TECHNICAL GUIDE

Building Reliable Applications with Durable Execution

AUTHOR
Loren Sands-Ramshaw - @lorendsr
Developer Relations Engineer, Temporal

This guide introduces the concept of **durable execution**, which is used by organizations like Stripe, Netflix, HashiCorp, Datadog, and many others to solve a wide range of problems in distributed systems. It explains how some of those problems are solved, shows the new programming possibilities that durable execution opens up, and includes an example application that demonstrates how simple it is to write durable code.

Understanding distributed systems

For developers working on a traditional request-response monolith backed by a single database that supports transactions, there tend to be little distributed system challenges. Failure modes tend to be comparatively simple and straightforward to resolve:

- If the client can't reach the server, the client retries.
- If the client reaches the server, but the server can't reach the database, the server responds with an error, and the client retries.
- If the server reaches the database, but the transaction fails, the server responds with an error, and the client retries.
- If the transaction succeeds but the server or network goes down before responding to the client, the client retries until the server is back up, and the transaction fails the second time (assuming the transaction has some check–like an idempotency token–to tell whether the update has already been applied), and the server reports to the client that the action has already been performed.

Many of these issues can be solved simply by retrying, and state can be accurately maintained without heroic efforts by developers.

However, as soon as we introduce a second place where state is stored and maintained—whether that's a service with its own database or an external API—handling failures and maintaining consistency (accuracy across all data stores) gets significantly more complex. For example, if our server has to charge a credit card and also update the database, we can no longer write simple code like:

```
function handleRequest() {
  paymentAPI.chargeCard()
  database.insertOrder()
  return 200
}
```

If the first step (charging the card) succeeds, but the second step (adding the order to the database) fails, then the system ends up in an inconsistent state; the customer was charged for a purchase, but there's no record of the purchase in the database. To try to maintain consistency (and avoid angry customers), it might make sense to automatically retry the second step until the order has been successfully written to the database.

This is a perfectly valid way to work around the problem (assuming `insertOrder()` is idempotent), if indeed the database is simply experiencing a brief hiccup. But the negative impacts of a less-than-ideal outcome are already becoming clear; if the process running this code fails before it updates the database, we'll again end up in an inconsistent state.

To address this risk, the application now needs to do three things:

- Persist the order details
- Persist which steps of the program we've completed
- Run a worker process that checks the database or a task queue for incomplete orders and continues with the next step

Now imagine that the application in question does something more complicated after these first two steps, like updating inventory records, generating a shipping label, or assigning a delivery driver. That, along with persisting retry state and adding timeouts for each step, is a lot of code to write, and it's easy to miss certain edge cases or failure modes (see the full, scalable architecture) along the way. All of which is to say, developers could build more reliable applications in less time if we didn't have to write and debug all of that code to handle failures. Fortunately, durable execution was designed with this exact situation in mind.

Durable execution

Durable execution guarantees that the code is executed to completion, no matter how (un)reliable the hardware the code is running on, or how often the network goes down, or how long downstream services are down. Retries and timeouts are performed automatically and transparently, and resources are freed up when nothing is happening (for example, while the code is waiting for a downstream service to come back up).

This is possible because durable execution systems like Temporal persist each step your code takes. If the process or container running the code dies, the code automatically continues running in another process with all state intact, including call stack and local variables.

All the complex plumbing logic that developers had to implement in our normal execution world—error handling, retry logic, saving state for each step, polling task queues—durable execution handles automatically[1].

There are a number of systems that provide durable execution, including Azure Durable Functions, AWS Simple Workflow Service, and Uber Cadence, a project which was open-sourced by Maxim Fateev and Samar Abbas, who went on to found Temporal in 2019. Temporal is also open source (MIT license) with a team of engineers working on the software full-time. If you've recently posted a Snapchat story, booked a stay through AirBnB, or ordered from Taco Bell online, you've already experienced a Temporal workflow.

Interested in how this all works? Check out this blog post: How Durable Execution Works

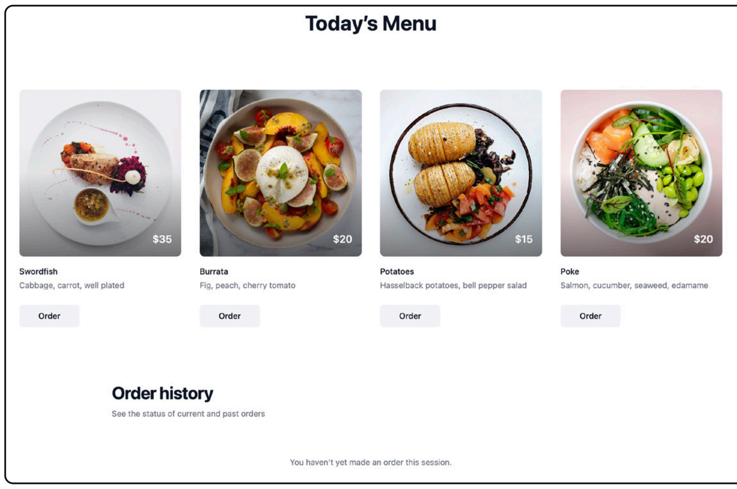
1 Of course, as a developer you can still exercise control over things like retry behavior—the point here is that there is some default retry logic at all, rather than none.

Writing durable code

Now that we've gone over consistency in distributed systems and what durable execution is, let's look at a practical example. To better illustrate what durable code looks like and what problems it solves, I built an example food delivery app, which you can find at temporal.menu.

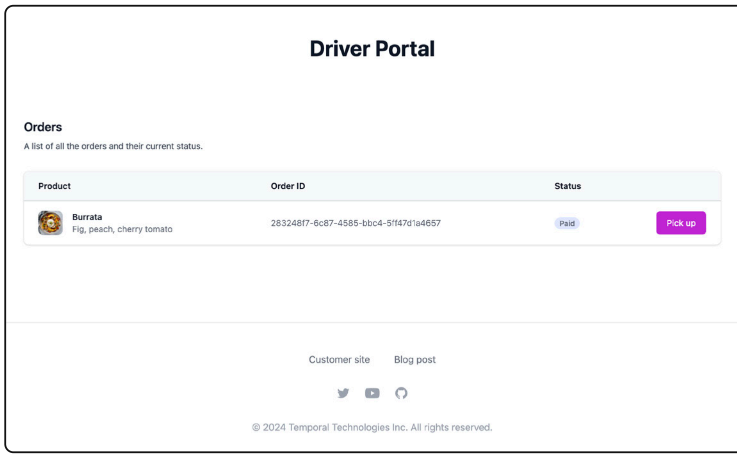
The sample app has four main pieces of functionality:

- Create an order and charge the customer
- Get order status
- Mark an order picked up
- Mark an order delivered


Today's Menu
$35
$20
$15
$20
Swordfish
Cabbage, carrot, well plated
Burrata
Fig, peach, cherry tomato
Potatoes
Hasselback potatoes, bell pepper salad
Poke
Salmon, cucumber, seaweed, edamame
Order
Order
Order
Order
Order history
See the status of current and past orders
You haven't yet made an order this session.

When we order an item from the menu, it appears in the delivery driver site (drive.temporal.menu), and the driver can mark the order as picked up, and then as delivered.

All of this functionality is implemented in a single function of durable TypeScript, but Temporal also has runtimes for Go, Java, Python, .NET, and PHP.


Driver Portal

Orders

A list of all the orders and their current status.

Product	Order ID	Status	
Burrata Fig, peach, cherry tomato	283248f7-6c87-4585-bbc4-5ff47d1a4657	Paid	Pick up

Create an order

Let’s take a look at the code for this app. We’ll see a few API routes but mostly go over each piece of the single durable function named `order`. If you’d like to run the app or view the code on your machine, this will download and set up the project (you need to install Node first):

```
npx @temporalio/create@latest --sample food-delivery
```

When the user clicks the order button, the React frontend calls the `createOrder` mutation defined by the tRPC backend (which sends a POST to the API server). The `createOrder` API route handler creates the order by starting a durable `order` function:

apps/menu/pages/api/[trpc].ts

You can also read through the code on GitHub.

```
import { initTRPC } from ‘@trpc/server’
import { z } from ‘zod’
import { taskQueue } from ‘common’
import { Context } from ‘common/trpc-context’
import { order } from ‘workflows’

const t = initTRPC.context<Context>().create()

export const appRouter = t.router({
  createOrder: t.procedure
    .input(z.object({ productId: z.number(), orderId: z.string() }))
    .mutation(async ({ input, ctx }) => {
      await ctx.temporal.workflow.start(order, {
        workflowId: input.orderId,
        args: [input.productId],
        taskQueue,
      })

      return ‘Order received and persisted!’
    }),
```

Durable functions—called **Workflows** in Temporal—are started using a Client instance from `@temporalio/client`, which has been added to the tRPC context under `ctx.temporal`. The route handler receives a validated `input` (an object with a `productId` number and `orderId` string) and it calls `ctx.temporal.workflow.start` to start an `order` Workflow, providing `input.productId` as an argument.

The `order` function starts out validating the input, setting up the initial state, and charging the customer:

packages/workflows/order.ts

```
type OrderState = 'Charging card' | 'Paid' | 'Picked up' | 'Delivered' | 'Refunding'

export async function order(productId: number): Promise<void> {
  const product = getProductById(productId)
  if (!product) {
    throw ApplicationFailure.create({ message: `Product ${productId} not found` })
  }

  let state: OrderState = 'Charging card'

  try {
    await chargeCustomer(product)
  } catch (err) {
    const message = `Failed to charge customer for ${product.name}. Error: ${errorMessage(err)}`
    await sendPushNotification(message)
    throw ApplicationFailure.create({ message })
  }

  state = 'Paid'
```

Any functions that might fail are automatically retried. In this case, `chargeCustomer` and `sendPushNotification` both talk to services that might be down at the moment or might return transient error messages like "Temporarily unavailable." Temporal will automatically retry running these functions (by default indefinitely with exponential backoff, but that's configurable). The functions can also throw non-retryable errors like "Card declined," in which case they won't be retried. Instead, the error will be thrown out of `chargeCustomer(product)` and caught by the catch block; the customer receives a notification that their payment method failed, and we throw an `ApplicationFailure` to fail the `order` Workflow. Sending the customer the failure notification is called a failure compensation, and reliably executing compensation logic is a valuable aspect of durable execution. It makes sagas (long-running transactions in which failures are handled by undoing/compensating for previous steps) easy to implement. See more failure compensations below in `refundAndNotify()`.

Get order status

The next bit of code requires some background: Normal functions can't run for a long time, because they'll take up resources while they're waiting for things to happen, and at some point they'll terminate when we deploy new code and the old containers get shut down. Durable functions can run for an arbitrary length of time for two reasons:

- They don't take up resources when they're waiting on something.
- It doesn't matter if the process running them gets shut down—execution will seamlessly be continued by another process.

So although some durable functions run for a short period of time—like a money transfer function—some run longer—like our order function, which ends when the order is delivered, or a customer loyalty program function that lasts for the lifetime of the customer.

It's useful to be able to interact with long-running functions, so Temporal provides what we call **Signals** for sending data into the function and **Queries** for getting data out of the function. The driver site shows the status of each order by sending Queries to the order functions through this API route:

apps/menu/pages/api/[trpc].ts

```
  getOrderStatus: t.procedure
    .input(z.string())
    .query(({ input: orderId, ctx }) => ctx.temporal.workflow.getHandle(orderId).
query(getStatusQuery)),
```

It gets a handle to the specific instance of the order function (called a **Workflow Execution**), sends the `getStatusQuery`, and returns the result. The `getStatusQuery` is defined in the order file and handled in the order function:

packages/workflows/order.ts

```
import { defineQuery, setHandler } from '@temporalio/workflow'

export const getStatusQuery = defineQuery<OrderStatus>('getStatus')

export async function order(productId: number): Promise<void> {
  let state: OrderState = 'Charging card'
  let deliveredAt: Date

  // …

  setHandler(getStatusQuery, () => {
    return { state, deliveredAt, productId }
  })
```

When the order function receives the `getStatusQuery`, the function passed to `setHandler` is called, which returns the values of local variables. After the call to `chargeCustomer` succeeds, the state is changed to `'Paid'`, and the driver site, which has been polling `getStatusQuery`, gets the updated state. It displays the "Pick up" button.

Picking up an order

When the driver taps the button to mark the order as picked up, the site sends a `pickUp` mutation to the API server, which sends a `pickedUpSignal` to the order function:

apps/driver/pages/api/[trpc].ts

```
  pickUp: t.procedure
    .input(z.string())
    .mutation(async ({ input: orderId, ctx }) =>
      ctx.temporal.workflow.getHandle(orderId).signal(pickedUpSignal)
    ),
```

The order function handles the Signal by updating the state:

packages/workflows/order.ts

```
export const pickedUpSignal = defineSignal('pickedUp')

export async function order(productId: number): Promise<void> {
  // …

  setHandler(pickedUpSignal, () => {
    if (state === 'Paid') {
      state = 'Picked up'
    }
  })
```

Meanwhile, further down in the function, after the customer was charged, the function has been waiting for the pickup to happen:

packages/workflows/order.ts

```
import { condition } from '@temporalio/workflow'

export async function order(productId: number): Promise<void> {
  // …

  try {
    await chargeCustomer(product)
  } catch (err) {
    // …
  }

  state = 'Paid'

  const notPickedUpInTime = !(await condition(() => state === 'Picked up', '1 min'))
  if (notPickedUpInTime) {
    state = 'Refunding'
    await refundAndNotify(
      product,
      '⚠️ No drivers were available to pick up your order. Your payment has been refunded.'
    )
    throw ApplicationFailure.create({ message: 'Not picked up in time' })
  }

  …
}

async function refundAndNotify(product: Product, message: string) {
  await refundOrder(product)
  await sendPushNotification(message)
}
```

`await condition(() => state === 'Picked up', '1 min')` waits for up to 1 minute for the state to change to `Picked up`. If a minute goes by without it changing, it returns false, and we refund the customer. (Either we have very high standards for the speed of our chefs and delivery drivers, or we want the users of a demo app to be able to see all the failure modes 😁.)

Delivery

Similarly, there's a `deliveredSignal` sent by the "Deliver" button, and if the driver doesn't complete delivery within a minute of pickup, the customer is refunded.

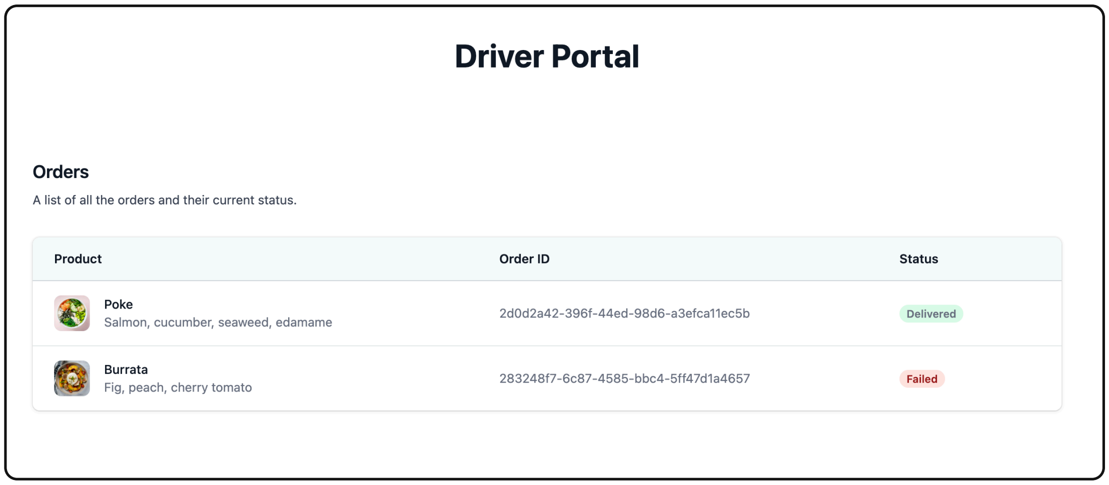
packages/workflows/order.ts

```
export const deliveredSignal = defineSignal('delivered')

export async function order(productId: number): Promise<void> {
  setHandler(deliveredSignal, () => {
    if (state === 'Picked up') {
      state = 'Delivered'
      deliveredAt = new Date()
    }
  })

  // …

  await sendPushNotification('🚗 Order picked up')

  const notDeliveredInTime = !(await condition(() => state === 'Delivered', '1 min'))
  if (notDeliveredInTime) {
    state = 'Refunding'
    await refundAndNotify(product, '⚠️ Your driver was unable to deliver your order. Your payment 
has been refunded.')
    throw ApplicationFailure.create({ message: 'Not delivered in time' })
  }

  await sendPushNotification('✅ Order delivered!')
```

If delivery was successful, the function waits for a minute for the customer to eat their meal and asks them to rate their experience.

```
  await sleep('1 min') // this could also be hours or even months

  await sendPushNotification(`✍️ Rate your meal. How was the ${product.name.toLowerCase()}?`)
}
```

After the final push notification, the order function's execution ends, and the Workflow Execution completes successfully. Even though the function has completed, we can still send Queries, since Temporal has the final state of the function saved. And we can test that by refreshing the page a minute after an order has been delivered: the `getStatusQuery` still works and "Delivered" is shown as the status:



Driver Portal

Orders

A list of all the orders and their current status.

Product	Order ID	Status
Poke Salmon, cucumber, seaweed, edamame	2d0d2a42-396f-44ed-98d6-a3efca11ec5b	Delivered
Burrata Fig, peach, cherry tomato	283248f7-6c87-4585-bbc4-5ff47d1a4657	Failed

New possibilities

Durable execution is programming on a higher level of abstraction, where you don't have to be concerned about transient faults in your infrastructure or dependencies. It opens up new possibilities like:

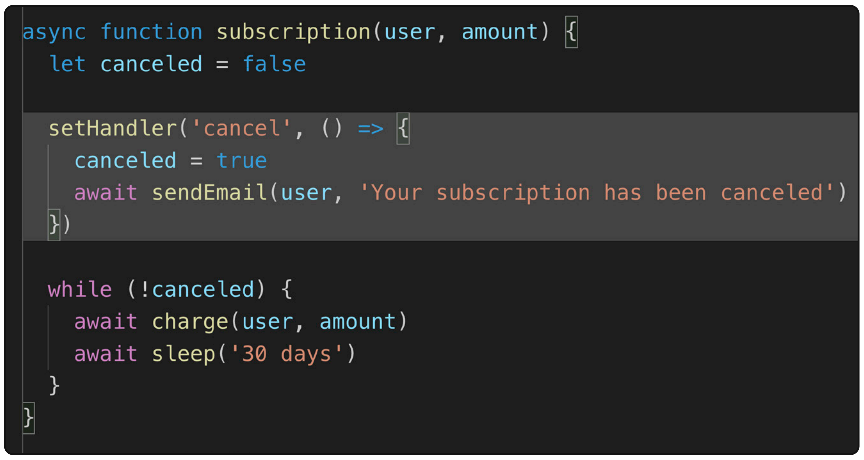
- **Writing code that sleeps for a month.** You can realistically instruct a function to sleep for any arbitrary length of time, be it weeks, months, or years. Thanks to durable execution, we don't need to be concerned about whether or not the process can safely be expected to run for that period of time—we can be confident that another process will continue running the function at the specified time. For example, a subscription function can charge the user's credit card every month in a loop:

```
async function subscription(user, amount) {
  let canceled = false

  setHandler('cancel', () => {
    canceled = true
    await sendEmail(user, 'Your subscription has been canceled'
  })

  while (!canceled) {
    await charge(user, amount)
    await sleep('30 days')
  }
}
```

- **Functions can receive RPCs.** Since the functions are potentially long-running, we may want to fetch their state, or tell them to do something different—like get how many times the user has been charged, or cancel the subscription:

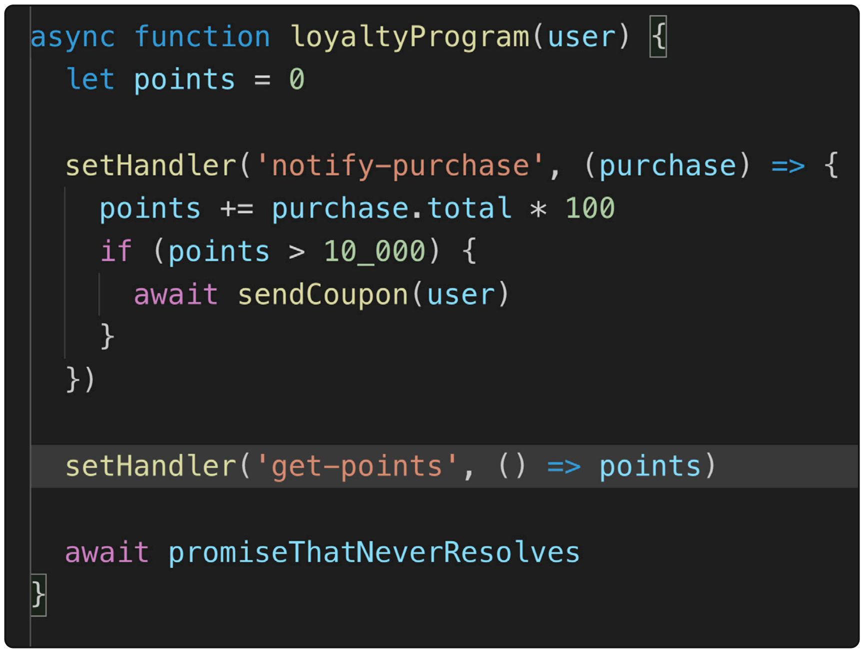
```
async function subscription(user, amount) {
  let canceled = false

  setHandler('cancel', () => {
    canceled = true
    await sendEmail(user, 'Your subscription has been canceled')
  })

  while (!canceled) {
    await charge(user, amount)
    await sleep('30 days')
  }
}
```

- **Functions can run forever.** For example, a loyalty program function can be started when a user signs up, receive an RPC whenever the user makes a purchase, and do something when the user hits the next reward level:



```
async function subscription(user, amount) {
  let canceled = false

  setHandler('cancel', () => {
    canceled = true
    await sendEmail(user, 'Your subscription has been canceled')
  })

  while (!canceled) {
    await charge(user, amount)
    await sleep('30 days')
  }
}
```

- **Store state in local variables instead of a database.** Since a function can run forever, and we can trust that a local variable will always be there and be accurate, we can send an RPC to get the variable's value instead of storing it in a DB:



```
async function loyaltyProgram(user) {
  let points = 0

  setHandler('notify-purchase', (purchase) => {
    points += purchase.total * 100
    if (points > 10_000) {
      await sendCoupon(user)
    }
  })

  setHandler('get-points', () => points)

  await promiseThatNeverResolves
}
```

Distributed systems, simplified

Durable execution makes it trivial or unnecessary to implement many distributed systems patterns, including event-driven architecture, task queues, saga patterns, cron jobs, state machines, circuit breakers, and transactional outboxes. For a more in-depth explanation of each of these, you can watch System Design on Easy Mode, which I presented at the All Things Open conference in Raleigh, NC.

Event-driven architecture

Using a message bus to communicate between services is great for loose coupling at runtime, but it's tightly coupled at design time. Making a breaking change to a message sent to a bus means finding all the other teams that depend on that message and getting them to deploy an update to their code before you can deploy your change. Durable execution is also loosely coupled at runtime, but it's a much better developer experience when building and evolving systems. For more on this topic, see the Replay conference keynote: The way forward for event-driven architecture.

Task queues

Anything that we would normally use task queues for can instead be accomplished with durable execution. Under the hood, every durable function–and every step the function takes–is put on a task queue and distributed across a pool of workers. All we need to do is provide our code to Temporal's worker library and ensure we're running enough worker processes to get through all the work in the desired time frame.

Sagas

Sagas are long-running transactions that don't hold locks; instead, each step is executed sequentially, and if a step fails, previous steps are undone with compensating steps. This is a common pattern when we need to alter state that's stored across multiple data stores, and it requires either choreography (event-based) or orchestration (central coordinator) to be accurately implemented. The Microservices Patterns book recommends using orchestration for non-trivial use cases (and I'd argue for all use cases) due to the complexity of choreography (see **event-driven architecture** above), and executing steps with durable execution is developer-friendly, automatic orchestration—it orchestrates each step our code takes.

Conclusion

We've seen how a multi-step order flow can be implemented with a single durable function. The function is guaranteed to complete in the presence of failures, including:

- Temporary issues with the network, downstream services, or third-party APIs
- The process running the function failing

This addressed a number of distributed systems concerns for us, and meant that:

- It was possible to use and rely on local variables instead of saving state to a database.
- We didn't need to set timers in a database for application logic like canceling an order that takes too long, or waiting for the rate-your-meal push notification.
- There was no need to write code to handle retrying and timing out all the functions we called that make network requests: `chargeCustomer`, `refundOrder`, and `sendPushNotification`.
- We didn't need to write polling logic for workers to notice when it's time to cancel the order, or retry a failed function.
- We didn't need to implement a state machine and have workers continue executing the next step in a multi-step process like `refundAndNotify()`.

And with durable execution, the example application also benefits from:

- **Increased reliability:** The more code we write, and the more complex the code is, the more bugs we have. With durable execution, we write less code and simpler code, which means fewer bugs. So much of the complexity is taken care of by the durable execution system, which has been tested and run for years by hundreds of companies at high scale under many failure scenarios.
- **Increased development velocity:** The logic of our order system is much easier to read, understand, and alter, since it's all contained in this single function. If our logic was spread out across various API endpoints, database entries, and workers, it would take our developers longer to onboard, develop, and debug. We often hear from companies migrating to Temporal that their engineers were afraid to touch the old system, as it was so complex that it was hard to know if a change would cause something to go wrong.
- **Increased observability:** Since every step our durable functions take is persisted, we can view the current state of every production function execution. Temporal provides a UI for viewing and searching through executions, where you can see what arguments a function was called with, when it was started, which step it's on, and the log of everything that the function has done so far. If the payment service is currently down, we'll see that the order is stuck at the `chargeCustomer` step and how many times the function has been retried, and when the next retry is scheduled for.
- **Increased debuggability:** In addition to the better visibility into our production system, we can also debug our production functions! We can download an execution's event log and use it to replay the function on our local machine, so that we can open the execution in a debugger and watch what happened. (For more info, see Time-Travel Debugging Production Code.) We can also fast-forward through time, so when your code calls `sleep('1 month')`, it resolves immediately.

I hope you're getting a sense of how much durable execution simplifies writing software. To learn more, I recommend these resources:

- Video: Getting to know Temporal
- Video: The way forward for event-driven architecture
- Course: Temporal 101
- Community Slack
- Part 2: How Durable Execution Works

Temporal

JOIN OUR COMMUNITY SLACK

EXPLORE PROJECT-BASED TUTORIALS

TS php GO