Crafting an Error Handling Strategy in Go

Version - 20241216

Crafting an Error Handling Strategy

▶ **00. About this Workshop**

01. Error Handling Concepts

02. Returning and Handling Errors

03. Timeouts

04. Retry Policies

05. Recovering from Failure

06. Conclusion

Logistics

- Introductions
- Schedule
- Facilities
- WiFi
- Asking questions
- Getting help with exercises

During this course, you will

- Recommend an error handling strategy
  - Explain how Temporal represents errors
  - Compare platform errors to application errors
  - Explain differences between timeouts and failures
  - Determine when it is appropriate to fail a Workflow Execution and when to fail an Activity Execution
- Implement an error handling strategy
  - Explain how Temporal handles retries
  - Apply a custom Retry Policy to Workflow and Activity Execution
  - Customize a Retry Policy for execution of a specific Activity
  - Determine when an error should be retried or deemed non-retryable
  - Define specific errors as non-retryable error types
- Integrate appropriate mechanisms for handling various types of errors
  - Implement Activity Heartbeating to detect failure in a long running Activity
  - Track Activity Execution progress using Heartbeat messages
  - Use Termination and Cancellation to end a Workflow Execution
  - Implement the Saga pattern to restore external state following failure in a Workflow Execution

Exercise Environment

- **We provide a development environment for you in this course**
  - 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)

GitPod link: https://t.mp/edu-errstrat-go-exercises

Network: Replay2025

Password: Durable!

GitPod Overview

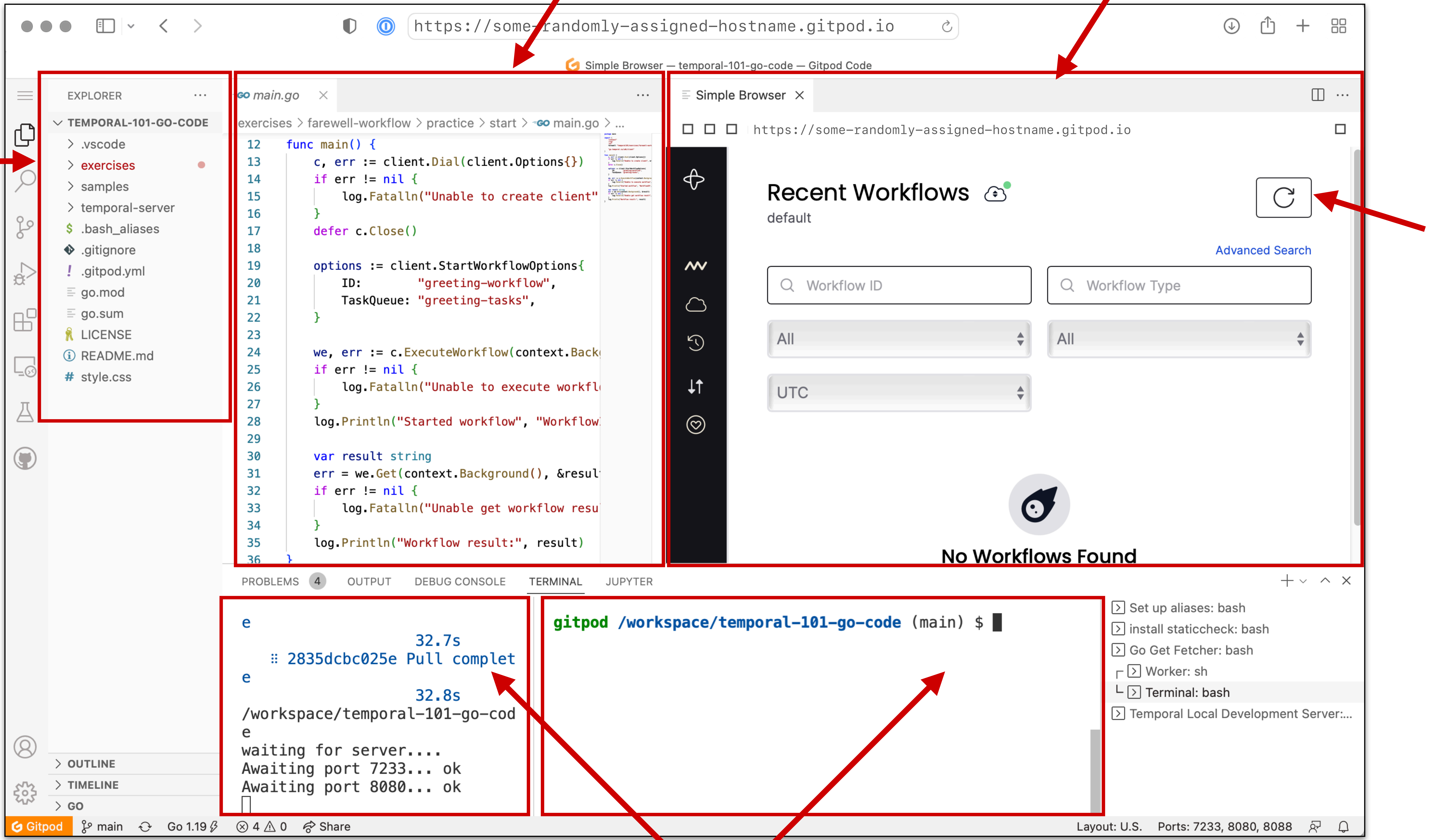
Code editor

Embedded browser
(displays Temporal Web UI)

File browser
source code for exercises

Refresh button
(for Web UI)

Terminals



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

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

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

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

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

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

Crafting an Error Handling Strategy

00. About this Workshop

▶ **01. Error Handling Concepts**

02. Returning and Handling Errors

03. Timeouts

04. Retry Policies

05. Recovering from Failure

06. Conclusion

Failures in a Temporal Application

- **Temporal guarantees Durable Execution for your Workflows**
  - Ensures that they run to completion despite adverse conditions, such as process termination
  - Despite running to completion, failures may still occur during Workflow Execution
- **Application developers are still responsible for handling failures**
  - You must identify when they occur, using clues such as errors and timeouts
  - You must determine how to mitigate them, perhaps through retries or conditional logic
- **Each failure belongs to one of two categories: Platform or Application**

Platform Failures

- **Occur for reasons outside the application’s control**
  - For example, a problem with a server or network
- **Platform failures generally resolve themselves after retrying**
- **Classification: Is the *platform* capable of detecting and mitigating this?**
  - Example: A microservice call that fails due to network outage is a platform failure
    - The platform can detect the outage when the request times out
    - The platform can mitigate it by retrying the call
    - Neither detection nor mitigation requires knowledge of the application itself

Application Failures

- **Occur due to problems in the application’s code or input data**
- **Retries generally do not resolve application failures**
- **Detection and mitigation require knowledge about the application**
  - Example: order processing fails due to expired payment card
    - No matter how many retries you perform, the card will still be expired
    - Application can detect this failure based on the error code returned by payment processor
    - Can mitigate by canceling the order, notifying customer, and returning items to inventory

Backward and Forward Recovery

- **Application failures often involve *backward recovery***
  - Backward recovery: Attempt to fix problem reverting previous change(s) in state
  - Example: Compensating transaction
- **Platform failures often involve *forward recovery***
  - Forward recovery: Attempt to fix problem by continuing processing from the point of failure
  - Example: Retrying a failed operation

The Temporal Error Model

- **Remember that Temporal supports polyglot programming**
- **If an Activity returns an error, it must be surfaced to the Workflow**
  - This must work regardless of which SDKs are used to implement the Activity or Workflow
- **As with data, errors transcend language boundaries in Temporal**
  - Errors are serialized using a language-neutral format (protobuf)

Instructor-Led Demo

The Temporal Error Model

Conceptual Types of Failures

- **Assign to one of three categories based on likelihood of reoccurrence**
  1. Transient
  2. Intermittent
  3. Permanent
- **This classification will help you to define an appropriate Retry Policy**

Transient Failures

- **Existence of past failure does not increase likelihood of future failures**
- **These are generally one-off failures that occur by chance**
  - For example, an administrator reboots a router just as you make a network request
  - Resolve a transient failure by retrying the operation after a short delay

Intermittent Failures

- **Existence of past failure increases likelihood of future failures**
- **These are caused by a problem that *eventually* resolves itself**
  - For example, calling a rate-limited service fails because you’ve issued too many requests
  - Resolve an intermittent failure through retries, but with a longer delay
  - Using a backoff coefficient to increase delay between retries can avoid overloading the system

Permanent Failures

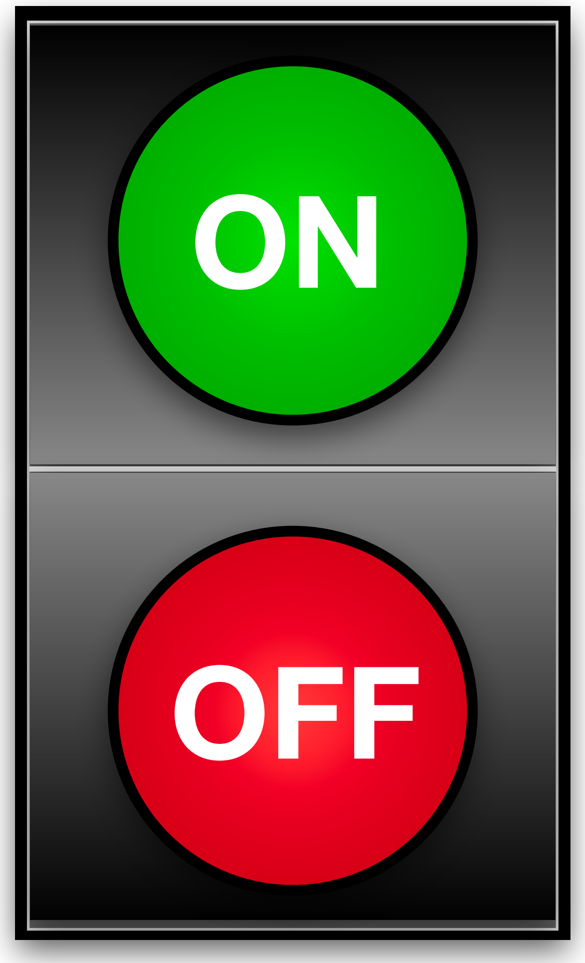
- **Existence of past failure guarantees likelihood of future failures**
- **These are caused by a problem that will *never* resolve itself**
  - For example, sending an e-mail notification fails due to an invalid address
  - Permanent failures require manual repair—you cannot resolve them through retries alone

Idempotence

- An operation is idempotent if subsequent invocations do not adversely change state beyond that of the initial invocation

- Consider the idempotence of buttons used to control device power

Toggle Button



Separate On/Off Buttons

Activity Idempotence

- **It is strongly recommended that you make your Activities idempotent**
  - A non-idempotent Activity could adversely affect the state of the system
- **For example, consider an Activity that performs the following steps**
  1. Queries a database
  2. Calls a microservice using data returned by the query
  3. Writes the result of the microservice call to the filesystem
- **This will be retried if any one of those steps fails**
  - You should balance the granularity of your Activities with the need to keep Event History small

Idempotence and At-Least-Once Execution

- **Idempotence is also important due to an edge case in distributed systems**
- **Consider the following scenario**
  - Worker polls the Temporal Service and accepts an Activity Task
  - Worker begins executing the Activity
  - Worker finishes executing the Activity
  - Worker crashes just before reporting the result to the Temporal Service
- **Activity will be retried since Event History does not indicate completion**
  - Therefore, idempotence is essential for preventing unwanted changes in application state

Idempotency Keys

- **You can achieve idempotency by ignoring duplicate requests**
  - This raises a question: How can one distinguish a *duplicate* request from one that looks similar?
- **Idempotency keys are unique identifiers associated with a request**
  - They are interpreted by the system receiving the request (e.g., a payment processor)
  - In a Temporal Activity, you can compose one from a Workflow Run ID and Activity ID
  - Guaranteed to be consistent across retry attempts, but unique among Workflow Executions

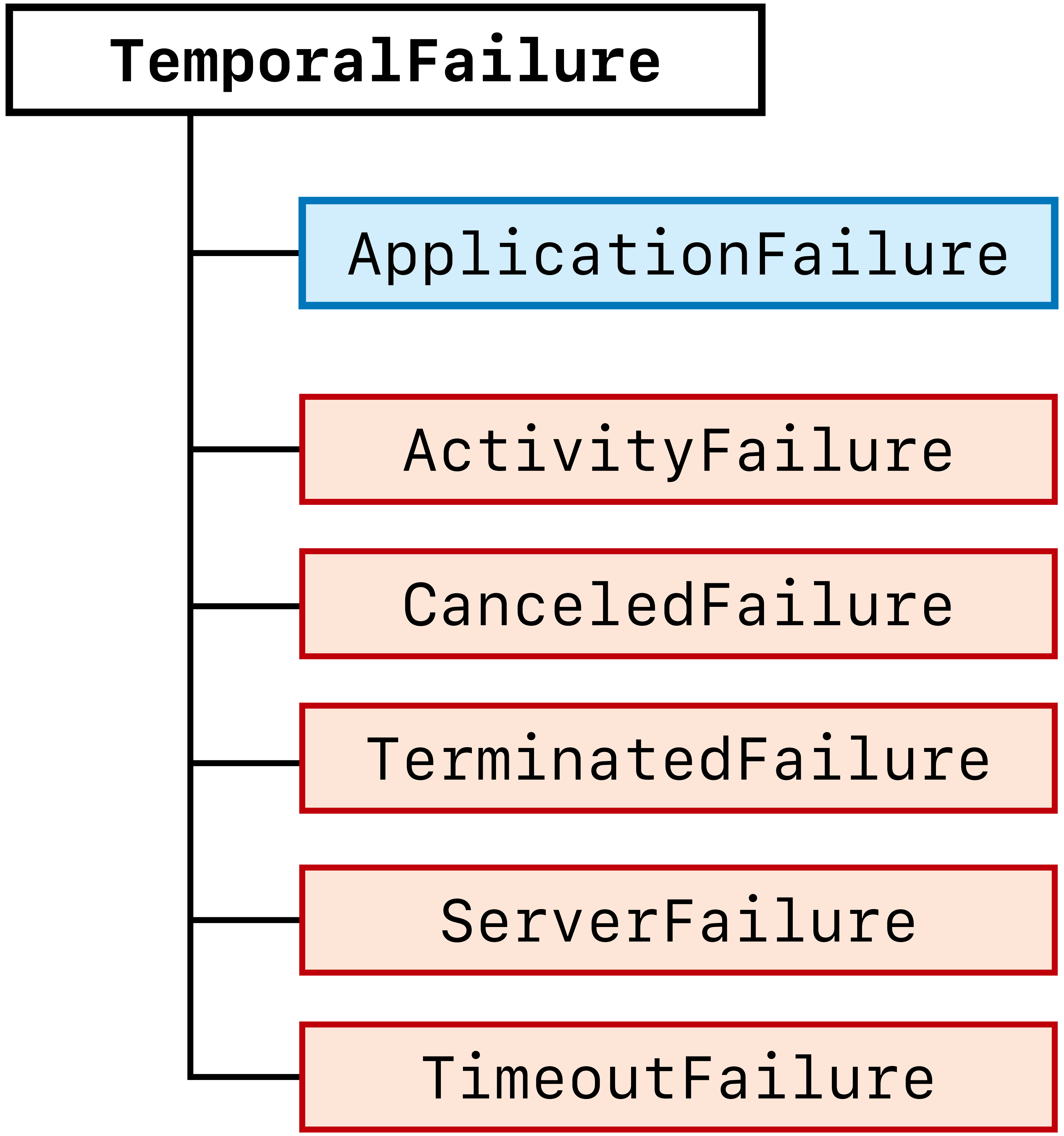
```java
import io.temporal.activity.Activity;
import io.temporal.activity.ActivityExecutionContext;

ActivityExecutionContext context = Activity.getExecutionContext();
String idempotencyKey = context.getInfo().getRunId() + "-" context.getInfo().getActivityId();
```

How Temporal Represents Failures

- **All failures in Temporal are represented in the API as a Temporal Failure**
- **You can use custom error types meaningful to your application**
  - For example, `InvalidCreditCardError` or `UserNotFoundError`
- **An error thrown by an Activity is surfaced as an `ActivityFailure`**
  - You can catch and handle it in your Workflow Definition, if desired

Examples of Temporal Failure Types



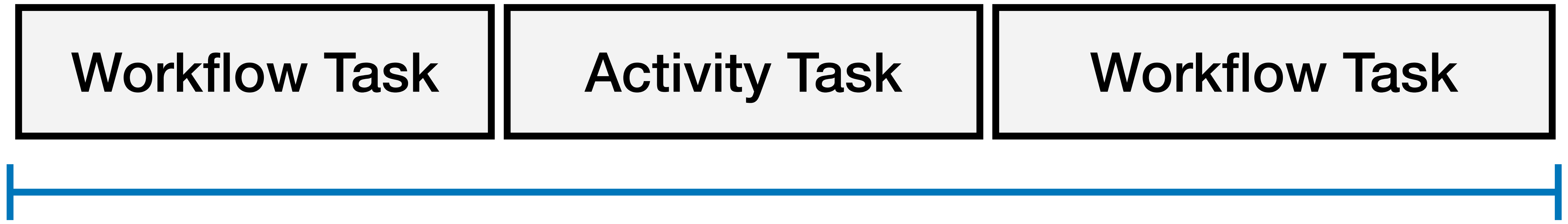
`TemporalFailure`	Base class, which represents failures that can cross Workflow and Activity boundaries
`ApplicationFailure`	**The only failure that should be thrown by user code.** Used to communicate application-specific failure
`ActivityFailure`	Indicates that an Activity failed to complete as expected
`CanceledFailure`	Indicates that the operation was canceled
`TerminatedFailure`	Indicates that the operation was terminated
`ServerFailure`	Indicates a failure originating in the Temporal Service
`TimeoutFailure`	Indicates that the Activity did not complete within its configured Timeout period

Failure Converter

- **Temporal invokes a Failure Converter when an error is returned**
  - The `FailureConverter` interface defines two methods
    - One serializes a `Throwable` into a Failure protobuf message
    - The other deserializes a Failure protobuf message into an instance of `TemporalFailure`
- **Temporal provides a default Failure Converter implementation**
  - It works well and we recommend it in virtually all cases
  - It is *possible*, though very rarely necessary, to create a custom Failure Converter
    - One of the few use cases is to redact sensitive information that appears in error messages

Workflow Task vs. Workflow Execution

- **Before we continues, let's review two important terms with similar names**
- **Workflow Execution**
  - The sequence of steps that result from executing a Workflow Definition
- **Workflow Task**
  - Drives progress for a *specific portion* of the Workflow Execution



A Workflow Execution may span multiple Workflow Tasks

When a Workflow Task Failure Is Retried…

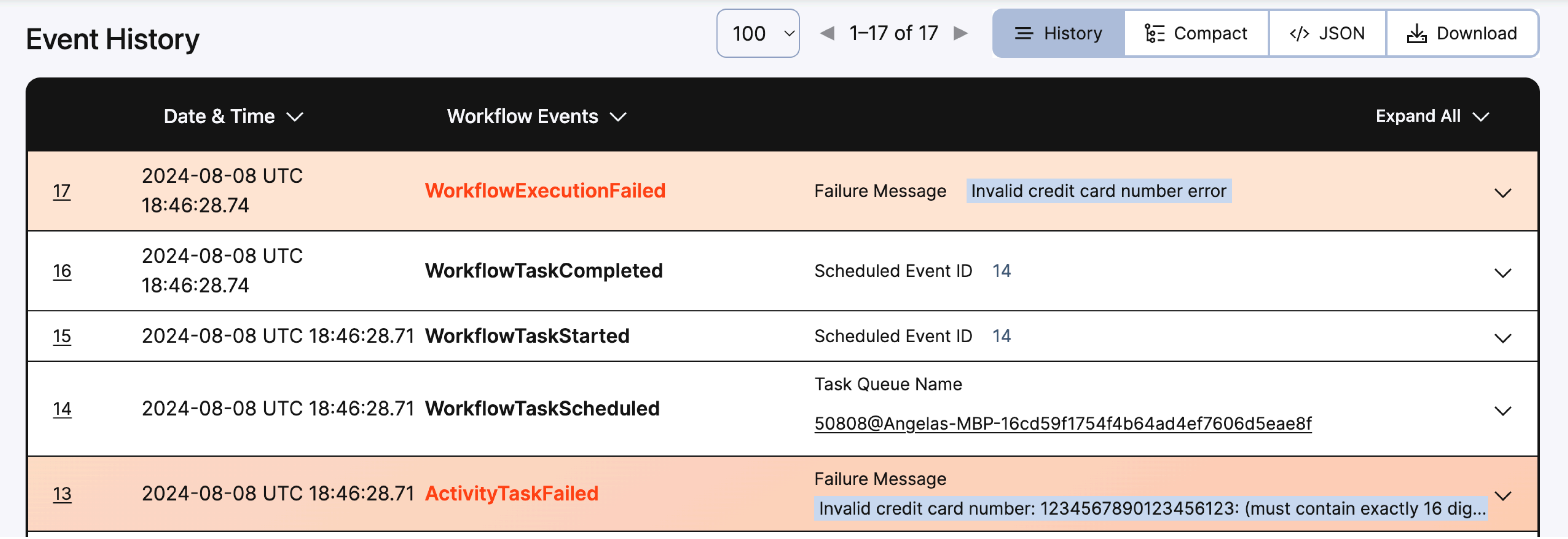
- **Worker that handled the Task evicts that Workflow Execution from cache**
  - This is a safety mechanism, since it's considered to be in an unknown state
  - The Temporal Service schedules a new Workflow Task
- **Worker that picks up the new Task must recreate state before continuing**
  - It first downloads the Event History from the Temporal Service
  - It then uses History Replay to reconstruct the previous state of the execution
  - Execution continues once this is complete

Workflow Execution Failures

- Returning an Error from a Workflow, or letting an Error propagate unhandled out of the Workflow function, will either cause a Workflow Task Failure or a Workflow Execution Failure
  - Workflow Task Failure: Happens when the Workflow calls `panic`. Temporal will automatically retry the task.
  - Workflow Execution Failure: Happens when the Workflow returns an Error. This causes a permanent, unsuccessful completion of Workflow Execution.

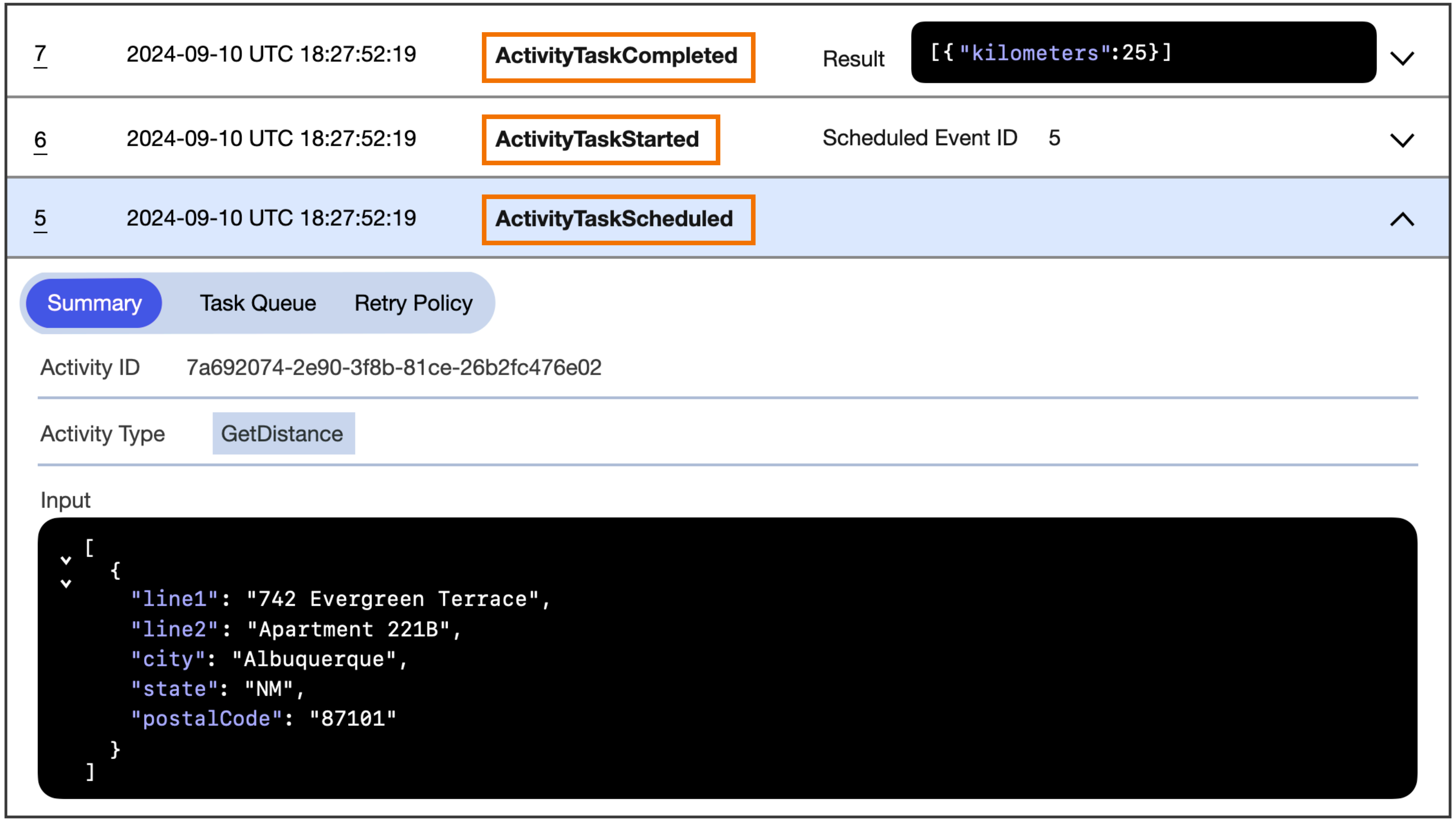
Workflow Execution Failure

- An Activity failure will never directly cause a Workflow Execution failure



	Date & Time	Workflow Events	Expand All
17	2024-08-08 UTC 18:46:28.74	WorkflowExecutionFailed	Failure Message Invalid credit card number error
16	2024-08-08 UTC 18:46:28.74	WorkflowTaskCompleted	Scheduled Event ID 14
15	2024-08-08 UTC 18:46:28.71	WorkflowTaskStarted	Scheduled Event ID 14
14	2024-08-08 UTC 18:46:28.71	WorkflowTaskScheduled	Task Queue Name 50808@Angelas-MBP-16cd59f1754f4b64ad4ef7606d5eae8f
13	2024-08-08 UTC 18:46:28.71	ActivityTaskFailed	Failure Message Invalid credit card number: 1234567890123456123: (must contain exactly 16 dig...

Activity Execution: Sequence of Events (1)



7	2024-09-10 UTC 18:27:52:19	ActivityTaskCompleted	Result	`[{"kilometers":25}]`
6	2024-09-10 UTC 18:27:52:19	ActivityTaskStarted	Scheduled Event ID	5
5	2024-09-10 UTC 18:27:52:19	ActivityTaskScheduled		

Activity ID 7a692074-2e90-3f8b-81ce-26b2fc476e02

Activity Type GetDistance

Input

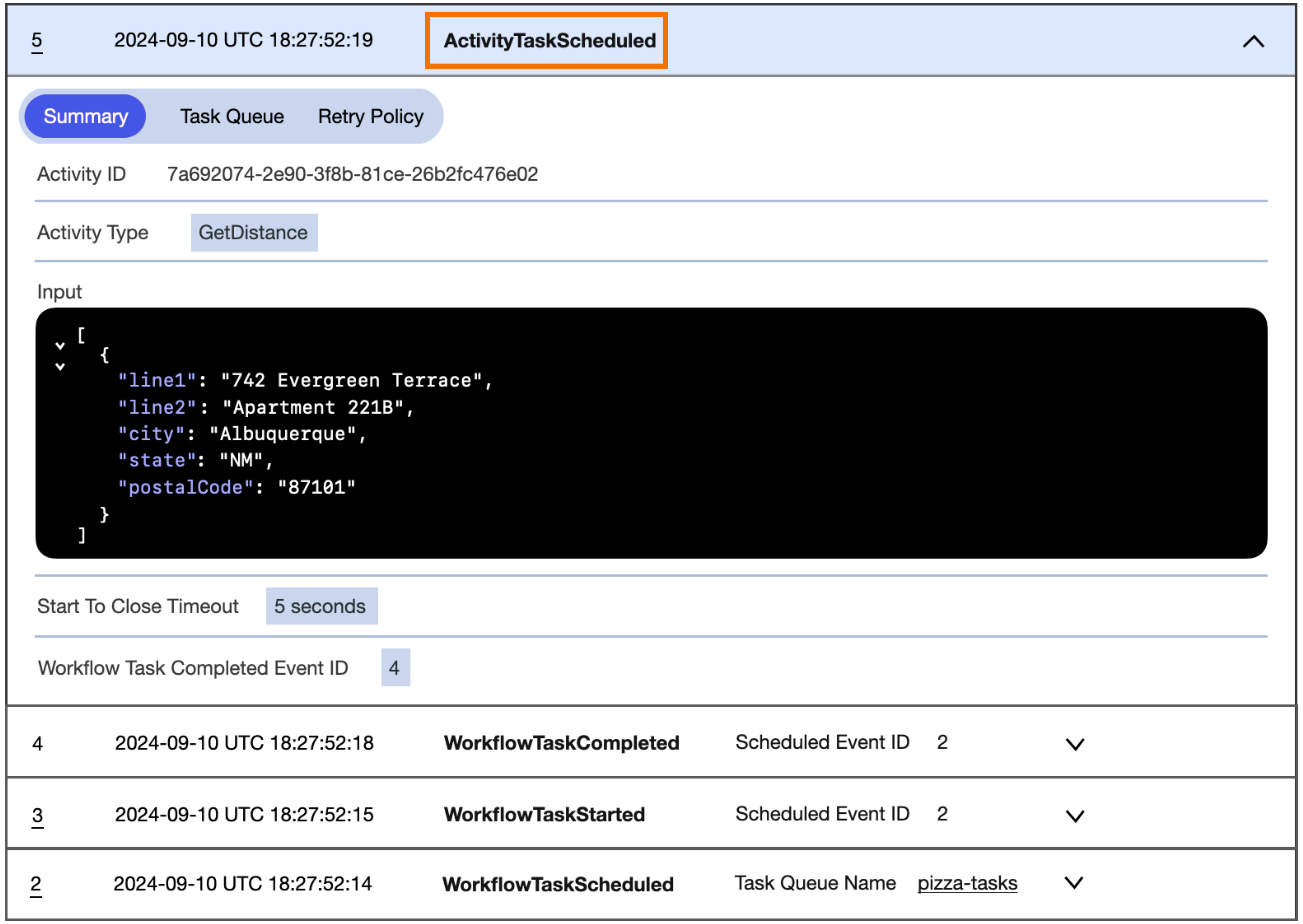
```
[
  {
    "line1": "742 Evergreen Terrace",
    "line2": "Apartment 221B",
    "city": "Albuquerque",
    "state": "NM",
    "postalCode": "87101"
  }
]
```

Activity Execution: Sequence of Events (2)

Order	Event Type	Event Description
1	ActivityTaskScheduled	Temporal Service adds the Activity Task to the Task Queue
2	ActivityTaskStarted	Worker accepts the Activity Task; it's removed from the Task Queue)
3	ActivityTaskCompleted	Worker reports result of Activity Execution to the Temporal Service

Viewing an Activity Execution (1)

- **`ActivityTaskScheduled` is the most recent Event visible for a running Activity**
  - You might have expected the `ActivityTaskStarted` Event
  - The `ActivityTaskStarted` Event is not written until the Activity Execution closes



5 2024-09-10 UTC 18:27:52:19 **ActivityTaskScheduled**

Activity ID 7a692074-2e90-3f8b-81ce-26b2fc476e02

Activity Type GetDistance

Input

```
[
  {
    "line1": "742 Evergreen Terrace",
    "line2": "Apartment 221B",
    "city": "Albuquerque",
    "state": "NM",
    "postalCode": "87101"
  }
]
```

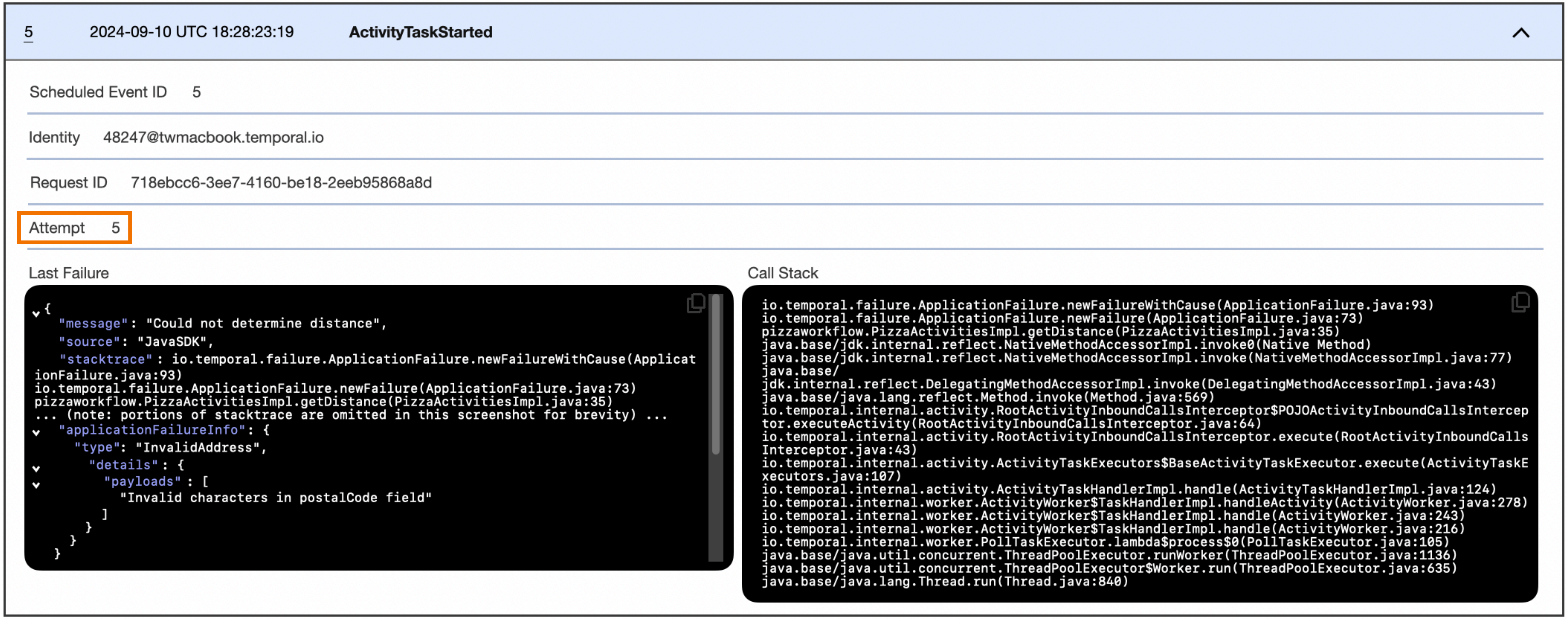
Start To Close Timeout 5 seconds

Workflow Task Completed Event ID 4

4	2024-09-10 UTC 18:27:52:18	**WorkflowTaskCompleted**	Scheduled Event ID	2
3	2024-09-10 UTC 18:27:52:15	**WorkflowTaskStarted**	Scheduled Event ID	2
2	2024-09-10 UTC 18:27:52:14	**WorkflowTaskScheduled**	Task Queue Name	pizza-tasks

Viewing an Activity Execution (2)

- The `ActivityTaskStarted` Event contains the retry attempt count



5 2024-09-10 UTC 18:28:23:19 **ActivityTaskStarted**

Scheduled Event ID 5

Identity 48247@twmacbook.temporal.io

Request ID 718ebcc6-3ee7-4160-be18-2eeb95868a8d

Attempt 5

Last Failure

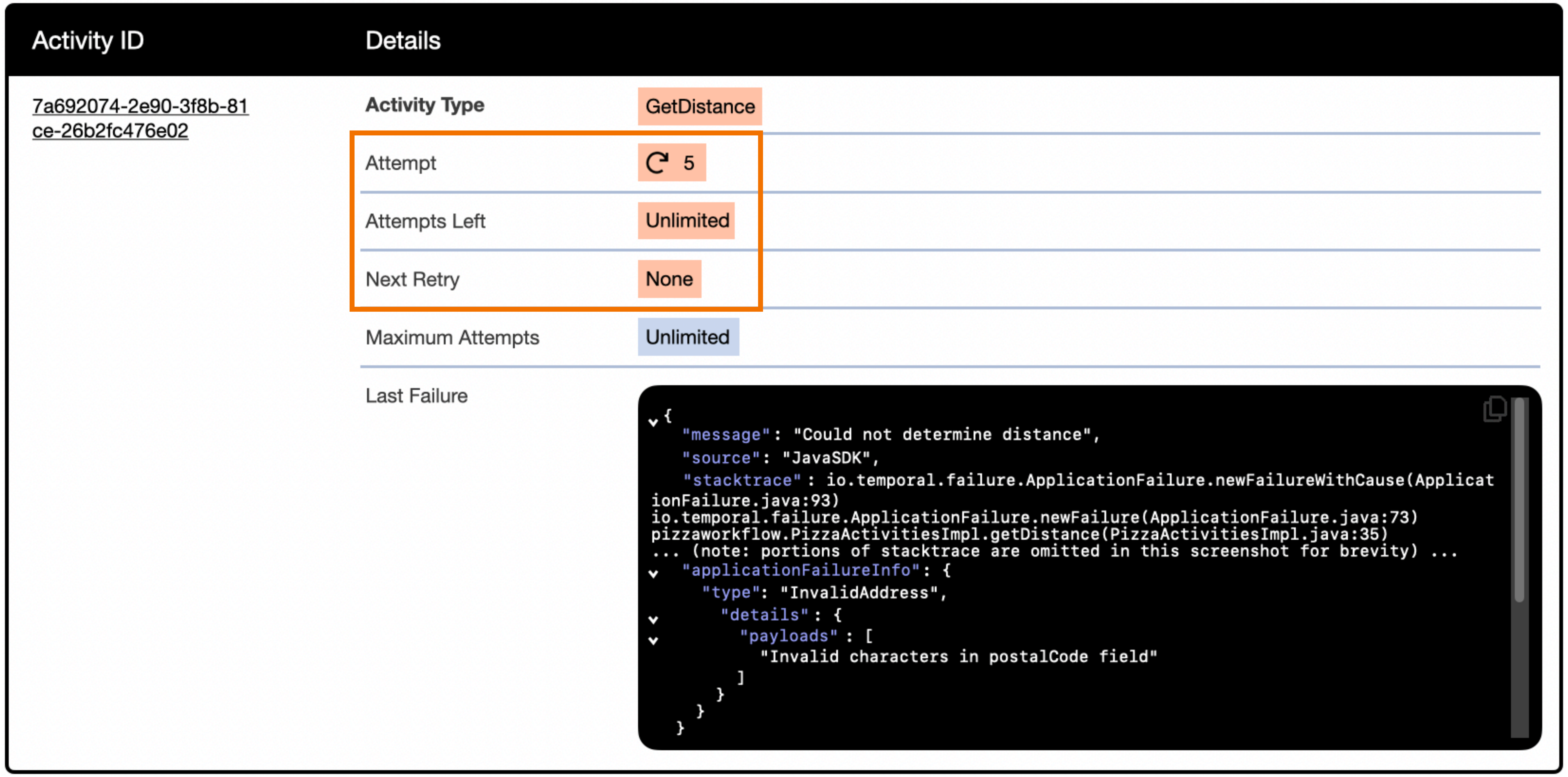
```
{
  "message": "Could not determine distance",
  "source": "JavaSDK",
  "stacktrace" : io.temporal.failure.ApplicationFailure.newFailureWithCause(Applicat
ionFailure.java:93)
io.temporal.failure.ApplicationFailure.newFailure(ApplicationFailure.java:73)
pizzaworkflow.PizzaActivitiesImpl.getDistance(PizzaActivitiesImpl.java:35)
... (note: portions of stacktrace are omitted in this screenshot for brevity) ...
  "applicationFailureInfo": {
    "type": "InvalidAddress",
      "details": {
        "payloads" : [
          "Invalid characters in postalCode field"
        ]
      }
    }
  }
```

Call Stack

```
io.temporal.failure.ApplicationFailure.newFailureWithCause(ApplicationFailure.java:93)
io.temporal.failure.ApplicationFailure.newFailure(ApplicationFailure.java:73)
pizzaworkflow.PizzaActivitiesImpl.getDistance(PizzaActivitiesImpl.java:35)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)
java.base/
jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
java.base/java.lang.reflect.Method.invoke(Method.java:569)
io.temporal.internal.activity.RootActivityInboundCallsInterceptor$POJOActivityInboundCallsIntercep
tor.executeActivity(RootActivityInboundCallsInterceptor.java:64)
io.temporal.internal.activity.RootActivityInboundCallsInterceptor.execute(RootActivityInboundCalls
Interceptor.java:43)
io.temporal.internal.activity.ActivityTaskExecutors$BaseActivityTaskExecutor.execute(ActivityTaskE
xecutors.java:107)
io.temporal.internal.activity.ActivityTaskHandlerImpl.handle(ActivityTaskHandlerImpl.java:124)
io.temporal.internal.worker.ActivityWorker$TaskHandlerImpl.handleActivity(ActivityWorker.java:278)
io.temporal.internal.worker.ActivityWorker$TaskHandlerImpl.handle(ActivityWorker.java:243)
io.temporal.internal.worker.ActivityWorker$TaskHandlerImpl.handle(ActivityWorker.java:216)
io.temporal.internal.worker.PollTaskExecutor.lambda$process$0(PollTaskExecutor.java:105)
java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1136)
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:635)
java.base/java.lang.Thread.run(Thread.java:840)
```

Viewing an Activity Execution (3)

- **The Web UI's "Pending Activities" section details ongoing retry attempts**
  - This is visible during Activity Execution—use it to check if your Activity is failing (and why)

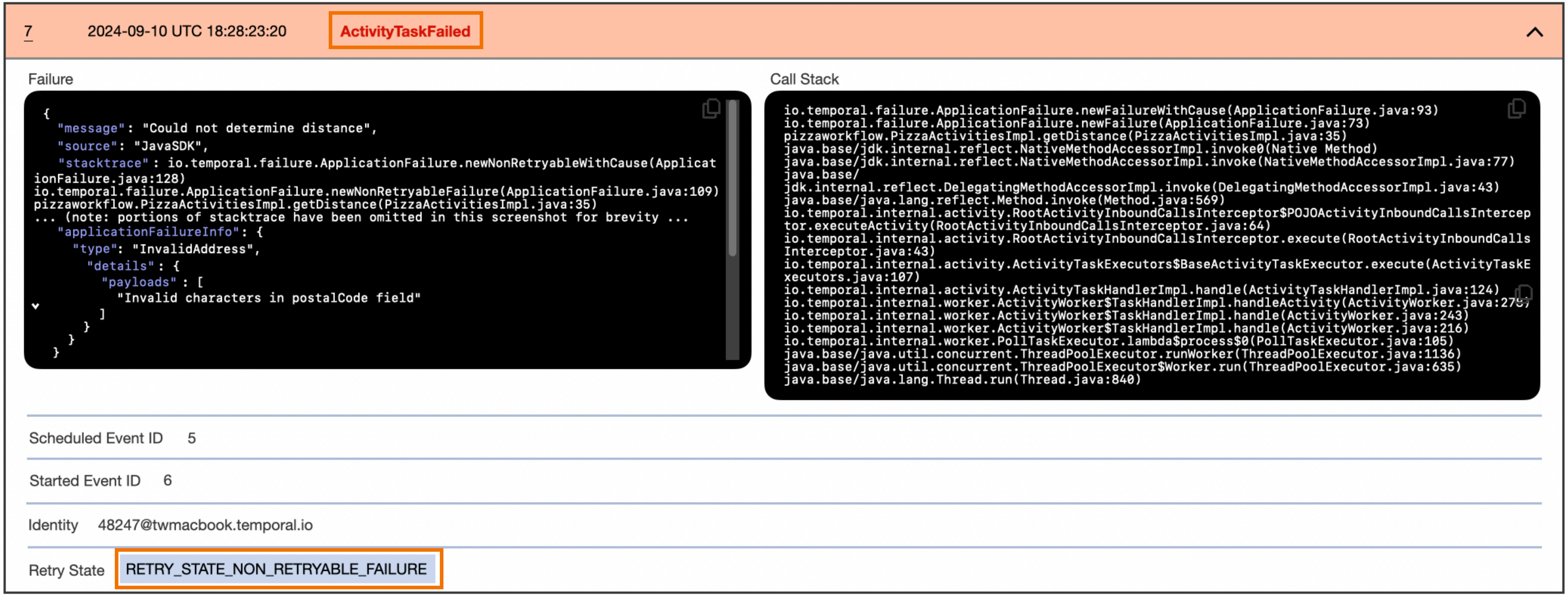


Activity ID	Details	
7a692074-2e90-3f8b-81ce-26b2fc476e02	**Activity Type**	GetDistance
	Attempt	⟳ 5
	Attempts Left	Unlimited
	Next Retry	None
	Maximum Attempts	Unlimited
	Last Failure	

```
{
   "message": "Could not determine distance",
   "source": "JavaSDK",
   "stacktrace" : io.temporal.failure.ApplicationFailure.newFailureWithCause(Applicat
ionFailure.java:93)
io.temporal.failure.ApplicationFailure.newFailure(ApplicationFailure.java:73)
pizzaworkflow.PizzaActivitiesImpl.getDistance(PizzaActivitiesImpl.java:35)
... (note: portions of stacktrace are omitted in this screenshot for brevity) ...
   "applicationFailureInfo": {
     "type": "InvalidAddress",
       "details" : {
         "payloads" : [
           "Invalid characters in postalCode field"
         ]
       }
     }
   }
```

Viewing an Activity Execution (4)

- **The `ActivityTaskFailed` Event provides details after the fact**



7 2024-09-10 UTC 18:28:23:20 ActivityTaskFailed

Failure

```
 {
   "message": "Could not determine distance",
   "source": "JavaSDK",
   "stacktrace" : io.temporal.failure.ApplicationFailure.newNonRetryableWithCause(Applicat
ionFailure.java:128)
io.temporal.failure.ApplicationFailure.newNonRetryableFailure(ApplicationFailure.java:109)
pizzaworkflow.PizzaActivitiesImpl.getDistance(PizzaActivitiesImpl.java:35)
... (note: portions of stacktrace have been omitted in this screenshot for brevity ...
   "applicationFailureInfo": {
     "type": "InvalidAddress",
       "details": {
         "payloads" : [
           "Invalid characters in postalCode field"
         ]
       }
     }
  }
```

Call Stack

```
io.temporal.failure.ApplicationFailure.newFailureWithCause(ApplicationFailure.java:93)
io.temporal.failure.ApplicationFailure.newFailure(ApplicationFailure.java:73)
pizzaworkflow.PizzaActivitiesImpl.getDistance(PizzaActivitiesImpl.java:35)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)
java.base/
jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
java.base/java.lang.reflect.Method.invoke(Method.java:569)
io.temporal.internal.activity.RootActivityInboundCallsInterceptor$POJOActivityInboundCallsIntercep
tor.executeActivity(RootActivityInboundCallsInterceptor.java:64)
io.temporal.internal.activity.RootActivityInboundCallsInterceptor.execute(RootActivityInboundCalls
Interceptor.java:43)
io.temporal.internal.activity.ActivityTaskExecutors$BaseActivityTaskExecutor.execute(ActivityTaskE
xecutors.java:107)
io.temporal.internal.activity.ActivityTaskHandlerImpl.handle(ActivityTaskHandlerImpl.java:124)
io.temporal.internal.worker.ActivityWorker$TaskHandlerImpl.handleActivity(ActivityWorker.java:278)
io.temporal.internal.worker.ActivityWorker$TaskHandlerImpl.handle(ActivityWorker.java:243)
io.temporal.internal.worker.ActivityWorker$TaskHandlerImpl.handle(ActivityWorker.java:216)
io.temporal.internal.worker.PollTaskExecutor.lambda$process$0(PollTaskExecutor.java:105)
java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1136)
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:635)
java.base/java.lang.Thread.run(Thread.java:840)
```

Scheduled Event ID 5

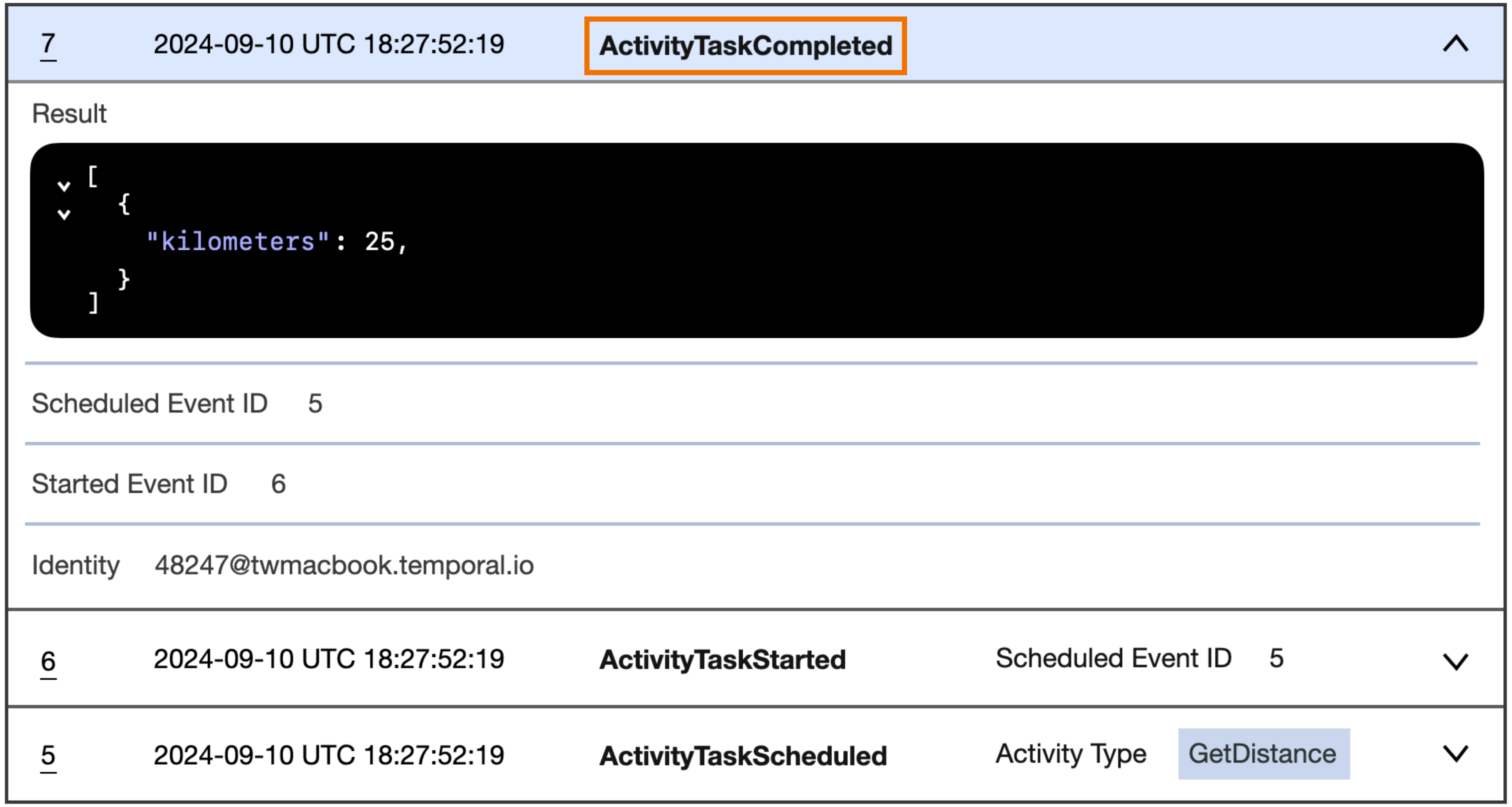
Started Event ID 6

Identity 48247@twmacbook.temporal.io

Retry State RETRY_STATE_NON_RETRYABLE_FAILURE

Viewing an Activity Execution (5)

- The `ActivityTaskCompleted` Event includes the result of execution



7 2024-09-10 UTC 18:27:52:19 **ActivityTaskCompleted**

Result

```
[
  {
    "kilometers": 25,
  }
]
```

Scheduled Event ID 5

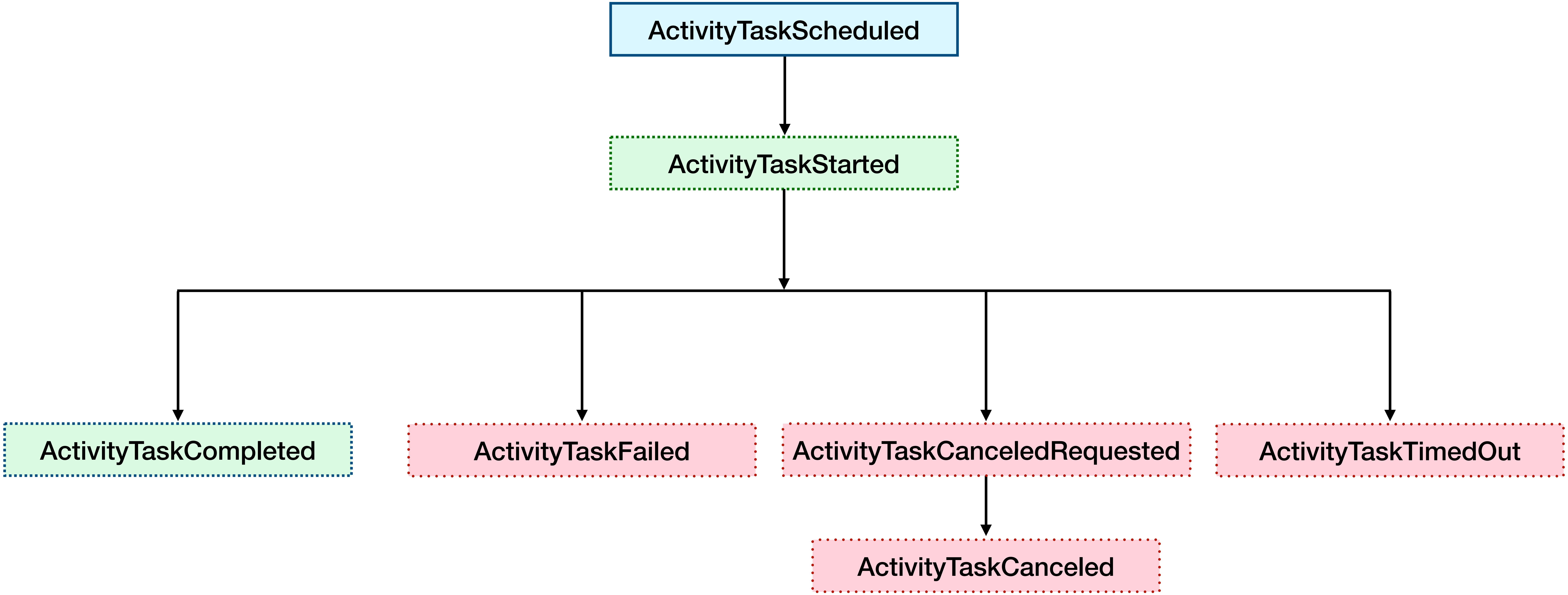
Started Event ID 6

Identity 48247@twmacbook.temporal.io

6 2024-09-10 UTC 18:27:52:19 **ActivityTaskStarted** Scheduled Event ID 5

5 2024-09-10 UTC 18:27:52:19 **ActivityTaskScheduled** Activity Type GetDistance

Events Related to Activity Execution



ActivityTaskScheduled

ActivityTaskStarted

ActivityTaskCompleted

ActivityTaskFailed

ActivityTaskCanceledRequested

ActivityTaskTimedOut

ActivityTaskCanceled

Error Handling Concepts Summary (1)

- **You can categorize failures are either platform or application**
  - **Platform**: occur from reasons beyond the control of your application code
  - **Application**: caused by problems with application code or input data
  - Determine which by considering if detecting and fixing requires knowledge of the application
- **You can also classify them according to likelihood of reoccurrence**
  - **Transient**: Not likely to happen again (handle by retrying with a short delay)
  - **Intermittent**: Likely to happen again (handle by retrying with a longer and increasing delay)
  - **Permanent**: Guaranteed to happen again (handling these will require manual intervention)

Error Handling Concepts Summary (2)

- **Idempotency is a general concern for distributed systems**
  - Will multiple invocations of your operation result in adverse changes to application state?
  - This is a concern for Activities in Temporal, since they may be executed multiple times
  - Temporal strongly recommends that you ensure your Activities are idempotent.

Crafting an Error Handling Strategy

00. About this Workshop

01. Error Handling Concepts

▶ **02. Returning and Handling Errors**

03. Timeouts

04. Retry Policies

05. Recovering from Failure

06. Conclusion

Returning Errors from Activities

- Application Failures are used to communicate application-specific failures in Workflows and Activities

Returning Errors from Activities

- Application Failures are used to communicate application-specific failures in Workflows and Activities

- In Activities, returning a `NewApplicationError` will cause the Activity to fail

Returning Errors from Activities

- Application Failures are used to communicate application-specific failures in Workflows and Activities

- In Activities, returning a `NewApplicationError` will cause the Activity to fail

```
if len(address.CardNumber) != 16 {
  return chargestatus, temporal.NewApplicationError("Credit Card Charge Error",
"CreditCardError", nil, nil)
} else {
  return chargestatus, nil
}
```

Returning Errors from Activities

- Application Failures are used to communicate application-specific failures in Workflows and Activities

- In Activities, returning a `NewApplicationError` will cause the Activity to fail

- Will be represented as an `ActivityTaskFailed` Event. This Event will display the error message specified in the `ApplicationFailure`.

Returning Errors from Activities

24 2024-08-14 UTC 18:35:44.69 ActivityTaskFailed

Failure

```
{
  "message": "Credit Card Charge Error",
  "source": "GoSDK",
  "applicationFailureInfo": {
    "type": "CreditCardError",
    "nonRetryable": true,
    "details": {
      "payloads": [
        null
      ]
    }
  }
}
```

Scheduled Event ID 22

Started Event ID 23

Identity 3756@Temporal.local@

Retry State RETRY_STATE_NON_RETRYABLE_FAILURE

Returning Errors from Activities

- Errors returned from Activities are converted to an `ApplicationFailure` and then wrapped in an `ActivityFailure.`
- This wrapper provides context such as:

Returning Errors from Activities

- Errors returned from Activities are converted to an `ApplicationFailure` and then wrapped in an `ActivityFailure.`
- This wrapper provides context such as:
  - Activity Type

Returning Errors from Activities

- Errors returned from Activities are converted to an `ApplicationFailure` and then wrapped in an `ActivityFailure.`
- This wrapper provides context such as:
  - Activity Type
  - Retry attempts

Returning Errors from Activities

- Errors returned from Activities are converted to an `ApplicationFailure` and then wrapped in an `ActivityFailure.`
- This wrapper provides context such as:
  - Activity Type
  - Retry attempts
  - Original cause

Non-Retryable Errors for Activities

- Permanent failures will not retry themselves

Non-Retryable Errors for Activities

- Permanent failures will not retry themselves
- Better to surface permanent failures instead of retry them.

Non-Retryable Errors for Activities

- Permanent failures will not retry themselves
- Better to surface permanent failures instead of retry them.

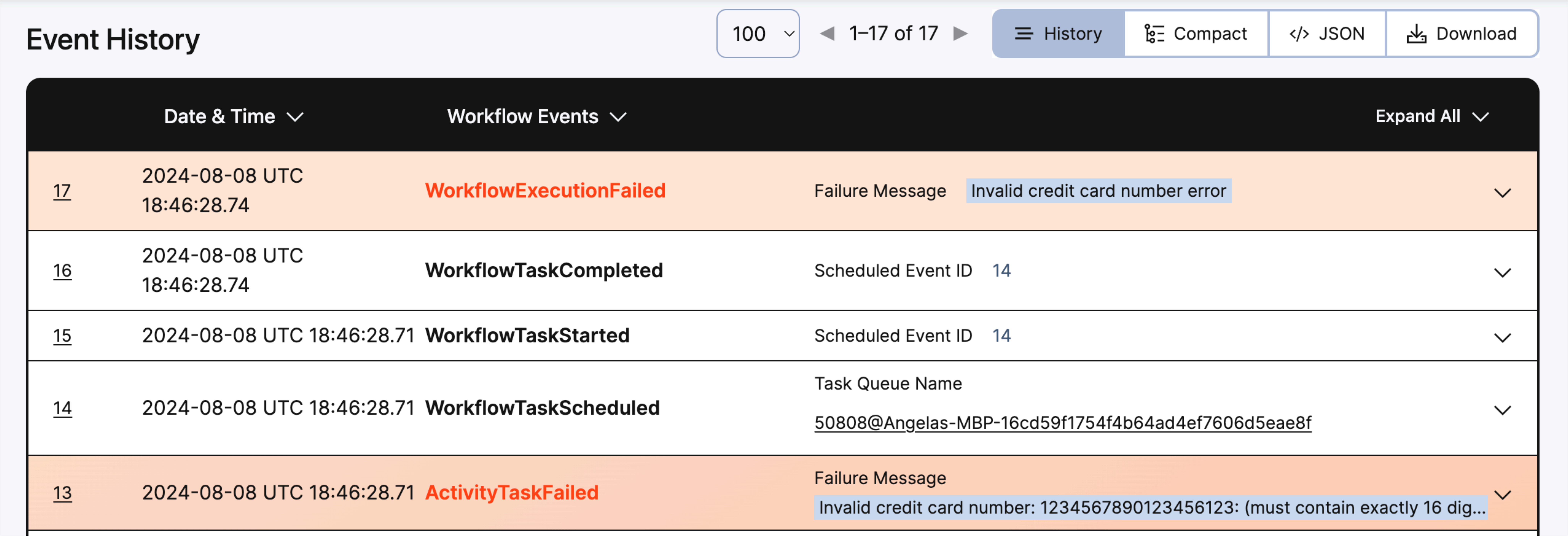
```go
if len(address.CardNumber) != 16 {
  return chargestatus, temporal.NewNonRetryableApplicationError("Credit Card Charge Error",
"CreditCardError", nil, nil)
} else {
  return chargestatus, nil
}
```

Surfacing Activity Failures

- An Activity failure will never directly cause a Workflow Execution failure.

Surfacing Activity Failures

- An Activity failure will never directly cause a Workflow Execution failure.



Event History

	Date & Time	Workflow Events		Expand All
17	2024-08-08 UTC 18:46:28.74	WorkflowExecutionFailed	Failure Message Invalid credit card number error	
16	2024-08-08 UTC 18:46:28.74	WorkflowTaskCompleted	Scheduled Event ID 14	
15	2024-08-08 UTC 18:46:28.71	WorkflowTaskStarted	Scheduled Event ID 14	
14	2024-08-08 UTC 18:46:28.71	WorkflowTaskScheduled	Task Queue Name 50808@Angelas-MBP-16cd59f1754f4b64ad4ef7606d5eae8f	
13	2024-08-08 UTC 18:46:28.71	ActivityTaskFailed	Failure Message Invalid credit card number: 1234567890123456123: (must contain exactly 16 dig...	

Returning Errors from Workflows

- In Go, only a `panic` will lead to a Workflow Task Failure, after which the Workflow Task will fail and be retried.

Returning Errors from Workflows

- In Go, only a `panic` will lead to a Workflow Task Failure, after which the Workflow Task will fail and be retried.

- Any error returned from the Workflow will cause the entire Workflow Execution to fail. **This behavior is unique to Go. Other SDKs will only fail the Workflow Execution on a Temporal Failure.**

Returning Errors from Workflows

- In Go, only a `panic` will lead to a Workflow Task Failure, after which the Workflow Task will fail and be retried.
- Any error returned from the Workflow will cause the entire Workflow Execution to fail. **This behavior is unique to Go. Other SDKs will only fail the Workflow Execution on a Temporal Failure.**
- Most types of Temporal Failures are triggered without being returned manually

Returning Errors from Workflows

- In Go, only a `panic` will lead to a Workflow Task Failure, after which the Workflow Task will fail and be retried.

- Any error returned from the Workflow will cause the entire Workflow Execution to fail. **This behavior is unique to Go. Other SDKs will only fail the Workflow Execution on a Temporal Failure.**

- Most types of Temporal Failures are triggered without being returned manually

- You can also explicitly fail the Workflow Execution by returning an `ApplicationFailure`

Workflow Execution Failure Summary

- An `ApplicationFailure` can be returned from a Workflow to fail the Workflow Execution

Workflow Execution Failure Summary

- An `ApplicationFailure` can be returned from a Workflow to fail the Workflow Execution

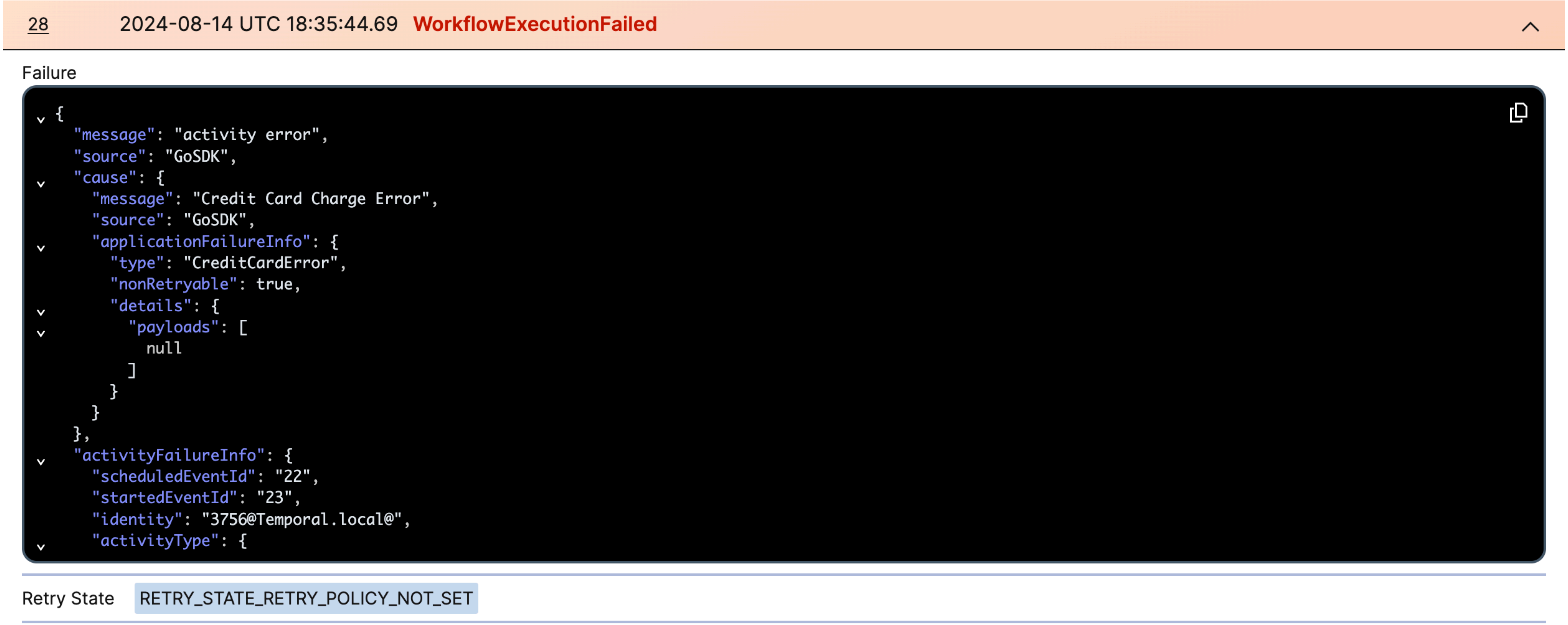
- Workflow Execution Failures put the Workflow Execution into the “Failed” state

Workflow Execution Failure Summary

- An `ApplicationFailure` can be returned from a Workflow to fail the Workflow Execution
- Workflow Execution Failures put the Workflow Execution into the “Failed” state

```
err = workflow.ExecuteActivity(ctx, ProcessCreditCard, order.Address).Get(ctx, &chargestatus)
if err != nil {
  var applicationErr *temporal.ApplicationError
  if errors.As(err, &applicationErr) {
    logger.Error("Unable to charge credit card", "Error", err)
  }
  return OrderConfirmation{}, err
}
```

Workflow Execution Failure Summary



28 2024-08-14 UTC 18:35:44.69 WorkflowExecutionFailed

Failure

```
{
  "message": "activity error",
  "source": "GoSDK",
  "cause": {
    "message": "Credit Card Charge Error",
    "source": "GoSDK",
    "applicationFailureInfo": {
      "type": "CreditCardError",
      "nonRetryable": true,
      "details": {
        "payloads": [
          null
        ]
      }
    }
  },
  "activityFailureInfo": {
    "scheduledEventId": "22",
    "startedEventId": "23",
    "identity": "3756@Temporal.local@",
    "activityType": {
```

Retry State RETRY_STATE_RETRY_POLICY_NOT_SET

Handling Errors

- Examples of `TemporalFailure` that you may see from your Workflow Code (and be able to catch) would include `ApplicationFailure, ActivityFailure, ChildWorkflowFailure.`

Handling Errors

- Examples of `TemporalFailure` that you may see from your Workflow Code (and be able to catch) would include `ApplicationFailure, ActivityFailure, ChildWorkflowFailure.`

- Allowing these to bubble up without handling appropriately will result in the Workflow Execution entering a 'Failed' state.

Exercise #1: Handling Errors

- **During this exercise, you will**
  - Return and handle errors in Temporal Workflows and Activities
  - Use non-retry able errors to fail an Activity
  - Locate the details of a failure in Temporal Workflows and Activities in the Event History
- **Refer to the README.md file in the exercise environment for details**
  - The code is below the **`exercises/handling-errors`**
    - Make your changes to the code in the **`practice`** subdirectory (look for TODO comments)
    - If you need a hint or want to verify your changes, look at the complete version in the **`solution`** subdirectory

Returning and Handling Errors Summary

- Returning an *ApplicationFailure* will cause the Activity to fail.

Returning and Handling Errors Summary

- Returning an *ApplicationFailure* will cause the Activity to fail.
- Errors returned from the Workflow will cause the entire Workflow Execution to fail.

Returning and Handling Errors Summary

- Returning an *ApplicationFailure* will cause the Activity to fail.
- Errors returned from the Workflow will cause the entire Workflow Execution to fail.
- You can return Non-Retryable Activities if you do not want an Activity to be retried.

Crafting an Error Handling Strategy

00. About this Workshop

01. Error Handling Concepts

02. Returning and Handling Errors

▶ **03. Timeouts**

04. Retry Policies

05. Recovering from Failure

06. Conclusion

What are Timeouts?

- A predefined duration provided for an operation to complete
- Temporal uses timeouts for two primary reasons:
  - Detect failure
  - Establish a maximum time duration for your business logic

Activity Timeouts

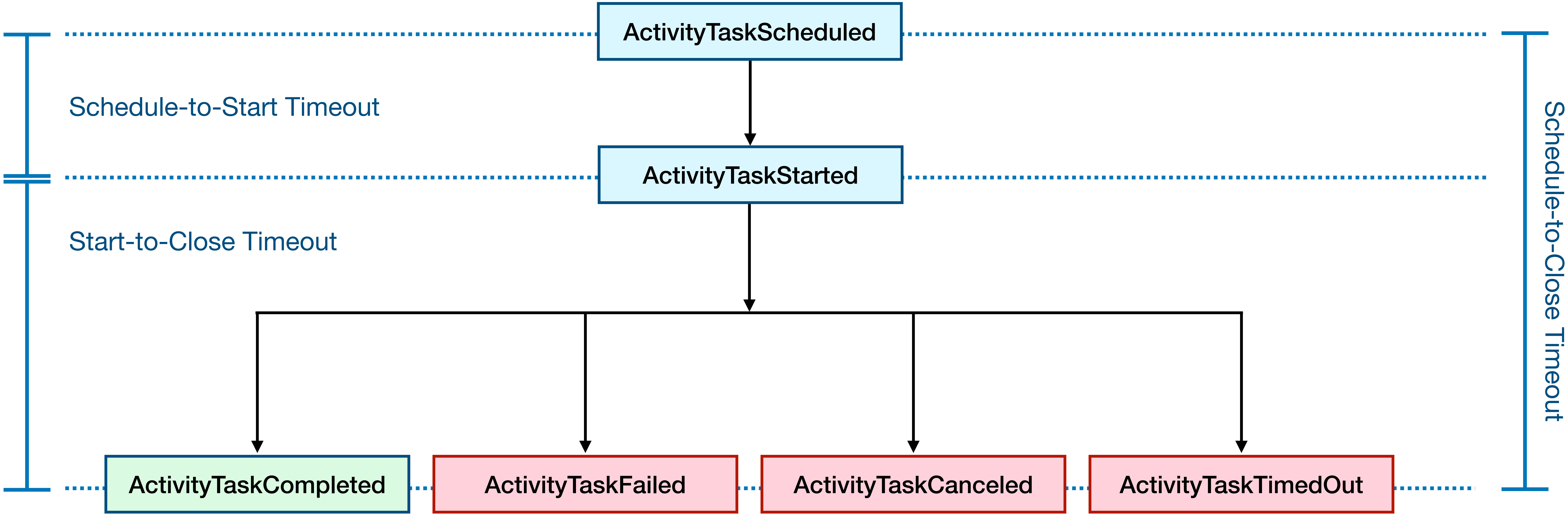
- Controls the maximum duration of a different aspect of an Activity Execution
- A measure of the time it takes to transition between one state to another
- Specified as an argument on the call to `proxyActivities`
- As with an Activity that fails, an Activity that times out will be retried
  - Based on details specified in the Retry Policy

Review of Activity Task States

Order	Event Type	Event Description
1	ActivityTaskScheduled	Temporal Service adds the Activity Task to the Task Queue
2	ActivityTaskStarted	Worker accepts the Activity Task; it’s removed from the Task Queue)
3	ActivityTaskCompleted	Worker reports result of Activity Execution to the Temporal Service

(One of many closed states)

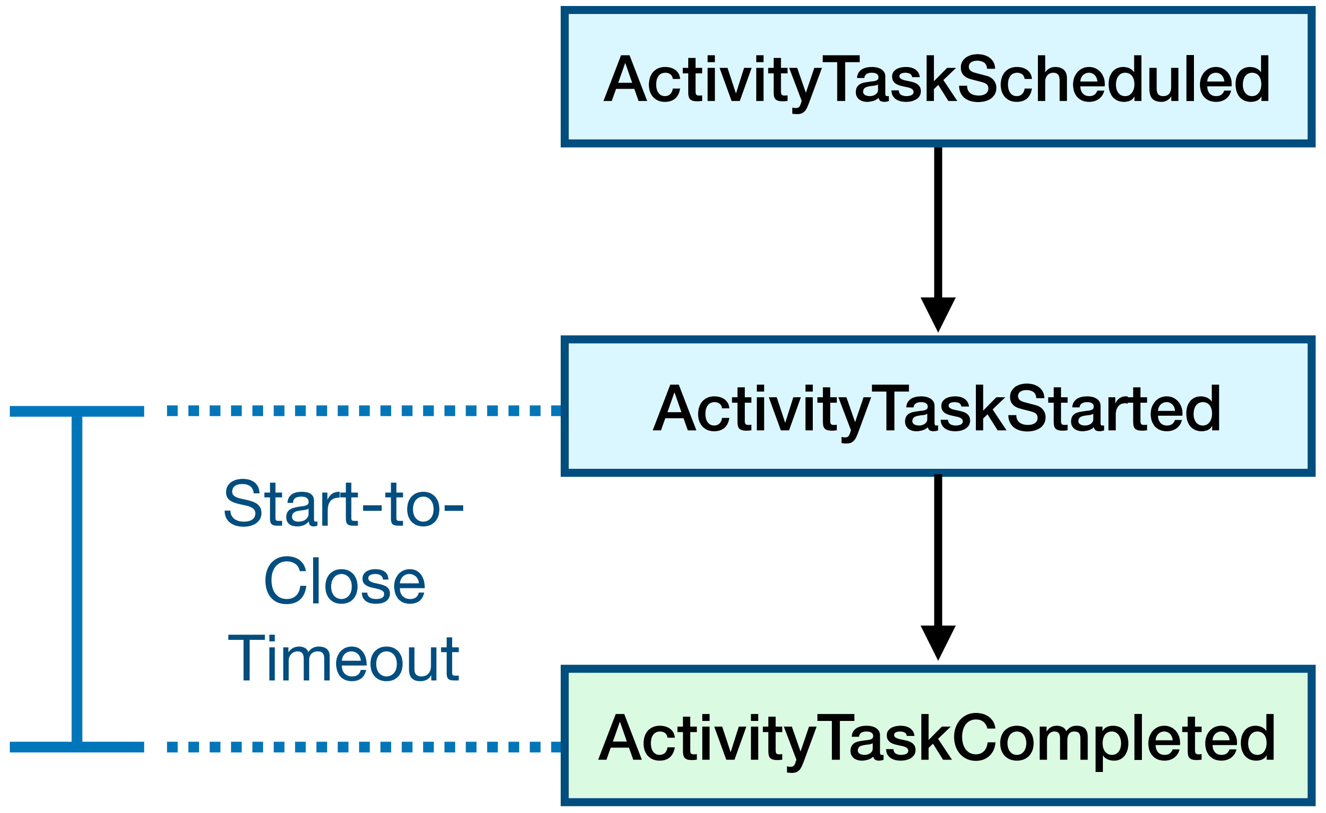
Understanding Activity Timeout Names



Start-to-Close Timeout

- **Limits maximum time allowed for a single Activity *Task* Execution**
  - Time is reset for each retry attempt, since that will take place in a new Activity Task
  - Recommended: Set duration slightly longer than *maximum* time you expect the Activity will take

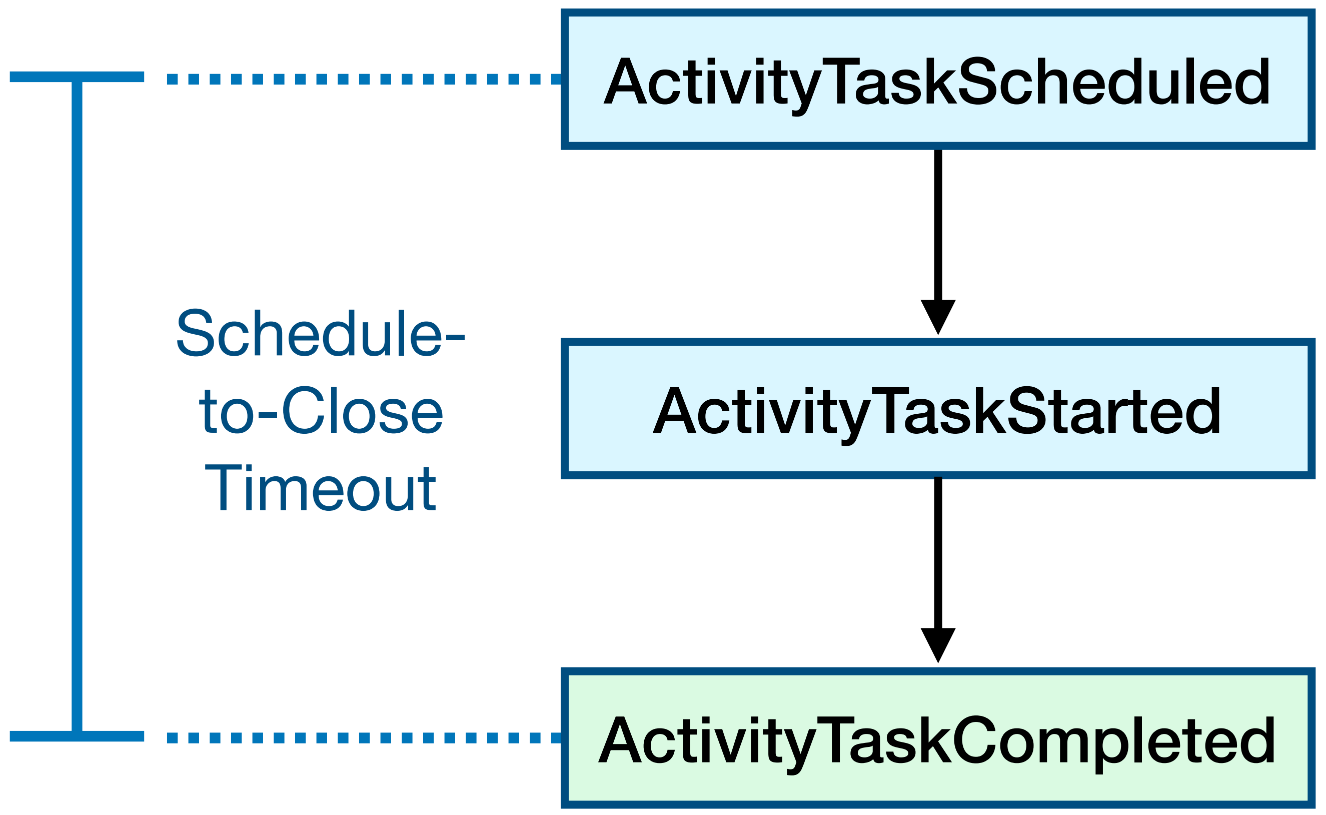
```
activityoptions := workflow.ActivityOptions{
  StartToCloseTimeout: 10 * time.Second,
}
ctx = workflow.WithActivityOptions(ctx, activityoptions)
var yourActivityResult YourActivityResult
err = workflow.ExecuteActivity(ctx, YourActivityDefinition,
yourActivityParam).Get(ctx, &yourActivityResult)
if err != nil {
  // ...
}
```



Schedule-to-Close Timeout

- **Limits maximum time allowed for entire Activity Execution**
  - Because it includes all retries, it is typically less predictable than a Start-to-Close Timeout

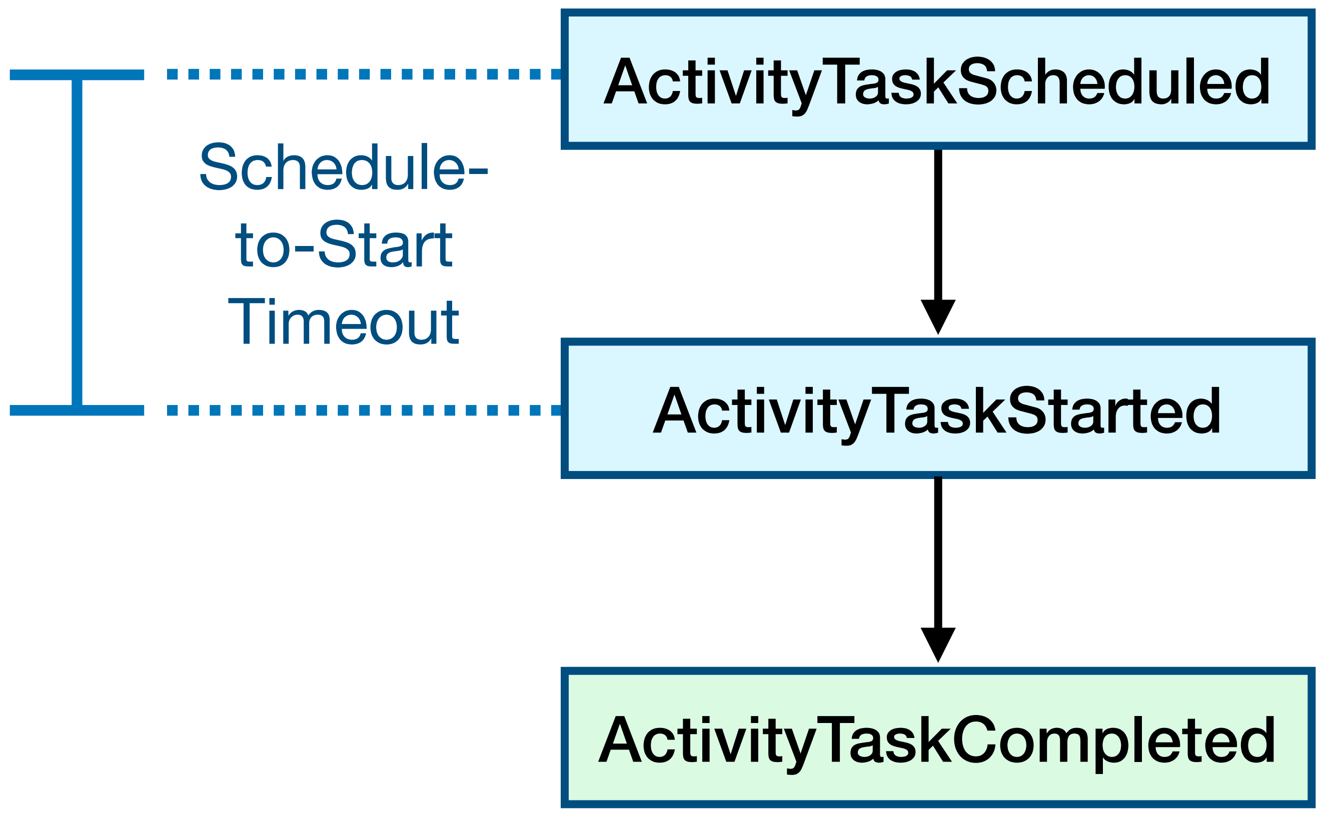
```
activityoptions := workflow.ActivityOptions{
  ScheduleToCloseTimeout: 10 * time.Second,
}
ctx = workflow.WithActivityOptions(ctx, activityoptions)
var yourActivityResult YourActivityResult
err = workflow.ExecuteActivity(ctx, YourActivityDefinition,
yourActivityParam).Get(ctx, &yourActivityResult)
if err != nil {
  // ...
}
```



Schedule-to-Start Timeout

- **Limits maximum time allowed for Activity Task to remain in Task Queue**
  - Ensures the Activity is started within a specified time frame, though it’s seldom recommended
  - If set, it is done *in addition to* a Start-to-Close or Schedule-to-Close Timeout

```
activityoptions := workflow.ActivityOptions{
  ScheduleToStartTimeout: 10 * time.Second,
}
ctx = workflow.WithActivityOptions(ctx, activityoptions)
var yourActivityResult YourActivityResult
err = workflow.ExecuteActivity(ctx, YourActivityDefinition,
yourActivityParam).Get(ctx, &yourActivityResult)
if err != nil {
  // ...
}
```



Schedule-to-Start Timeout

ActivityTaskScheduled

ActivityTaskStarted

ActivityTaskCompleted

Activity Timeout Best Practices

- **You are required to set a Schedule-to-Close or Start-to-Close Timeout**
  - It can be difficult to predict how long execution might take when retries are involved
  - Therefore, setting Start-to-Close is usually the better choice
- **Retry Policies allow you to specify a maximum number of retry attempts**
  - However, using Timeouts to limit the duration is typically more useful
  - Business logic tends to be concerned with how long something takes (for example, SLAs)

Workflow Timeouts

- Control the maximum duration of a different aspect of a Workflow Execution

- We generally do not recommend setting Workflow Timeouts

Workflow Execution Timeout

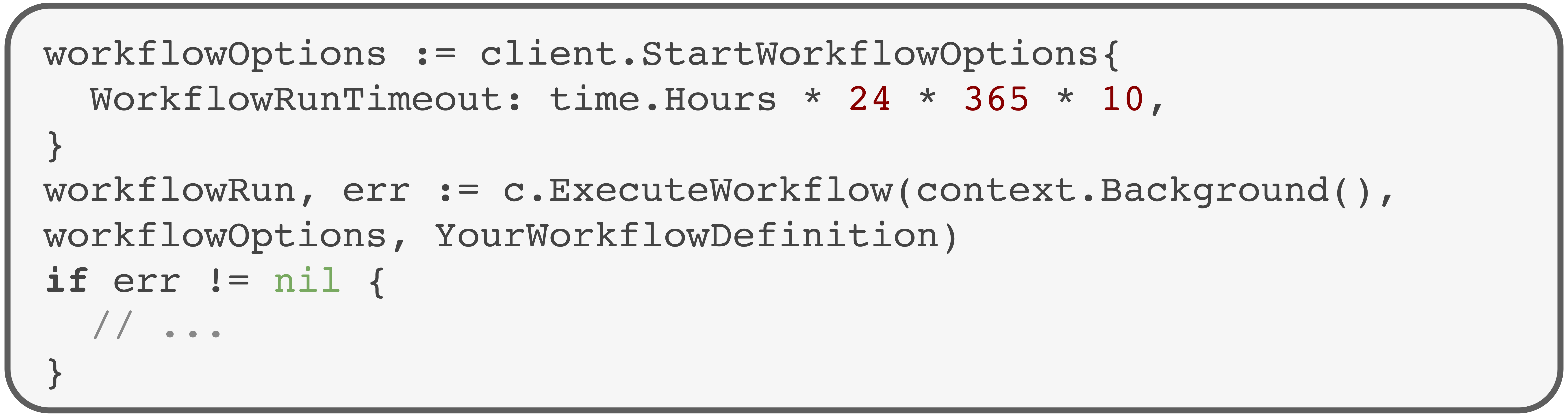
- Restricts the maximum amount of time that a single Workflow Execution can be executed, including retries and any usage of Continue-As-New

- Default is infinite

```
workflowOptions := client.StartWorkflowOptions{
  WorkflowExecutionTimeout: time.Hours * 24 * 365 * 10,
}
workflowRun, err := c.ExecuteWorkflow(context.Background(),
workflowOptions, YourWorkflowDefinition)
if err != nil {
  // ...
}
```

Workflow Run Timeout

- A Workflow Run is the instance of a specific Workflow Execution
- Restricts the maximum duration of a single Workflow Run
- This does not include retries or Continue-As-New
- Default is infinite



```go
workflowOptions := client.StartWorkflowOptions{
  WorkflowRunTimeout: time.Hours * 24 * 365 * 10,
}
workflowRun, err := c.ExecuteWorkflow(context.Background(),
workflowOptions, YourWorkflowDefinition)
if err != nil {
  // ...
}
```

Workflow Task Timeout

- Restricts the maximum amount of time that a Worker can execute a Workflow Task, beginning from when the Worker has accepted that Workflow Task through its completion

- Default value of is ten seconds

```
workflowOptions := client.StartWorkflowOptions{
  WorkflowTaskTimeout: time.Hours * 24 * 365 * 10,
}
workflowRun, err := c.ExecuteWorkflow(context.Background(), workflowOptions,
YourWorkflowDefinition)
if err != nil {
  // ...
}
```

Best Practices

- We generally do not recommend setting Workflow Timeouts
- If you need to perform an action inside your Workflow after a specific period time, we recommend using a Timer

Activity Heartbeats

- A periodic message sent by the Activity to the Temporal Service that serves multiple purposes:

Activity Heartbeats

- A periodic message sent by the Activity to the Temporal Service that serves multiple purposes:
  - Progress indication

Activity Heartbeats

- A periodic message sent by the Activity to the Temporal Service that serves multiple purposes:
  - Progress indication
  - Worker Health Check

Activity Heartbeats

- A periodic message sent by the Activity to the Temporal Service that serves multiple purposes:
  - Progress indication
  - Worker Health Check
  - Cancellation Detection

How to Send a Heartbeat Message

```
func YourActivityDefinition(ctx, YourActivityDefinitionParam)
(YourActivityDefinitionResult, error) {
    // ...
    activity.RecordHeartbeat(ctx, details)
    // ...
}
```

Heartbeats and Cancellations

- For an Activity to be cancellable, it must perform Heartbeating.

Heartbeats and Cancellations

- For an Activity to be cancellable, it must perform Heartbeating.

- If you need to cancel a long-running Activity Execution, make sure it is configured to send Heartbeats periodically.

Heartbeat Timeout

- The maximum time allowed between Activity Heartbeats

Heartbeat Timeout

- The maximum time allowed between Activity Heartbeats
- `HeartbeatTimeout` must be set in order for Temporal to track the Heartbeats sent by the Activity

Heartbeat Timeout

- The maximum time allowed between Activity Heartbeats
- `HeartbeatTimeout` must be set in order for Temporal to track the Heartbeats sent by the Activity

```
activityoptions := workflow.ActivityOptions{
  HeartbeatTimeout: 10 * time.Second,
}
```

Heartbeat Timeout

- To ensure efficient, handling of long-running Activities:
  - Set your `StartToClose` Timeout to be slightly longer than the maximum duration of your Activity

Heartbeat Timeout

- To ensure efficient, handling of long-running Activities:
  - Set your `StartToClose` Timeout to be slightly longer than the maximum duration of your Activity
  - Your `HeartbeatTimeout` should be fairly short

Heartbeat Timeout

- To ensure efficient, handling of long-running Activities:
    - Set your `StartToClose` Timeout to be slightly longer than the maximum duration of your Activity
    - Your `HeartbeatTimeout` should be fairly short
- When the `HeartbeatTimeout` is specified, the Activity must send Heartbeats at intervals shorter than the `HeartbeatTimeout.`

Heartbeat Throttling

- Heartbeats may be throttled by the Worker

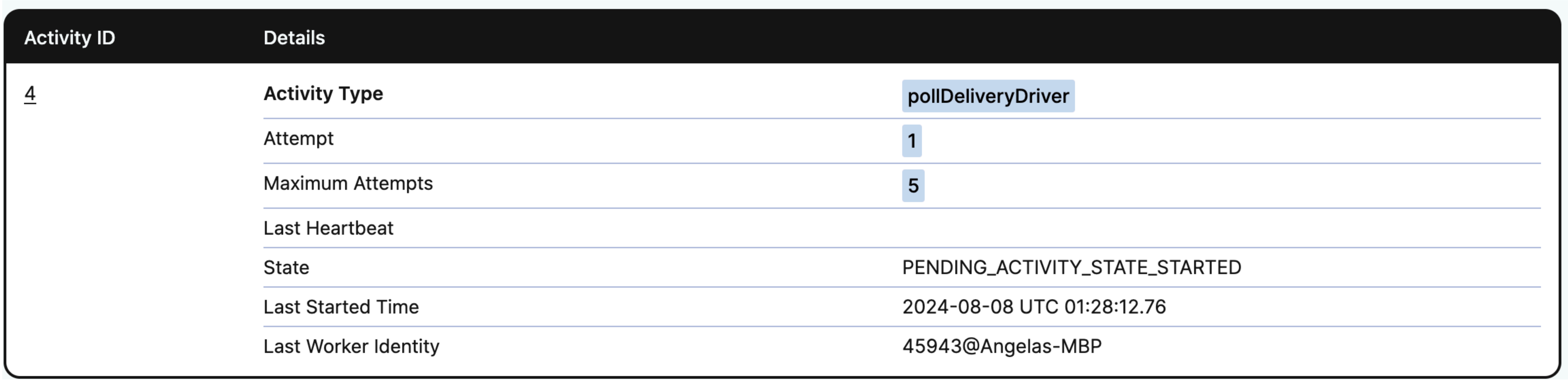
Heartbeat Throttling

- Heartbeats may be throttled by the Worker
- Throttling allows the Worker to reduce network traffic and load on the Temporal Service

Heartbeat Throttling

- Heartbeats may be throttled by the Worker
- Throttling allows the Worker to reduce network traffic and load on the Temporal Service
- Throttling does not apply to the final Heartbeat message in the case of Activity Failure.

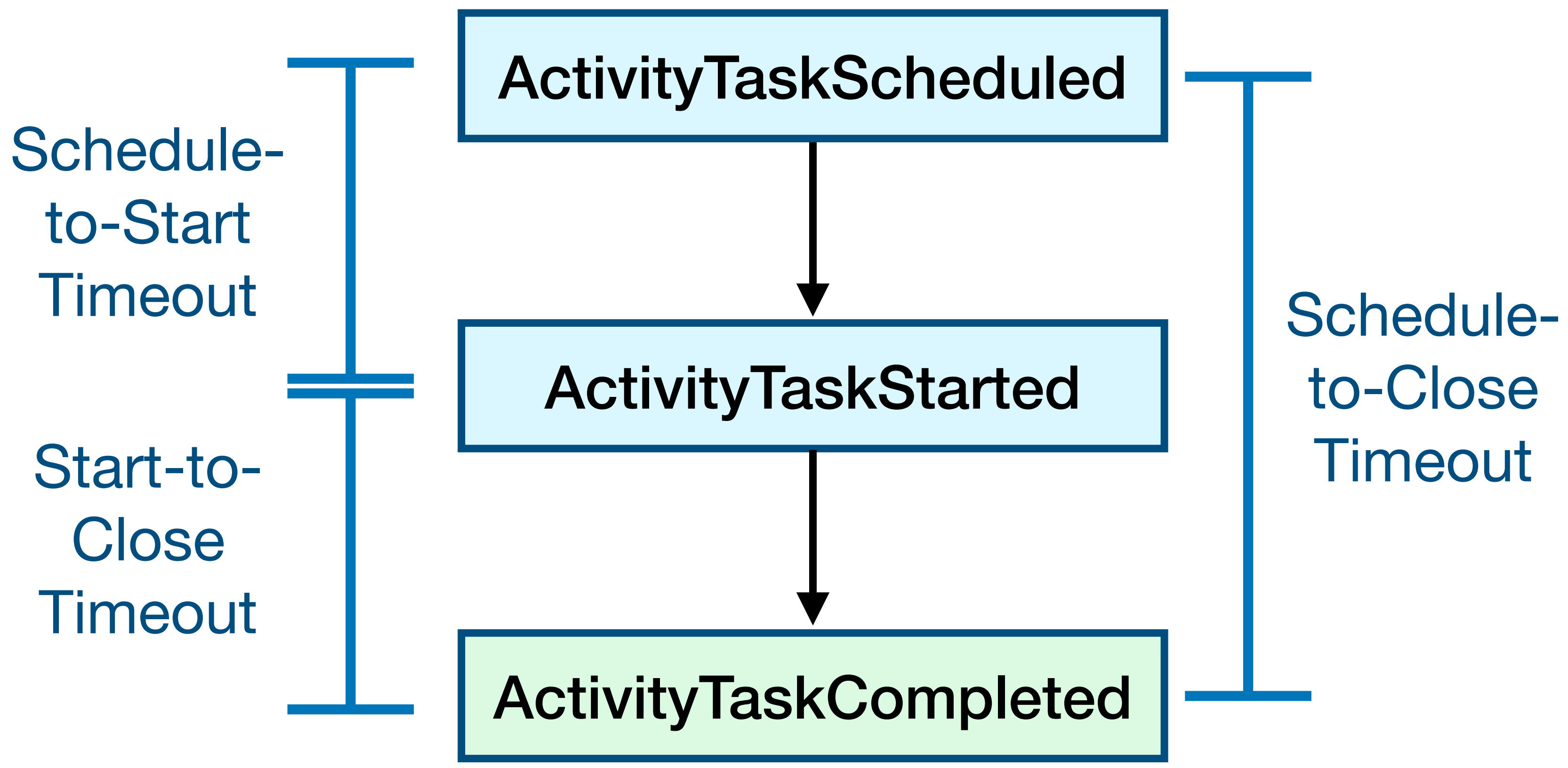
Heartbeat Throttling



Activity ID	Details	
4	**Activity Type**	pollDeliveryDriver
	Attempt	1
	Maximum Attempts	5
	Last Heartbeat	
	State	PENDING_ACTIVITY_STATE_STARTED
	Last Started Time	2024-08-08 UTC 01:28:12.76
	Last Worker Identity	45943@Angelas-MBP

Timeouts Summary

- **Timeouts define the expected duration for an operation to complete**
  - They allow your application to remain responsive and enable Temporal to detect failure
  - You can set different Timeouts for each Activity Execution in a Workflow
- **You are required to set a Schedule-to-Close or Start-to-Close Timeout**
  - We recommend setting Start-to-Close Timeout in most cases
  - We do not recommend setting a Workflow Timeout
- **Activity Heartbeats improve failure detection**
  - Recommended for long-running Activities



Timeouts Summary

- Activity Timeouts can be set globally in your Activity Retry Policy or in each individual Activity invocation.

Timeouts Summary

- Activity Timeouts can be set globally in your Activity Retry Policy or in each individual Activity invocation.

- Configuring a *Schedule-To-Close* or *Start-To-Close* timeout in your Activity options is mandatory.

Timeouts Summary

- **Activity Timeouts can be set globally in your Activity Retry Policy or in each individual Activity invocation.**
- **Configuring a *Schedule-To-Close* or *Start-To-Close* timeout in your Activity options is mandatory.**
- **Unlike Activity Timeouts, we generally do not recommend setting Workflow Timeouts.**

Timeouts Summary

- Activity Timeouts can be set globally in your Activity Retry Policy or in each individual Activity invocation.
- Configuring a *Schedule-To-Close* or *Start-To-Close* timeout in your Activity options is mandatory.
- Unlike Activity Timeouts, we generally do not recommend setting Workflow Timeouts.
- Activity heartbeats are used to indicate progress and check Worker health
- They also enable the Worker to check if the Activity Execution has been canceled

Timeouts Summary

- Activity Timeouts can be set globally in your Activity Retry Policy or in each individual Activity invocation.
- Configuring a *Schedule-To-Close* or *Start-To-Close* timeout in your Activity options is mandatory.
- Unlike Activity Timeouts, we generally do not recommend setting Workflow Timeouts.
- Activity heartbeats are used to indicate progress and check Worker health
- They also enable the Worker to check if the Activity Execution has been canceled
- A *Heartbeat Timeout* must be set in order for Temporal to track the Heartbeats sent by the Activity

Crafting an Error Handling Strategy

00. About this Workshop

01. Error Handling Concepts

02. Returning and Handling Errors

03. Timeouts

▶ **04. Retry Policies**

05. Recovering from Failure

06. Conclusion

What is a Retry Policy?

- Temporal's default behavior is to automatically retry an Activity that fails

What is a Retry Policy?

- Temporal’s default behavior is to automatically retry an Activity that fails

- A collection of attributes that instructs the Temporal Service how to retry a failure of a Workflow Execution or an Activity Task Execution

What is a Retry Policy?

- Temporal’s default behavior is to automatically retry an Activity that fails
- A collection of attributes that instructs the Temporal Service how to retry a failure of a Workflow Execution or an Activity Task Execution
- In contrast to the Activities it contains, the Workflow Execution itself is not associated with a Retry Policy by default.

What is a Retry Policy?

- Temporal’s default behavior is to automatically retry an Activity that fails
- A collection of attributes that instructs the Temporal Service how to retry a failure of a Workflow Execution or an Activity Task Execution
- In contrast to the Activities it contains, the Workflow Execution itself is not associated with a Retry Policy by default.
- The retry policies do not apply to the Workflow Task Executions, which always retry indefinitely.

Default Retry Policies

- Activities in Temporal are associated with a Retry Policy by default, Workflows are not.

Retry Policy for Activities

- Default is to retry, with a short delay between each attempt

Retry Policy for Activities

- Customize Retry Policy by creating a RetryPolicy{} object

Method	Specifies	Default Value
`InitialInterval`	Duration before the first retry	1 second
`BackoffCoefficient`	Multiplier used for subsequent retries	`2.0`
`MaximumInterval`	Maximum duration between retries, in seconds	`100 * InitialInterval`
`MaximumAttempts`	Maximum number of retry attempts before giving up	`0` (unlimited)
`NonRetryableErrorTypes`	List of application failure types that won't be retried	`[ ]` (empty array)

```
retrypolicy := &temporal.RetryPolicy{ MaximumAttempts: 3 }

options := workflow.ActivityOptions{ RetryPolicy: retrypolicy }
```

Retry Policy for Workflow Executions

- Workflow Executions do not retry by default

Retry Policy for Workflow Executions

- Workflow Executions do not retry by default
- Retry policies should be used with Workflow Executions only in certain situations. For example:
  - A Temporal Cron Job

Retry Policy for Workflow Executions

- Workflow Executions do not retry by default
- Retry policies should be used with Workflow Executions only in certain situations. For example:
    - A Temporal Cron Job
    - Child Workflows to group a subset of Activities

Retry Policy for Workflow Executions

- Workflow Executions do not retry by default
- Retry policies should be used with Workflow Executions only in certain situations. For example:
    - A Temporal Cron Job
    - Child Workflows to group a subset of Activities
- We do not recommend associating a Retry Policy with your Workflow Execution

Custom Retry Policy for Activity Execution

- Transient failure: Resolved by retrying the operation immediately after the failure

- Intermittent failure: Addressed by retrying the operation, but these retries should be spread out over a longer period of time to allow underlying cause to be resolved

- Permanent failure: Cannot be resolved solely through retries, needs manual intervention

Custom Retry Policy for Activity Execution

```
retrypolicy := &temporal.RetryPolicy{
MaximumInterval:        time.Second * 10,
MaximumAttempts:        3,
}

options := workflow.ActivityOptions{
StartToCloseTimeout: time.Second * 5,
HeartbeatTimeout:    10 * time.Second,
RetryPolicy:         retrypolicy,
}

activityRun, err := workflow.ExecuteActivity(ctx, options, ActivityDefinition)
```

Common Use Cases for Defining a Custom Retry Policy

- Making calls to a service experiencing heavy load

Common Use Cases for Defining a Custom Retry Policy

- Making calls to a service experiencing heavy load
- If an external service implements rate limiting

Common Use Cases for Defining a Custom Retry Policy

- Making calls to a service experiencing heavy load
- If an external service implements rate limiting
- A service charges for each call received

Best Practices for Retry Policies

- Don’t unnecessarily set maximum attempts to 1

Best Practices for Retry Policies

- Don’t unnecessarily set maximum attempts to 1
- Recognize that each Activity Execution can have its own retry policy

Best Practices for Retry Policies

- Don’t unnecessarily set maximum attempts to 1
- Recognize that each Activity Execution can have its own retry policy
- Avoid retry policies for Workflow Executions

Customizing a Retry Policy for a Specific Activity

- You can set `ActivityOptions` for each different Activity Execution.

Customizing a Retry Policy for a Specific Activity

- You can set `ActivityOptions` for each different Activity Execution.
- You can also customize a retry policy if an Activity is invoked conditionally

Customizing a Retry Policy for a Specific Activity

```
retrypolicy_lowbackoff := &temporal.RetryPolicy{
  InitialInterval:    time.Second,
  BackoffCoefficient: 2.0,
  MaximumInterval:    time.Second * 100,
}

activityOptions_lowbackoff := workflow.ActivityOptions{
  RetryPolicy: retrypolicy_lowbackoff,
}

retrypolicy_highbackoff := &temporal.RetryPolicy{
  InitialInterval:    time.Second,
  BackoffCoefficient: 20.0,
  MaximumInterval:    time.Second * 100,
}

activityOptions_highbackoff := workflow.ActivityOptions{
  RetryPolicy: retrypolicy_highbackoff,
}

if x == true {
activityRun, err := workflow.ExecuteActivity(ctx, activityOptions_lowbackoff, ActivityDefinition) } else {
activityRun, err := workflow.ExecuteActivity(ctx, activityOptions_highbackoff, ActivityDefinition)
}
```

Defining Errors as Non-Retryable

```
retrypolicy := &temporal.RetryPolicy{
    MaximumInterval:        time.Second * 10,
    MaximumAttempts:        3,
    NonRetryableErrorTypes: []string{"CreditCardError"},
}
```

Defining Errors as Non-Retryable

- Non-retryable errors are specified in the array of non-retry able errors

Defining Errors as Non-Retryable

- Non-retryable errors are specified in the array of non-retry able errors
- By default, this is an empty array

Defining Errors as Non-Retryable

- Non-retryable errors are specified in the array of non-retry able errors
- By default, this is an empty array
- Non-retryable errors should be used when the implementor of the Activity knows that the failure is unrecoverable

Exercise #2: Non-Retryable Error Types

- **During this exercise, you will**
  - Configure non-retry able error types for Activities
  - Implement customized retry policies for Activities
  - Add Heartbeats and Heartbeat timeouts to help users monitor the health of Activities
- **Refer to the README.md file in the exercise environment for details**
  - The code is below the **`exercises/non-retryable-error-types`**
    - Make your changes to the code in the **`practice`** subdirectory (look for TODO comments)
    - If you need a hint or want to verify your changes, look at the complete version in the **`solution`** subdirectory

Retry Policies Summary

- **Temporal’s default behavior is to automatically retry an Activity until it either succeeds or is canceled**

Retry Policies Summary

- Temporal's default behavior is to automatically retry an Activity until it either succeeds or is canceled
- We generally do not recommend associating a Retry Policy with your Workflow Execution

Retry Policies Summary

- Temporal’s default behavior is to automatically retry an Activity until it either succeeds or is canceled
- We generally do not recommend associating a Retry Policy with your Workflow Execution
- You can create as many retry policies as you want for your Activities and customize these retry policies

Crafting an Error Handling Strategy

00. About this Workshop

01. Error Handling Concepts

02. Returning and Handling Errors

03. Timeouts

04. Retry Policies

▶ **05. Recovering from Failure**

06. Conclusion

Handling a Workflow Execution that Cannot Complete

- Canceling your Workflow Execution
- Terminating your Workflow Execution
- Resetting your Workflow Execution

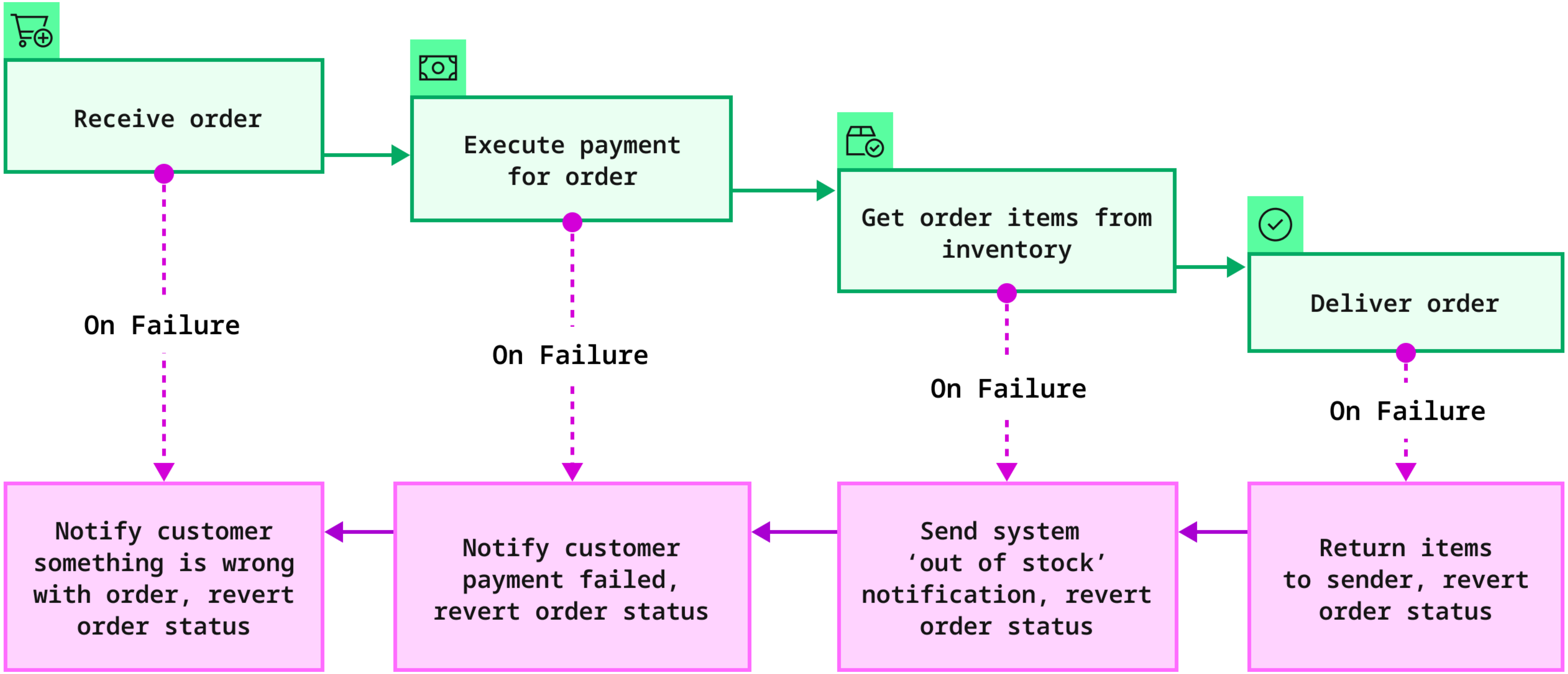
Rollback Actions and the Saga Pattern

- A saga is a pattern used in distributed systems to manage a sequence of local transactions

Rollback Actions and the Saga Pattern

- A saga is a pattern used in distributed systems to manage a sequence of local transactions
- If any transaction in the sequence fails, the saga executions actions to rollback the previous operations. This is known as a compensating action.
- Examples:
  - E-Commerce Transaction
  - Distributed Data Updates

Rollback Actions and the Saga Pattern


Receive order
Execute payment for order
Get order items from inventory
Deliver order
On Failure
On Failure
On Failure
On Failure
Notify customer something is wrong with order, revert order status
Notify customer payment failed, revert order status
Send system ‘out of stock’ notification, revert order status
Return items to sender, revert order status

Rollback Actions and the Saga Pattern

```
err = workflow.ExecuteActivity(ctx, UpdateInventory, order.Items).Get(ctx, nil)
if err != nil {
  return OrderConfirmation{}, err
} else {
...
}
```

Rollback Actions and the Saga Pattern

```
err = workflow.ExecuteActivity(ctx, UpdateInventory, order.Items).Get(ctx, nil)
if err != nil {
  return OrderConfirmation{}, err
}

defer func() {
  if err != nil {
    errCompensation := workflow.ExecuteActivity(ctx, RevertInventory,
order.Items).Get(ctx, nil)
  }
}()
```

Exercise #3: Implementing a Rollback Action with the Saga Pattern

- **During this exercise, you will**
  - Orchestrate Activities using a Saga pattern to implement compensating transactions
  - Handle failures with rollback logic
- **Refer to the README.md file in the exercise environment for details**
  - The code is below the **`exercises/rollback-with-saga`**
    - Make your changes to the code in the **`practice`** subdirectory (look for TODO comments)
    - If you need a hint or want to verify your changes, look at the complete version in the **`solution`** subdirectory

Recovering from Failure Summary

- Canceling Workflow Executions allows them to terminate gracefully

Recovering from Failure Summary

- Canceling Workflow Executions allows them to terminate gracefully
- Terminating your Workflow Execution forcefully stops it without any cleanup

Recovering from Failure Summary

- Canceling Workflow Executions allows them to terminate gracefully
- Terminating your Workflow Execution forcefully stops it without any cleanup
- The saga pattern is used a scenarios where a series of related tasks need to be performed in sequence, each dependent on the success of the previous one

Recovering from Failure Summary

- Canceling Workflow Executions allows them to terminate gracefully
- Terminating your Workflow Execution forcefully stops it without any cleanup
- The saga pattern is used a scenarios where a series of related tasks need to be performed in sequence, each dependent on the success of the previous one.
- In the saga pattern, a compensating action is an action used to rollback previous operations if any transaction in the sequence fails.

Crafting an Error Handling Strategy

00. About this Workshop

01. Error Handling Concepts

02. Returning and Handling Errors

03. Timeouts

04. Retry Policies

05. Recovering from Failure

▶ **06. Conclusion**

Error Handling Concepts Summary (1)

- **You can categorize failures are either platform or application**
  - **Platform**: occur from reasons beyond the control of your application code
  - **Application**: caused by problems with application code or input data
  - Determine which by considering if detecting and fixing requires knowledge of the application
- **You can also classify them according to likelihood of reoccurrence**
  - **Transient**: Not likely to happen again (handle by retrying with a short delay)
  - **Intermittent**: Likely to happen again (handle by retrying with a longer and increasing delay)
  - **Permanent**: Guaranteed to happen again (handling these will require manual intervention)

Error Handling Concepts Summary (2)

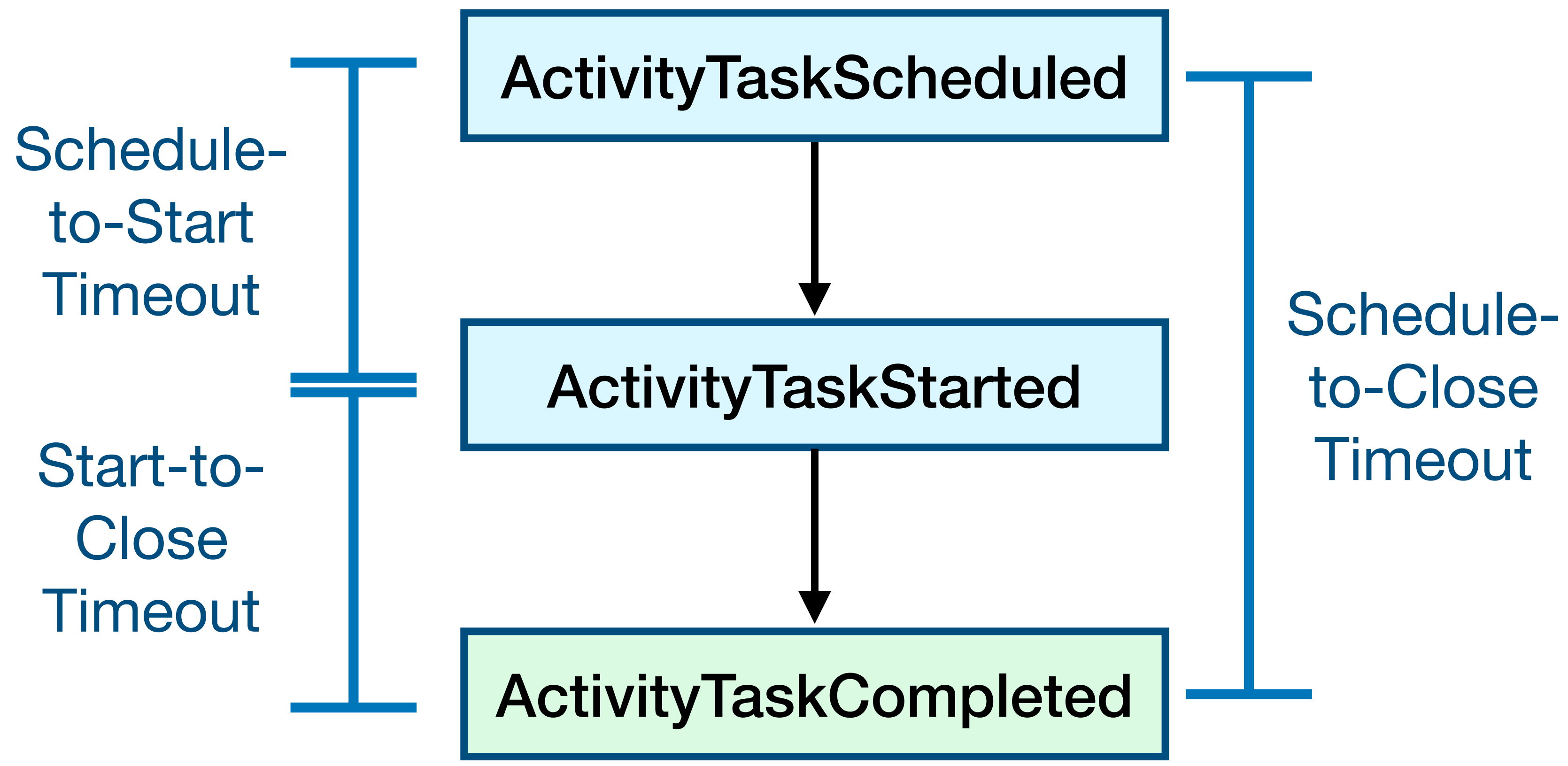
- **Idempotency is a general concern for distributed systems**
    - Will multiple invocations of your operation result in adverse changes to application state?
    - This is a concern for Activities in Temporal, since they may be executed multiple times
    - Temporal strongly recommends that you ensure your Activities are idempotent

Returning and Handling Errors Summary

- **Throwing an `ApplicationFailure` from an Activity causes it to fail**
  - The `ActivityTaskFailed` in Event History includes details of the failure
  - Will retry according to policy, but the developer can force it to be non-retryable if desired
- **What happens when you return an error from a Workflow?**
  - The *Workflow Execution* will fail.

Timeouts Summary

- **Timeouts define the expected duration for an operation to complete**
  - They allow your application to remain responsive and enable Temporal to detect failure
  - You can set different Timeouts for each Activity Execution in a Workflow
- **You are required to set a Schedule-to-Close or Start-to-Close Timeout**
  - We recommend setting Start-to-Close Timeout in most cases
  - We do not recommend setting a Workflow Timeout
- **Activity Heartbeats improve failure detection**
  - Recommended for long-running Activities



Retry Policies Summary (1)

- **Workflow Executions have the benefit of Durable Execution**
  - They must be deterministic, so they rely on Activities to perform failure-prone operations
- **Activities that fail are automatically retried, based on a Retry Policy**
  - Workflow Executions are not retried by default and it’s uncommon to configure that behavior
- **By default, the Activity is re-attempted one second after failure**
  - Delay doubles before each subsequent attempt until reaching maximum of 100 seconds
  - Retries continue until the Activity completes, is canceled, or Workflow Execution ends
  - Provides a reasonable balance for addressing both transient and intermittent failures

Retry Policies Summary (2)

- **This Retry Policy is customizable**
  - You may wish to increase the delay or backoff coefficient for a specific intermittent failure
  - Every Activity Execution in a Workflow can specific a different Retry Policy
- **Use care when specifying maximum attempts in a Retry Policy**
  - Setting this to 1 may have unintended consequences
  - It’s often better to use an Activity Timeout to place a limit on Activity Execution
  - You can also designate a particular type of error as non-retryable

Recovering from Failure Summary

- **Temporal provides a few options for recovering from persistent failure**
  1. Canceling a Workflow Execution is graceful and allows for clean up before closing
  2. Terminating a Workflow Execution is forceful and does not allow cleanup before closing
  3. Resetting a Workflow Execution allows it to continue from a previous point in Event History
- **The application may also support rolling back to a previous state**
  - Often achieved with the Saga pattern
    - Tracks a series of related operations, each dependent on success of the previous one
    - Upon failure, it uses *compensating transactions* to revert changes to application state
  - Java SDK provides built-in Saga support, but it's straightforward to implement in other SDKs

TEMPORAL'S CODE EXCHANGE

Share what you've built with Temporal

Temporal has a thriving community building code for each other – we'd love to see what you've built!

TEMPORAL.IO/CODE-EXCHANGE

Thank You