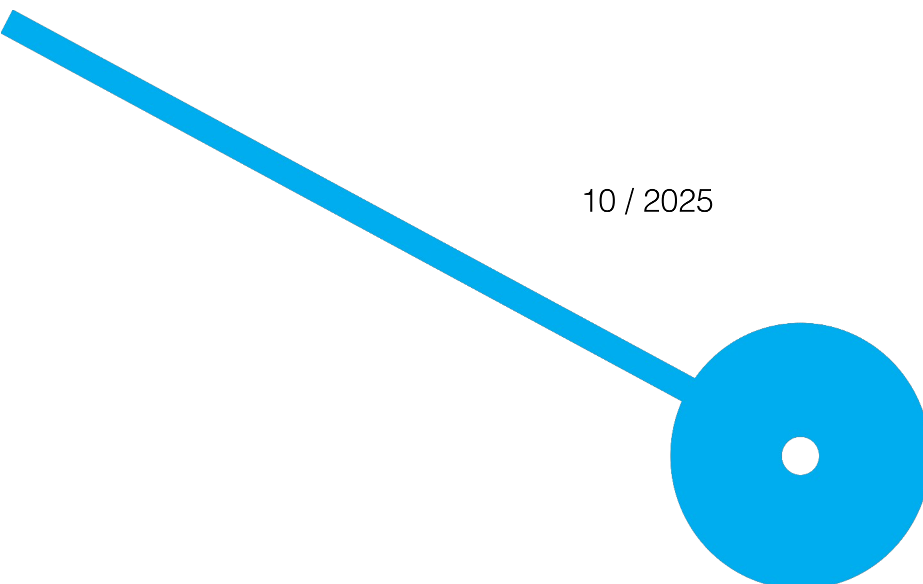
P.PORTO

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO
POLITÉCNICO DO PORTO

Architecting Serverless Applications at the Edge with Workflows

Micael André Cunha Dias

8200383



10 / 2025

P.PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

Architecting Serverless Applications at the Edge with Workflows

Micael André Cunha Dias
8200383

Advisor(s)

PhD Ricardo Jorge da Silva Santos

Project Report submitted in fulfilment of the requirements for the Master's degree in Informatics Engineering in the School of Management and Technology of the Polytechnic of Porto.

10 / 2025

Integrity statement

I, Micael André Cunha Dias, student nº 8200383, of the Master's Degree in Informatics Engineering of the School of Management and Technology of the Polytechnic of Porto, declare that I have not plagiarized or self-plagiarized, therefore the work entitled "Architecting Serverless Architectures at the Edge with Workflows" is original and of my own authorship, not having been used previously for any other purpose. I further declare that all sources used are cited, in the text and in the final bibliography, according to the referencing rules adopted in the institution.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Ricardo Jorge da Silva Santos, for all the guidance and support he provided throughout the development of this master's thesis. His insights, encouragement, and availability, despite his many other responsibilities, were in helping me navigate the challenges of this final academic stage.

Over the past five years at the Escola Superior de Tecnologia e Gestão, of Politécnico do Porto, I had the privilege of meeting many incredible people and forming friendships that enriched my journey. I am sincerely thankful for all the good moments, memorable adventures, and the valuable lessons I learned from each of you. You all made the university feel like a second home.

I also wish to extend my heartfelt appreciation to my family. Their unwavering support, constant encouragement, and the many sacrifices they made were essential to this achievement. I am deeply grateful for everything they have done to help me reach this milestone.

Lastly, I want to acknowledge the personal effort it took to balance a full-time 9-to-5 job while being a master's student. The commitment I had to put on was a significant challenge, and reaching the end of this journey feels even more rewarding because of it.

Micael André Cunha Dias

／# Abstract

The global gaming industry increasingly depends on digital monetization strategies subscriptions, consumables, in-game currencies, and asset sales as a foundation for sustainable growth. Yet while payment processors such as Stripe [1] provide secure and scalable access to financial infrastructure, game creators continue to face significant challenges in reliably orchestrating the end-to-end fulfillment of purchases. Failures in coordinating payments, inventory updates, and payouts not only erode player trust but also threaten developer revenues. This dissertation addresses these challenges by exploring how workflow orchestration principles can be applied to the domain of gaming monetization.

We revisit distributed-systems design from monoliths to microservices and serverless, arguing that durable execution [2] is the missing abstraction for reliable, long-lived, stateful workflows. We compare leading orchestration platforms Temporal [2], AWS Step Functions [3], Cloudflare Workflows [4], Uber Cadence [5], Netflix Conductor [6], and Camunda [7] along axes of durability, responsiveness, developer ergonomics, and governance, revealing fundamental trade-offs between progress guarantees and operational latency, and between expressive power and auditability. Building on these findings, we propose an edge-first workflow architecture that pairs Cloudflare's global runtime with Stripe's [1] payment primitives to deliver retryable and resilient payments and payouts.

The contributions of this thesis are twofold: a conceptual framework situating payment workflows within distributed systems and orchestration theory, and a practical design blueprint that illustrates how these principles can be implemented to support real-world game economies. By bridging the gap between creative innovation and financial infrastructure, this research provides game developers with a scalable, resilient foundation for monetization, enabling them to focus on player experience while trusting the underlying system to deliver seamless and reliable commerce.

Keywords: Distributed systems, Workflow orchestration, Durable execution, Cloudflare workers, Cloudflare workflows

Resumo

A indústria global dos videojogos depende cada vez mais de estratégias de monetização digital subscrições, consumíveis, moedas virtuais e vendas de ativos como base para um crescimento sustentável. No entanto, embora processadores de pagamento como a Stripe [1] forneçam acesso seguro e escalável à infraestrutura financeira, os criadores de jogos continuam a enfrentar desafios significativos na orquestração fiável do cumprimento ponta-a-ponta das compras. As falhas na coordenação de pagamentos, atualizações de inventário e pagamentos a terceiros não só corroem a confiança dos jogadores como também ameaçam as receitas dos desenvolvedores. Esta dissertação aborda esses desafios ao explorar como os princípios de orquestração de fluxos de trabalho podem ser aplicados ao domínio da monetização em jogos.

Revisitamos o design de sistemas distribuídos dos monólitos aos microserviços e serverless defendendo que a execução durável [2] é a abstração em falta para fluxos de trabalho fiáveis, duradouros e com estado. Comparamos as principais plataformas de orquestração Temporal [2], AWS Step Functions [3], Cloudflare Workflows [4], Uber Cadence [5], Netflix Conductor [6] e Camunda [7] ao longo dos eixos de durabilidade, capacidade de resposta, ergonomia para programadores e governação, revelando compromissos fundamentais entre garantias de progresso e latência operacional, e entre poder expressivo e auditabilidade. Com base nestas conclusões, propomos uma arquitetura de fluxos de trabalho edge-first que combina o runtime global da Cloudflare com os mecanismos de pagamento da Stripe [1], de forma a fornecer pagamentos e transferências resistentes e com capacidade de repetição.

As contribuições desta tese são duplas: por um lado, um enquadramento conceptual que situa os fluxos de pagamento no contexto dos sistemas distribuídos e da teoria da orquestração; por outro, um plano prático de design que ilustra como estes princípios podem ser implementados para suportar economias de jogo reais. Ao unir inovação criativa e infraestrutura financeira, esta investigação fornece aos desenvolvedores de jogos uma base escalável e resiliente para monetização, permitindo-lhes focar-se na experiência do jogador enquanto confiam no sistema subjacente para oferecer um comércio contínuo e fiável.

Palavras-chave: Sistemas distribuídos, Orquestração de fluxos de trabalho, Execução durável, Cloudflare Workers, Cloudflare Workflows

Table of Contents

Chapter 1 Introduction....10

1.1 Context and motivation....11

1.2 Objectives expected....13

1.3 Document structure....14

Chapter 2 State of the Art....17

2.1 Distributed systems....21

2.2 Monoliths....21

2.3 Microservices....22

2.4 Serverless computing....22

2.5 Edge computing....23

2.6 Function-as-a-service (FaaS)....23

2.7 Backend-as-a-Service (BaaS)....24

2.8 Durable Execution....24

2.9 Workflow Definition....25

2.10 Workflow Execution....26

2.11 Workflow Activities....26

2.12 Workflow Workers....27

2.13 Workflow Engines....27

Chapter 3 High level Architecture of Workflow Engines....28

3.1 Netflix Conductor....29

3.2 Temporal....32

3.3 Cloudflare Workflows....34

3.4 AWS Step Functions....37

3.5 Camunda....39

Chapter 4 Conceptual Architecture....42

4.1 Architecture Interface....44

4.2 Workflow Skeleton....46

4.3 Relational Storage....50

4.4 Design patterns....53

4.4.1 Retry and timeout policy for workflow steps....53

4.4.2 Event-first and polling fallback with race condition.....55
4.4.3 Early-return gating for replayable workflows.....59
4.4.4 Batch prepared SQL statements.....60
4.4.5 Proxied steps factory.....63
4.5 Workflows.....65
4.5.1 Setup bank connection workflow.....67
4.5.2 Initiate identity session workflow.....71
4.5.3 Setup payment instrument workflow.....73
4.5.4 Initiate payment workflow.....76
4.5.5 Setup payout instrument.....78
4.5.6 Initiate payout workflow.....80
4.6 Deployment.....82
4.7 Planned Timeline.....84
Chapter 5 Results achieved.....86
Chapter 6 Conclusion and Future Work.....88
6.1 Become whitelabel.....90
6.2 Revenue sharing mechanism.....90

Table of Figures

Figure 1: The macro problem with microservices..................17
Figure 2: Netflix Conductor workflow definition in JSON format..................28
Figure 3: High-level Architecture of Netflix Conductor..................30
Figure 4: High-level Architecture of Temporal..................31
Figure 5: Temporal workflow definition with Typescript..................32
Figure 6: High-level Architecture of Cloudflare Workflows..................34
Figure 7: Data Ingestion and Durable Execution on Cloudflare Workflows..................35
Figure 8: Designing Workflows on AWS Step Functions..................37
Figure 9: High-level Architecture of Camunda BPMN Engine..................39
Figure 10: Interface(s) between the system being developed and other systems..................43
Figure 11: Workflow Skeleton (Part 1)..................46
Figure 12: Workflow Skeleton (Part 2)..................48
Figure 13: Storage Entity-Relationship Diagram..................51
Figure 14: Workflow step config..................53
Figure 15: Workflow step..................54
Figure 16: Workflow handling webhook..................55
Figure 17: Workflow polling..................56
Figure 18: Workflow race condition..................57
Figure 19: Early-return workflow..................59
Figure 20: Batch prepared SQL statements (workflow activity)..................60
Figure 21: Internals of store step..................61
Figure 22: Store workflow step..................62
Figure 23: Workflow provider interface..................63
Figure 24: Workflow steps factory..................64
Figure 25: Setup bank connection workflow (Part 1)..................67
Figure 26: Setup bank connection workflow (Part 2)..................69
Figure 27: Initiate identity session workflow (create)..................70
Figure 28: Initiate identity session workflow (polling + webhook)..................71
Figure 29: Setup payment instrument workflow (setup intent)..................73
Figure 30: Setup payment instrument (polling)..................74

Figure 31: Initiate payment workflow (authorization and capture)....75
Figure 32: Initiate payment workflow (polling payment status)....76
Figure 33: Setup payout instrument workflow....77
Figure 34: Initiate payout workflow (assert balance + initiate payout)....79
Figure 35: Initiate payout workflow (polling)....80
Figure 36: Workers deployment....81
Figure 37: Project Timeline....82

Acronyms

API	Application Programing Interface
AWS	Amazon Web Services
BaaS	Backend as a Service
FaaS	Function as a Service
BPMN	Business Model Process Notation
DMN	Decision Model Notation
CI	Continuous Integration
CD	Continuous Deployment
ER	Entity Relationship
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
JSON	JavaScript Object Notation
RPC	Remote Procedure Call
SQL	Structured Query Language
UI	User Interface
RCON	Remote Console
ELK	ElasticSearch, Logstash, Kibana
KYC	Know Your Customer
AML	Anti-Money Laundering

Chapter 1 Introduction

In today's global digital economy, the gaming industry occupies a unique and rapidly expanding space. From small independent developers experimenting with creative mechanics to large-scale studios running complex online universes, game creators are increasingly reliant on monetization strategies to sustain their work. Digital distribution has dramatically lowered the barriers to reaching global audiences, but it has also heightened the demand for reliable, frictionless systems to handle payments, subscriptions, digital asset sales, and in-game credits. For creators, the challenge is not simply to produce engaging gameplay but to ensure that their content can be effectively commercialized and that revenues flow seamlessly across borders and financial systems.

Monetization in gaming introduces both technical and business complexities. Players expect instant, secure, and transparent transactions, regardless of their location or device. Regulatory environments require compliance with payment processing standards and anti-fraud protections. Game economies, especially those involving subscriptions, consumables, or virtual currencies, demand accurate real-time tracking of usage, renewals, and refunds. Finally, creators themselves expect timely and reliable payouts, often in multiple currencies, with the same level of trust and precision they associate with banking institutions. Building this infrastructure from the ground up is a formidable challenge. It is not only time-consuming and error-prone, but it also diverts resources away from the creative and innovative work that defines the core of game development.

This thesis explores how to design and implement a robust payments solution tailored specifically for the needs of game creators. The central hypothesis is that modern workflow orchestration engines and programmable payment processors can be combined to provide a foundation that is both flexible and reliable. Rather than reinventing the wheel, game creators can leverage such a platform to integrate monetization directly into their ecosystems while retaining the ability to scale as their player base grows. By offering automation of recurring tasks, resilience to infrastructure failures, and customizable flows for different business models, such a system can reduce the operational burden on creators and improve player trust in the stability of game economies.

The platform envisioned in this work aspires to be an all-in-one commerce solution for the gaming ecosystem. Whether serving an indie developer launching their first multiplayer server or a global studio with millions of concurrent users, the goal is to provide a blueprint for monetization infrastructure that grows alongside the creator. Key to this design is the emphasis on workflow orchestration: by formalizing payment and payout processes as workflows, we can guarantee durability, ensure recoverability from failures, and enable modularity for future extensions. Orchestration also provides transparency, offering creators a clear view into the state of their transactions and enabling fine-grained control over retry policies, error handling, and integrations with external services.

Through this exploration, the thesis aims not merely to propose an abstract architecture but to demonstrate how orchestration principles can be applied concretely to the domain of gaming monetization. The contribution is twofold: first, a conceptual framework that situates payment workflows within the broader theory of distributed systems and workflow engines; and second, a practical design that illustrates how these ideas can be implemented to deliver real-world value to creators. The ambition is to provide game developers with the confidence and precision needed to manage commerce within their ecosystems, freeing them to focus on their creative vision while trusting the underlying infrastructure to handle the complexity of global payments and payouts.

1.1 Context and motivation

In-game monetization has evolved from a peripheral feature into a cornerstone of the global gaming economy. What was once limited to expansion packs or premium upgrades has expanded into a vast ecosystem of digital transactions: consumable items, character skins, battle passes, subscriptions, and in-game currencies. Players now expect these transactions to be seamless. A purchase should be instantaneous, secure, and reflected immediately within the player's in-game inventory. Any disruption to this flow, delays, duplicate charges, or missing items, has a direct impact on both player satisfaction and developer revenue. In an industry where trust and immersion are paramount, friction in the monetization pipeline risks eroding the player's experience and damaging the reputation of the game.

For developers, however, achieving this ideal is far from straightforward. The technical challenge lies less in the mechanics of accepting payments, which are well supported by

payment processors, and more in orchestrating the reliable fulfillment of purchases. Once a player confirms a purchase, the system must coordinate across multiple components: verifying payment authorization, recording the transaction, updating the player's inventory, and ensuring idempotency so that items are not delivered multiple times. Each of these steps may involve different services, databases, and external APIs, all of which introduce opportunities for partial failures.

The complexity increases when monetization spans multiple platforms. Some game shops may exist as browser-based storefronts, while others are embedded directly within the game client. Regardless of where the purchase is initiated, the infrastructure must reconcile payment confirmation with inventory updates in real time. Consider a scenario where a network outage occurs after a payment is confirmed but before the item is credited to the player's account. Without robust orchestration, the developer risks both dissatisfied players demanding refunds and a permanent mismatch between financial and gameplay records. These edge cases, while rare, accumulate at scale to create substantial financial and reputational risk.

A further challenge lies in the lack of durable, fault-tolerant workflows available to most game creators. Large studios may invest in custom infrastructure to handle retries, reconciliation, and state tracking across distributed systems, but smaller developers often lack the expertise or resources to build such systems. As a result, many game shops implement fragile, ad hoc solutions. Failures are hard to detect and even harder to diagnose, as logs and state are scattered across payment APIs, inventory databases, and game servers. This lack of visibility impedes debugging and prolongs recovery from incidents. As games grow in popularity and transaction volumes increase, the shortcomings of such solutions are amplified, placing further strain on development teams.

These issues highlight the need for a specialized orchestration layer that abstracts away the operational complexity of monetization workflows while preserving flexibility and control for developers. Such a platform would not only accept and process payments but also guarantee that purchased items are delivered reliably to the correct player inventory. By offering built-in durability, automated retries, and full auditability, it would provide the resilience required for real-time gaming environments. Moreover, it would empower developers by surfacing clear visibility into transaction states, allowing them to debug and monitor workflows without reinventing core infrastructure.

This thesis is therefore motivated by the urgent demand to bridge the gap between payment processing and in-game fulfillment. The proposed system aspires to serve as a payment and fulfillment orchestrator for game creators of all sizes, from indie developers to global studios. Its ambition is to ensure that every purchase translates into a reliable, consistent player experience while relieving developers of the burden of implementing complex distributed workflows themselves. By combining the robustness of workflow orchestration engines with the flexibility of modern payment APIs, this research aims to provide a blueprint for monetization infrastructure that is scalable, resilient, and aligned with the realities of contemporary game economies.

1.2 Objectives expected

The overarching objective of this project is to design and implement a robust, scalable monetization platform tailored to the needs of game creators. In an environment where digital economies are central to gaming experiences, the ability to sell content, process payments securely, and deliver purchased items seamlessly to players has become essential. This project aims to bridge the gap between the creative focus of game development and the operational demands of global commerce by providing an integrated platform that combines workflow orchestration, modern payment infrastructure, and reliable delivery mechanisms. The emphasis is not only on technical feasibility but also on developer usability, operational reliability, and adaptability to diverse business models.

To achieve this vision, the project pursues a set of specific objectives that structure both its scope and methodology.

The first objective is to demonstrate the platform through a working prototype. This reference implementation will take the form of a game shop that integrates all components of the monetization process, from cart management to payment authorization and the final delivery of digital items into players' inventories. The purpose of this prototype is twofold: to validate the architectural design in practice and to serve as a blueprint for other developers wishing to adopt the platform. By making the implementation concrete and reproducible, the project provides a practical entry point for integration rather than remaining purely conceptual.

The second objective is to ensure observability, debuggability, and maintainability of the platform. Distributed workflows, particularly those involving payments, are highly sensitive

to failure and demand strong reliability guarantees. To achieve this, the system will be implemented in idiomatic, modular Typescript code, with a clear emphasis on maintainability and developer ergonomics. Cloudflare Workflow engine [4] will provide visibility into ongoing processes through its UI, allowing developers to inspect workflow histories, debug failures, and monitor long-running executions. By embedding observability into the core of the platform, the project ensures that developers and operators can trust its correctness and quickly resolve issues when they arise.

The third objective is to integrate with Stripe [1] for both payment processing and creator payouts. Stripe [1] provides a mature, secure, and globally adopted set of APIs for managing financial transactions. By leveraging Stripe's ecosystem [1], the platform gains access to compliance mechanisms, fraud detection, and multi-currency support without reinventing critical financial infrastructure. On the input side, Stripe [1] will be used to process players' purchases, ensuring seamless and secure payment flows. On the output side, Stripe will automate payouts to creators' bank accounts, reducing operational overhead and guaranteeing that revenues flow reliably from players to developers. This dual integration positions the platform as a comprehensive solution that handles the full financial lifecycle of game monetization.

1.3 Document structure

To begin with, chapter 1, introduction, lays the groundwork for the entire thesis. It sets the stage by presenting the context and motivation behind the research, showing why the chosen subject is relevant and necessary in today's technological landscape. This chapter also articulates the objectives that are expected to be met, both in terms of theoretical contributions and practical applications. Finally, it outlines the overall structure of the document, guiding the reader through the progression of ideas and arguments that will unfold in the subsequent chapters.

Following this, chapter 2, theoretical background, establishes the conceptual and technical foundation upon which the rest of the thesis is built. It begins with distributed systems, discussing their principles and role in modern computing. From there, it contrasts traditional monoliths with the microservices architectural style, highlighting their strengths and limitations. The chapter then moves into serverless computing, introducing Function-as-a-

Service (FaaS) [8] and Backend-as-a-Service (BaaS) [9], and examining how they enable scalable, event-driven systems. It further explores the concept of durable execution and defines workflows in detail, including their execution models, activities, and the responsibilities of workflow workers. This extensive overview ensures that readers unfamiliar with these paradigms can fully grasp the technical discussions that follow.

Building on this foundation, chapter 3, state of the art, surveys existing approaches, platforms, and frameworks that address the challenges of workflow orchestration [10]. It first discusses the macro-level problems associated with microservices, particularly the complexities they introduce at scale. It then examines the gap between serverless computing and orchestration, pointing out the coordination difficulties that arise. After setting this context, the chapter provides a landscape of engines and platforms currently in use, analyzing solutions such as Temporal, AWS Step Functions [3], Cloudflare Workflows [4], Netflix Conductor, Uber Cadence [5], and Camunda [7]. To bring coherence to this landscape, a comparison framework is introduced. This framework defines analytical dimensions, contrasts primary platforms with additional ones, and synthesizes their relative merits and shortcomings. By the end of this chapter, the reader gains a comprehensive picture of the state of research and practice in workflow orchestration [10].

In continuation, chapter 4, conceptual architecture, presents the proposed design and solution in detail. It begins with the cloud platform, focusing on the choice of Cloudflare Workers [11] and Cloudflare Workflows [4], and explaining why these technologies are particularly suitable. It then introduces the architecture, followed by a detailed discussion of storage. This includes schema relationships, normalization strategies, and the modeling of business entities such as merchants, shops, and players. It also addresses financial structures like bank connections, balances, and payouts, and the role of identity sessions in meeting regulatory requirements. Next, the chapter presents several design patterns aimed at ensuring robustness, efficiency, and reliability. These include retry and timeout policies for workflow steps, event-first strategies with polling fallbacks, early-return gating for replayable workflows, batching prepared SQL statements, and the introduction of proxied steps factories. Afterwards, the focus shifts to workflows themselves, which are presented through concrete cases such as setting up bank connections, initiating identity sessions, configuring payment instruments, initiating payments, managing payout instruments, and executing payouts. The chapter

concludes with a discussion of deployment strategies and a timeline for planned implementation, thus bridging conceptual design with practical realization.

Subsequently, chapter 5, results achieved, documents the outcomes of the proposed system. It evaluates the architecture and workflows in practice, reflecting on their performance, reliability, and scalability. This chapter not only demonstrates the technical validity of the approach but also highlights the contributions of the work in comparison to existing solutions. By situating the results within the broader research landscape, it provides evidence of both innovation and practical applicability.

Looking ahead, chapter 6, conclusion and future work, identifies potential directions for extending and refining the system. Among these directions are the transformation of the solution into a white-label product, enabling broader adoption across industries, and the design of a revenue-sharing mechanism to support sustainable business models. This chapter illustrates the long-term vision and scalability of the research, emphasizing its relevance beyond the immediate implementation and brings the thesis to a close by reflecting on the research journey as a whole. It revisits the initial motivation and objectives, evaluates how they have been addressed, and summarizes the main findings. Moreover, it considers the limitations of the current work and the contributions it has made to both theory and practice.

Chapter 2 State of the Art

Over the past two decades, the architecture of software systems has undergone a dramatic transformation, evolving from tightly coupled monoliths to increasingly granular and distributed microservices [12]. This shift has been driven by several converging pressures: the explosive growth of global-scale applications, rising expectations for constant availability, and the demand for organizations to continuously deliver new functionality at speed. In the early era of software engineering, the monolithic model was the dominant paradigm [34].

Monoliths offered simplicity, predictability, and strong consistency guarantees. All application logic, state, and data resided within a single runtime and database, creating an environment where state transitions were atomic, transactional, and inherently reliable. Failures were straightforward, if an operation failed, it failed completely, leaving no ambiguity in the system's state. This deterministic behavior allowed developers to reason intuitively about application correctness and provided a resilient foundation for building software [35].

As systems grew in scale, however, the limitations of monoliths became apparent. Vertical scaling, adding more power to a single machine, inevitably encountered hardware ceilings, while horizontal scaling, cloning the entire monolithic application across servers, introduced inefficiencies, resource contention, and operational overhead. These bottlenecks, combined with the organizational challenge of coordinating large developer teams within a single codebase, pushed architects to seek alternatives.

The adoption of containerization and orchestration platforms such as Docker [37] and Kubernetes [38] accelerated this transition, providing the tooling to manage vast fleets of services. This trend culminated in microservices, which decomposed applications into autonomous, loosely coupled units. Each microservice could be developed, deployed, and scaled independently, enabling organizations to move faster, foster team autonomy, and take advantage of diverse infrastructure technologies.

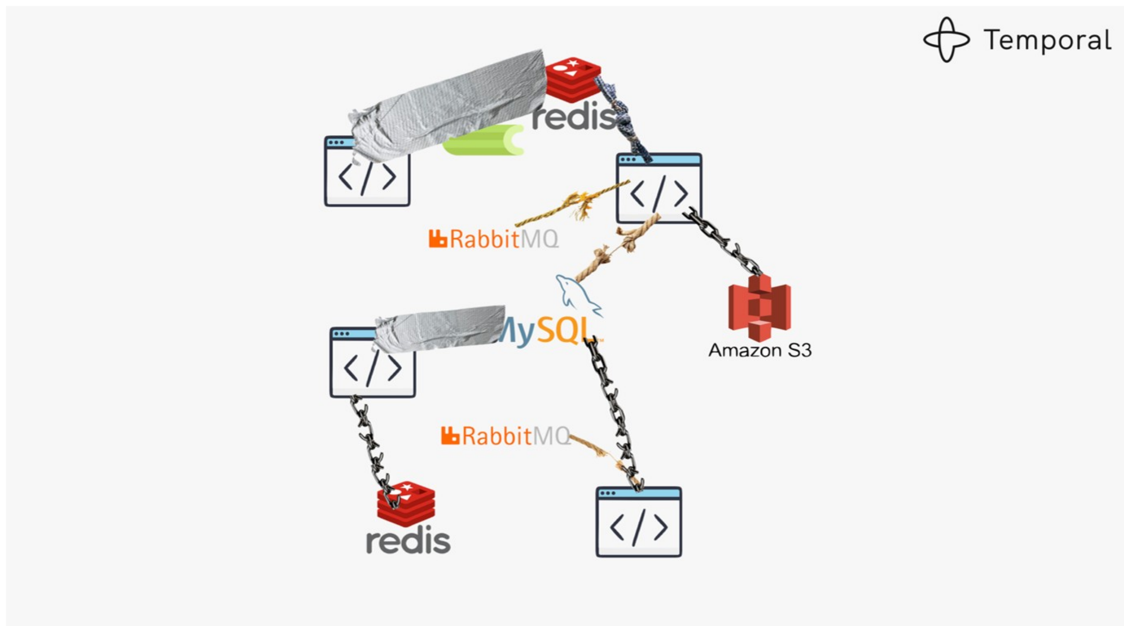


Figure 1: The macro problem with microservices

Yet this transformation introduced a new and intractable class of problems. At the heart of the challenge lies the decentralization of state. In a monolith, the database was the single source of truth, and transactions were atomic and reliable [13]. In microservice architectures, state is fragmented across dozens or even hundreds of services, each with its own database, queues, caches, and replication mechanisms [12]. What was once a local database transaction becomes a distributed Saga [14], involving multiple independent services, each of which may fail in isolation. Inter-service communication is mediated by unreliable networks, where timeouts, dropped packets, and transient outages are the norm rather than the exception.

For developers, this represents a profound shift in mental models. The assumption of deterministic all-or-nothing semantics no longer holds. Every remote call introduces the possibility of partial completion, and with it the need for compensating actions, retries, exponential backoff strategies, idempotent operations, and other fault-tolerance patterns [15]. Business logic that would once have been expressed concisely within a monolithic transaction is now entangled with layers of error-handling boilerplate. This erodes readability and maintainability, while also diffusing responsibility across teams and services.

The ad-hoc nature of these reliability mechanisms exacerbates the problem. Different teams implement retries, timeouts, and compensation differently, leading to a proliferation of subtle

inconsistencies [12]. Some services may handle failures gracefully, while others may silently drop requests or leave data in an indeterminate state. The result is a fragile ecosystem where race conditions, cascading failures, and inconsistent user experiences become difficult to avoid and harder still to diagnose.

The subsequent rise of serverless computing, through Function-as-a-Service (FaaS) [9] and Backend-as-a-Service (BaaS) [10], further abstracted infrastructure concerns and enabled developers to deploy highly elastic, event-driven applications with minimal operational overhead. Yet, as applications became increasingly fragmented across services, functions, and managed backends, the limitations of these paradigms became apparent: ensuring durability, orchestrating long-running processes, and achieving exactly-once semantics remained open problems.

By abstracting away infrastructure management, serverless platforms such as AWS Lambda [20], Google Cloud Functions [21], Azure Functions [22] and Cloudflare Workers [3] allowed developers to focus exclusively on implementing business logic in the form of stateless, event-driven functions. This model proved particularly effective for workloads characterized by high variability and short execution times, offering automatic scaling, usage-based billing, and seamless integration with managed services. Similarly, Backend-as-a-Service (BaaS) [10] platforms such as Firebase [26] further accelerated development by providing fully managed capabilities for identity management, storage, and messaging through standardized Application Programming Interfaces (APIs). Together, these paradigms lowered the barrier to building cloud-native applications, democratizing access to scalable infrastructure for organizations of all sizes.

Yet, despite their benefits, Funtion-as-a-Service (FaaS) [9] revealed structural limitations when applied to complex business processes. The stateless nature of serverless functions makes them ill-suited for long-running or stateful operations, requiring developers to externalize state management into databases, queues, or caches [8]. Error handling and retries must be implemented manually, often resulting in duplicated side effects, inconsistent states, or brittle compensation logic. Furthermore, workflows spanning multiple functions or services quickly become opaque and difficult to reason about, as control flow is fragmented across event triggers and asynchronous callbacks. In practice, what begins as an elegant event-driven

design frequently devolves into ad hoc orchestration patterns that are hard to debug, maintain, or extend.

These challenges created a growing demand for more systematic approaches to coordination. Workflow orchestration [11] frameworks emerged as a response, bridging the gap between the simplicity of serverless execution and the reliability requirements of real-world distributed systems. By introducing durable state management, deterministic execution, and built-in mechanisms for retries, compensation, and observability, orchestration engines such as AWS Step Functions [2], Temporal [1], and Cloudflare Workflows [7] provide a foundation for modeling complex processes as cohesive, fault-tolerant workflows. In this way, orchestration represents not a rejection of serverless principles, but rather their natural extension retaining elasticity and simplicity while reintroducing the durability and control necessary for long-lived, business-critical operations.

Within this context, workflow orchestration [10] has emerged as a response to the coordination gap left by microservices and serverless platforms. Orchestration frameworks [16] provide the mechanisms to define, execute, and monitor complex processes that span heterogeneous systems, offering guarantees of durability, recoverability, and consistency. They encapsulate the patterns of retries, compensation, and state tracking that would otherwise need to be reimplemented in an ad-hoc manner, thereby reducing engineering overhead while improving operational reliability.

In many ways, the industry has traded one form of complexity for another. The monolithic model constrained scalability but simplified reasoning about correctness and state. Microservices unlocked scalability and organizational velocity but forced developers to grapple with the full spectrum of distributed systems challenges, consensus, durability, idempotency, and eventual consistency [17]. The pressing question for the future is whether new abstractions, such as durable workflow engines or stateful platforms that combine the reliability of monoliths with the scalability of distributed systems, can close this gap. Until such paradigms are mainstream, engineering teams will continue to spend disproportionate effort just ensuring that their distributed systems remain coherent and reliable [16].

2.1 Distributed systems

A distributed system is a collection of processes working together to accomplish some task [18]. Each process is a deterministic program unit able to execute separately from, and concurrently with, other processes [18].

In a distributed system the processing model is a higher order unit: a loosely coupled collection of machines executes multiple programs that cooperate to achieve a business objective. The smallest unit of work we distribute is a process (think Docker container) or a group of processes (think Kubernetes pod) [18].

Developers typically employ networked microservices with Hypertext Transfer Protocol (HTTP) or Remote Procedure Calls (RPCs), use queues, distributed data stores, replicated state machines, conflict-free replicated data types, leader election, or any number of other approaches to implement algorithms across the cloud to make progress on business logic [18]. *Such systems necessarily have a significant amount of complexity that exists onl*y to handle things like consistency and durability in a distributed environment.

2.2 Monoliths

A monolith is a single, unified application where all components of the system are built, deployed, and scaled together [19]. In this model, the entire application usually shares one codebase and often one central database. All parts of the system, business logic, user interface, and data access are tightly coupled within the same deployment unit [17].

The monolithic approach is often easier to start with because development is straightforward. Everything is in one place, which reduces complexity at the beginning of a project. Since all components run in the same process, performance benefits from simple in-process communication, and consistency is easier to guarantee because a single database handles all transactions [20].

However, as systems grow, monoliths become harder to maintain and evolve. Scaling requires replicating the entire application rather than just the parts under heavy demand. Multiple teams working on the same codebase can run into coordination problems, and even small changes necessitate redeploying the whole system. A single bug or failure in one part of the

application can impact the entire system, and deployments carry more risk as the codebase becomes larger and more complex. For small projects or startups, monoliths are often a natural starting point, but they tend to become bottlenecks as organizational and technical complexity increase [17].

2.3 Microservices

Microservices represent a different architectural style in which an application is decomposed into a collection of small, independent services [21]. Each service is responsible for a specific business capability and is deployed separately. Services communicate with each other over well-defined network interfaces, commonly through REST, Google Remote Procedure Call (gRPC), or message queues. Unlike monoliths, microservices often own their own data storage, which leads to decentralized data management [18].

This approach provides several advantages in terms of agility and scalability. Teams can work on different services independently, using the technologies that best fit their needs, and deploy changes without affecting the entire system. Only the services experiencing high load need to be scaled, which improves efficiency. Microservices also enhance resilience: if one service fails, the rest of the system can often continue functioning, provided fault isolation is well designed [17].

At the same time, microservices bring their own challenges. Communication between services requires network calls, which introduce latency and potential points of failure. Ensuring consistency across multiple services and their independent data stores is complex and often demands advanced distributed systems patterns. Operating a microservices-based system requires significant investment in infrastructure, including service discovery, monitoring, logging, tracing, and automated deployment pipelines. As a result, while microservices offer flexibility and resilience for large, complex applications, they come with substantial operational overhead compared to monolithic architectures [22].

2.4 Serverless computing

As distributed systems continued to evolve beyond monoliths and microservices, a new paradigm emerged under the umbrella of *serverless computing [23]*. Also referred to as *Function-as-a-Service (FaaS) [8]*, this model marked a further abstraction of infrastructure

management, enabling developers to deploy small, stateless functions that are triggered by events and scale transparently with demand. Platforms such as AWS Lambda [24], Google Cloud Functions [25], Azure Functions [26] and Cloudflare Workers [11] epitomize this approach, offering elastic scalability, automatic provisioning, and usage-based billing models that reduce operational overhead. In this way, serverless architectures extended the principles of cloud-native design by shifting even more responsibility for reliability, availability, and scaling to the underlying platform, allowing engineers to focus exclusively on implementing business logic.

2.5 Edge computing

As latency-sensitive and data-intensive workloads proliferated, computation began to move closer to where data is produced and consumed. Edge computing [27] denotes this shift: executing code on infrastructure geographically proximate to users, devices, and sensors rather than exclusively in centralized cloud regions. The primary motivations are straightforward yet compelling single-digit to tens-of-milliseconds response times for interactive experiences; regulatory and contractual data-locality constraints; bandwidth and egress-cost reduction via local preprocessing; and improved resiliency when backhaul connectivity is intermittent.

Programming models at the edge borrow heavily from serverless while adapting runtimes for ultra-low latency and high density. Lightweight isolate/V8 environments [28] offer sub-millisecond cold starts and strong in-process isolation for request-time logic. The developer experience mirrors cloud Function-as-a-service (FaaS) [9], ephemeral stateless handlers, usage-based billing but with additional network-adjacent primitives such as programmable caches, request coalescing, and fine-grained egress control.

2.6 Function-as-a-service (FaaS)

Function-as-a-service (FaaS) [8] represents the core abstraction of serverless computing. In this model, developers write individual functions that encapsulate discrete units of business logic, which are then deployed to a cloud provider's infrastructure. These functions are ephemeral and stateless, meaning they are instantiated only when triggered by an event, such as an HTTP request, a database update, or a message in a queue. The provider manages every

aspect of execution, including provisioning, load balancing, and automatic scaling, ensuring that resources are allocated elastically in response to workload fluctuations [29].

The appeal of Function-as-a-service (FaaS) [8] lies in its simplicity and efficiency. Developers are relieved of the burden of managing servers, containers, or runtime environments, and instead focus on fine-grained logic that responds to specific events. Billing is tied directly to execution time and resource consumption, offering a cost model that aligns naturally with bursty or unpredictable workloads. These properties make Function-as-a-service (FaaS) [8] particularly attractive for microservices-based systems, real-time data processing, and applications with variable demand.

2.7 Backend-as-a-Service (BaaS)

Backend-as-a-Service (BaaS) [9] constitutes another dimension of serverless computing, targeting a different layer of abstraction. Whereas Function as a Service (FaaS) [8] provides execution environments for user-defined functions, Backend-as-a-Service (BaaS) [9] delivers fully managed backend services that expose common functionality through APIs. These services typically include authentication and authorization, database storage, push notifications, file hosting, analytics, and messaging. Popular esxamples include Firebase [30], AWS Amplify [31], and Cloudflare [11].

Backend-as-a-Service (BaaS) [9] reduces the time and expertise required to build backend systems by outsourcing responsibility for non-differentiating infrastructure to the provider. Instead of implementing identity management or data synchronization from scratch, developers integrate their applications with pre-built, cloud-hosted services that scale transparently with user demand. This model accelerates development cycles, enabling teams to focus on delivering domain-specific features while relying on the BaaS provider to ensure reliability, availability, and compliance.

2.8 Durable Execution

One of the core properties of a robust distributed system is durable execution [32]. This refers to the ability of a system to ensure that computational tasks run to completion, regardless of failures in hardware, network, or dependent services.

Durable execution abstracts away much of the complexity typically associated with retries, failure recovery, and timeout management. From the developer's perspective, the code executes seamlessly, even if the underlying environment is transient or unstable. When the system becomes idle such as when it awaits a response from an external service it efficiently releases compute resources and resumes execution when conditions allow [16].

Unlike traditional retry mechanisms, which often require explicit error handling and custom recovery logic, durable execution abstracts these concerns away from the developer. Tasks can be retried automatically and transparently, based on well-defined retry policies, without duplicating side effects or violating consistency guarantees. When a workflow or activity becomes idle for example, while waiting on an external service the system releases compute resources, conserving infrastructure while maintaining logical progress [16].

2.9 Workflow Definition

In distributed systems, a workflow [32] defines the sequence of operations required to achieve a business objective by orchestrating the execution of multiple distributed tasks. It represents the logical flow of an application, including control structures such as conditionals, loops, retries, and parallel execution. workflows are stateful entities that may span extended periods, allowing them to coordinate complex, long-running processes without requiring continuous resource allocation.

A key requirement for workflows is determinism. To ensure reliable execution and support replay-based fault recovery, a workflow must produce the same behavior given the same inputs, regardless of when or where it runs. All non-deterministic behavior, such as external API calls or time-based operations, is offloaded to activities. The workflow engine records these activity results and reuses them during re-execution, enabling consistency even across restarts or failures [33].

Workflows abstract the coordination logic away from the operational logic, promoting modularity and fault isolation. This model simplifies the implementation of distributed applications by handling retries, timeouts, and error recovery within the workflow engine itself. As a result, developers can focus on modeling business processes without managing the underlying complexities of distributed execution [33].

2.10 Workflow Execution

A workflow execution refers to a concrete instance of a workflow definition being carried out. While the workflow definition specifies the sequence of activities, control flow, and error-handling logic in an abstract manner, the execution represents the runtime manifestation of that definition, initiated in response to a specific request or event. Each execution is parameterized by its input, which allows the same definition to be reused across multiple scenarios [34]. For example, an order-processing workflow definition may be executed once to handle order #123 and again to process order #567, with both executions following the same logical flow but operating on distinct data.

Each execution proceeds independently, with its progress, intermediate state, and outcome recorded by the workflow engine. This separation guarantees that multiple executions can coexist without interference, while the engine enforces determinism to ensure reproducibility and reliability across failures or restarts [34].

2.11 Workflow Activities

Within the architecture of distributed systems, activities represent the fundamental units of execution responsible for carrying out discrete operations [35]. These operations often involve side effects, such as invoking external APIs, writing to or reading from databases, processing files, or performing other I/O-bound tasks. Unlike workflows, which are stateful and deterministic, activities are typically stateless and may produce non-deterministic results due to their interaction with external systems.

Because activities operate in an environment where partial failure is expected, they are designed to be idempotent ensuring that repeated execution yields the same result without adverse effects. This property is crucial for enabling automatic retries and failure recovery mechanisms. Activities are triggered and managed by a workflow engine, which tracks their execution, ensures consistency, and retries them transparently in the event of failure [35]. From the developer's perspective, activities are implemented as standard functions or methods, and are invoked within the workflow as if they were local calls, despite executing in separate processes or even on separate machines.

2.12 Workflow Workers

Workers are long-running service processes responsible for polling task queues, executing activities, and reporting their results. They act as the execution environment for the activity code [36].

Each worker registers the set of activity functions it can handle and continuously listens for tasks from the orchestration layer (i.e., the workflow engine or runtime) [36]. Once an activity task is received, the worker executes the associated function and returns the result or error.

A key attribute of workers is their stateless nature. All persistent application state is maintained by the workflow engine, allowing workers to focus solely on executing assigned tasks. This statelessness enables workers to be easily replicated or replaced, contributing to the system's horizontal scalability and fault isolation. If a worker crashes or becomes unavailable, tasks are reassigned to other available workers without compromising the correctness or progress of the workflow [18].

2.13 Workflow Engines

A workflow engine [10] is an application that runs digital workflow software. Also called orchestration engines, workflow engines enable businesses to create and automate workflows, often by using workflow-as-code, low-code no-code visual builders. Workflow engines store business logic and executable steps for orchestrating workflows, and automate the triggers, actions and events that comprise a particular workflow.

It enables execution as if failures don't even exist. The application will run reliably even if it encounters problems, such as network outages or server crashes, which would be catastrophic for a typical application [37].

Chapter 3 High level Architecture of Workflow Engines

Workflow engines share a common architectural blueprint: definition of workflows, a control or orchestration engine, task execution workers, queueing and dispatch mechanisms, durable persistence of state and history, scheduling/triggering subsystems, monitoring interfaces, connectors to external systems, and error/recovery support. The trade-space lies in how deterministic execution is enforced, how scalable and decoupled each layer is, the richness of the connector ecosystem, and the operational load placed on the user versus the framework.

The above architectures as described in the article exemplify how different engines occupy different points in that design space. This chapter explores the below five well-known workflow orchestration engines:

1. Netflix Conductor: A state-machine-based orchestrator designed for microservices coordination, featuring JavaScript Object Notation (JSON-defined) workflows, polling-based execution, and distributed task management [38].

2. Temporal: A durable execution system leveraging an event-sourcing model [39] for fault-tolerant workflows, offering deterministic replay, long-running execution, and fully programmable workflow definitions [40].

3. Cloudflare Workflows: A serverless, edge-native orchestration framework that executes workflows across Cloudflare's global edge network [27]. It adopts an event-driven model, integrates seamlessly with HyperText Transport Protocol (HTTP-based) services, and utilizes Durable Objects [32] for state persistence, enabling low-latency and highly distributed automation.

4. AWS Step Functions: An AWS-managed orchestration service that models workflows as state machines using the Amazon States Language (ASL) [41]. It supports sequential, parallel, and conditional execution flows, integrates with AWS services, and ensures reliability through persistent execution history and error-handling mechanisms [42].

5. Camunda: A lightweight yet enterprise-grade orchestration platform built on the BPMN 2.0 (Business Process Model and Notation) standard. Camunda [7] emphasizes human–machine workflow integration, providing both code-driven and model-driven

approaches, along with rich monitoring and decision-automation capabilities via DMN (Decision Model and Notation).

3.1 Netflix Conductor

Conductor [6] expresses workflows in a JavaScript Object Notation (JSON-based) Domain System Language (DSL) and uses an Application Programming Interface (API) layer for interactions. The orchestration logic is factored into services (workflow, task, decider). Tasks are enqueued in distributed queues; worker nodes poll for tasks and update statuses. The persistence layer supports scalable backends (e.g. Cassandra [43]), and the system includes observability dashboards. The primary components of Netflix Conductor [6] include:

1. Workflow Definitions: Workflows within Netflix Conductor [6] are specified through a JavaScript Object Notation (JSON-based) Domain-Specific Language (DSL). This representation formally encodes the sequencing of tasks, their interdependencies, and conditional execution paths. Consequently, workflow modifications can be implemented without necessitating alterations to the underlying microservices.

Figure 2 provides an illustrative example of a workflow definition in JSON format.

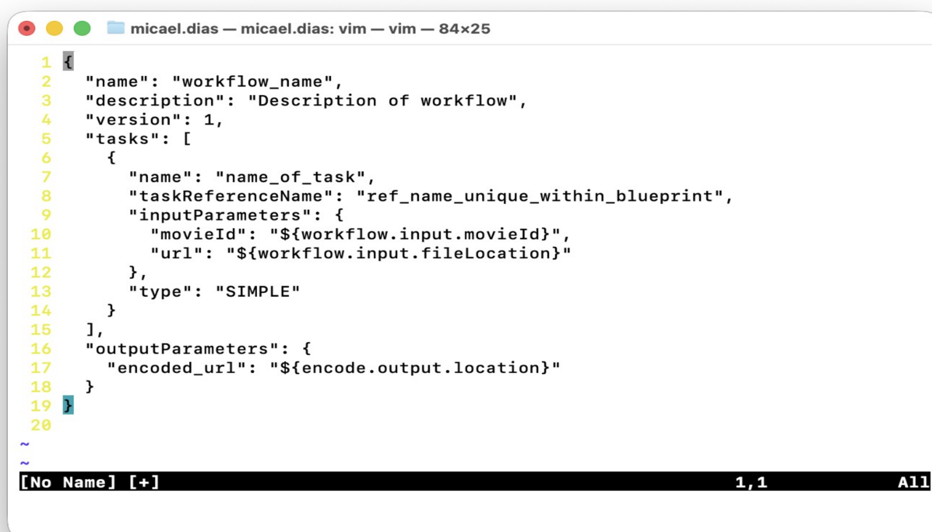


Figure 2: Netflix Conductor workflow definition in JSON format

2. Application Programming Interface (API) Layer: The API layer constitutes the interaction medium between the orchestration engine and distributed task workers. It exposes a set of endpoints through which workers can poll for pending tasks, update execution statuses, and query workflow specifications. This design ensures interoperability, fosters loose coupling, and enables seamless coordination across heterogeneous services [43].

3. The service layer integrates the principal subsystems that collectively facilitate workflow execution and orchestration. These subsystems include:

   (a) Workflow Service: Responsible for managing the complete lifecycle of workflows, encompassing initiation, monitoring of execution progress, and governance of state transitions [38].

   (b) Task Service: Dedicated to the management of task-level execution, including status tracking, retry mechanisms, and fault tolerance strategies [44].

   (c) Decider Service: Operates as the decision-making component that continuously evaluates workflow states to identify subsequent executable tasks. It encapsulates scheduling logic, dependency resolution, and conditional branching [43].

   (d) Queue Service: Administers distributed task queues, ensuring equitable distribution of workload, efficient polling mechanisms, and reliable worker-task assignment [43].

4. Task Queues and Workers: Conductor [6] employs a distributed queuing mechanism to manage the scheduling and execution of tasks. This architecture ensures efficient task allocation to Workers [45], while providing robustness through mechanisms for handling delays, retries, and fault tolerance [38].

5. Tasks embedded within workflows are executed by independent worker applications [45]. These workers may either (i) expose RESTful interfaces for direct invocation by the orchestration engine or (ii) implement a polling-based paradigm to retrieve and execute pending tasks.

6. Monitoring and Observability: Observability constitutes a critical dimension of Conductor's operational model. Real-time monitoring capabilities can be provisioned for each workflow, with support for integration into widely adopted observability

platforms such as Prometheus [46], Datadog [47], and the Elasticsearch, Logstash, Kibana (ELK) stack [48]. These integrations allow for comprehensive collection, visualization, and analysis of operational metrics. Additionally, Conductor [6] provides a browser-based User Interface (UI) that offers practitioners an intuitive dashboard for workflow debugging, execution tracing, and performance monitoring.

Figure 3 illustrates the high-level architecture of Netflix Conductor [6]. Central to this design is a poll-based task execution model, wherein worker nodes interact asynchronously with the orchestration engine [38]. The orchestration engine is responsible for scheduling tasks and persisting workflow execution metadata within both a database and an indexing service.

Tasks are enqueued into distributed Task Queue [38], which serve as the primary coordination mechanism between the orchestrator and the workers. Worker nodes [49] continuously poll these queues to retrieve pending tasks. Upon acquiring a task, a worker executes the associated business logic and subsequently communicates the execution outcome by updating task status through the Management and Execution Service [50].

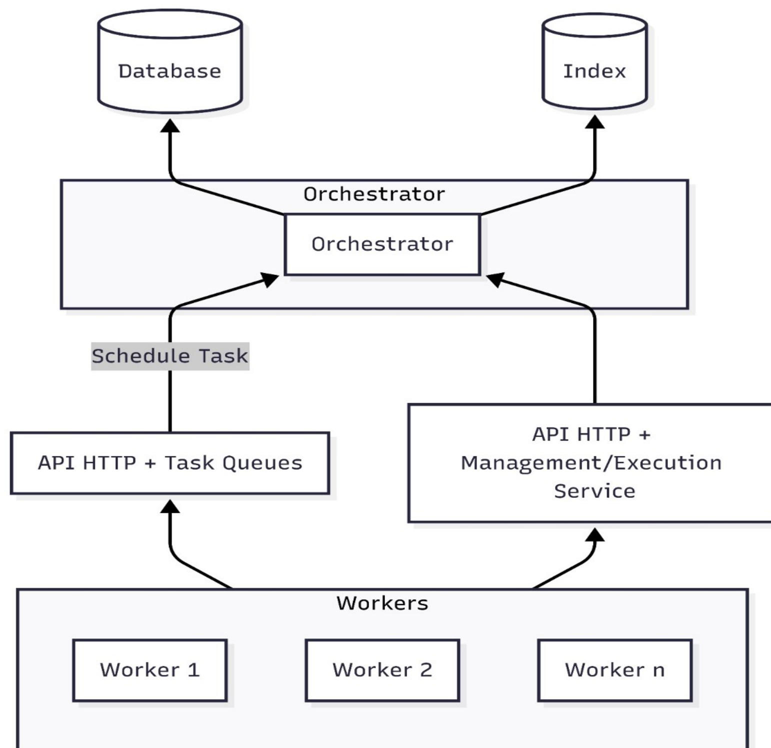


Figure 3: High-level Architecture of Netflix Conductor

3.2 Temporal

Temporal [51] is predicated on an event-sourcing execution model, which provides resilience and robustness in workflow orchestration [52]. Under this paradigm, workflows are capable of continuing execution even after failures, as all relevant execution state is durably recorded.

Figure 4 illustrates the architecture of a Temporal cluster [40]. Once a workflow begins execution, its metadata is persisted through the History Service [53], ensuring the possibility of deterministic replay. Worker nodes [54] poll Temporal's Task Queues [55] to obtain executable tasks, while the Frontend Service manages client interactions and overall coordination. The Matching Service [40] directs tasks to workers, and the History Service maintains the authoritative execution record.

This architecture enables deterministic execution, long-running workflow support, fault tolerance, and high scalability across distributed environments. As a result, Temporal is particularly well-suited for microservices orchestration and large-scale business process automation.

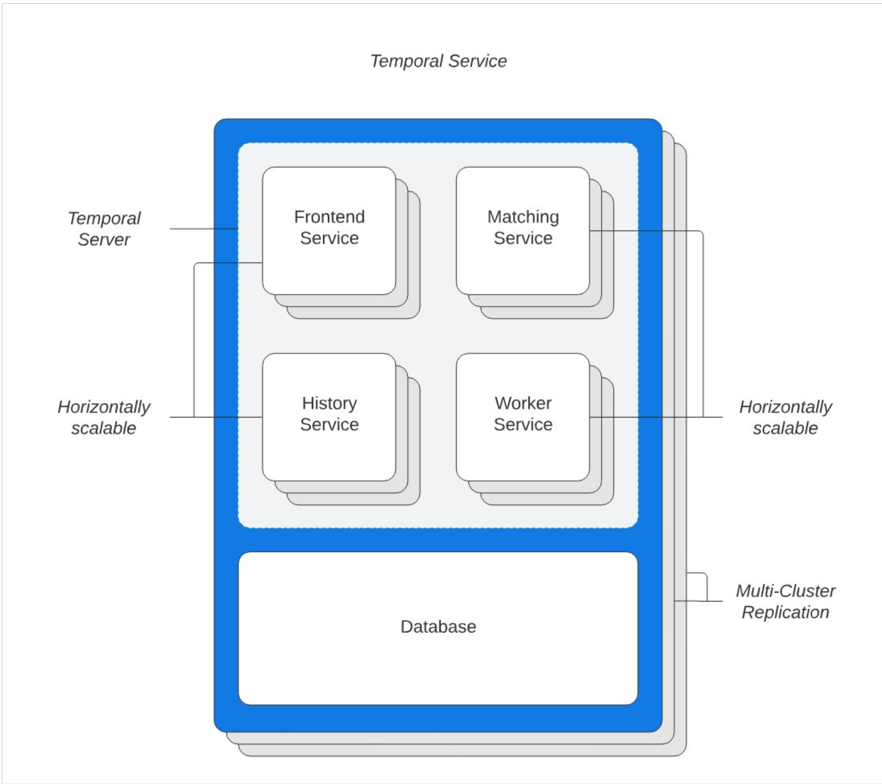


Figure 4: High-level Architecture of Temporal

The primary architectural components of Temporal [51] are as follows:

1. The Temporal Server constitutes the central coordinating entity of the system. It is responsible for scheduling workflows, ensuring persistence of execution state, and orchestrating task execution. This component is subdivided into specialized services:
   (a) Frontend Service: Exposes a gRPC-based API through which clients and worker processes interact with the system [40].
   (b) History Service: Maintains a comprehensive record of workflow event histories, thereby preserving state transitions and enabling workflow replay [40].
2. Workflows Definitions: Unlike orchestrators that rely on declarative specifications (e.g., JSON-DSL or YAML), Temporal workflows are authored as executable code. This approach affords developers full programmability and flexibility in defining workflow logic.
   (a) Workflows are deterministic in execution, ensuring identical outcomes for identical inputs [56].
   (b) This determinism enables workflows to be replayed reliably from historical logs [57], a feature that underpins system resilience and guarantees consistent execution semantics.

Figure 5 provides an illustrative example of a workflow definition in Typescript.

```
import { proxyActivities } from '@temporalio/workflow';
// Only import the activity types, not the functions themselves
import type * as activities from './activities';

// Retrieve the Activity Handle by passing in the Activity types and options
const activityHandle = proxyActivities<typeof activities>({
  startToCloseTimeout: '1 minute',
});

// Deconstruct the individual Activity functions from the Activity Handle
const { greet } = activityHandle;

// A workflow that calls an activity
export async function example(name: string): Promise<string> {
  return await greet(name);
}
```

Figure 5: Temporal workflow definition with Typescript

3. Activities: Activities [58] represent external operations invoked within workflows, such as database queries, API interactions, or computational procedures. These are executed by Workers [54], which operate asynchronously and autonomously. Workers [54] incorporate built-in mechanisms for retries, error handling, and state persistence, thereby simplifying fault-tolerance management and ensuring the robustness of long-running workflows.
4. Task Queues and Matching Service: Execution tasks are enqueued within Task Queues [55], which function as decoupling mechanisms between workflow logic and execution processes.
    (a) The Worker nodes [59] poll these queues to retrieve and execute workflow tasks.
    (b) The Matching Service [40] mediates this interaction by assigning queued tasks to appropriate workers.
5. History Service and Event Logs: The History Service [53] serves as a persistent log of all workflow events, capturing state transitions such as initiation, task scheduling, task completion, and workflow termination. Temporal [51] provides multiple perspectives for analyzing workflow histories:
    (a) Timeline View: Displays execution primitives and durations, enabling performance analysis.
    (b) Compact View: Groups related events (e.g., scheduling, start, and completion of an activity) into collapsible sections for efficient navigation.
    (c) Full History View: Offers a complete record of workflow events, sortable in ascending or descending order, thereby supporting fine-grained auditing of workflow execution.

3.3 Cloudflare Workflows

Cloudflare Workflows [4] is a serverless orchestration framework designed to enable the composition and automation of distributed applications directly within Cloudflare's global edge network [27]. Unlike traditional orchestrators that rely on centralized clusters,

Cloudflare leverages its edge infrastructure to execute workflows closer to the end-user, thereby reducing latency and enhancing resilience.

Figure 6 shows a high level, Cloudflare Workflows leverage the Cloudflare global edge network [27] to distribute workflow orchestration [60]. Workflow definitions are deployed as Worker scripts, which are automatically replicated across the network. Events trigger workflow executions, tasks are dispatched to Worker [11] runtimes, and state is persisted via Durable Objects or KV storage when required. This design achieves low-latency execution, horizontal scalability, and geographic redundancy without the need for dedicated orchestration clusters.

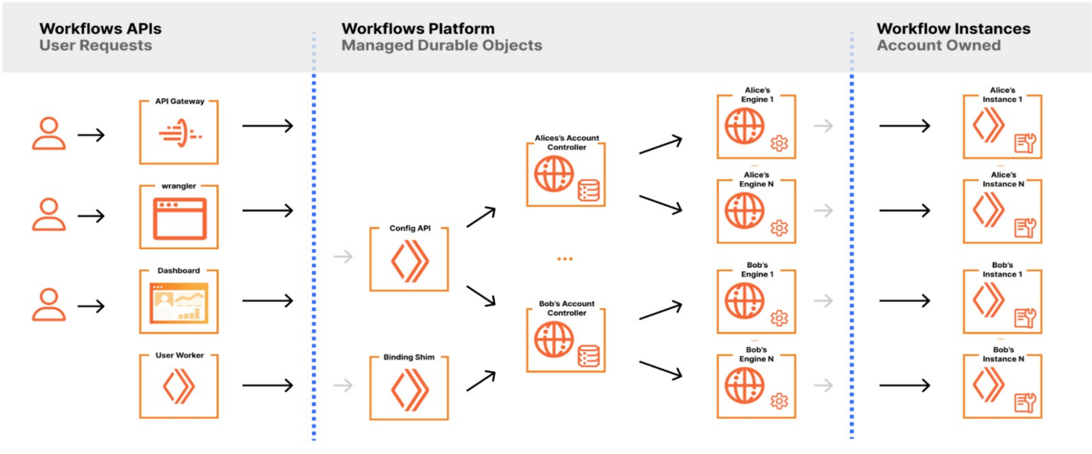


Figure 6: High-level Architecture of Cloudflare Workflows

The primary architectural components of Cloudflare Workflows [4] are as follows:

1. Workflow Definitions: Workflows within Cloudflare are authored declaratively, typically using JavaScript-based constructs that define sequences of tasks and control-flow structures. This design choice aligns with the event-driven serverless paradigm, where developers specify the orchestration logic through lightweight scripts rather than provisioning or managing dedicated orchestration infrastructure [60].

   Figure 7 shows a use case workflow of Data Ingestion and Durable Execution using Cloudflare Workflows.

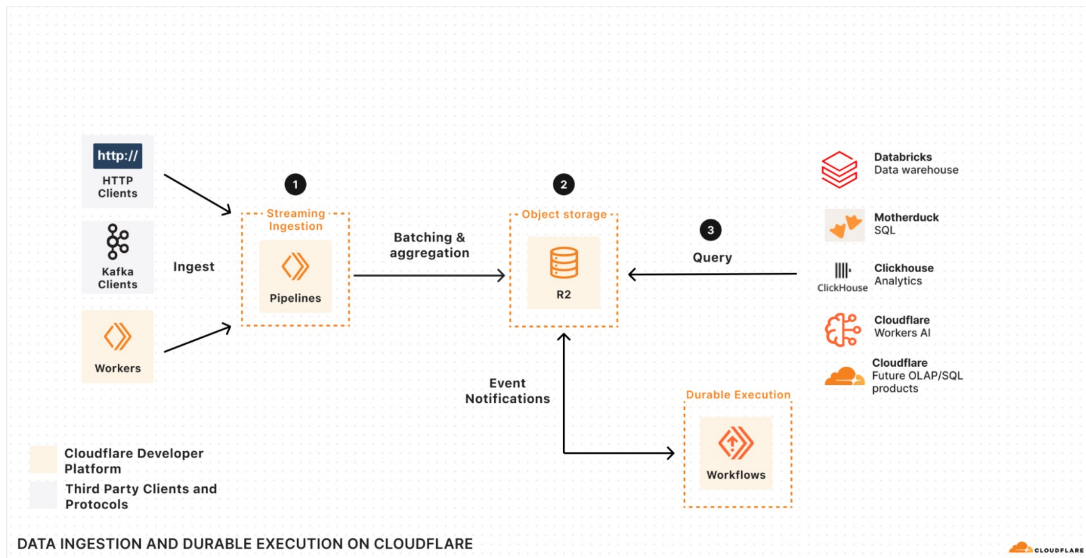


Figure 7: Data Ingestion and Durable Execution on Cloudflare Workflows

2. Orchestration Engine: The orchestration engine is embedded within Cloudflare's edge environment [27] and is responsible for coordinating workflow execution. It interprets workflow definitions, schedules tasks, and manages control-flow constructs such as branching, retries, and error handling [60].

3. Task Execution: Tasks within Cloudflare Workflows are commonly expressed as API invocations, event handlers, or interactions with external services. Because task execution occurs within Workers, tasks inherit statelessness, idempotency, and resilience, supporting retries and compensatory logic in case of failures [60].

4. Task Queues and Event-Driven Model: Cloudflare Workflows follows an event-driven execution model rather than a traditional polling-based approach.

   (a) Tasks are triggered by events (e.g., HyperText Transport Protocol (HTTP) requests, cron schedules, or messages from Cloudflare Queues [61]) and are dispatched to Worker instances for execution [60].

   (b) The event-driven paradigm eliminates idle polling, improves resource efficiency, and supports near-real-time orchestration at scale [60].

5. Monitoring and Observability: Cloudflare provides observability mechanisms integrated into the Workers platform. Workflow executions can also be monitored through real-time logs, analytics dashboards, and third-party observability integrations (e.g., Datadog [47], Sentry [62]).

3.4 AWS Step Functions

AWS Step Functions is a serverless orchestration service that lets developers create and manage multi-step application workflows in the cloud [42]. By using the service's drag-and-drop visual editor, teams can easily assemble individual microservices into unified workflows. At each step of a given workflow, Step Functions manages input, output, error handling, and retries, so that developers can focus on higher-value business logic for their applications.

The primary architectural components of AWS Step Functions [42] are as follows:

1. State Machine Execution: Within AWS Step Functions, workflows are represented as state machines, where each step of the workflow corresponds to a distinct state.

   Figure 8 illustrates an example of a workflow defined as a state machine using ASL.

   (a) State transitions are governed by predefined conditions, thereby supporting a variety of execution patterns, including sequential flows, parallel branches, and conditional paths [63].

   (b) State machines are defined using the Amazon States Language (ASL), a Javascript Object Notation (JSON-based) specification that encodes execution logic, branching conditions, error handling strategies, and task transitions [41].

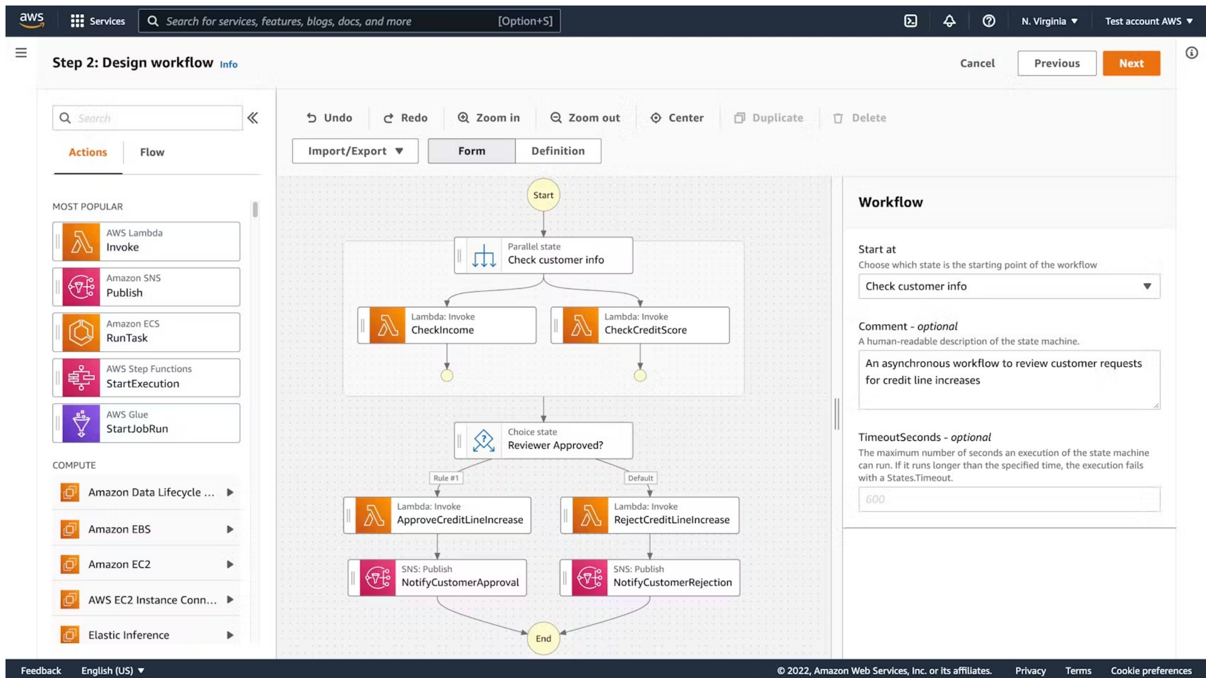


Figure 8: Designing Workflows on AWS Step Functions

2. Task States represent the fundamental execution units in AWS Step Functions. Their role and characteristics can be summarized as follows:

   (a) Each Task State corresponds to a discrete unit of work within a workflow and serves as the mechanism for invoking external actions, computational tasks, or service calls [64].

   (b) Task States ensure robust and fault-tolerant execution. Their design promotes modularity, traceability, and reliability in orchestrating complex workflows by supporting timeouts, retries, and error handling policies [64].

3. Choice, Parallel, and Wait States: In addition to Task States [65], AWS Step Functions [63] supports specialized state types that enhance workflow flexibility and efficiency:

   (a) Choice States: Enable conditional branching, allowing workflows to dynamically follow different execution paths based on runtime inputs or contextual data [66].

   (b) Parallel States: Support the concurrent execution of multiple branches, thereby increasing throughput and efficiency in large-scale batch or distributed workflows [29].

(c) Wait States: Introduce execution delays or synchronization points, often used for scheduled triggers or coordination with external events [67].

4. Execution Logging and Monitoring: AWS Step Functions [63] maintains comprehensive execution logs that capture state transitions, inputs, outputs, and error events. For observability, the service integrates natively with Amazon CloudWatch [68], supporting real-time monitoring, performance tracking, and alerting

3.5 Camunda

Camunda is an open-source workflow and decision automation platform that implements the Business Process Model and Notation (BPMN) and Decision Model and Notation (DMN) standards [7]. Unlike serverless or event-sourcing orchestrators, Camunda emphasizes model-driven workflow execution, combining machine tasks with human-centric processes. Its architecture provides flexibility for both cloud-native microservices orchestration and enterprise-scale business process automation [69].

While platforms like Temporal and Conductor emphasize programmatic or event-sourced orchestration, Camunda's model-driven and standards-compliant approach makes it particularly suitable for organizations requiring Business Process Model and Notation (BPMN) and hybrid human-machine workflows.

As shown on figure 9 at a high level, Camunda consists of the Process Engine [70], executing Business Process Model and Notation (BPMN) workflows, the Decision Engine [71] evaluating Decision Model and Notation (DMN) tables, and auxiliary modules for persistence, task management, and monitoring. Workflows are deployed as Business Process Model and Notation (BPMN) definitions, executed by the engine, and persisted in a relational database. Tasks are either executed automatically or delegated to workers or human participants. Monitoring interfaces provide visibility across all execution stages.

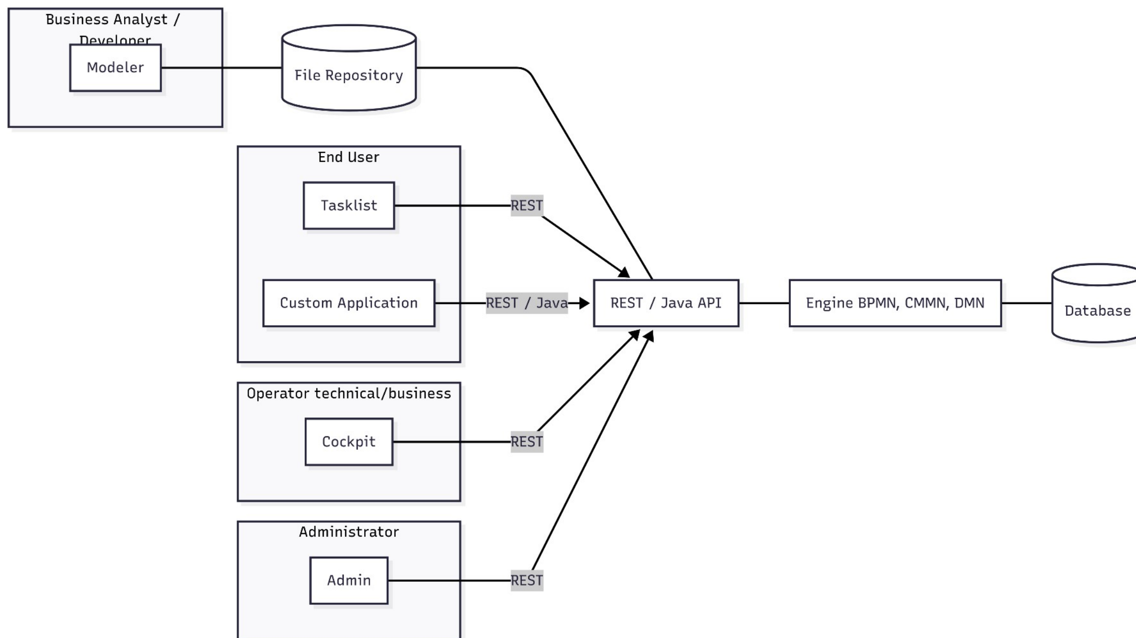


Figure 9: High-level Architecture of Camunda BPMN Engine

The primary architectural components of Camunda [7] are as follows:

1. Workflow Definitions: Workflows in Camunda are defined using the BPMN 2.0 standard, a graphical modeling notation widely adopted in industry. BPMN enables workflows to incorporate not only service tasks but also user tasks, decision points, and gateways, thereby integrating automated and human-driven processes within a unified execution model [72].

   (a) Declarative graphical models can be executed directly by the Camunda Engine .

   (b) Complex process logic, including sequential flows, parallel branches, event handling, and compensation mechanisms, can be visually specified and deployed without low-level orchestration code [73].

2. Orchestration Engine: At the core of Camunda lies the Process Engine [70], which interprets BPMN models and manages workflow execution.

   (a) The engine coordinates both automated service tasks (e.g., microservice invocations, API calls) and user tasks requiring human intervention.

   (b) Workflow execution is fully persistent, ensuring that state is maintained across restarts and system failures.

(c) The engine supports synchronous and asynchronous execution, enabling scalability across distributed applications.

3. Task Execution and Workers: In Camunda [7], workflow execution is realized through a combination of service tasks, user tasks, and external workers, each fulfilling distinct roles within the orchestration model:

   (a) Service Tasks: Invocations of external systems (e.g., REST APIs, messaging systems) executed automatically by the engine.

   (b) User Tasks: Workflow tasks requiring human input, managed through task forms or Camunda Tasklist [74], ensuring integration of people into automated processes.

   (c) Job Workers: Using the external task pattern, Camunda allows distributed workers to subscribe to specific tasks, poll for execution, and report back results, enabling integration with microservice architectures [75].

4. Persistence and State Management: Camunda workflows are fully persistent, with execution state stored in a relational database [76]. This ensures durability and the ability to recover workflows from their last known state.

   (a) Supported databases include PostgreSQL, MySQL, Oracle, and others [76].

5. Monitoring and Observability: Camunda provides operational monitoring and visualization tools:

   (a) Cockpit: An administrative console for monitoring live and historical process executions, debugging workflows, and inspecting task states [77].

   (b) Optimize: An advanced analytics tool for tracking performance metrics and bottlenecks through dashboards [78].

   (c) Tasklist: A user interface (UI) for end-users to manage and complete assigned user tasks [79].

Chapter 4 Conceptual Architecture

The preceding chapters established the theoretical foundations of distributed workflows, surveyed the state of the art in orchestration platforms, and identified ongoing research and industry challenges. This chapter presents the design of the system developed in this dissertation, which seeks to operationalize the insights gained from the literature into a concrete architecture tailored to gaming monetization.

The design follows a set of guiding principles derived directly from the analytical framework and research threads: (i) durability of execution through workflow-as-code, (ii) developer ergonomics and observability to facilitate maintainability, (iii) secure and auditable integration with financial infrastructures, and (iv) extensibility to support evolving business requirements and workflows.

Designing a backend platform involves a multitude of technical decisions ranging from how to structure the codebase and define API contracts, to selecting deployment strategies and configuring infrastructure. While many of these decisions may appear subjective or driven by developer preference, such variability often leads to fragmentation across systems. When every project follows its own conventions, supporting tools must be overly generalized, leading to increased complexity and reduced efficiency. In this project, deliberate efforts were made to enforce architectural consistency, promote reusable patterns, and minimize unnecessary divergence between services.

The architecture is anchored in the Cloudflare Cloud platform, which provides a globally distributed execution environment with built-in primitives for compute, storage, orchestration, and communication. Cloudflare's edge network [27] allows services to run close to users and external financial infrastructures, minimizing latency while preserving consistency through centralized workflow coordination. Its integration of Workers [80], Workflows [80], D1 [81], and Queues [61] offers a cohesive substrate for building resilient financial workflows without managing servers or bespoke infrastructure.

The proposed architecture adopts a layered approach in which a workflow orchestration engine sits at the center, surrounded by persistence and observability services, and extended outward through integrations with payment processors. The aim is to keep interactive paths fast and predictable while ensuring that multi-step processes complete reliably even when

they depend on external actors or require human action. Each layer has a clearly defined responsibility and communicates with its neighbors through constrained, versioned contracts so that the system can evolve without widespread rewrites.

Cloudflare Workflows [4] serves as the orchestration core. It expresses long-lived, replayable state transitions that may span minutes or days, coordinating retries with backoff, timed waits, and compensations where necessary. Workflows [16] are intentionally thin in computation and heavy in control: they decide what should happen next and persist those decisions, delegating side-effects to domain services. This separation allows the platform to recover deterministically from failures and to resume at the precise step where progress last stopped, which is essential for processes such as identity verification, instrument provisioning, payments, and payouts.

Stripe [1] provides the financial backbone for both payments and payouts. It offers access to global payment rails, enforces regulatory obligations, and emits webhooks that describe the authoritative outcome of financial events. Within this architecture, Stripe [1] is treated as an external source of truth whose notifications are verified, recorded, and reconciled with local state. Rather than assume success based on outbound calls alone, the system advances records only when a Stripe [1] event confirms forward progress, which reduces ambiguity and simplifies audits.

Cloudflare D1 [81] is the relational store for transactional data. It holds canonical records for resources, along with operational ledgers such as idempotency keys, webhook receipts, and an append-only event trail. Writes are coordinated to a primary region to preserve correctness, while replicated reads deliver low-latency access elsewhere. The data model favors immutability for monetary facts, explicit status transitions for lifecycles, and durable auditability so that any outcome can be reconstructed without relying on volatile logs.

Cloudflare Workers [11] operate at the network edge [82] as the ingress and egress boundary for APIs and webhooks. They authenticate and authorize requests, validate inputs, and perform only the synchronous work necessary to return a definitive response. When a call requires external confirmation or multi-step coordination, a Worker [28] records a succinct state change in D1 [81], starts or resumes a Workflow [80], and acknowledges promptly. Workers [80] also verify inbound webhooks before handing them off for durable processing, keeping the public edge responsive even during traffic spikes.

Cloudflare Queues [61] act as the event bus for publishing workflow state updates. A Worker [28] can publish a challenge message that carries the information a client needs to complete authentication, and the eventual resolution is represented as a resume signal that re-enters the waiting workflow. Delivery is at-least-once by construction, so consumers are idempotent and messages that repeatedly fail are moved to a dead-letter stream for inspection and safe replay.

4.1 Architecture Interface

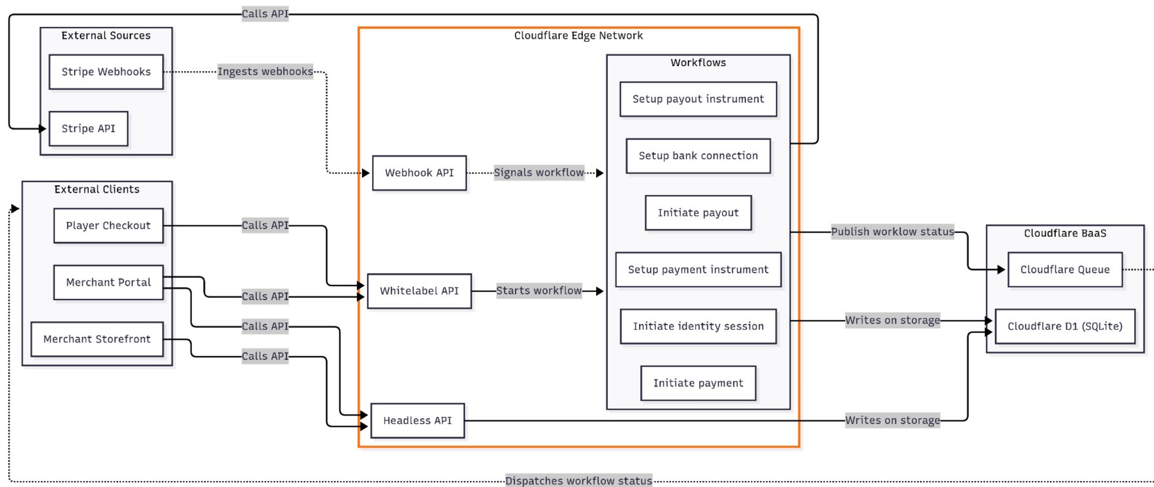


Figure 10: Interface(s) between the system being developed and other systems

Figure 10 presents a reference architecture for payment orchestration that combines edge-resident request handling with event-driven coordination and managed backend services. The system is partitioned into four logical zones: (i) External Sources, which comprise third-party payment infrastructure (e.g., Stripe Webhooks and the Stripe API); (ii) External Clients, including end-user and merchant-facing applications such as Player Checkout, Merchant Portal, and Merchant Storefront; (iii) the Cloudflare Edge Network [82], which hosts ingress endpoints and the workflow runtime; and (iv) Cloudflare Backend as a Service (BaaS) [9], which provides durable messaging and storage including Cloudflare Queue [61] and Cloudflare D1 [81].

All ingress paths converge on a Workflows [80] service that executes the domain logic as explicit, named processes: *setup payout instrument*, *setup bank connection*, *initiate payout*, *setup payment instrument*, *initiate identity session*, and *initiate payment*. Two trigger modalities are distinguished: the Whitelabel API starts workflows (synchronous initiation tied

to client requests), while the Webhook API signals workflows (asynchronous advancement of pre-existing state machines).

Taking by example the use case where a player initiates a payment through the checkout page, the request is first handled by Cloudflare Workers [28], from this point, the request enters Cloudflare Workflows [80], which form the backbone of the transaction lifecycle. Workflows represent a shift from traditional synchronous request-response models. Unlike typical serverless functions, which terminate after a single execution, Cloudflare Workflows [80] are asynchronous and durable. They are designed to coordinate long-running processes, handle retries automatically, and maintain state across failures. This makes them particularly well suited to financial operations, where each step of a transaction must be reliable and traceable.

However, the asynchronous nature of Workflows introduces a challenge. Because they do not return results immediately in the way a synchronous API might, the system cannot rely on a conventional request-response pattern to update the user interface. To address this, Cloudflare Queues [38] were incorporated into the architecture. Queues act as a publishing channel through which the Workflow can send updates at each stage of execution. For example, when a workflow validates input, contacts the payment provider, receives authorization, or writes data to the database, it publishes a corresponding message into a queue. These messages are then consumed by Workers, which push updates back to the merchant portal or checkout page.

The integration with the payment provider is another critical component. Workflows communicate directly with this external service to handle the actual payment authorization and settlement process. Once a response success, decline, or error is received, the workflow processes it, logs the outcome, and publishes an appropriate status update via Queues so that the UI reflects the latest state.

Persistent data is stored in Cloudflare D1 [81], which functions as the system's serverless SQL database. D1 records all transaction logs, workflow metadata, and merchant-related data. This database not only provides consistent storage and historical information.

The overall flow of the system can therefore be understood as a tightly integrated loop. A player initiates a payment on the checkout page, which is routed through Workers into a durable workflow. The workflow manages the payment lifecycle, interacting with the provider, logging information to D1 [81], and publishing asynchronous status updates into Queues [61]. Workers then consume these messages and propagate updates to both the player-

facing checkout page and the merchant-facing portal. As a result, users on both sides of the platform are able to follow the progress of each transaction in real time, without being constrained by the asynchronous execution model of Cloudflare Workflows [80].

In conclusion, the architecture is an example of how modern serverless technologies can be combined to create a payment processing platform that is both scalable and reliable. By leveraging Cloudflare's global edge network [82] and integrating with established payment infrastructures, the platform achieves the performance, resilience, and trustworthiness required for mission-critical financial applications.

4.2 Workflow Skeleton

We standardize all workflows on a single, edge-orchestrated template. Concretely, every workflow must be implemented as an idempotent state machine that (i) invokes a provider through a dedicated adapter, (ii) persists the returned context and storage actions before any control-flow decision, (iii) branches on *finality* (early return on *OK* or *FAILED*), and (iv) if non-final (*PROCESSING*), concurrently awaits a webhook and performs bounded-interval polling using the previously persisted context.

Upon reaching a terminal state, the workflow must publish a status event to the message queue for asynchronous consumption. This template ensures safety (durable, idempotent progression), liveness (event-driven or time-driven completion), and observability (uniform status publication), and it decouples orchestration from provider integration, thereby improving reliability and evolvability across heterogeneous providers.

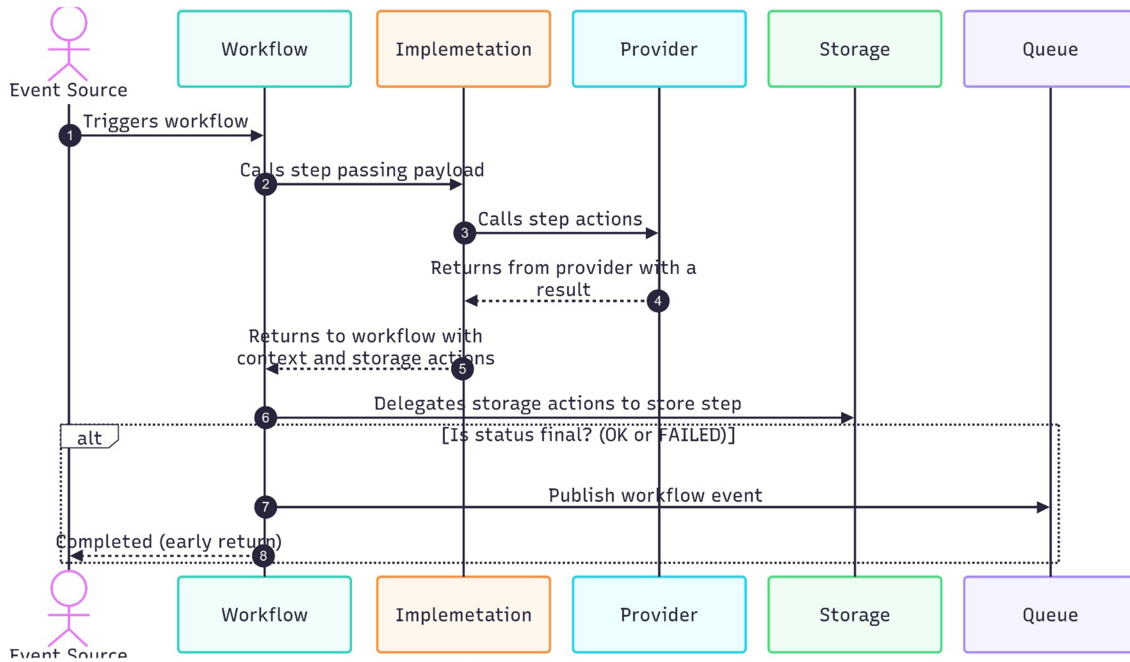


Figure 11: Workflow Skeleton (Part 1)

As show in *figure 11* illustrates a skeleton workflow where execution begins when the Event Source [39] triggers the Workflow, which synchronously invokes the Implementation with an initial payload.

The Implementation issues "step actions" to the Provider and returns a tuple comprising the provider context (status, correlation identifiers, and any opaque tokens required for subsequent calls) and a set of storage actions. The Workflow delegates these writes to Storage before making any further control-flow decision, thereby establishing the invariant that every externally observed state transition is durably recorded.

The second *figure 12* models two alternative mechanisms: the Webhook path and the Polling path, through which the workflow continues to interact with the provider and manage its state. These paths are executed in parallel, though the system ultimately converges back into a unified sequence of operations.

1. Webhook Path: in the webhook-based strategy, the workflow remains in a waiting state until it receives an asynchronous notification. This notification is delivered directly from the provider in the form of a webhook payload.

(a) Upon receiving the payload, the workflow delegates its processing to the implementation component, which interprets the payload, updates the contextual information, and determines any required storage actions.

(b) The workflow then executes those storage operations by communicating with the storage system.

2. Polling Path: By contrast, the polling strategy adopts a proactive model. The workflow begins by entering a predefined sleep interval (ten minutes, in this case) before initiating its first query.

    (a) It then repeatedly invokes the implementation with the previously stored execution context. The implementation interacts with the provider to request updated results. If the provider indicates that the operation is not yet in a final state, the implementation passes this intermediate result back to the workflow, which again delegates the corresponding storage operations.

    (b) The workflow then re-enters its sleep interval before repeating the polling process. This cycle continues iteratively until the provider signals a final status (either success or failure).

3. Convergence: Once either the webhook-based or polling-based path yields a final result, the workflow publishes a completion event to the queue. This event serves as a signal to downstream systems or subscribers. Finally, the workflow communicates the completion status back to the original caller, marking the end of execution.

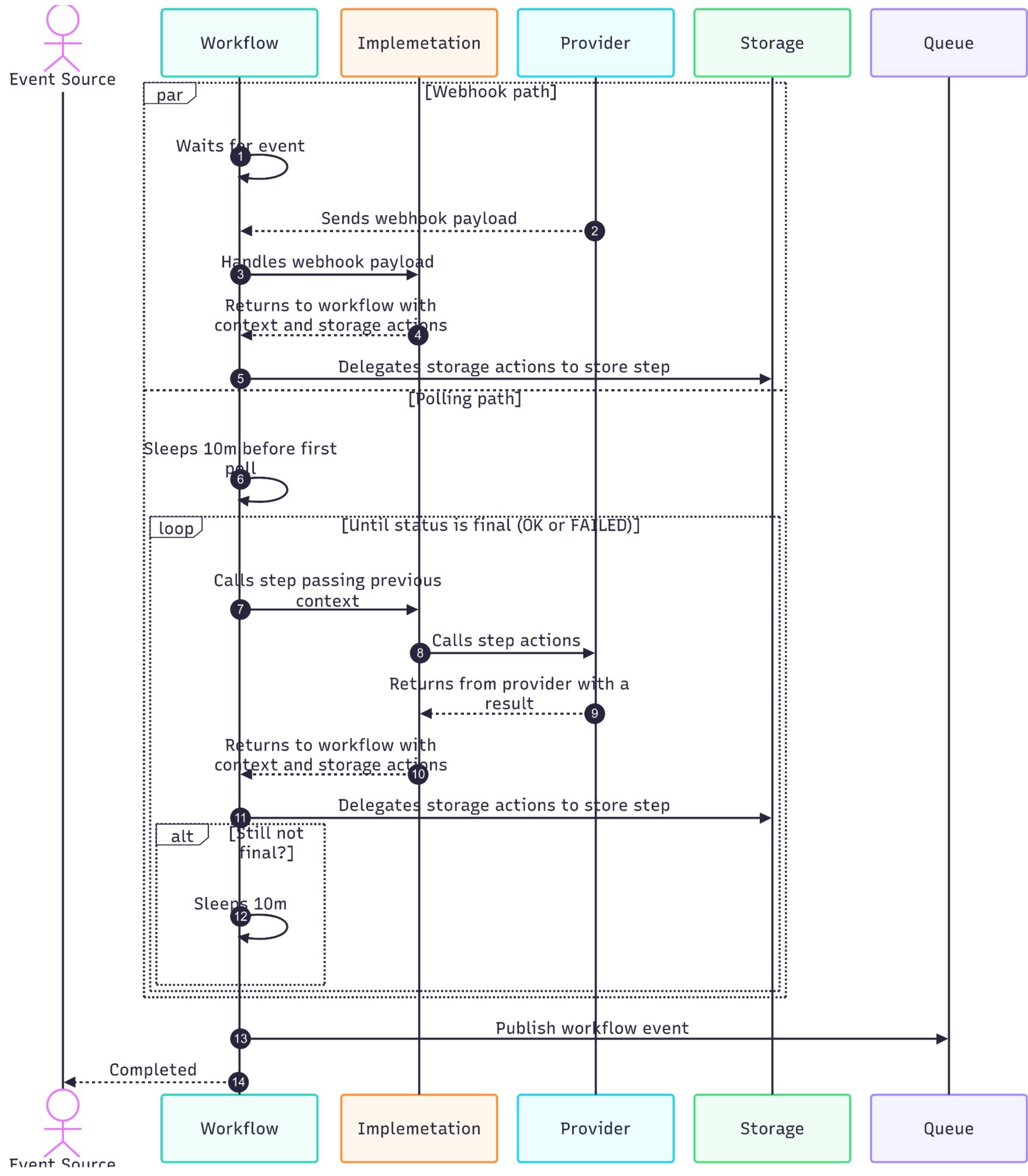


Figure 12: Workflow Skeleton (Part 2)

4.3 Relational Storage

The storage design underpins the reliability, scalability, and compliance of the platform. Built around the entity-relationship model shown in *figure 13*, it encodes the interaction between merchants, players, and financial entities, forming the foundation of the logical schema. Each entity maps to a real-world concept, while the relationships enforce business rules, ensure regulatory traceability, and support high-volume operations.

The schema is centered on the Merchant entity, representing a legally accountable party. Merchants hold critical attributes (tax identification number, politically exposed person, know your customer fields) that satisfy regulatory obligations. Each merchant is tied to exactly one Bank Connection, which acts as the verified financial gateway to external providers. From this point extend the key financial structures:

1. Balances, representing multi-currency holdings under the bank connection.
2. Payout Instruments (e.g., debit cards), which define the channels for moving funds out.
3. Payouts, which record the actual disbursement of funds through these instruments.
4. Identity Sessions, ensuring that every merchant interaction is authenticated and auditable.

On the operational side, the diagram models the Player entity, which represents end-users engaging with merchant services. Players authenticate through OAuth Clients, ensuring all activity is verified. They may own multiple Payment Instruments (e.g., debit cards), each of which can process many Payments.

This layered flow between *OAuth Client, Player, Payment Instrument, Payment* ensures traceability from the point of authentication to the execution of individual transactions. Payments themselves are also tied back to the Bank Connection, closing the loop between player activity and merchant financial structures.

Integrations and extensions. The design includes Remote Console (RCON) clients, which represent server-side integrations (e.g., game or service endpoints). Though independent, they are associated with merchants via shops in the operational model, ensuring that technical control remains tied to commercial entities.

The schema adheres to the First Normal Form (1NF) to the Third Normal Form (3NF) normalization principles:

1. Attributes are atomic (e.g., payments record amount, currency, timestamps).
2. Dependencies are tied to primary keys (e.g., a payment's depends solely on its unique id).
3. Redundancy is eliminated (e.g., merchant details are stored only once, not copied into bank connections).

Normalization ensures consistency and compliance, while the relationships encoded as foreign keys with cascading operations prevent orphaned or invalid records. For example, deleting a merchant automatically cascades through its bank connection, balances, and payouts.

PAYOUT		
string	UUID	PK
string	PayoutInstrumentId	FK
string	Currency	
number	Amount	
string	Provider	
string	Reference	
string	Status	
timestamp	CreatedAt	
timestamp	UpdatedAt	

has

PAYOUT_INSTRUMENT		
string	UUID	PK
boolean	CVV	
boolean	AVS	
number	ExpirationMonth	
number	ExpirationYear	
number	LastFour	
boolean	Instant	
string	Provider	
string	Reference	
string	Status	
timestamp	CreatedAt	
timestamp	UpdatedAt	

PAYMENT		
string	UUID	PK
string	PlayerID	FK
string	PaymentInstrumentID	FK
string	Currency	
number	Amount	
string	Provider	
string	Reference	
string	Status	
timestamp	CreatedAt	
timestamp	UpdatedAt	

IDENTITY_SESSION		
string	UUID	PK
string	MerchantId	FK
string	BankConnectionId	FK
string	Provider	
string	Reference	
string	Status	
timestamp	CreatedAt	
timestamp	UpdatedAt	

BALANCE		
string	UUID	PK
number	Amount	
string	Currency	
timestamp	CreatedAt	
timestamp	UpdatedAt	

uses has has has charges

BANK_CONNECTION		
string	UUID	PK
string	MerchantId	FK
string	Provider	
string	Reference	
string	Status	
timestamp	CreatedAt	
timestamp	UpdatedAt	

PAYMENT_INSTRUMENT		
string	UUID	PK
string	PlayerID	FK
boolean	CVV	
boolean	AVS	
number	ExpirationMonth	
number	ExpirationYear	
number	LastFour	
string	Provider	
string	Reference	
string	Status	
timestamp	CreatedAt	
timestamp	UpdatedAt	

OAUTH_CLIENTS		
string	UUID	PK
string	Label	
string	Thumbnail	
string	AuthorizationURL	
string	TokenURL	
string	ProfileURL	
string[]	Scopes	
timestamp	CreatedAt	
timestamp	UpdatedAt	

initiates starts

has owns has

PLAYER		
string	UUID	PK
string	Reference	
timestamp	CreatedAt	
timestamp	UpdatedAt	

RCON_CLIENTS		
string	UUID	PK
string	Label	
string	Thumbnail	
string	Hostname	
number	Port	
string	Secret	
timestamp	CreatedAt	
timestamp	UpdatedAt	

MERCHANT		
string	UUID	PK
string	TIN	
boolean	PEP	
string	DOB	
string	AddressLine1	
string	AddressLine2	
string	ZIP	
string	Country	
string	State	
timestamp	CreatedAt	
timestamp	UpdatedAt	

Figure 13: Storage Entity-Relationship Diagram

4.4 Design patterns

This system is built for real-world messiness: network calls fail, webhooks arrive late or twice, and workflows may be replayed after a crash or deploy. Rather than trying to prevent those conditions, the design makes them harmless. Each pattern below keeps effects behind durable boundaries, advances state only when it's truly forward progress, and makes time explicit through deadlines, backoff, and timestamps. When the same work is attempted again because of a retry, a duplicate trigger, or replay the observable outcome is the same.

The patterns work together. Steps are idempotent so the same request cannot change the world twice. Early-return gates stop a workflow as soon as its record is already in a terminal state, which makes replays cheap. Database writes are prepared and committed as a single batch so the "state change" and its audit entries never drift apart. Notifications use an outbox so the decision and the message share one fate. For liveness, the workflow prefers events (e.g., Stripe [1] webhooks) but will fall back to timed polling after a grace window; retries use exponential backoff with per-attempt timeouts so the platform waits cheaply rather than spinning.

What matters is that each pattern has clear preconditions and guarantees. If a proposal does not move the canonical record forward, it is ignored. If a step already succeeded, subsequent attempts short-circuit to the same result. If a dependency is down, the system sleeps durably and tries again later. The result is predictable behavior under stress without special-case code paths.

4.4.1 Retry and timeout policy for workflow steps

The retry and timeout configuration of workflow steps defines how the orchestrator behaves when external dependencies are slow or unavailable. Conceptually, each step is given unlimited retries, an initial delay of five seconds, an exponential backoff strategy, and a one-minute per-attempt time budget. This policy ensures that transient failures are absorbed by the workflow engine rather than exposed as user-visible errors.

In practice, the orchestrator attempts the step once and waits up to a minute for the dependency to respond. If that window elapses or the call fails, the orchestrator schedules another attempt for a later point in time. The first retry is scheduled five seconds after the failure, with each subsequent delay growing exponentially until the dependency eventually

recovers. Because these delays are implemented as durable sleeps, they do not consume active compute cycles and place no additional load on the dependency while it is struggling. Instead, the workflow effectively pauses itself, resuming execution only when the next retry window arrives.

This policy is particularly well-suited to financial operations where many steps are expected to succeed eventually, even if temporary errors occur. Examples include finalizing a charge, posting a payout instruction, or confirming the setup of a payment instrument. In such cases, the orchestrator's retry strategy converts short-lived incidents into minor delays, ensuring that the workflow continues to make forward progress without compromising correctness.

```
12
13  /**
14   * Workflow step config.
15   */
16  const workflowStepConfig: WorkflowStepConfig = {
17    retries: {
18      limit: Infinity,
19      delay: '5 seconds',
20      backoff: 'exponential',
21    },
22    timeout: '1 minute',
23  }
24
```

Figure 14: Workflow step config

The configuration object that encodes this policy is illustrated in figure 14, where retries are defined with an infinite limit, a fixed initial delay, exponential backoff, and a per-attempt timeout. A concrete example of the policy in action can be seen in 15, where an identity session webhook is executed as a step with the retry configuration applied. This demonstrates how each step carries its own resilience rules, guaranteeing that the workflow as a whole remains robust even in the presence of unreliable external systems.

```
48  const identitySessionResult = await step.do(
49    'Handle identity session webhook',
50    workflowStepConfig,
51    async () ⇒ await implementation.handleWebhook(webhook.payload, params, context)
52  )
```

Figure 15: Workflow step

4.4.2 Event-first and polling fallback with race condition

In distributed workflows that rely on third-party providers, the arrival of a signal can be uncertain: sometimes an event will be pushed via a webhook in near-real time, while in other cases the system must proactively poll the provider to obtain an updated status. Designing for both possibilities is essential when reliability and correctness are critical.

The pattern adopted here combines an event-driven path with a timed polling path and treats them as a conceptual race. Whichever path produces a definitive outcome first commits it to the workflow, while the other path becomes a harmless no-op thanks to idempotent and monotonic state transitions. This ensures that duplicate updates or late arrivals cannot corrupt the final state.

The event-first branch waits for a provider signal up to a defined grace window. This makes the workflow highly responsive under ideal conditions, since the provider typically knows the result as soon as it is available. To implement this, the workflow suspends its execution, awaiting a webhook that matches the correct provider notification kind (in this case, a bank connection event).

Figure 16 illustrates the event-first strategy. The handler captures the webhook payload, passes it to the domain implementation for processing, and persists the resulting state. If the webhook never arrives within the timeout, the workflow gracefully transitions to the fallback mechanism.

```
35  async function handleWebhook(
36    implementation: StripeImplementation,
37    step: WorkflowStep,
38    params: WorkflowParams,
39    context: WorkflowContext
40  ) {
41    const webhook = await step.waitForEvent<Stripe.Account>('Wait for bank connection webhook', {
42      type: ProviderNotificationKind.BANK_CONNECTION,
43      timeout: '5 minute',
44    })
45    const bankConnectionResult = await step.do(
46      'Handle bank connection webhook',
47      workflowStepConfig,
48      async () => await implementation.handleWebhook(webhook.payload, params, context)
49    )
50
51    await step.do(
52      'Store bank connection result',
53      workflowStepConfig,
54      async () => await store(bankConnectionResult.storageActions)
55    )
56
57    return { workflowContext: { ...context } }
58  }
```

Figure 16: Workflow handling webhook

While the event-first path is ideal, systems cannot rely exclusively on external actors behaving correctly. Some providers may drop events, deliver them late, or fail to notify altogether. For this reason, a timed polling path ensures that the workflow eventually converges to the correct state.

In the polling branch, the system repeatedly queries the provider at fixed intervals until the connection reaches a final status (such as ok or failed). After each poll, intermediate results are stored, enabling progress to be tracked transparently and without data loss.

Figure 17 shows the polling loop. Note how each iteration refreshes the workflow context, persists the current state, and pauses for a short period before retrying. If the status is not final, the loop continues, effectively guaranteeing eventual progress even if no webhook is ever delivered.

```
63  async function pollBankConnection(
64    implementation: StripeImplementation,
65    step: WorkflowStep,
66    params: WorkflowParams,
67    context: WorkflowContext
68  ): Promise<{ workflowContext: WorkflowContext }> {
69    await step.sleep('Waiting webhook before polling', '10 minutes')
70
71    while (!isFinalBankConnectionStatus(context.status)) {
72      const bankConnectionResult = await step.do(
73        'Polling bank connection',
74        workflowStepConfig,
75        async () => await implementation.getBankConnection(params, context)
76      )
77
78      context.status = bankConnectionResult.workflowContext.status
79
80      await step.do(
81        'Store bank connection result',
82        workflowStepConfig,
83        async () => await store(bankConnectionResult.storageActions)
84      )
85
86      if (!isFinalBankConnectionStatus(context.status)) {
87        await step.sleep('Still waiting for webhook before finishing', '10 minutes')
88      }
89    }
90
91    return { workflowContext: { ...context } }
92  }
```

Figure 17: Workflow polling

The two branches are then combined into a race. By launching both concurrently, the workflow obtains the best of both worlds: low-latency responsiveness if the webhook arrives quickly, and high-reliability fallback if it does not.

The orchestration is implemented via a promise race, which resolves as soon as either branch completes. The result of the “winning” branch is treated as authoritative, while the “losing” branch is harmlessly discarded. Thanks to the use of idempotent persistence and monotonic state transitions, redundant or delayed updates cannot overwrite or regress the committed outcome. Figure 18 depicts the race condition orchestration. The event-driven handler and polling routine are executed side by side, with the first to produce a definitive result determining the prepared statements outcome.

```
140  const bankConnectionResult = await Promise.race([
141    handleWebhook(
142      implementation,
143      step,
144      event.payload,
145      createBankConnectionResult.workflowContext
146    ),
147    pollBankConnection(
148      implementation,
149      step,
150      event.payload,
151      createBankConnectionResult.workflowContext
152    ),
153  ])
```

Figure 18: Workflow race condition

This composition balances responsiveness with reliability. The event-first path minimizes latency and network overhead in the common case, while the polling path provides robustness against unreliable providers. By explicitly modeling the two as a race, the system avoids deadlocks and ensures liveness i.e., the workflow will always make progress even if one path fails entirely.

Another key property is safety: because the workflow state machine is designed to be idempotent, duplicate or redundant state writes cause no harm. This allows both branches to run without complex coordination, simplifying implementation while preserving correctness.

Finally, this design reflects a broader architectural principle: whenever external signals may be unreliable, pairing an optimistic, event-driven mechanism with a pessimistic, polling fallback provides a practical balance between performance and fault-tolerance.

4.4.3 Early-return gating for replayable workflows

By workflow replay we mean the deterministic re-execution of a workflow's control logic from durable history after a crash, deployment, or operator re-drive. When a workflow is replayed, the orchestrator reconstructs all prior decisions by replaying the recorded outcomes of steps, and then resumes evaluation from that history. Correctness in this model depends on two guarantees: first, that all externally visible effects are mediated through steps whose results are stored durably; and second, that the control flow surrounding those steps is deterministic with respect to the recorded history.

The early-return gating pattern provides a simple way of giving replay a fixed-point semantics. Each run of the workflow begins by materializing or refreshing the canonical record of the subject in the authoritative store. Immediately afterwards, the workflow evaluates a terminal predicate over that record. If the predicate holds that is, if the record already reflects a terminal state then the workflow emits any required terminal notification and returns immediately.

This order of operations is crucial. Because the durable write comes first, followed by the predicate check, and only then by any further effects, re-execution from history always yields the same observation and the same decision. In other words, once the workflow has reached a terminal state, subsequent evaluations will converge to the same outcome without producing additional side effects. The early-return gate therefore enforces idempotency by construction.

In more formal terms, the terminal state of the record becomes a fixed point of the workflow's transition function. Once reached, every replay or re-drive of the workflow simply re-evaluates the predicate, observes that the fixed point has been attained, and exits cleanly. This ensures convergence under replay and protects the workflow against duplicated effects, even when re-executed multiple times.

```
116
117     await step.do(
118       'Store identity session result',
119       workflowStepConfig,
120       async () => await store(createIdentitySessionResult.storageActions)
121     )      You, 6 days ago • chore: hard fork workflows
122
123     if (isFinalIdentitySessionStatus(createIdentitySessionResult.workflowContext.status)) {
124 >     console.info('Setup identity session workflow completed', {⋯
129       })
130
131       await step.do(
132         'Publishing identity session status',
133         workflowStepConfig,
134         async () =>
135           await implementation.publishIdentitySessionStatus(
136             event.payload,
137             createIdentitySessionResult.workflowContext
138           )
139       )
140
141       return
142     }
143
```

Figure 19: Early-return workflow

The application of this pattern to the identity session workflow is illustrated in figure 19. The figure shows how the workflow first stores the identity session result, then evaluates whether the session has reached its final state, and only if necessary proceeds to publish updates. If the session is already terminal, the workflow logs completion and returns immediately, making subsequent replays safe and deterministic.

4.4.4 Batch prepared SQL statements

One of the key challenges in designing financial workflows is ensuring that every important step is durably recorded. Each interaction with an external provider, such as Stripe [1], or each state transition in a workflow must be captured in storage so that the system remains auditable and can recover from interruptions without losing context. To meet this requirement, the platform relies on Cloudflare D1, a globally distributed SQL database. Instead of issuing individual SQL commands for every operation, the system uses prepared SQL statements that can be executed in batches, providing both efficiency and reliability.

Prepared statements are generated at runtime with all the necessary parameters, such as references, statuses, and error codes. For example, after a bank connection is created, the

workflow prepares an insert statement to log the provider call and an update statement to record the connection's current status. These statements are not executed immediately. Instead, they are collected into an array of storage actions that will later be passed into the persistence layer as a group. This design provides a clear separation between workflow logic and persistence logic, ensuring that the workflow can build up a full set of required operations before committing them to storage. The logic for constructing such storage actions can be seen in the implementation shown in figure 20.

```
197        return {
198          workflowContext: {
199            ...context,
200            error: bankConnection.error,
201            status: bankConnection.status,
202          },
203          storageActions: [
204            await this.providerCallStorage.prepareInsertProviderCall({
205              reference: params.reference,
206              provider: Provider.STRIPE,
207              request: { id: context.reference },
208              response: accountResponse as unknown as Record<string, unknown>,
209              kind: ProviderCallKind.GET_VERIFICATION_SESSION,
210              status: accountProviderCallStatus,
211            }),
212            await this.bankConnectionStorage.prepareUpdateBankConnection(
213              {
214                error: bankConnection.error,        You, 3 days ago • chore: http
215                status: bankConnection.status,
216              },
217              { id: params.reference }
218            ),
219          ],
220        }
221      }
```

Figure 20: Batch prepared SQL statements (workflow activity)

The batching itself is handled by the D1 storage API. When the workflow is ready to persist its changes, the storage actions are sent to the database in a single batched request. Internally, the system executes all prepared statements together and checks the results of each. If any of the statements fail, the error is captured, logged, and raised back into the workflow. This guarantees that the workflow never advances while the storage layer is in an inconsistent

state. Because workflows in Cloudflare are durable by design, a failed persistence step triggers a retry, and the batch is attempted again until it succeeds. In this way, persistence failures do not result in lost records or corrupted state. The function responsible for executing these batches is illustrated in figure 21.

```
import { env } from "cloudflare:workers";

/**
 * Store.
 */
export async function store(statements: Array<D1PreparedStatement>) {
  const results = await env.D1_STORAGE.batch(statements);

  for (const result of results) {
    if (result.error) {
      console.error("Failed to store statement", {
        error: result.error,
        meta: result.meta,
      });

      throw new Error("Failed to store statement");
    }
  }
}
```

Figure 21: Internals of store step

The integration between workflows and storage is seamless. At the end of each workflow step, the prepared statements are wrapped into a storage action array. When the workflow executes the step, it also invokes the store function, which submits the batch to D1. For example, after the creation of a bank connection, the workflow returns both the updated workflow context and the prepared storage actions. The next step then executes the store function, which runs all the statements as a batch and validates the results. This ensures that both the application logic and the persistence layer stay synchronized. A typical example of this integration, where workflow steps call the store function with prepared storage actions, is shown in figure 22.

```
114  await step.do(
115    'Store bank connection result',
116    workflowStepConfig,
117    async () => await store(createBankConnectionResult.storageActions)
118  )
```

Figure 22: Store workflow step

The benefit of this approach becomes especially clear in financial operations, where data consistency is critical. By batching prepared statements, the platform reduces the number of database calls, lowers latency, and ensures that related operations are persisted together rather than individually. If a provider call is logged, its outcome is stored at the same time as the update to the bank connection record. This grouping of operations prevents partial persistence, where some parts of a transaction might be written while others are lost.

Error handling is also made stronger through batching. Since every result from the batch is checked, the workflow is immediately aware if a persistence operation has failed. Rather than silently ignoring errors, the system surfaces them, logs detailed diagnostic information, and retries the step. This makes the storage layer resilient to transient errors, such as network interruptions or temporary database unavailability, while maintaining the durability guarantees expected of financial systems.

In practice, the use of batch prepared SQL statements has proven to be a fundamental component of the architecture. It combines efficiency, since multiple writes are grouped together, with reliability, since every step is checked and retried if needed. Most importantly, it provides confidence that every significant workflow action is permanently recorded, whether that is an authorization request, a payout initiation, or the setup of a new payment instrument.

4.4.5 Proxied steps factory

A key architectural decision in the design of the workflow engine is the introduction of a proxied steps factory. The goal of this component is to abstract away the differences between payment providers and expose a unified interface for workflows. Instead of embedding provider-specific logic directly into workflows, the system delegates that responsibility to a provider implementation, which is created dynamically by a factory. This makes the

workflows both simpler and more portable, as they always interact with a consistent set of steps, regardless of the underlying provider.

The process begins with the definition of a provider interface. This interface specifies the operations that any provider must implement, such as initiating a payout, fetching the status of a payout, handling webhooks, and publishing payout status updates. Each method returns both the updated workflow context and a set of prepared SQL storage actions, ensuring that every provider call can be durably recorded in D1. The Provider Interface establishes the contract that concrete implementations must follow, and it guarantees consistency across providers. An excerpt of this interface is shown in figure 23.

```
8   interface ProviderInterface {
9     initiatePayout(params: WorkflowParams): Promise<{
10      workflowContext: WorkflowContext
11      storageActions: Array<D1PreparedStatement>
12    }>
13
14    getPayout(
15      params: WorkflowParams,
16      context: WorkflowContext
17    ): Promise<{
18      workflowContext: WorkflowContext
19      storageActions: Array<D1PreparedStatement>
20    }>
21
22    handleWebhook(
23      webhookBody: Stripe.Payout,
24      params: WorkflowParams,
25      context: WorkflowContext
26    ): Promise<{
27      workflowContext: WorkflowContext
28      storageActions: Array<D1PreparedStatement>
29    }>
30
31    publishPayoutStatus(params: WorkflowParams, context: WorkflowContext): Promise<void>
32  }
```

Figure 23: Workflow provider interface

Concrete implementations of the interface are created in the steps factory. The factory takes a provider identifier as input and returns an object that maps each workflow step to the corresponding provider method. For example, when Stripe [1] is selected as the provider, the factory instantiates an implementation, binds its methods to the interface contract, and exposes them back to the workflow as callable steps. If an unsupported provider is requested,

the factory raises an error, preventing the system from executing undefined behavior. The implementation of the steps factory is shown in figure 24.

```
34  export class StepsFactory {
35    static createSteps(provider: Provider) {
36      let impl: ProviderInterface
37
38      if (provider === Provider.STRIPE) {
39        impl = new StripeImplementation(env)
40      } else {
41        throw new Error(`Unsupported provider`)
42      }
43
44      return {
45        initiatePayout: impl.initiatePayout.bind(impl),
46        store,
47        getPayout: impl.getPayout.bind(impl),
48        handleWebhook: impl.handleWebhook.bind(impl),
49        publishPayoutStatus: impl.publishPayoutStatus.bind(impl),
50      }
51    }
52  }
53
```

Figure 24: Workflow steps factory

This design has several advantages. First, it encapsulates provider-specific logic within its own implementation, keeping the workflow definitions clean and provider-agnostic. Second, it allows new providers to be added without modifying existing workflows. As long as the new provider implements the same interface, the workflows can operate against it seamlessly. Third, it integrates naturally with the persistence layer, since each provider method is responsible for returning both its results and the prepared storage actions that need to be persisted.

4.5 Workflows

Workflows provide the orchestration substrate that sequences side-effecting operations under explicit step boundaries, uniform retry policies, and deterministic replay. A workflow is defined as a series of named steps. Each step encapsulates a single unit of intent, executes

with a bounded timeout, and is retried according to a declared backoff strategy. Between steps, the system records both intent and result, ensuring that replays drive the same transitions without duplicating effects. This structure externalises failure management from business logic: steps are retried by configuration, not by ad hoc code, and long-running operations progress through well-defined states.

At the perimeter of execution, the workflow admits two classes of progression: event-driven and time-driven. Event-driven progression reacts to external signals delivered via a messaging or webhook channel. Time-driven progression uses sleeping delays and periodic polling to advance when external parties are slow or silent. The two modes coexist within the same workflow and are composed explicitly. A typical pattern is to initiate an operation, publish the resulting challenge to downstream consumers, and then race an event wait against a poll loop. If the external signal arrives, the workflow processes it and commits the resulting state changes; if it does not, the workflow polls at configured intervals until a terminal state is observed. This hybrid approach embraces the realities of at-least-once delivery and variable third-party latency without sacrificing determinism.

Each step conforms to the "prepare many, commit once" discipline. The step body performs only pure computation and provider interaction necessary to determine the next state, returning a set of prepared storage mutations rather than executing them inline. At the step boundary, the runtime commits the batch atomically. Consequently, the only mutating operation per step is a single, transactional commit controlled by the step's timeout and retry policy. This design yields strong invariants: either all intended writes for the step become visible together, or none do; upstream behaviour remains referentially transparent under replay; and audits can attribute durable changes to specific, named steps.

Long-running workflows commonly require a decision about completion. The orchestration therefore distinguishes between provisional and terminal statuses, where terminal statuses are explicitly enumerated and checked before any additional waiting or polling occurs. If a step concludes in a terminal status, the workflow publishes the outcome and exits. If not, the workflow publishes an intermediate challenge or status update, then proceeds to await either the external signal or the culmination of polling. Publication itself is a step-scoped side effect executed under the same idempotent commit discipline, ensuring that notifications are not duplicated across retries.

Operational concerns are captured declaratively. Retry limits, base delays, and exponential backoff are configured per step rather than embedded in logic, allowing uniform behaviour and easy tuning. Timeouts bound resource use and provide clear failure semantics. Sleeping delays are named and parameterized to make waiting explicit and observable. Because these behaviors are part of the workflow definition rather than its business code, they are stable under change and consistently applied across heterogeneous operations.

Deterministic identification ties together requests, workflow instances, and aggregates. Instance identifiers are derived from stable inputs to enable idempotent creation and unambiguous correlation. Parameter payloads are closed over all inputs necessary for downstream execution, preventing dependence on mutable external state and making replays reproducible. When combined with step-level idempotency and atomic commits, this determinism ensures that duplicate invocations converge on the same outcome or produce no additional effect.

Finally, observability is integral to the model. Each step runs with a human-readable name, emits structured logs at start and completion, and produces storage actions that form the durable audit trail. The runtime's record of intents and results, the atomic commit boundaries, and the explicit waits and sleeps together yield a transparent execution history. In aggregate, these properties explicit step boundaries, declarative retries and timeouts, hybrid event/poll progression, atomic state transitions, deterministic identification, and rich observability constitute a workflow model that is robust to failure, predictable under replay, and adaptable to the asymmetric and unreliable nature of external integrations.

4.5.1 Setup bank connection workflow

The workflow for establishing a bank connection has been designed as a structured orchestration process in which the workflow engine coordinates the interactions between internal services and an external provider. The purpose of this design is to ensure reliability, fault tolerance, and responsiveness in the execution of a process that is inherently asynchronous and dependent on third-party systems. The sequence of operations is depicted in figure 25, which illustrates the main participants namely the workflow orchestrator, the worker abstraction, the external provider, and the storage system and their interactions across different phases of the process.

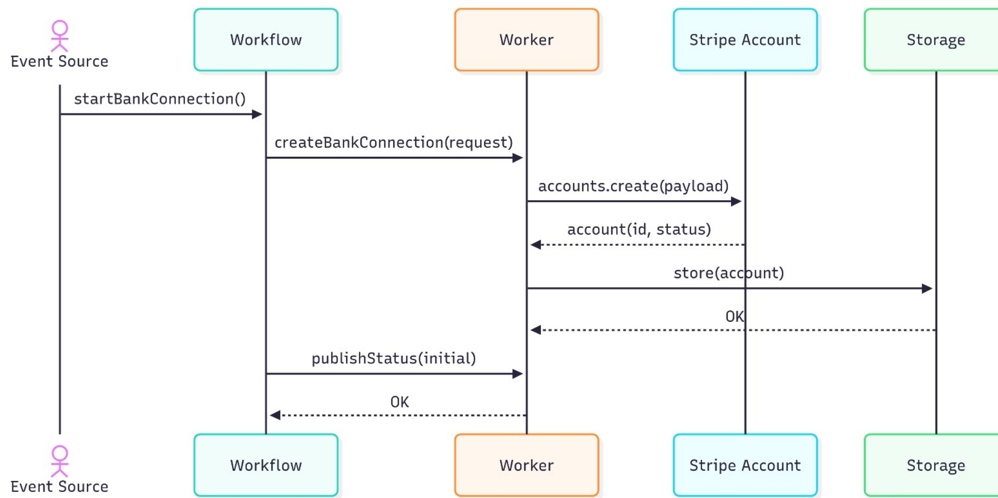


Figure 25: Setup bank connection workflow (Part 1)

The workflow begins by instructing the worker to initiate the creation of a bank connection with the external provider. This step is not executed directly by the orchestrator but is instead wrapped in a controlled execution context that applies retry and timeout strategies, logs structured metadata, and ensures idempotency. The response returned by the provider is represented as a compound object that includes the current status of the connection as well as a plan describing how the resulting state should be durably recorded. This separation of concerns ensures that orchestration remains deterministic while leaving the specific persistence logic to be executed in a subsequent stage.

Following the creation of the bank connection, the workflow commits the returned result to persistent storage. As shown in figure 25, the persistence operation is delegated to the storage system via the worker. By storing the state immediately after creation, the workflow guarantees that it can recover and resume execution in the event of failure. This persistence mechanism also enforces idempotency by ensuring that repeated invocations of the same workflow step do not generate conflicting or duplicated records. In practical terms, persistence is crucial for maintaining the integrity of the workflow state across restarts and for supporting exactly-once semantics in downstream communication.

Once the state has been persisted, the workflow checks whether the bank connection has reached a final status. Final states include successful completion, cancellation, or failure. If the state is indeed terminal, the workflow follows the fast-path branch illustrated in the upper

portion of *figure 26*. In this branch, the workflow records a structured completion log and disseminates the final status to dependent systems through a publication step. This enables immediate termination of the workflow without unnecessary delays, thereby optimizing efficiency for cases in which the provider produces a final outcome at the moment of creation.

If the connection has not yet reached a final state, the workflow proceeds into a concurrency phase that is illustrated in the lower section of *figure 26*. In this phase, the orchestrator employs a dual strategy for status resolution that combines event-driven and polling-based mechanisms. On one hand, a webhook handler listens for asynchronous notifications from the provider, which can deliver updated connection statuses. On the other hand, a polling routine periodically queries the provider for updates. These two mechanisms operate concurrently, and the workflow adopts whichever result is received first. This design guarantees that progress is made even if one mechanism fails: webhook latency or absence is compensated by polling, while polling delays are minimized when webhook notifications arrive promptly.

Once a definitive status update has been obtained, either via webhook or polling, the workflow records this update in the storage system and proceeds to publish the result to downstream consumers. The publication stage is carried out in a reliable manner, ensuring that subscribers receive the final state exactly once, regardless of whether the workflow has been retried or resumed after failure. In this way, the workflow provides strong delivery guarantees while maintaining correctness in the face of distributed system uncertainties.

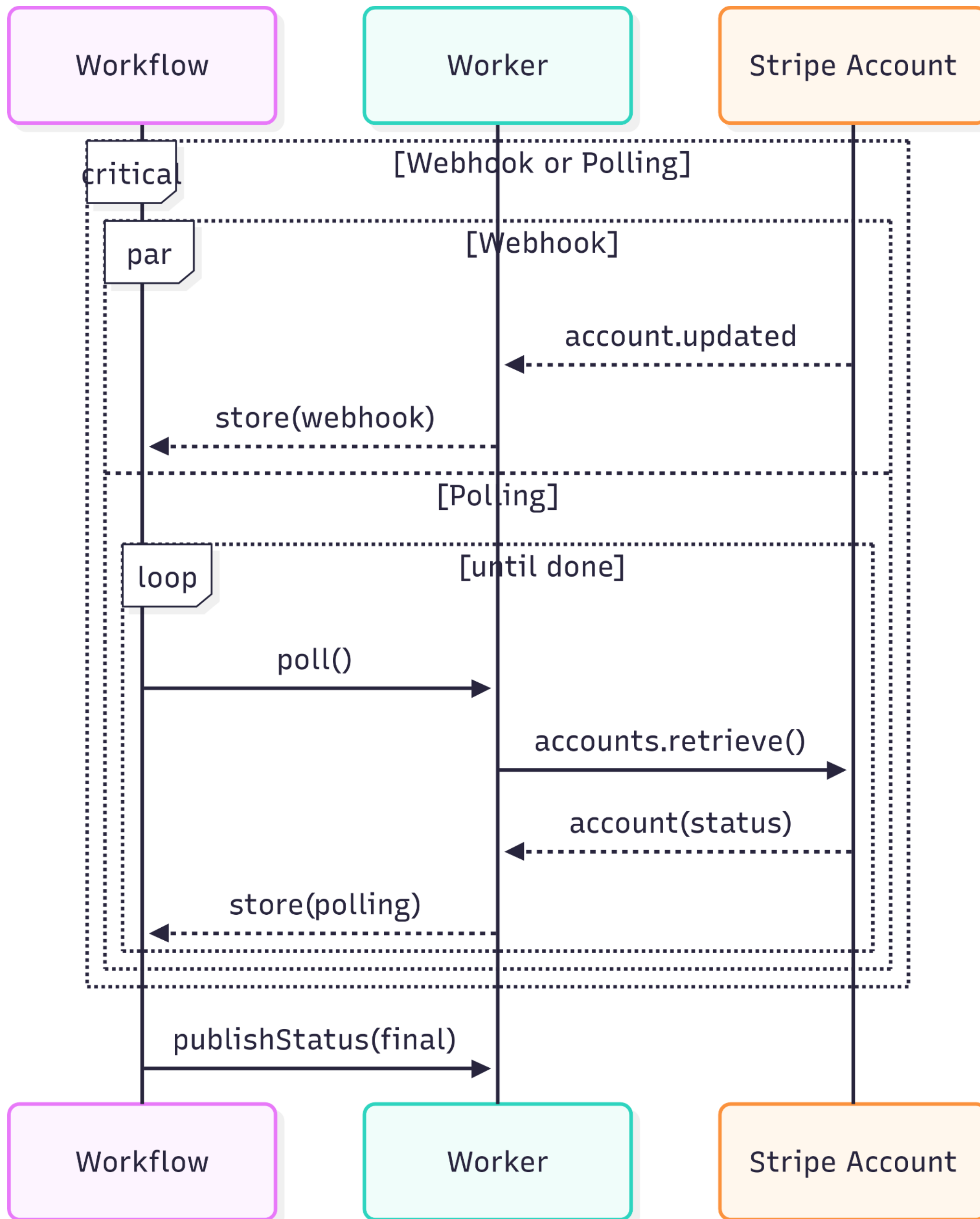


Figure 26: Setup bank connection workflow (Part 2)

4.5.2 Initiate identity session workflow

The identity session workflow is designed to coordinate the interaction between the workflow orchestrator, the external identity provider, e.g. Stripe Identity [83], and the persistence layer. Its purpose is to manage the creation, monitoring, and finalization of identity verification sessions in a manner that is both reliable and resilient to failures. *Figures 27 and 28* presents the sequence of interactions, showing the conditional branches and concurrent mechanisms that ensure timely completion of the process.

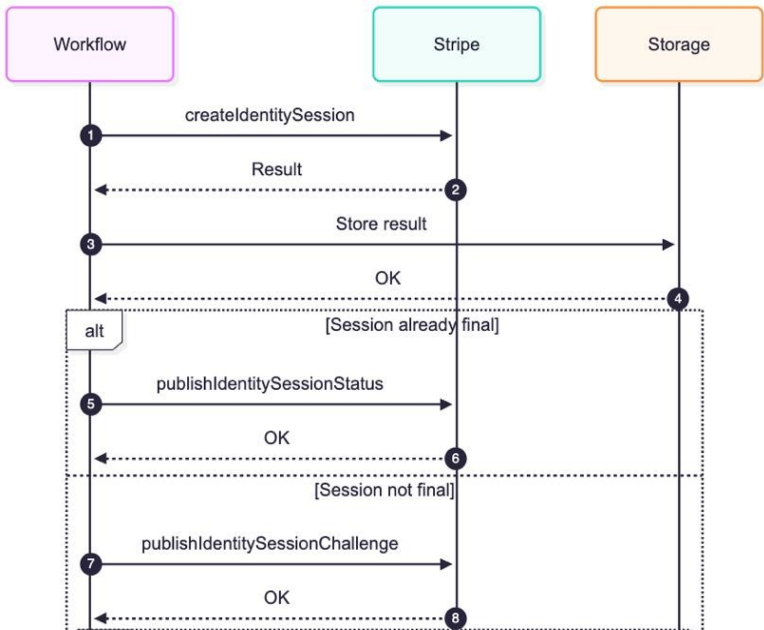


Figure 27: Initiate identity session workflow (create)

The process begins with the workflow initiating a request to create a new identity session with the external provider. The provider returns a result object that contains the initial status of the verification session, together with metadata needed for subsequent processing. This result is immediately persisted in the storage system. Storing the initial state at this point guarantees recoverability in case of interruptions and ensures that the workflow can be resumed without loss of progress.

Once the session has been created and stored, the workflow determines whether the verification session is already in a terminal state. Terminal states typically include outcomes

such as successful verification or failure. If the session is already terminal, the workflow takes the fast-path branch: it publishes the final status back to the provider and logs the completion of the workflow. This path represents the most efficient outcome, since no further interaction is required.

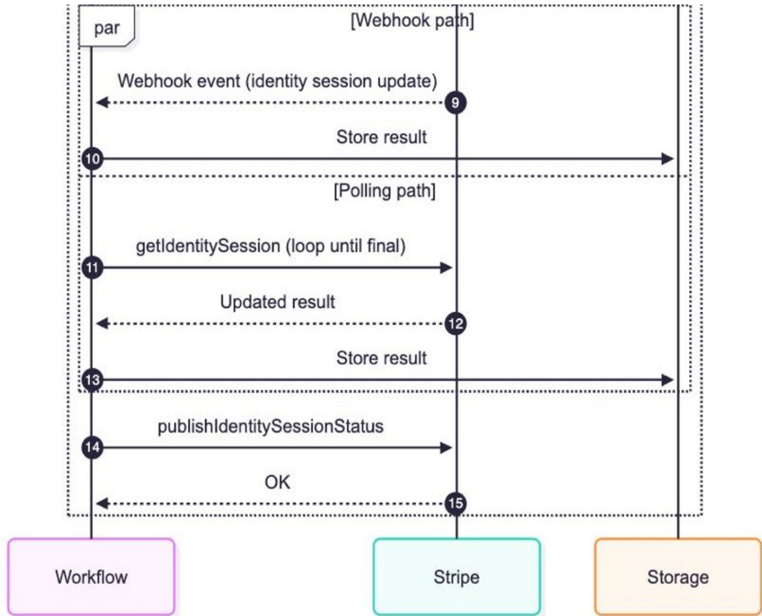


Figure 28: Initiate identity session workflow (polling + webhook)

If the session is not in a terminal state, the workflow instead publishes a challenge. This challenge represents additional verification actions required from the user, such as document upload or biometric confirmation. Publishing the challenge ensures that the user is notified and can proceed with the required steps, while the workflow transitions into a waiting phase.

In the waiting phase, the workflow adopts a dual strategy for resolving the identity session status. As shown in *figure 28*, this involves two concurrent paths:

1. Webhook path: The provider may proactively send a notification when the identity session has progressed. The workflow receives this webhook, processes the updated result, and persists the new status in storage.
2. Polling path: In parallel, the workflow periodically queries the provider to check whether the identity session has reached a terminal state. Each update received through polling is stored in the same way, and the cycle continues until a final result is observed.

The use of both mechanisms concurrently ensures resilience. If webhooks are delayed or lost, polling guarantees eventual progress. Conversely, if polling intervals are long, webhook delivery minimizes latency by providing updates as soon as they occur. The workflow accepts the first definitive result received and disregards subsequent duplicate updates.

Once a terminal status has been reached whether through webhook notification or polling the workflow concludes by publishing the final session status to the provider. This step ensures that downstream systems receive an authoritative record of the outcome. The result is also persisted, enabling auditability and supporting recovery if needed. A structured completion log is generated, including identifiers, merchant information, and any errors, which provides observability for monitoring and debugging purposes.

4.5.3 Setup payment instrument workflow

The setup payment instrument workflow manages the process of registering a new payment instrument, such as a card or other funding source, and tracking its status until it is finalized. This ensures that players can securely add and verify their payment methods before initiating transactions.

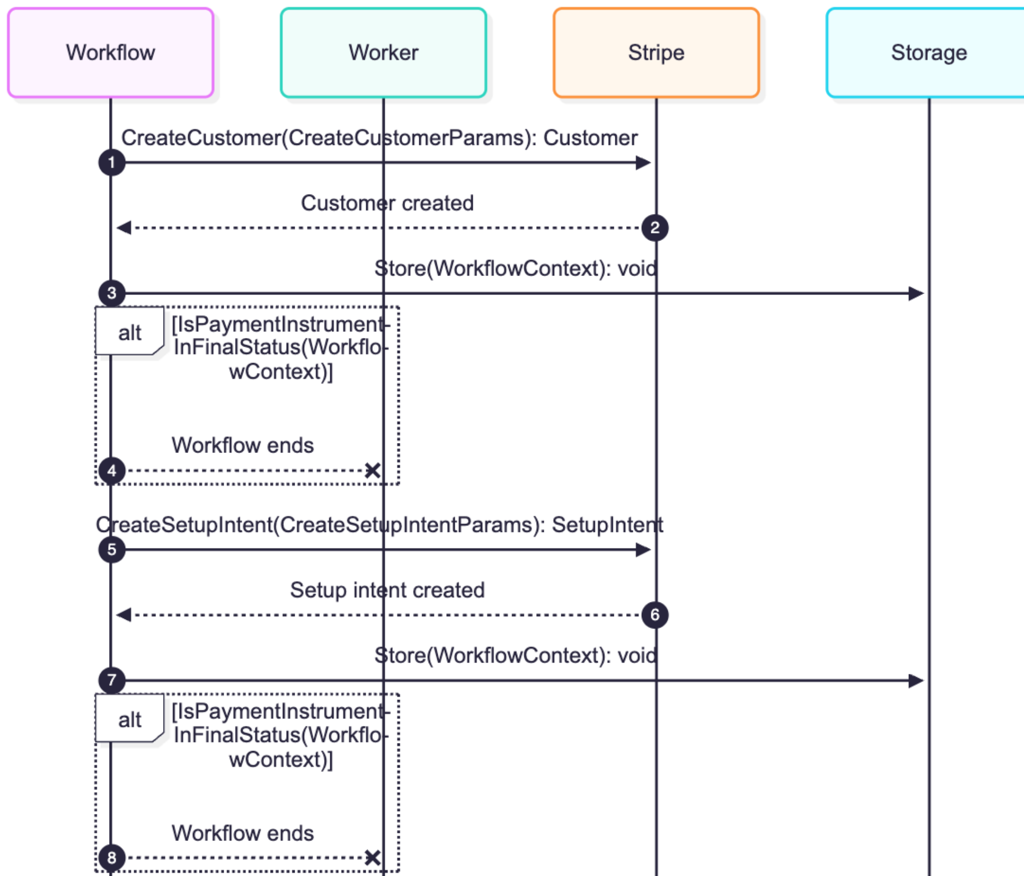


Figure 29: Setup payment instrument workflow (setup intent)

The workflow begins with the creation of a customer record. The workflow invokes a worker, which calls Stripe's API [1] with the necessary parameters to create a customer object. Stripe responds by confirming that the customer has been created successfully. This result is stored in the storage layer, ensuring that the customer identifier and context are persisted for later steps. After storing, the workflow checks whether the payment instrument is already in a final state. If so, the workflow ends immediately; otherwise, it proceeds to the next phase.

The next step is to create a setup intent for the payment instrument. The workflow again calls a worker, which requests Stripe [1]to create a Setup Intent using the provided parameters. Stripe [1] responds with confirmation that the Setup Intent has been created, which indicates that the instrument is now in the process of being validated and confirmed. As with the customer creation step, the workflow stores the Setup Intent details in the storage layer. The workflow then checks once more whether the payment instrument has reached a final state. If it has, the process ends here.

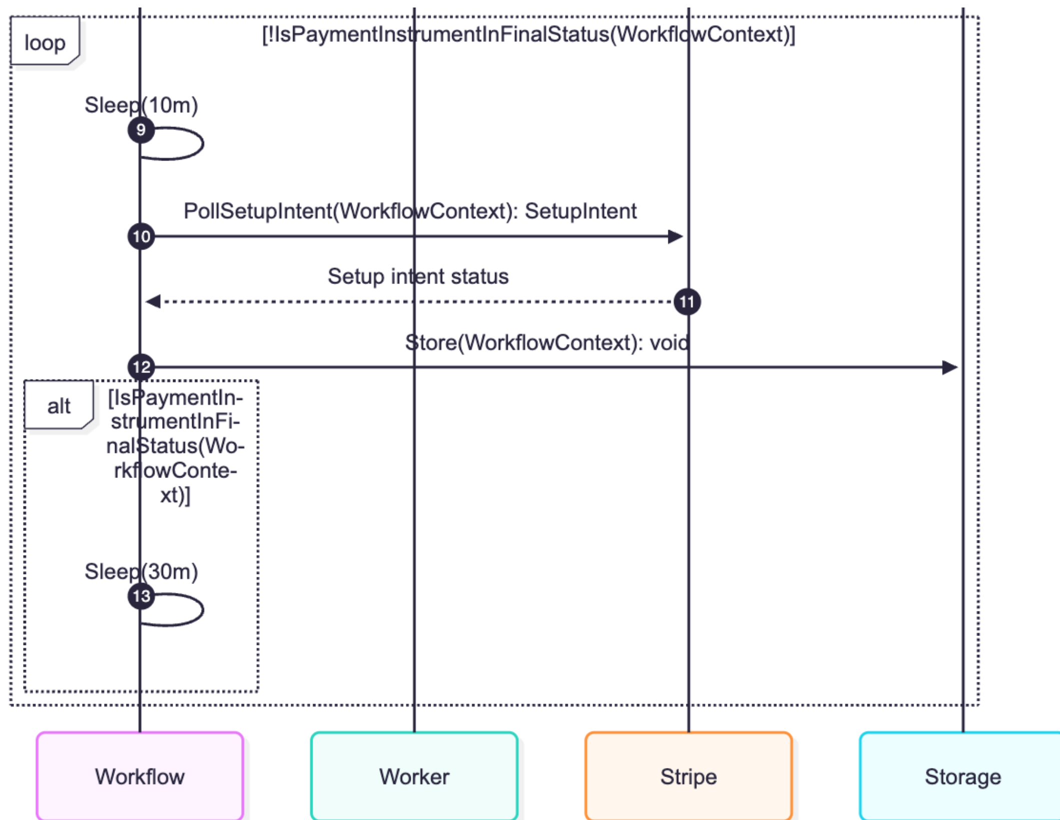


Figure 30: Setup payment instrument (polling)

If the Setup Intent is not yet finalized, the workflow enters a monitoring loop. The Workflow suspends execution for ten minutes before polling Stripe for the current status of the Setup Intent. Stripe returns the latest status, which the workflow records in storage. The system checks whether the instrument has reached a terminal state, such as successfully confirmed or failed. If not, the Workflow continues by sleeping again, this time for thirty minutes, before polling Stripe [1]once more. This loop repeats until the provider responds with a final status.

By persisting state at each step and using polling to track progress, the Setup Payment Instrument Workflow ensures that new instruments are reliably registered and validated. This design accounts for the asynchronous nature of payment provider operations, while providing durability and resilience in the face of delays or interruptions.

4.5.4 Initiate payment workflow

The initiate payment workflow is responsible for managing the full lifecycle of a payment, from authorization to capture, and eventually to confirmation of its final state. Because payment processing can involve multiple asynchronous steps and interactions with external systems, the workflow is designed to persist state after each critical operation and to continue polling until the transaction outcome is definitive.

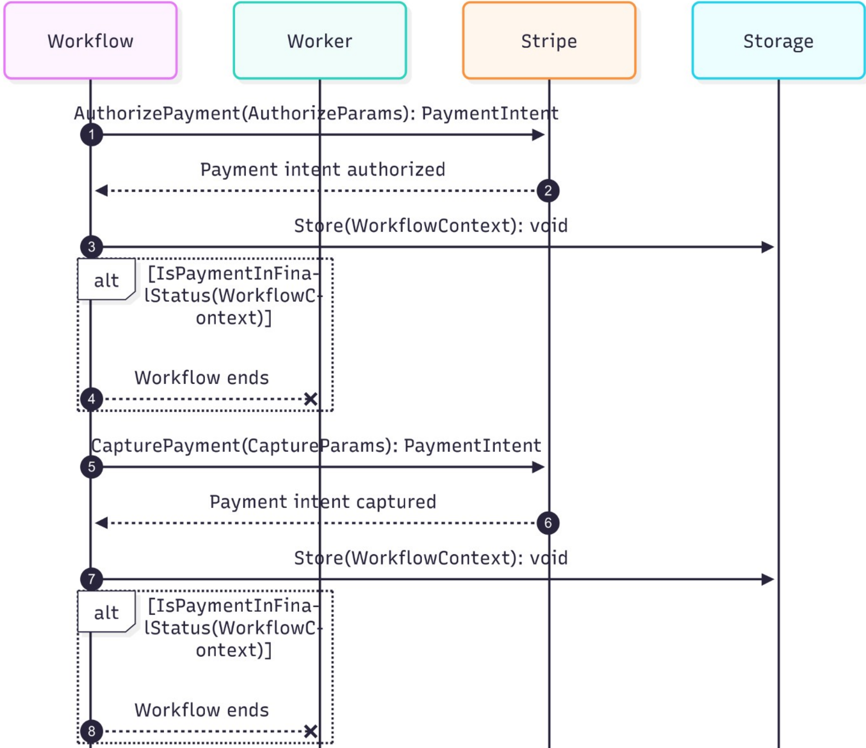


Figure 31: Initiate payment workflow (authorization and capture)

The workflow begins by sending an authorization request. The Workflow invokes a Worker, which forwards the request to Stripe [1] with the necessary parameters. Stripe responds by creating and authorizing a payment intent. Once this confirmation is received, the workflow stores the payment intent details in the storage layer, ensuring that the transaction state is check pointed and can be recovered at any point. At this stage, the workflow checks if the payment has already reached a final state. In some cases, such as when a payment is declined immediately, the workflow can end here.

If the payment is not yet finalized, the workflow proceeds to the capture phase. The Workflow issues a capture request through the Worker to the payment provider. Stripe [1] processes the capture and returns confirmation that the payment intent has been captured. As with the authorization step, the workflow records this information in the storage layer. Another check is then performed to determine if the payment has entered a terminal state. If so, the workflow terminates successfully. If not, the workflow transitions into the monitoring phase.

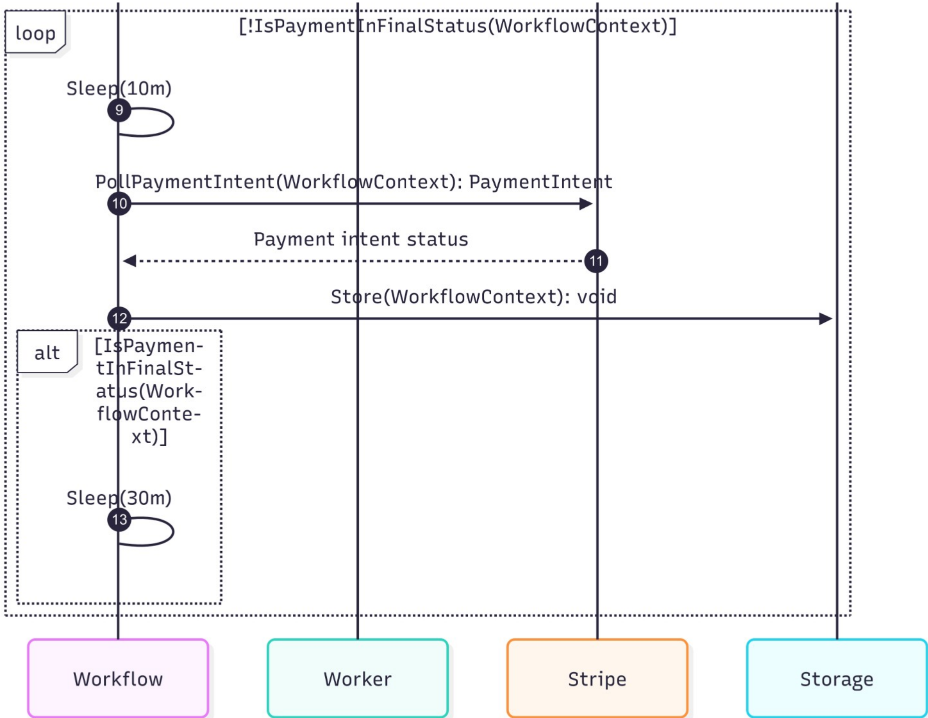


Figure 32: Initiate payment workflow (polling payment status)

The polling phase addresses the fact that not all payments settle immediately. External factors, such as banking systems, fraud detection, or asynchronous checks, may leave a payment in a pending state. To handle this, the workflow enters a loop. It first pauses execution for ten minutes before sending a poll request to Stripe [1] via the Worker. Stripe responds with the latest status of the payment intent, which is then stored. The workflow evaluates whether the payment is now in a final state. If it is, the workflow ends. If not, the workflow sleeps again, this time for thirty minutes, before polling Stripe once more. This cycle continues until the payment reaches a definitive conclusion, whether success, failure, or cancellation.

By persisting every significant step and leveraging polling loops, the Initiate Payment Workflow ensures that no payment is left unresolved. The workflow is resilient to failures, capable of resuming after interruptions, and reliable in tracking payments through to their final state. This design guarantees that merchants and players always have access to the most accurate status of their transactions, despite the inherent asynchrony of payment networks.

4.5.5 Setup payout instrument

In addition to managing incoming payments from players, the platform also requires the ability to configure and validate payout instruments for merchants. This process ensures that funds can be securely and reliably transferred from the payment provider to the merchant's external financial account, such as a bank account or card. Because payouts involve multiple asynchronous operations and dependencies on third-party services, the system implements a dedicated workflow known as the setup payout instrument workflow.

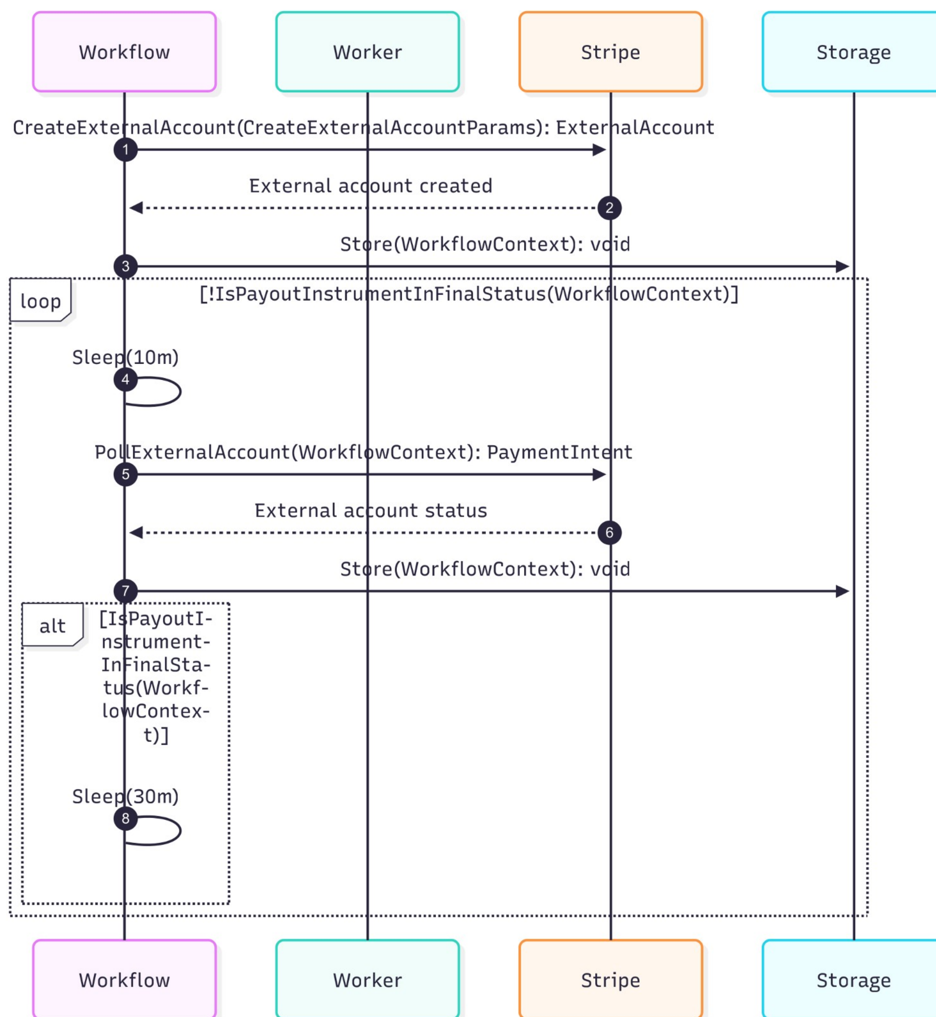


Figure 33: Setup payout instrument workflow

The workflow begins when a request to create a new external account is initiated. This request is triggered within the Cloudflare Workflow engine, which invokes the Worker layer to call the payment provider's API. In the illustrated case, the payment provider is represented by Stripe [1]. The request contains the parameters required to register the external account, such as account identifiers, banking details, or compliance metadata. Stripe [1] then processes the request and responds with confirmation that the external account has been created. This marks the first checkpoint in the workflow.

Once the external account is successfully established, the workflow proceeds to record the information. Using the context provided by the workflow execution environment, the system stores a reference to the account within the storage layer, ensuring that the platform maintains an authoritative record of payout instruments. This persistent storage step is essential, as it guarantees that subsequent stages of the workflow are tied to the correct account identifier and that the system remains auditable.

The workflow then enters a polling loop, which reflects the asynchronous nature of payout setup. Because the account creation process may not be immediately finalized or approved, the workflow must wait until the instrument reaches a terminal state. To achieve this, the workflow pauses execution using a sleep interval, in this case ten minutes, before attempting to poll the payment provider again. This approach balances responsiveness with resource efficiency: by waiting between checks, the system avoids unnecessary load on the provider's API while still progressing toward final confirmation.

At each iteration of the loop, the workflow sends a request to Stripe [1] to retrieve the current status of the external account. Stripe [1] responds with updated status information, which is then stored once again in the system's storage layer through the Worker. This repeated cycle of polling and storing ensures that the platform maintains an up-to-date view of the payout instrument's state.

The workflow includes a conditional branch to handle cases where the payout instrument has not yet reached a final status. If the account is still in a pending or intermediate state, the workflow continues to sleep for longer intervals extending to thirty minutes in later cycles before repeating the polling process. This staged backoff strategy prevents unnecessary churn while still ensuring that eventual completion is detected.

When the payout instrument finally reaches a terminal state, whether approved, rejected, or requiring remediation, the workflow exits the polling loop. At this point, the final status is stored in the system, making it accessible to both the merchant portal and backend reconciliation services.

4.5.6 Initiate payout workflow

The initiate payout workflow is designed to handle the process of transferring funds from the platform to a merchant's registered payout instrument, such as a bank account. It ensures that

payouts are only initiated when sufficient funds are available and that their status is tracked until the transaction reaches completion.

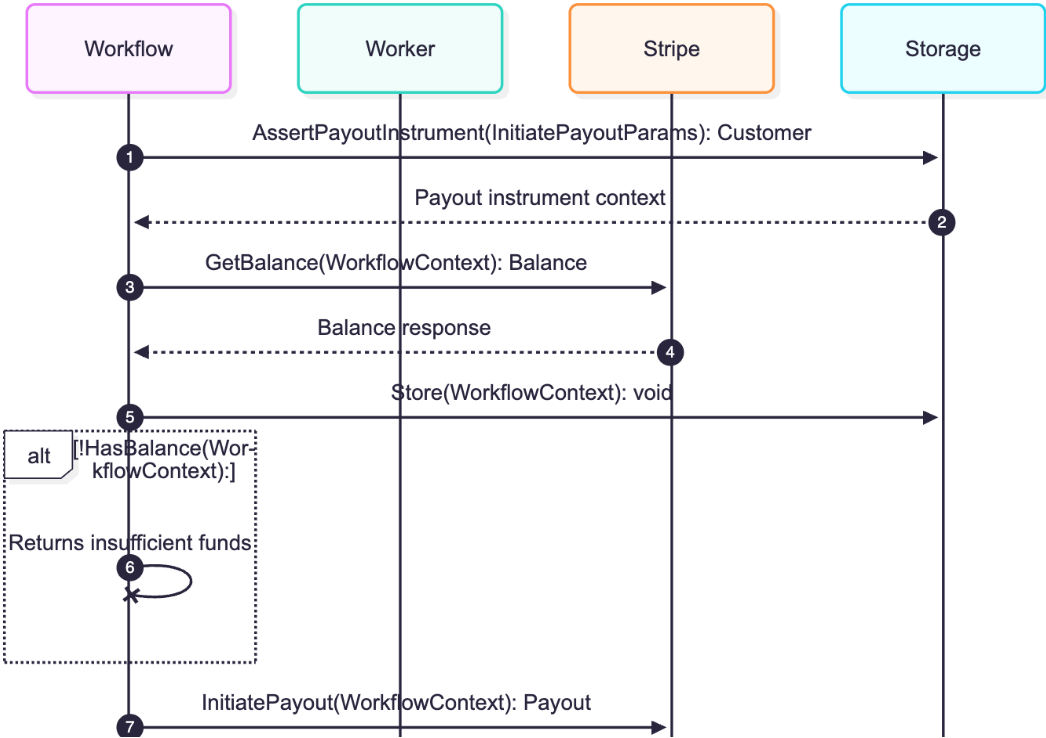


Figure 34: Initiate payout workflow (assert balance + initiate payout)

The workflow starts by asserting the payout instrument. The Workflow calls a Worker, which communicates with Stripe [1] to validate and retrieve the payout instrument context. Stripe [1] responds with the details of the instrument, which are passed back to the workflow. This step confirms that the merchant has a valid payout account on file.

Next, the workflow issues a balance check. Using the context of the payout request, it asks Stripe [1] to provide the merchant's balance. Stripe [1] responds with the balance details, and the workflow stores this information in the storage layer. The system then evaluates whether the available balance is sufficient to cover the requested payout. If the balance is too low, the workflow immediately terminates and returns an "insufficient funds" result.

If there are enough funds, the workflow proceeds to initiate the payout. The Workflow invokes a Worker, which submits the payout request to the provider. Stripe [1] processes this and begins the transfer of funds to the merchant's external account. Because this step may

take time to complete, the workflow does not end immediately; instead, it transitions into a monitoring phase.

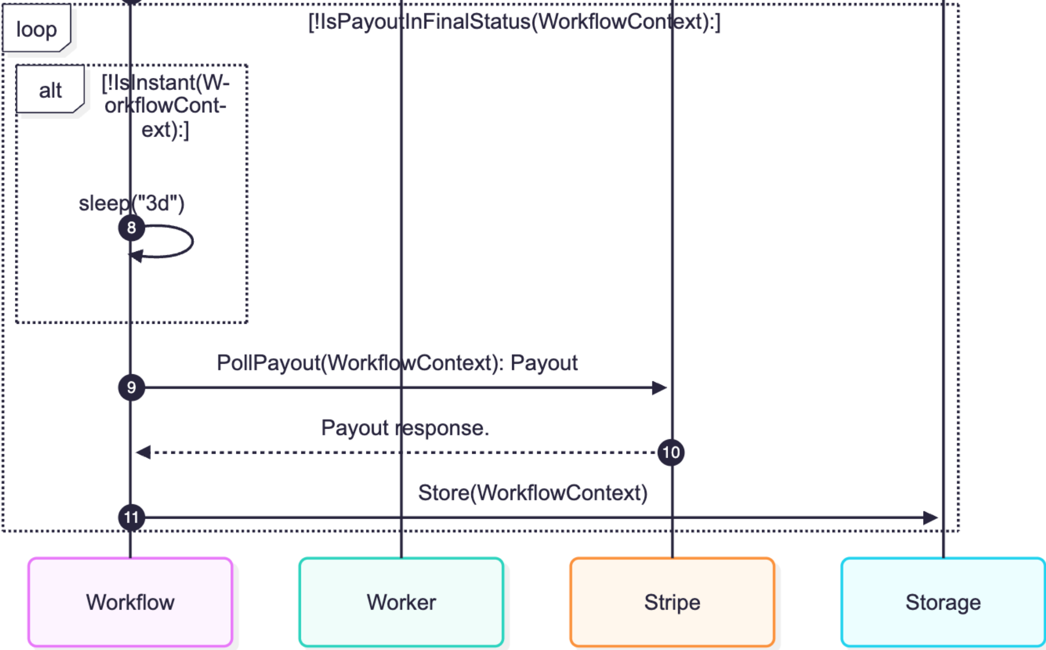


Figure 35: Initiate payout workflow (polling)

The monitoring phase is built around a polling loop. If the payout is not finalized instantly, the workflow pauses execution for three days before checking again. After the pause, it calls the Worker to poll Stripe [1] for the payout status. Stripe [1] returns the current state of the payout, and the workflow records this information in storage. The system then checks whether the payout has reached a final state, such as success, failure, or cancellation. If not, the workflow repeats the sleep-poll cycle until a definitive result is obtained.

4.6 Deployment

The deployment process begins at the level of source code management. A repository hosted on a Git [18] provider is connected to Cloudflare, creating a direct link between version control and the edge network. When a developer commits and pushes new code, the event automatically triggers a build process within Cloudflare's infrastructure. This build step packages the worker application and prepares it for execution within the distributed runtime environment of Cloudflare Workers [11].

Figure 36: Workers deployment

In order to illustrate the deployment workflow adopted in this project, *figure 36* provides a schematic representation of the process as it is implemented through Cloudflare Workers Builds [84]. The figure demonstrates the three key stages of the pipeline: (i) the connection of a source code repository hosted on a Git [85] provider, such as GitHub [86] or GitLab [87]; (ii) the act of committing and pushing code changes; and (iii) the subsequent build and deployment of the Worker to Cloudflare's infrastructure. This visual encapsulates the principle of continuous integration and deployment at the edge, where every commit serves as a potential release candidate.

Once the repository is linked to Cloudflare, a developer's commit (e.g., *git commit -m "Your commit message"*) initiates an automated workflow. As highlighted in the diagram, this workflow consists first of building the Worker application and then of deploying it directly to the global edge network. Crucially, the pipeline is not restricted to a single target environment. By associating separate repository branches with different Cloudflare Workers [11], it is possible to maintain both staging and production deployments in parallel. In practical terms, the *staging branch* is configured to deploy to a Worker dedicated to quality assurance and pre-release testing, while the *main branch* is reserved for deploying validated code to the production Worker that serves end users.

The use of a branching model, combined with Cloudflare's native CI/CD [88] capabilities, provides an effective mechanism for enforcing release discipline. Features and fixes are first validated in staging, under conditions that closely approximate the production environment, before they are promoted to the main branch. This ensures that production remains stable while allowing rapid iteration in development. Furthermore, Cloudflare facilitates the use of environment-specific secrets and variables, thereby isolating sensitive production credentials from test configurations.

4.7 Planned Timeline

The development of the platform unfolds across a nine-month window, extending from January 2025 to October 2025 as show in the *figure 37*, in a sequence of overlapping phases that mirror the architecture itself. The timeline is not arbitrary; rather, it embodies the engineering rationale that resilience and observability must be secured before functional complexity is layered on. This structured approach ensures that each milestone is not only a practical achievement but also a validation of underlying design hypotheses.

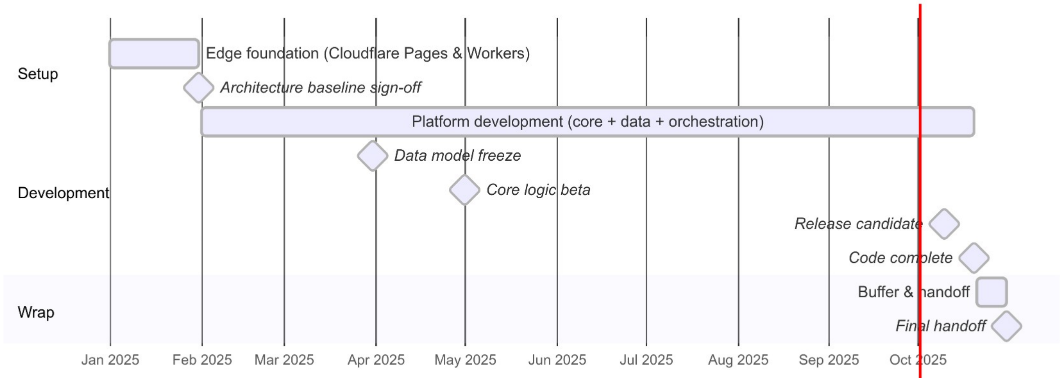


Figure 37: Project Timeline

The development begins in January 2025 with the establishment of the edge foundation. At this stage, Cloudflare Pages are deployed to host both the merchant portal and the player checkout, while Cloudflare Workers provide validation, authentication stubs, and request tracing. This initial effort serves as a proof of concept, validating that an edge-first architecture can sustain low-latency, globally distributed transactions.

In February 2025, the focus shifts to orchestration. Cloudflare Workflows are introduced to model financial operations as durable, long-lived processes, and Cloudflare Queues are employed to reconcile asynchronous execution with synchronous user expectations.

By March 2025, attention turns to persistence. The data layer is realized through Cloudflare D1, where the entity-relationship model is formalized as a normalized schema with strict integrity constraints. Atomic commits guarantee correctness under concurrency, turning theoretical principles of normalization and idempotency into practical operations.

The timeline advances to June 2025, when core financial logic is integrated. Transaction lifecycles, settlement flows, and reconciliation mechanisms are implemented and tested in controlled scenarios to validate their correctness and reliability.

In September 2025, the system undergoes final feature integration and systemic validation. Interactions between orchestration, persistence, and financial logic are stress-tested to ensure resilience and to confirm the underlying architectural hypotheses.

Finally, in October 2025, development reaches its conclusion in parallel with thesis writing. The platform achieves operational readiness, while the thesis captures the engineering journey, demonstrating how design decisions, implementation strategies, and validation milestones coalesce into a coherent system.

Chapter 5 Results achieved

The culmination of this project is the design, implementation, and validation of a robust, scalable monetization platform for game creators, integrating workflow orchestration principles with modern payment infrastructure. This chapter presents the results achieved, structured around the objectives defined earlier, and evaluates how the system performed in terms of functionality, reliability, and developer usability. It also highlights the lessons learned during development and assesses the broader contribution of the work to the field of distributed workflows and game monetization.

One of the primary objectives of the project was to build a reference implementation demonstrating the feasibility of the platform. This was accomplished through the development of a fully functional game shop prototype that integrates all stages of the monetization workflow: cart management, payment authorization, transaction execution, and the delivery of digital items into player inventories.

The prototype validated that end-to-end monetization flows could be reliably orchestrated using Cloudflare Workflows [4] as the workflow engine, with Stripe [1] serving as the payment processor. By simulating real-world purchase scenarios, the system confirmed that purchased items were consistently credited to player accounts, even in the presence of transient failures such as network outages or worker crashes. This demonstrated the platform's ability to provide durable execution guarantees and to recover seamlessly from interruptions without compromising correctness.

The platform was developed with a strong emphasis on developer experience. Using idiomatic Javascript code and Cloudflare workflows model, the system achieved modularity and readability, reducing complexity in the orchestration layer. Cloudflare Workflows [1] built-in visibility features proved valuable for observability and debugging: developers could inspect workflow histories, trace execution paths, and identify points of failure from the User Interface (UI).

Unit and integration tests were created to validate workflow correctness under both normal and failure conditions. These tests confirmed the maintainability of the system, ensuring that developers could extend or modify workflows without introducing regressions. The result is a

system that not only works in practice but also remains accessible to developers who must maintain it over time.

Stripe [1] integration was achieved for both incoming payments and outgoing payouts. On the input side, Stripe's [1] Application Programming Interfaces (APIs) were used to process player purchases, ensuring compliance with security and fraud prevention standards while supporting global payment methods. On the output side, automated payouts were implemented, enabling creators to receive funds directly in their bank accounts.

Together, these outcomes confirm that the project's central objective, to design and implement a robust monetization platform for game creators, was met.

Beyond fulfilling its immediate objectives, the project contributes to a broader understanding of workflow orchestration in gaming contexts. By applying distributed systems principles to the specific domain of in-game monetization, it demonstrates how workflow-as-code can simplify the development of reliable, stateful processes. The platform highlights the advantages of combining durable execution with modern payment infrastructure, offering a reusable blueprint for game developers and potentially for other digital content industries.

Chapter 6 Conclusion and Future Work

This dissertation set out to investigate how workflow orchestration principles can be applied to the monetization challenges faced by game creators. From the outset, the research identified a critical gap between the technical capabilities of modern payment processors and the operational realities of in-game fulfillment. While services such as Stripe [1] provide secure and scalable access to financial rails, the orchestration of long-lived, fault-tolerant workflows remains largely the responsibility of developers. For many studios, particularly smaller teams, this creates a barrier to reliable monetization, with direct consequences for player satisfaction and revenue stability.

To address this gap, the work combined theoretical foundations from distributed systems with practical experimentation in serverless environments. The analysis traced the evolution from monoliths to microservices and serverless computing, highlighting the coordination and durability problems that persist in fragmented architectures. By surveying state-of-the-art orchestration platforms Temporal [2], AWS Step Functions [3], Cloudflare Workflows [4], among others the research situated its approach within a broader technological landscape and clarified the design trade-offs between durability, responsiveness, and ecosystem integration.

The proposed architecture demonstrated how Cloudflare's edge-first platform, combined with Stripe's [1] payment infrastructure, can serve as the backbone of a resilient monetization system. Workflows were modeled as durable, replayable processes, supported by patterns such as idempotent persistence, early-return gating, and event-first strategies with polling fallbacks. Together, these mechanisms guaranteed that financial transactions could complete reliably, even in the face of partial failures, asynchronous delays, or external system outages. A functional prototype validated the design in practice, confirming that payments and payouts could be orchestrated end-to-end with consistency, observability, and developer ergonomics.

The contributions of this thesis are therefore twofold. First, it advances a conceptual framework that positions workflow orchestration as a natural extension of distributed systems theory to the domain of gaming monetization. Second, it delivers a concrete architecture and implementation that can serve as a blueprint for practitioners seeking to integrate robust payments into their games without diverting disproportionate resources to infrastructure. In

doing so, the research underscores the potential of orchestration engines to reduce operational overhead, enhance resilience, and strengthen trust in digital game economies.

At the same time, several limitations must be acknowledged. The prototype focused primarily on payments and payouts within a single provider ecosystem. Broader considerations such as multi-provider interoperability, fraud detection, or support for blockchain-based assets remain outside the present scope. Furthermore, while the architecture demonstrated reliability under simulated conditions, large-scale deployment in production environments may introduce new constraints around cost efficiency, compliance in multiple jurisdictions, and long-term maintainability.

Looking ahead, the future work outlined in this dissertation provides clear avenues for extension. Whitelabel capabilities would allow the platform to reach beyond gaming into adjacent digital industries. Revenue-sharing workflows would enable richer ecosystems where multiple stakeholders can participate fairly in monetization. Additional directions include integrating advanced fraud detection, supporting player-driven marketplaces, and exploring interoperability with emerging forms of digital ownership.

As the platform matures and adoption broadens, several opportunities arise to expand its applicability, flexibility, and monetization potential. The prototype developed in this thesis demonstrates the feasibility of orchestrating robust payment and payout workflows tailored to the needs of game creators. However, as with any foundational system, the path forward includes enhancements that would transform the platform from a proof-of-concept into a fully fledged, production-grade ecosystem.

This chapter outlines the key areas where future development could extend the platform’s capabilities. The focus is on two primary directions: enabling whitelabel partner and supporting revenue-sharing mechanism, though additional opportunities exist in compliance, scalability, and ecosystem integration.

In conclusion, this research has shown that durable workflow orchestration offers a promising foundation for solving one of the most pressing challenges in digital game development: reliable, scalable monetization. By aligning distributed systems principles with the operational realities of global payment flows, it contributes both to academic understanding and to practical solutions. Ultimately, the ambition of this work is to give creators the confidence to

focus on their creative vision, knowing that the underlying infrastructure can be trusted to deliver seamless, transparent, and resilient commerce in the digital worlds they build.

6.1 Become whitelabel

One of the most promising avenues for expansion is the transformation of the platform into a white-label solution. In this model, third-party developers and publishers would be able to adopt the platform's payment and fulfillment infrastructure as their own, embedding it seamlessly into their games, storefronts, or marketplaces while retaining their branding and user experience.

To enable this transition, the platform would require a dedicated whitelabel Application Programming Interface (API) layer that abstracts tenant-specific configurations. This includes the setup of payment instruments, payout methods, and customized workflows for each tenant. Financial logic would need to be modularized so that multiple independent tenants could securely coexist within the same infrastructure, each maintaining separate ledgers, transaction histories, and audit trails.

Beyond Application Programming Interface (API) development, white-label adoption would demand a structured onboarding framework for new tenants. This framework would handle identity verification, compliance checks (e.g., Know Your Customer (KYC) and Anti-Money Laundering (AML) requirements), and secure provisioning of financial instruments. Operational workflows would need to be extensible, enabling tenants to configure their own retry logic, reporting dashboards, and settlement cycles while still inheriting the reliability guarantees of the underlying orchestration engine.

The impact of this feature would be transformative. By offering white-label capabilities, the platform could extend beyond the gaming industry to other digital ecosystems, such as music, e-learning, or virtual events, positioning itself as a general-purpose monetization infrastructure for creators and communities.

6.2 Revenue sharing mechanism

Another critical area for future work is the implementation of a revenue-sharing system. As the platform facilitates purchases and in-game transactions, stakeholders often extend beyond

the primary developer to include content creators, mod developers, marketplace operators, or game server owners. A flexible and transparent mechanism for splitting revenues among these parties would unlock new monetization models and foster collaborative ecosystems.

Supporting revenue sharing requires significant extensions to the platform's financial logic. Workflows must be capable of handling multi-party payment distribution, applying configurable rules based on the transaction context, contractual agreements, or marketplace logic. For example, a purchase of a virtual item might allocate 70% of the revenue to the developer, 20% to a mod creator, and 10% to the server operator. These splits should be applied programmatically, ensuring consistency, reducing manual reconciliation, and providing clear visibility to all parties involved.

Introducing revenue-sharing capabilities would significantly expand the platform's appeal. Instead of serving only as a transactional engine, it would become the foundation for platform-driven ecosystems, where third-party creators can participate confidently, knowing that their contributions will be compensated fairly and automatically.

Bibliography

[1] "Stripe | Financial Infrastructure to Grow Your Revenue." Accessed: Sept. 24, 2025. [Online]. Available: https://stripe.com/en-pt

[2] "Durable Execution Solutions," Temporal.io. Accessed: Sept. 24, 2025. [Online]. Available: https://temporal.io/

[3] "Workflow Orchestration - AWS Step Functions - AWS," Amazon Web Services, Inc. Accessed: Sept. 24, 2025. [Online]. Available: https://aws.amazon.com/step-functions/

[4] "Cloudflare Workflows," Cloudflare Docs. Accessed: Sept. 24, 2025. [Online]. Available: https://developers.cloudflare.com/workflows/

[5] E. Demirkaya, "Announcing Cadence 1.0: The Powerful Workflow Platform Built for Scale and Reliability," Uber Blog. Accessed: Sept. 24, 2025. [Online]. Available: https://www.uber.com/en-EG/blog/announcing-cadence/

[6] "Conductor OSS Foundation." Accessed: Sept. 24, 2025. [Online]. Available: https://conductor-oss.org/

[7] "The Universal Process Orchestrator | Camunda." Accessed: Sept. 24, 2025. [Online]. Available: https://camunda.com/

[8] "What is Function-as-a-Service (FaaS)?" Accessed: Sept. 24, 2025. [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/

[9] "What is BaaS? | Backend-as-a-Service vs. serverless." Accessed: Sept. 24, 2025. [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/

[10] "What Is a Workflow Engine? | IBM." Accessed: Sept. 24, 2025. [Online]. Available: https://www.ibm.com/think/topics/workflow-engine

[11] "Cloudflare Workers©." Accessed: Sept. 24, 2025. [Online]. Available: https://workers.cloudflare.com/

[12] "The macro problem with microservices - Stack Overflow." Accessed: Sept. 28, 2025. [Online]. Available: https://stackoverflow.blog/2020/11/23/the-macro-problem-with-microservices/

[13] "Achieving ACID Properties in Micro-Services Architecture: Patterns and Best Practices | by Anil Kumar Tiwari | Medium." Accessed: Sept. 30, 2025. [Online].

Available: https://medium.com/@anilgeit/achieving-acid-properties-in-micro-services-architecture-patterns-and-best-practices-4979ddccec5c

[14] F. Halili, A. Nuhiji, and D. M. Veliu, “Polyglot Persistence in Microservices: Managing Data Diversity in Distributed Systems,” Sept. 08, 2025, *arXiv*: arXiv:2509.08014. doi: 10.48550/arXiv.2509.08014.

[15] shahzad bhatti, “Robust Retry Strategies for Building Resilient Distributed Systems,” Medium. Accessed: Sept. 30, 2025. [Online]. Available: https://shahbhat.medium.com/robust-retry-strategies-for-building-resilient-distributed-systems-8432705f5207

[16] A. Nadeem and M. Z. Malik, “A case for microservices orchestration using workflow engines,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, in ICSE-NIER ’22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 6–10. doi: 10.1145/3510455.3512777.

[17] “(PDF) Microservices vs. Monoliths: Comparative Analysis for Scalable Software Architecture Design,” *ResearchGate*, Aug. 2025, doi: 10.18034/ei.v11i2.734.

[18] “(PDF) Distributed Systems: Concepts, Principles, Models and Algorithms,” *ResearchGate*, Aug. 2025, Accessed: Sept. 28, 2025. [Online]. Available: https://www.researchgate.net/publication/361407695_Distributed_Systems_Concepts_Principles_Models_and_Algorithms

[19] “What is Monolithic Architecture? | IBM.” Accessed: Sept. 28, 2025. [Online]. Available: https://www.ibm.com/think/topics/monolithic-architecture

[20] M. Fowler, “bliki: Monolith First,” martinfowler.com. Accessed: Sept. 28, 2025. [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html

[21] “What Are Microservices? | IBM.” Accessed: Sept. 28, 2025. [Online]. Available: https://www.ibm.com/think/topics/microservices

[22] M. Söylemez, B. Tekinerdogan, and A. Kolukısa Tarhan, “Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review,” *Applied Sciences*, vol. 12, no. 11, p. 5507, Jan. 2022, doi: 10.3390/app12115507.

[23] “Serverless Computing: State-of-the-Art, Challenges and Opportunities | IEEE Journals & Magazine | IEEE Xplore.” Accessed: Sept. 24, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9756233

[24] “Serverless Function, FaaS Serverless - AWS Lambda - AWS,” Amazon Web Services, Inc. Accessed: Sept. 24, 2025. [Online]. Available: https://aws.amazon.com/lambda/

[25] “Cloud Run functions,” Google Cloud. Accessed: Sept. 24, 2025. [Online]. Available: https://cloud.google.com/functions

[26] “Azure Functions | Microsoft Azure.” Accessed: Sept. 24, 2025. [Online]. Available: https://azure.microsoft.com/en-us/products/functions

[27] “What is edge computing? | Benefits of the edge.” Accessed: Sept. 25, 2025. [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/

[28] “How Workers works,” Cloudflare Docs. Accessed: Sept. 25, 2025. [Online]. Available: https://developers.cloudflare.com/workers/reference/how-workers-works/

[29] I. Baldini *et al.*, “The serverless trilemma: function composition for serverless computing,” in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, in Onward! 2017. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 89–103. doi: 10.1145/3133850.3133855.

[30] “Firebase | Google’s Mobile and Web App Development Platform,” Firebase. Accessed: Sept. 24, 2025. [Online]. Available: https://firebase.google.com/

[31] “AWS Amplify,” Amazon Web Services, Inc. Accessed: Sept. 24, 2025. [Online]. Available: https://aws.amazon.com/amplify/

[32] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, “Serverless Workflows with Durable Functions and Netherite,” Feb. 26, 2021, *arXiv*: arXiv:2103.00033. doi: 10.48550/arXiv.2103.00033.

[33] D. M. R. Ferreira and J. J. Pinto Ferreira, “Developing a reusable workflow engine,” *Journal of Systems Architecture*, vol. 50, no. 6, pp. 309–324, June 2004, doi: 10.1016/j.sysarc.2003.09.004.

[34] “Temporal Workflow Execution Overview | Temporal Platform Documentation.” Accessed: Sept. 28, 2025. [Online]. Available: https://docs.temporal.io/workflow-execution

[35] “What is a Temporal Activity? | Temporal Platform Documentation.” Accessed: Sept. 28, 2025. [Online]. Available: https://docs.temporal.io/activities

[36] "What is a Temporal Worker? | Temporal Platform Documentation." Accessed: Sept. 28, 2025. [Online]. Available: https://docs.temporal.io/workers

[37] "What is Temporal? | Temporal Platform Documentation." Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/temporal

[38] "Architecture Overview - Conductor Documentation." Accessed: Oct. 01, 2025. [Online]. Available: https://conductor-oss.github.io/conductor/devguide/architecture/index.html

[39] M. Fowler, "Event Sourcing," martinfowler.com. Accessed: Sept. 28, 2025. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html

[40] "Temporal Service | Temporal Platform Documentation." Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/temporal-service

[41] "Amazon States Language." Accessed: Oct. 01, 2025. [Online]. Available: https://states-language.net/

[42] Datadog, "What is AWS Step Functions? How it Works & Use Cases," Datadog. Accessed: Sept. 30, 2025. [Online]. Available: https://www.datadoghq.com/knowledge-center/aws-step-functions/

[43] "Technical Details - Conductor Documentation." Accessed: Oct. 01, 2025. [Online]. Available: https://conductor-oss.github.io/conductor/devguide/architecture/technicaldetails.html#grpc-framework

[44] "Task Lifecycle - Conductor Documentation." Accessed: Oct. 01, 2025. [Online]. Available: https://conductor-oss.github.io/conductor/devguide/architecture/tasklifecycle.html

[45] "Workers - Conductor Documentation." Accessed: Oct. 01, 2025. [Online]. Available: https://conductor-oss.github.io/conductor/devguide/concepts/workers.html

[46] "Prometheus - Monitoring system & time series database." Accessed: Oct. 01, 2025. [Online]. Available: https://prometheus.io/

[47] Datadog, "Full Stack Observability & Security Built for Enterprise Scale," Datadog. Accessed: Oct. 01, 2025. [Online]. Available: https://www.datadoghq.com/dg/monitor/free-trial/

[48] "Elastic Stack: (ELK) Elasticsearch, Kibana & Logstash," Elastic. Accessed: Oct. 01, 2025. [Online]. Available: https://www.elastic.co/en-us/elastic-stack

[49] “Workers - Conductor Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://conductor-oss.github.io/conductor/devguide/concepts/workers.html

[50] “Technical Details - Conductor Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://conductor-oss.github.io/conductor/devguide/architecture/technicaldetails.html#dynamic-workflow-executions

[51] “Durable Execution Solutions | Temporal.” Accessed: Oct. 01, 2025. [Online]. Available: https://temporal.io/

[52] “What is Workflow Orchestration? | IBM.” Accessed: Oct. 01, 2025. [Online]. Available: https://www.ibm.com/think/topics/workflow-orchestration

[53] “Events and Event History | Temporal Platform Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/workflow-execution/event

[54] “What is a Temporal Worker? | Temporal Platform Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/workers

[55] “Task Queues | Temporal Platform Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/task-queue

[56] “Temporal Workflow Definition | Temporal Platform Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/workflow-definition

[57] “Event History Walkthrough with the TypeScript SDK | Temporal Platform Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/encyclopedia/event-history/event-history-typescript

[58] “What is a Temporal Activity? | Temporal Platform Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/activities

[59] “What is a Temporal Worker? | Temporal Platform Documentation.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.temporal.io/workers

[60] “Build durable applications on Cloudflare Workers: you write the Workflows, we take care of the rest,” The Cloudflare Blog. Accessed: Oct. 01, 2025. [Online]. Available: https://blog.cloudflare.com/building-workflows-durable-execution-on-workers/

[61] “Cloudflare Queues,” Cloudflare Docs. Accessed: Sept. 24, 2025. [Online]. Available: https://developers.cloudflare.com/queues/

[62] “Application Performance Monitoring & Error Tracking Software,” Sentry. Accessed: Oct. 01, 2025. [Online]. Available: https://sentry.io/welcome/

[63] “What is Step Functions? - AWS Step Functions.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html

[64] “Learn about state machines in Step Functions - AWS Step Functions.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-statemachines.html

[65] “Discovering workflow states to use in Step Functions - AWS Step Functions.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/workflow-states.html

[66] “Choice workflow state - AWS Step Functions.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/state-choice.html

[67] “Wait workflow state - AWS Step Functions.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/state-wait.html

[68] “Monitoring Step Functions metrics using Amazon CloudWatch - AWS Step Functions.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/procedure-cw-metrics.html

[69] G. David, D. R. Zmaranda, R.-Ş. Győrödi, and C. A. Győrödi, “Exploring the Impact of Workflow Engines on Business Process Management in Enterprise Applications. A case-study: Camunda,” in *2023 17th International Conference on Engineering of Modern Electric Systems (EMES)*, June 2023, pp. 1–4. doi: 10.1109/EMES58375.2023.10171706.

[70] “Process Engine API | docs.camunda.org.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.camunda.org/manual/latest/user-guide/process-engine/process-engine-api/

[71] J. Johnson, “Decision Engines: What They Are and What They Do,” Camunda. Accessed: Oct. 01, 2025. [Online]. Available: https://camunda.com/blog/2024/01/decision-engines-what-they-are-what-they-do/

[72] “Processes | Camunda 8 Docs.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.camunda.io/docs/components/concepts/processes/

[73] “Workflow patterns | Camunda 8 Docs.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.camunda.io/docs/components/concepts/workflow-patterns/

[74] “User tasks | Camunda 8 Docs.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.camunda.io/docs/components/modeler/bpmn/user-tasks/

[75] “Job workers | Camunda 8 Docs.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.camunda.io/docs/components/concepts/job-workers/

[76] “Supported Environments | docs.camunda.org.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.camunda.org/manual/latest/introduction/supported-environments/

[77] “Cockpit | docs.camunda.org.” Accessed: Oct. 01, 2025. [Online]. Available: https://docs.camunda.org/manual/latest/webapps/cockpit/

[78] “Optimize,” Camunda. Accessed: Oct. 01, 2025. [Online]. Available: https://camunda.com/platform/optimize/

[79] “Tasklist: Manage human workflows,” Camunda. Accessed: Oct. 01, 2025. [Online]. Available: https://camunda.com/platform/tasklist/

[80] “Cloudflare Workflows,” Cloudflare Docs. Accessed: Sept. 29, 2025. [Online]. Available: https://developers.cloudflare.com/workflows/

[81] “Cloudflare D1,” Cloudflare Docs. Accessed: Sept. 29, 2025. [Online]. Available: https://developers.cloudflare.com/d1/

[82] “What is edge computing? | Benefits of the edge.” Accessed: Sept. 29, 2025. [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/

[83] “Stripe Identity | Identify Verification for Payments.” Accessed: Oct. 01, 2025. [Online]. Available: https://stripe.com/en-pt/identity

[84] “Builds,” Cloudflare Docs. Accessed: Sept. 24, 2025. [Online]. Available: https://developers.cloudflare.com/workers/ci-cd/builds/

[85] “Git.” Accessed: Sept. 24, 2025. [Online]. Available: https://git-scm.com/

[86] “Build software better, together,” GitHub. Accessed: Sept. 24, 2025. [Online]. Available: https://github.com

[87] “The most-comprehensive AI-powered DevSecOps platform,” about.gitlab.com. Accessed: Sept. 24, 2025. [Online]. Available: https://about.gitlab.com/

[88] “CI/CD,” Cloudflare Docs. Accessed: Sept. 24, 2025. [Online]. Available: https://developers.cloudflare.com/workers/ci-cd/

[89] “Cloudflare Pages.” Accessed: Sept. 24, 2025. [Online]. Available: https://pages.cloudflare.com/