100 Temporal Mistakes

...and how to avoid them!

What we'll cover

- Activities: Errors, retries, idempotency, timeouts, worker disruption
- Workflows: Runtime limitations, determinism, change versioning
- Recommendations to simplify working with Temporal day to day

Follow Along:

Follow along: jacob.work/100TM

Jacob LeGrone

100 Temporal Mistakes

...and how to avoid them!

What we'll cover

- Activities: Errors, retries, idempotency, timeouts, worker disruption
- Workflows: Runtime limitations, determinism, change versioning
- Recommendations to simplify working with Temporal day to day

Follow Along:

Follow along: jacob.work/100TM

Jacob LeGrone

100 Temporal Mistakes

...and how to avoid them!

What we'll cover

- Activities: Errors, retries, idempotency, timeouts, worker disruption
- Workflows: Runtime limitations, determinism, change versioning
- Recommendations to simplify working with Temporal day to day

Follow Along:

Follow along: jacob.work/100TM

Jacob LeGrone

Part One: Activities

Activities are how Temporal workers interact with the outside world.

Activities must deal with:

- Downstream service errors
- Worker crashes & hangs
- Temporal server disruptions

Temporal's shared responsibility model for activities:

Temporal server provides an "at least once" execution semantic and retries activities by default.

It's on us to:

- Implement idempotency
- Gracefully handle cancelation
- Not amplify bad requests

Temporal's shared responsibility model for activities:

Temporal server provides an "at least once" execution semantic and retries activities by default.

It's on us to:

- Implement idempotency
- Gracefully handle cancelation
- Not amplify bad requests

Activities: Handling Downstream Service Errors

```
func ChargePayment(ctx context.Context, req ChargeRequest) (*ChargeResponse, error) {
    httpReq := newPaymentHTTPReq(req) // POST api.example.com/v1/payments/charge
    // Send request
    resp, err := httpClient.Do(httpReq)
    if err != nil { return nil, err }

    switch resp.StatusCode {
    case http.StatusOK:
        // Decode the response and return
        var cr ChargeResponse
        err = json.NewDecoder(resp.Body).Decode(&cr)
        return &cr, err
    default:
        return nil, fmt.Errorf("unexpected http status: %s", resp.StatusCode)
    }
}
```

Activities: Avoid Amplifying Invalid Requests

```go
func ChargePayment(ctx context.Context, req ChargeRequest) (*ChargeResponse, error) {
    httpReq := newPaymentHTTPReq(req) // POST api.example.com/v1/payments/charge
    // Send request ...

    switch resp.StatusCode {
    case http.StatusOK: /* Decode the response and return ... */
    case http.StatusBadRequest:
        // Return non-retryable error
        return nil, temporal.NewNonRetryableApplicationError(resp.Status, "http_400", nil)
    default:
        return nil, fmt.Errorf("unexpected http status: %s", resp.StatusCode)
    }
}
```

Activities: Avoid Overloading Services With Retries

```go
func ChargePayment(ctx context.Context, req ChargeRequest) (*ChargeResponse, error) {
    httpReq := newPaymentHTTPReq(req) // POST api.example.com/v1/payments/charge
    // Send request ...

    switch resp.StatusCode {
    case http.StatusOK: /* Decode the response and return ... */
    case http.StatusBadRequest: /* Return non-retryable error ... */
    case http.StatusTooManyRequests:
        // Honor the server's hint when present; otherwise back off
        // more aggressively than the policy's default.
        delay := activityhelpers.ParseRetryAfter(resp.Header.Get("Retry-After"), time.Now())
        if delay == 0 {
            delay = activityhelpers.GetNextRetryDelay(ctx) * 2
        }
        return nil, temporal.NewApplicationErrorWithOptions(resp.Status, "http_429",
            temporal.ApplicationErrorOptions{NextRetryDelay: delay})
    default:
        return nil, fmt.Errorf("unexpected http status: %s", resp.StatusCode)
    }
}
```

Activities: Avoid Overloading Services With Retries

```go
func ChargePayment(ctx context.Context, req ChargeRequest) (*ChargeResponse, error) {
    httpReq := newPaymentHTTPReq(req) // POST api.example.com/v1/payments/charge
    // Send request ...

    switch resp.StatusCode {
    case http.StatusOK: /* Decode the response and return ... */
    case http.StatusBadRequest: /* Return non-retryable error ... */
    case http.StatusTooManyRequests:
        // Honor the server's hint when present; otherwise back off
        // more aggressively than the policy's default.
        delay := activityhelpers.ParseRetryAfter(resp.Header.Get("Retry-After"), time.Now())
        if delay == 0 {
            delay = activityhelpers.GetNextRetryDelay(ctx) * 2
        }
        return nil, temporal.NewApplicationErrorWithOptions(resp.Status, "http_429",
            temporal.ApplicationErrorOptions{NextRetryDelay: delay})
    default:
        return nil, fmt.Errorf("unexpected http status: %s", resp.StatusCode)
    }
}
```

Activities: Weathering System Outages

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ... generate a charge request for the item & customer

    ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
        StartToCloseTimeout:    30 * time.Second,
        ScheduleToCloseTimeout: time.Minute,
        RetryPolicy: { /* ... */ },
    })
    resp, err := workflowhelpers.AwaitActivity(ctx, ChargePayment, chargeRequest)

    // ...
}
```

Activities: Weathering System Outages

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ... generate a charge request for the item & customer

    ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
        StartToCloseTimeout:    30 * time.Second,
        ScheduleToCloseTimeout: time.Minute,
        RetryPolicy: { /* ... */ },
    })
    resp, err := workflowhelpers.AwaitActivity(ctx, ChargePayment, chargeRequest)

    // ...
}
```

Activities: Weathering System Outages

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ... generate a charge request for the item & customer

    ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
        StartToCloseTimeout:    30 * time.Second,
        ScheduleToCloseTimeout: time.Minute,
        RetryPolicy: { /* ... */ },
    })
    resp, err := workflowhelpers.AwaitActivity(ctx, ChargePayment, chargeRequest)

    // ...
}
```

Activities: Weathering System Outages (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
     // ... generate a charge request for the item & customer

     ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
         StartToCloseTimeout:    30 * time.Second,
-        ScheduleToCloseTimeout: time.Minute,
+        ScheduleToCloseTimeout: time.Hour, // allow retrying for up to 1 hour
         RetryPolicy: { /* ... */ },
     })

     resp, err := workflowhelpers.AwaitActivity(ctx, ChargePayment, chargeRequest)
     // ...
 }
```

Activities: Weathering System Outages (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
     // ... generate a charge request for the item & customer

     ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
         StartToCloseTimeout:    30 * time.Second,
-        ScheduleToCloseTimeout: time.Minute,
+        ScheduleToCloseTimeout: time.Hour, // allow retrying for up to 1 hour
         RetryPolicy: { /* ... */ },
     })

     resp, err := workflowhelpers.AwaitActivity(ctx, ChargePayment, chargeRequest)
     // ...
 }
```

Activities: Weathering System Outages (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
     // ... generate a charge request for the item & customer

     ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
         StartToCloseTimeout:    30 * time.Second,
-        ScheduleToCloseTimeout: time.Minute,
+        ScheduleToCloseTimeout: time.Hour, // allow retrying for up to 1 hour
         RetryPolicy: { /* ... */ },
     })

     resp, err := workflowhelpers.AwaitActivity(ctx, ChargePayment, chargeRequest)
     // ...
 }
```

Activities: Weathering System Outages (continued)

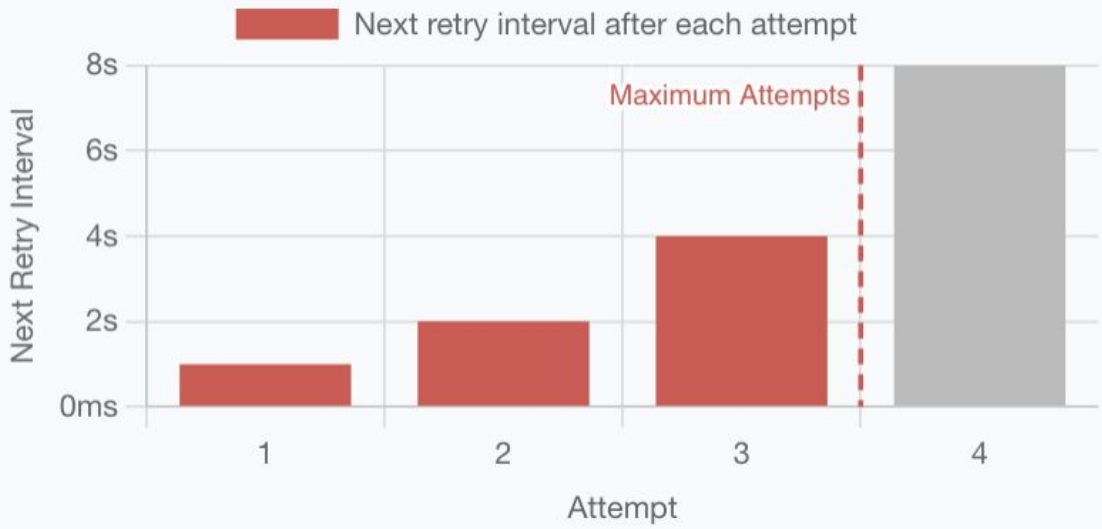
```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ... generate a charge request for the item & customer

    ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
        StartToCloseTimeout:    30 * time.Second,
        ScheduleToCloseTimeout: time.Hour, // allow retrying for up to 1 hour
        RetryPolicy: &temporal.RetryPolicy{
            MaximumAttempts:    3,
            InitialInterval:    time.Second,
            BackoffCoefficient: 2,
            MaximumInterval:    100 * time.Second,
        },
    })

    // ... execute the ChargePayment activity
}
```

Activities: Weathering System Outages (continued)

Status	Failed
Result	ActivityTaskFailed
Details	Exhausted Maximum Attempts
Time Elapsed	6 seconds 300 milliseconds
Attempts	3



docs.temporal.io/develop/activity-retry-simulator

Activities: Weathering System Outages (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ... generate a charge request for the item & customer

    ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
        StartToCloseTimeout:    30 * time.Second,
        ScheduleToCloseTimeout: time.Hour,
        RetryPolicy: &temporal.RetryPolicy{
            // MaximumAttempts: 0, // unlimited; bound by ScheduleToCloseTimeout
            InitialInterval:    time.Second,
            BackoffCoefficient: 2,
            MaximumInterval:    100 * time.Second,
        },
    })

    // ... execute the ChargePayment activity
}
```

Activities: Weathering System Outages (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ... generate a charge request for the item & customer

    ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
        StartToCloseTimeout:    30 * time.Second,
        ScheduleToCloseTimeout: time.Hour,
        RetryPolicy: &temporal.RetryPolicy{
            // MaximumAttempts: 0, // unlimited; bound by ScheduleToCloseTimeout
            InitialInterval:    time.Second,
            BackoffCoefficient: 2,
            MaximumInterval:    100 * time.Second,
        },
    })

    // ... execute the ChargePayment activity
}
```

Activities: Weathering System Outages (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ... generate a charge request for the item & customer

    ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
        StartToCloseTimeout:    30 * time.Second,
        ScheduleToCloseTimeout: time.Hour,
        RetryPolicy: &temporal.RetryPolicy{
            // MaximumAttempts: 0, // unlimited; bound by ScheduleToCloseTimeout
            InitialInterval:    time.Second,
            BackoffCoefficient: 2,
            MaximumInterval:    100 * time.Second,
        },
    })

    // ... execute the ChargePayment activity
}
```

Activities: Weathering System Outages (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ... generate a charge request for the item & customer

    ctx = workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
        StartToCloseTimeout:    30 * time.Second,
        ScheduleToCloseTimeout: time.Hour,
    })

    // ... execute the ChargePayment activity
}
```

Activities: Implementing Idempotency

Idempotence is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application.

wikipedia.org/wiki/Idempotence

Activities: Implementing Idempotency

Common techniques to achieve idempotency:

- Passing idempotency key to external APIs
- Derive a key from the Workflow ID and activity ID. Pass this to downstream systems (like Stripe) to ignore duplicate requests.
- Applying database constraints
- Use `INSERT ... ON CONFLICT` or conditional writes to ensure records aren't created twice.
- Using naturally idempotent operations
- Design side effects as state settings (Set to X) rather than increments (+1), or use upserts with fixed IDs.
- May help to decompose into multiple activities.

Activities: Implementing Idempotency

Common techniques to achieve idempotency:

- Passing idempotency key to external APIs
- Derive a key from the Workflow ID and activity ID. Pass this to downstream systems (like Stripe) to ignore duplicate requests.
- Applying database constraints
- Use `INSERT ... ON CONFLICT` or conditional writes to ensure records aren't created twice.
- Using naturally idempotent operations
- Design side effects as state settings (Set to X) rather than increments (+1), or use upserts with fixed IDs.
- May help to decompose into multiple activities.
- *Setting `MaxAttempts: 1` in retry policy*

Activities: Idempotency Keys

```
func ChargePayment(ctx context.Context, req ChargeRequest) (*ChargeResponse, error) {
    httpReq := newPaymentHTTPReq(req) // POST api.example.com/v1/payments/charge
    httpReq.Header.Set("Idempotency-Key", getIdempotencyToken(ctx))
    // Send request ...

    switch resp.StatusCode {
    case http.StatusOK: /* Decode the response and return ... */
    case http.StatusBadRequest: /* Return non-retryable error ... */
    case http.StatusTooManyRequests: /* Back off more aggressively ... */
    default: /* Unexpected status ... */
    }
}

func getIdempotencyToken(ctx context.Context) string {
    i := activity.GetInfo(ctx)
    key := fmt.Sprintf("%s:%s:%s", i.WorkflowExecution.ID, i.WorkflowExecution.RunID, i.ActivityID)
    return fmt.Sprintf("%x", sha256.Sum256([]byte(key)))
}
```

Activities: Idempotency Keys

```
func ChargePayment(ctx context.Context, req ChargeRequest) (*ChargeResponse, error) {
    httpReq := newPaymentHTTPReq(req) // POST api.example.com/v1/payments/charge
    httpReq.Header.Set("Idempotency-Key", getIdempotencyToken(ctx))
    // Send request ...

    switch resp.StatusCode {
    case http.StatusOK: /* Decode the response and return ... */
    case http.StatusBadRequest: /* Return non-retryable error ... */
    case http.StatusTooManyRequests: /* Back off more aggressively ... */
    default: /* Unexpected status ... */
    }
}

func getIdempotencyToken(ctx context.Context) string {
    i := activity.GetInfo(ctx)
    key := fmt.Sprintf("%s:%s:%s", i.WorkflowExecution.ID, i.WorkflowExecution.RunID, i.ActivityID)
    return fmt.Sprintf("%x", sha256.Sum256([]byte(key)))
}
```

Activities: Natural Idempotency

```
func RunKubernetesJob(ctx context.Context, req RunJobRequest) (*RunJobResponse, error) {
    jobs := k8sClient.BatchV1().Jobs(req.Namespace)
    // Create the job
    if _, err := jobs.Create(ctx, &batchv1.Job{
        Name: req.Name,
    }); err != nil {
        return nil, err
    }

    // Poll for final status
    for {
        activity.RecordHeartbeat(ctx)
        j, err := jobs.Get(ctx, req.Name)
        if err != nil { return nil, err }
        if status := getJobStatus(j); status.IsTerminal() {
            return &RunJobResponse{Status: status}, nil
        }
        time.Sleep(15 * time.Second)
    }
}
```

Activities: Natural Idempotency

```
func RunKubernetesJob(ctx context.Context, req RunJobRequest) (*RunJobResponse, error) {
    jobs := k8sClient.BatchV1().Jobs(req.Namespace)
    // Create the job
    if _, err := jobs.Create(ctx, &batchv1.Job{
        Name: req.Name,
    }); err != nil {
        return nil, err
    }

    // Poll for final status
    for {
        activity.RecordHeartbeat(ctx)
        j, err := jobs.Get(ctx, req.Name)
        if err != nil { return nil, err }
        if status := getJobStatus(j); status.IsTerminal() {
            return &RunJobResponse{Status: status}, nil
        }
        time.Sleep(15 * time.Second)
    }
}
```

Activities: Natural Idempotency (continued)

```
func RunKubernetesJob(ctx context.Context, req RunJobRequest) (*RunJobResponse, error) {
     jobs := k8sClient.BatchV1().Jobs(req.Namespace)
     // Create the job if it doesn't exist
     if _, err := jobs.Create(ctx, &batchv1.Job{
         Name: req.Name,
-    }); err != nil {
+    }); err != nil && !apierrors.IsAlreadyExists(err) {
         return nil, err
     }

     // Poll for final status
     // ...
 }
```

Activities: Natural Idempotency (continued)

```
func RunKubernetesJob(ctx context.Context, req RunJobRequest) (*RunJobResponse, error) {
     jobs := k8sClient.BatchV1().Jobs(req.Namespace)
     // Create the job if it doesn't exist
     if _, err := jobs.Create(ctx, &batchv1.Job{
         Name: req.Name,
-    }); err != nil {
+    }); err != nil && !apierrors.IsAlreadyExists(err) {
         return nil, err
     }

     // Poll for final status
     // ...
 }
```

Activities: Natural Idempotency (continued)

```
func StartKubernetesJob(ctx context.Context, req StartJobRequest) error {
    jobs := k8sClient.BatchV1().Jobs(req.Namespace)
    // Create the job if it doesn't exist
    _, err := jobs.Create(ctx, &batchv1.Job{ /* ... */ })
    if err != nil && !apierrors.IsAlreadyExists(err) {
        return err
    }
    return nil
}

func AwaitKubernetesJob(ctx context.Context, req AwaitJobRequest) (*AwaitJobResponse, error) {
    for { /* Poll for final status... */ }
}
```

Activities: Natural Idempotency (continued)

```
func StartKubernetesJob(ctx context.Context, req StartJobRequest) error {
    jobs := k8sClient.BatchV1().Jobs(req.Namespace)
    // Create the job if it doesn't exist
    _, err := jobs.Create(ctx, &batchv1.Job{ /* ... */ })
    if err != nil && !apierrors.IsAlreadyExists(err) {
        return err
    }
    return nil
}

func AwaitKubernetesJob(ctx context.Context, req AwaitJobRequest) (*AwaitJobResponse, error) {
    for { /* Poll for final status... */ }
}
```

Activities: Handling Worker Disruptions

```
func RunKubernetesJob(ctx workflow.Context, req RunJobRequest) (*RunJobResponse, error) {
    // Execute the StartKubernetesJob activity
    // ...

    // Wait for the job to complete
    return workflowhelpers.AwaitActivity(
        workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
            StartToCloseTimeout:    time.Hour,
            ScheduleToCloseTimeout: time.Hour,
        }),
        AwaitKubernetesJob,
        AwaitJobRequest{Name: req.Name, Namespace: req.Namespace},
    )
}
```

Activities: Handling Worker Disruptions

```
func RunKubernetesJob(ctx workflow.Context, req RunJobRequest) (*RunJobResponse, error) {
    // Execute the StartKubernetesJob activity
    // ...

    // Wait for the job to complete
    return workflowhelpers.AwaitActivity(
        workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
            StartToCloseTimeout:    time.Hour,
            ScheduleToCloseTimeout: time.Hour,
        }),
        AwaitKubernetesJob,
        AwaitJobRequest{Name: req.Name, Namespace: req.Namespace},
    )
}
```

Activities: Handling Worker Disruptions (continued)

```
func RunKubernetesJob(ctx workflow.Context, req RunJobRequest) (*RunJobResponse, error) {
    // Execute the StartKubernetesJob activity
    // ...

    // Wait for the job to complete
    return workflowhelpers.AwaitActivity(
        workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
            // Long-running activity: prefer Heartbeat over StartToClose timeout.
            HeartbeatTimeout:       30 * time.Second,
            ScheduleToCloseTimeout: time.Hour,
        }),
        AwaitKubernetesJob,
        AwaitJobRequest{Name: req.Name, Namespace: req.Namespace},
    )
}
```

Activities: Handling Worker Disruptions (continued)

```
func RunKubernetesJob(ctx workflow.Context, req RunJobRequest) (*RunJobResponse, error) {
    // Execute the StartKubernetesJob activity
    // ...

    // Wait for the job to complete
    return workflowhelpers.AwaitActivity(
        workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
            // Long-running activity: prefer Heartbeat over StartToClose timeout.
            HeartbeatTimeout:       30 * time.Second,
            ScheduleToCloseTimeout: time.Hour,
        }),
        AwaitKubernetesJob,
        AwaitJobRequest{Name: req.Name, Namespace: req.Namespace},
    )
}
```

Activities: Handling Worker Disruptions (continued)

```
func RunKubernetesJob(ctx workflow.Context, req RunJobRequest) (*RunJobResponse, error) {
    // Execute the StartKubernetesJob activity
    // ...

    // Wait for the job to complete
    return workflowhelpers.AwaitActivity(
        workflow.WithActivityOptions(ctx, workflow.ActivityOptions{
            // Long-running activity: prefer Heartbeat over StartToClose timeout.
            HeartbeatTimeout:       30 * time.Second,
            ScheduleToCloseTimeout: time.Hour,
        }),
        AwaitKubernetesJob,
        AwaitJobRequest{Name: req.Name, Namespace: req.Namespace},
    )
}
```

Activities: A Grand Unified Theory

1. Activities should be idempotent.

2. Always set a **ScheduleToClose** timeout. Base the value on how long the activity should continue retrying during a worst case outage.

3. Always set either **Heartbeat** or **StartToClose** timeout. Use **StartToClose** timeout only when the activity is guaranteed to not run past that duration and it is acceptable to wait the whole duration before a retry.

4. Activities that perform cleanup on cancelation MUST send heartbeats.

5. Prefer unlimited attempts with **ScheduleToClose** as the bound.

6. Respect error conventions from downstream services. Translate these into Temporal application errors to skip retry or adjust backoff behavior.

** Consider implementing and/or enforcing these policies in an interceptor.

Activities: A Grand Unified Theory

1. Activities should be idempotent.

2. Always set a **ScheduleToClose** timeout. Base the value on how long the activity should continue retrying during a worst case outage.

3. Always set either **Heartbeat** or **StartToClose** timeout. Use **StartToClose** timeout only when the activity is guaranteed to not run past that duration and it is acceptable to wait the whole duration before a retry.

4. Activities that perform cleanup on cancelation MUST send heartbeats.

5. Prefer unlimited attempts with **ScheduleToClose** as the bound.

6. Respect error conventions from downstream services. Translate these into Temporal application errors to skip retry or adjust backoff behavior.

** Consider implementing and/or enforcing these policies in an interceptor.

Part Two: Workflows

Need to be robust to:

Must also:

- Server-imposed history & payload limits
- Workflow code changing across deployments
- Signals arriving in unpredictable order
- Cancelation requests at any point in execution
- Stay deterministic across replays
- Yield quickly to the workflow task event loop

Workflows: Living Within Server Limits

Server-imposed limits to be aware of:

- **Individual payload size**: ~2MB per workflow/activity input or output, signal, or update.
- **Workflow history bytes**: 50MB (<10MB recommended). Results in termination.
- **Workflow history length**: 50k events (<10k recommended). Results in termination.
- **Workflow task timeout**: 10s default (120s max configurable). Results in failed workflow task (retried).
- **Workflow lock contention**: No hard limit, but aim for no more than ~1 workflow state change per second.

Mitigations:

- Use **ContinueAsNew** to start a fresh execution with reset history. Check `GetContinueAsNewSuggested()` to know when the server is recommending it.
- Avoid passing large payloads from activities to workflows, or use external payload storage.

Workflows: Living Within Server Limits

Server-imposed limits to be aware of:

- **Individual payload size**: ~2MB per workflow/activity input or output, signal, or update.
- **Workflow history bytes**: 50MB (<10MB recommended). Results in termination.
- **Workflow history length**: 50k events (<10k recommended). Results in termination.
- **Workflow task timeout**: 10s default (120s max configurable). Results in failed workflow task (retried).
- **Workflow lock contention**: No hard limit, but aim for no more than ~1 workflow state change per second.

Mitigations:

- Use **ContinueAsNew** to start a fresh execution with reset history. Check `GetContinueAsNewSuggested()` to know when the server is recommending it.
- Avoid passing large payloads from activities to workflows, or use external payload storage.

Workflows: Living Within Server Limits

Server-imposed limits to be aware of:

- **Individual payload size**: ~2MB per workflow/activity input or output, signal, or update.
- **Workflow history bytes**: 50MB (<10MB recommended). Results in termination.
- **Workflow history length**: 50k events (<10k recommended). Results in termination.
- **Workflow task timeout**: 10s default (120s max configurable). Results in failed workflow task (retried).
- **Workflow lock contention**: No hard limit, but aim for no more than ~1 workflow state change per second.

Mitigations:

- Use **ContinueAsNew** to start a fresh execution with reset history. Check `GetContinueAsNewSuggested()` to know when the server is recommending it.
- Avoid passing large payloads from activities to workflows, or use external payload storage.

Workflows: Keeping Code Deterministic

Common sources of non-determinism in workflow code:

- Network calls (HTTP, DB queries, gRPC) -- re-execute on every replay and may return different results
- System time (`time.Now()` vs `workflow.Now()`)
- Random number generation
- Usage of environment variables
- Interactions with the filesystem
- Goroutines spawned outside `workflow.Go` -- the SDK doesn't record their scheduling, and they often race with the workflow function for shared state
- Variable references from outside the workflow function scope

Workflows: Keeping Code Deterministic (continued)

Tools to catch non-determinism:

- **Go**: `workflowcheck` -- opt-in static analyzer that flags non-deterministic calls in workflow code.
- **Python**: Workflow sandbox -- enabled by default; restricts imports and module access at runtime.
- **TypeScript**: V8 isolate sandboxing is built-in -- workflow code runs in a separate V8 context with no Node.js APIs.

Workflows: Versioning Code Changes

Change versioning is used to gate new behavior in workflow functions in order to maintain backwards compatability with workflows started on earlier versions of the worker.

Change version lifecycle:

1. Add new behavior, gated with version check
2. Wait for workflows started on previous version of the worker to complete
3. Remove the old behavior and the version check

Workflows: Versioning Code Changes

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
     // ...

     sel.AddFuture(workflow.NewTimer(ctx, 12*time.Hour), func(f workflow.Future) {
         err = workflow.ErrDeadlineExceeded
     })

     sel.Select(ctx)
     if err != nil {
+        delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 1)
+        switch delayVersion {
+        case 1:
+            workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
+        }
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }
```

Workflows: Versioning Code Changes

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
     // ...

     sel.AddFuture(workflow.NewTimer(ctx, 12*time.Hour), func(f workflow.Future) {
         err = workflow.ErrDeadlineExceeded
     })

     sel.Select(ctx)
     if err != nil {
+        delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 1)
+        switch delayVersion {
+        case 1:
+            workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
+        }
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }
```

Workflows: Versioning Code Changes

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
     // ...

     sel.AddFuture(workflow.NewTimer(ctx, 12*time.Hour), func(f workflow.Future) {
         err = workflow.ErrDeadlineExceeded
     })

     sel.Select(ctx)
     if err != nil {
+        delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 1)
+        switch delayVersion {
+        case 1:
+            workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
+        }
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }
```

Workflows: Evaluate Change Versions Up Front

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
+    delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 1)
     // ...

     sel.AddFuture(workflow.NewTimer(ctx, 12*time.Hour), func(f workflow.Future) {
         err = workflow.ErrDeadlineExceeded
     })

     sel.Select(ctx)
     if err != nil {
+        switch delayVersion {
+        case 1:
+            workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
+        }
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }
```

Workflows: Evaluate Change Versions Up Front (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
-    delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 1)
+    delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 2)
     // ...

     sel.Select(ctx)
     if err != nil {
         switch delayVersion {
         case 1:
             workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
+        case 2:
+            // Cancel the in-flight shipment before issuing the refund.
+            if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
+                log.Warn("failed to cancel shipment", "error", e)
+            }
+            workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
         }
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }
```

Workflows: Evaluate Change Versions Up Front (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
-    delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 1)
+    delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 2)
     // ...

     sel.Select(ctx)
     if err != nil {
         switch delayVersion {
         case 1:
             workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
+        case 2:
+            // Cancel the in-flight shipment before issuing the refund.
+            if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
+                log.Warn("failed to cancel shipment", "error", e)
+            }
+            workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
         }
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }
```

Workflows: Verifying Replay Safety

```
# Find the earliest workflow that did not hit either patch branch
LIST_QUERY="WorkflowType='PurchaseItem'"
LIST_QUERY+=" AND TemporalChangeVersion NOT IN ('handle-shipment-delay-1', 'handle-shipment-delay-2')"
temporal workflow list --query "$LIST_QUERY" --order-by 'StartTime ASC' --limit 1

# Find the earliest workflow that took version 1 of the patch.
LIST_QUERY="WorkflowType='PurchaseItem'"
LIST_QUERY+=" AND TemporalChangeVersion IN ('handle-shipment-delay-1')"
temporal workflow list --query "$LIST_QUERY" --order-by 'StartTime ASC' --limit 1

# Download workflow history to a test fixture path.
temporal workflow show --workflow-id <ID> --output json \
  > testdata/purchase_item_history_<PATCH_VERSION>.json
```

Workflows: Verifying Replay Safety

```
# Find the earliest workflow that did not hit either patch branch
LIST_QUERY="WorkflowType='PurchaseItem'"
LIST_QUERY+=" AND TemporalChangeVersion NOT IN ('handle-shipment-delay-1', 'handle-shipment-delay-2')"
temporal workflow list --query "$LIST_QUERY" --order-by 'StartTime ASC' --limit 1

# Find the earliest workflow that took version 1 of the patch.
LIST_QUERY="WorkflowType='PurchaseItem'"
LIST_QUERY+=" AND TemporalChangeVersion IN ('handle-shipment-delay-1')"
temporal workflow list --query "$LIST_QUERY" --order-by 'StartTime ASC' --limit 1

# Download workflow history to a test fixture path.
temporal workflow show --workflow-id <ID> --output json \
  > testdata/purchase_item_history_<PATCH_VERSION>.json
```

Workflows: Verifying Replay Safety (continued)

Check the code coverage for the version branches in your workflow -- if they aren't covered, the replay test is not validating your change.

Find more techniques at temporal.io/resources/on-demand/replay-safety-at-datadog

```
func TestReplayWorkflowHistory(t *testing.T) {
    testhelpers.AssertWorkflowReplayFromJSONFiles(t,
        PurchaseItem,
        "testdata/purchase_item_history_v0.json",
        "testdata/purchase_item_history_v1.json",
    )
}
```

Workflows: Verifying Replay Safety (continued)

Check the code coverage for the version branches in your workflow -- if they aren't covered, the replay test is not validating your change.

Find more techniques at temporal.io/resources/on-demand/replay-safety-at-datadog

```
func TestReplayWorkflowHistory(t *testing.T) {
    testhelpers.AssertWorkflowReplayFromJSONFiles(t,
        PurchaseItem,
        "testdata/purchase_item_history_v0.json",
        "testdata/purchase_item_history_v1.json",
    )
}
```

Workflows: Cleaning Up Change Versions

```
#!/bin/sh
# Prints 0 when no in-flight workflows are still on v0 or v1.

# Timestamp for 5 minutes ago, to avoid counting workflows that have been
# scheduled but haven't yet been picked up by a worker. (use `date` on Linux)
CUTOFF=$(gdate -u -d '5 minutes ago' +%Y-%m-%dT%H:%M:%S.%3NZ)

# Count workflows still running on a handle-shipment-delay version < 2.
QUERY="WorkflowType='PurchaseItem'"
QUERY+=" AND ExecutionStatus='Running'"
QUERY+=" AND TemporalChangeVersion NOT IN ('handle-shipment-delay-2')"
QUERY+=" AND StartTime < '$CUTOFF'"

temporal workflow count --query "$QUERY"
```

Workflows: Cleaning Up Change Versions

```
#!/bin/sh
# Prints 0 when no in-flight workflows are still on v0 or v1.

# Timestamp for 5 minutes ago, to avoid counting workflows that have been
# scheduled but haven't yet been picked up by a worker. (use `date` on Linux)
CUTOFF=$(gdate -u -d '5 minutes ago' +%Y-%m-%dT%H:%M:%S.%3NZ)

# Count workflows still running on a handle-shipment-delay version < 2.
QUERY="WorkflowType='PurchaseItem'"
QUERY+=" AND ExecutionStatus='Running'"
QUERY+=" AND TemporalChangeVersion NOT IN ('handle-shipment-delay-2')"
QUERY+=" AND StartTime < '$CUTOFF'"

temporal workflow count --query "$QUERY"
```

Workflows: Cleaning Up Change Versions (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
-    delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 2)
+    // TODO: drop this once the new worker is rolled out everywhere.
+    _ = workflow.GetVersion(ctx, "handle-shipment-delay", 2, 2)
     // ...

     sel.Select(ctx)
     if err != nil {
-        switch delayVersion {
-        case 1: /* Refund only ... */
-        case 2:
         if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
             log.Warn("failed to cancel shipment", "error", e)
         }
         workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }
```

Workflows: Cleaning Up Change Versions (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
-    delayVersion := workflow.GetVersion(ctx, "handle-shipment-delay", workflow.DefaultVersion, 2)
+    // TODO: drop this once the new worker is rolled out everywhere.
+    _ = workflow.GetVersion(ctx, "handle-shipment-delay", 2, 2)
     // ...

     sel.Select(ctx)
     if err != nil {
-        switch delayVersion {
-        case 1: /* Refund only ... */
-        case 2:
         if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
             log.Warn("failed to cancel shipment", "error", e)
         }
         workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }
```

Workflows: Disconnected Child Workflows

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ...
    sel.Select(ctx)
    if err != nil {
        if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
            log.Warn("failed to cancel shipment", "error", e)
        }
        workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
        return nil, err
    }
    return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
}
```

Workflows: Disconnected Child Workflows

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ...
    sel.Select(ctx)
    if err != nil {
        if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
            log.Warn("failed to cancel shipment", "error", e)
        }
        workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
        return nil, err
    }
    return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
}
```

Workflows: Disconnected Child Workflows

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ...
    sel.Select(ctx)
    if err != nil {
        if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
            log.Warn("failed to cancel shipment", "error", e)
        }
        workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
        return nil, err
    }
    return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
}
```

Workflows: Disconnected Child Workflows

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ...
    sel.Select(ctx)
    if err != nil {
        if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
            log.Warn("failed to cancel shipment", "error", e)
        }
        workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
        return nil, err
    }
    return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
}
```

Workflows: Disconnected Child Workflows

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
    // ...
    sel.Select(ctx)
    if err != nil {
        if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
            log.Warn("failed to cancel shipment", "error", e)
        }
        workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
        return nil, err
    }
    return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
}
```

Workflows: Disconnected Child Workflows (continued)

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
     // ...
     sel.Select(ctx)
     if err != nil {
         if e := workflowhelpers.AwaitActivity(ctx, CancelShipment, cancelRequest); e != nil {
             log.Warn("failed to cancel shipment", "error", e)
         }
-        workflow.ExecuteChildWorkflow(ctx, RefundPayment, refundRequest)
+        execDisconnectedChildWorkflow(ctx, RefundPayment, refundRequest)
         return nil, err
     }
     return &PurchaseResponse{TrackingID: shipmentResponse.TrackingID}, nil
 }

 // Start a child workflow using disconnected context and wait for it to be scheduled.
 func execDisconnectedChildWorkflow(ctx workflow.Context, childWorkflow any, args ...any) {
     ctx = workflow.WithParentClosePolicy(ctx, enums.PARENT_CLOSE_POLICY_ABANDON)
     ctx, _ = workflow.NewDisconnectedContext(ctx)
     fut := workflow.ExecuteChildWorkflow(ctx, childWorkflow, args...)
     if err := fut.GetChildWorkflowExecution().Get(ctx, nil); err != nil { panic(err) }
 }
```

Workflows: Use Timers, Not Timeouts

```
func PurchaseItem(ctx workflow.Context, req PurchaseRequest) (*PurchaseResponse, error) {
     // ...

+    // Reserve at least 1 minute for compensation before the hard timeout.
+    sel.AddFuture(getSoftTimeout(ctx, time.Minute), func(f workflow.Future) {
+        // Run compensating actions now! Workflow terminating in 1 minute...
+    })

     // ...
 }

 func getSoftTimeout(ctx workflow.Context, padding time.Duration) workflow.Future {
     timeout := workflow.GetInfo(ctx).WorkflowRunTimeout
     if timeout <= padding {
         panic("WorkflowRunTimeout too short")
     }
     return workflow.NewTimer(ctx, timeout - padding)
 }
```

Workflows: Cap Workflow Lifetime With ContinueAsNew

```
func SubscriptionWorkflow(ctx workflow.Context, state SubscriptionState) error {
     // ...
+    sel.AddFuture(workflow.NewTimer(ctx, 24*time.Hour), func(f workflow.Future) {
+        // Noop; just unblock the selector
+    })

     for sel.HasPending() {
         sel.Select(ctx)
+        if continueAsNewSuggested(ctx) {
+            return workflow.NewContinueAsNewError(ctx, SubscriptionWorkflow, state)
+        }
         // ...
     }
 }

 func continueAsNewSuggested(ctx workflow.Context) bool {
     info := workflow.GetInfo(ctx)
     return info.GetContinueAsNewSuggested() ||
         workflow.Now(ctx).After(info.WorkflowStartTime + 24*time.Hour - time.Second)
 }
```

Workflows: Design Guidelines

Workflow functions:

- MUST be deterministic. Use the Temporal SDK for time, randomness, and side effects.
- MUST evaluate all change versions as the first step (at the top of the function).
- SHOULD use internal timers rather than execution timeouts if they need to run compensating actions.
- SHOULD be designed to complete or ContinueAsNew within 24 hours or when the server suggests ContinueAsNew.
- SHOULD drain all signals before completing or ContinueAsNew.
- SHOULD fan out large batches of work to child workflows.

Temporal workers:

- MUST be configured with a BuildID
- SHOULD have a replay testing harness.
- SHOULD be onboarded to worker versioning and pinned workflows.
- SHOULD enable external payload storage if activity responses are ~1MB or larger.

Acknowledgements

- 100 Go Mistakes by Teiva Harsanyi
- Loïc Minaudier (software engineer @Datadog)
- Participants in the "Temporal Pain Points" birds of a feather (Replay 2025)