Crafting an Error Handling Strategy

.NET

20241216

Crafting an Error Handling Strategy

Logistics

- **Introductions**
- **Schedule**
- **Facilities**
- **WiFi**
- **Asking questions and providing feedback**
- **Course conventions: “Activity” vs “activity”**
- **Prerequisites: Temporal 101, 102**

Network: Replay2025
Password: Durable!

We welcome your feedback

`t.mp/replay25ws`

During this course, you will

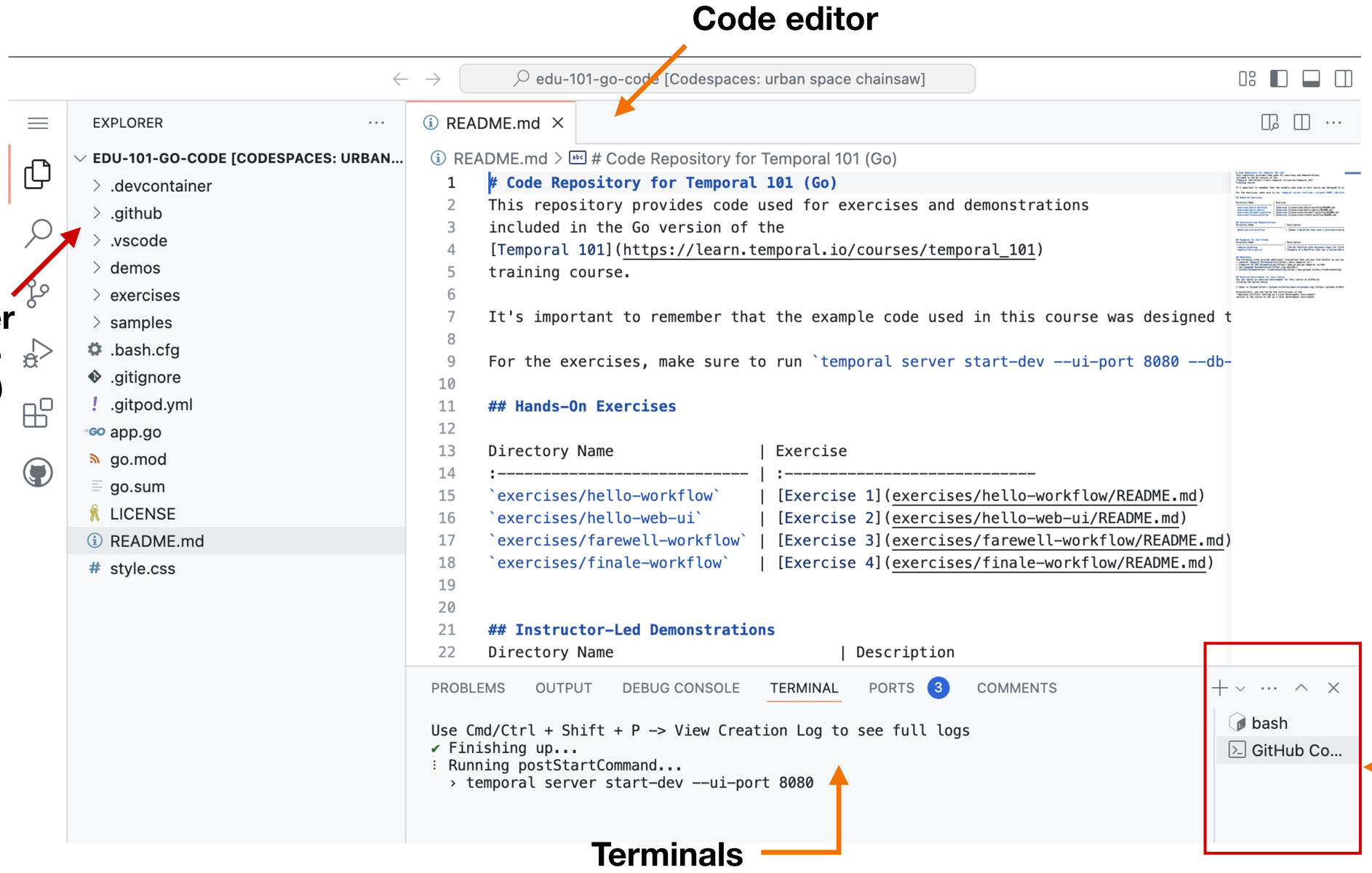
- Recommend an error handling strategy
  - Explain how Temporal represents errors
  - Compare platform errors to application errors
  - Explain differences between timeouts and failures
  - Determine when it is appropriate to fail a Workflow Execution and when to fail an Activity Execution
- Implement 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 workshop**
  - It uses GitHub Codespaces to deploy a Temporal Service, plus a code editor and terminal
  - You access it through your browser (requires you to log in to GitHub)
  - Your instructor will now demonstrate how to access and use it

t.mp/edu-errstrat-dotnet-code

Codespaces Overview



Crafting an Error Handling Strategy

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?**

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

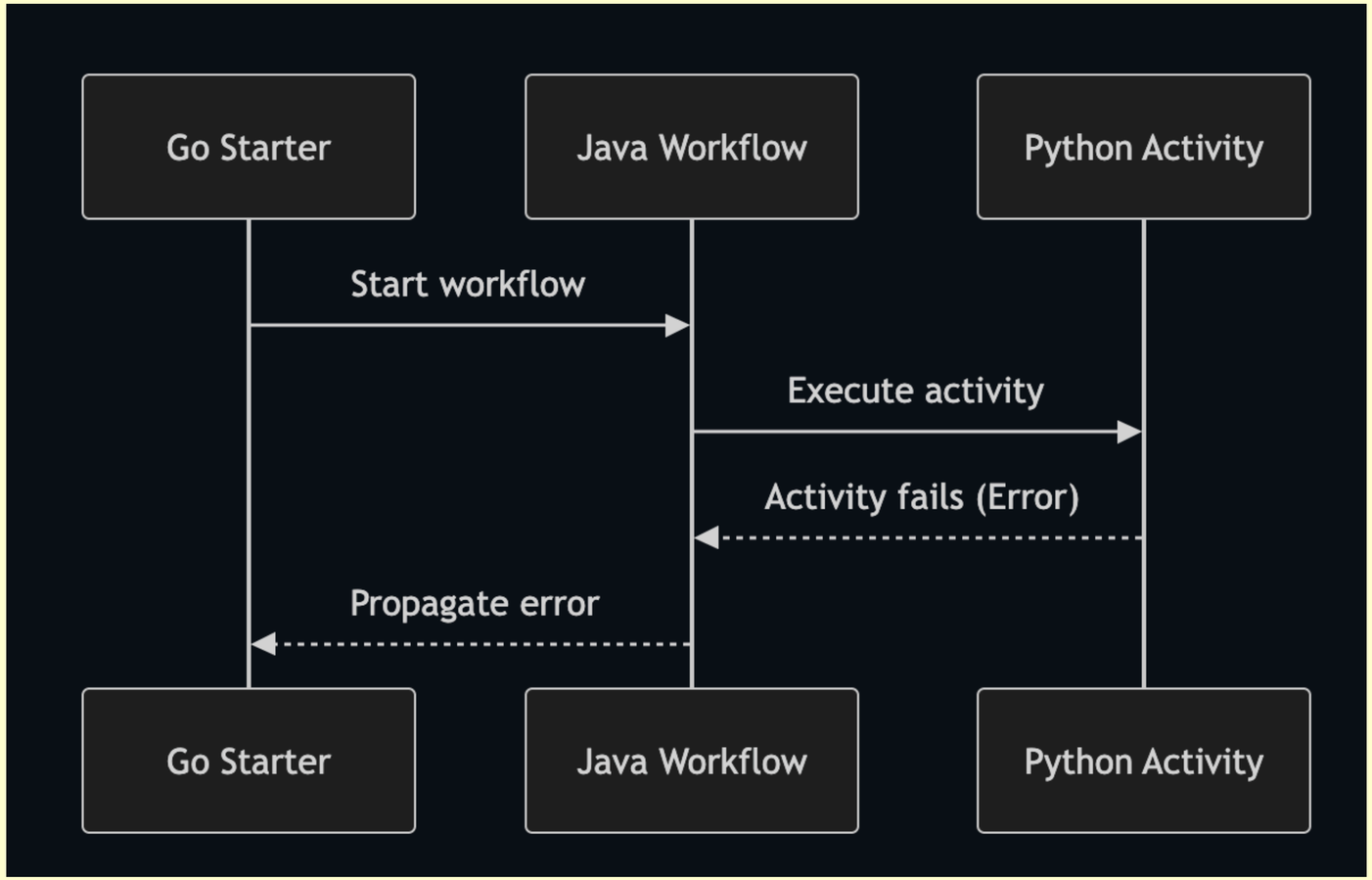
- **Application failures often involve *backward recovery***
  - Backward recovery: Attempt to fix problem reverting previous change(s) in state
  - Example: Compensating transaction
- **Platform failures often involve *forward recovery***
  - Forward recovery: Attempt to fix problem by continuing processing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

The Temporal Error Model

- Temporal preserves errors across language and process boundaries
- Each SDK uses its own native error handling to handle errors within application code - Temporal converts errors to a common format
- Temporal achieves this by using protocol buffers as a message layer
- Temporal uses a custom set of protobufs to define errors, allowing for errors to cross process and language boundaries

Instructor-Led Demo #1

Cross-Language Error Propagation



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

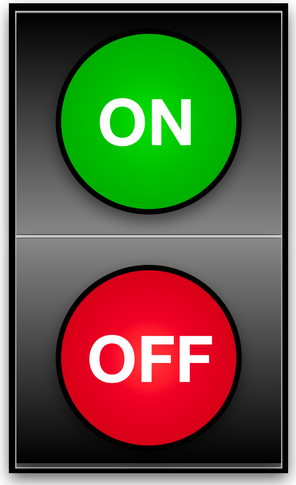
- **Existence of past failure guarantees likelihood of future failures**
- **These are caused by a problem that will *never* resolve itself**
  - For example, sending an e-mail notification fails due to an invalid address
  - Permanent failures require manual repair—you cannot resolve them through retries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

```
var handle = client.GetWorkflowHandle("my-workflow-id");
var workflow_description = await handle.DescribeAsync();
string runId = workflow_description.RunId;
```

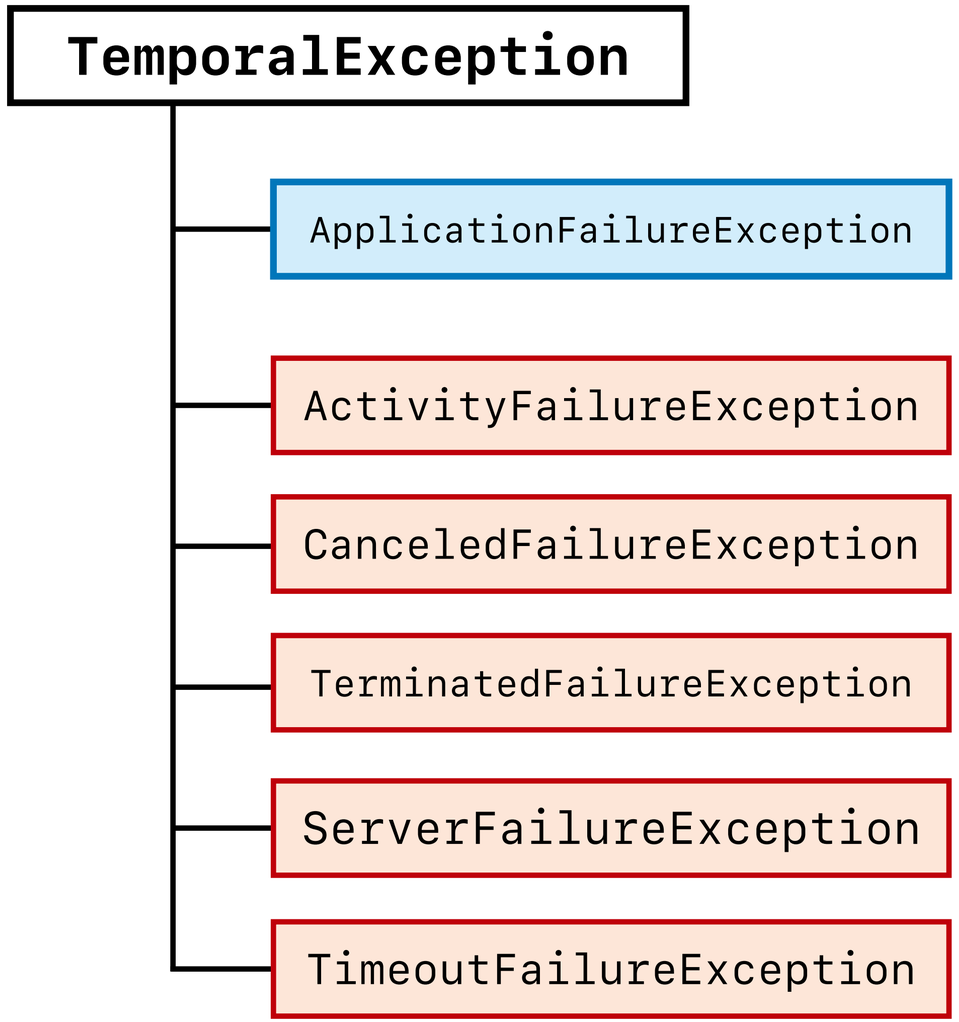
How Temporal Represents Failures (1)

- **All failures in Temporal are represented in the API as a Temporal Failure**
  - **`TemporalException`** is the C# base class that Temporal Failures extend
- **You should not extend the TemporalException class or any of its children**
  - Consistency in error handling
  - Compatibility with the Temporal Service
  - Serialization/deserialization

How Temporal Represents Failures (2)

- **An exception thrown by an Activity is surfaced as an `ActivityFailureException`**
  - You can catch and handle it in your Workflow Definition, if desired
- **You can use custom exception types meaningful to your application**
  - For example, `InvalidCreditCardException` or `UserNotFoundException`

Examples of Temporal Failure Types



Base class, which represents failures that can cross Workflow and Activity boundaries

The only failure that should be thrown by user code.
Used to communicate application-specific failure

Indicates that an Activity failed to complete as expected

Indicates that the operation was canceled

Indicates that the operation was terminated

Indicates a failure originating in the Temporal Service

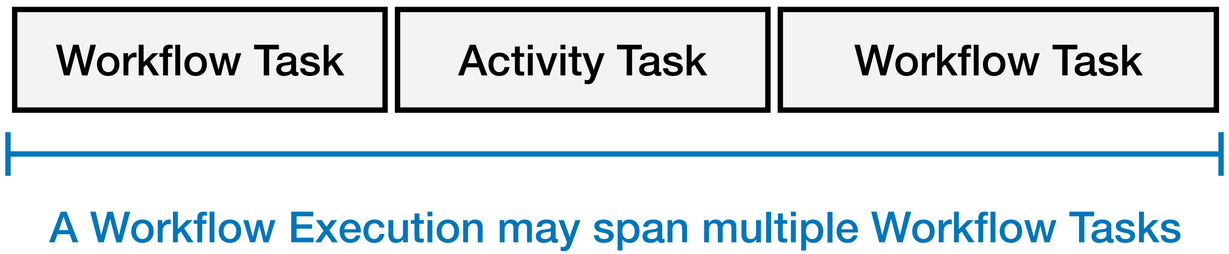
Indicates that the Activity did not complete within its configured Timeout period

Failure Converter

- **Temporal invokes a Failure Converter when an exception is thrown**
- You can encrypt sensitive information and stack trace by using a custom codec.

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



Workflow Task Failures

- **You can throw an exception from your Workflow Definition**
  - What happens will depend on the exception's type
- **If it does not extend `TemporalException`, the Workflow *Task* fails**
  - This may occur due to a bug in your code that's unrelated to Temporal
    - For example, an `ArrayIndexOutOfBoundsException`
  - May also occur for reasons specific to Temporal, such as a non-deterministic error
  - When a Workflow Task fails, it is retried automatically

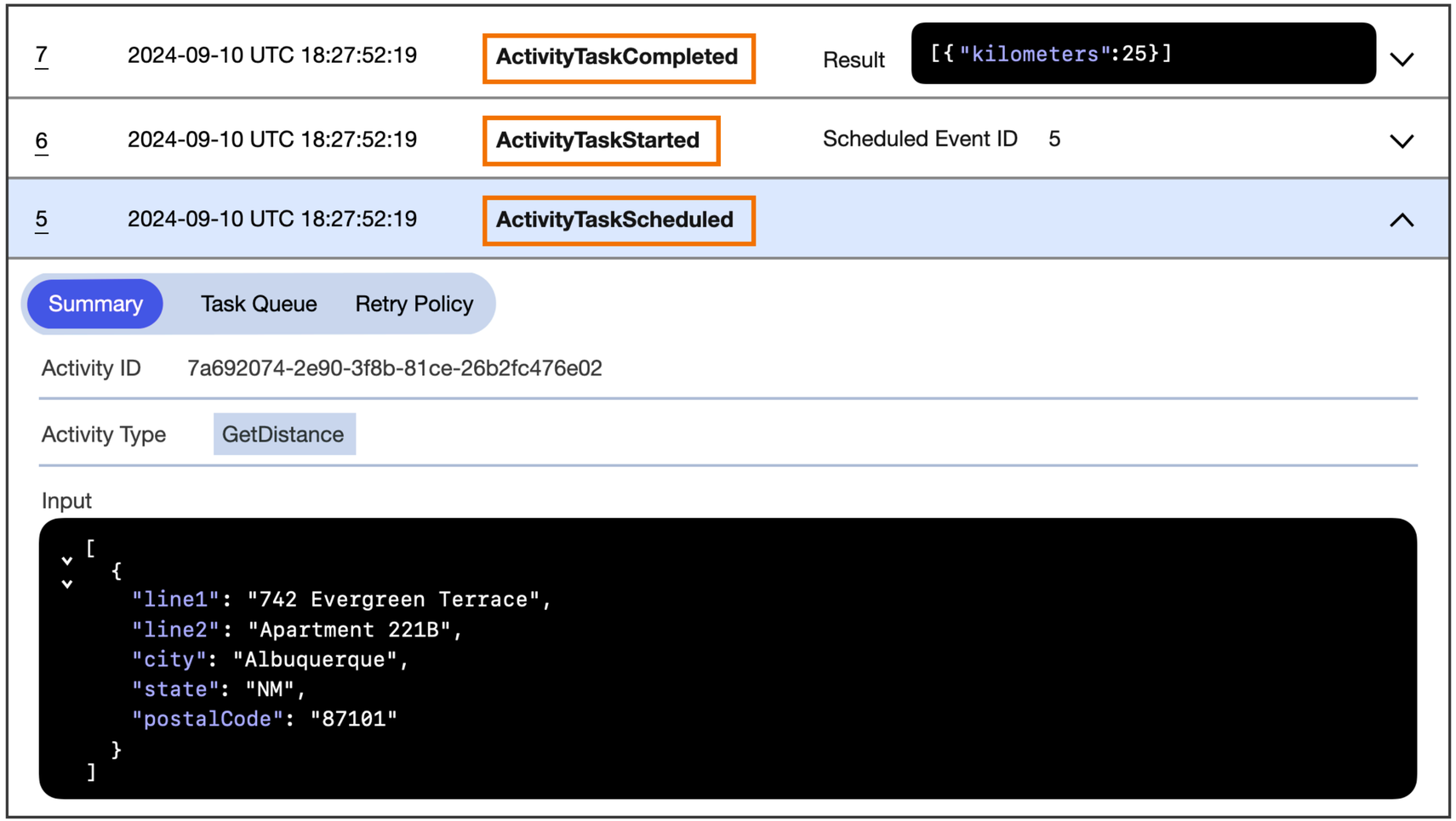
When a Workflow Task Failure Is Retried…

- **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

- **If Workflow code throws an exception that derives from `TemporalException`, the Workflow Execution will fail**
  - Unlike with a Workflow Task failure, there is no automatic retry
- **Remember that `ApplicationFailureException` extends `TemporalException`**
  - Developers may intentionally throw `ApplicationFailureException` from a Workflow Definition
    - This will cause the Workflow Execution to close with a status of Failed

Activity Execution: Sequence of Events (1)

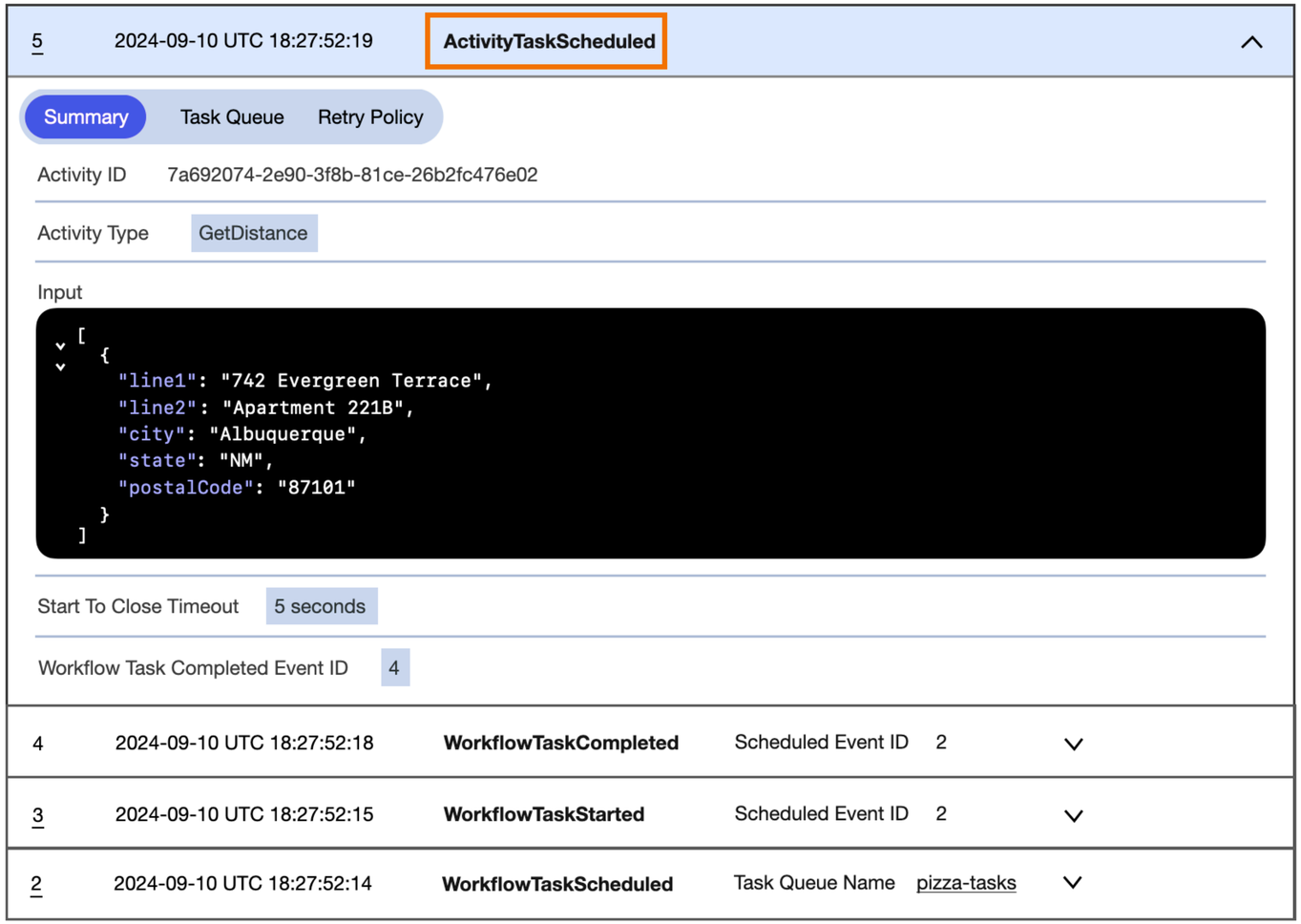


Activity Execution: Sequence of Events (2)

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

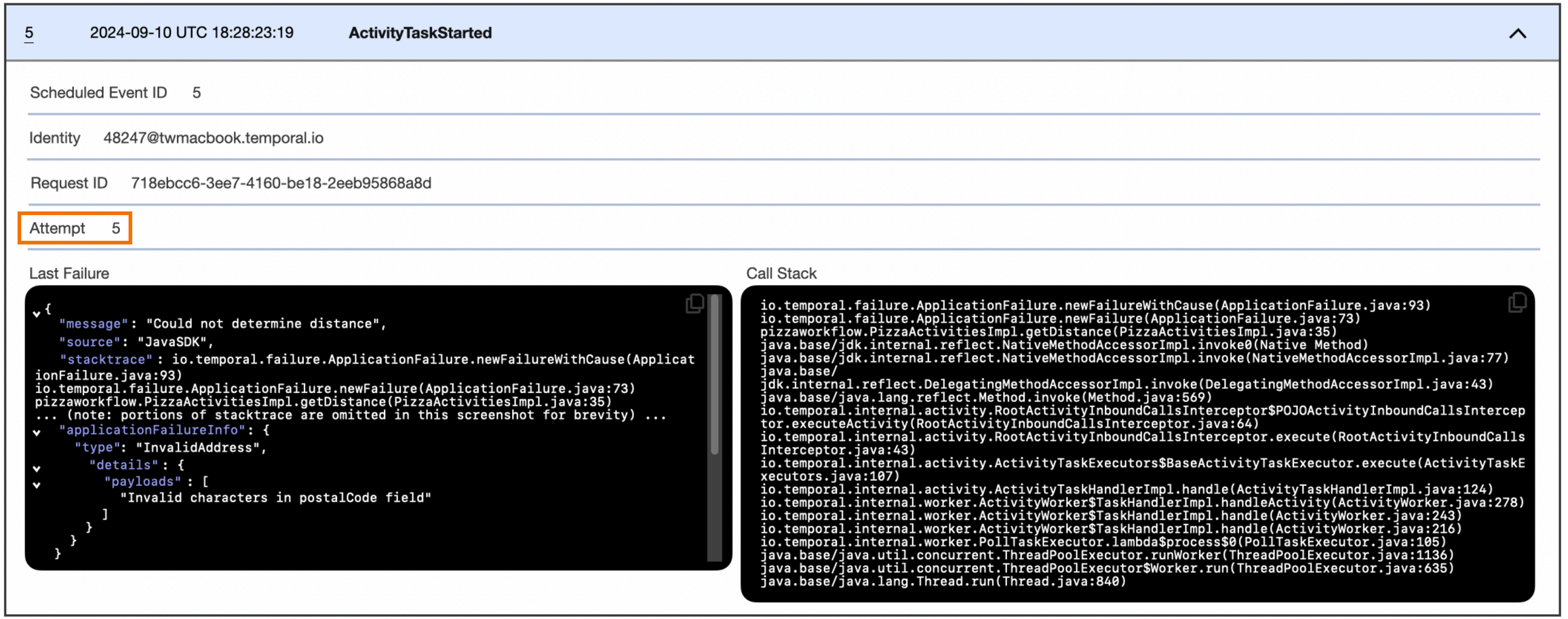
Viewing an Activity Execution (1)

- **`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



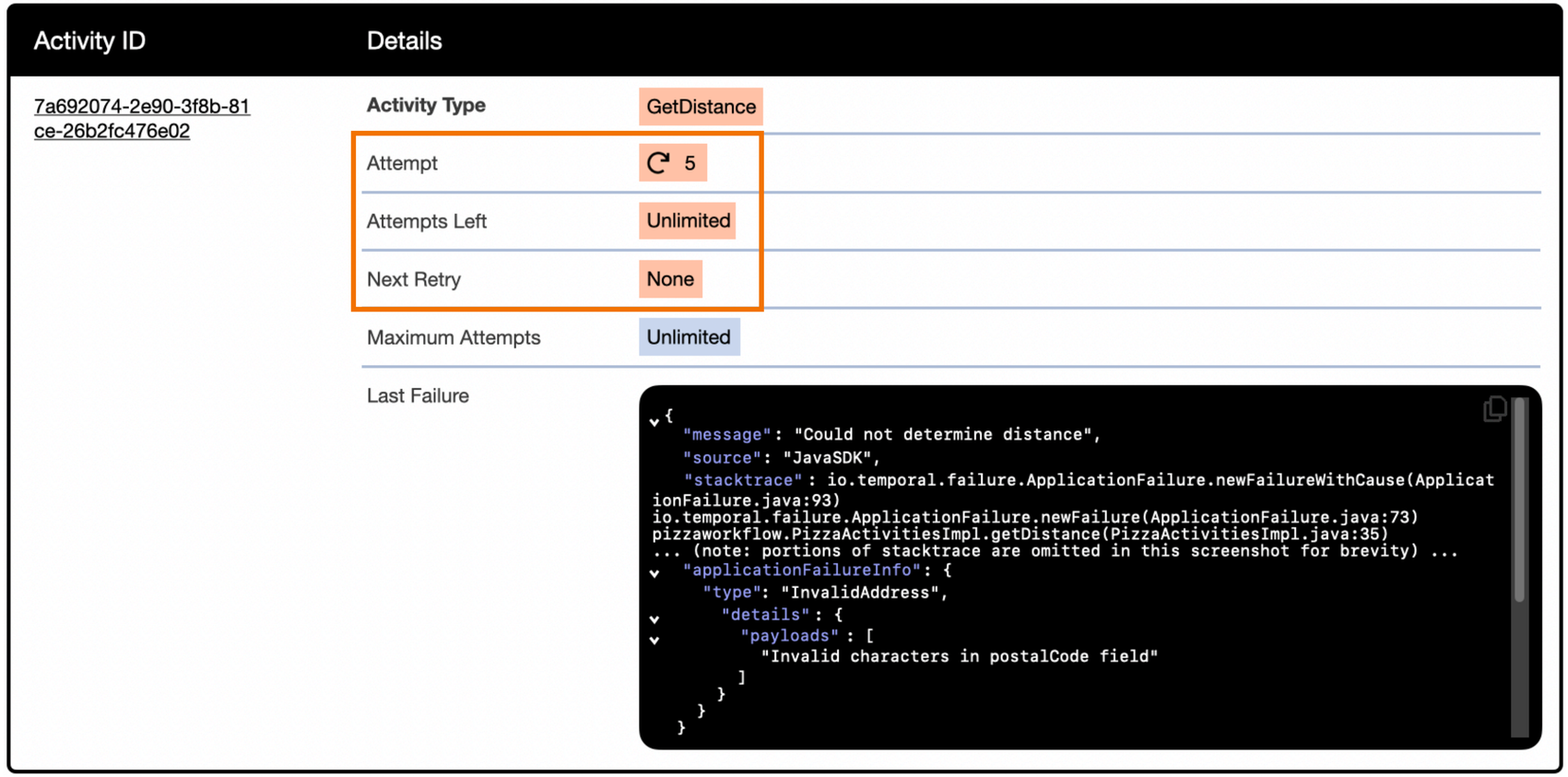
Viewing an Activity Execution (2)

- The `ActivityTaskStarted` Event contains the retry attempt count



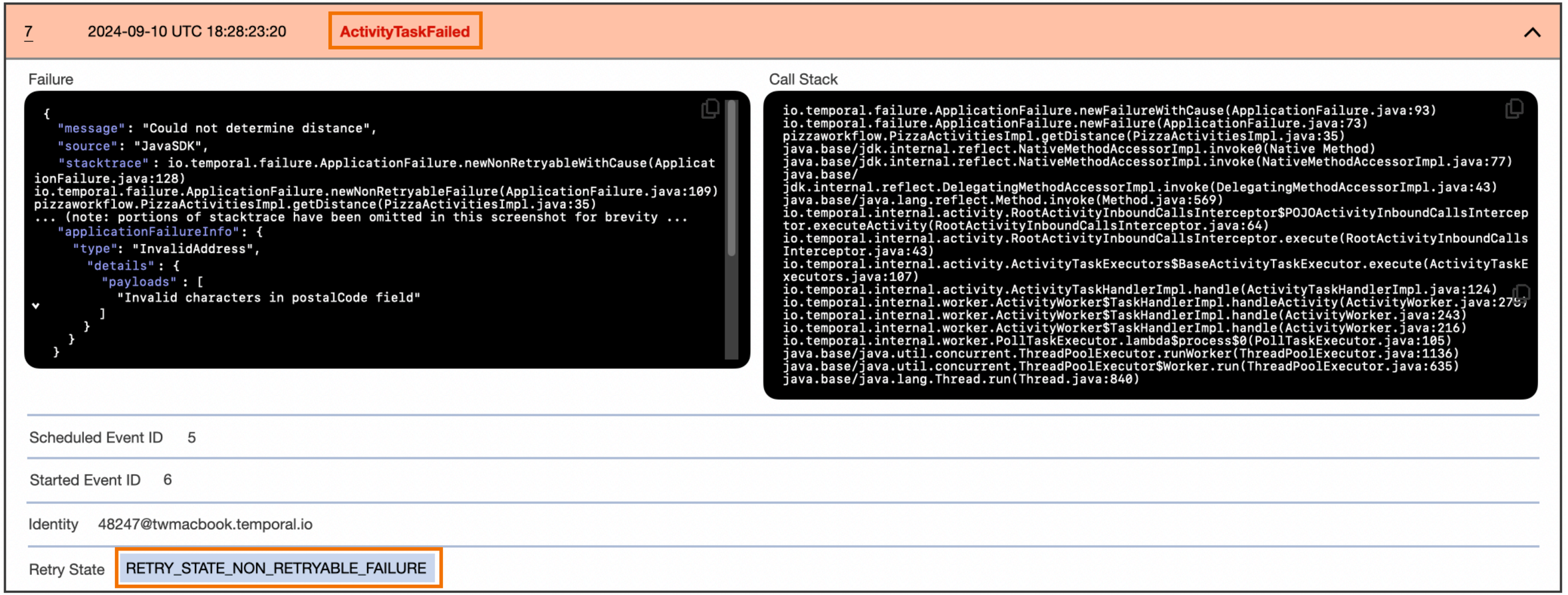
Viewing an Activity Execution (3)

- **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)



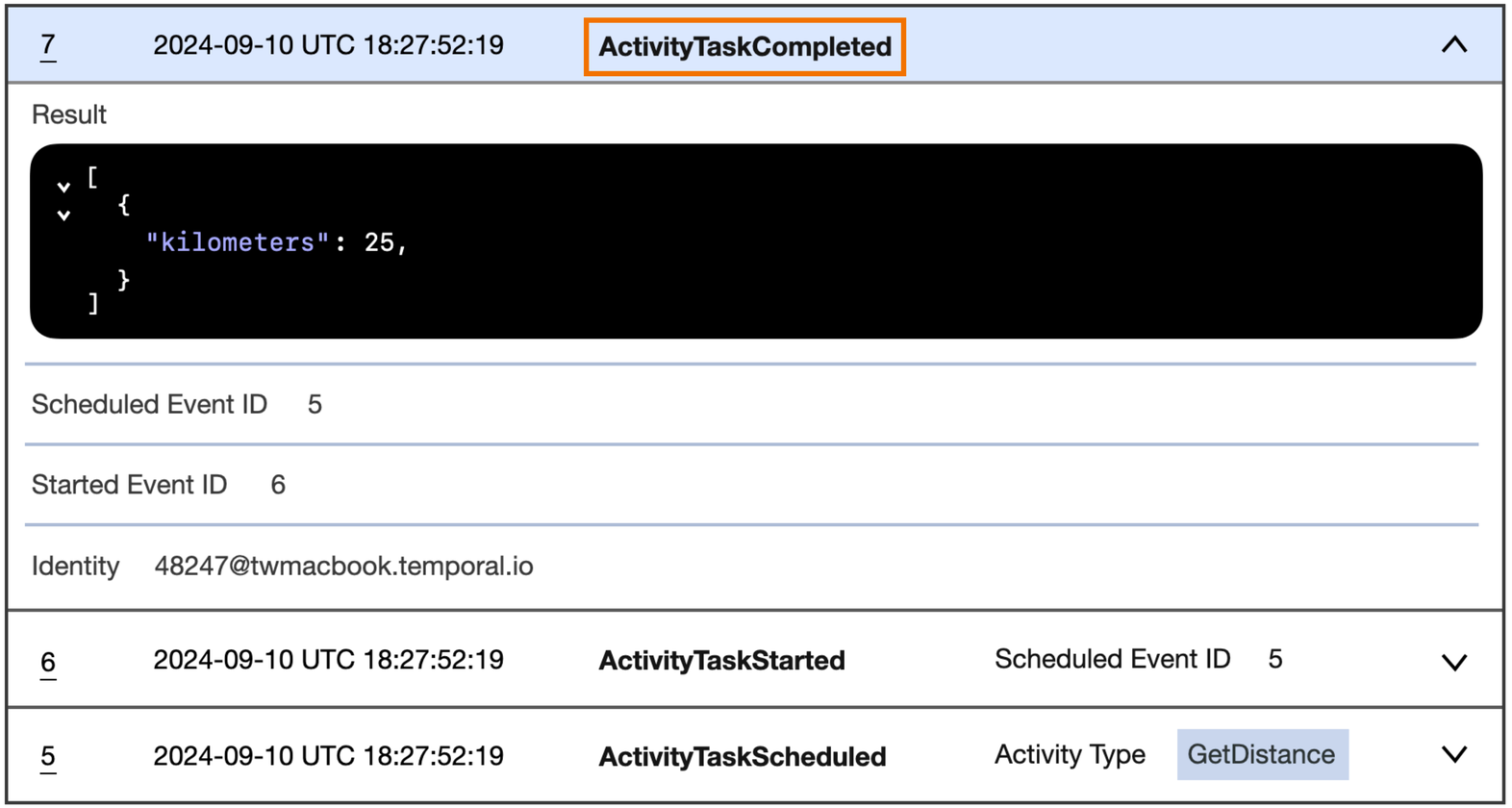
Viewing an Activity Execution (4)

- The `ActivityTaskFailed` Event provides details after the fact

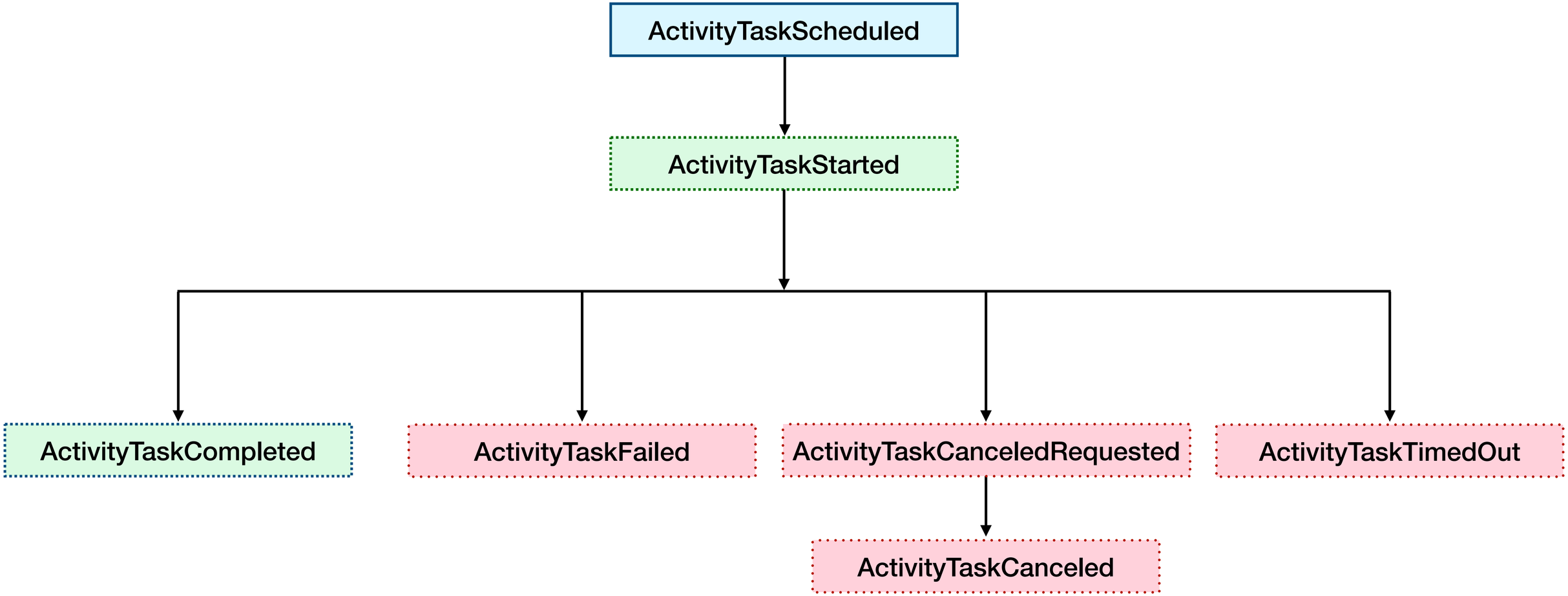


Viewing an Activity Execution (5)

- The `ActivityTaskCompleted` Event includes the result of execution

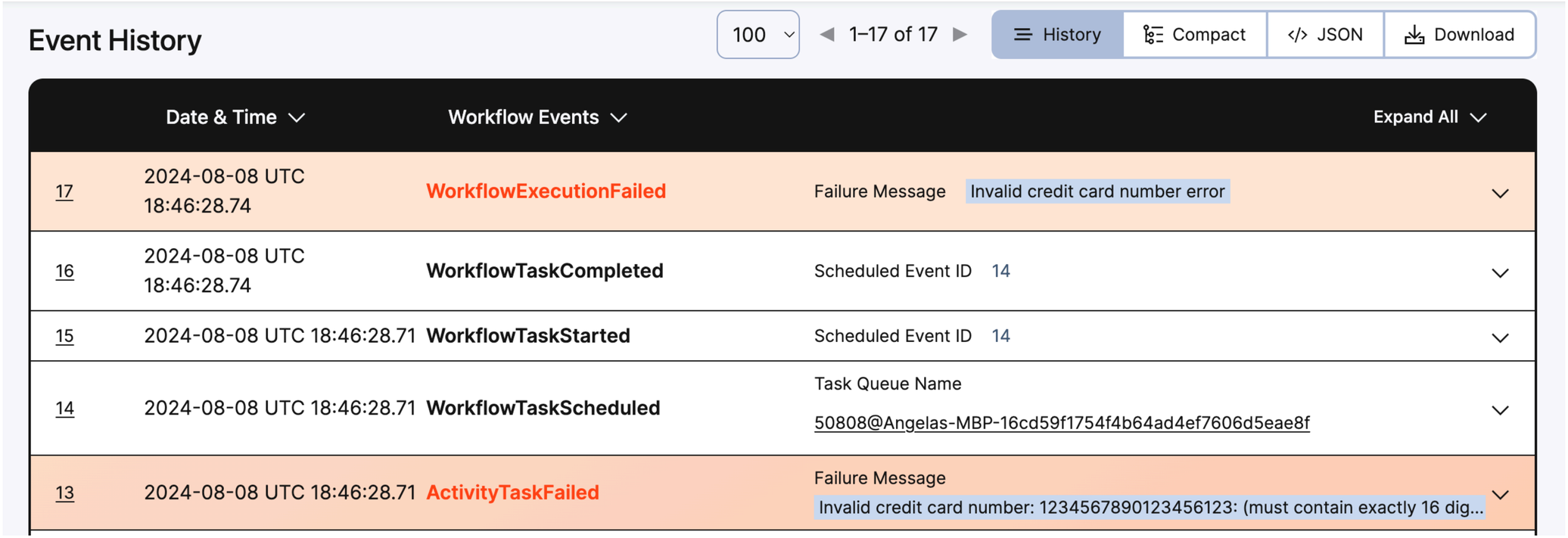


Events Related to Activity Execution



Workflow Execution Failure

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



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**
  - This is a concern for Activities in Temporal, since they may be executed multiple times
  - Temporal strongly recommends that you ensure you Activities are idempotent
- **In the .NET SDK, all failures descend from `TemporalException`**
  - You should not extend this class nor any of its subclasses
    - `ApplicationFailureException` is the only one that developers should throw
  - What happens when you throw an exception from your Workflow code depends on its type
    - If derived from `TemporalException`, Workflow Execution fails; if not, Workflow Task fails

Crafting an Error Handling Strategy

Throwing Exceptions from Activities (1)

- **Use Application Failures to communicate application-specific failures**
  - From both Workflows and Activities
- **Throwing an `ApplicationFailureException` from an Activity causes it to fail**

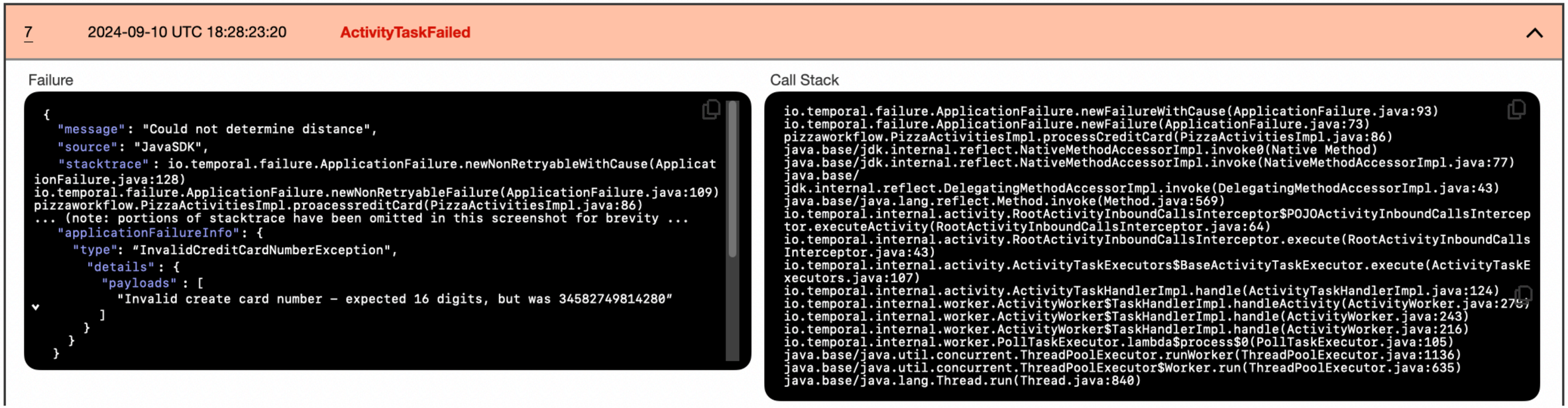
Throwing Exceptions from Activities (1)

- **Use Application Failures to communicate application-specific failures**
  - From both Workflows and Activities
- **Throwing an `ApplicationFailureException` from an Activity causes it to fail**
  - This will be represented as `ActivityTaskFailed` in the Event History
  - The Event will include the error message specified in the `ApplicationFailureException`

```
if (creditCardNumber.Length != 16)
{
    throw new ApplicationFailureException("Invalid credit card number: must contain
exactly 16 digits",
    details: new[] { creditCardNumber },
    errorType: "InvalidCreditCardErr");
}
```

Throwing Exceptions from Activities (2)

- **This is how that exception appears in the Event History**
  - The `ActivityTaskFailed` Event contains details of the failure



Throwing Exceptions from Activities (3)

- **Exception thrown from Activity is converted to `ApplicationFailureException`**
  - This is then wrapped in an `ActivityFailureException`
- **This wrapper provides some context, such as**
  - Activity Type
  - Retry Attempts
  - Cause
- **An Activity failure will never directly cause a Workflow Execution Failure**

Non-Retryable Errors for Activities

- **Recall that permanent errors require manual intervention**
  - Will continue to fail regardless of how many times you retry payment
- **Specify these as non-retryable so you can fix them manually**

Non-Retryable Errors for Activities

- **Recall that permanent errors require manual intervention**
  - Will continue to fail regardless of how many times you retry payment
- **Specify these as non-retryable so you can fix them manually**

```
var attempt = ActivityExecutionContext.Current.Info.Attempt;
throw new ApplicationFailureException(
    $"Something bad happened on attempt {attempt}",
    errorType: "my_failure_type",
    nonRetryable: true);
```

- **It is also possible to specify non-retryable *types* in the Retry Policy**

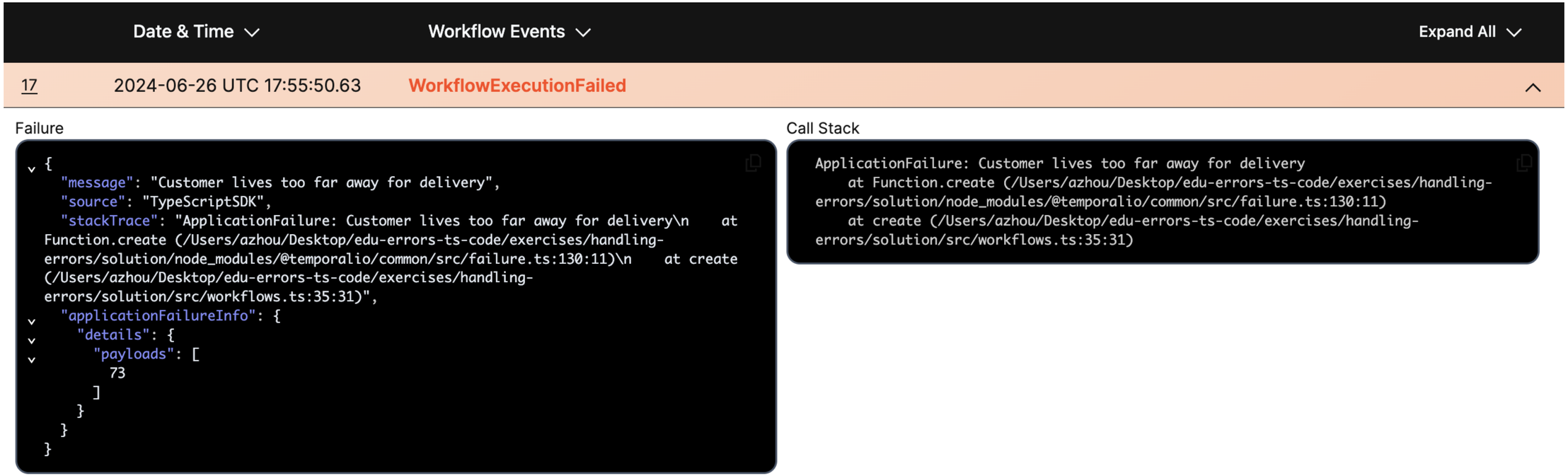
Throwing Exceptions from Workflows (1)

- **Throwing most exceptions from a Workflow cause *Workflow Task* to fail**
  - Workflow Tasks are automatically retried, although this results in History Replay
- **Throwing `ApplicationFailureException` fails a Workflow Execution**
  - `ApplicationFailureException` is the only subclass of `TemporalFailure` you should throw
  - This causes the Workflow Execution to close with a status of Failed

```
throw new ApplicationFailureException(
    $"Something bad happened",
    errorType: "my_failure_type");
```

Throwing Exceptions from Workflows (2)

- **This is how that exception appears in the Event History**
  - The `WorkflowExecutionFailed` Event contains details of the failure



Handling Problems in the Workflow

- **Subclasses of `TemporalException` may be visible to your Workflow code**
  - For example, `ApplicationFailureException` or `ActivityFailureException`
- **Allowing these to propagate will result in Workflow Execution failure**
  - You therefore need to catch and handle them

Exercise #1: Handling Errors

- **During this exercise, you will**
  - Throw and handle exceptions in Temporal Workflows and Activities
  - Use non-retryable 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

t.mp/edu-errstrat-dotnet-code

Throwing and Handling Exceptions Summary

- **Throwing `ApplicationFailureException` from an Activity fails it**
  - 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 throw an exception from a Workflow?**
  - It depends on whether that exception derives from `TemporalException`
    - If it does, then the *Workflow Execution* will fail
    - If it does not, then the current *Workflow Task* will fail (and will be retried)

Crafting an Error Handling Strategy

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 `ExecuteActivityAsync`

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

Activity Timeouts

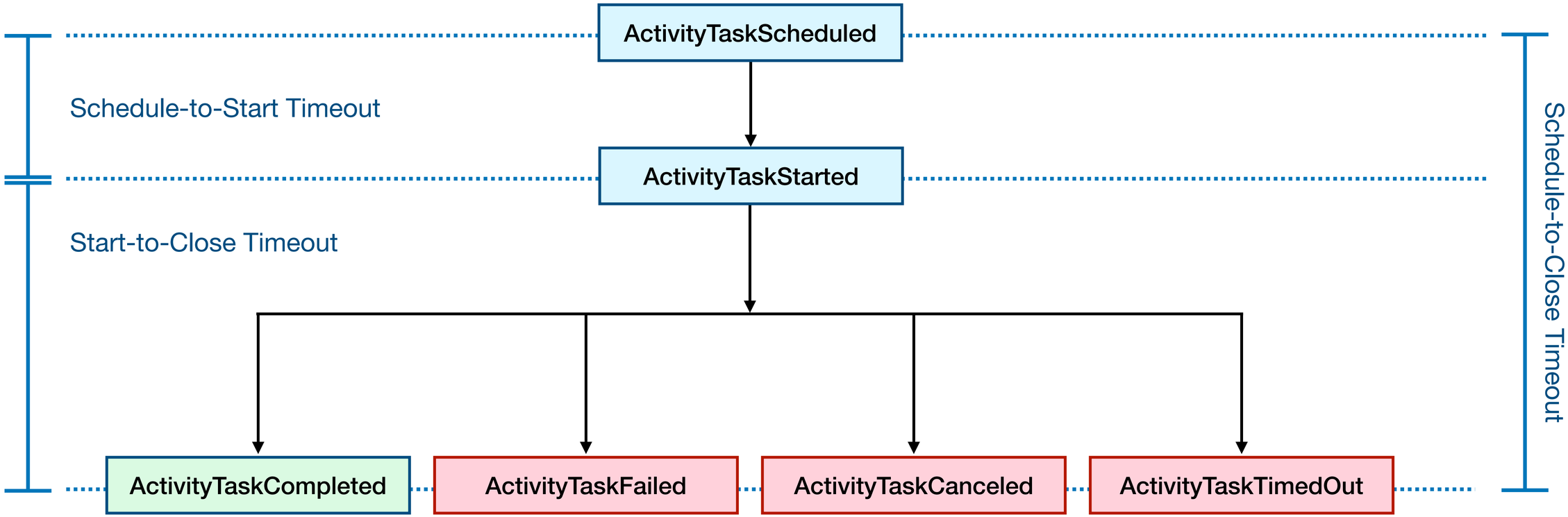
- Controls the maximum duration of a different aspect of an Activity Execution
- A measure of the time it takes to transition between one state to another
- Specified as an argument on the call to `ExecuteActivityAsync`
- As with an Activity that fails, an Activity that times out will be retried
  -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

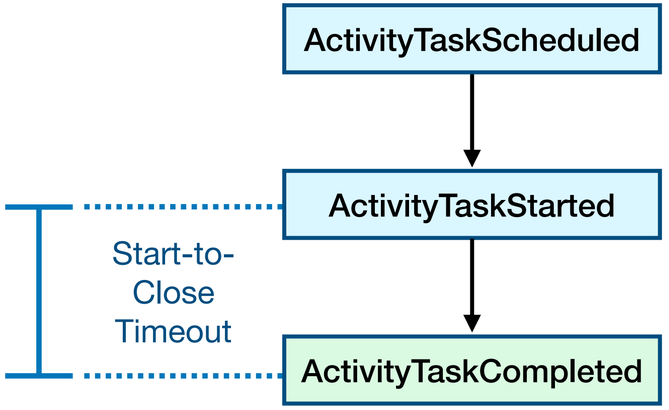
Understanding Activity Timeout Names



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

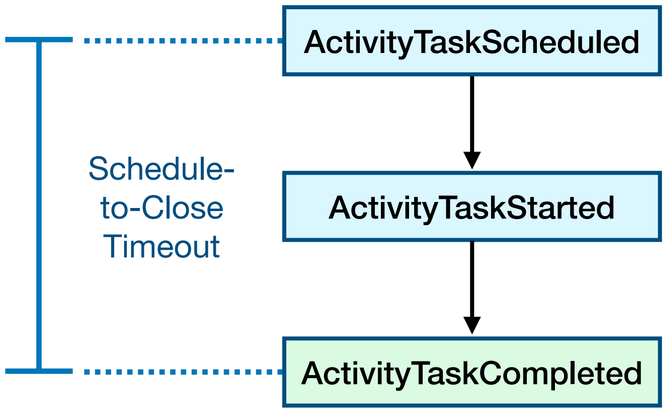
```
return await Workflow.ExecuteActivityAsync(
    (MyActivities a) => a.MyActivity(param),
    new() { StartToCloseTimeout = TimeSpan.FromMinutes(5) });
```



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

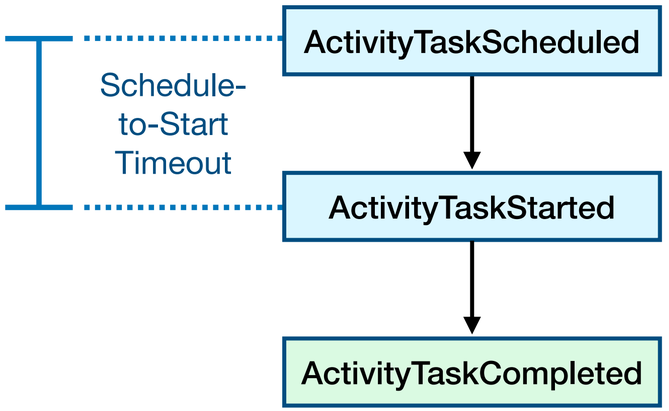
```
return await Workflow.ExecuteActivityAsync(
    (MyActivities a) => a.MyActivity(param),
    new() { ScheduleToCloseTimeout = TimeSpan.FromMinutes(5) });
```



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

```
return await Workflow.ExecuteActivityAsync(
    (MyActivities a) => a.MyActivity(param),
    new() { ScheduleToStartTimeout = TimeSpan.FromMinutes(5) });
```



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 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

```
var result = await client.ExecuteWorkflowAsync(
    (MyWorkflow wf) => wf.RunAsync(),
    new(id: "my-workflow-id", taskQueue: "my-task-queue")
    {
        WorkflowExecutionTimeout = TimeSpan.FromMinutes(5),
    });
```

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

```
var result = await client.ExecuteWorkflowAsync(
    (MyWorkflow wf) => wf.RunAsync(),
    new(id: "my-workflow-id", taskQueue: "my-task-queue")
    {
        WorkflowRunTimeout = TimeSpan.FromMinutes(5),
    });
```

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:
  - Progress indication
  - Worker Health Check
  - Cancellation Detection

How to Send a Heartbeat Message

```
public static async Task FakeProgressAsync(int sleepIntervalMs = 1000)
{
    // Allow for resuming from heartbeat
    var startingPoint = ActivityExecutionContext.Current.Info.HeartbeatDetails.Count > 0
        ? await ActivityExecutionContext.Current.Info.HeartbeatDetailAtAsync<int>(0)
        : 1;

    ActivityExecutionContext.Current.Logger.LogInformation("Starting activity at
progress: {StartingPoint}", startingPoint);

    for (var progress = startingPoint; progress <= 100; ++progress)
    {
        await Task.Delay(sleepIntervalMs,
ActivityExecutionContext.Current.CancellationToken);
        ActivityExecutionContext.Current.Logger.LogInformation("Progress: {Progress}",
progress);
        ActivityExecutionContext.Current.Heartbeat(progress);
    }
}
```

How to Send a Heartbeat Message

```
public static async Task FakeProgressAsync(int sleepIntervalMs = 1000)
{
    // Allow for resuming from heartbeat
    var startingPoint = ActivityExecutionContext.Current.Info.HeartbeatDetails.Count > 0
        ? await ActivityExecutionContext.Current.Info.HeartbeatDetailAtAsync<int>(0)
        : 1;

    ActivityExecutionContext.Current.Logger.LogInformation("Starting activity at
progress: {StartingPoint}", startingPoint);

    for (var progress = startingPoint; progress <= 100; ++progress)
    {
        await Task.Delay(sleepIntervalMs,
ActivityExecutionContext.Current.CancellationToken);
        ActivityExecutionContext.Current.Logger.LogInformation("Progress: {Progress}",
progress);
        ActivityExecutionContext.Current.Heartbeat(progress);
    }
}
```

How to Send a Heartbeat Message

```
public static async Task FakeProgressAsync(int sleepIntervalMs = 1000)
{
    // Allow for resuming from heartbeat
    var startingPoint = ActivityExecutionContext.Current.Info.HeartbeatDetails.Count > 0
        ? await ActivityExecutionContext.Current.Info.HeartbeatDetailAtAsync<int>(0)
        : 1;

    ActivityExecutionContext.Current.Logger.LogInformation("Starting activity at
progress: {StartingPoint}", startingPoint);

    for (var progress = startingPoint; progress <= 100; ++progress)
    {
        await Task.Delay(sleepIntervalMs,
ActivityExecutionContext.Current.CancellationToken);
        ActivityExecutionContext.Current.Logger.LogInformation("Progress: {Progress}",
progress);
        ActivityExecutionContext.Current.Heartbeat(progress);
    }
}
```

How to Send a Heartbeat Message

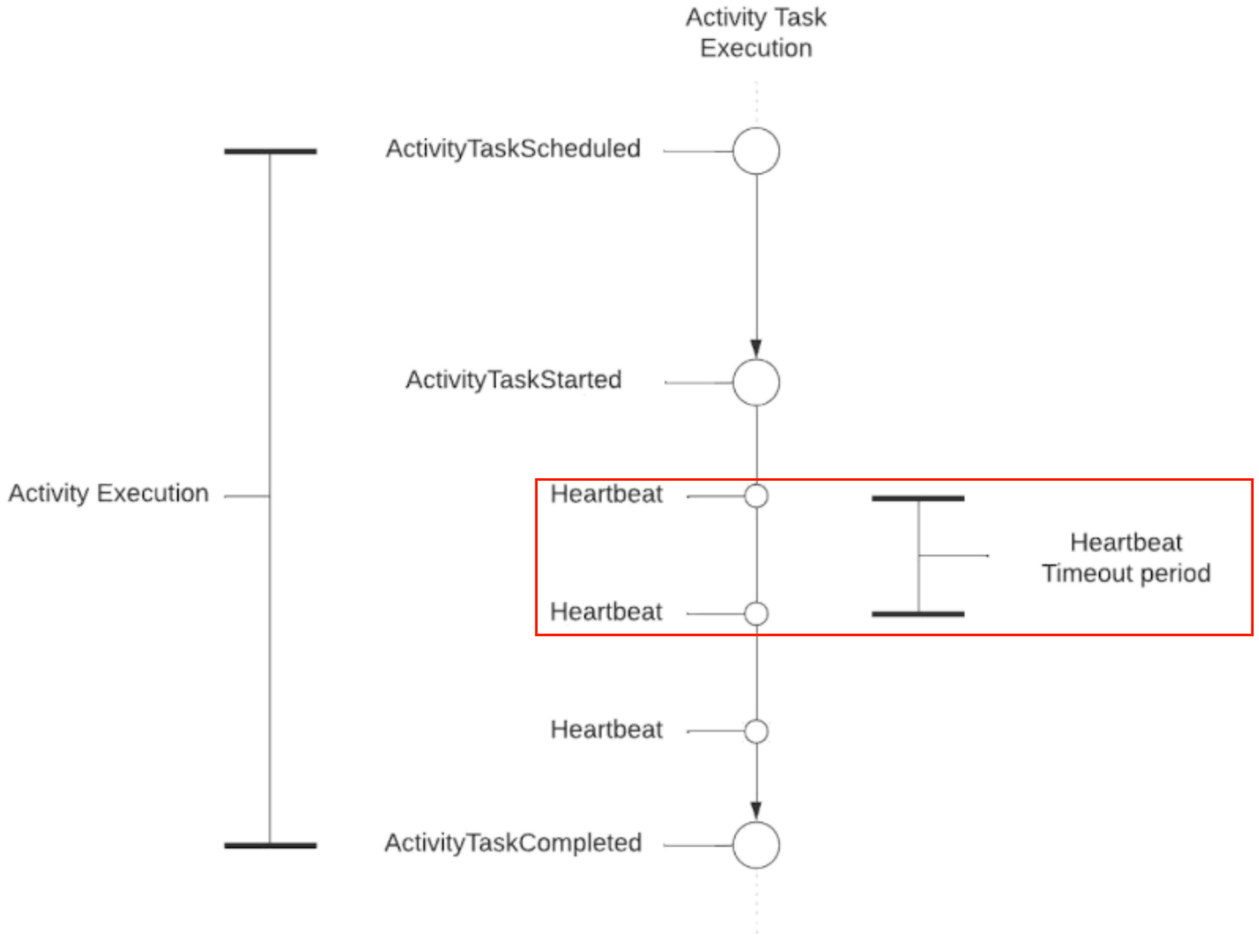
```
public static async Task FakeProgressAsync(int sleepIntervalMs = 1000)
{
    // Allow for resuming from heartbeat
    var startingPoint = ActivityExecutionContext.Current.Info.HeartbeatDetails.Count > 0
        ? await ActivityExecutionContext.Current.Info.HeartbeatDetailAtAsync<int>(0)
        : 1;

    ActivityExecutionContext.Current.Logger.LogInformation("Starting activity at
progress: {StartingPoint}", startingPoint);

    for (var progress = startingPoint; progress <= 100; ++progress)
    {
        await Task.Delay(sleepIntervalMs,
ActivityExecutionContext.Current.CancellationToken);
        ActivityExecutionContext.Current.Logger.LogInformation("Progress: {Progress}",
progress);
        ActivityExecutionContext.Current.Heartbeat(progress);
    }
}
```

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
- The Heartbeat Timeout must be set in order for Temporal to track the Heartbeats sent by the Activity

```
await Workflow.ExecuteActivityAsync(
    (MyActivities a) => a.MyActivity(param),
    new()
    {
        StartToCloseTimeout = TimeSpan.FromMinutes(5),
        HeartbeatTimeout = TimeSpan.FromSeconds(30),
    });
```

Heartbeat Timeout

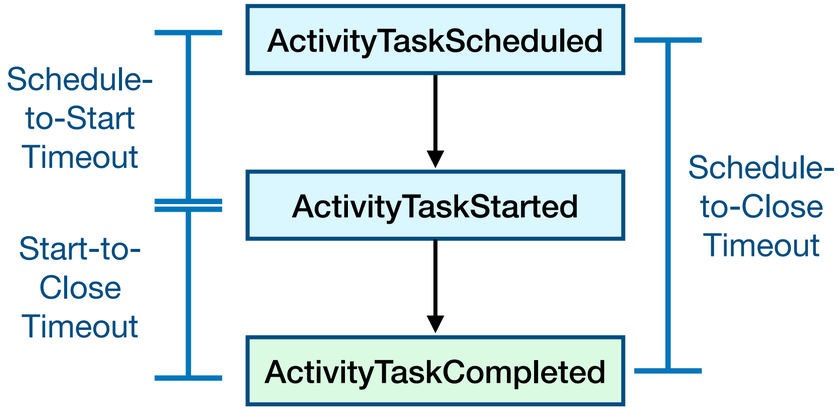
- To ensure efficient, handling of long-running Activities:
  - Set your Start-to-Close Timeout to be slightly longer than the maximum duration of your Activity
  - Your Heartbeat Timeout should be fairly short
- When the Heartbeat Timeout is specified, the Activity must send Heartbeats at intervals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

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

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



Crafting an Error Handling Strategy

Retry Policies

- **By default, Temporal automatically retries an Activity that fails**
  - A Retry Policy defines the details of how those retries are carried out
- **Unlike Activities, Workflow Executions are not retried by default**
  - While failed *Workflow Executions* are not retried automatically, failed *Workflow Tasks* are
  - Workflow Tasks retry automatically and 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 RetryPolicy by creating a `RetryPolicy`

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 list)

```
var retryPolicy = new RetryPolicy{
    BackoffCoefficient = 2.0,
    MaximumAttempts = 500
};
```

Retry Policy for Workflow Executions

- Workflow Executions do not retry by default
- 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

```
var options = new ActivityOptions
{
    StartToCloseTimeout = TimeSpan.FromSeconds(60),
    HeartbeatTimeout = TimeSpan.FromSeconds(30),
    RetryPolicy = new()
    {
        InitialInterval = TimeSpan.FromSeconds(1),
        BackoffCoefficient = 1,
        MaximumInterval = TimeSpan.FromSeconds(1),
        MaximumAttempts = 5,
        NonRetryableErrorTypes = new[]
{ "InvalidCreditCardErr" },
    },
};

await Workflow.ExecuteActivityAsync((Activities act) =>
act.ValidateCreditCardAsync(customer),options);
```

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
- 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 use the `RetryPolicy` for each different Activity Execution
- You can also customize a Retry Policy if an Activity is invoked conditionally

Defining Errors as Non-Retryable

```
if (creditCardNumber.Length != 16)
{
    throw new ApplicationFailureException("Invalid credit a number: must
contain exactly 16 digits",
    details: new[] { creditCardNumber }, errorType:
"InvalidCreditCardErr");
}
```

Defining Errors as Non-Retryable

- Non-retryable errors are specified in the list of non-retryable errors

```
var options = new ActivityOptions
{
    StartToCloseTimeout = TimeSpan.FromSeconds(60),
    HeartbeatTimeout = TimeSpan.FromSeconds(30),
    RetryPolicy = new()
    {
        InitialInterval = TimeSpan.FromSeconds(1),
        BackoffCoefficient = 1,
        MaximumInterval = TimeSpan.FromSeconds(1),
        MaximumAttempts = 5,
        NonRetryableErrorTypes = new[] { "InvalidCreditCardErr" },
    },
};

await Workflow.ExecuteActivityAsync((Activities act) =>
act.ValidateCreditCardAsync(customer),options);
```

Defining Errors as Non-Retryable

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

t.mp/edu-errstrat-dotnet-code

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)

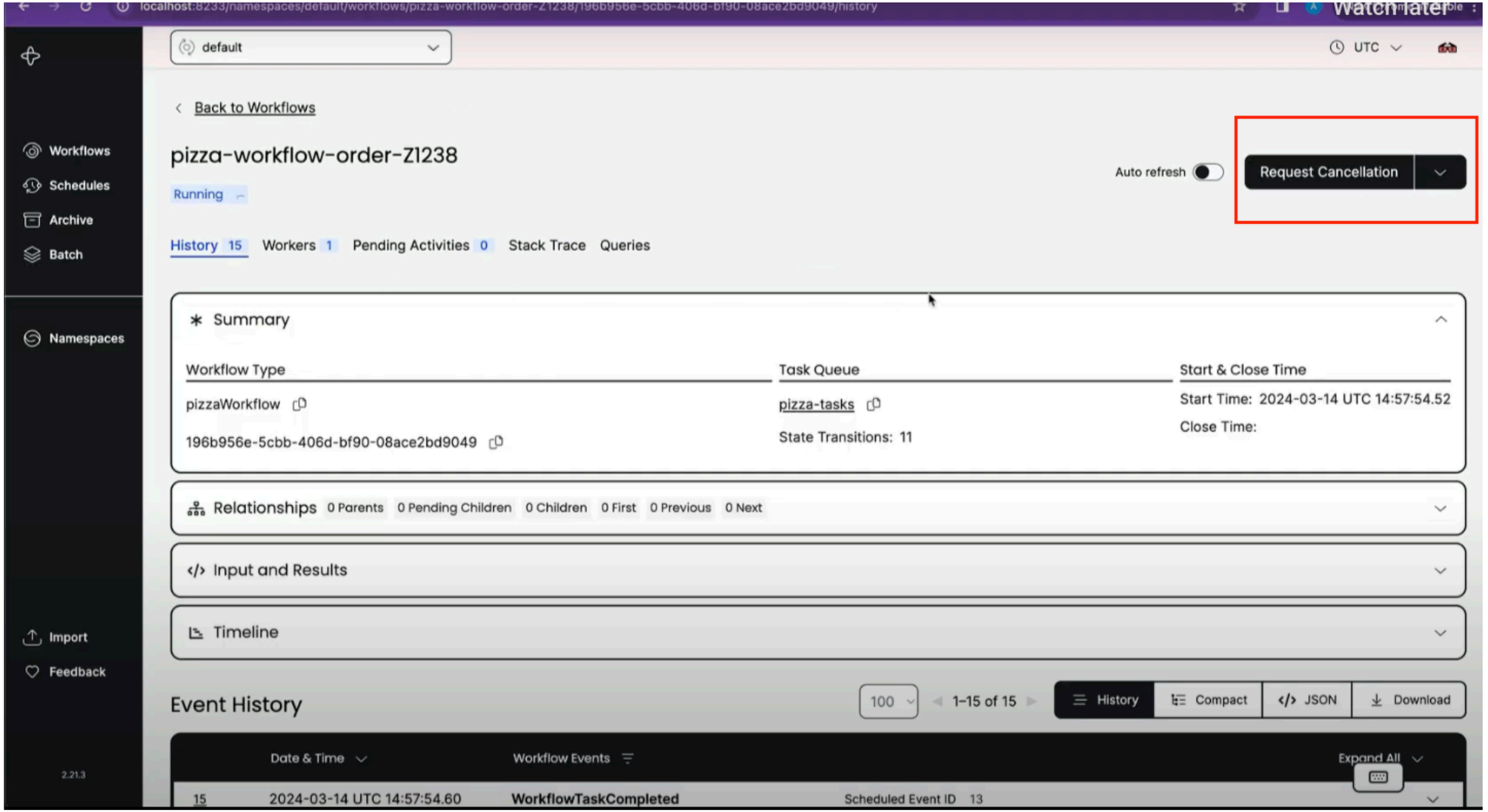
- **This Retry Policy is customizable**
  - You may wish to increase the delay or backoff coefficient for a specific intermittent failure
  - Every Activity Execution in a Workflow can specific a different Retry Policy
- **Use care when specifying maximum attempts in a Retry Policy**
  - Setting this 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

Crafting an Error Handling Strategy

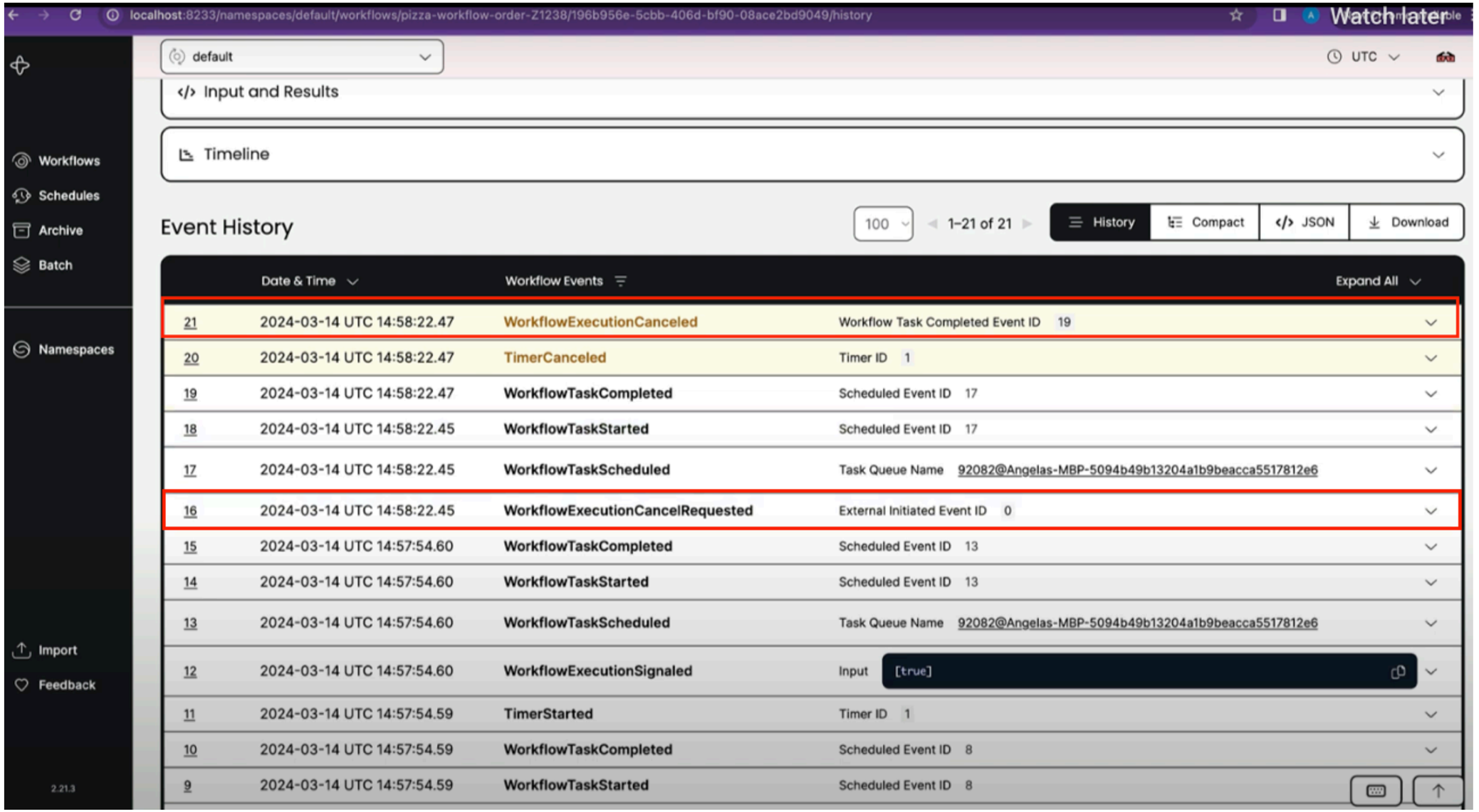
Handling a Workflow Execution that Cannot Complete

- Canceling your Workflow Execution

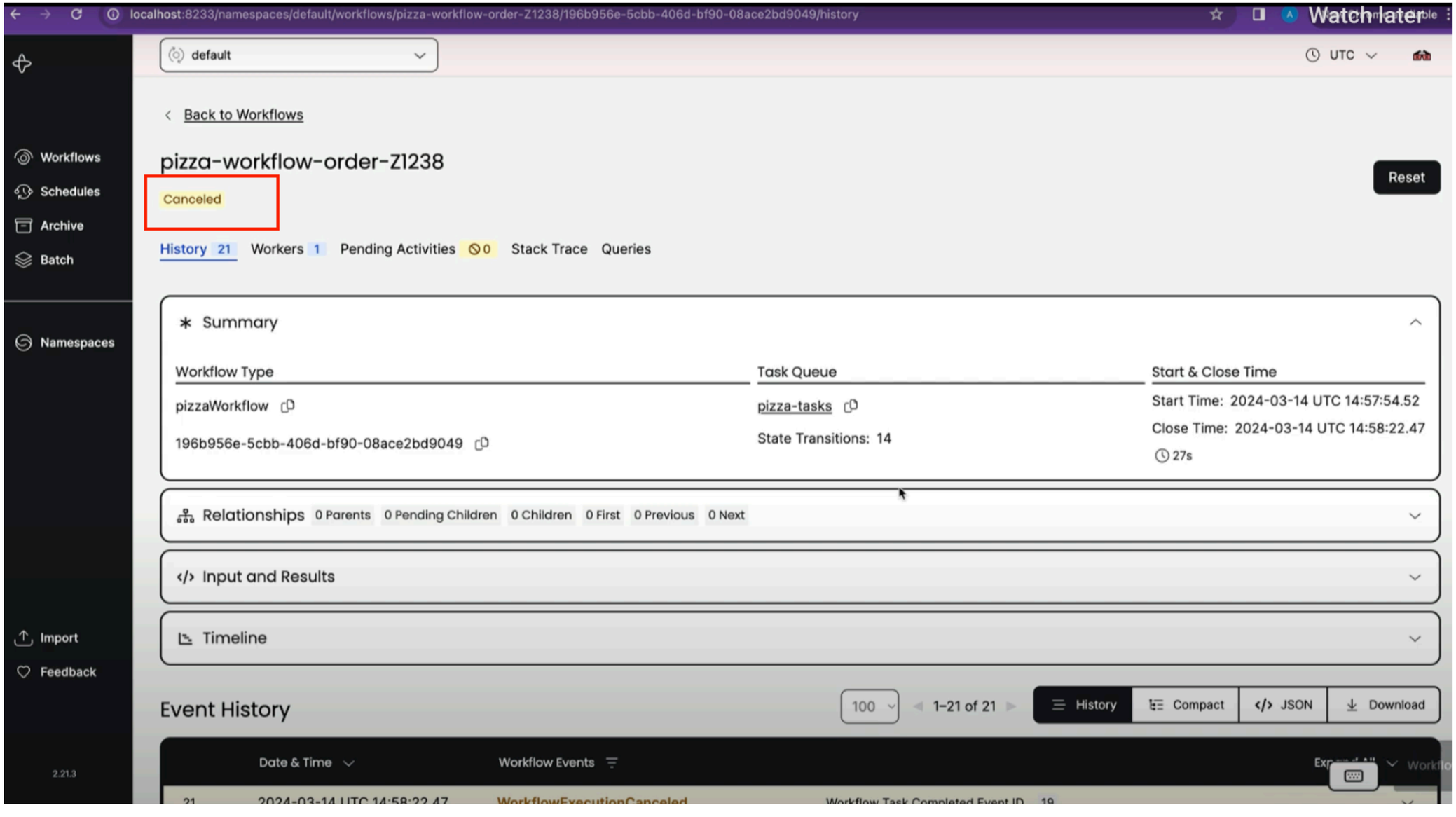
Canceling a Workflow Execution



Canceling a Workflow Execution



Canceling a Workflow Execution



Canceling a Workflow Execution from the CLI

```
temporal workflow cancel --workflow-id=meaningful-business-id
```

- Records a `WorkflowExecutionCancelRequested` Event in Event History
- A new Workflow Task will be scheduled, and the Workflow Execution performs cleanup work

Canceling a Workflow Execution with the SDK

- You need to use `WorkflowHandle` method to get a reference to the Workflow
- You will get the most recent run

```
var handle = myClient.GetWorkflowHandle("my-workflow-id");

await handle.CancelAsync();
```

Handling a Workflow Execution that Cannot Complete

- Canceling your Workflow Execution
- Terminating your Workflow Execution

Handling a Workflow Execution that Cannot Complete

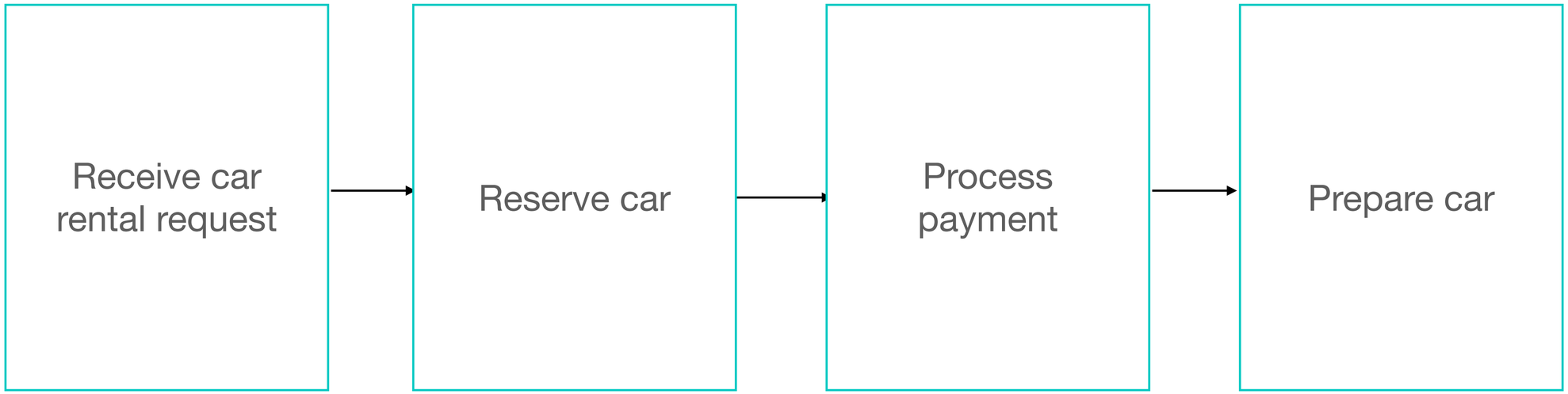
- Canceling your Workflow Execution
- Terminating your Workflow Execution
- Resetting your Workflow Execution

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

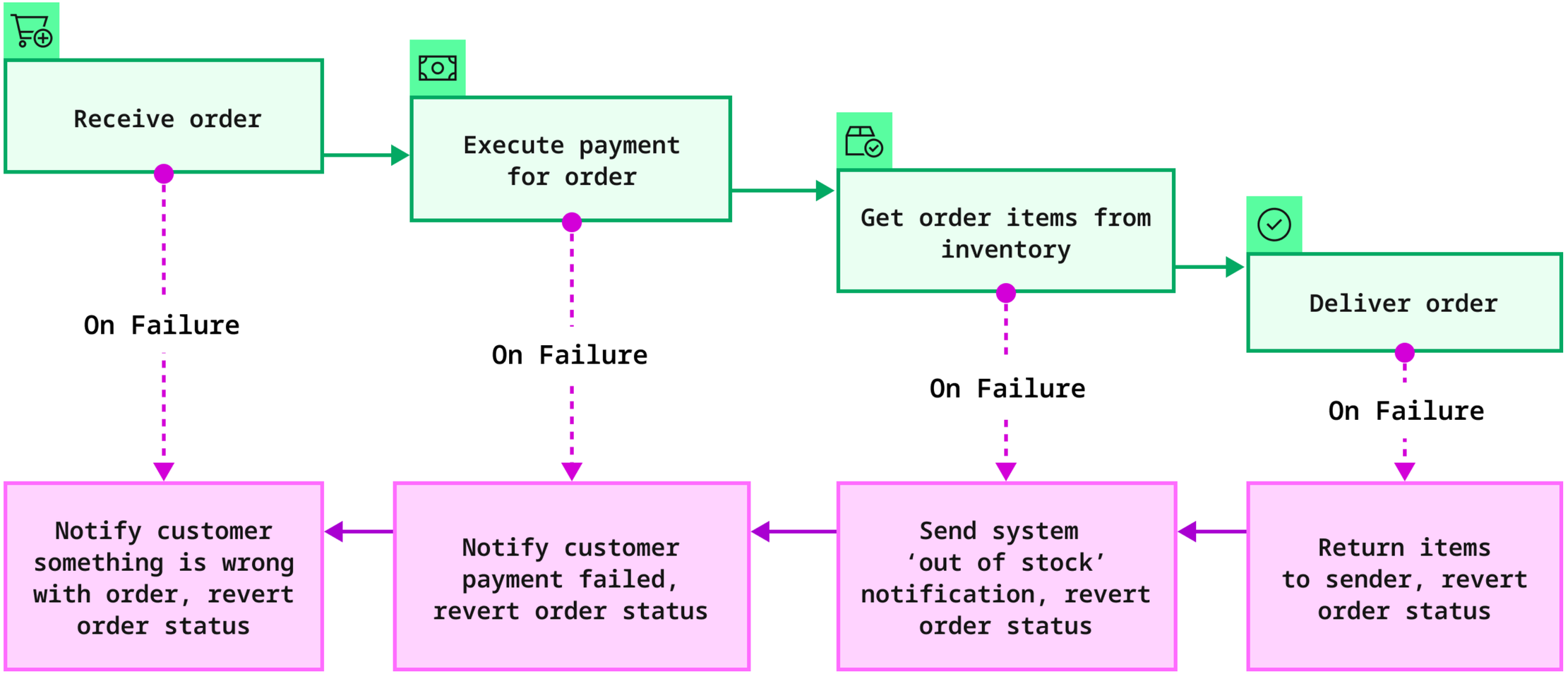
- A saga is a pattern used in distributed systems to manage a sequence of local transactions



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 executes actions to rollback the previous operations. This is known as a compensating action. Examples:
- Examples:
  - E-Commerce Transaction
  - Distributed Data Updates

Rollback Actions and the Saga Pattern



Rollback Actions and the Saga Pattern

```
private async Task CompensateAsync(List<Func<Task>> compensations)
{
    compensations.Reverse();
    foreach (var comp in compensations)
    {
        try
        {
            await comp.Invoke();
        }
        catch (Exception ex)
        {
            Workflow.Logger.LogError(ex, "Failed to compensate");
            // swallow errors
        }
    }
}
```

Rollback Actions and the Saga Pattern

```
[Workflow]
public class SagaWorkflow
{
    [WorkflowRun]
    public async Task RunAsync(TransferDetails transfer)
    {
        List<Func<Task>> compensations = new();

        var options = new ActivityOptions() { StartToCloseTimeout = TimeSpan.FromSeconds(90) };

        try
        {
            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.WithdrawCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Withdraw(transfer), options);

            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.DepositCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Deposit(transfer), options);

            // throw new Exception
            await Workflow.ExecuteActivityAsync(() => Activities.StepWithError(transfer), options);
        }
        catch (Exception)
        {
            await CompensateAsync(compensations);
            throw;
        }
    }
}
```

Rollback Actions and the Saga Pattern

```
[Workflow]
public class SagaWorkflow
{
    [WorkflowRun]
    public async Task RunAsync(TransferDetails transfer)
    {
        List<Func<Task>> compensations = new();

        var options = new ActivityOptions() { StartToCloseTimeout = TimeSpan.FromSeconds(90) };

        try
        {
            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.WithdrawCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Withdraw(transfer), options);

            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.DepositCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Deposit(transfer), options);

            // throw new Exception
            await Workflow.ExecuteActivityAsync(() => Activities.StepWithError(transfer), options);
        }
        catch (Exception)
        {
            await CompensateAsync(compensations);
            throw;
        }
    }
}
```

Rollback Actions and the Saga Pattern

```
[Workflow]
public class SagaWorkflow
{
    [WorkflowRun]
    public async Task RunAsync(TransferDetails transfer)
    {
        List<Func<Task>> compensations = new();

        var options = new ActivityOptions() { StartToCloseTimeout = TimeSpan.FromSeconds(90) };

        try
        {
            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.WithdrawCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Withdraw(transfer), options);

            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.DepositCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Deposit(transfer), options);

            // throw new Exception
            await Workflow.ExecuteActivityAsync(() => Activities.StepWithError(transfer), options);
        }
        catch (Exception)
        {
            await CompensateAsync(compensations);
            throw;
        }
    }
}
```

Rollback Actions and the Saga Pattern

```
[Workflow]
public class SagaWorkflow
{
    [WorkflowRun]
    public async Task RunAsync(TransferDetails transfer)
    {
        List<Func<Task>> compensations = new();

        var options = new ActivityOptions() { StartToCloseTimeout = TimeSpan.FromSeconds(90) };

        try
        {
            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.WithdrawCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Withdraw(transfer), options);

            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.DepositCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Deposit(transfer), options);

            // throw new Exception
            await Workflow.ExecuteActivityAsync(() => Activities.StepWithError(transfer), options);
        }
        catch (Exception)
        {
            await CompensateAsync(compensations);
            throw;
        }
    }
}
```

Rollback Actions and the Saga Pattern

```
[Workflow]
public class SagaWorkflow
{
    [WorkflowRun]
    public async Task RunAsync(TransferDetails transfer)
    {
        List<Func<Task>> compensations = new();

        var options = new ActivityOptions() { StartToCloseTimeout = TimeSpan.FromSeconds(90) };

        try
        {
            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.WithdrawCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Withdraw(transfer), options);

            compensations.Add(async () => await Workflow.ExecuteActivityAsync(
                () => Activities.DepositCompensation(transfer), options));
            await Workflow.ExecuteActivityAsync(() => Activities.Deposit(transfer), options);

            // throw new Exception
            await Workflow.ExecuteActivityAsync(() => Activities.StepWithError(transfer), options);
        }
        catch (Exception)
        {
            await CompensateAsync(compensations);
            throw;
        }
    }
}
```

Xd

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

t.mp/edu-errstrat-dotnet-code

Recovering from Failure Summary (1)

- **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

Recovering from Failure Summary (2)

- **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

Crafting an Error Handling Strategy

Error Handling Concepts Summary (1)

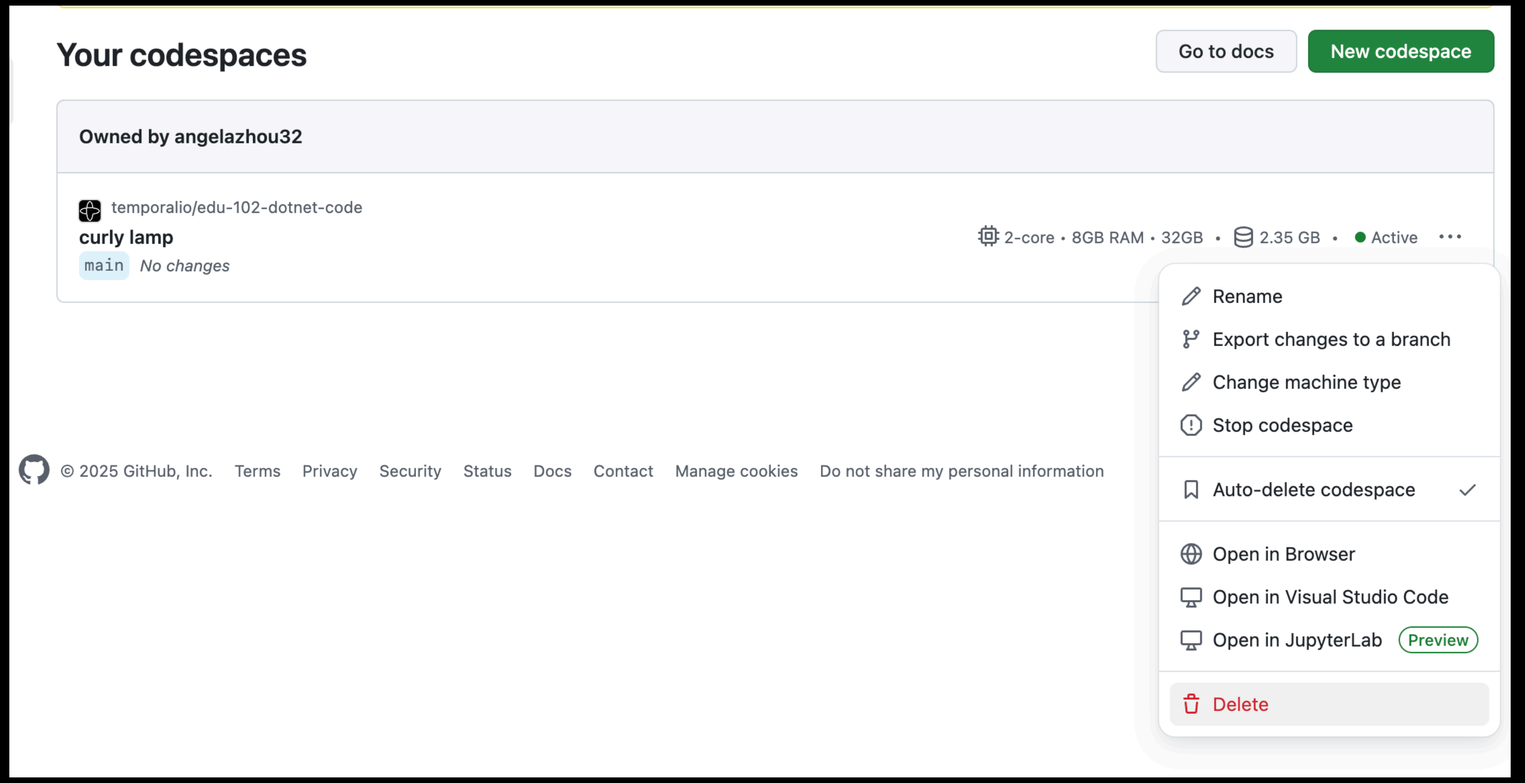
- **You can categorize failures are either platform or application**
  - **Platform**: occur from reasons beyond the control of your application code
  - **Application**: caused by problems with application code or input data
  - Determine which by considering if detecting and fixing requires knowledge 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 you Activities are idempotent
- **In the .NET SDK, all failures descend from `TemporalFailureException`**
  - You should not extend this class nor any of its subclasses
    - `ApplicationFailureException` is the only one that developers should throw
  - What happens when you throw an exception from your Workflow code depends on its type
    - If derived from `TemporalFailureException`, Workflow Execution fails

Don’t forget to manually delete your code spaces

https://github.com/codespaces



Thank you for your time and attention

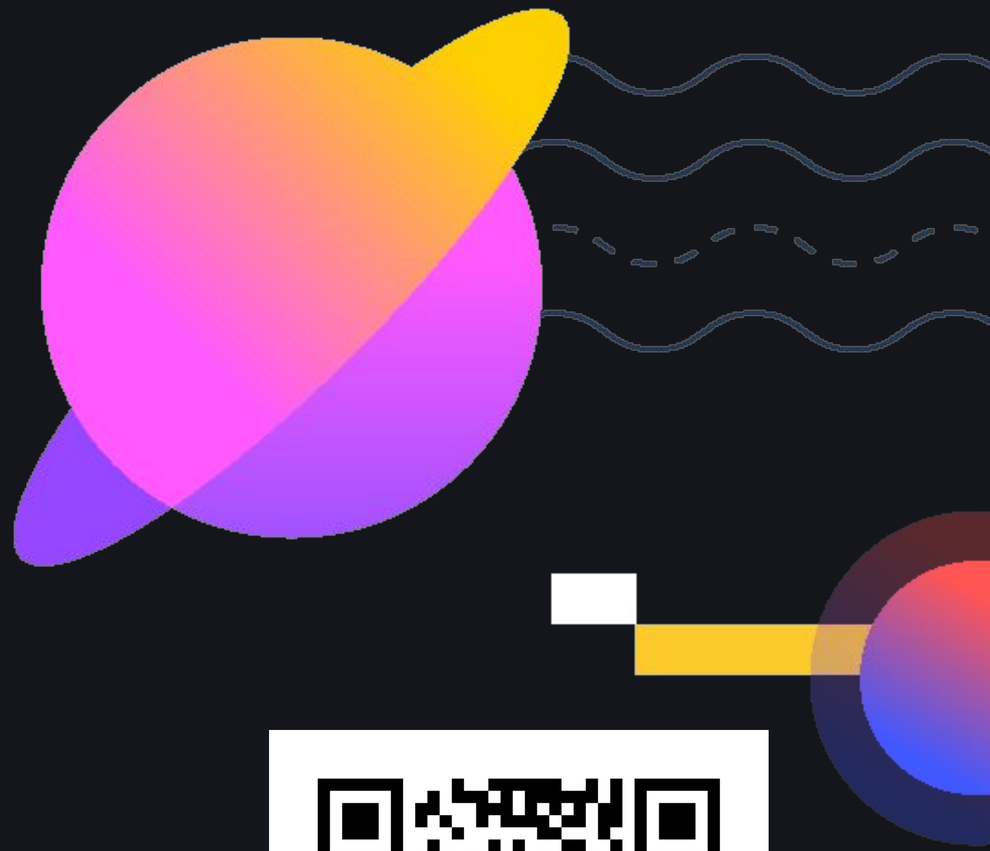
We welcome your feedback

t.mp/replay25ws

TEMPORAL'S CODE EXCHANGE

Share what you've built with Temporal

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



TEMPORAL.IO/CODE-EXCHANGE