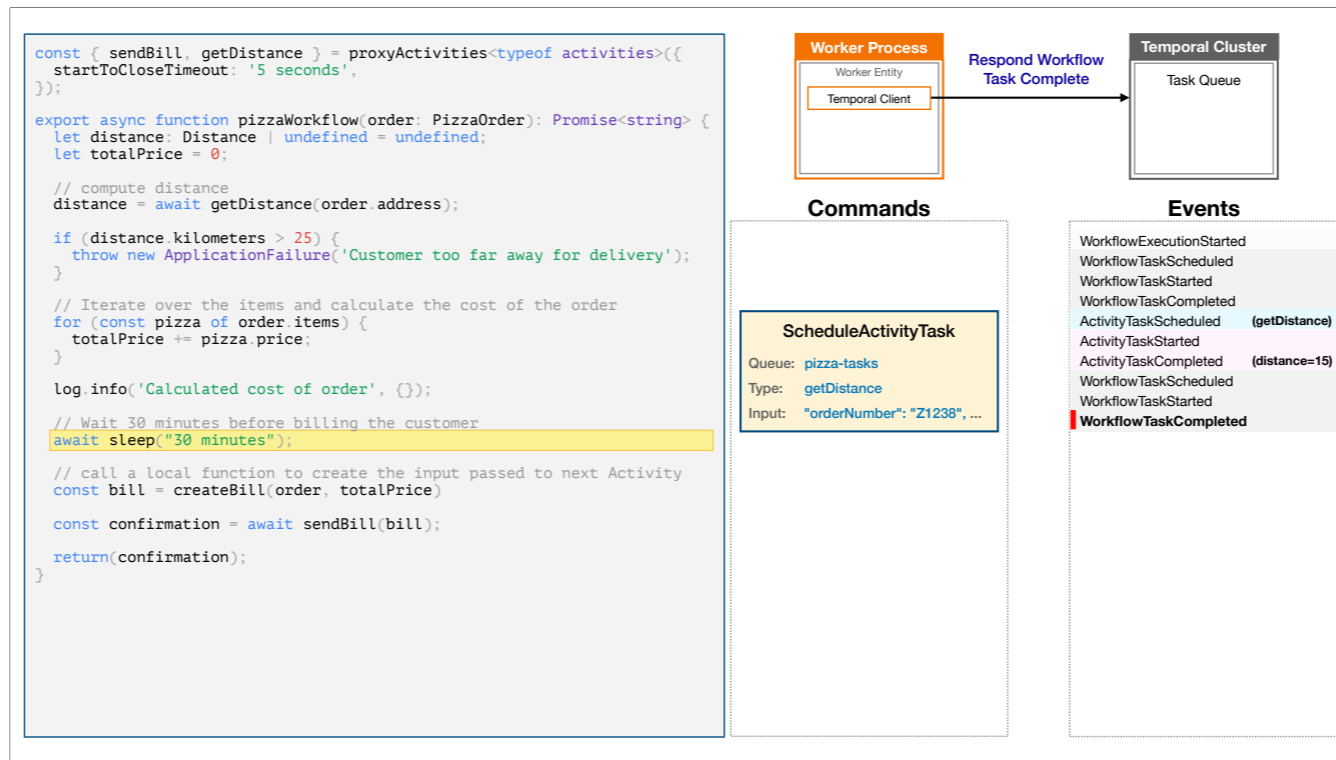
The price is totaled up, which happens in the Workflow. Once again, there's no communication with the Temporal Cluster.



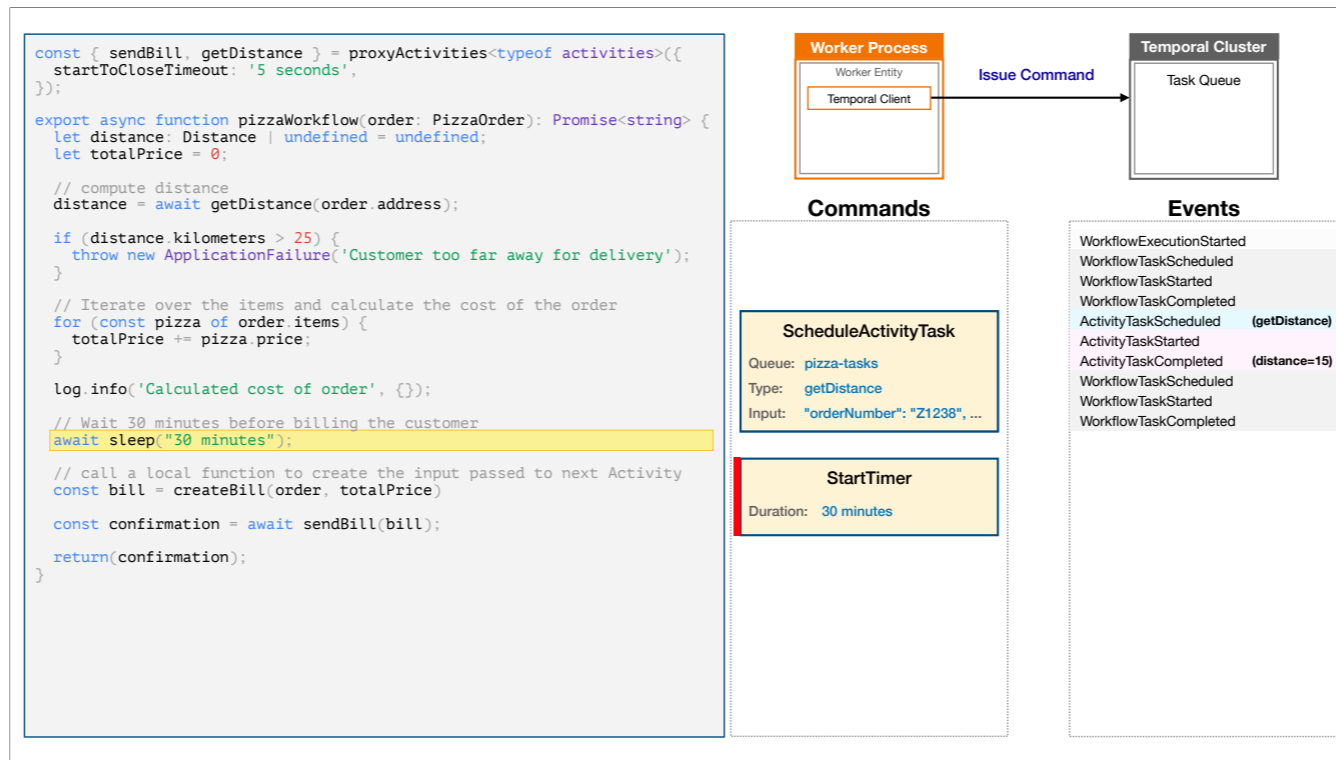
This log statement also happens locally.



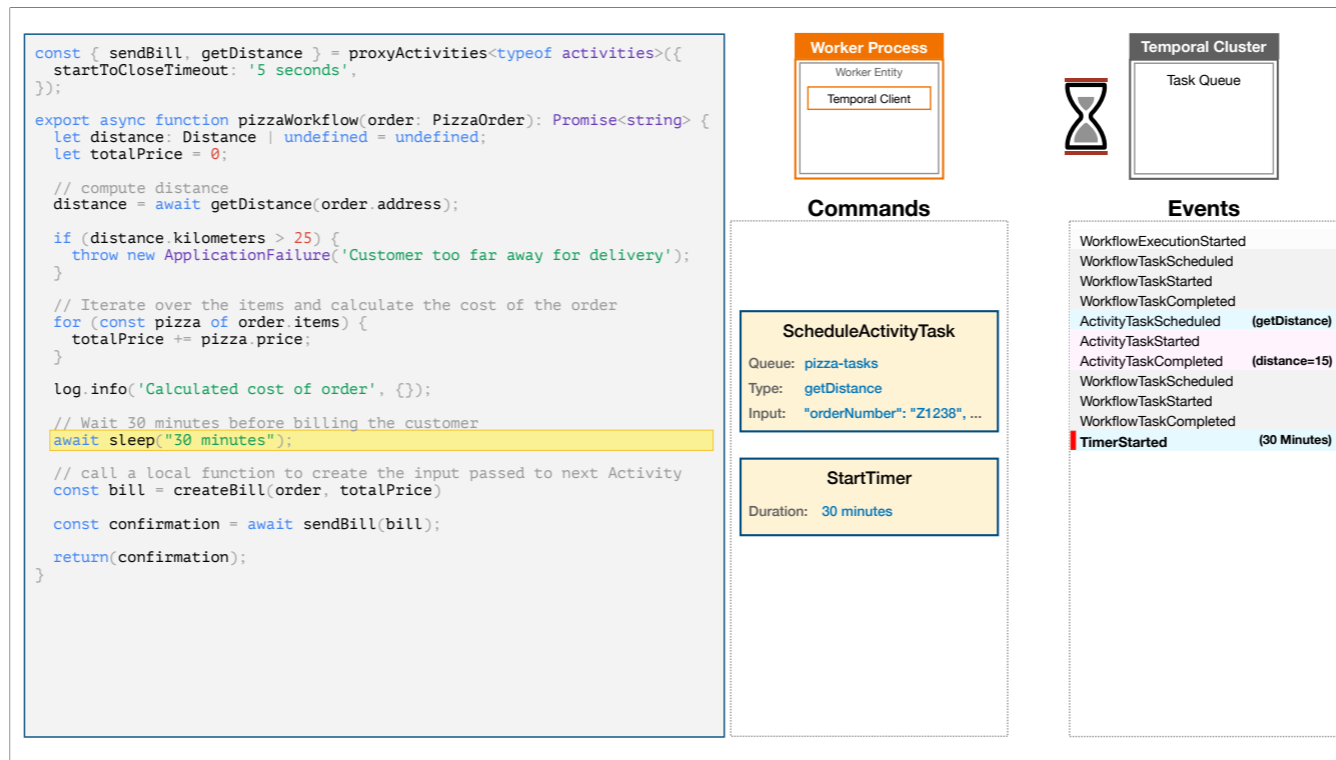
When it hits the `workflow.Sleep` call...



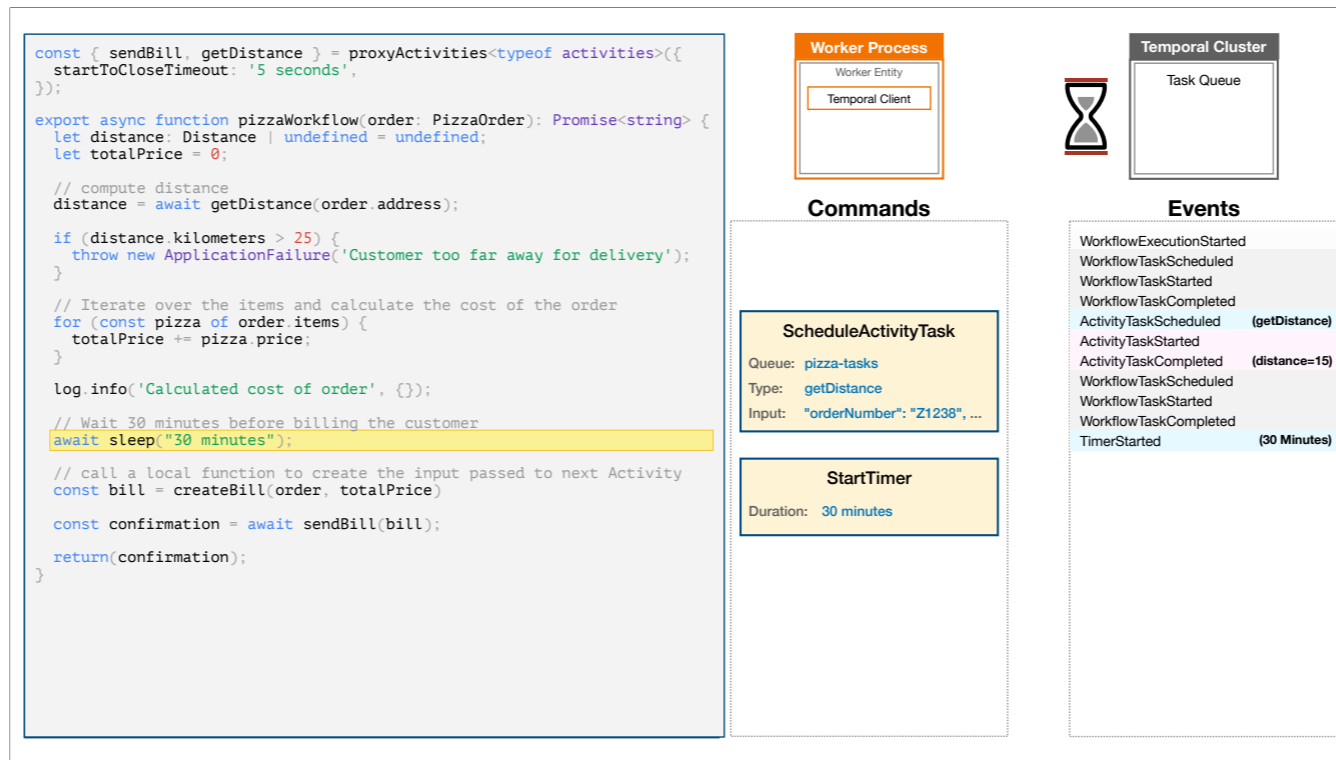
it completes the current Workflow Task...



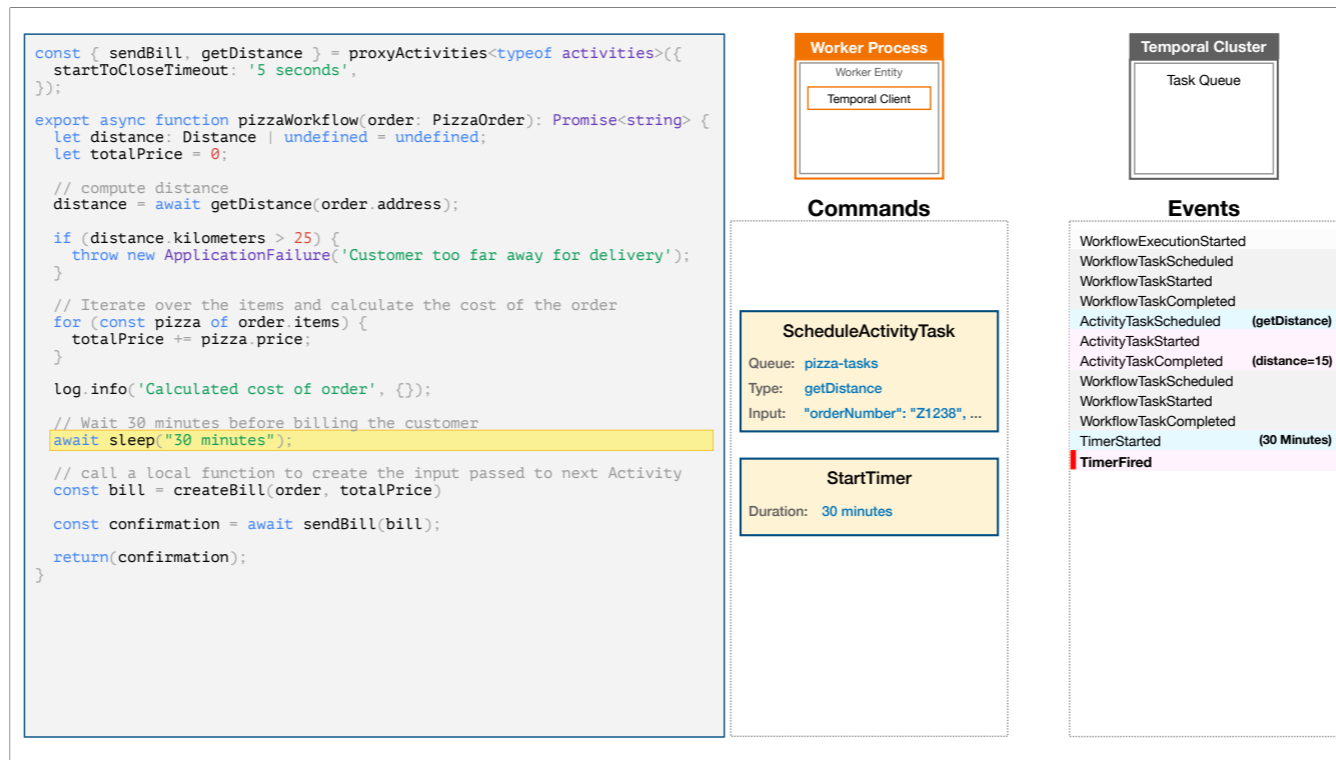
and issues a Command to the cluster, requesting it to set a Timer for 30 minutes.



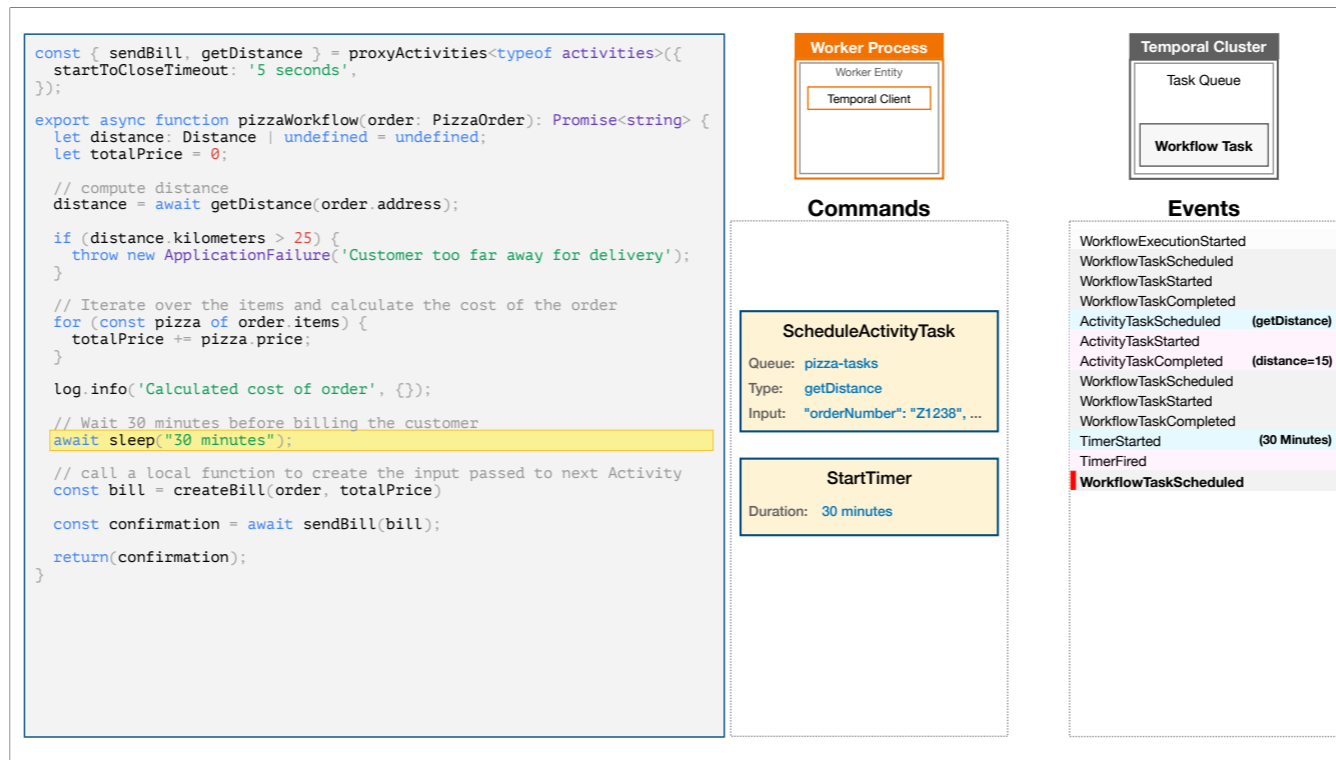
The cluster logs a `TimerStarted` Event in response. The Workflow cannot progress until that Timer fires, so the cluster does not queue any new Tasks for this Workflow Execution until that happens.



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.



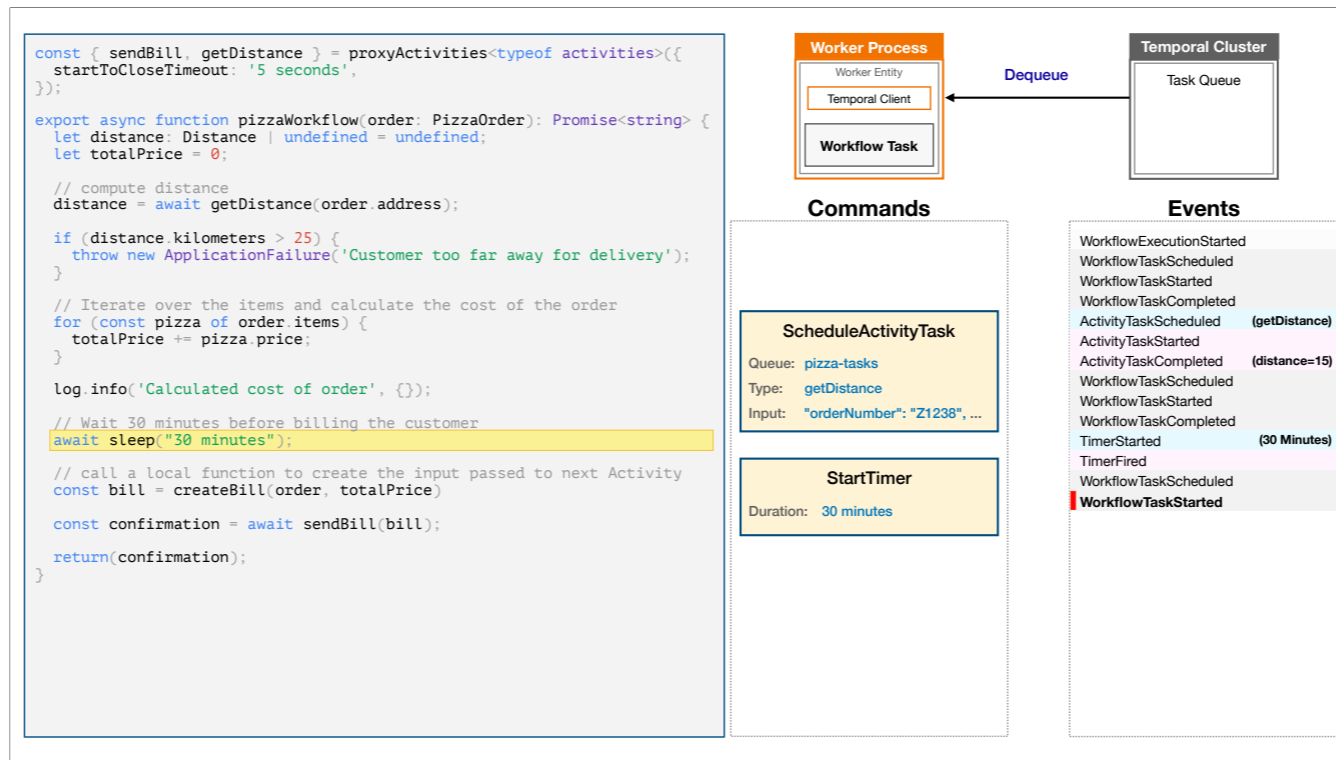
After 30 minutes has elapsed, the Timer fires, and the cluster logs a `TimerFired` Event. How does the Worker know that it can continue running the Workflow code now?



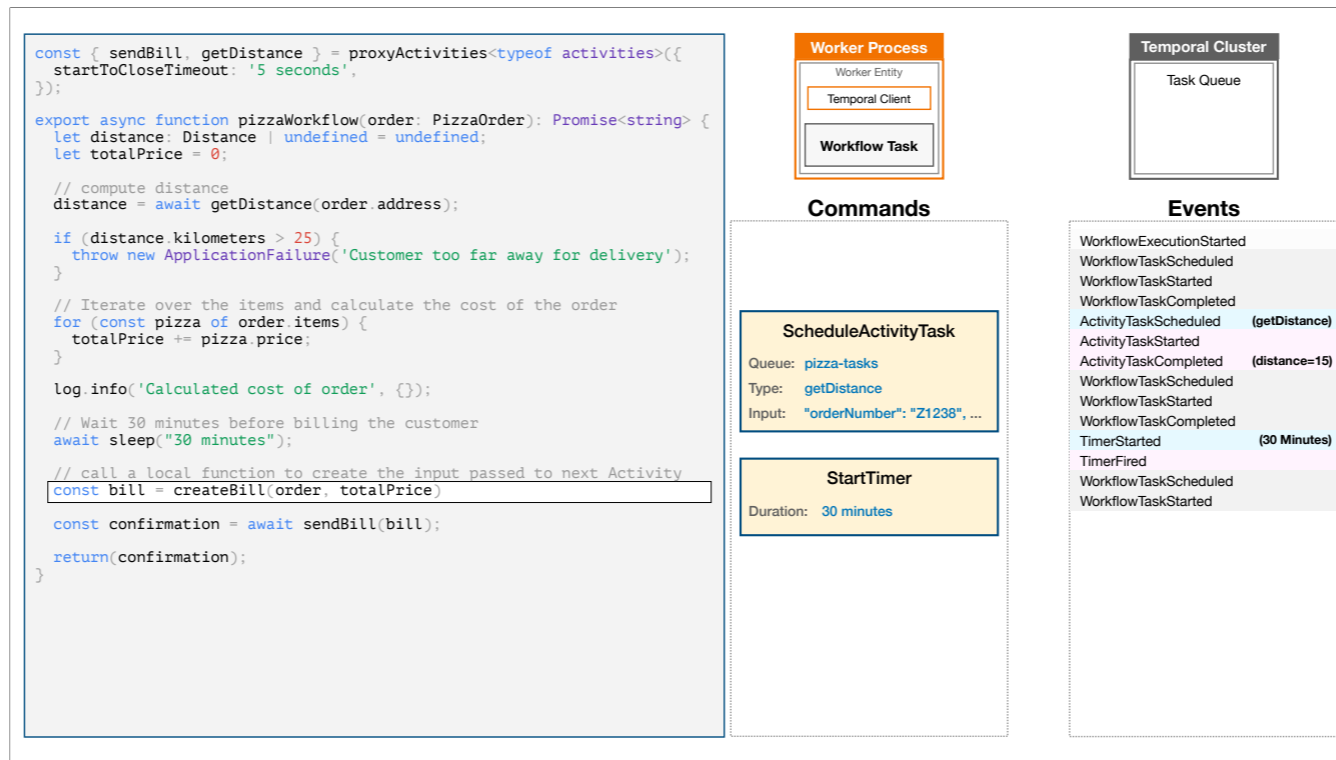
It's because the cluster now adds a new Workflow Task to the queue,



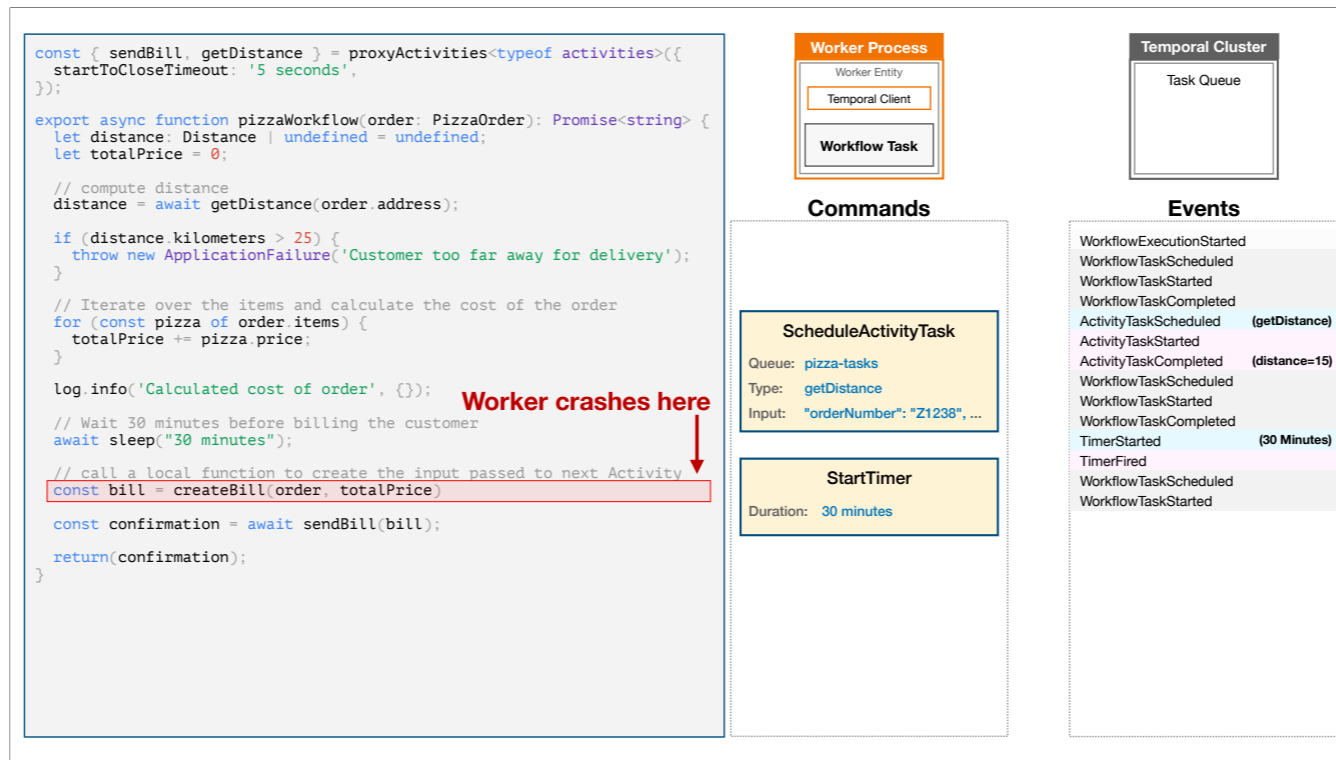
which the Worker will find when it polls again.



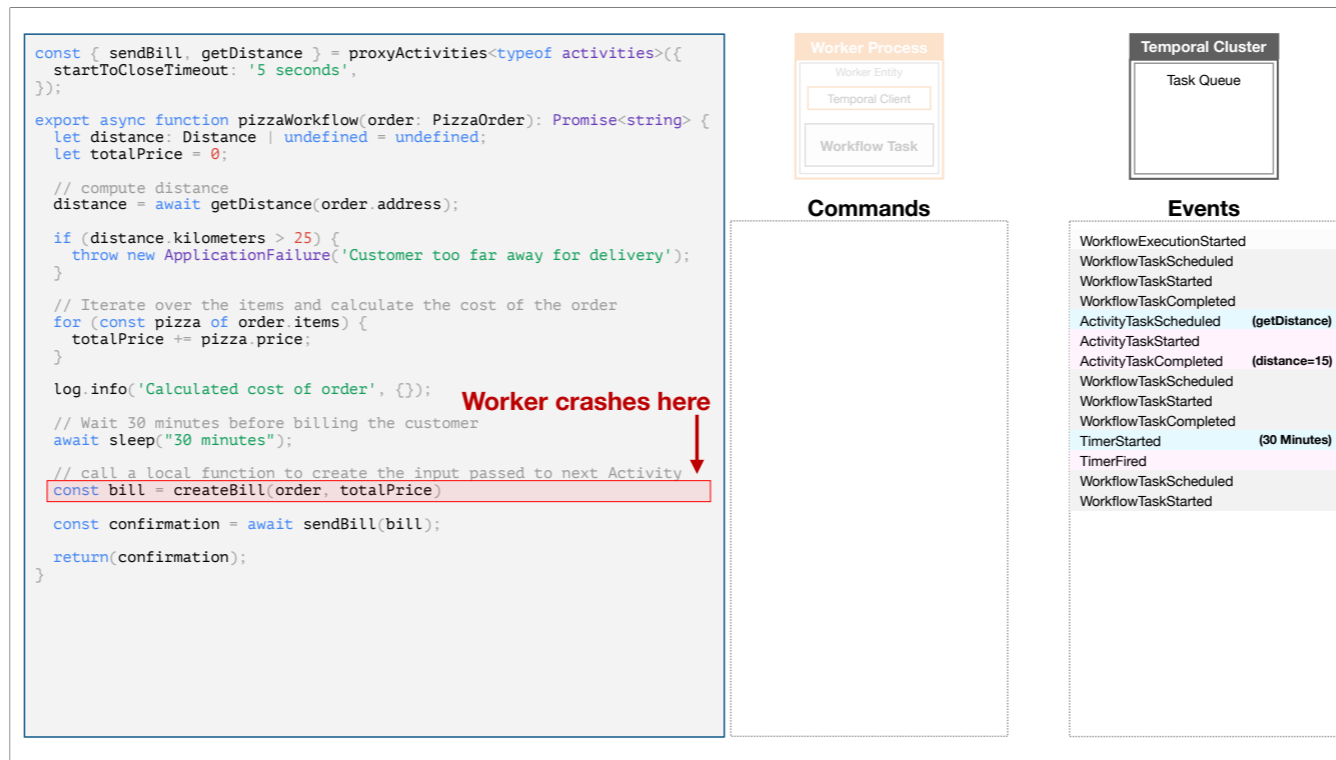
After it accepts this Task,



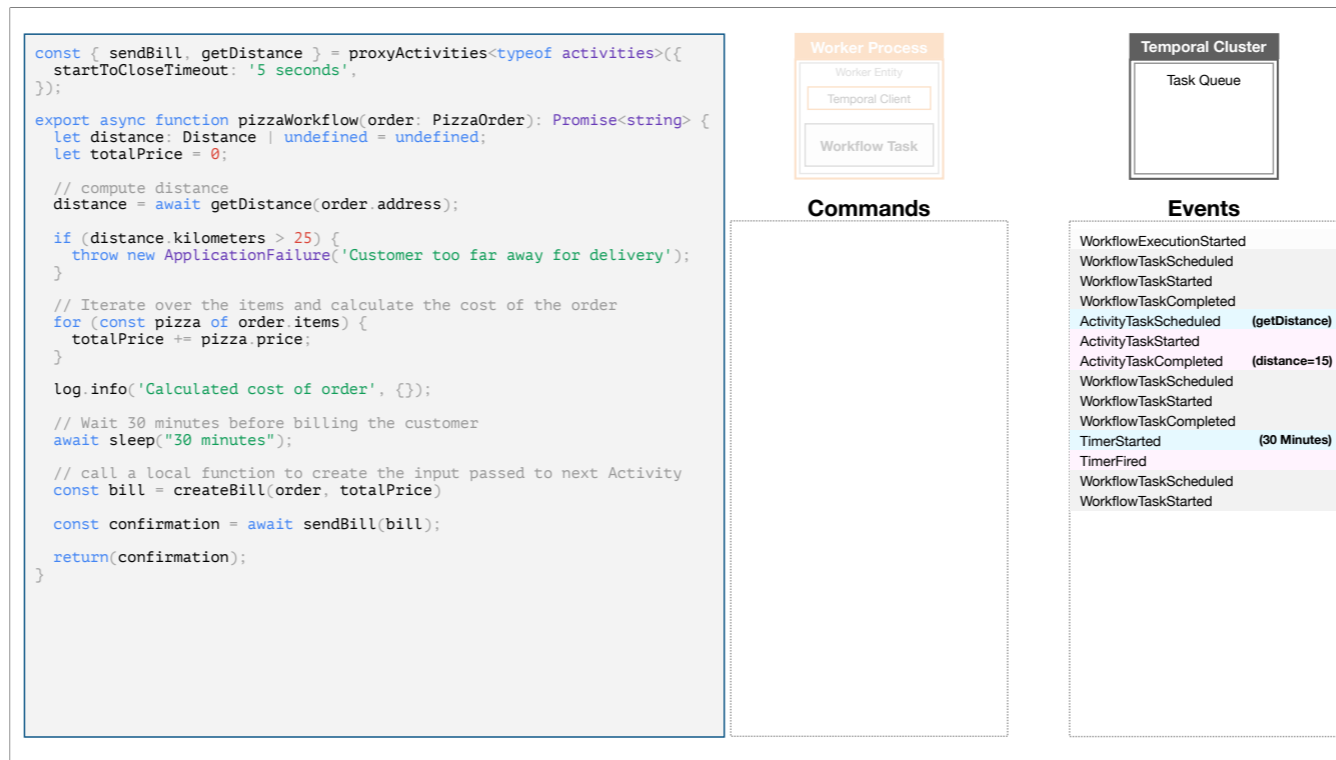
it continues execution of the Workflow code.



However, let's suppose that the Worker happens to crash right here. How does Temporal recover the state of the Workflow?



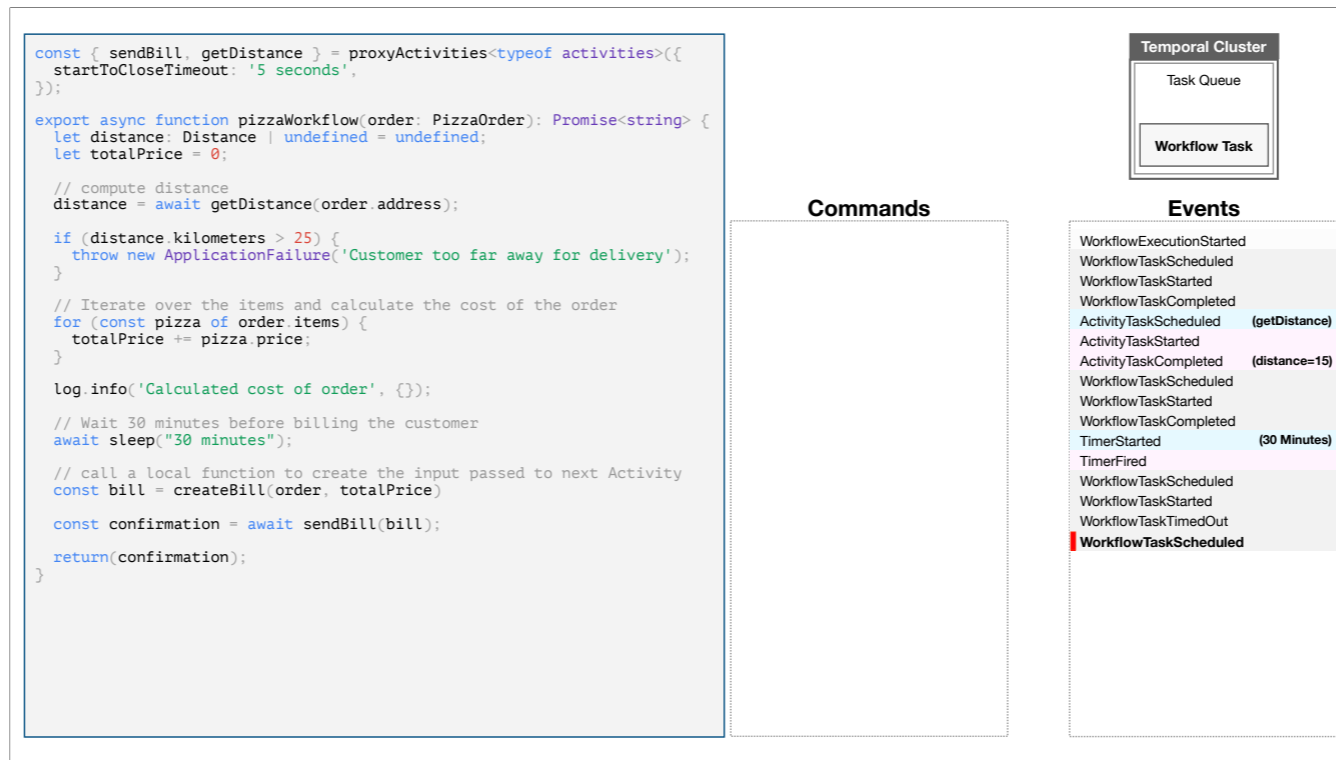
Workers are external to the cluster and operate with autonomy. They long-poll the Task Queue, but only when seeking work to perform. They retrieve tasks from the queue, rather than being assigned tasks directly by the cluster.



However, once a Worker has accepted a Task, it is expected to complete that task within a predefined duration, known as a Timeout.



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.

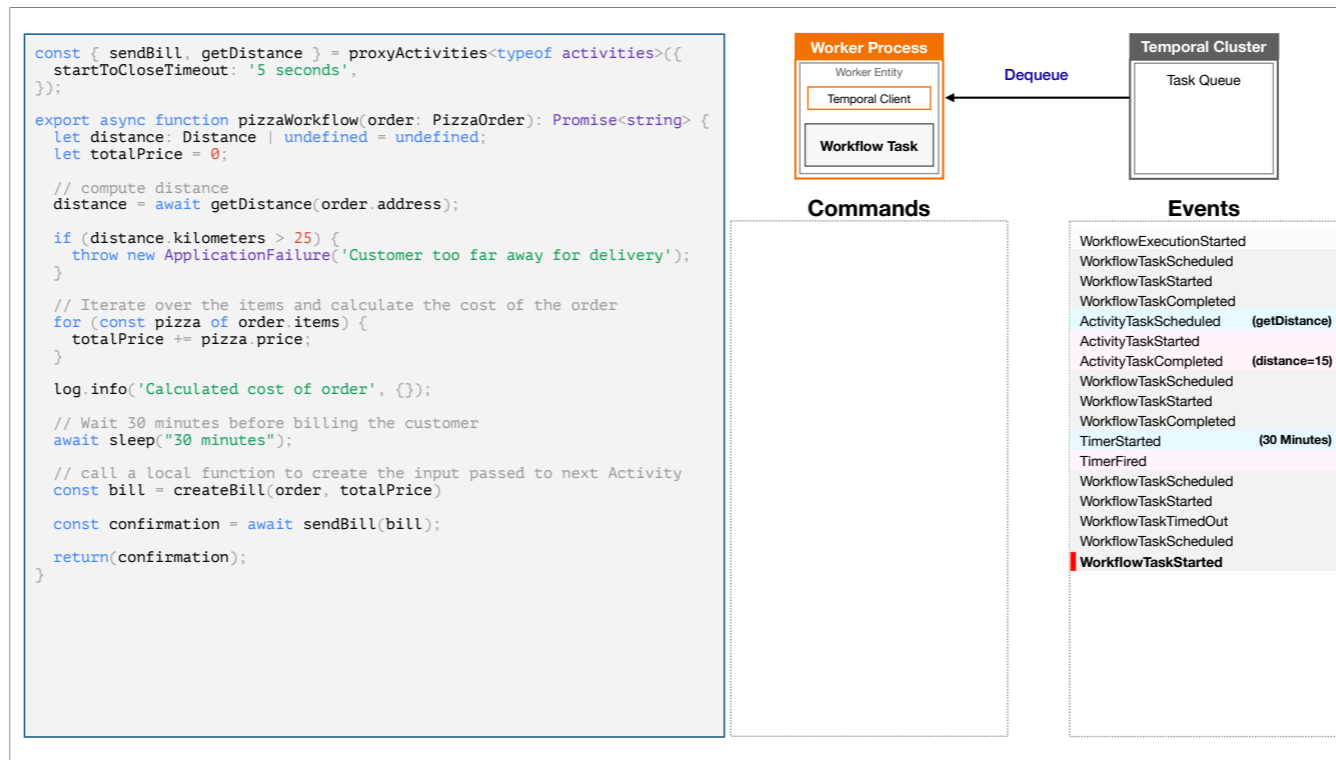


Therefore, if the Worker failed to complete this Workflow Task within that time, the cluster assumes that the Worker has gone down, and will schedule a new Workflow Task.

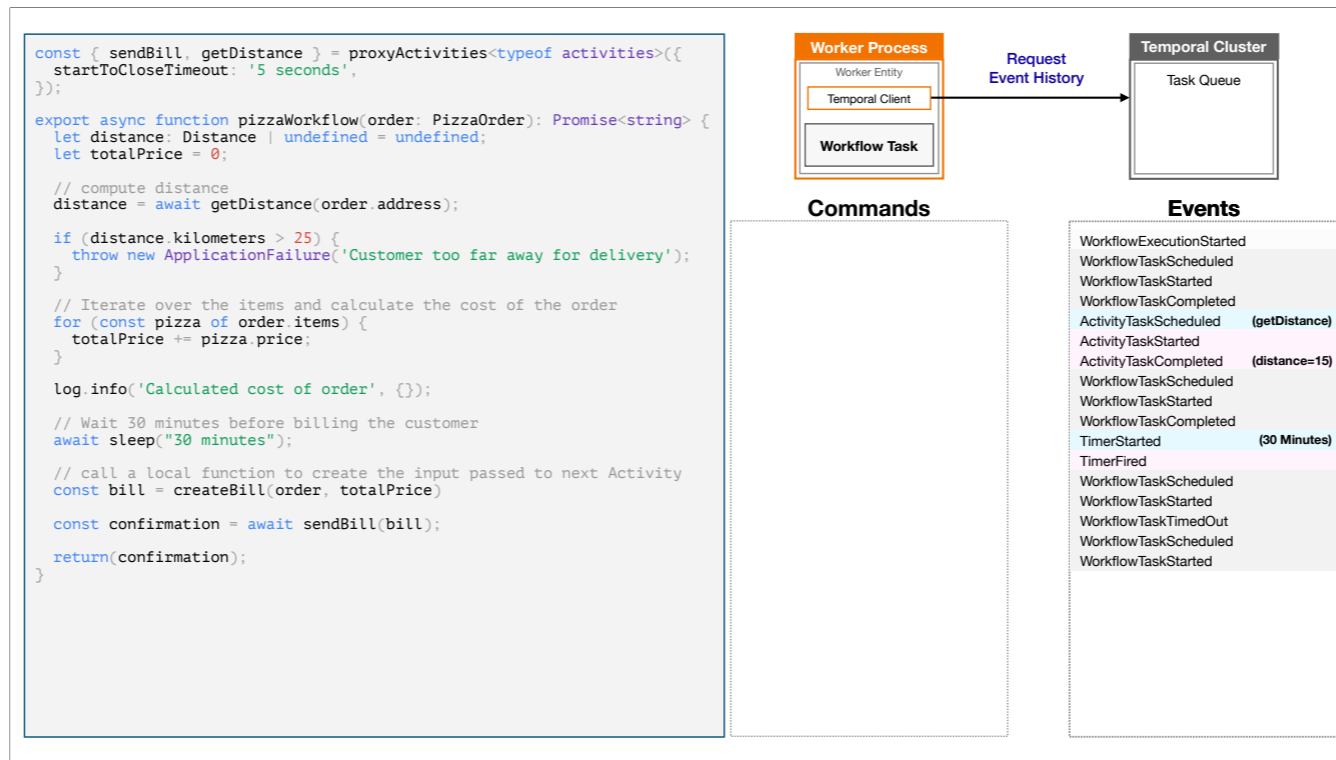
Unlike the **original** Workflow Task, this one is not added to the "sticky queue" used to favor the Worker that previously accepted Workflow Tasks for this execution. Instead, it's placed into the Task Queue specified when Workflow Execution began, which means that it will be available to any available Worker polling this Task Queue.



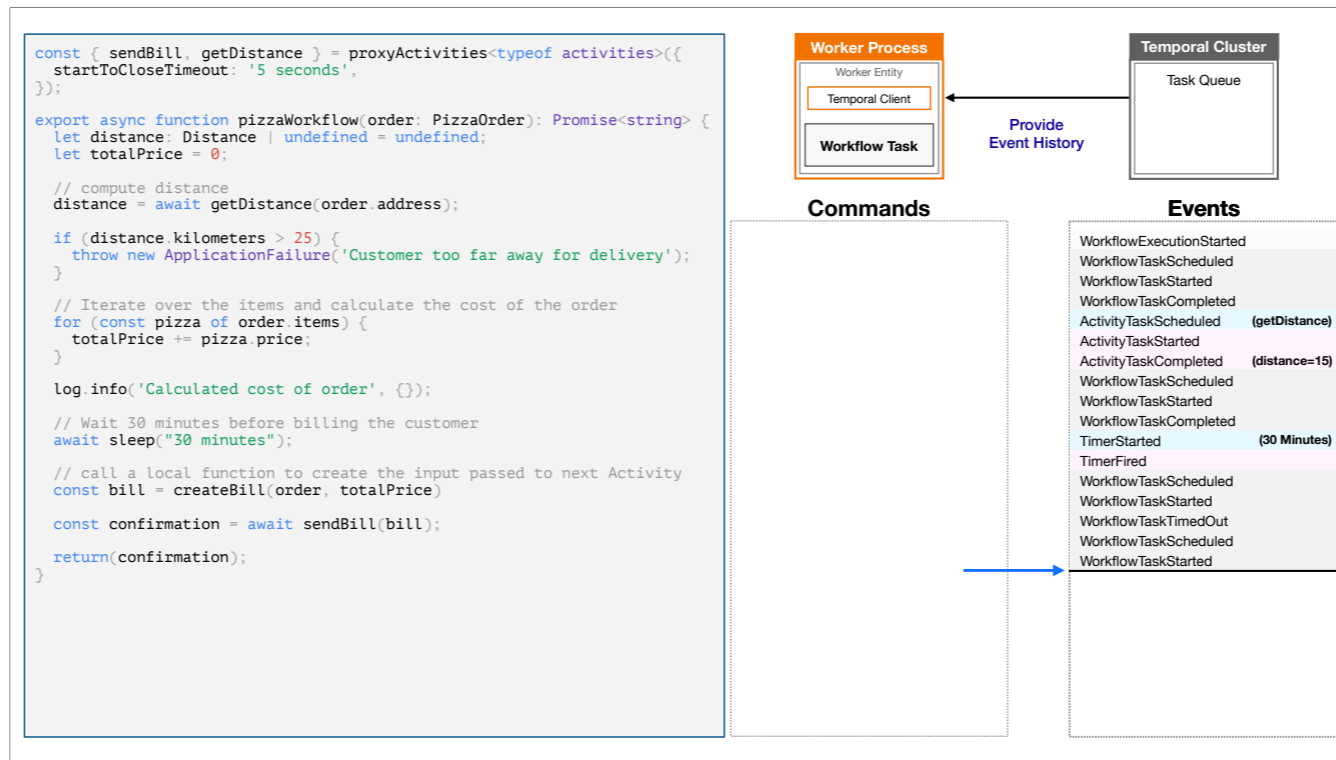
This Task will remain in the queue until another Worker polls and accepts it. That might be done by another one that's already running in the Worker fleet or by a new Worker Process created by restarting the one that crashed.



In either case, the Worker accepts the task...

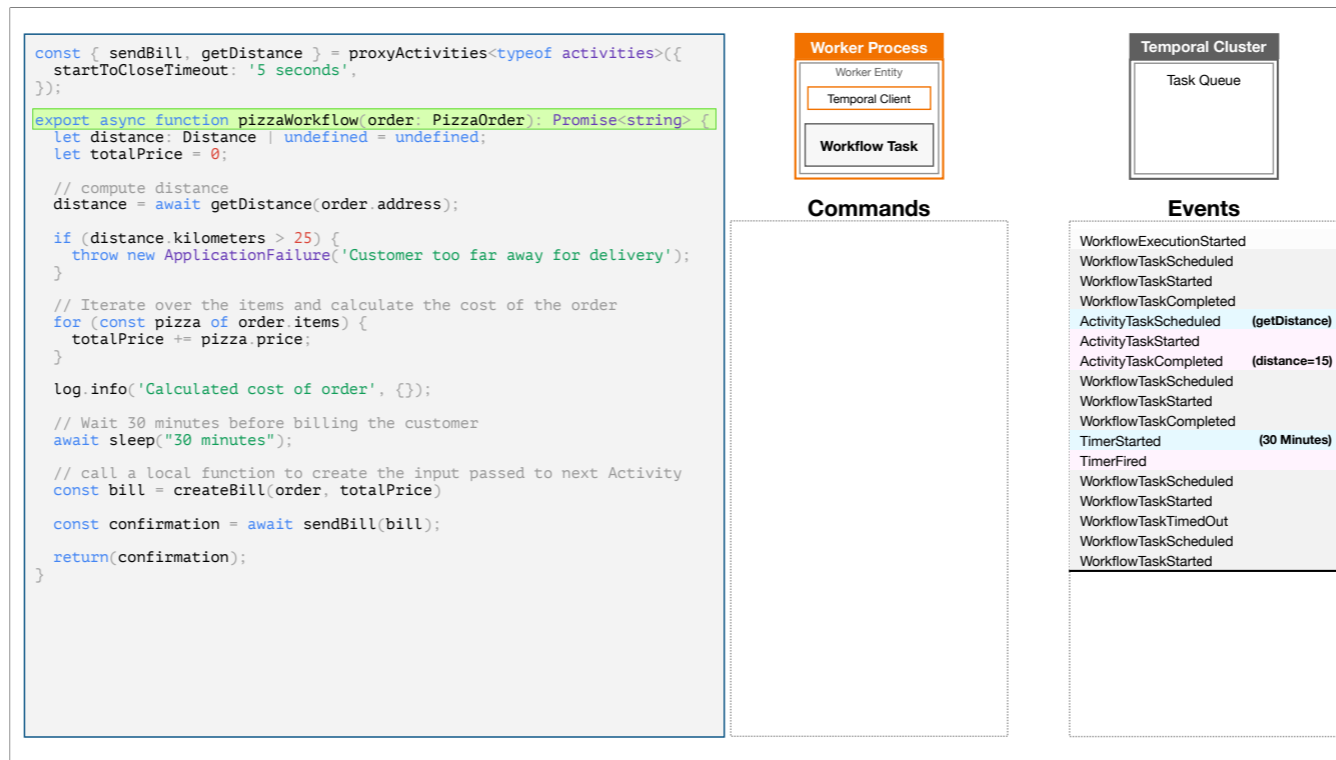


The Worker will need the current Event History for this execution, so it requests it from the cluster.



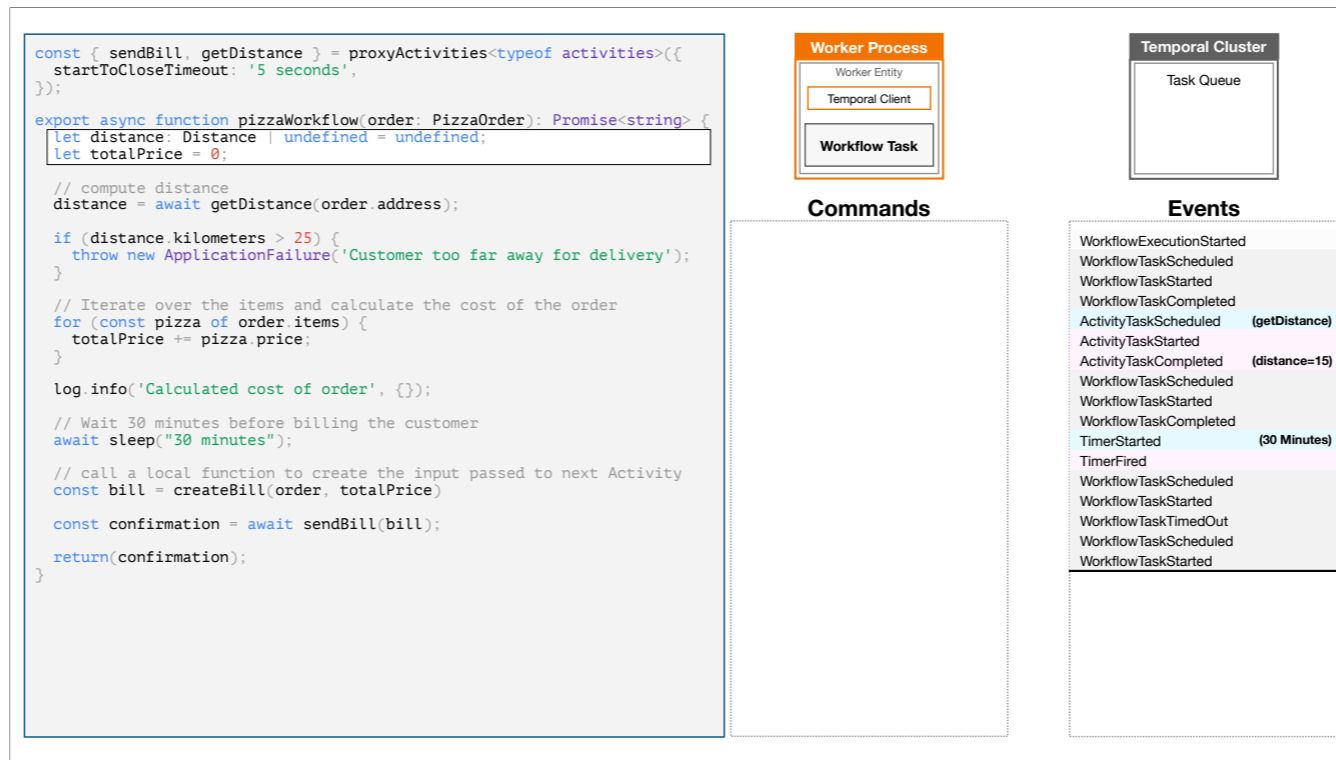
The Worker streams the Event History from the cluster.

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.

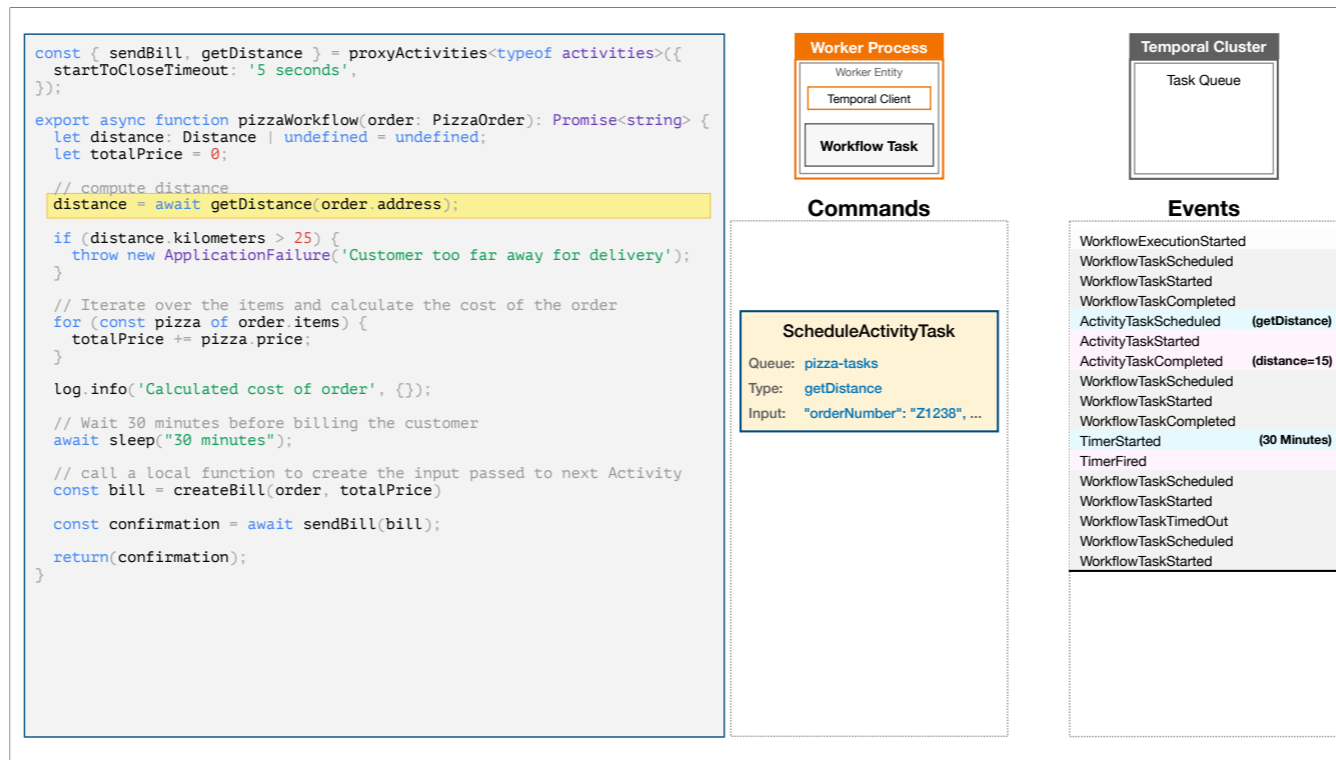


The Worker then begins a re-execution of the code, using the same input, which was stored in the `WorkflowExecutionStarted` Event.

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.

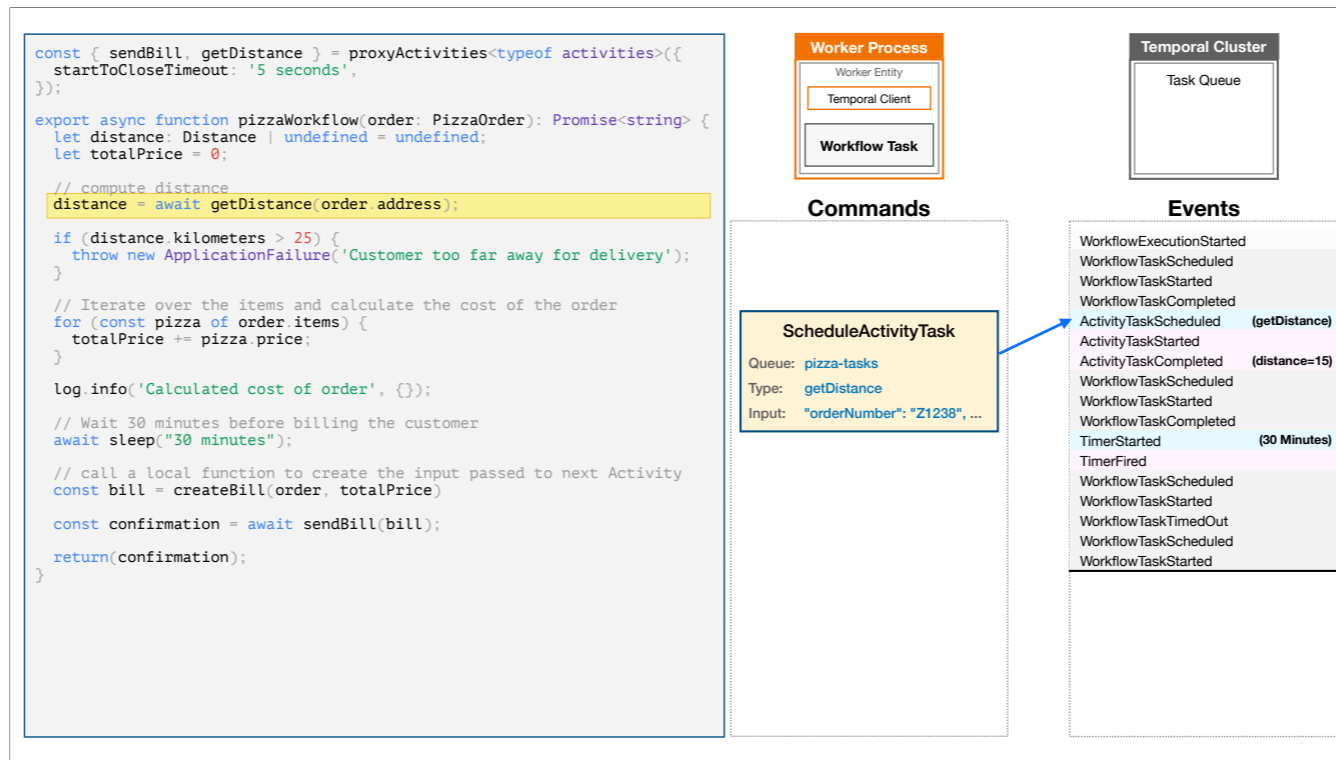


By the way, because the Workflow code is deterministic, the state of all variables encountered so far is identical to what it was before the crash.

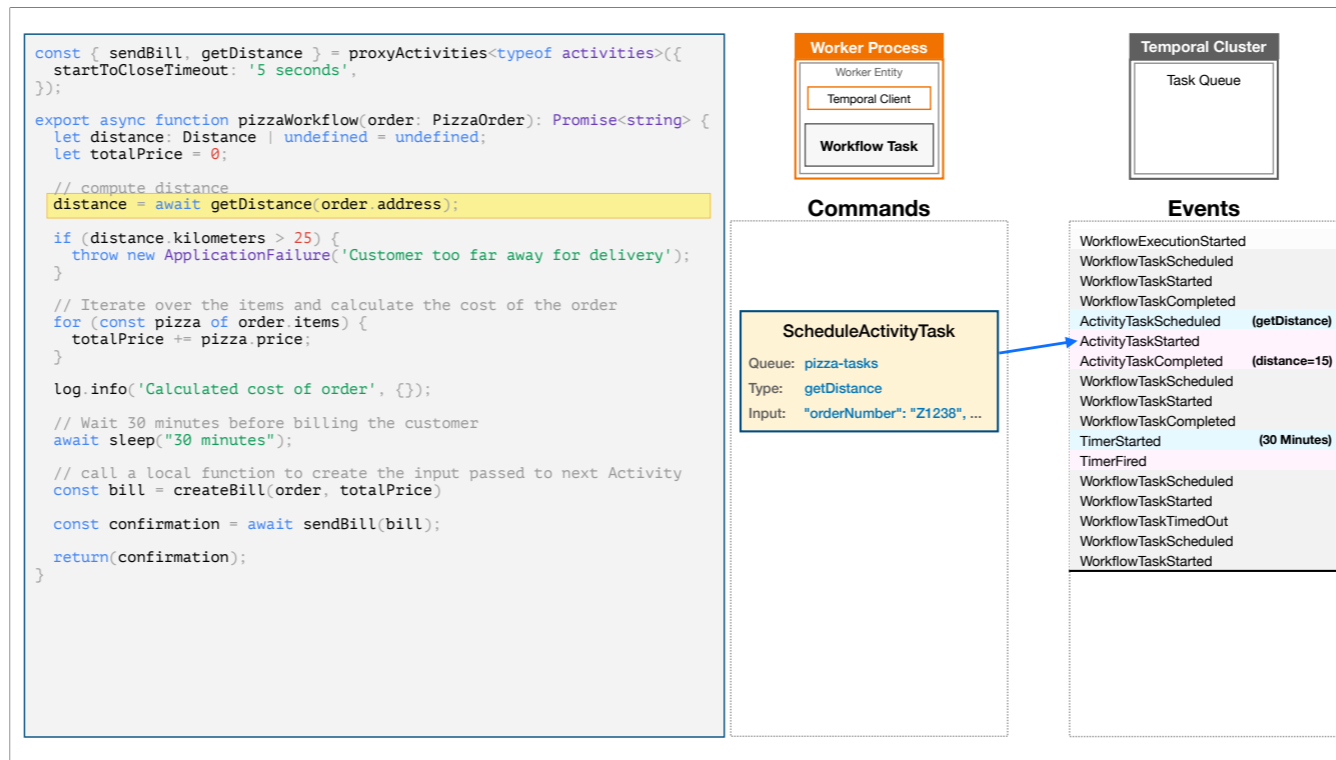


When it reaches the call to schedule the Activity, it creates a Command, but does not issue it to the cluster.

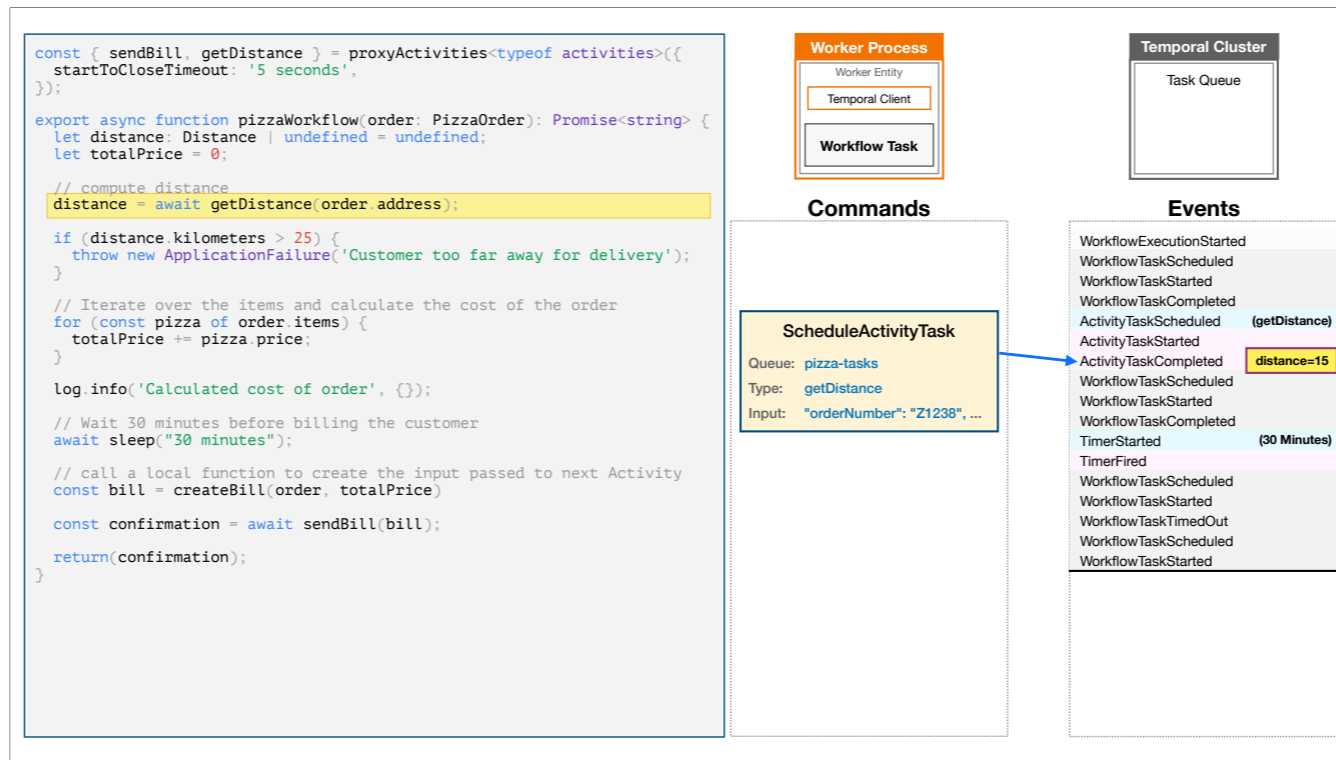
Instead, it inspects the Event History and finds three Events related to this Activity.



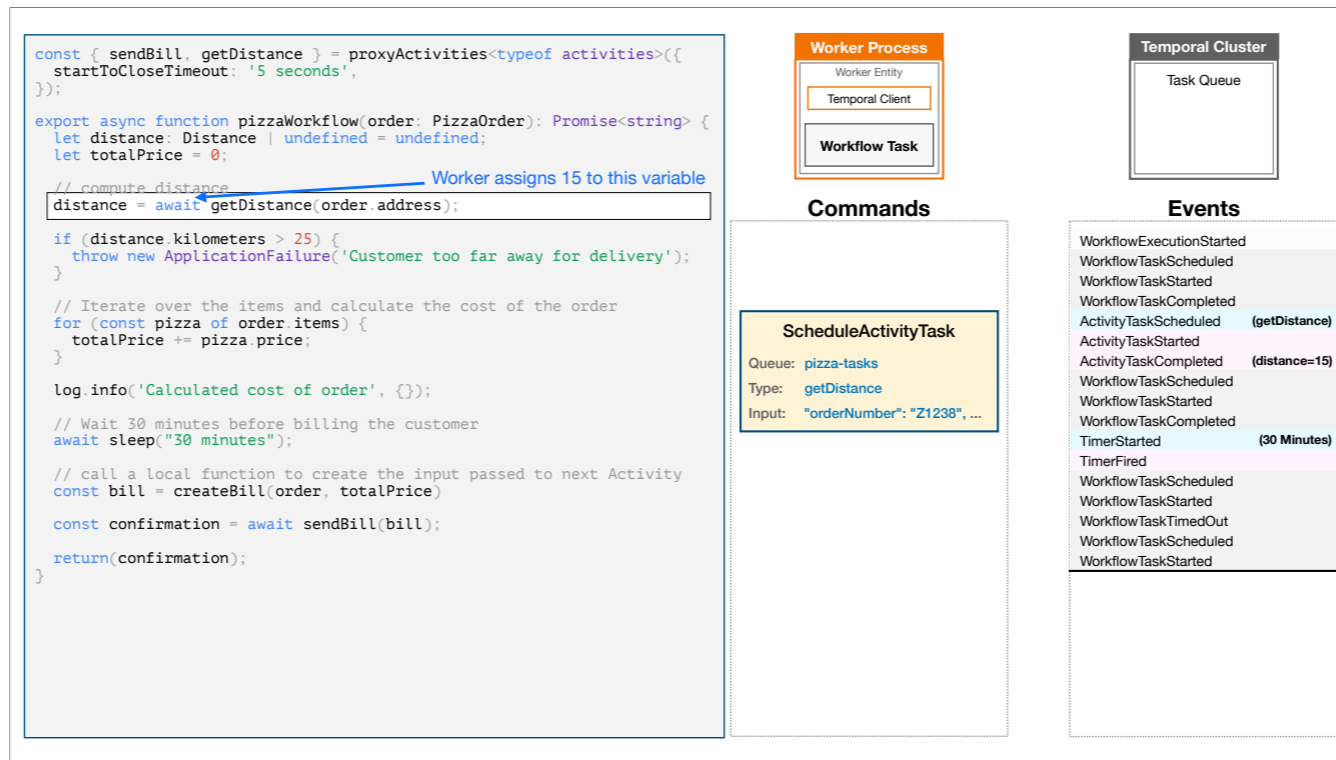
The first one indicates that the Task was previously scheduled by the cluster,



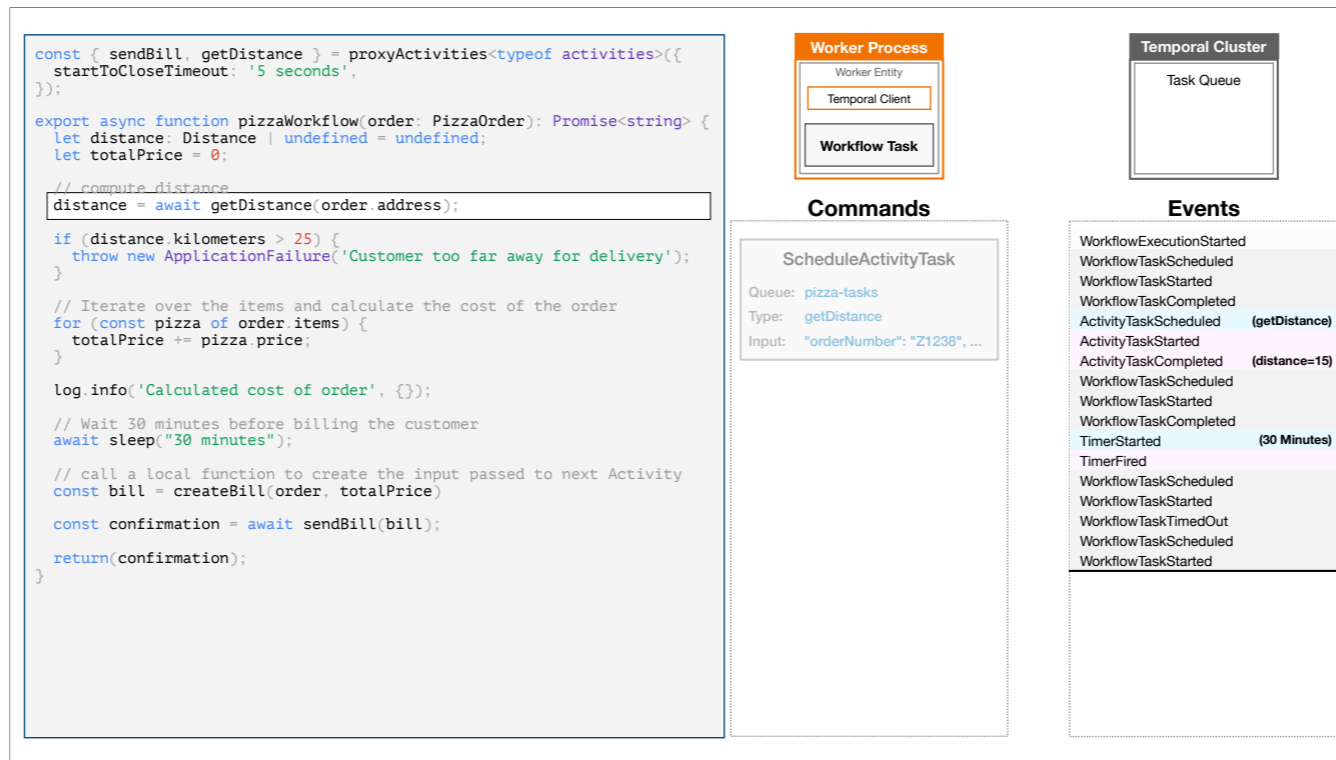
The second indicates that a Worker dequeued the Task,



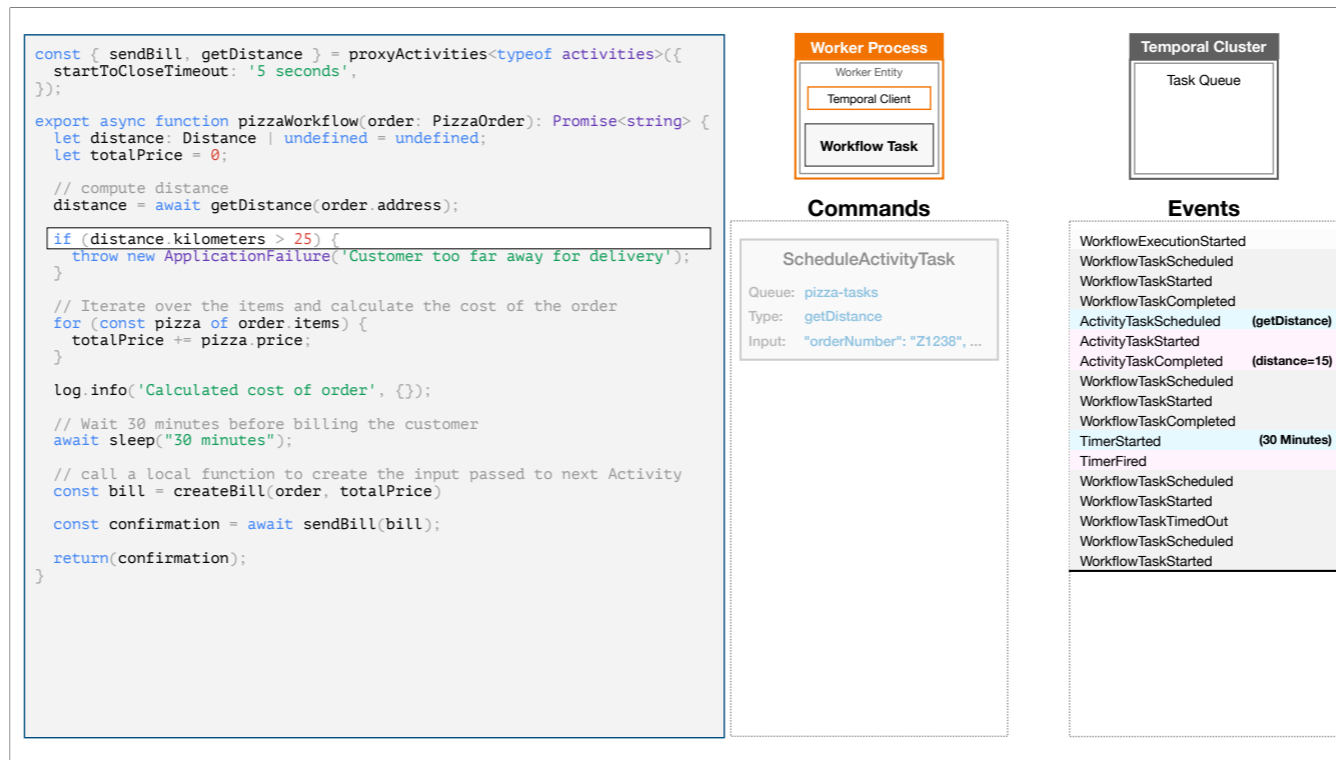
and the third indicates that the Worker successfully completed the Task for the `getDistance` Activity, having returned a value of `15`.



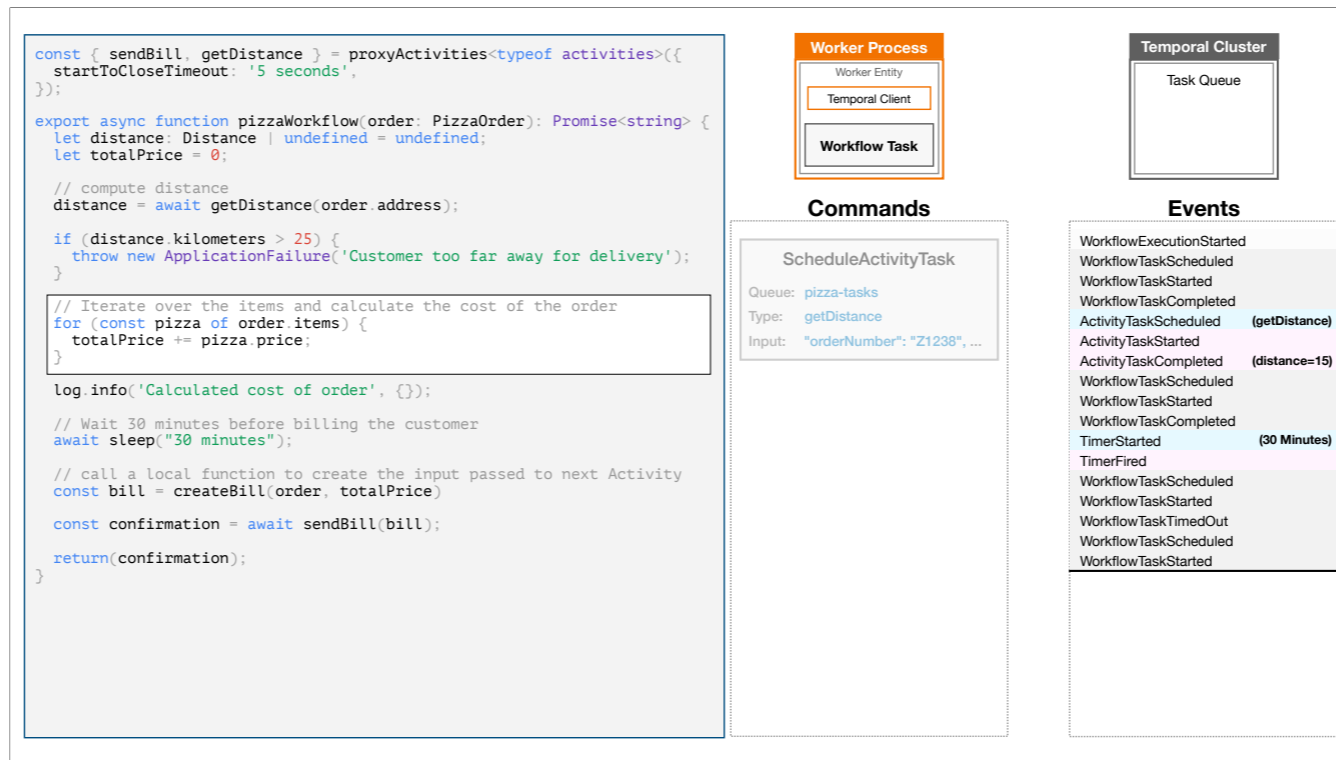
Instead of executing the Activity again during replay, the Worker assigns the value returned by the __previous__ execution.



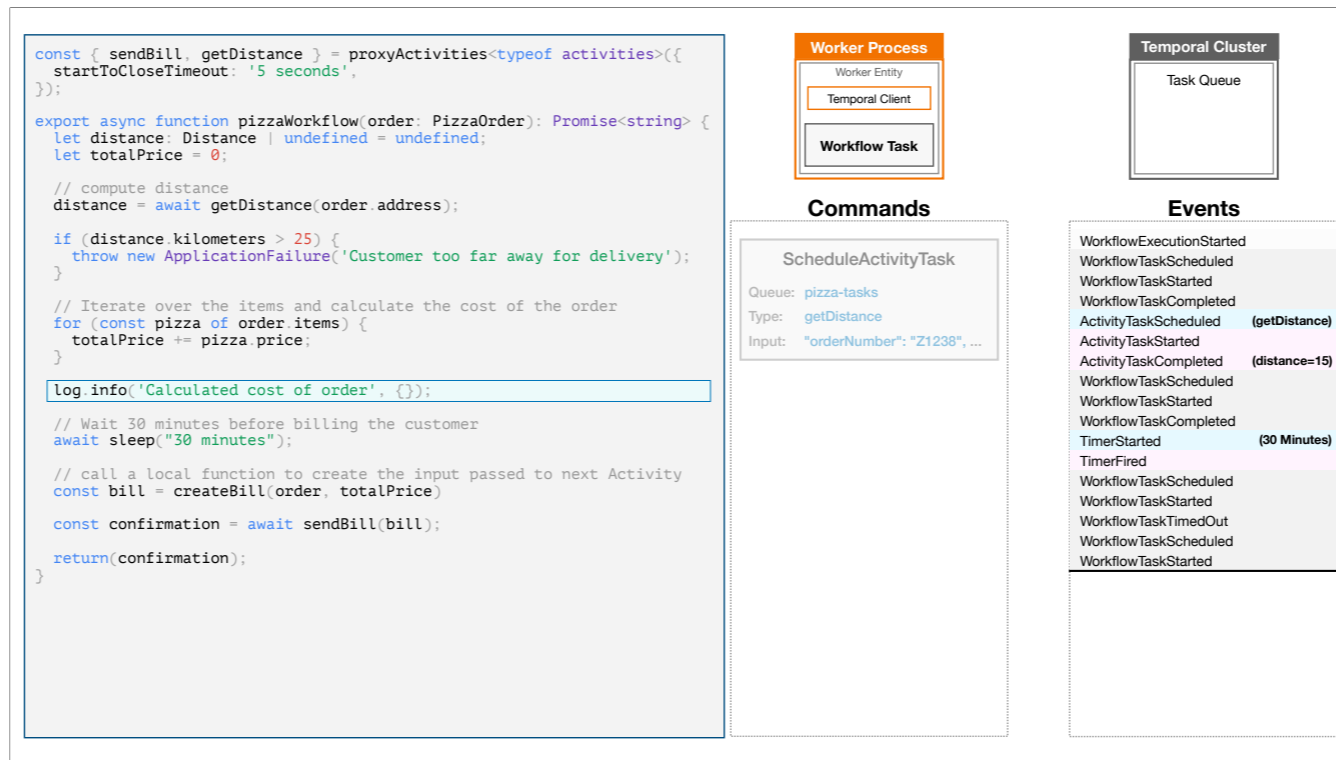
This also means that the Worker has no need to issue the Command to the Temporal Cluster. While Activity code isn't required to be deterministic, the fact that the Worker reuses the result stored in the `ActivityTaskCompleted` Event from the original execution eliminates the possibility that an Activity could behave differently during History Replay than it did originally.



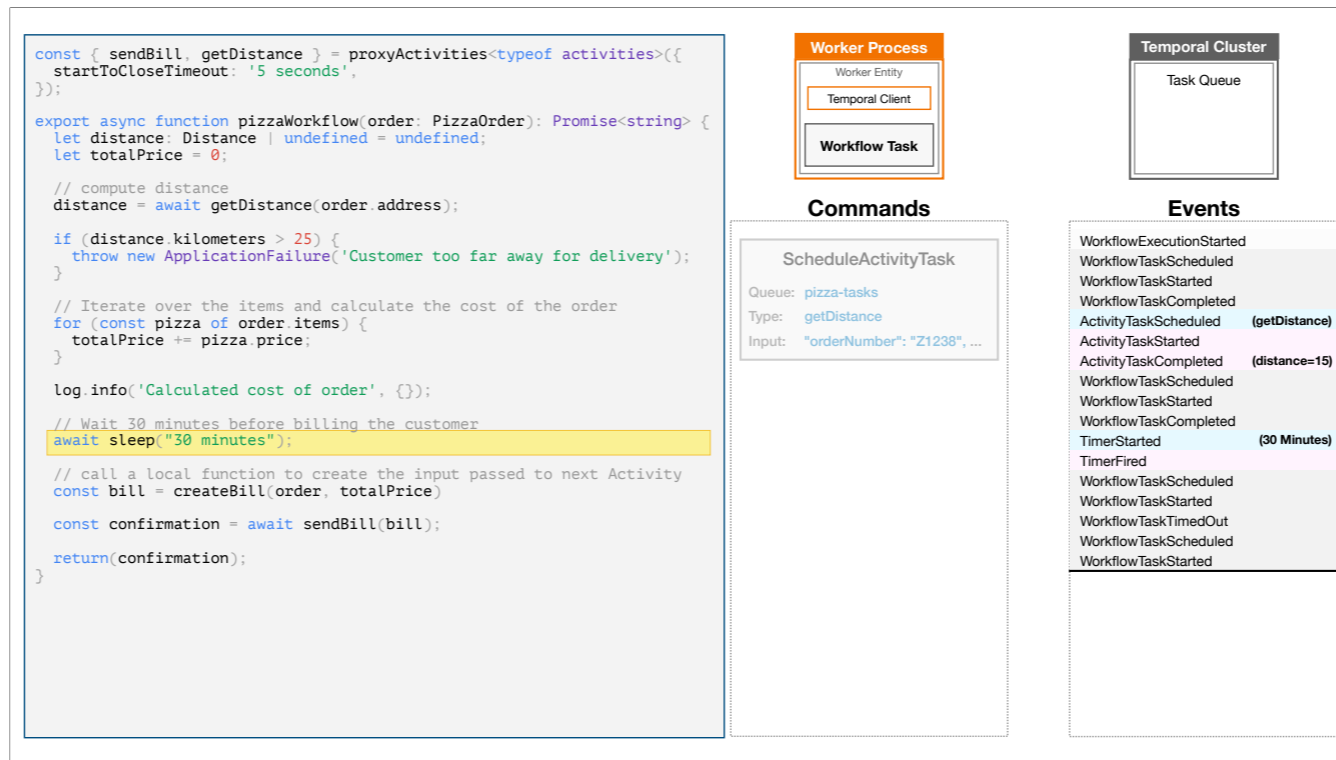
The execution of each statement helps to restore the previous state of the Workflow.



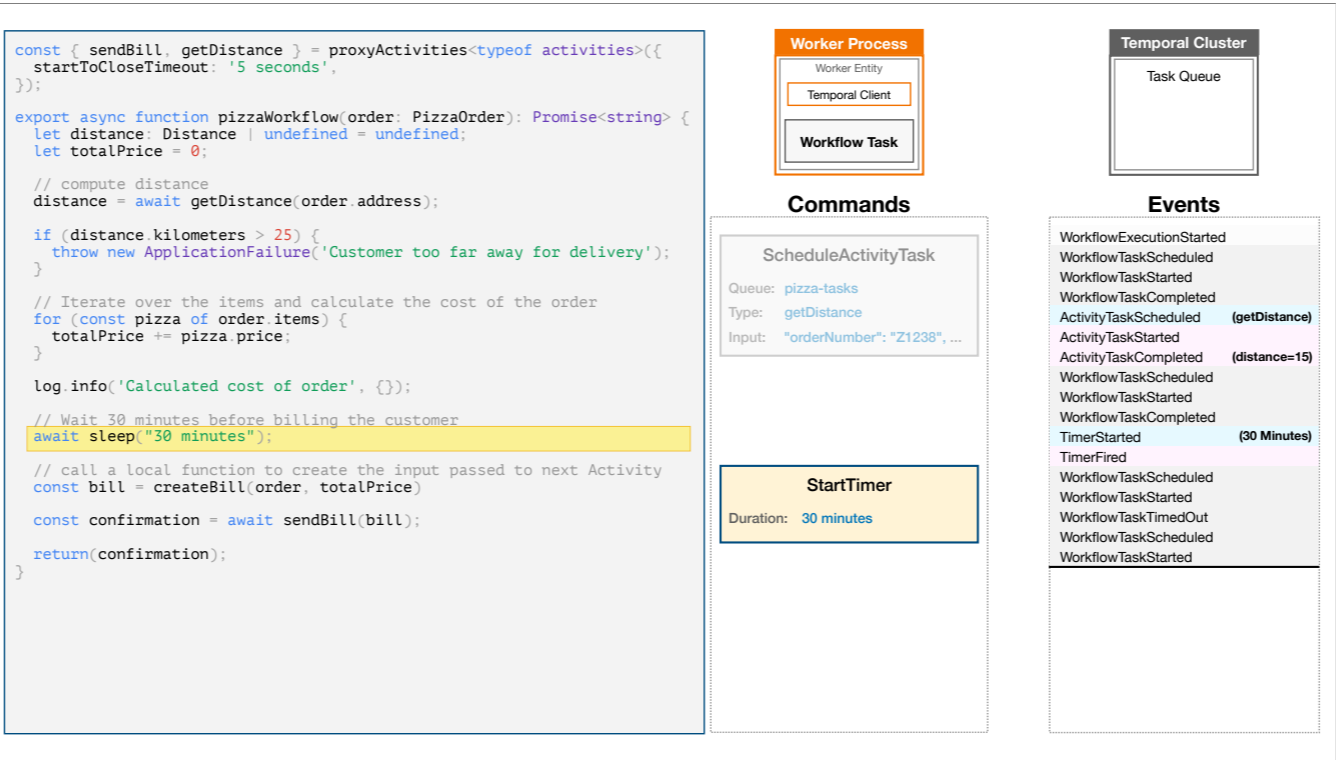
For example, the total gets recomputed.



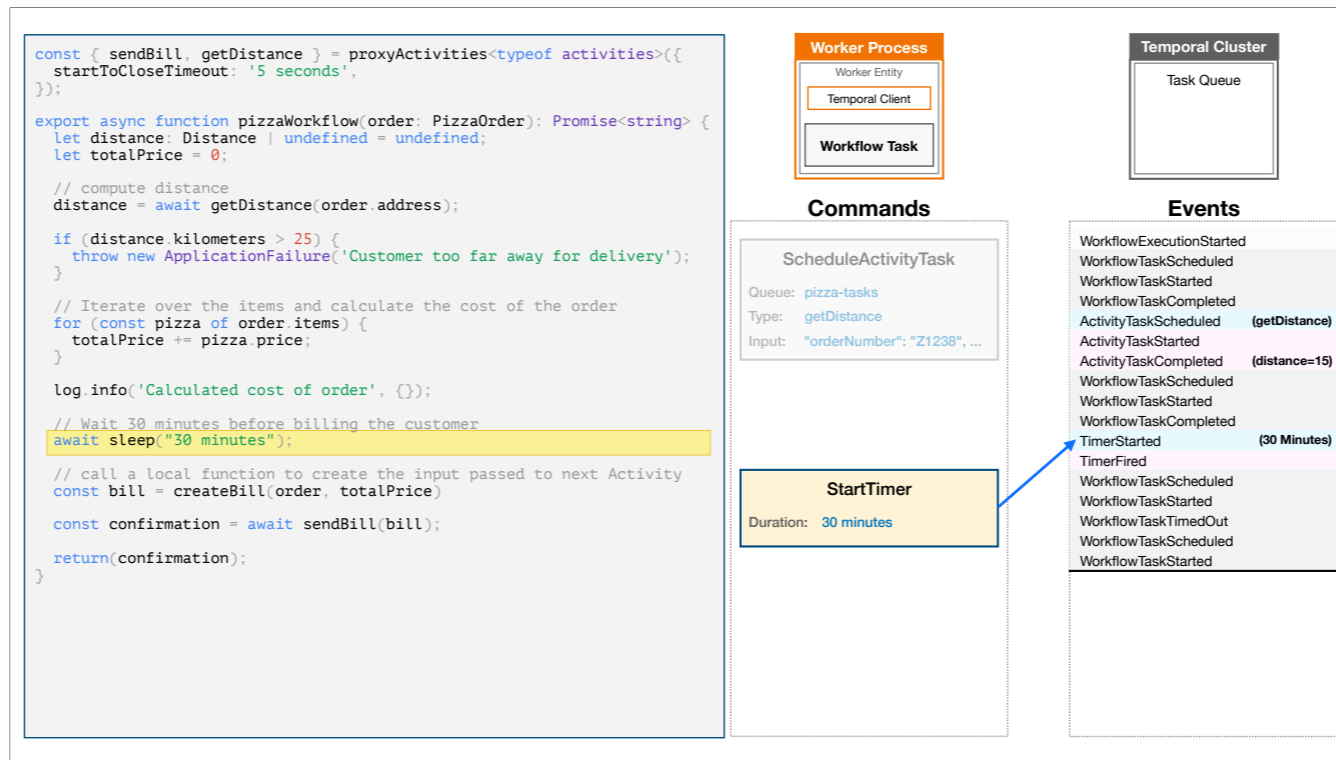
This logging statement is one of the things that behaves differently during replay. Temporal's logger is replay-aware, so it suppresses output during replay so that the logs won't show duplicate messages.



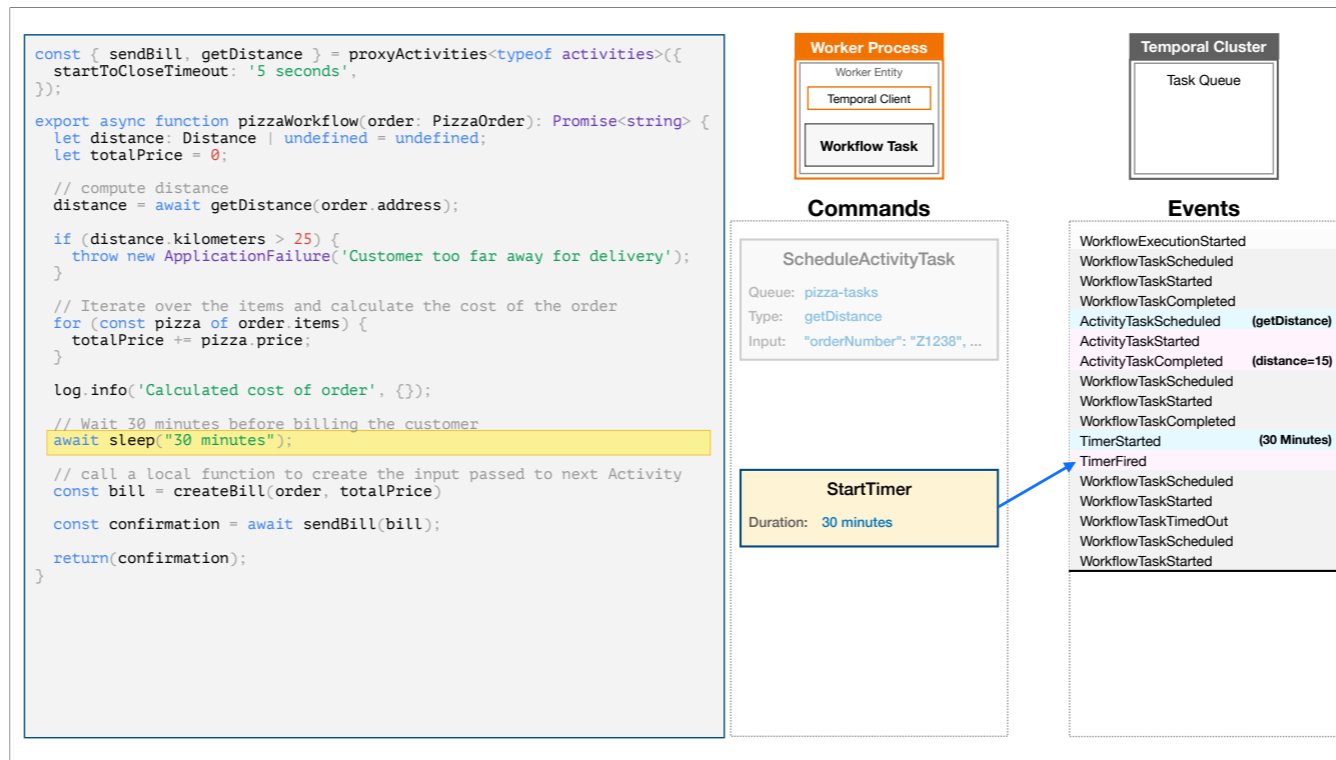
When the Worker reaches the `workflow.Sleep` statement, it evaluates the Event History as it did with the Activity.



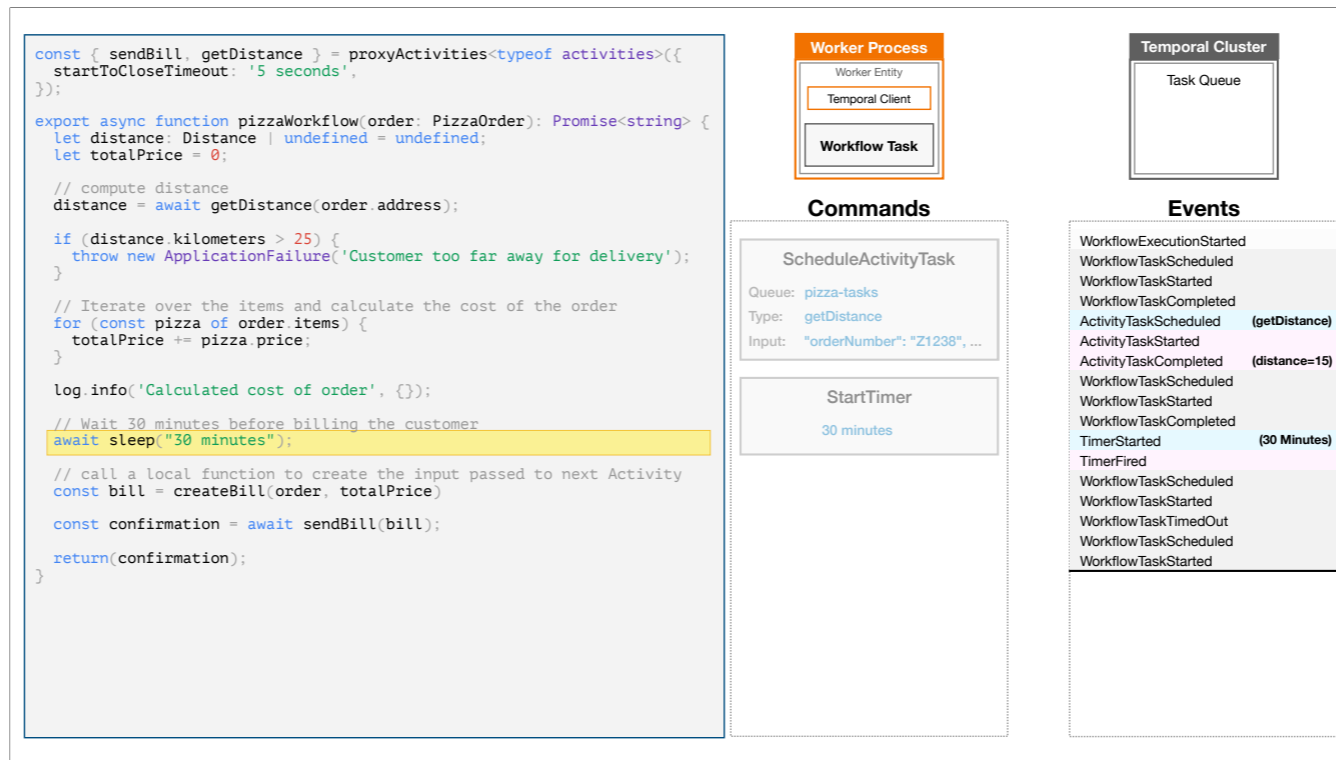
It creates a Command



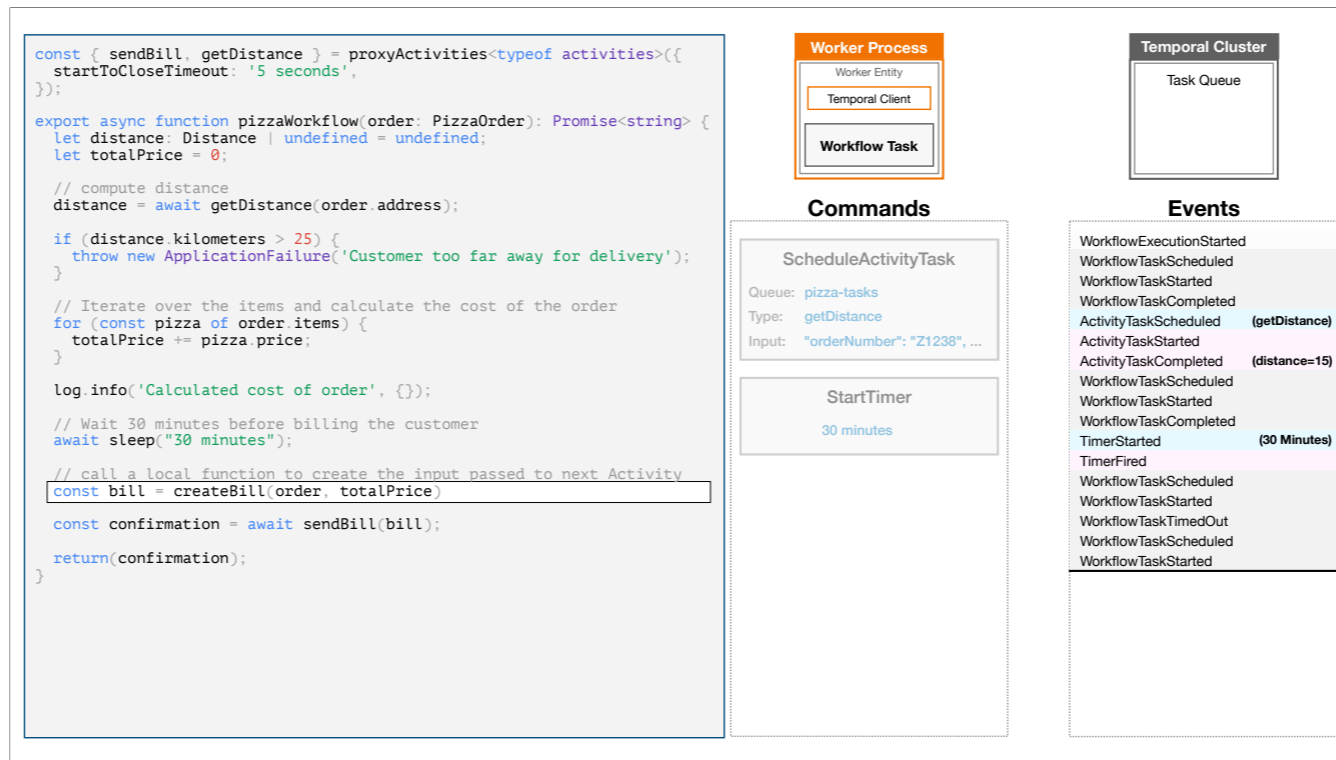
and then checks the Event History to see whether the Timer was started



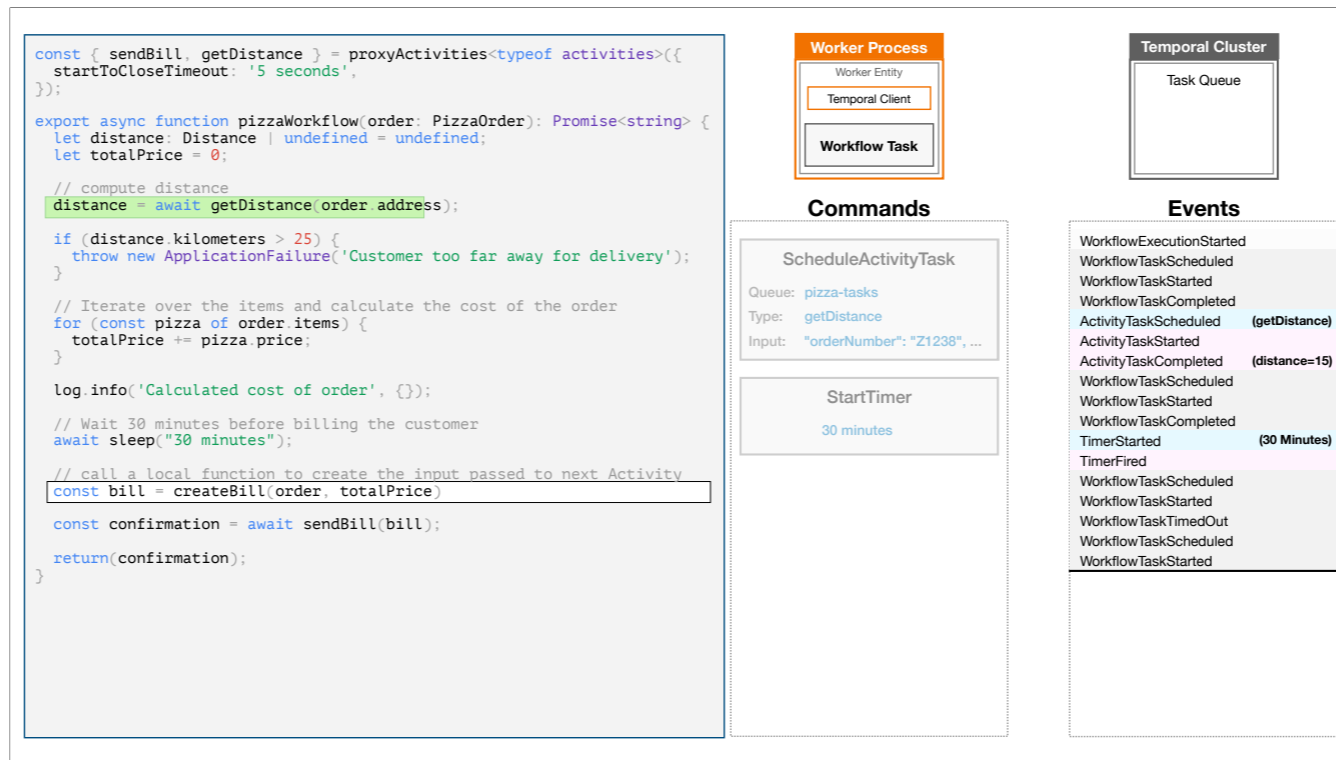
and fired during the previous execution.



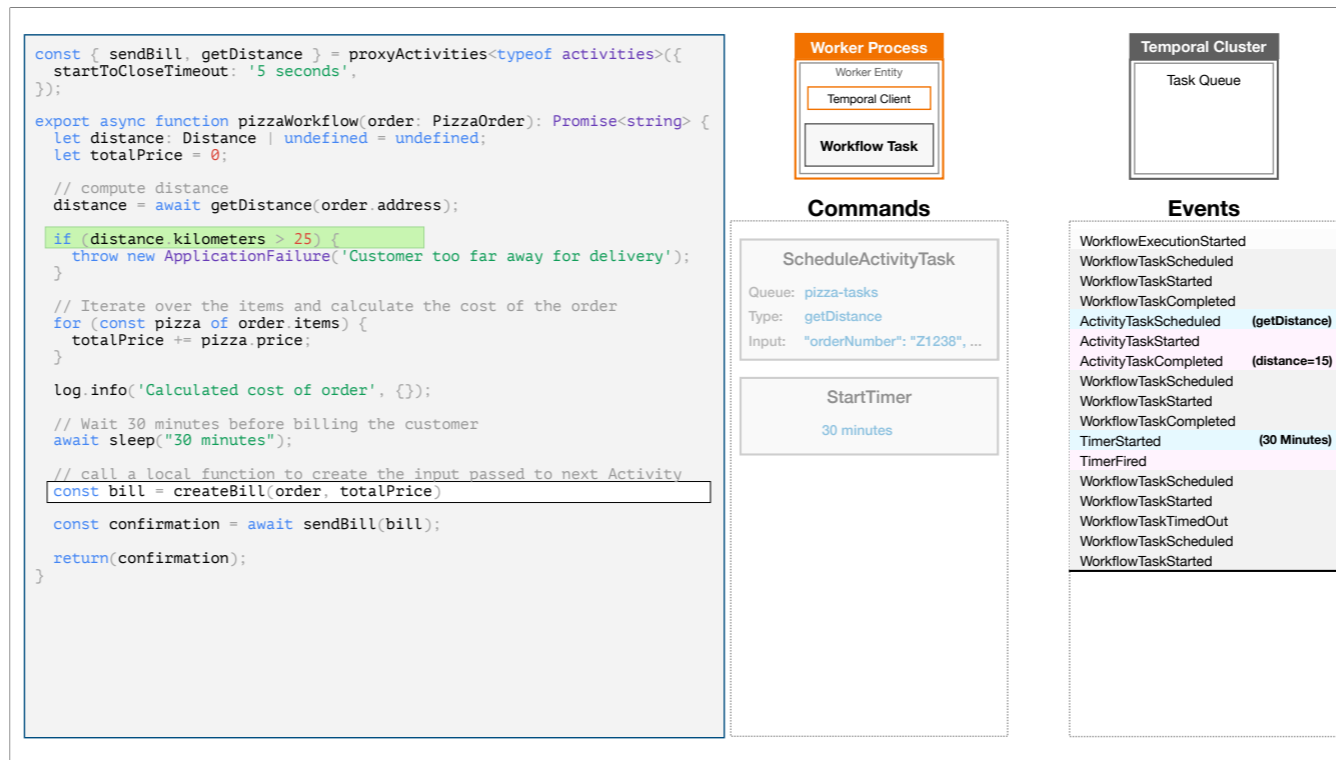
Since the history indicates that both of these things happened, the Worker does not issue the Command to the cluster.



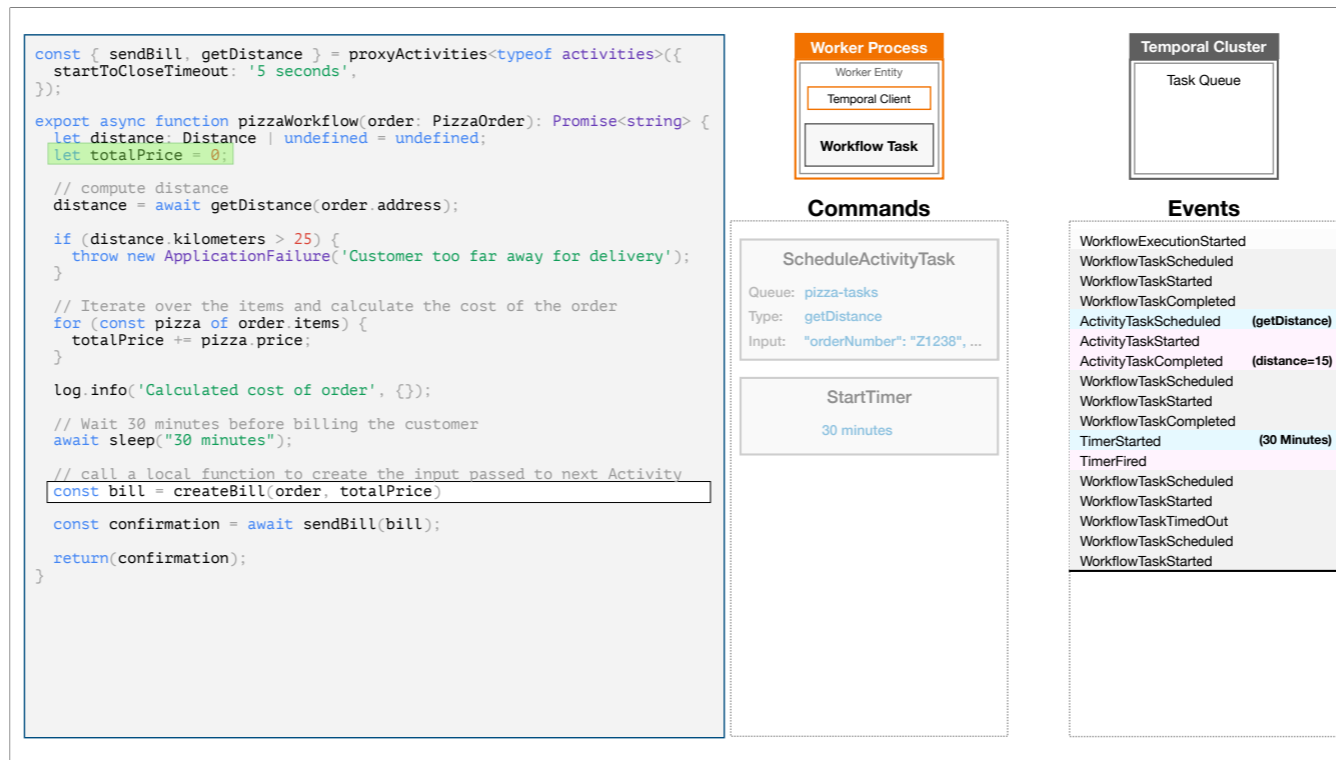
At this point, the Worker has reached the point where the crash occurred, and replaying the code has completely restored the state of the Workflow prior to the crash



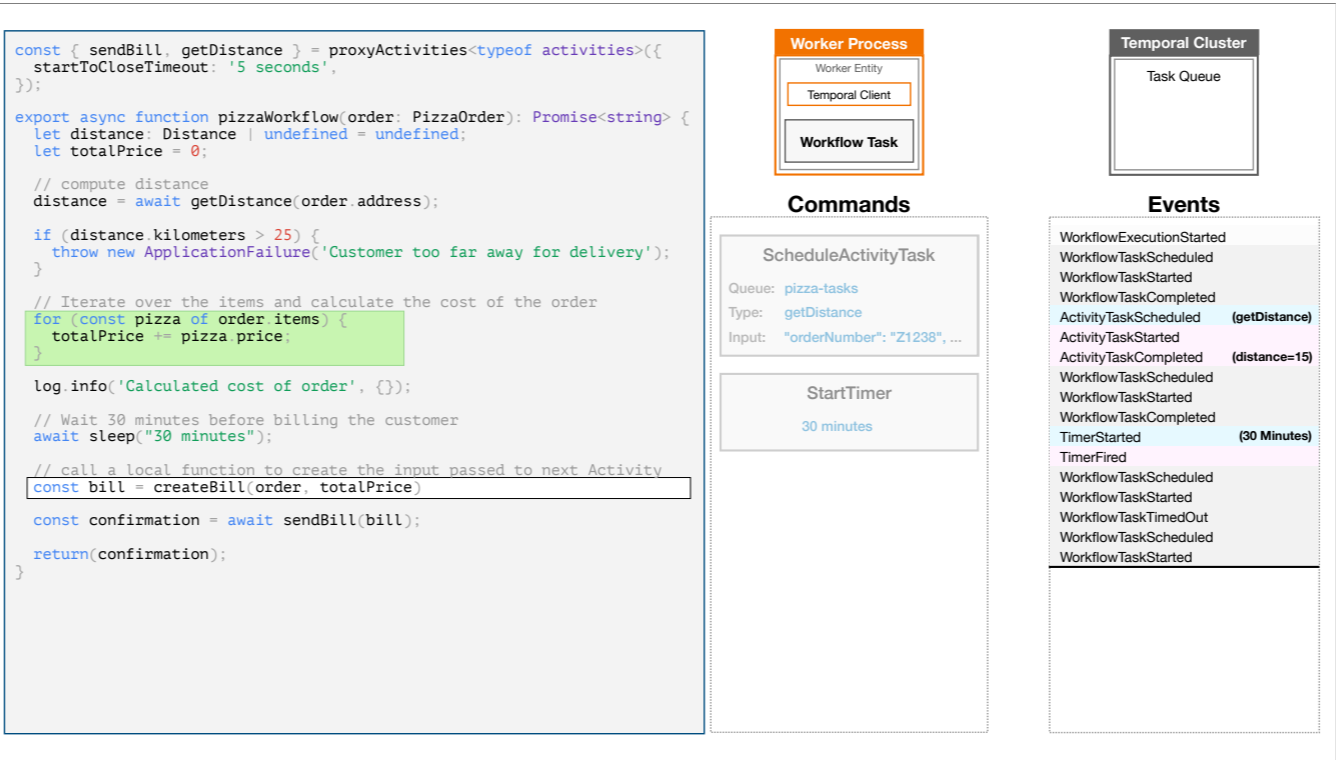
For example, the distance variable has the same value it did prior to the crash, which was originally returned by executing the `getDistance` Activity.



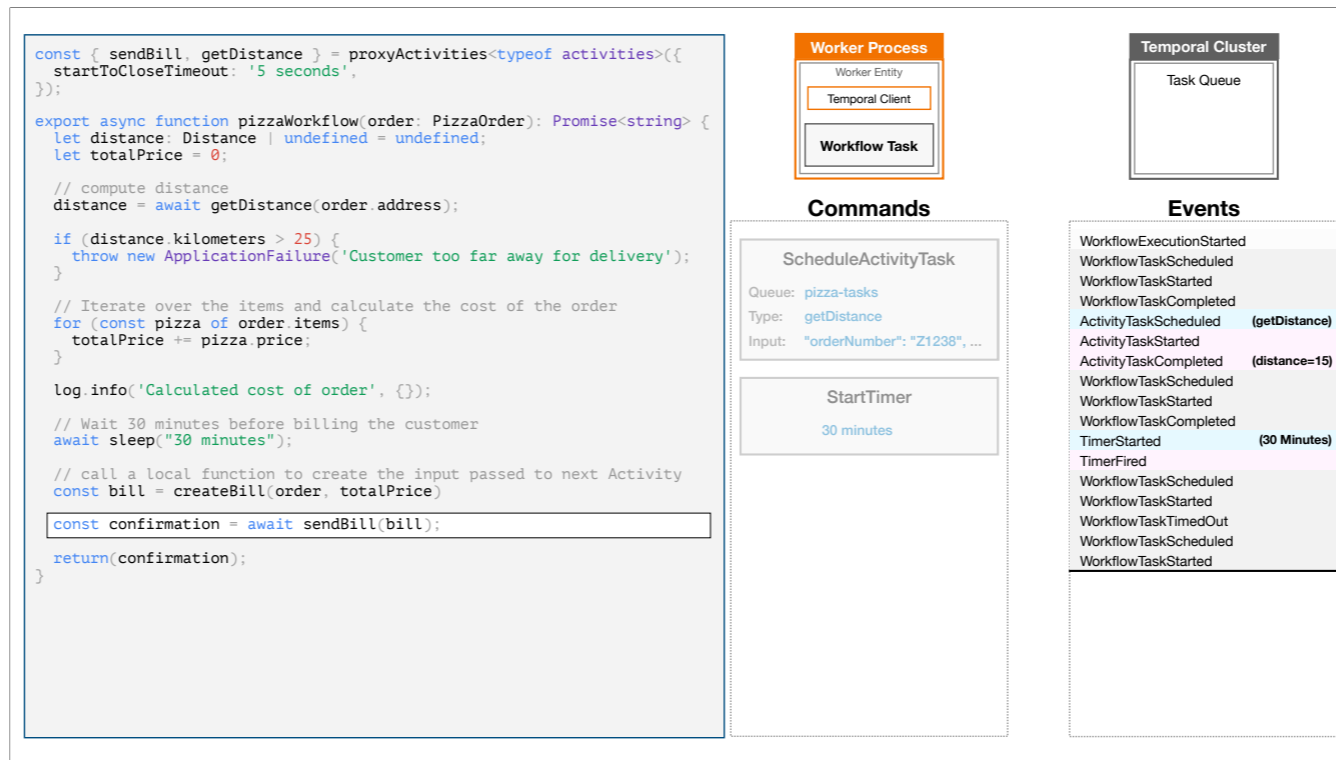
Since replay uses the same input data as before, this also means that the conditional statement on line 20 evaluates to `false`, just like it did before.



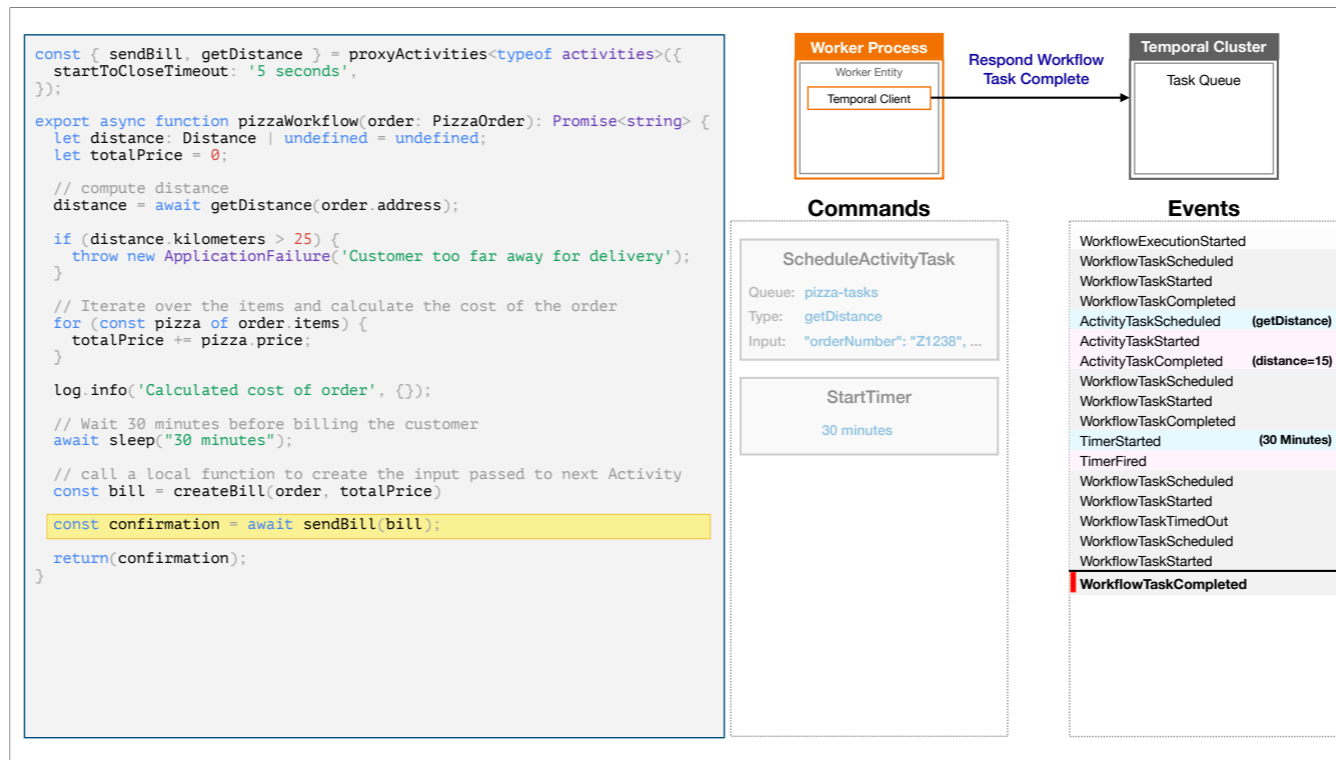
The same is true for the variable which was used to calculate the total price of the order.



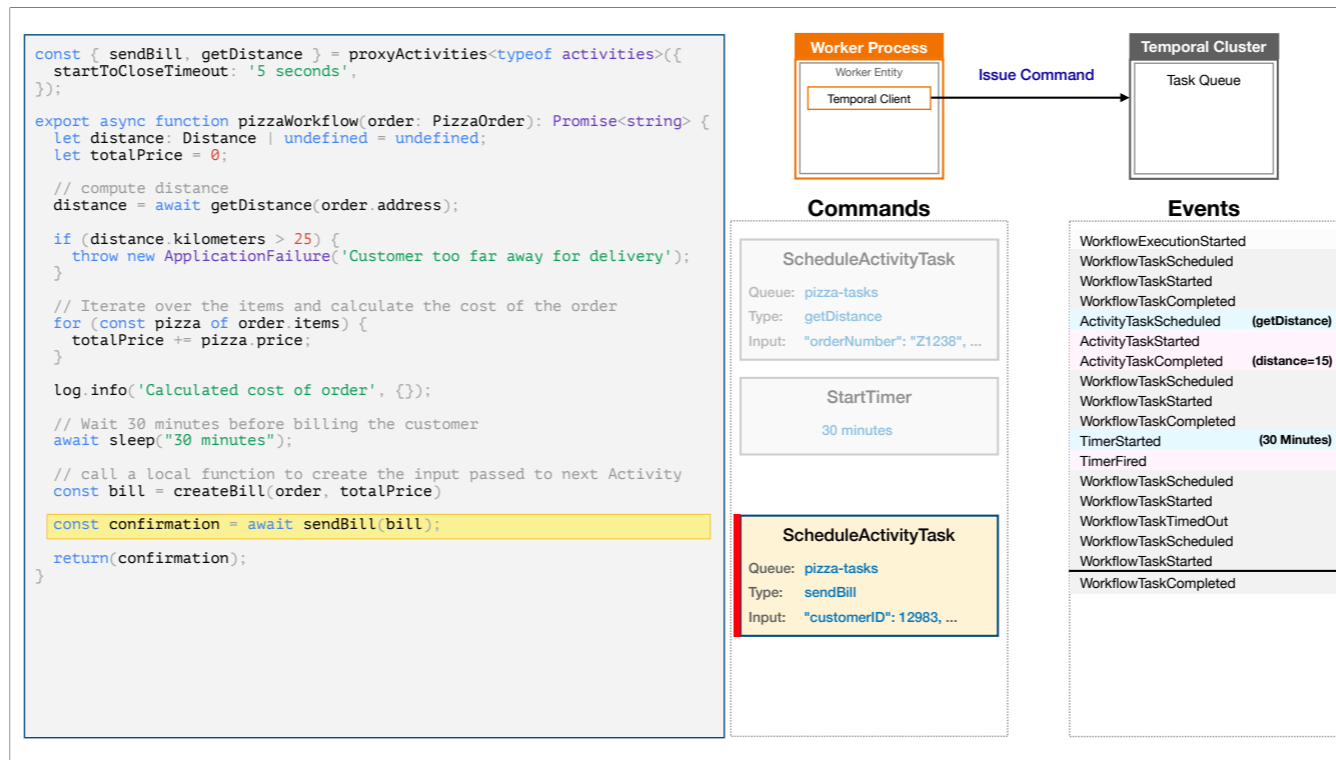
It has the same value it did as well.



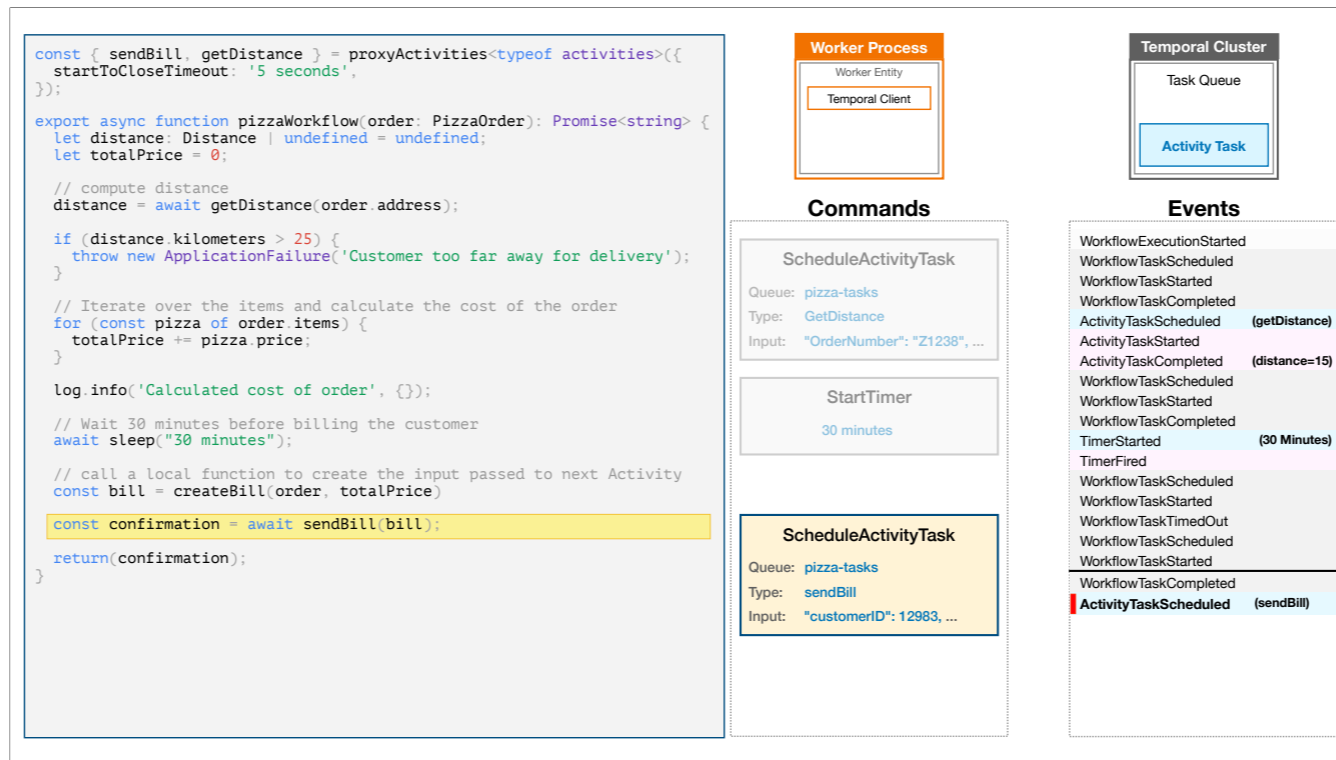
The Worker has now reached a statement `__beyond__` where the crash occurred, which is evident because the Event History doesn't contain any Events related to this Activity. Further execution of this Workflow continues on as is the crash had never happened.



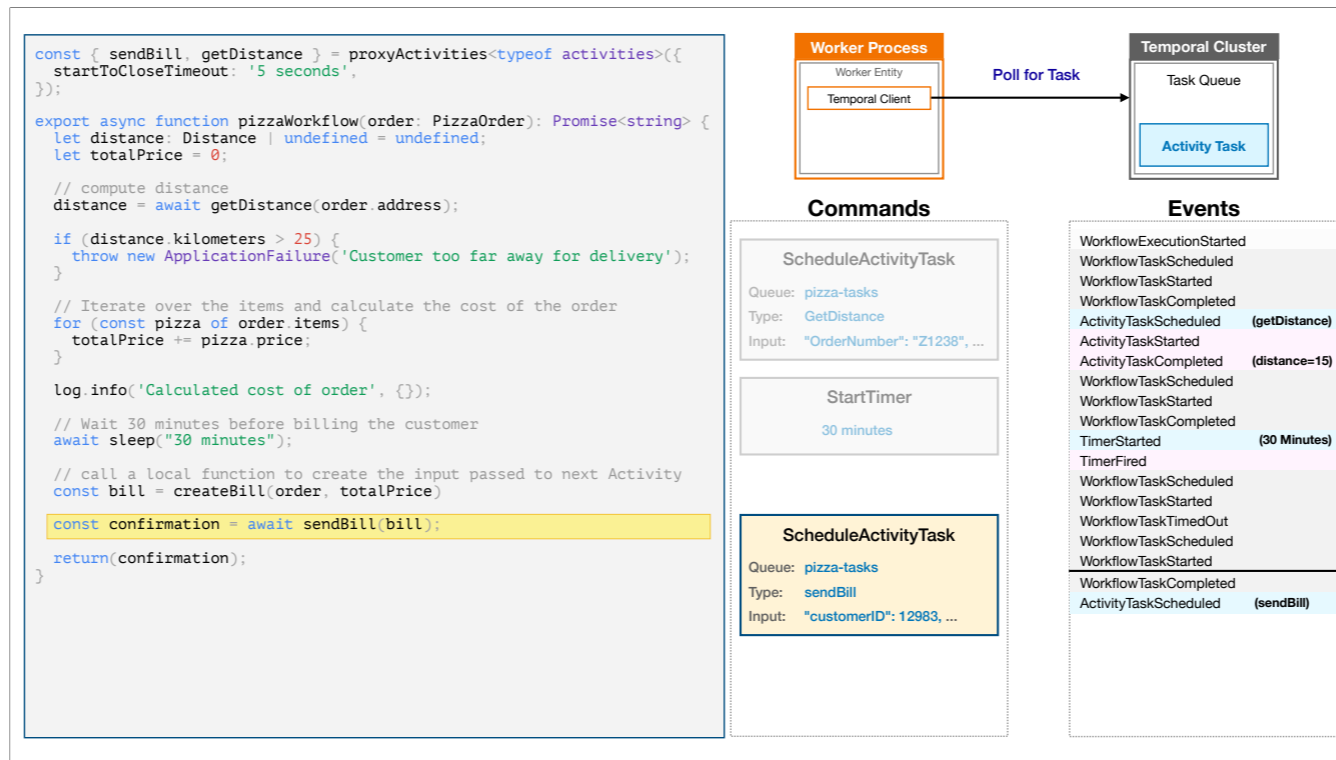
Because it has encountered an `ExecuteActivity` call, the Worker completes the current Workflow Task,



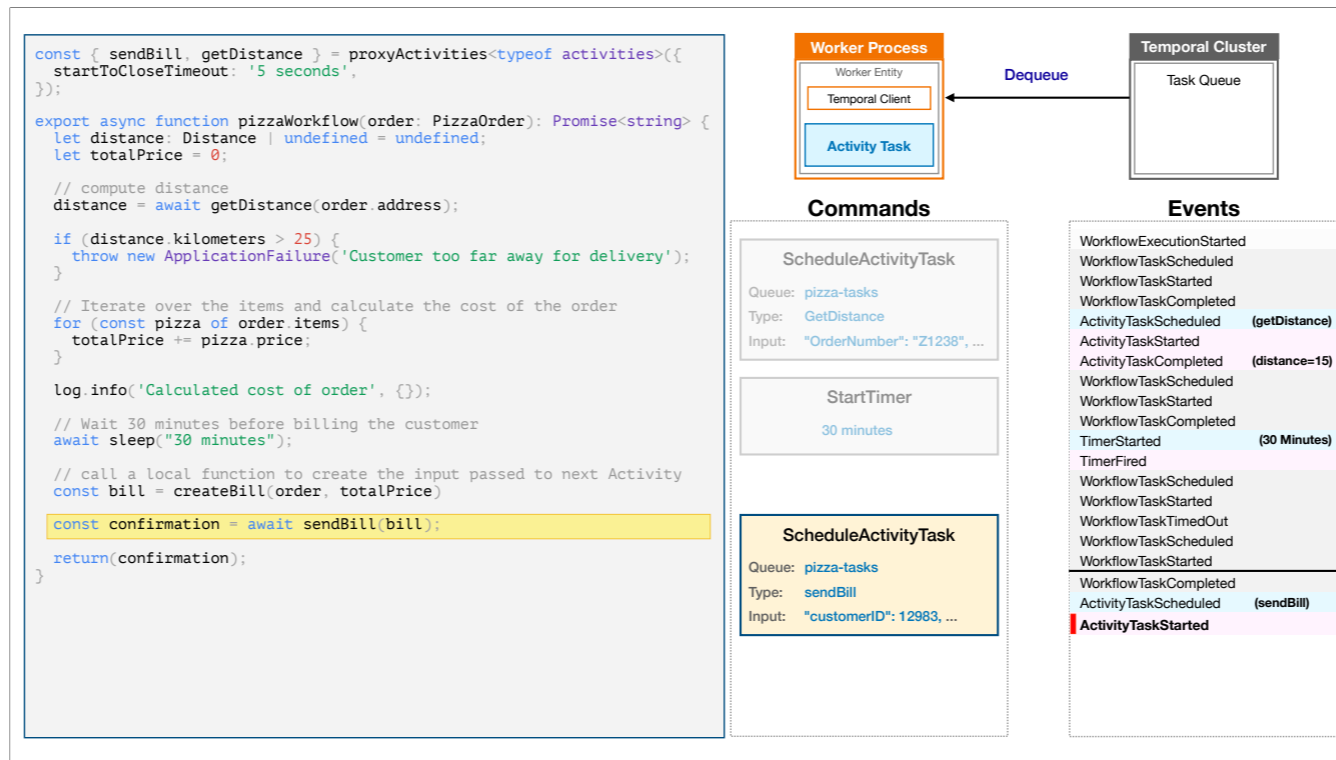
and issues a Command to the cluster, requesting execution of this Activity.



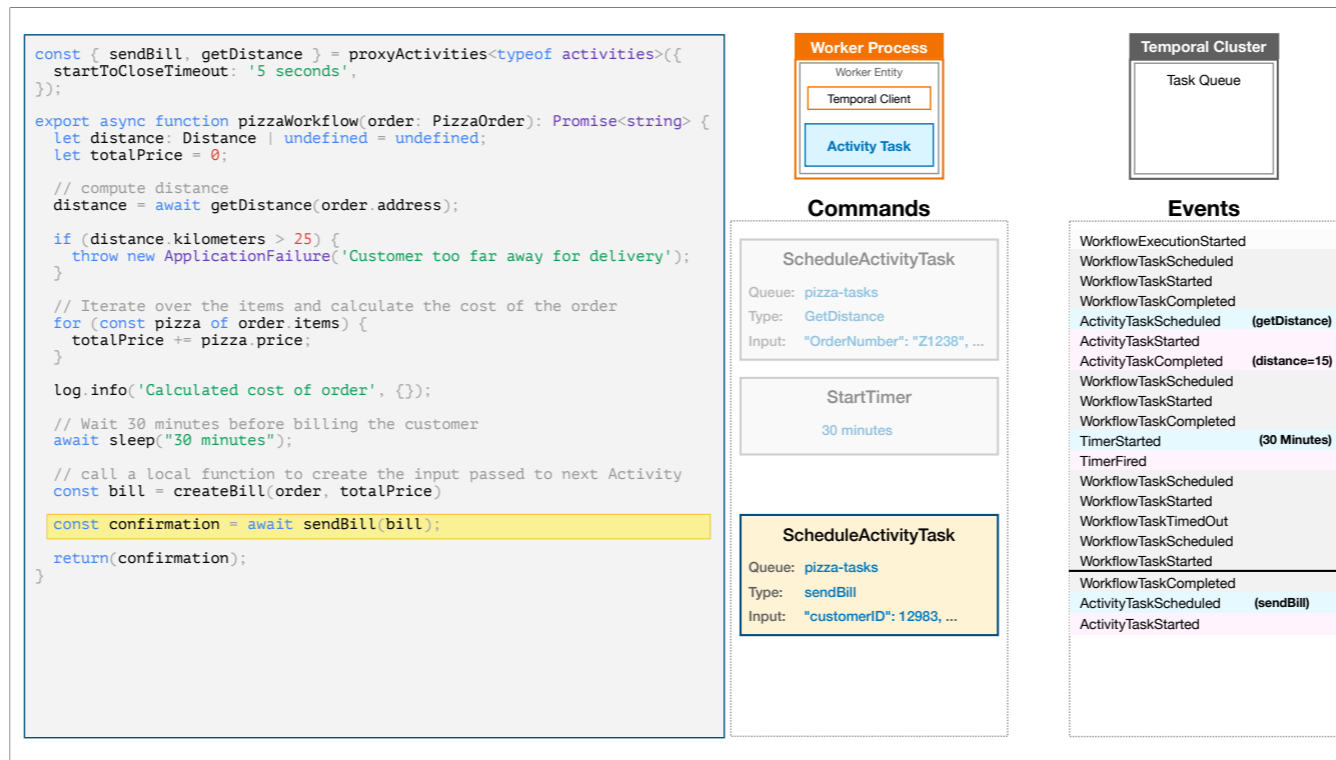
The cluster schedules an Activity Task.



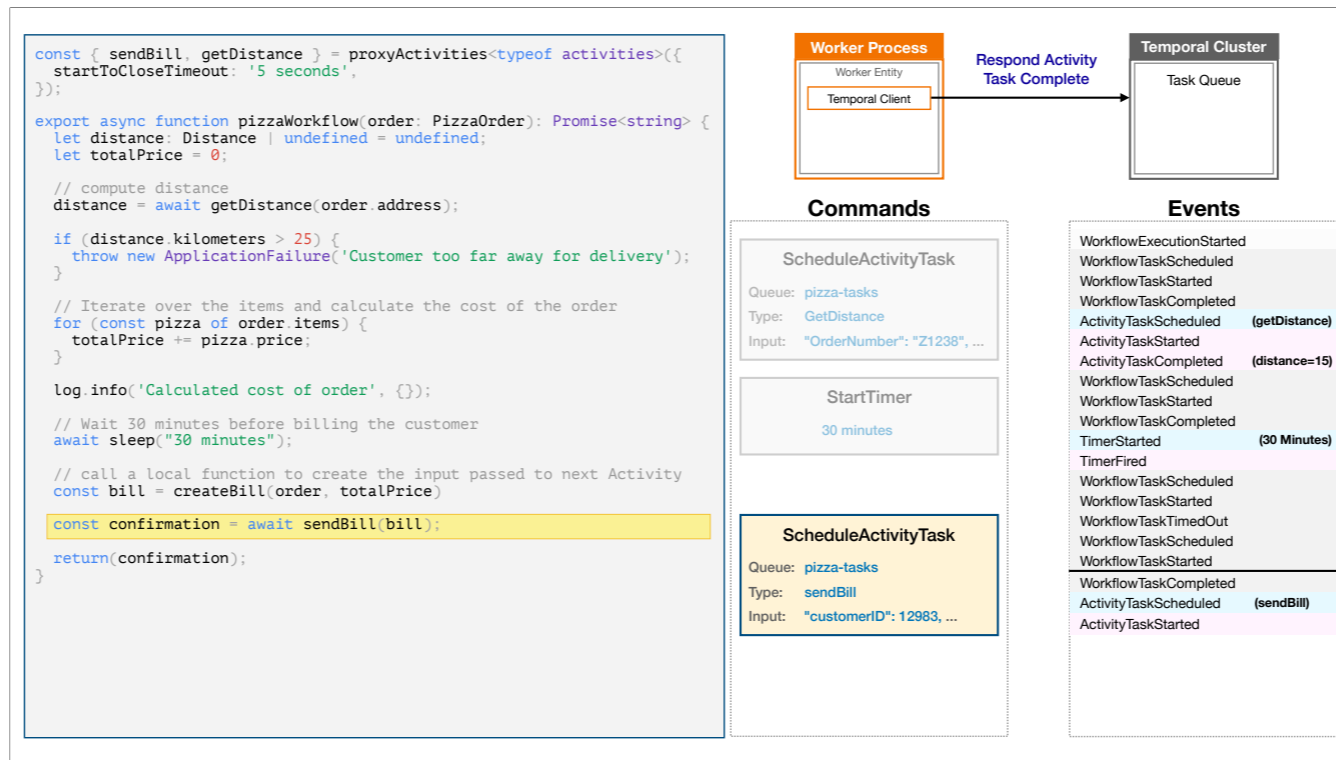
When the Worker polls,



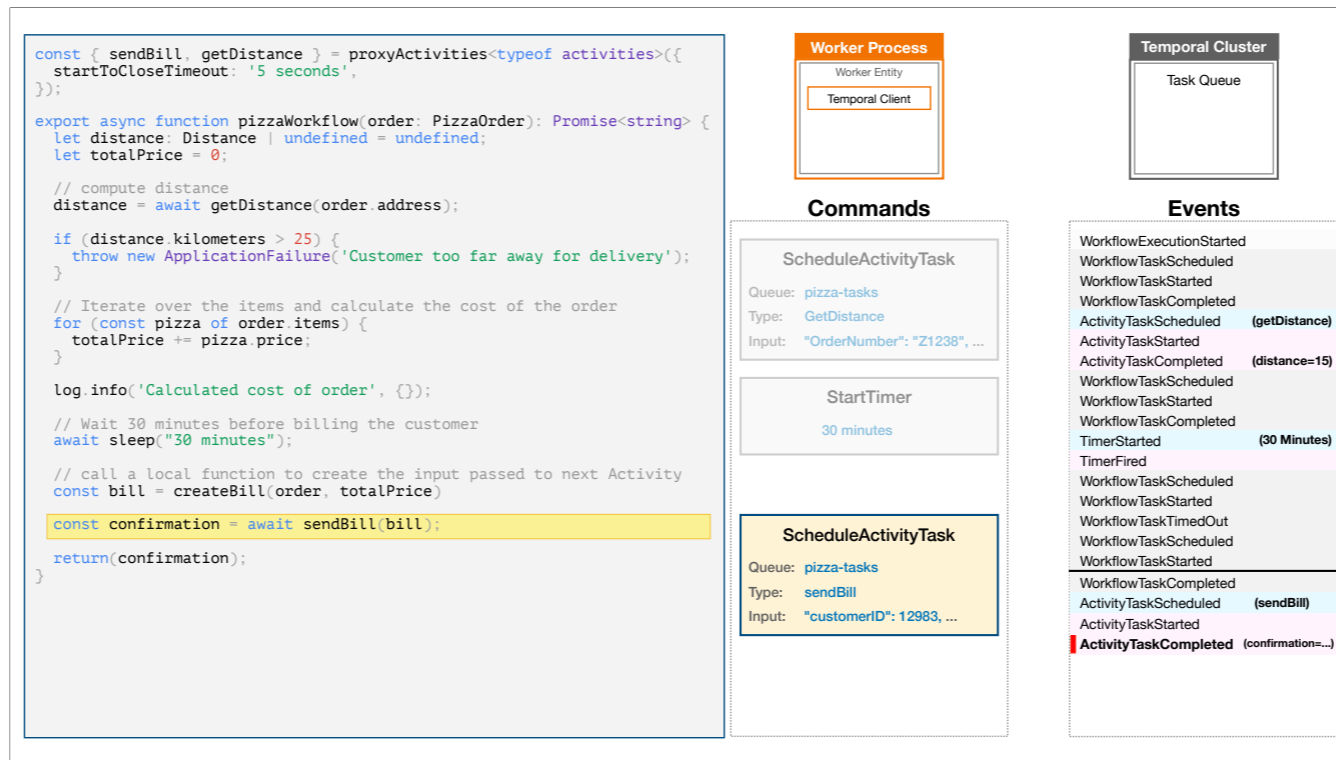
it accepts the Activity Task



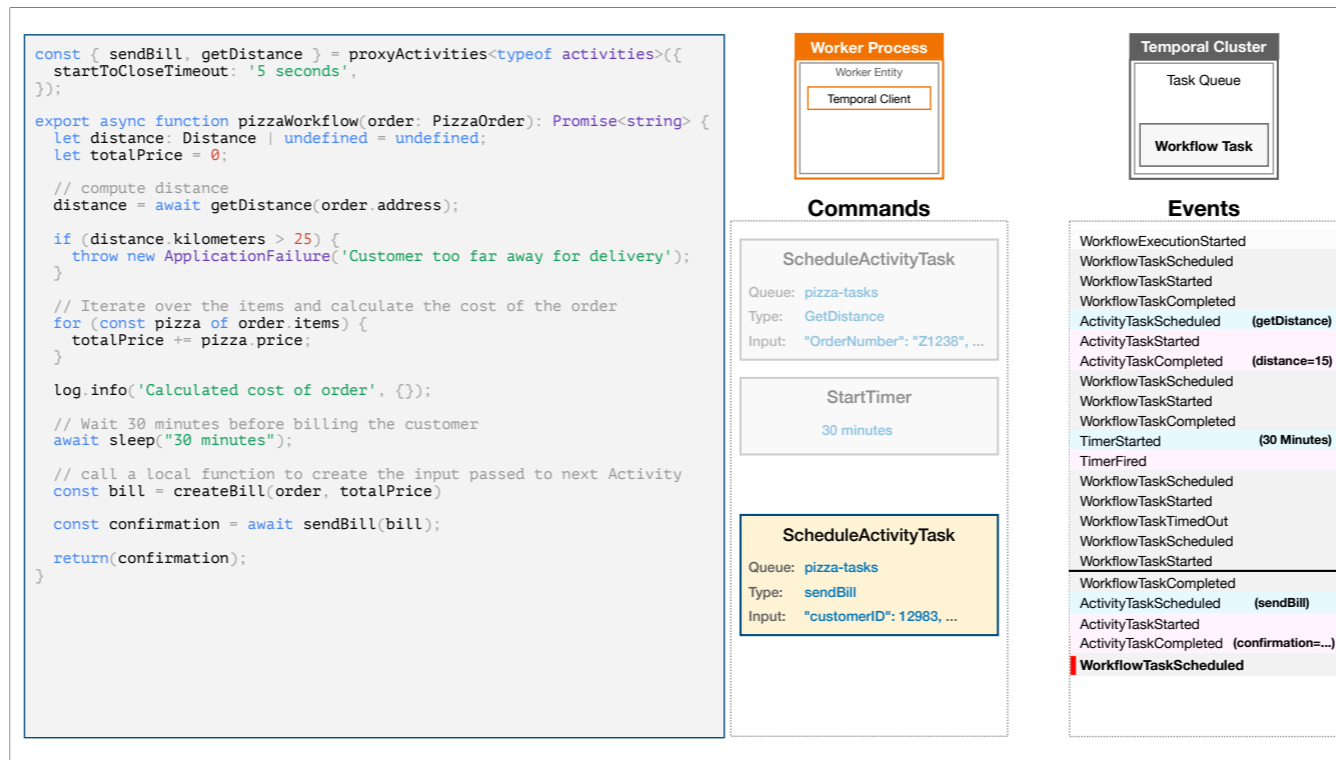
and executes the code for this Activity.



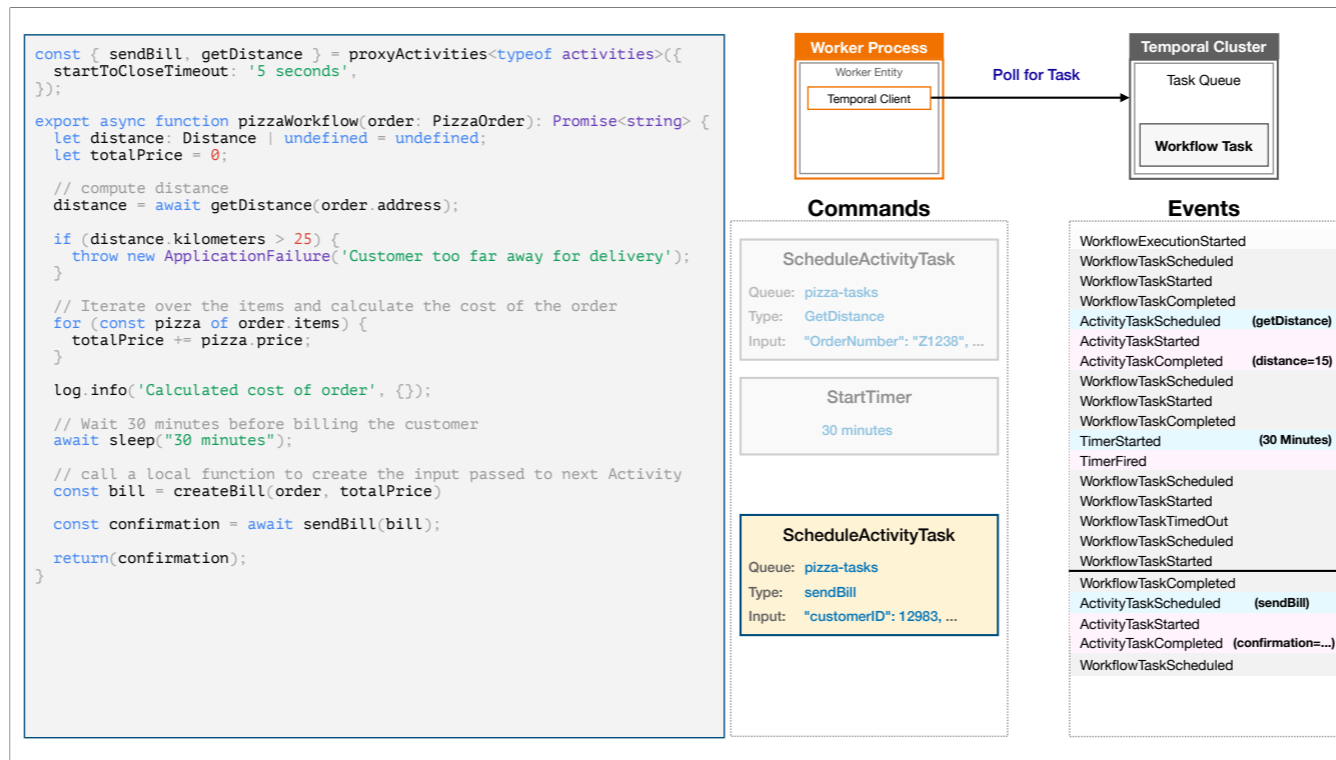
When the Activity function returns a result, the Worker notifies the cluster,



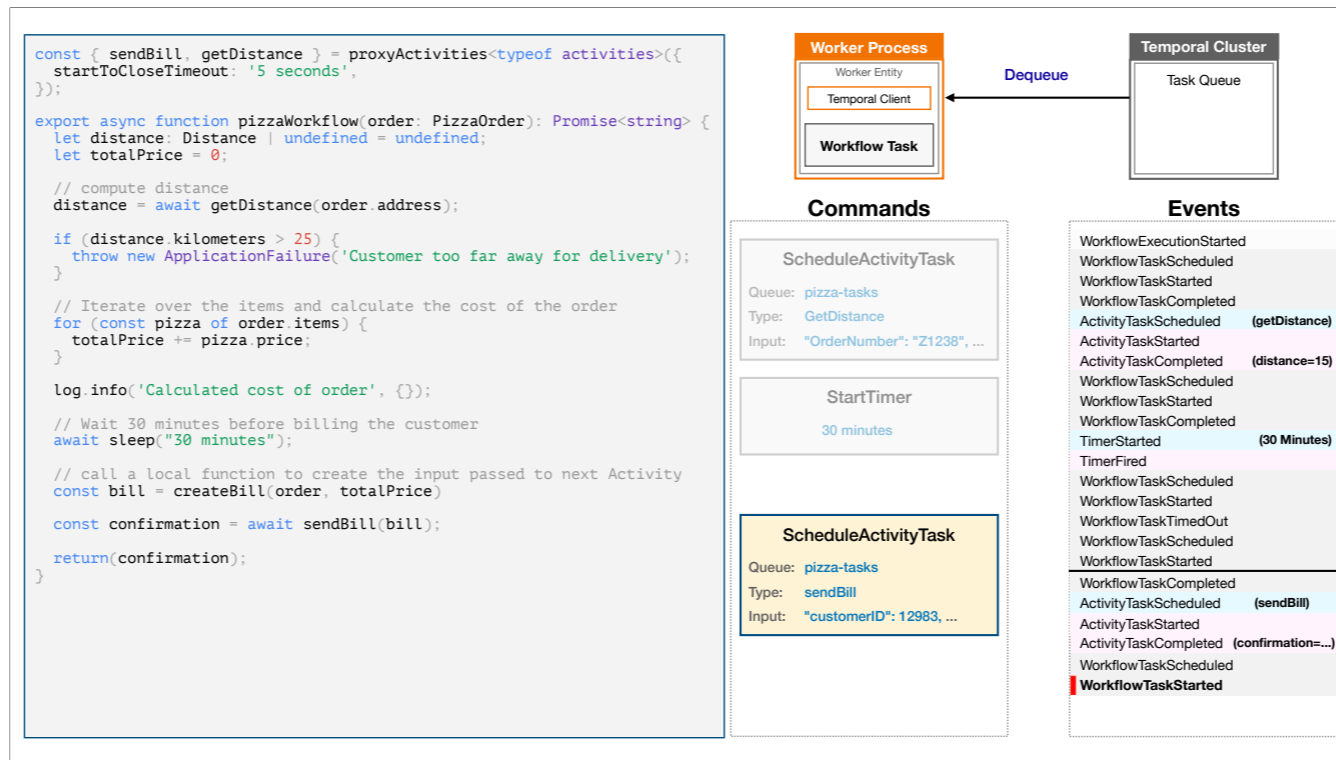
which logs an `ActivityTaskCompleted` Event.



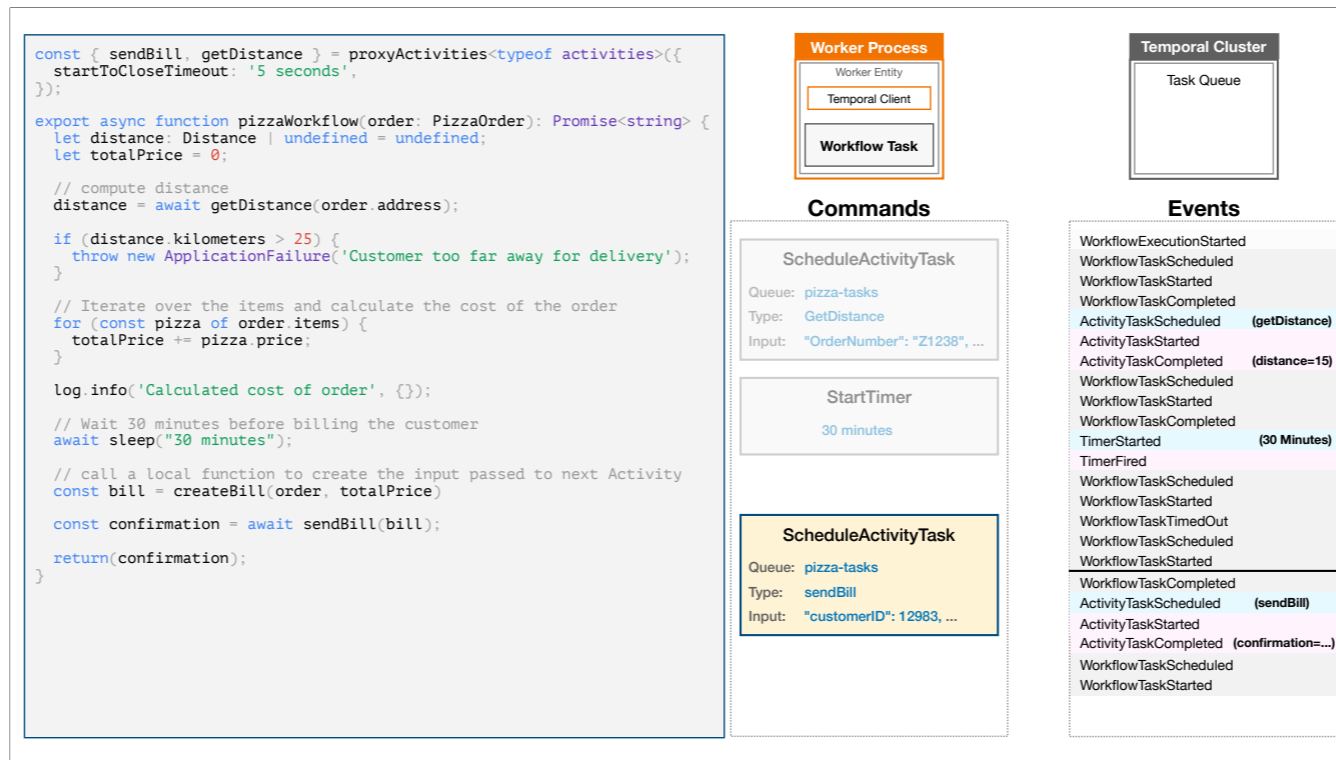
But since the cluster hasn't yet received a Command that says the Workflow Execution has completed or failed, the cluster schedules another Workflow Task to continue progress of this execution.



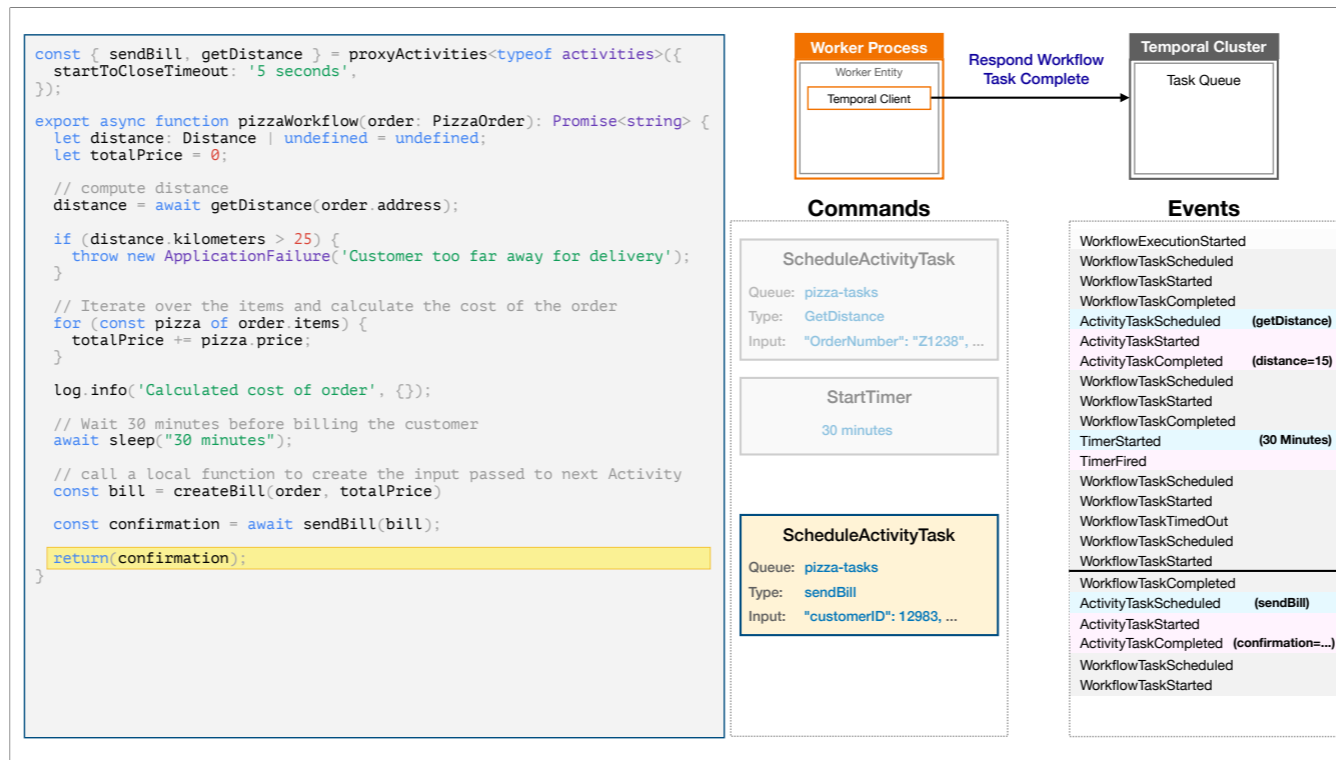
The Worker polls the Task Queue,



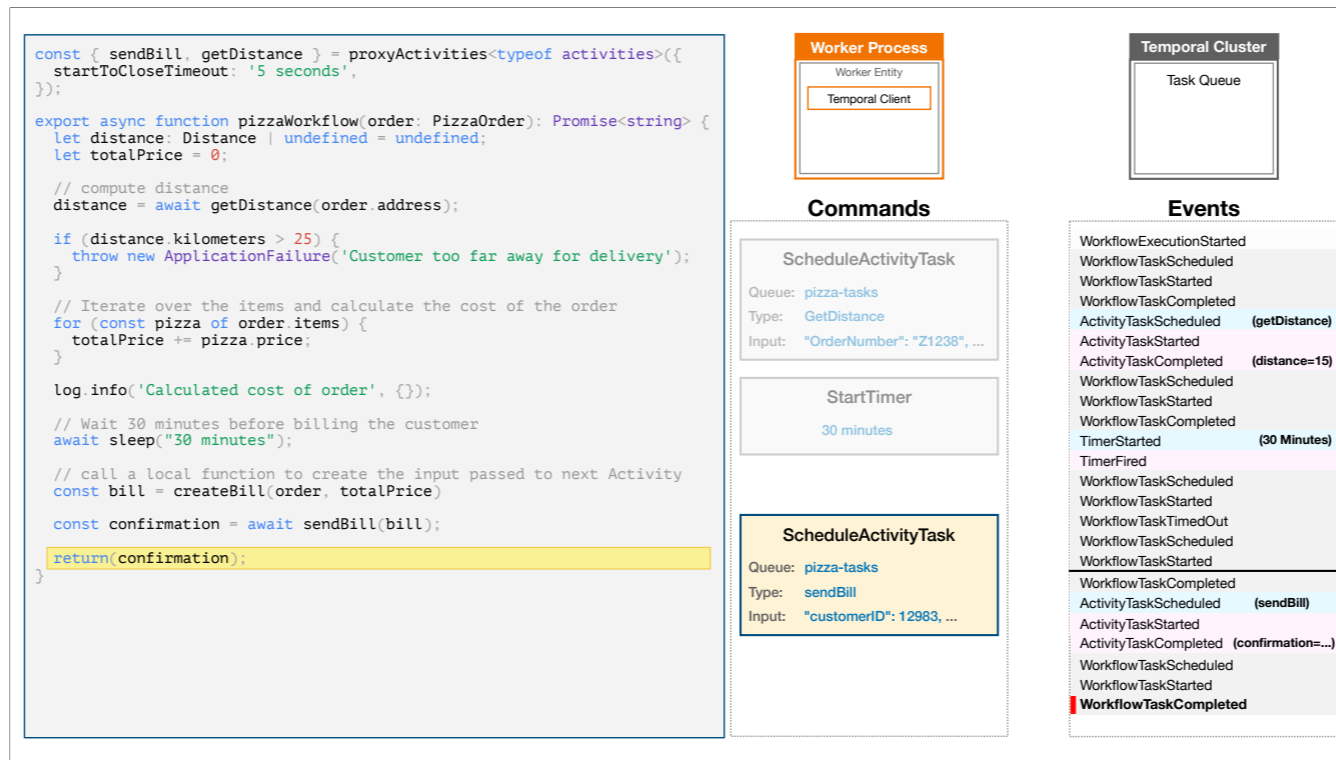
accepts the Workflow Task,



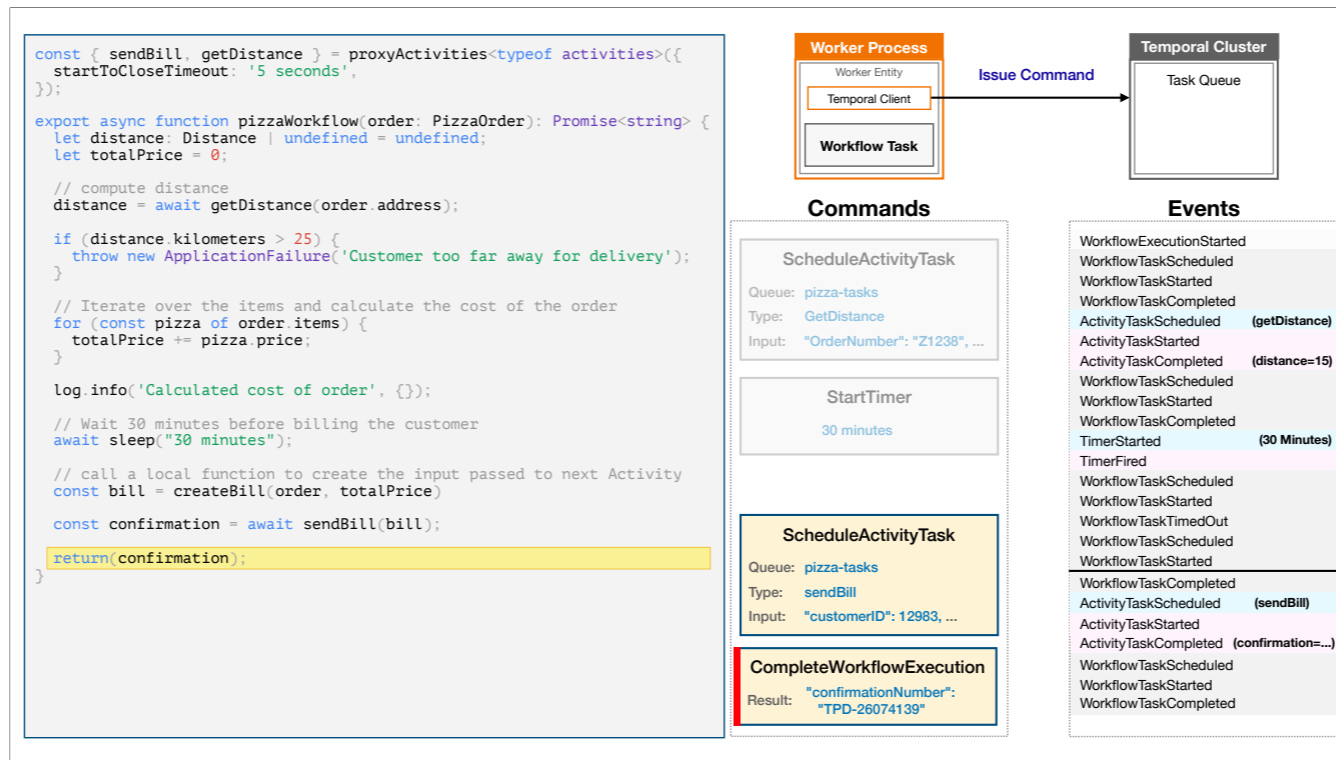
and resumes execution of the remaining Workflow code.



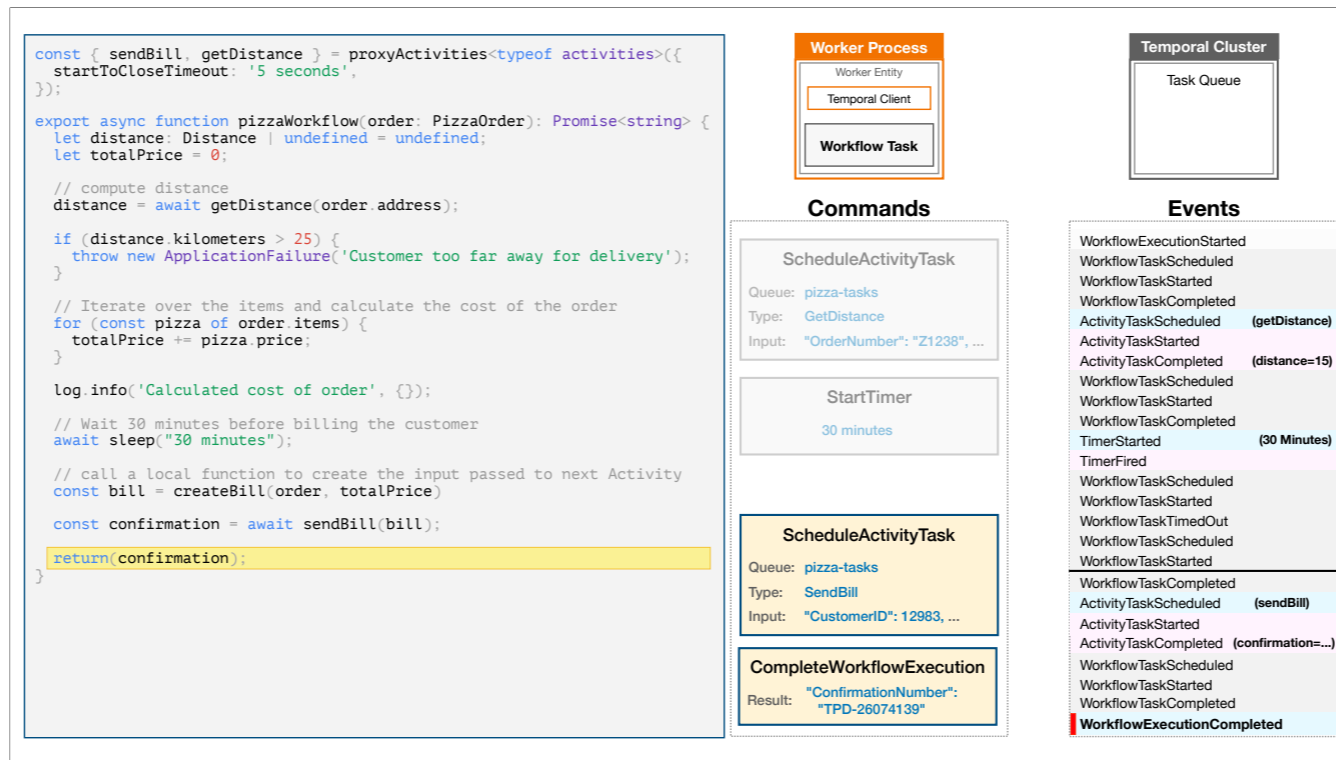
When it reaches the end,



it notifies that the cluster that the current Workflow Task is complete, and the cluster logs an Event to reflect this.



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.



The cluster then logs `WorkflowExecutionCompleted` as the final Event in the history. The Workflow Execution has now closed.

Why Temporal Requires Determinism for Workflows



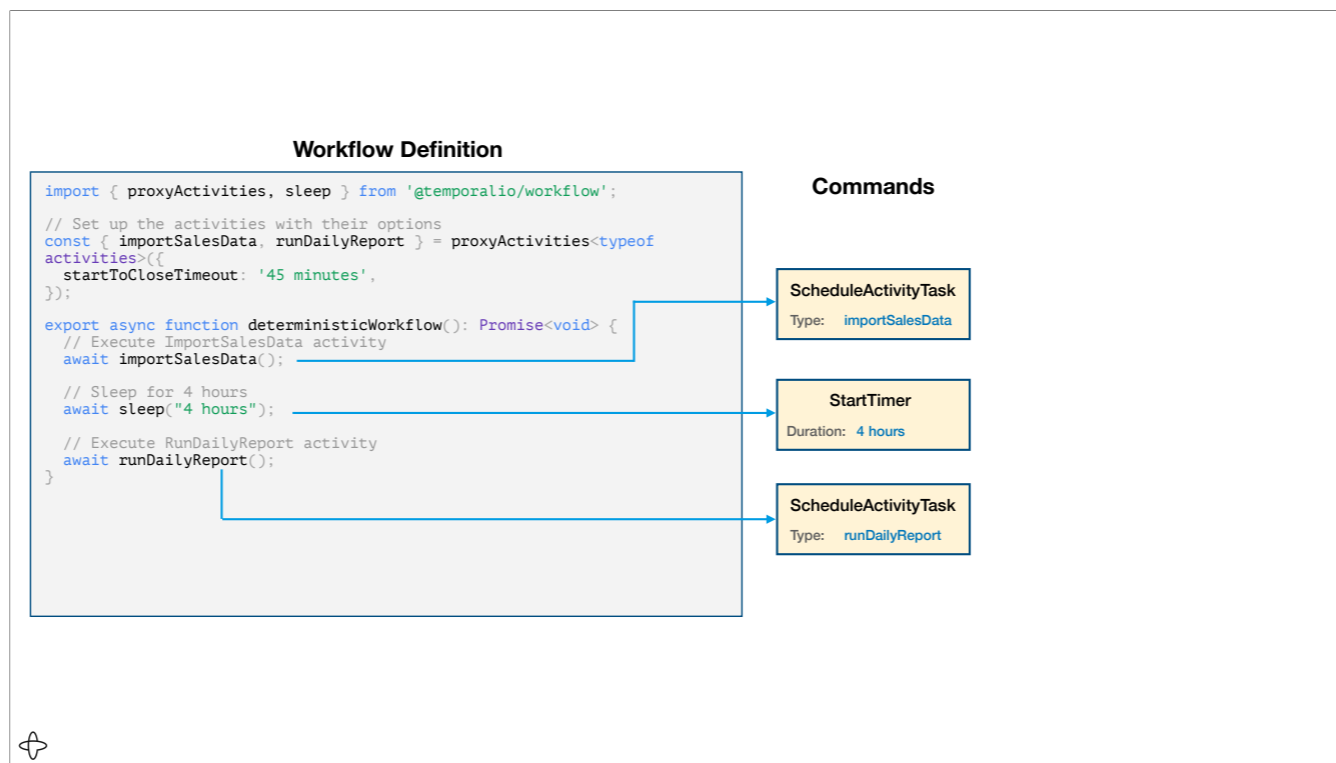
In Temporal 101, we mentioned that Workflow code must be deterministic. I'm going to explain not only what that means, but also why it's important, but first I want to reiterate a few important details about Workflow Execution to provide some context for this explanation.

Workflow Definition

```
import { proxyActivities, sleep } from '@temporalio/workflow';  
  
// Set up the activities with their options  
const { importSalesData, runDailyReport } = proxyActivities<typeof  
activities>({  
  startToCloseTimeout: '45 minutes',  
});  
  
export async function deterministicWorkflow(): Promise<void> {  
  // Execute ImportSalesData activity  
  await importSalesData();  
  
  // Sleep for 4 hours  
  await sleep("4 hours");  
  
  // Execute RunDailyReport activity  
  await runDailyReport();  
}
```

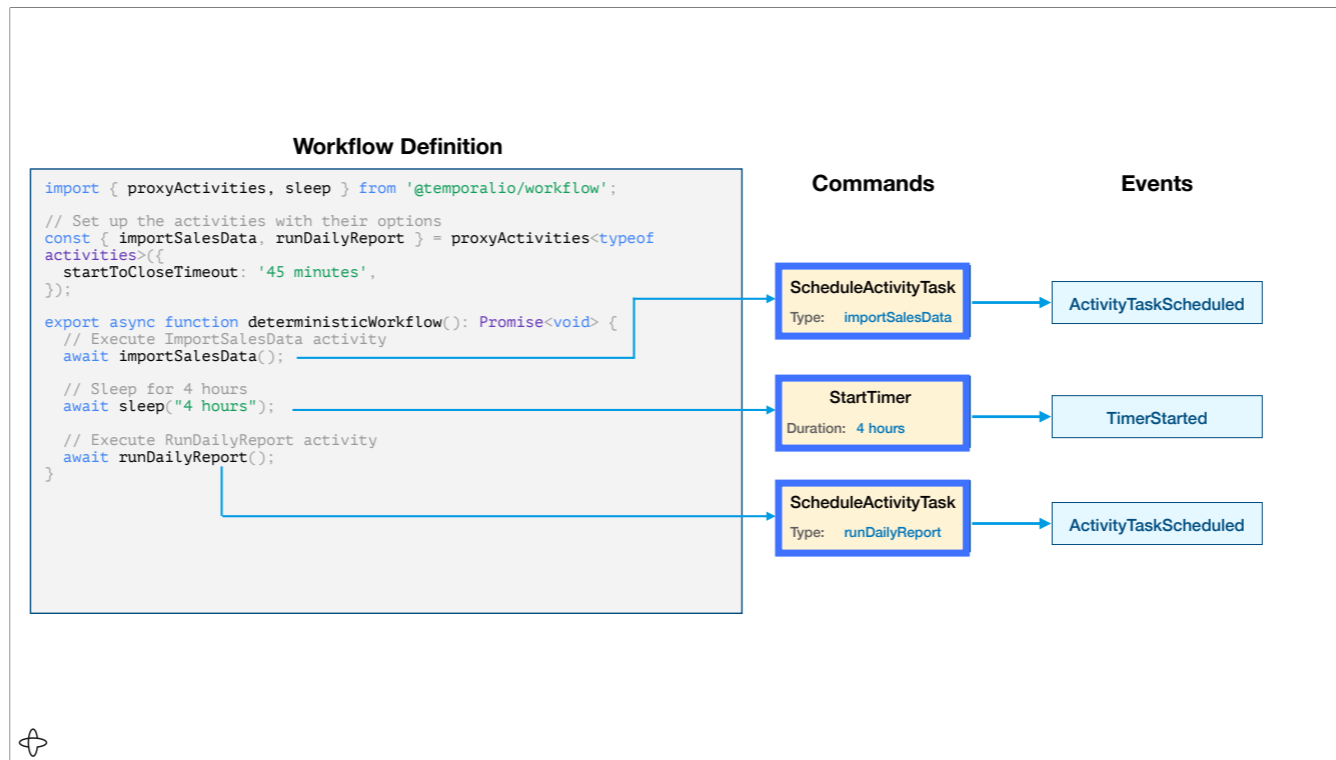


Let's take a look at this workflow definition.



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.



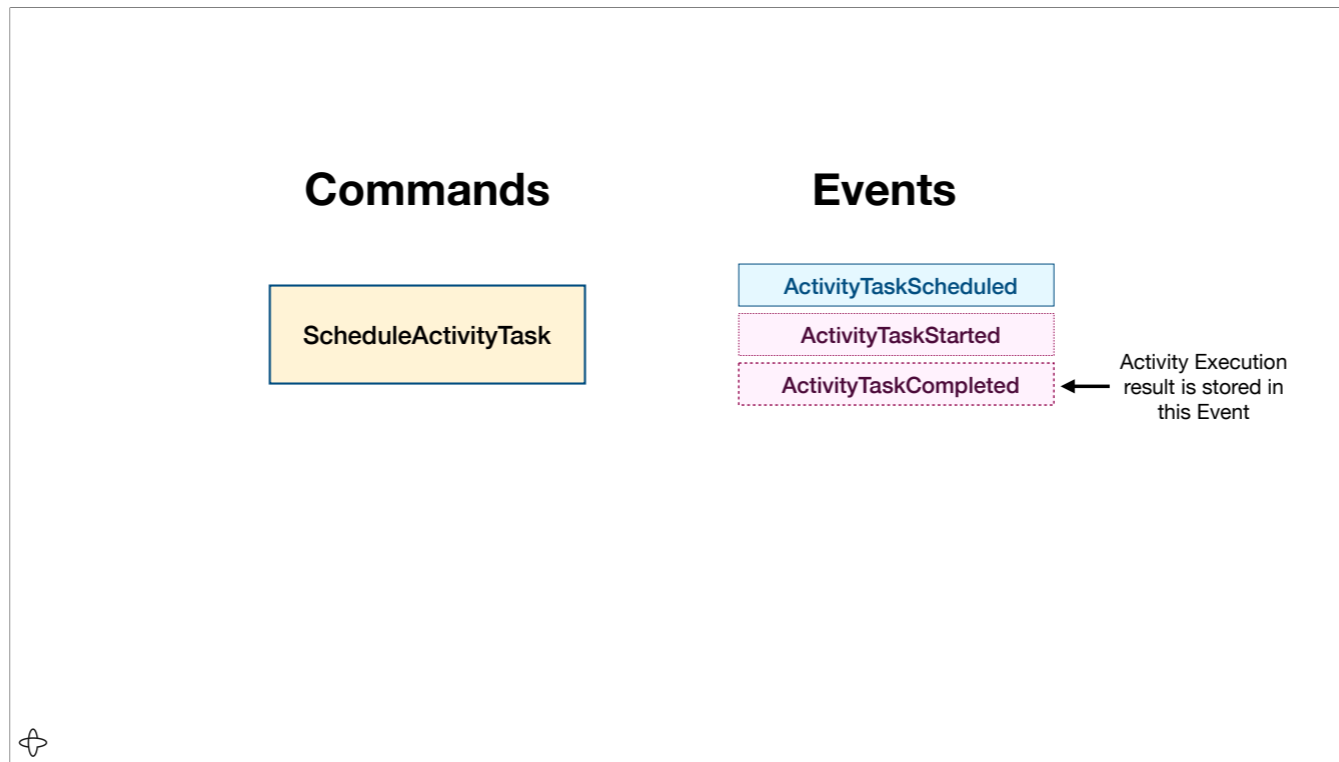
Certain Events in the history are a direct result of a particular Command issued by a Worker.

For example, the `ScheduleActivityTask` Command results in an `ActivityTaskScheduled` Event

[advance]

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.



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

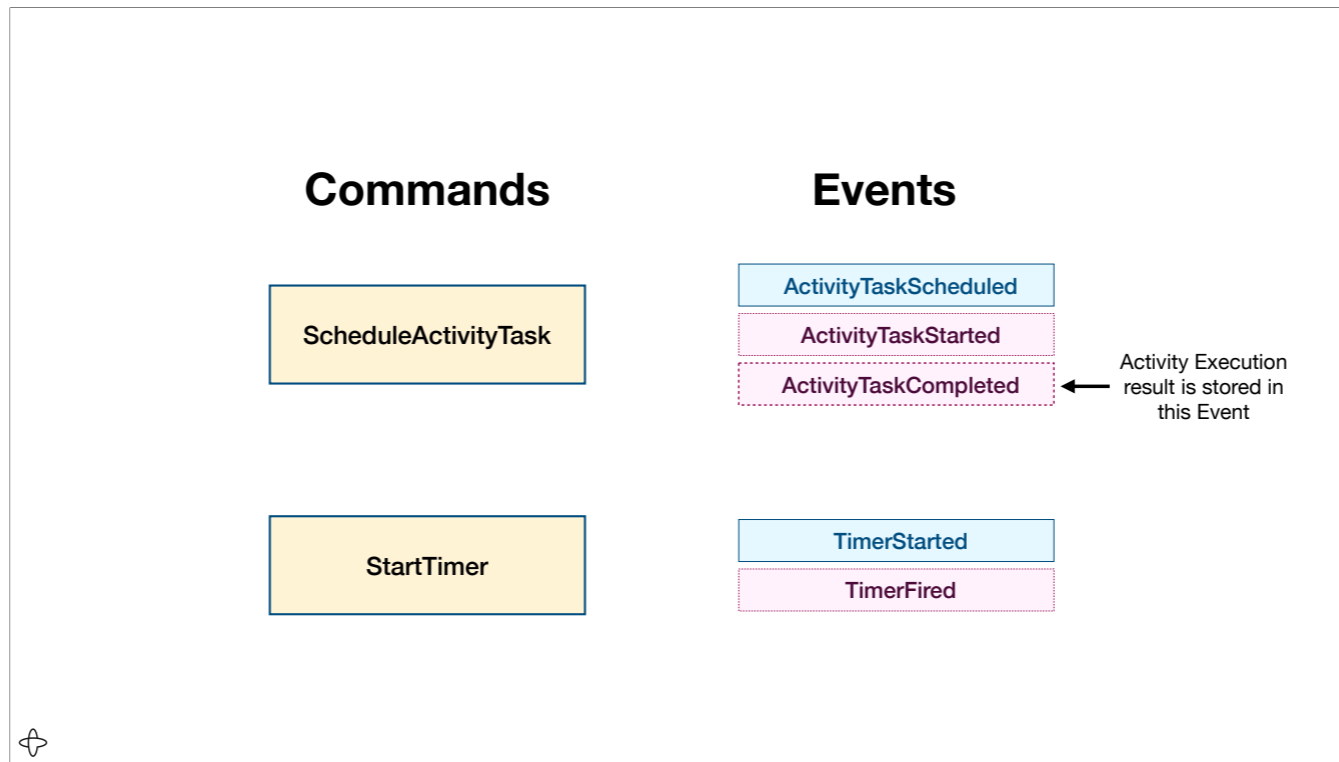
Activity Type, it's clear that this Activity already ran successfully. In this case, the Worker does not issue the Command to the cluster requesting a new execution of the Activity.

[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.



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

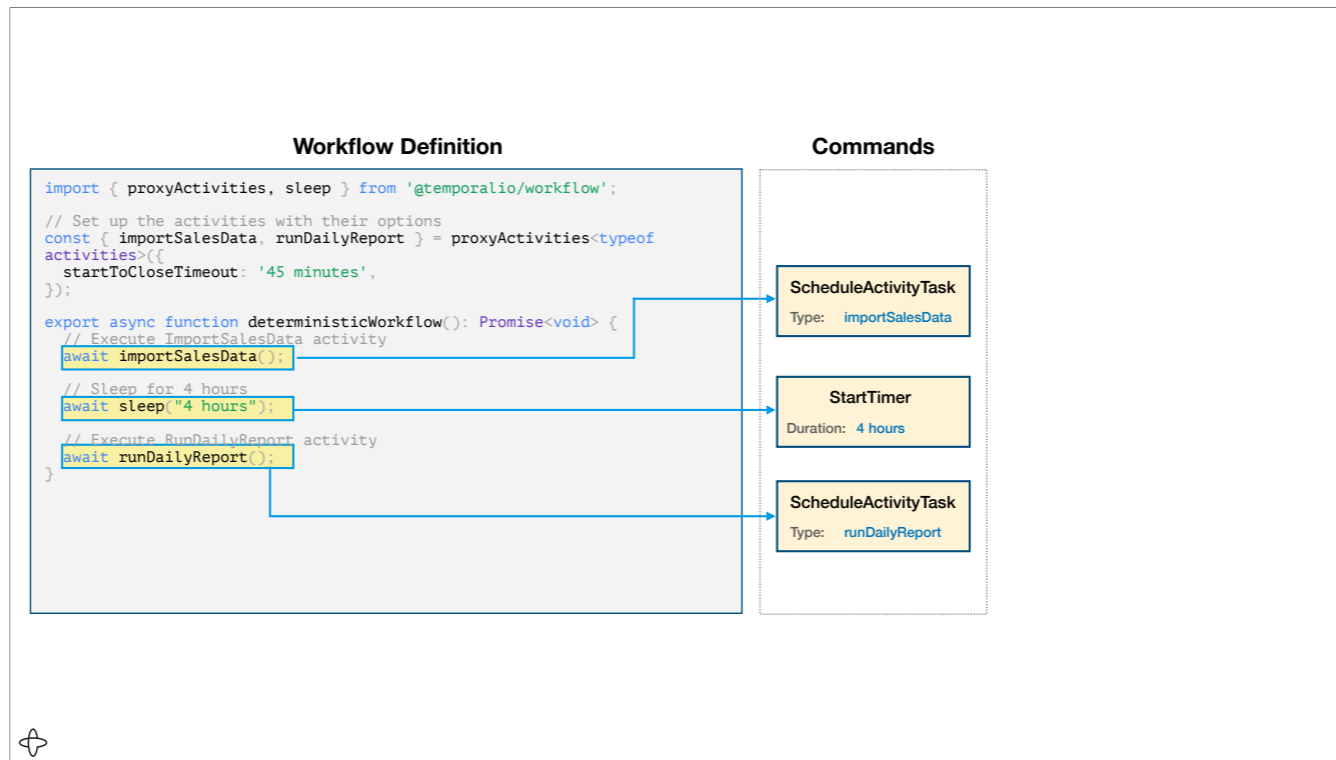
Temporal's ability to guarantee durable execution of your Workflow depends on deterministic Workflows.

Workflow Definition

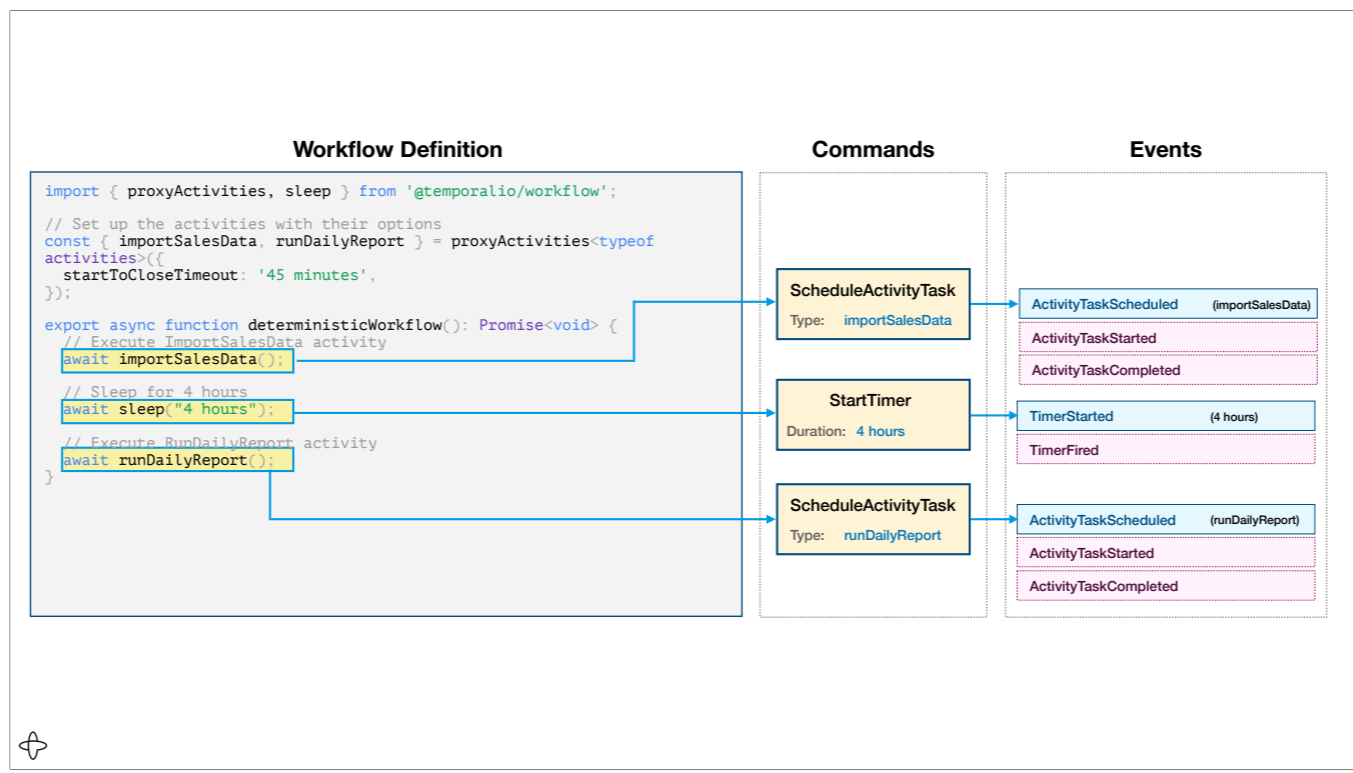
```
import { proxyActivities, sleep } from '@temporalio/workflow';  
  
// Set up the activities with their options  
const { importSalesData, runDailyReport } = proxyActivities<typeof  
activities>({  
  startToCloseTimeout: '45 minutes',  
});  
  
export async function deterministicWorkflow(): Promise<void> {  
  // Execute ImportSalesData activity  
  await importSalesData();  
  
  // Sleep for 4 hours  
  await sleep("4 hours");  
  
  // Execute RunDailyReport activity  
  await runDailyReport();  
}
```



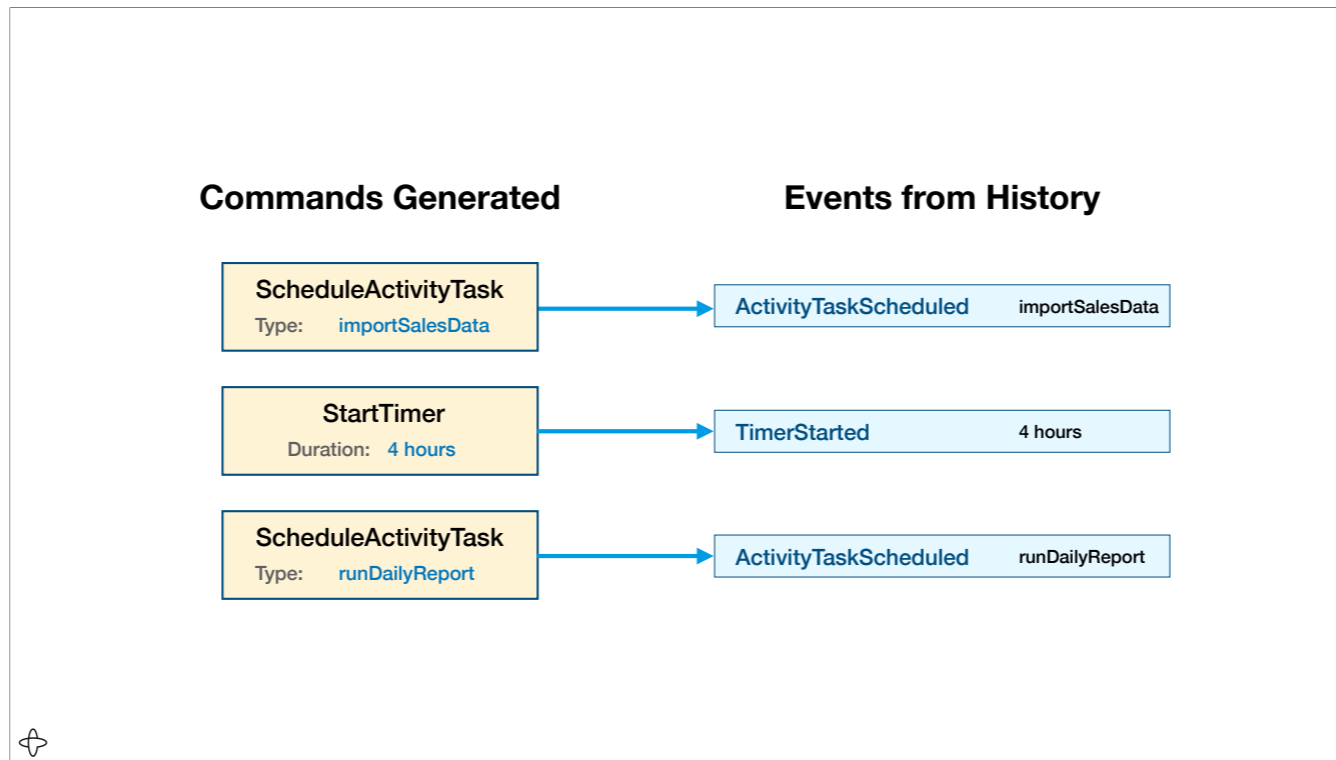
You've already learned that Temporal uses Workflow Replay to recover the state of a Workflow Execution. It does this if the Worker crashes, but it may also do this at other, less predictable times,



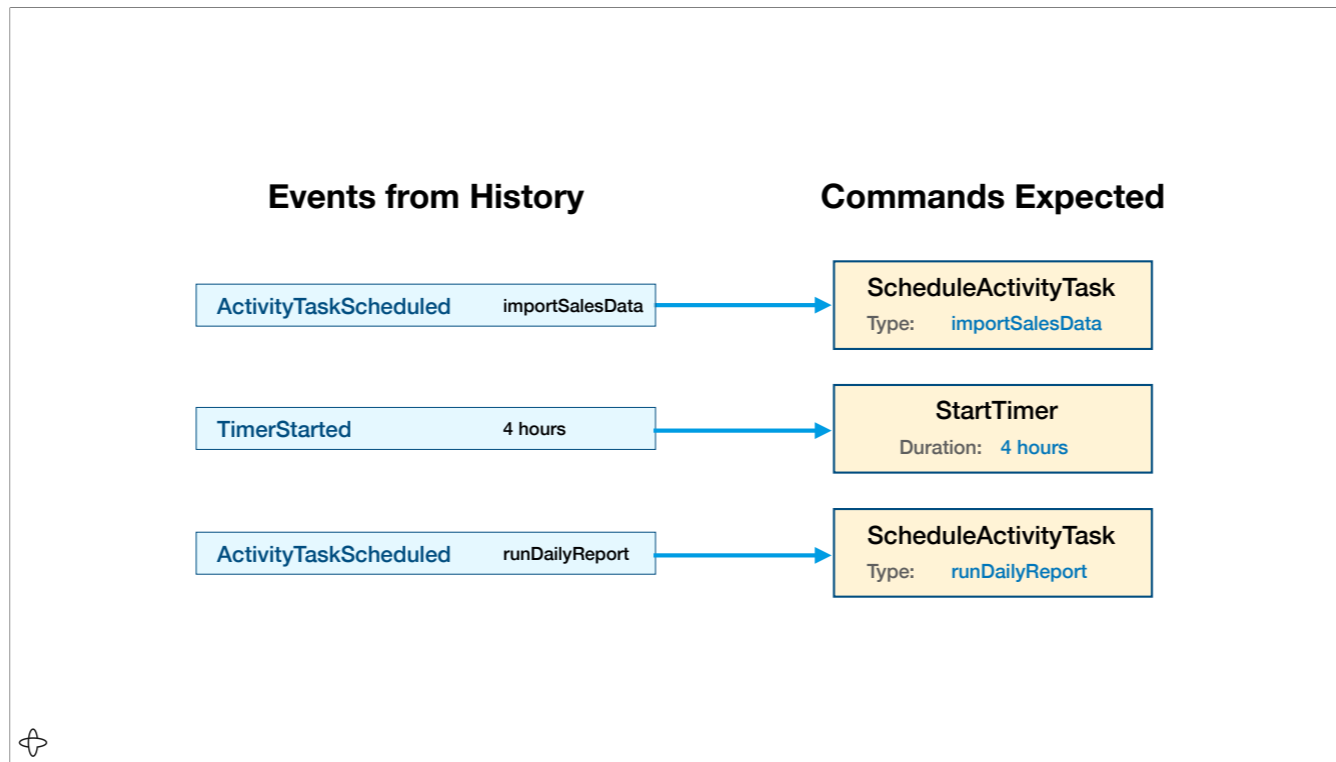
As you've seen, the Worker checks whether a Command created by replaying of the code...



has a corresponding Event in the history, which it uses to determine whether the previous execution reached this point in the code.



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	Commands Created	Relevant Events Logged
<pre>import { proxyActivities, log, sleep } from '@temporalio/workflow'; // Set up the activities with their options const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({ startToCloseTimeout: '45 minutes', }); export async function deterministicWorkflow(): Promise<void> { // Execute ImportSalesData activity await importSalesData(); if (getRandomNumber(1, 100) >= 50) { await sleep("4 hours"); } log.info("Preparing to run daily report", {}); // Execute RunDailyReport activity await runDailyReport(); } // middle square method function getRandomNumber(min:number, max:number) { let seed = 1254; seed = Math.floor(((seed * seed) % 10000) / 100); return min + seed % (max - min + 1); }</pre>		

✚

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 { proxyActivities, log, sleep } from '@temporalio/workflow';  
  
// Set up the activities with their options  
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({  
  startToCloseTimeout: '45 minutes',  
});  
  
export async function deterministicWorkflow(): Promise<void> {  
  // Execute ImportSalesData activity  
  await importSalesData();  
  
  if (getRandomNumber(1, 100) >= 50) {  
    await sleep("4 hours");  
  }  
  
  log.info("Preparing to run daily report", {});  
  
  // Execute RunDailyReport activity  
  await runDailyReport();  
}  
  
// middle square method  
function getRandomNumber(min:number, max:number) {  
  let seed = 1234;  
  seed = Math.floor(((seed * seed) % 10000) / 100);  
  return min + seed % (max - min + 1);  
}
```

Commands Created

ScheduleActivityTask

Type: importSalesData

Relevant Events Logged



In this case, let's say that execution is successful,

A Non-Deterministic Workflow Definition

```
import { proxyActivities, log, sleep } from '@temporalio/workflow';  
  
// Set up the activities with their options  
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({  
  startToCloseTimeout: '45 minutes',  
});  
  
export async function deterministicWorkflow(): Promise<void> {  
  // Execute ImportSalesData activity  
  await importSalesData();  
  
  if (getRandomNumber(1, 100) >= 50) {  
    await sleep("4 hours");  
  }  
  
  log.info("Preparing to run daily report", {});  
  
  // Execute RunDailyReport activity  
  await runDailyReport();  
}  
  
// middle square method  
function getRandomNumber(min:number, max:number) {  
  let seed = 1234;  
  seed = Math.floor(((seed * seed) % 10000) / 100);  
  return min + seed % (max - min + 1);  
}
```

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 { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function deterministicWorkflow(): Promise<void> {
  // Execute ImportSalesData activity
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}


// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}
```

Commands Created

ScheduleActivityTask Type: <code>importSalesData</code>

Relevant Events Logged

ActivityTaskScheduled (<code>importSalesData</code>)
ActivityTaskStarted
ActivityTaskCompleted



Next, there's a conditional statement, which evaluates the value of a randomly-generated number.

A Non-Deterministic Workflow Definition

```

import { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function determineDailyReport() {
  // Execute ImportSalesData
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}

```

Commands Created

Relevant Events Logged

Happens to return 84 during this execution

ScheduleActivityTask

Type: importSalesData

ActivityTaskScheduled	(importSalesData)
ActivityTaskStarted	
ActivityTaskCompleted	

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 { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function deterministicWorkflow(): Promise<void> {
  // Execute ImportSalesData activity
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}
```

Commands Created

ScheduleActivityTask

Type: `importSalesData`

Relevant Events Logged

ActivityTaskScheduled (importSalesData)

ActivityTaskStarted

ActivityTaskCompleted



This contains a `sleep` statement,

A Non-Deterministic Workflow Definition

```
import { proxyActivities, log, sleep } from '@temporalio/workflow';  
  
// Set up the activities with their options  
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({  
  startToCloseTimeout: '45 minutes',  
});  
  
export async function deterministicWorkflow(): Promise<void> {  
  // Execute ImportSalesData activity  
  await importSalesData();  
  
  if (getRandomNumber(1, 100) >= 50) {  
    await sleep("4 hours");  
  }  
  
  log.info("Preparing to run daily report", {});  
  
  // Execute RunDailyReport activity  
  await runDailyReport();  
}  
  
// middle square method  
function getRandomNumber(min:number, max:number) {  
  let seed = 1234;  
  seed = Math.floor(((seed * seed) % 10000) / 100);  
  return min + seed % (max - min + 1);  
}
```

Commands Created

ScheduleActivityTask
Type: `importSalesData`

StartTimer
Duration: `4 hours`

Relevant Events Logged

ActivityTaskScheduled (`importSalesData`)
ActivityTaskStarted
ActivityTaskCompleted



so the Worker issues a Command to the cluster, requesting that it starts a Timer.

A Non-Deterministic Workflow Definition

```
import { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function deterministicWorkflow(): Promise<void> {
  // Execute ImportSalesData activity
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}
```

Commands Created

ScheduleActivityTask

Type: `importSalesData`

StartTimer

Duration: `4 hours`

Relevant Events Logged

`ActivityTaskScheduled` (`importSalesData`)

`ActivityTaskStarted`

`ActivityTaskCompleted`

`TimerStarted` (`4 hours`)

`TimerFired`



The cluster starts the Timer, logs an Event, and then logs another Event when the Timer fires.

A Non-Deterministic Workflow Definition

```

import { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function deterministicWorkflow(): Promise<void> {
  // Execute ImportSalesData activity
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}

```

Worker crashes here

Commands Created

ScheduleActivityTask

Type: `importSalesData`

StartTimer

Duration: `4 hours`

Relevant Events Logged

`ActivityTaskScheduled` (`importSalesData`)

`ActivityTaskStarted`

`ActivityTaskCompleted`

`TimerStarted` (`4 hours`)

`TimerFired`

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 { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function deterministicWorkflow(): Promise<void> {
  // Execute ImportSalesData activity
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}

```

Commands Created

Relevant History Events

ActivityTaskScheduled (ImportSalesData)

ActivityTaskStarted

ActivityTaskCompleted

TimerStarted (4 hours)

TimerFired

Commands Expected (Based on History)

ScheduleActivityTask

Type: ImportSalesData

StartTimer

4 hours

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 { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function deterministicWorkflow(): Promise<void> {
  // Execute ImportSalesData activity
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}
```

Commands Created

ScheduleActivityTask
Type: `importSalesData`

Relevant History Events

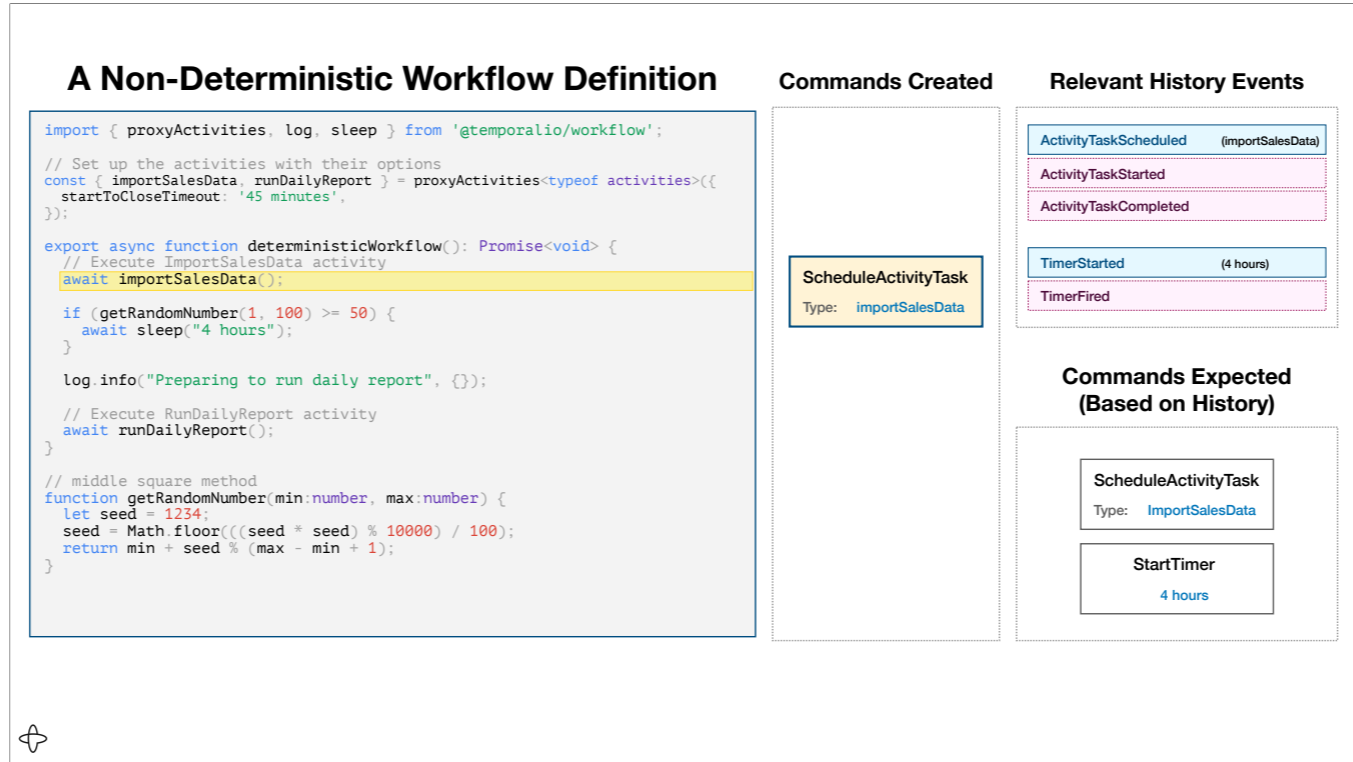
ActivityTaskScheduled (importSalesData)
ActivityTaskStarted
ActivityTaskCompleted

TimerStarted (4 hours)
TimerFired

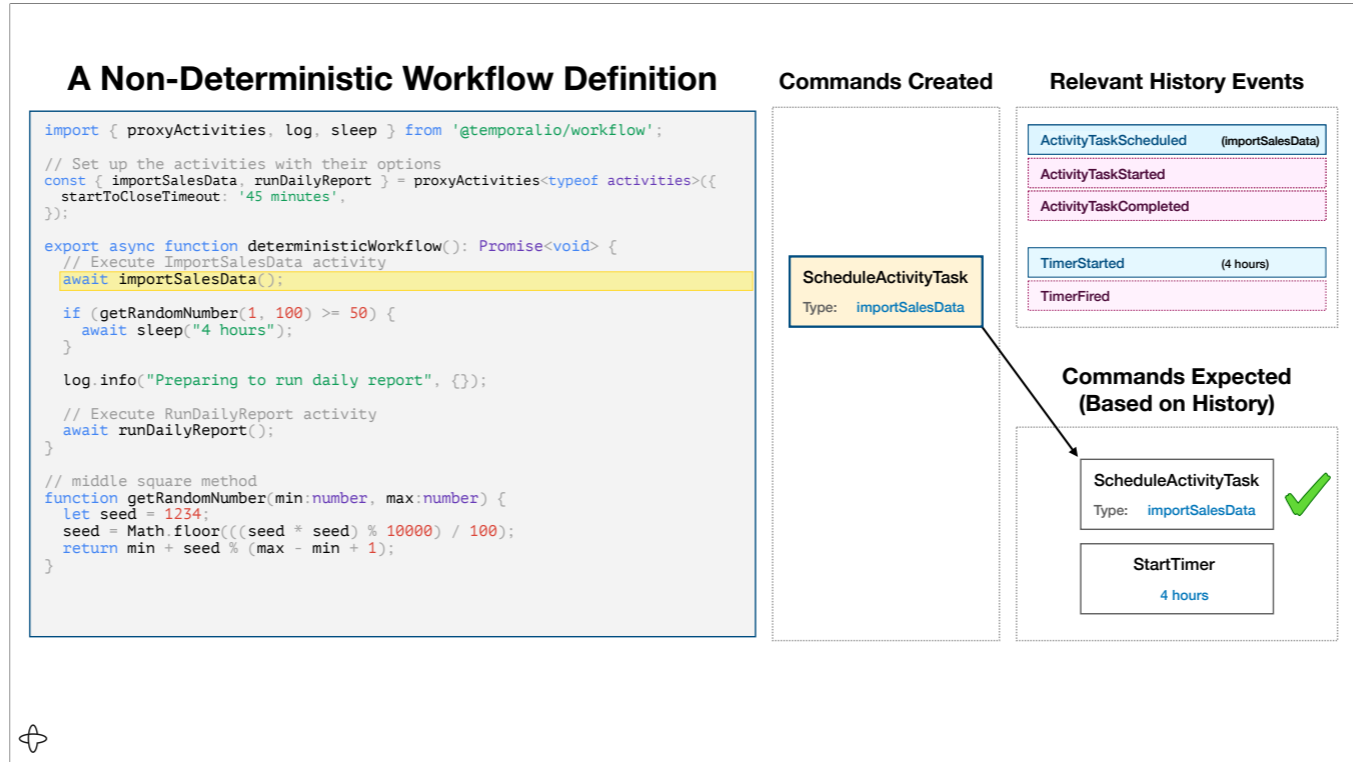
Commands Expected (Based on History)

ScheduleActivityTask
Type: `ImportSalesData`

StartTimer
4 hours



As it executes the code during replay, it reaches the first `activity` call, and creates a `ScheduleActivityTask` Command.



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 { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function deterministicWorkflow(): Promise<void> {
  // Execute ImportSalesData activity
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}

```

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 { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function determineDailyReport() {
  // Execute ImportSalesData
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}

```

Happens to return 14 during this execution

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

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 { proxyActivities, log, sleep } from '@temporalio/workflow';

// Set up the activities with their options
const { importSalesData, runDailyReport } = proxyActivities<typeof activities>({
  startToCloseTimeout: '45 minutes',
});

export async function deterministicWorkflow(): Promise<void> {
  // Execute ImportSalesData activity
  await importSalesData();

  if (getRandomNumber(1, 100) >= 50) {
    await sleep("4 hours");
  }

  log.info("Preparing to run daily report", {});

  // Execute RunDailyReport activity
  await runDailyReport();
}

// middle square method
function getRandomNumber(min:number, max:number) {
  let seed = 1234;
  seed = Math.floor(((seed * seed) % 10000) / 100);
  return min + seed % (max - min + 1);
}

```

Commands Created

ScheduleActivityTask
Type: `importSalesData`

ScheduleActivityTask
Type: `runDailyReport`

Relevant History Events

ActivityTaskScheduled (importSalesData)

ActivityTaskStarted

ActivityTaskCompleted

TimerStarted (4 hours)

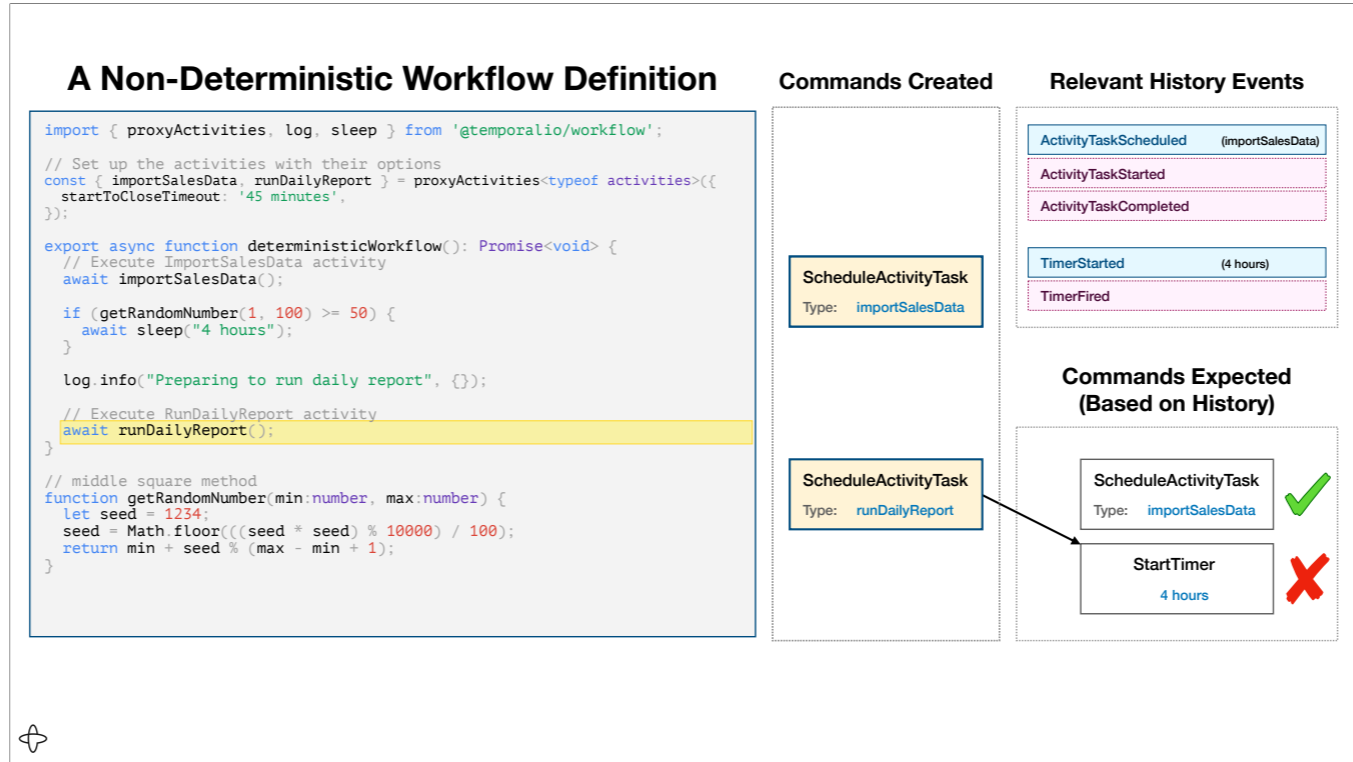
TimerFired

Commands Expected (Based on History)

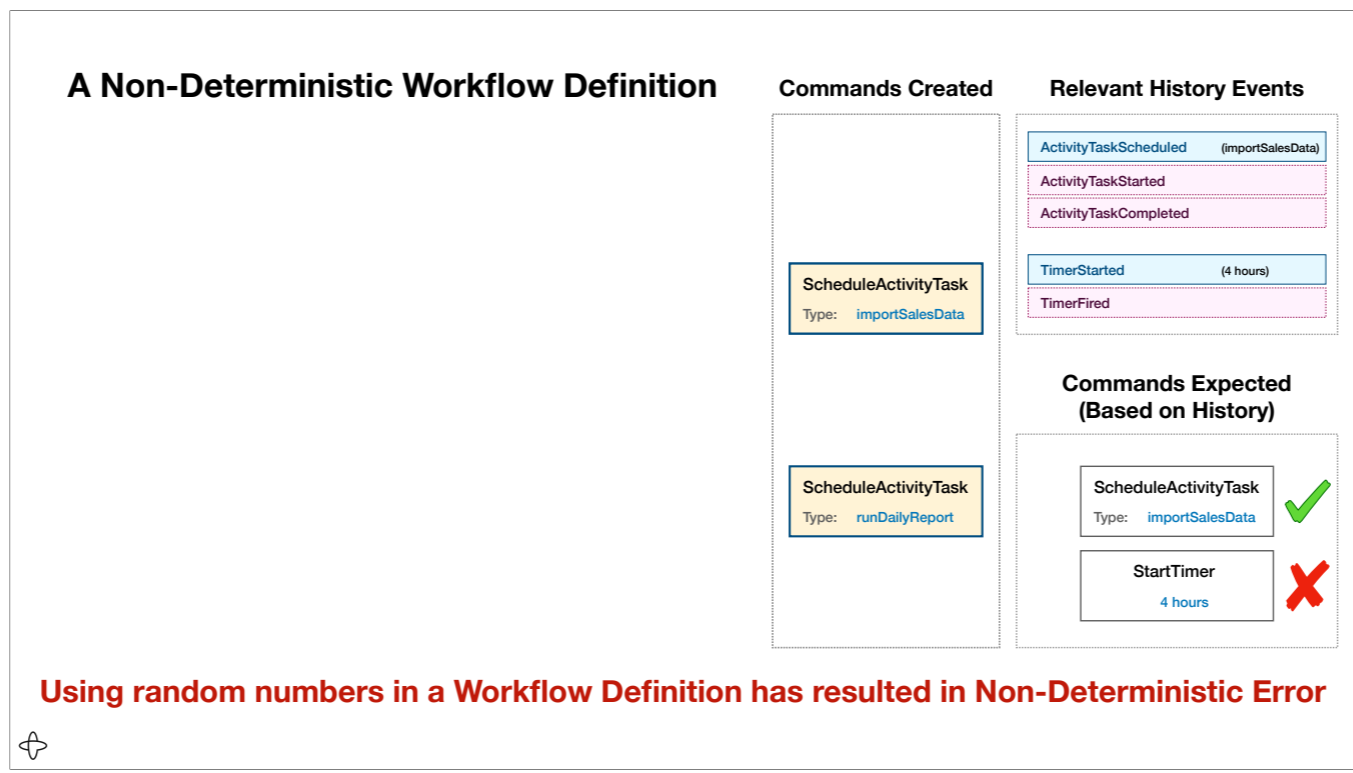
ScheduleActivityTask
Type: `importSalesData` ✓

StartTimer
4 hours

Eventually, the Workflow requests execution of another Activity, so the Worker creates another `ScheduleActivityTask` Command.



However, this is a different Command than it expected to find at this position in the history.



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.

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 like `sleep`
- **Iterating over data structures with unknown ordering**
- **Storing or Evaluating the Run ID of a Workflow Execution**



Here are some things you can do to avoid causing determinism problems

Avoid accessing external systems, such as databases or network services. Do these things in your Activities instead.

Avoid writing business logic or calling functions that rely on system time.

Avoid iterating over data structures with unknown ordering.

Avoid storing or evaluating the Run ID of a Workflow Execution

Certain operations in Temporal, such as a Workflow Execution that uses

Continue-as-New to avoid exceeding limits on Event History size, result in a new run of that same execution. In this case, each run will have the same Workflow ID but will have a unique Run ID.

TypeScript SDK Workflow Sandbox

- The TypeScript SDK runs your code in a sandbox that will help you check for non-deterministic code
 - The code is bundled on Worker creation using Webpack, and can import any package as long as it does not reference Node.js or DOM APIs
 - To make the Workflow runtime deterministic, functions like **Math.random()**, **Date**, and **setTimeout()** are replaced by deterministic versions



The TypeScript SDK runs your code in a sandbox that will help you check for non-deterministic code

The code is bundled on Worker creation using Webpack, and can import any package as long as it does not reference Node.js or DOM APIs

To make the Workflow runtime deterministic, functions like `Math.random()`, `Date`, and `setTimeout()` are replaced by deterministic versions

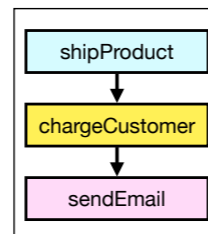
How Workflow Changes Can Lead to Non-Deterministic Errors



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 that's more subtle...
- **Consider the following scenario**
 - You deploy and execute the following Workflow, which calls three Activities...



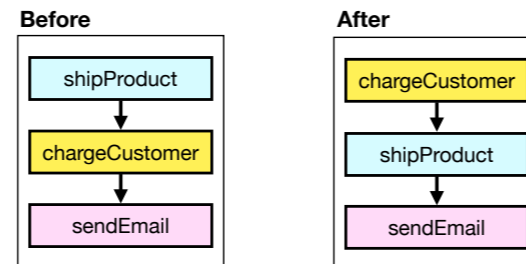
As you've just learned, non-deterministic code can cause problems. But there's another problem that's more subtle. Imagine you've deployed and execute the following Workflow that calls three activities in sequence:

shipProduct
chargeCustomer
sendEmail

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?**



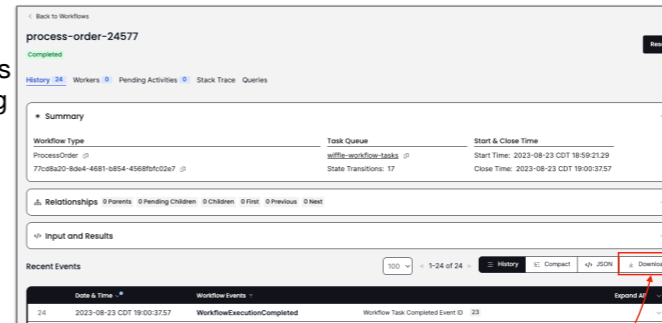
While that workflow is running, you decide to update the code and change the order of the activities. You decide to charge the customer first before shipping the product. What happens to the open executions when you restart the workers to deploy your changes?

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.

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.

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 with the Patch API
 - Worker Versioning



You can test for this by replaying the history of your previous executions using your new code before you deploy. You can download the history from the Web UI or from the command line client and then create tests that play this back.

Once you've determined the problem, you can solve it by using the Patch API to change running workflows, or by using the Worker Versioning feature, 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 upcoming online version of this course).

Temporal 102

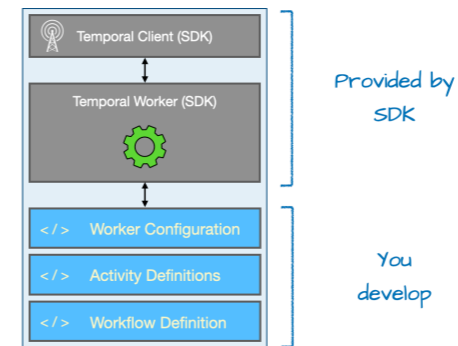
- 00. About this Workshop
- 01. Understanding Key Concepts in Temporal
- 02. Improving Your Temporal Application Code
- 03. Using Timers in a Workflow Definition
- 04. Testing Your Temporal Application Code
- 05. Understanding Event History
- 06. Debugging Workflow History
- 07. Deploying Your Application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



Temporal applications contain code that you develop, including your 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 that execute Activities or calls to create Timers with sleep, or when returning a result from the Workflow Definition
- **Commands sent by the Worker lead to Events logged by Temporal Cluster / Cloud**



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 that execute Activities or calls to create Timers with 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



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



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



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 objects (not individual fields) 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



Recommended best practices for Temporal app development

Use objects (not individual fields) as input/output of your Workflow and Activity definitions

Be aware of the platform's limits on Event History size and item count

Replace non-deterministic code in Workflow Definitions with Workflow-safe counterparts

Use Temporal's replay-aware logging API

Essential Points (7)

- **We don't dictate how to build, deploy, or run Temporal applications**
 - Typical advice: Build, deploy, and run as you would any other application in that language
 - However, we recommend running ≥ 2 Workers per Task Queue (availability/scalability)



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