MODULE 11 · STUDY NOTES

# Defining Workflows to Orchestrate Functions

*How AWS Step Functions coordinates distributed applications — state machines and the Amazon States Language, the eight state types, Standard vs. Express workflows, activities, and the Step Functions API*

At the café, the coffee inventory now updates automatically on the website, and Frank would like to request reports with the latest inventory on demand. The data lives in an Amazon RDS database, a Lambda function can format it into a report, and Amazon SNS can email Frank when the report is ready — but something has to run those steps in the right order, wait where it must wait, retry what fails, and remember where it is. That coordinator is **AWS Step Functions**. This module builds toward Sofía's solution: first the problem of coordinating tasks across the independent components of a distributed, microservices application; then Step Functions itself — a serverless orchestration service whose workflows are **state machines** written in the **Amazon States Language**; the **eight state types** that make up a workflow; the difference between **Standard** and **Express** workflows; the use cases and **activities** that extend Step Functions to human-in-the-loop and externally hosted work; and finally the **API** you call to create and run state machines.

LEARNING OBJECTIVES

- Recognize the dynamics of workflow coordination in distributed applications
- Describe AWS Step Functions
- Identify state types
- Indicate common use cases for Step Functions
- Recall Step Functions API operations

## 11.1 Coordinating tasks in distributed applications

Modern cloud architecture decouples an application into smaller, independent components called **microservices**, which are easier to develop, deploy, and maintain. From those individual components — each performing a discrete function or task — you build **distributed applications**, and you can deploy, scale, and update the components of the application quickly and independently.

As an application grows, though, the number of components increases and so do the patterns for running tasks. Consider a serverless application built from **AWS Lambda functions**: you might need to invoke one function immediately after another and *only if the first succeeds*; run two functions **in parallel** and feed the combined results into a third; or **choose** which of two functions to invoke based on another function's output. Coordinating those patterns by hand quickly becomes fragile.

Failure makes it harder still. A function invocation can error for several reasons — your code raises an exception, times out, or runs out of memory, or the runtime itself encounters an error and stops. When an error occurs, your code might have run **completely, partially, or not at all**. In most cases the client or service that invoked the function **automatically retries**, so your code must be able to process the same event repeatedly without unwanted effects. If a function manages resources or writes to a database, you must handle the case where the same request is made several times.

**COMMON PITFALL**

Automatic retries make **idempotency** a requirement, not a nicety. Because an invoker retries after an error — and the code may already have run partially — a function that writes to a database or manages resources must be written so that processing the same event more than once produces no unwanted side effects.

**WHAT A COORDINATION LAYER MUST PROVIDE**

- **Scale automatically** in response to changing workloads
- **Handle errors and timeouts** gracefully
- **Maintain application state** while it runs — track which step it is in and store the data moving between steps
- **Visibility** into the application so you can troubleshoot errors and track performance

**SECTION TAKEAWAYS**

- Microservices are smaller, independent components that make up an application in modern cloud architecture.
- You can use microservices to quickly deploy, scale, and update application components.
- A coordination layer is necessary to handle automatic scaling, errors, and timeouts.

## 11.2 Introducing Step Functions

**AWS Step Functions** is a web service you use to coordinate the components of distributed applications and microservices with **visual workflows**. It provides a reliable way to step through the functions of your application: a graphical console lets you visualize the components as a series of steps, it **automatically triggers and tracks** each step and **retries when errors occur** so the application runs in order and as expected, and it **logs the state of each step** so you can diagnose and debug problems quickly.

**TABLE 11.1** Benefits of Step Functions

| BENEFIT | WHAT IT DELIVERS |
|---|---|
| **Productivity — build and update quickly** | Visual workflows translate business requirements into technical ones fast; build applications in minutes and exchange or reorganize components without customizing code |
| **Agility — scale and recover reliably** | Automatically scales the operations and underlying compute in response to changing workloads, keeping workflow performance consistent as request frequency rises |
| **Resiliency — evolve applications easily** | Manages state, checkpoints, and restarts; built-in try/catch, retry, and rollback handle errors and exceptions automatically |

**WORKFLOW LOGIC STEP FUNCTIONS MANAGES FOR YOU**

- Run tasks **in sequence** and **in parallel**
- **Select a task based on data** (branching)
- Manage **try-catch-finally** behavior and timeouts
- **Retry** failed or timed-out tasks — whether a task takes seconds or months — respond differently to different error types, and recover gracefully by returning to designated cleanup and recovery code

You define a workflow as a series of steps and **transitions** between them. A workflow is also called a **state machine**, and its **states** are the elements it contains. A state machine is an object that has a set number of operating conditions that depend on the object's previous condition to determine output — so individual states can make decisions based on their input, perform actions, and pass output to other states. **Tasks do all of the work** in a state machine (a task uses an activity, a Lambda function, or another service's API). Each state machine starts with an **input**; the state transforms that input and passes the output to the next state, and you can visualize and track it all in the Step Functions console with near-real-time information.

> **WORKED EXAMPLE – DEFINING A STATE MACHINE IN ASL**
>
> State machines are defined with the **Amazon States Language (ASL)**, a JSON-based structured language that Step Functions renders as a real-time graphical view. A minimal "Hello World" machine sets `"StartAt": "HelloWorld"` and defines one state, `HelloWorld`, with `"Type": "Pass"`, `"Result": "Hello World!"`, and `"End": true` — the console draws it as Start → HelloWorld → End. When you would rather build visually, **AWS Step Functions Workflow Studio** is a low-code, guided interface for prototyping and building the same state machines faster.

**TABLE 11.2** Common state fields when running a state machine

| FIELD | ROLE |
|---|---|
| `StartAt` | Top-level field naming the state where the run begins — **required** |
| `Type` | The state's type (Task, Choice, Pass, …) — **required** |
| `Next` | The next state to run when the current state finishes (the transition target) |
| `End` | Ends the state machine at this state when set to `true` |

When a state machine launches, it begins at the state referenced in the top-level `StartAt` field. After Step Functions runs a state, it uses that state's `Next` field to determine which state to advance to — a **transition**. A **terminal state** is a `Succeed` state, a `Fail` state, or any state whose `End` field is set to `true`; when the run reaches a terminal state, or a runtime error occurs, the state stops running and returns a result.

> **SECTION TAKEAWAYS**
>
> - Step Functions coordinates components of distributed applications and microservices using visual workflows.
> - A state machine is an object with a set number of operating conditions that depend on the object's previous condition to determine output.
> - State machines are defined using the Amazon States Language.

## 11.3 State types

A finite state machine expresses an algorithm as a number of **states**, their relationships, and their input and output. States are the elements of your state machine, and each can make decisions based on its input, perform actions, and pass output to other states. Step Functions offers **eight** state types.

**TABLE 11.3** State types at a glance

| STATE | PURPOSE |
|---|---|
| **Task** | A single unit of work |
| **Choice** | Adds branching logic to a state machine |
| **Fail** | Stops running a state and marks it as a failure |
| **Succeed** | Stops running a state successfully |
| **Pass** | Passes its input to its output, without performing work |
| **Wait** | Delays continuing for a specified time |
| **Parallel** | Creates parallel branches for running states |
| **Map** | Dynamically iterates steps over the entries of an array |

**Task — the only state that does work.** Tasks do all of the work in a state machine. A Task (`"Type": "Task"`) performs work in one of three ways: by running an **activity**, by invoking an **AWS Lambda function**, or by passing parameters to the **API actions of other AWS services**. The state is given the **Amazon Resource Name (ARN)** of the activity or function in its `Resource` field and can set controls such as `TimeoutSeconds`, `HeartbeatSeconds`, and `Next`. An **activity** is program code that waits for an operator to perform an action or provide input; it can be hosted on Amazon EC2, Amazon ECS, or even mobile devices, and it polls Step Functions with the `GetActivityTask`, `SendTaskSuccess`, `SendTaskFailure`, and `SendTaskHeartbeat` API actions.

**Choice — branching logic.** A Choice state (`"Type": "Choice"`) must have a `Choices` field whose value is a non-empty array of **Choice Rules**. Each Choice Rule pairs a **comparison** — an input `Variable`, a comparison type such as `NumericEquals`, and the value to compare against — with a `Next` field that must match a state name in the machine. A `Default` field names the state to run when no rule matches. In the deck's example, one rule checks whether a value equals `1` and another whether it equals `2`; the `$` symbol marks a variable.

**Fail and Succeed — terminal stops.** A Fail state (`"Type": "Fail"`) stops the state machine and marks it a **failure**; it requires only `Type` and offers two optional fields — `Cause` (a custom failure string for operational or diagnostic use) and `Error` (an error name usable for retry/catch, operational, or diagnostic purposes). A Succeed state (`"Type": "Succeed"`) stops a state **successfully** and is a useful target for Choice branches that only need to stop. Because both always exit the machine, **neither has a `Next` field and neither requires an `End` field**.

**Pass and Wait — helpers.** A Pass state (`"Type": "Pass"`) passes its input straight to its output without doing work; it can inject fixed data through a `Result` field, which makes it handy for constructing and debugging state machines. A Wait state (`"Type": "Wait"`) delays the workflow for a set time — either a **relative** delay in `Seconds` (for example, a 10-second pause) or an **absolute** end time given as a timestamp — before transitioning to its `Next` state.

**Parallel — concurrent branches.** A Parallel state (`"Type": "Parallel"`) runs several branches at once. Its required `Branches` field is an array; Step Functions starts each branch at that branch's own `StartAt` state and runs the branches **as concurrently as possible**, then waits until **all** branches reach a terminal state before processing the Parallel state's `Next` field. If any branch fails — from an unhandled error or by transitioning to a Fail state — the **entire Parallel state is considered to have failed and all of its branches are stopped**.

**Map — iterate over an array.** A Map state (`"Type": "Map"`) creates a set of steps and runs them for **multiple entries of an array** in the state's input. The `Iterator` field is **required** and defines the steps to run for each item; supporting fields in the deck's example include `InputPath`, `ItemsPath`, `MaxConcurrency`, and `ResultPath`. Use Map to apply the same processing to every element of a collection.

**COMMON PITFALL**

Watch the terminal-state and Parallel details — they are favorite exam points. `Succeed` and `Fail` need no `End` field because they already end the run, and a **single failing branch fails the whole `Parallel` state** and stops its siblings. Branching is the job of `Choice`; running work concurrently is the job of `Parallel`.

**SECTION TAKEAWAYS**

- Individual states can make decisions based on their input, perform actions, and pass and manipulate output to other states.
- Add branching logic to a state machine by adding a Choice state.
- Run multiple functions concurrently by using the Parallel state.
- Succeed and Fail states do not require an End field.

## 11.4 Standard and Express workflows

Step Functions offers **two workflow types — Standard and Express**. You define your state machine with the **same Amazon States Language** either way; what changes is how the state machine **behaves when it runs**, along with its limits and pricing. Choose the type per use case.

**TABLE 11.4** Standard vs. Express workflows

| ATTRIBUTE | STANDARD | EXPRESS |
|---|---|---|
| **Best for** | Long-running, auditable workflows | High-volume, event-processing workloads |
| **Maximum run duration** | Up to 1 year | Up to 5 minutes |
| **Execution model** | Exactly-once (runs once unless `Retry` is set in ASL) | At-least-once (async) or at-most-once (sync); can run more than once |
| **Invocation rate** | 2,000 per second | 100,000 per second |
| **Transition rate** | 4,000 per second | Nearly unlimited |
| **Pricing** | Per processed state transition | Per number and duration of requests, and memory consumed |
| **Activities & patterns** | Supports Step Functions activities and all service and integration patterns | No activity support; all service integrations and most patterns |

**Standard workflows** show a history of previous runs and provide **visual debugging**; the full run history can be retrieved through the Step Functions API for up to **90 days** after a run completes. Their **exactly-once** model means they never run more than once unless you specify the `Retry` behavior in ASL — which makes them suitable for **non-idempotent** actions whose invocations always result in a change of state.

**Express workflows** are ideal for **high-event-rate** workloads such as streaming data processing and Internet of Things (IoT) data ingestion. **Asynchronous** Express workflows use an **at-least-once** model and **synchronous** Express workflows use an **at-most-once** model, so an invocation **can run more than once**. Express workflows run for up to 5 minutes and are billed on the **number and duration of requests and the memory consumed**.

**COMMON PITFALL**

Match the model to the workload. If a question stresses **long-running, auditable, or non-idempotent** work, the answer is **Standard** (exactly-once, up to a year, billed per transition). If it stresses **high-volume streaming or IoT events**, the answer is **Express** (up to 5 minutes, at-least-once / at-most-once, billed by requests and memory) — and note Express does **not** support activities.

## 11.5 Step Functions use cases and activities

Step Functions fits a broad range of jobs, and any of them can rely on **Standard or Express** workflows depending on requirements.

**COMMON STEP FUNCTIONS USE CASES**

- Process data · automate tasks · orchestrate an application
- Modernize a monolithic application into coordinated components
- Transcode media files · sequence batch processing jobs
- Send messages from, and publish events from, automated and serverless workflows
- Coordinate container tasks in microservices and serverless applications
- Access databases from serverless workflows
- Sequence the steps of machine learning (ML) workflows and coordinate extract, transform, and load (ETL) jobs

Some workflows need work done by code that Step Functions does not run directly — including **human-in-the-loop** steps. **Activities** are a Step Functions feature that associates such code (an **activity worker**) with a specific Task in a state machine. An activity worker can be an application on an EC2 instance, a Lambda function, or a mobile device — **any application that can make an HTTP connection**, hosted anywhere. When a run reaches an **activity task state**, the workflow **waits for a worker to poll** for the task.

**WORKED EXAMPLE – A MANUAL APPROVAL STEP**

A promotion-approval workflow uses an activity. Amazon CloudWatch invokes a Lambda function (the activity worker) on a schedule; the worker polls Step Functions by calling **`GetActivityTask`**, sending the activity's ARN. The response returns the task **input** (a JSON string) and a **`taskToken`** (a unique identifier for the task). The worker then calls **Amazon SES**, which emails a manager to approve or deny the promotion. The approve/deny response is sent as a request to **Amazon API Gateway**, which calls **`SendTaskSuccess`** (approve) or **`SendTaskFailure`** (deny) using the vended `taskToken`. Notice the design: the code that *fetches the work and acquires the token* is separate from the code that *responds with the completion status and returns the token*.

Step Functions **generates the `taskToken`** when a task is assigned to a worker, and the worker returns that same token so the result is tied to the right task; a task waiting on a token will wait until the run reaches the **one-year** service quota, governed by a `Heartbeat` threshold declared in the ASL. The same pattern of activities and **Wait** states appears in longer pipelines — for example, the automated, continuous deployment of **Amazon SageMaker** models, where the state machine repeatedly **polls and waits** for build and training tasks to finish before moving on.

SECTION TAKEAWAYS

- Activities are a Step Functions feature that associate code with a specific task in a state machine.
- A taskToken represents a specific task and is generated by Step Functions when a task is assigned to a worker.
- An activity worker can run on any application that is able to make an HTTP connection.

## 11.6 The Step Functions API and quotas

You can access and use Step Functions through the **AWS Management Console**, the **AWS SDKs**, or the **API**. To run a state machine, you either call the **`StartExecution`** API action directly, or associate your Step Functions API actions with operations in **Amazon API Gateway** — when an HTTPS request hits an API operation, API Gateway invokes your Step Functions API actions.

**TABLE 11.5** Step Functions API operations

| OPERATION | WHAT IT DOES |
|---|---|
| **`Create`** | Upload state machines that are defined in JSON; register activity workers |
| **`StartExecution`** | Start a run and return an Amazon Resource Name (ARN) that identifies the run |
| **`StopExecution`** | Stop a run and return the date that the state machine stopped |
| **`List`** | List all state machines, runs, and activities |
| **`Describe`** | Describe individual state machines, runs, and activities |
| **`GetExecutionHistory`** | Get the history of a state machine run as a list of events |

Step Functions also places **quotas** (formerly called limits) on the sizes of certain state machine parameters — for example, how many API actions you can make during a time period, or how many state machines you can define. These are **not hard quotas**: they exist to keep a misconfigured state machine from consuming all of the system's resources, and they **can differ significantly between Standard and Express** workflows.

CATEGORIES OF STEP FUNCTIONS QUOTAS

- **General** quotas, such as names in Step Functions
- **Accounts** — for example, the maximum number of registered state machines or API actions
- **State machine runs** — for example, maximum runtime
- **Task runs** — for example, maximum task runtime
- **API action throttling** and **state throttling**

SECTION TAKEAWAYS

- You can access Step Functions by using the AWS Management Console, AWS SDKs, or API.
- Step Functions places quotas on the size of certain state machine parameters.
- Use API Gateway to associate your Step Functions APIs with methods in an API Gateway API.

## Module summary

- Microservices decompose an application into small, independent components; because components multiply and invokers auto-retry on error, you need a coordination layer that scales automatically, handles errors and timeouts, maintains state, and gives visibility — and function code must be idempotent.
- AWS Step Functions is a serverless orchestration service that coordinates distributed applications and microservices with visual workflows, automatically triggering, tracking, retrying, and logging each step.
- A workflow is a state machine written in the Amazon States Language (JSON); states are its elements, Tasks do all the work, and each state transforms its input and passes output to the next.
- A run starts at `StartAt`, advances by each state's `Next` field (a transition), and ends at a terminal state — `Succeed`, `Fail`, or `End: true` — or on a runtime error.
- The eight state types are Task, Choice, Fail, Succeed, Pass, Wait, Parallel, and Map; Succeed and Fail are terminal and need no End field, and a single failing Parallel branch fails the whole state.
- Standard workflows run up to a year, are exactly-once, and are billed per state transition; Express workflows run up to 5 minutes, are at-least-once/at-most-once, and are billed by request count, duration, and memory — both defined in the same ASL.
- Activities link externally hosted code (an activity worker reachable over HTTP) to a Task; the worker polls with GetActivityTask, receives a taskToken, and reports back with SendTaskSuccess or SendTaskFailure — enabling manual-approval and other human-in-the-loop steps.
- Access Step Functions through the console, SDKs, or API (StartExecution starts a run and returns its ARN; other operations include Create, StopExecution, List, Describe, GetExecutionHistory), and note that quotas are soft and may differ between workflow types.

## Review questions

1. **Why do distributed, microservices-based applications need a coordination layer, and what must it handle?**

   *Answer:* As components multiply, tasks must run in sequence, in parallel, or conditionally, and any function can fail (exceptions, timeouts, out-of-memory, runtime errors). A coordination layer scales automatically with changing workloads, handles errors and timeouts, maintains application state (tracking the current step and the data passing between steps), and provides visibility for troubleshooting and performance tracking.

2. **What is a state machine, and what language defines one?**

   *Answer:* A state machine is a workflow — an object with a set number of operating conditions whose output depends on its previous condition; its elements are states. It is defined with the Amazon States Language (ASL), a JSON-based structured language, and the language is the same for Standard and Express workflows.

3. **Trace how a run moves from state to state and how it ends.**

   *Answer:* It begins at the top-level `StartAt` state with an input; each state runs its `Type`, transforms the input, and uses its `Next` field to transition to the following state. The run ends at a terminal state — a Succeed, a Fail, or any state with `End` set to `true` — or when a runtime error occurs, at which point it returns a result.

**4. Name the eight state types, and give one situation for each of Choice, Parallel, and Map.**

*Answer:* Task, Choice, Fail, Succeed, Pass, Wait, Parallel, Map. Choice adds branching by comparing input and routing to a Next state; Parallel runs several branches concurrently (for example, run several functions at once); Map iterates the same steps over each entry of an array.

---

**5. How do Succeed and Fail states differ from other states in their required fields, and what optional fields does Fail add?**

*Answer:* Both are terminal, so neither has a `Next` field and neither requires an `End` field. Fail adds two optional fields: `Cause` (a custom failure string for diagnostics) and `Error` (an error name usable for retry/catch and diagnostics).

---

**6. A workflow must run for several days, be auditable, and never accidentally repeat a non-idempotent step. Standard or Express — and why?**

*Answer:* Standard. It supports runs up to one year, retains a run history for visual debugging and auditing (retrievable via the API for up to 90 days), and uses an exactly-once model, so it never runs more than once unless `Retry` is set. Express caps at 5 minutes and is at-least-once/at-most-once, so an invocation can run more than once.

---

**7. Walk through how an activity implements a manual approval step.**

*Answer:* A Task set to an activity waits for an activity worker (any app that can make an HTTP connection) to poll with `GetActivityTask`, which returns the task input and a `taskToken`. In the deck's example, CloudWatch triggers a Lambda worker that calls Amazon SES to email a manager; the approve/deny response goes through API Gateway, which calls `SendTaskSuccess` or `SendTaskFailure` with the same `taskToken`, tying the result to the task.

---

**8. How can you start a state machine run, and which API operation returns an ARN identifying the run?**

*Answer:* Call `StartExecution` directly (through the console, an SDK, or the API), or wire your Step Functions API actions to Amazon API Gateway so an HTTPS request invokes them. `StartExecution` returns an ARN that identifies the run.

---