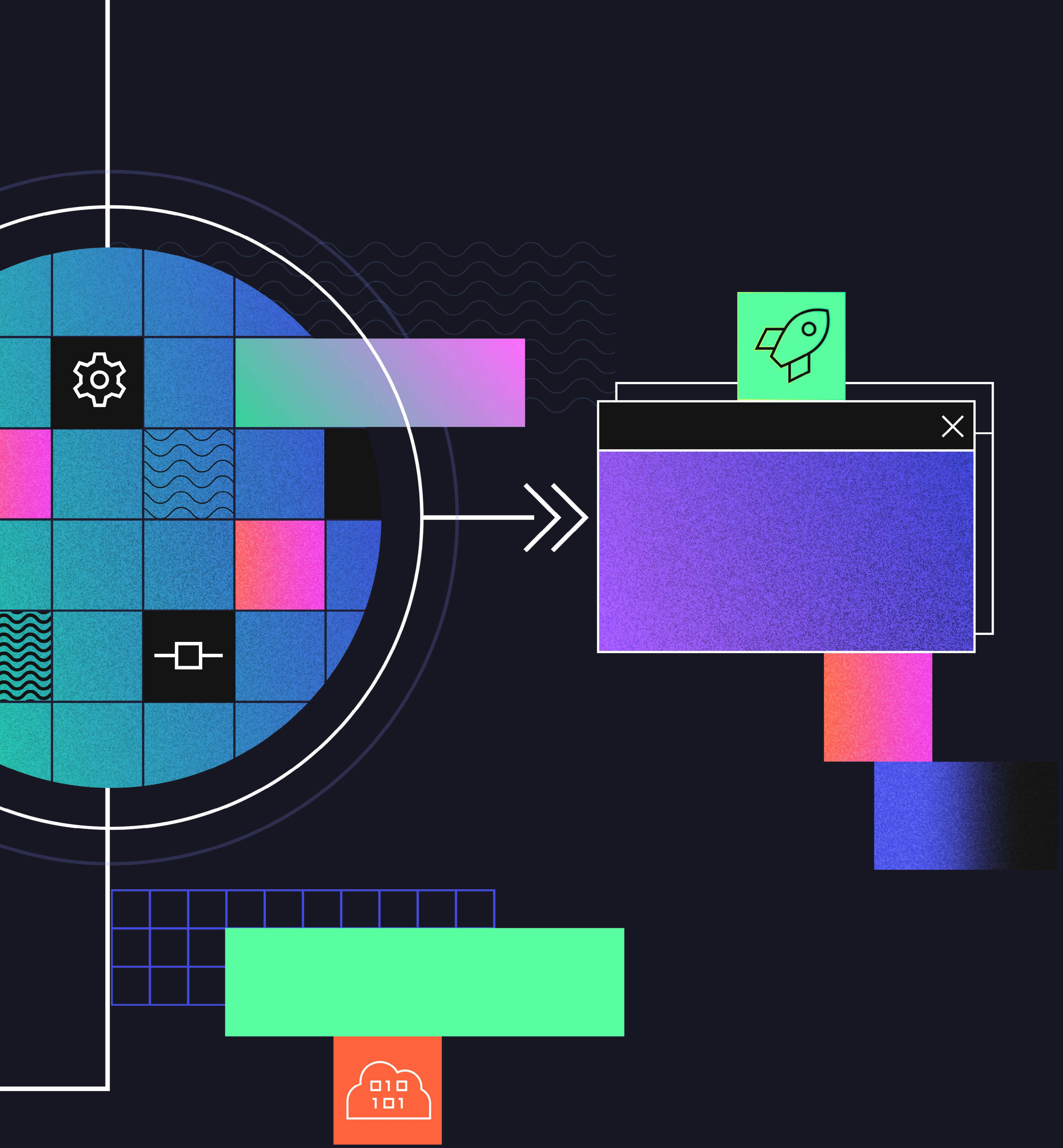
TECH GUIDE

Event-Driven Architectures

A better way forward

Temporal



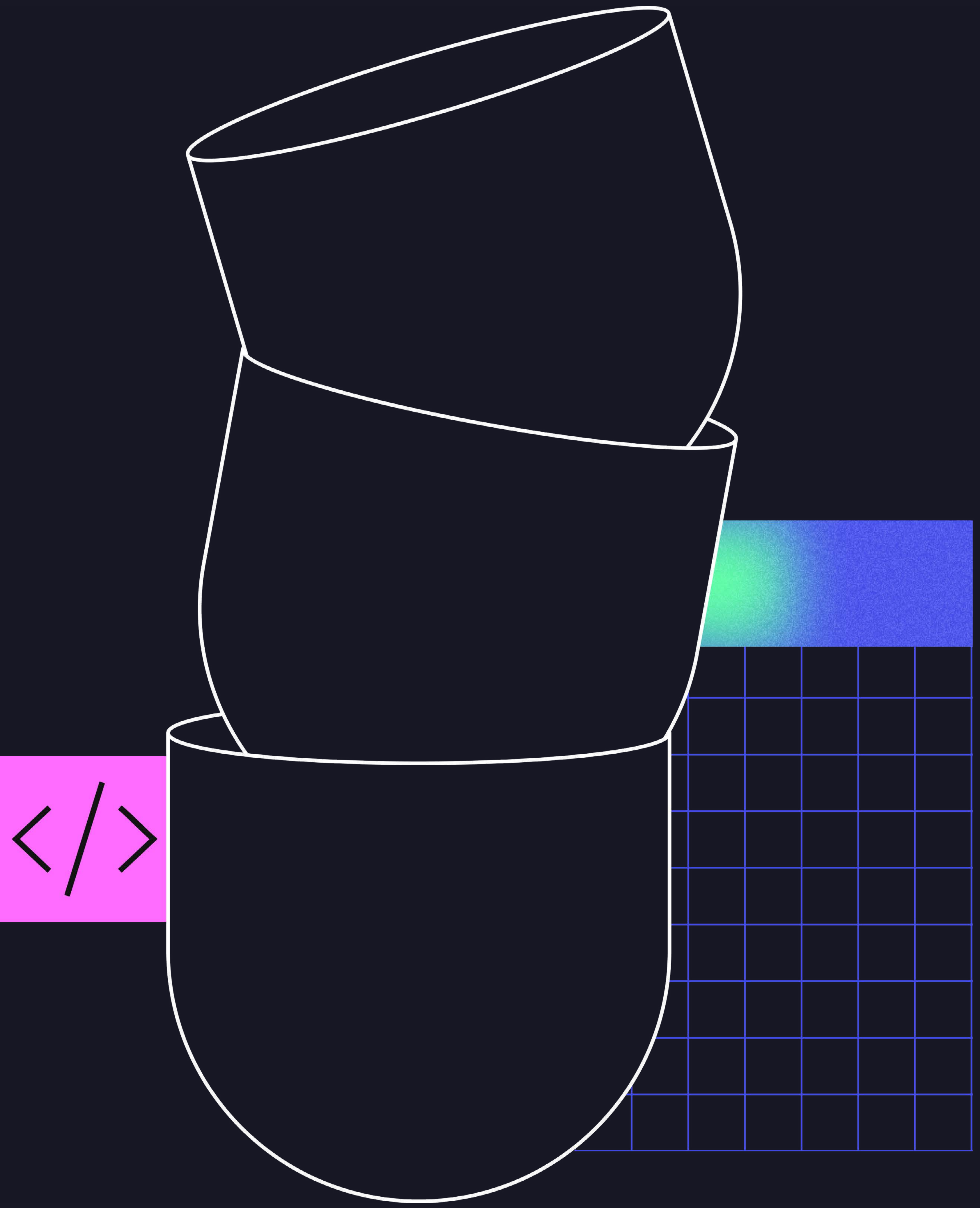
Event-driven architecture (EDA), which aims to communicate changes in state among components in a distributed application, has become a popular approach to application design in recent years.

Event-driven systems offer some benefits over more traditional monolithic applications, such as separation of concerns and ease in refactoring components. However, these benefits come at a non-trivial cost.

Many developers who have followed this approach report that it made their applications more complex, harder to debug, and more difficult to evolve.

Temporal (originally named Cadence) is an open source project originally created at Uber to solve many of the pain points developers encounter when working on event-driven systems.

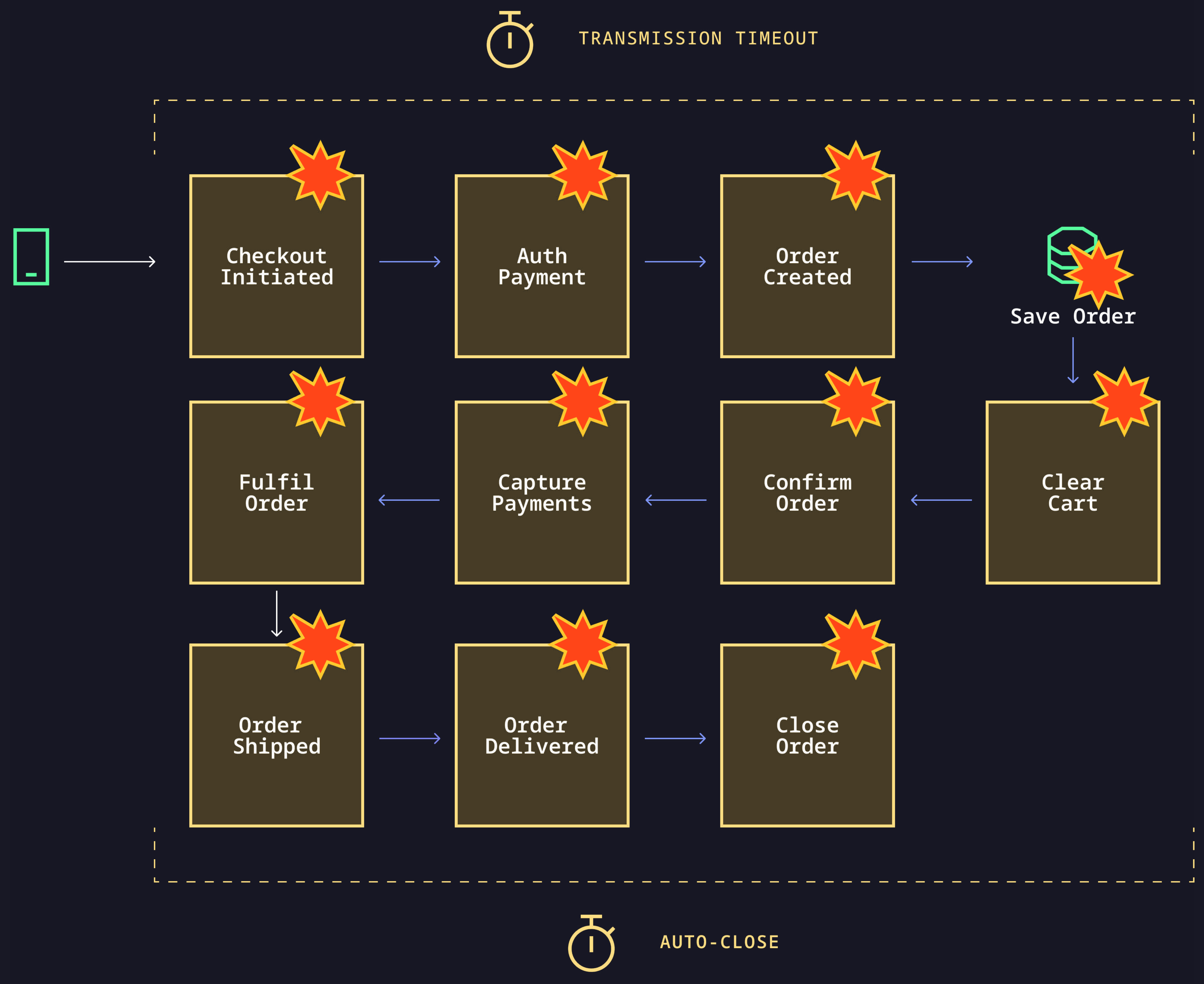
This guide will discuss some of the pros and cons of the event-driven approach, and how to mitigate some key pain points.



The Pros and Cons of Event-Driven Architectures

In event-driven architectures, an ***event*** generally corresponds to a meaningful change to state, and components can generally react independently after receiving an incoming event notification. If you purchase a one-of-a-kind teacup from an online shop, the state of the teacup changes from “available” to “sold.”

In turn, this event–the sale of the teacup–would likely trigger a series of downstream reactions from different components: your credit card is charged, an email appears confirming your purchase, the warehouse receives instructions to pack the teacup and ship to your chosen address.



A example of an event-driven architecture for an e-commerce shop selling tea cups.

In a traditional monolithic architecture, all of those downstream reactions would be handled in the same code base: the email messages, the credit card, the warehouse process. Although that kind of architecture is intuitive, and often faster, it becomes brittle over time. If there's an error in the credit card processing code, you might only notice the issue because of some odd behavior in the warehouse code, which makes the error tricky to track down.

It also lacks flexibility–if you want to change your warehouse's inventory system, or accept a new kind of payment (say, electronic funds transfer or cryptocurrency), you'd have to make major changes to your large code base.

By comparison, event-driven architecture distributes the complexity of an application, making it more straightforward to add features, but more challenging to test all possible states or edge cases.

Event-driven application architectures allow for a highly distributed and resilient application. Teacup inventory records can be replicated across multiple data centers, with updates propagating each time a new order is received. If teacups suddenly surge in popularity as a consumer product, it's relatively trivial to add capacity by creating a new copy of application state and firing incoming events at it.

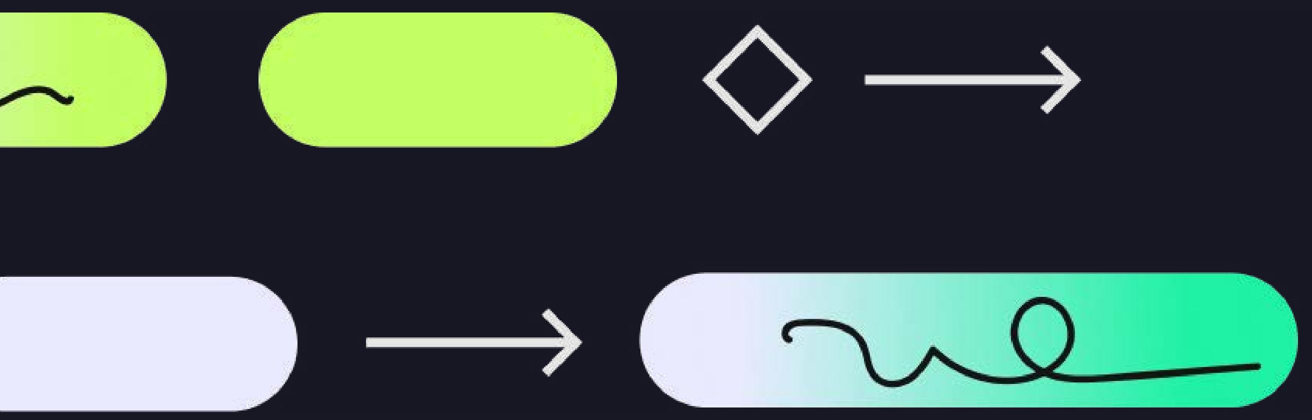
In practice, however, event-driven architectures are far from perfect.

Logic is no longer centralized in one place, but scattered across all the different services, making it harder to track down and resolve the source of issues. A change to one service–say, accepting a new currency or changing `dateTime` from US-Pacific to UTC–might have no impact at all, or it might bring business operations to a halt. In the latter case, the change could be rolled back as a solution, but only after that change has been identified as the root cause.

Distributing state across services is the core essential compromise of event-driven architectures. This compromise requires the addition of significant complexity and overhead to deal with error conditions and their resolutions. Because of this, individual services start to function like ad hoc state machines, storing state in a local database as they consume and produce messages.

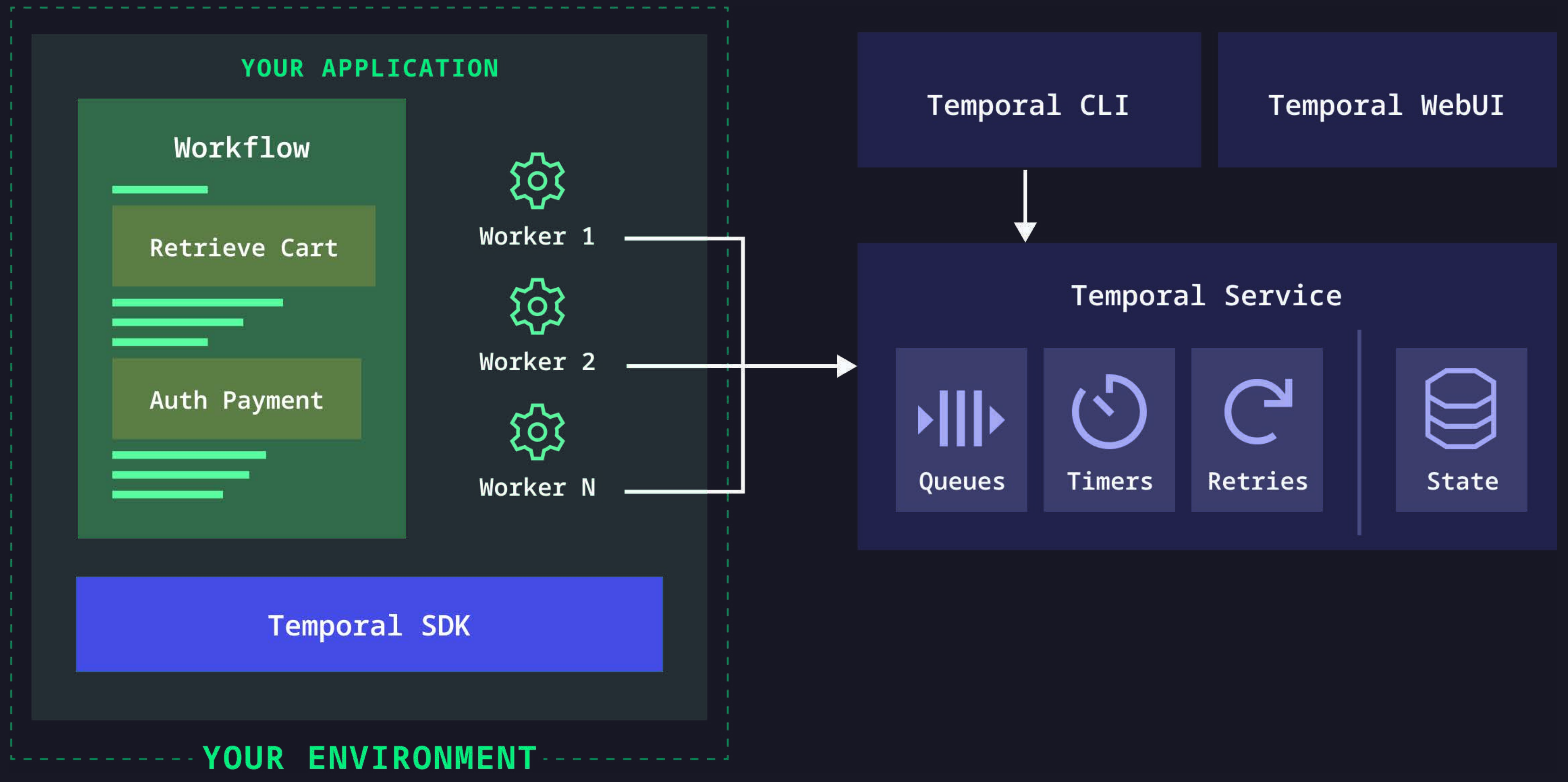
With the state distributed across services, you have limited visibility into system operations, which makes it harder to diagnose and resolve issues. Further, events aren't coordinated between services, nor are they transactional. In other kinds of systems, transactions ensure that all parts of a process either complete successfully or fail together. In contrast, services in event-driven architectures operate independently, becoming difficult to maintain over time.

In a perfect world, development teams eliminate these obstacles, allowing developers to represent their application's business logic directly as code. Or they could find ready-to-use solutions that create logical transactions while retaining the strengths of event-driven design. That's where Temporal comes in.



That's where Temporal comes in.

Evolve Your Way of Thinking



Temporal abstracts away the complexity of building scalable distributed systems by centralizing business logic into a series of steps we call a **Workflow**.

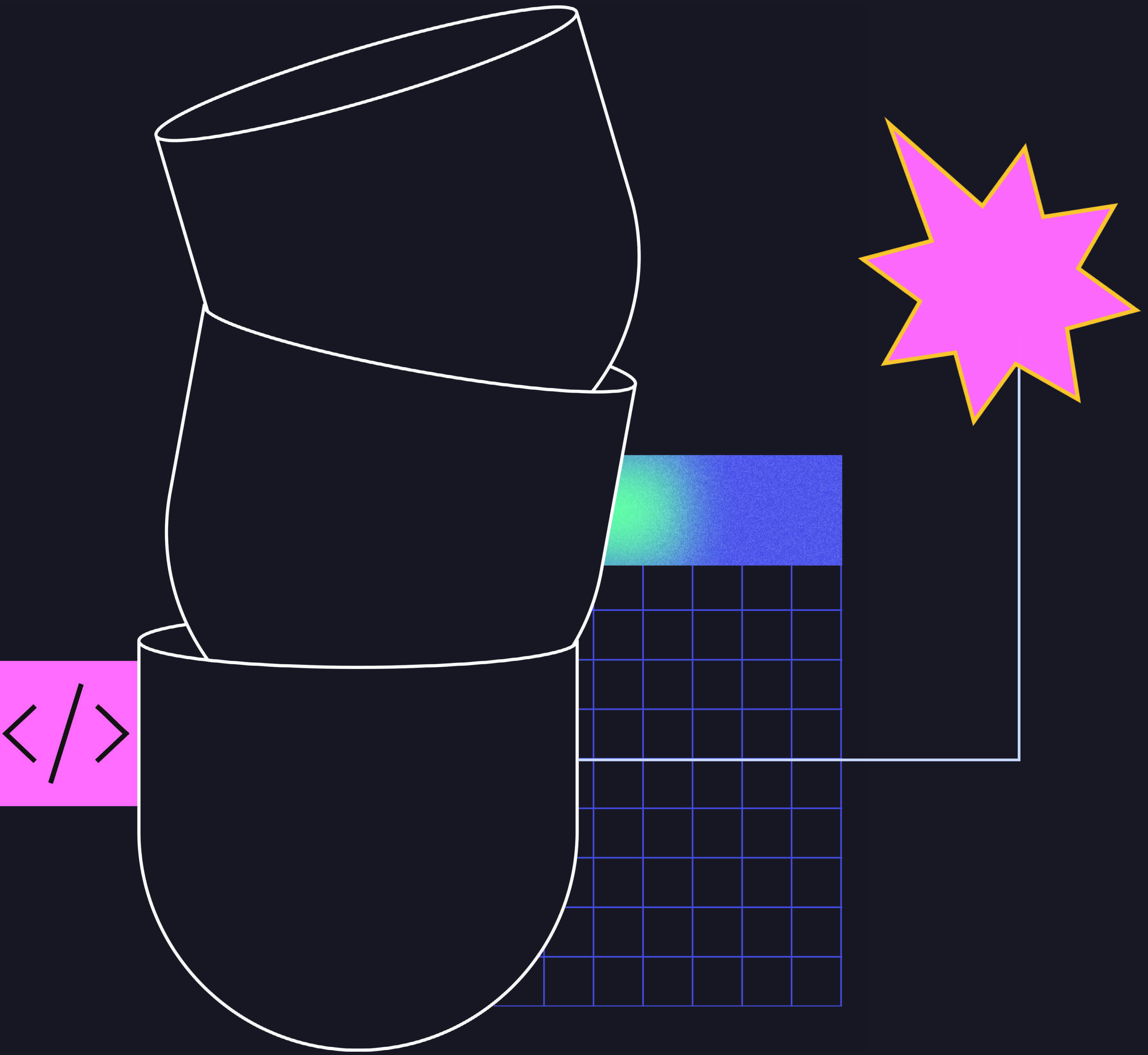
A Temporal Workflow could involve moving money between bank accounts, processing online orders, deploying cloud infrastructure, or something else entirely. Because the running state of a Workflow is always durable and fault-tolerant, any execution of the Workflow can be paused, restarted, recovered, or replayed from any point.

Activities–the individual units of work in a Temporal Workflow–interact with other external and internal services to execute tasks such as writing to a database, calling an API, or making network requests. The Workflow state is automatically captured and recorded as each Activity completes. Preserving code execution state is what allows Workflows to automatically resume operations at the correct point following an outage or other incident.

TEMPORAL TERMS

Conceptually, a **Workflow** is a sequence of steps written in a general-purpose programming language. The series of steps defined by written code are known as a **Workflow Definition**. When these steps are carried out by running the code, the result is a **Workflow Execution**.

In our teacup e-commerce example, the checkout process might be structured as a single **Workflow Definition**, while a single customer completing a purchase successfully would constitute a **Workflow Execution**.



Recall our earlier example of a would-be teacup buyer. Imagine some part of the order flow encounters an obstacle–anything from a rate limit error on your payments endpoint (these are *unusually* popular teacups, after all) to a system-wide failure or datacenter outage. In an event-driven architecture, developers would have to add handling for all of these potential failures or collisions to any place in the code base where teacup order state is tracked, to avoid making teacup-related promises that cannot be fulfilled.

Depending on how this failure and retry logic has been implemented, orders that are actively being processed might be lost entirely, or might be inconsistently executed. A teacup might be removed from inventory but no corresponding shipment order reaches the warehouse, or a buyer might be charged twice for the same purchase.

Temporal's durable execution approach allows developers to write remote calls that can seamlessly resume execution upon system recovery, reconstructing your application state as needed. Temporal also facilitates infinite retries (or a different custom retry behavior of your choice) and offers various other utilities for mitigation, compensation, and recovery.

No system is failure-proof, and while outages are (ideally) rare, they're also a fact of life. Adopting Temporal simply ensures that applications can gracefully recover when these issues do–inevitably–happen, allowing developers to resume incomplete operations when the impacted services are back online.

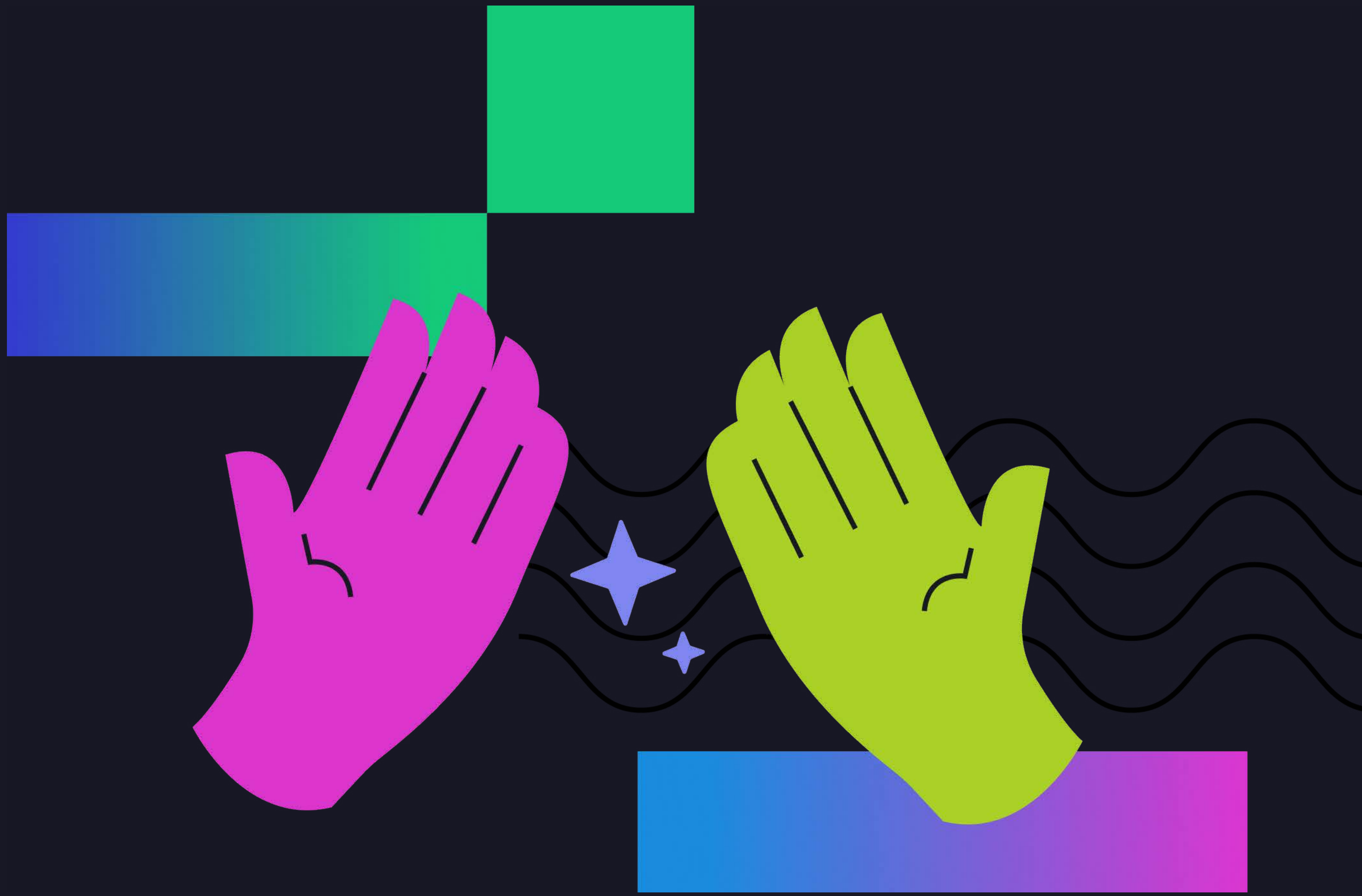
A Better Developer Experience

Despite their shortcomings, event-driven architectures excel in certain areas, such as ease of extensibility and scalability. That is why Temporal retains the benefits of event-driven architecture. It helps you build decoupled services as Activities while abstracting out the most difficult and problematic aspects of state tracking and error handling.

Best of all, you can implement Temporal in your preferred language or languages – pick the right tool for each job in your system – while using your favorite libraries, IDEs, and other tools. Incorporating durable execution into event-driven architecture means Temporal significantly enhances your developer experience.

You can use Temporal for time-based actions or for tracking sequential steps in EDAs, or even in monolithic architectures. Temporal is also suitable for synchronous services, ensuring consistency and reliability in actions such as scanning QR codes. It can serve as the communication medium between services, enabling seamless integration.

Temporal implementation doesn't demand an all-or-nothing approach. It supports incremental adoption, so that you can tailor your adoption to your unique needs and team dynamics.



In Conclusion

Temporal offers a paradigm shift in building applications, providing developers with a powerful toolset to navigate the complexities of event-driven architectures.

By adopting Temporal, your organization can enhance reliability, maintainability, and scalability while delivering a better experience for your developers.

Get started with Temporal
SET UP YOUR LOCAL ENVIRONMENT NOW

See how it works in your language
WITH EXAMPLE APPLICATIONS

TS php GO Java .NET