What a Microservices Process Orchestration Framework Is and When to Use One

Published 14 September 2021 - ID G00754546 - 28 min read

By Matt Brasier

Initiatives: Integration Architecture and Platforms for Technical Professionals

Microservices orchestration frameworks are an emerging approach for orchestrating processes that involve multiple microservices. Application technical professionals must balance the governance benefits of an orchestration framework against the effort required to build an operational environment.

Overview

Key Findings

- Leaders in microservices architecture are building and using lightweight orchestration frameworks to coordinate the execution of multiple microservices that deliver business outcomes. Many of these frameworks are published as open-source projects.
- Traditional BPM tools do not run on platforms suitable for microservices, nor do they offer the scalability needed to orchestrate processes that use fine-grained services with frequent calls. Service meshes help manage service connectivity, but do not provide end-to-end process coordination.
- Orchestration frameworks provide features for process definition, versioning, analytics and runtime operations. These capabilities are complex and time-consuming to build and support yourself.
- Emerging microservices orchestration frameworks — such as Zeebe, Cadence and Conductor — are not distributed as production-ready services. Significant work is required to configure and deploy them for production use.

Recommendations

Application technical professionals responsible for building microservices applications should:

- Get end-to-end visibility of business outcomes for processes that span multiple, fine-grained service instances by using a microservices orchestration framework.
- Deliver a resilient microservices orchestration capability by deploying a resilient data store, creating operational support tools and processes, and architecting for growth.
- Select a framework with a process definition language that your development teams find easy to use to minimize the learning curve and reduce the complexity of building process orchestration flows.
- Minimize the performance overhead and bottlenecks traditionally associated with orchestration by using a hybrid of process orchestration and choreography.

Analysis

In a highly decoupled microservices application, business outcomes are an emergent property of executing processes that span multiple independent microservices. However, in focusing on building and running their individual microservices, teams may lose sight of business outcomes. Microservices orchestration frameworks provide a set of components to coordinate and monitor the execution of processes that include multiple service interactions. Such frameworks are a new and still emerging technology class, although they can trace their roots to more-traditional integration and process orchestration platforms. This research answers the question:

> **What is a microservices orchestration framework, and when should I use one to coordinate processes in a microservices architecture?**

Do You Need a Microservices Orchestration Framework?

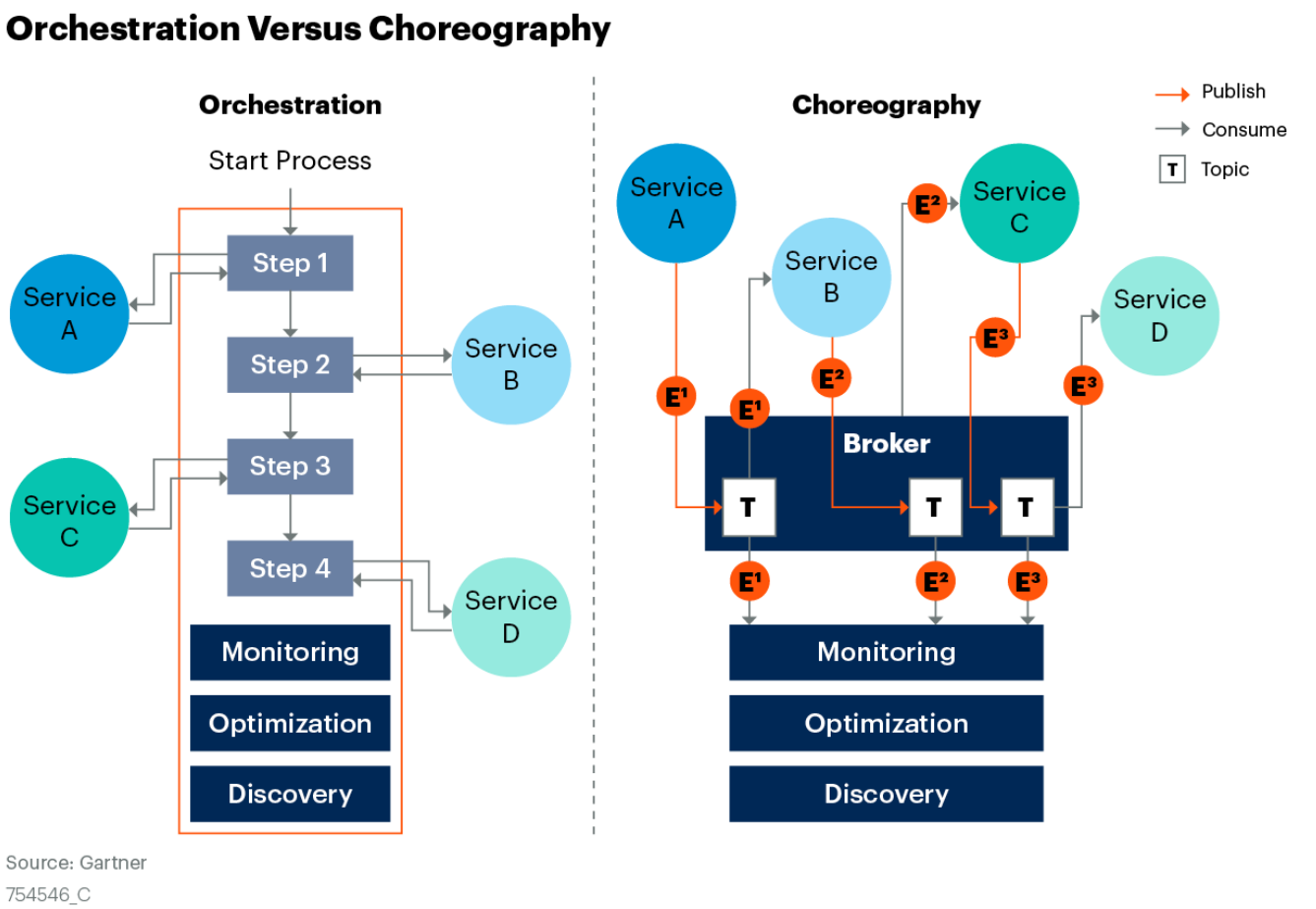
Microservices architecture involves decomposing your application into a set of services to improve agility and allow teams to scale. Each microservice is an independently deployable component with well-defined interfaces, so when you need to implement a change, the scope of change can be limited to a single service. However, it is unlikely that your end users want a set of microservices. Instead, they want an application, API or process to execute, so you must coordinate the execution of multiple microservices to deliver the outcomes that users want.

The saga pattern is a popular microservice pattern that you can use to coordinate a set of microservices to ensure that a consistent business outcome is achieved. Sagas are often used as alternatives or replacements for distributed transaction support, because popular microservices platforms and protocols do not support two-phase commit. Saga coordination is an example of a situation in which the orchestration of microservices is required, and microservices orchestration

frameworks can make a good technology choice when you need to manage a saga. However, microservices orchestration extends beyond just sagas into business process execution.

If you need to execute multiple services to deliver your desired outcome, there are two main approaches you could take. The first is orchestration, in which a central orchestrator component acts as the coordinator and is responsible for invoking each service. The second approach is choreography, in which the services are loosely coupled, and each responds to events or changes of state in the services before it. (Figure 1 illustrates these architectures.)

Figure 1: Orchestration Versus Choreography



Most traditional service-oriented architectures (SOAs) were orchestrated by an enterprise service bus (ESB), whereas others adopted a choreographed approach, based on asynchronous messaging. Unfortunately, the monolithic nature of traditional ESBs, coupled with their high costs, meant that orchestration capabilities became associated with being hard to scale.

When microservices architecture emerged, it quickly became apparent that using a monolithic ESB to orchestrate a set of microservices would not be a sufficient solution. When combined with the more decoupled nature of event-based approaches, choreography was often seen as the preferred approach for combining microservices.

However, choreography has its downside:

- It is easy to forget that microservice functionality can be part of one or more significant process flows. Requirements for parts of the end-to-end application flow that span services owned by multiple teams may not beimplemented, because each team focuses only on its own requirements.
- It is easier for something to “disappear into the gaps” between services. For example, undiscovered errors can cause processes to fail silently, or to produce unexpected results in downstream services.
- It is hard to get good visibility and oversight into the current state of process instances or gather metrics for process monitoring and improvement.

These challenges are the same as those encountered by architects creating enterprisewide business processes by interacting with multiple enterprise applications. The microservices orchestration scenario is on a smaller scale, with narrower scope of process functionality, and with potentially more numerous processes and instances. As such, you can use the same orchestration approach to solve them, but you must implement orchestration in a way that:

- Scales as the number of processes executing increases
- Scales as the number of teams deploying processes increases
- Enables you to meet the agility goals that were the reason you selected microservices orchestration in the first place

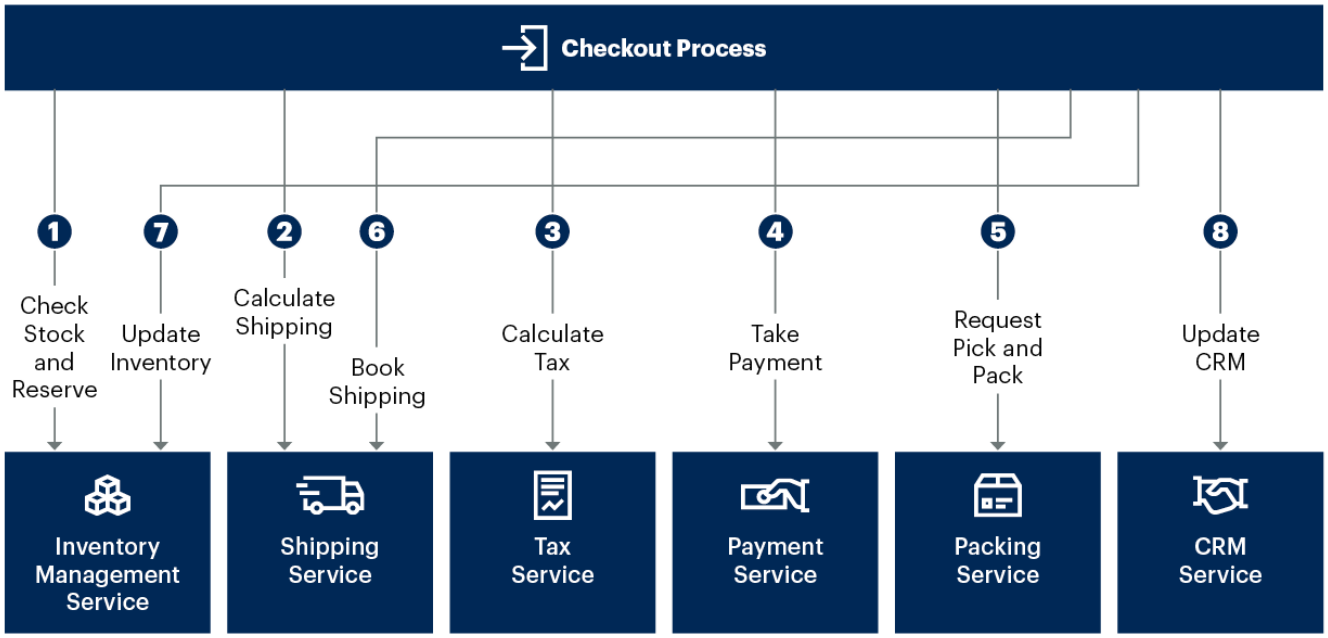
Microservices orchestration frameworks, such as Amazon Web Services (AWS) Step Functions, Azure Durable Functions, Cadence (from Uber), Conductor (from Netflix) and Zeebe (from Camunda), have emerged to offer scalable, low-overhead orchestration.

What Is a Microservices Orchestration Framework?

A microservices orchestration framework provides developers with tools to define, execute and monitor long-running business processes that coordinate multiple service interactions. This includes a process definition language or notation and a runtime capable of executing and monitoring process instances in a scalable and reliable way, using a cloud-native platform/environment. For example, to “check out” a shopping cart without having a dedicated orchestration component (and without using choreography), you may need your mobile application to make the calls illustrated in Figure 2.

Figure 2: An Example Business Process

An Example Business Process



Source: Gartner
754546_C

Gartner

1. Check and reserve stock in the inventory service.
2. Calculate shipping costs in the shipping service.
3. Calculate the tax payable in the tax service.
4. Take payment from the customer using the payment service.
5. Request someone pick and pack the item using the packing service.
6. Book transport using the shipping service.
7. Update the stock levels in the inventory management service.
8. Update the customer purchase history in the CRM service.

You could do this using multiple approaches, including custom code, microservices orchestration frameworks, a more traditional business process management suite (BPMS) platform, custom code in a backend for frontend (BFF) or API composition layer, or an integration technology or framework. The advantages of microservices orchestration frameworks include:

- They are optimized for modern runtimes, such as containers and cloud platforms (where many microservices are deployed).
- They are designed to handle a high throughput of service and process executions.

- They use distributed data stores to minimize the bottleneck created by state persistence.
- They are optimized for use by application developers, rather than professional process modelers or integrators.

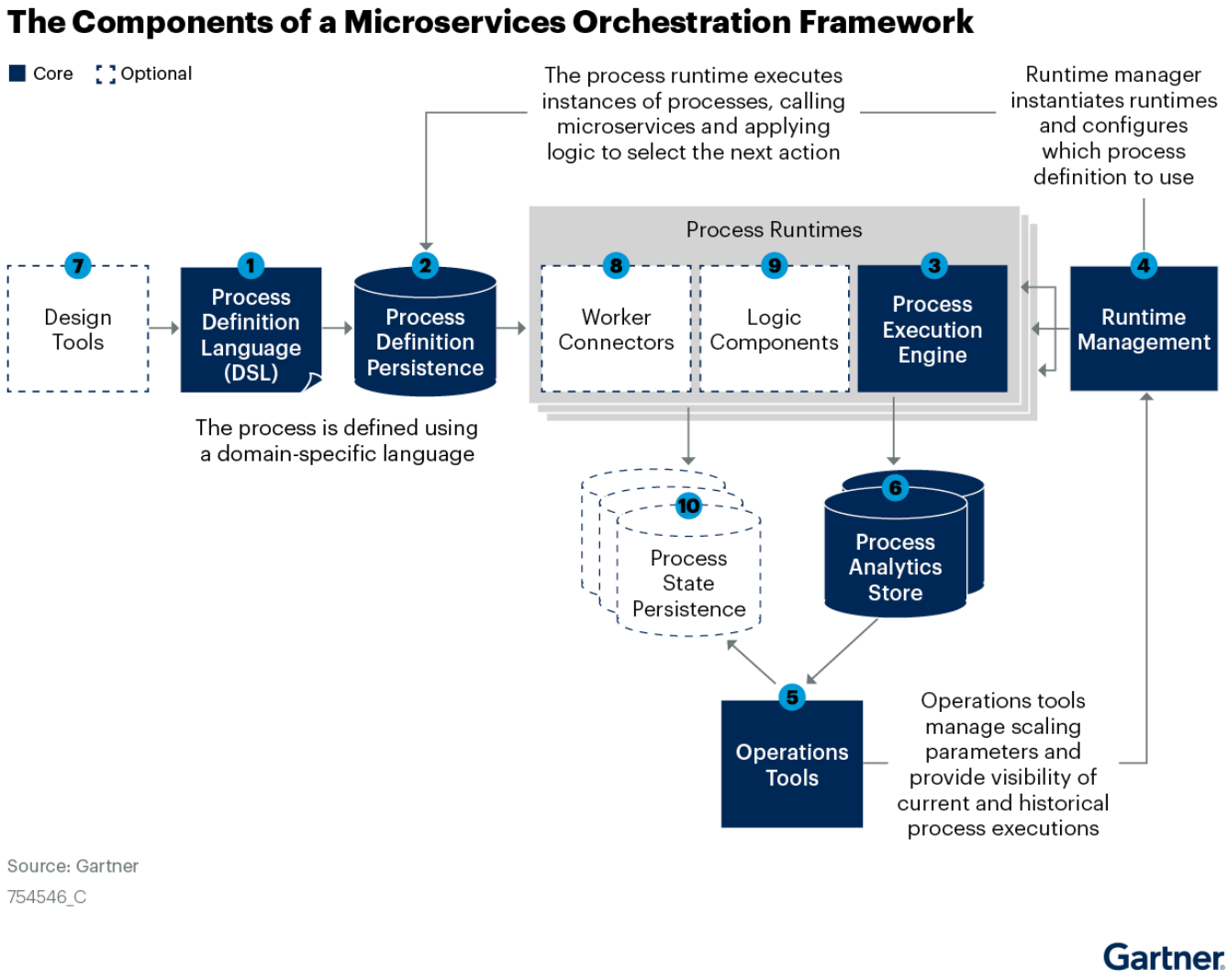
(For more details, refer to the section on What Are the Alternatives to Microservices Orchestration Frameworks?)

Microservices orchestration frameworks are still an emerging technology, and are not generally ready for production use “out of the box” and have limited documentation on how to deploy them in a production environment. However, some easier-to-use packaged commercial distributions are available, such as Camunda Cloud’s self-hosted edition (Zeebe) and temporal.io (a fork of Cadence). In addition, cloud-hosted microservices orchestration services, such as AWS Step Functions and Azure Durable Functions, can be used to orchestrate microservices.You can find more detail on what you will need to do to get your chosen framework ready for production use in the section on Use Patterns When Applying Microservices Orchestration Frameworks.

What Are the Components of a Microservices Orchestration Framework?

The core components of a microservices orchestration framework are a domain-specific language (DSL), process store, process execution engine, runtime manager and operations tools. Some frameworks and platforms offer additional components, including design tools, connectors, logic components, process state persistence and analytics. In addition, some frameworks also include components for monitoring and reporting or process design. The components are illustrated below in Figure 3.

Figure 3: The Components of a Microservices Orchestration Framework



The components are described in the sections that follow.

Process Definition Language

A process definition language is necessary for all frameworks to enable you to define the expected (and alternate) sequence of microservices calls/transactions/requests/invocations that must be executed to deliver the outcome. The process definition language is a key differentiator between frameworks. Zeebe uses Business Process Model and Notation (BPMN), Conductor uses a JavaScript Object Notation (JSON) DSL, and Cadence provides libraries for common programming languages.

> **To maximize developer efficiency, you must treat the process definition languages as a critical factor when selecting your microservices orchestration framework.**

A process definition language enables you to define both the microservices endpoints that must be called, together with the logic operations and data mapping needed between each call.

Process Definition Persistence

The process definitions you create must be stored somewhere to make them available to the runtime engine. Depending on the framework you choose, the process definition store may be provided by the framework (such as in the same storage used for process state) or external (such as a Git repository). The process definition store provides version and life cycle management to allow the development and release of new versions of existing processes.

Process Execution Engine

A process execution engine is a state machine responsible for coordinating the invocation of worker connectors according to the process flow you defined. For microservices orchestration scenarios, the engine must offer high throughput and scalability, because it orchestrates a larger number of more-fine-grained services than a traditional enterprise workflow engine. The process execution engine is responsible for error handling when calling microservices, and will allow you to implement retry policies or conditional logic to deal gracefully with scenarios where microservices fail.

Runtime Management

To deliver the scalability outlined above, the framework must run many workflow instances in parallel. This is typically achieved by having orchestration components run as containers in a container management system and delegating the management of runtime instances to the container management infrastructure. This delegation makes it easier for you to fit a microservices orchestration framework into your microservices runtime environment. Still, it does mean that you will need to do much of the work required to package and tune the orchestration framework components for operational use, rather than relying on a platform that comes configured for out-of-the-box production.

Operations Tools

Operational tools allow you to control your microservices orchestration environment, including deploying new flows, updating existing flows, managing scaling parameters and debugging individual instances. Modern agile development approaches place a high value on continuous and automated build and testing processes, and so microservices orchestration frameworks provide operations interfaces that lend themselves well to automation, such as APIs and scripts. The framework may also include graphical operations tools for either or both of the process administrator and platform administrator roles.

Process Analytics Store

One of the benefits of using an orchestration approach is that it provides a central location to gather historical metrics regarding process execution. These metrics allow you to debug processes, identify opportunities for improvement, plan for future capacity, or make other business decisions. Most frameworks provide an analytics store based on Elasticsearch.

Design Tools (Optional)

Some platforms, such as Camunda Cloud self-hosted edition, provide graphical workflow design tools. Such tools enable you to visually design the orchestration flow, which can be a valuable way to simplify communication between business stakeholders and microservice developers. In addition, for complex orchestration logic with lots of loops, forks or other constructs, a visualization of the orchestration flow can make it simpler for architects and developers to implement the correct flow-of-control.

Worker Connectors (Optional)

To orchestrate a set of microservices, you must communicate with the microservices by sending them requests, events, messages, or some other invocation mechanism. A connector library provides an abstraction of these underlying communication protocols into the terminology of the process definition language. Worker connectors enable developers to invoke microservices. Some frameworks provide standard worker connectors for common protocols used in microservices, such as REST APIs or Apache Kafka topics. Other frameworks do not provide a set of connectors. Instead, they provide the ability to execute workers written in one or more programming languages (allowing you to benefit from client libraries provided for those programming languages). You can then use the libraries and operations available in the programming language to perform the invocation of the actual microservice.

Logic Components (Optional)

To create the flow of control between services, you need to define the logical operations in the flow, such as conditional branches, loops and forks. Some frameworks provide components that implement these features for you (such as the BPMN 2.0 components in Zeebe). Others do not include logic components as part of the framework. Instead, they rely on you implementing the necessary logic in a programming language.

Process State Persistence (Optional)

To provide resilience and support long-running processes, it is necessary to store the state of a process instance, so that it can be resumed later. Traditional enterprise BPMS and workflow systems use a relational database for this persistence, quickly becoming a bottleneck to scaling and performance. Modern frameworks either use a distributed data store or rely on external state persistence, such as having events persisted within an event broker. The use of a distributed data store for state data makes reporting slightly more complex, and so a separate process analytics store is often included. These patterns of state persistence are discussed in more detail in the Use Patterns When Applying Microservices Orchestration Frameworks section.

Table 1 shows how the common microservices orchestration frameworks have implemented the features above.

Table 1: Features of Common Microservices Orchestration Frameworks

Framework	*Created By*	*Process Definition Language*	*State Persistence Technology*	*Runti Platfo Deplo Tools*
Conductor	Netflix	JSON DSL	Dynomite (an abstraction over Redis/Memcached) Elasticsearch	Dync Elast Tom
Cadence	Uber	Java, Go, Python, C#	Cassandra or MySQL, Elasticsearch, Object Store (e.g., S3), Prometheus	Dock tune prod Char
Azure Durable Functions	Microsoft	C#, JavaScript, TypeScript, Python, F#, PowerShell	Azure Storage (Netherite and SQL server in preview)	Micr Azur Kube KED
Step Functions	Amazon Web Services (AWS)	Amazon States Language	Provided by AWS Platform	AWS
Zeebe	Camunda	BPMN 2.0 (model) C#/Go/Java/Kotlin/Node.js (workers)	Internal RocksDB (runtime state) Elasticsearch (history)	Dock Oper Char

Source: Gartner (September 2021)

What Are the Alternatives to Microservices Orchestration Frameworks?

Orchestrating a set of services is not a new problem; in fact, we have been finding ways to perform orchestration for as long as we have been composing systems from discrete subsystems and applications. Previous technologies used for orchestration include integration technology such as ESBs or integration frameworks, BPMSs and messaging solutions. Recent alternatives include integration PaaS (iPaaS) platforms, such as Boomi and MuleSoft Anypoint Platform, and serverless function orchestration frameworks, including Azure Durable Functions and AWS Step Functions. (For more details on the different types of integration technology, see Choosing Application Integration Platform Technology.)

BPMSs and integration frameworks are for orchestrating coarse-grained applications, and lack the scalability required for microservices orchestration in a dynamic scaling environment. A comparison of the general features and capabilities of these technologies compared with microservices orchestration frameworks is shown in Table 2.

Table 2: Microservices Orchestration Frameworks, Compared With Alternatives

Feature	*BPM*	*Microservices Orchestration Framework*	*Integration Framework*
State Persistence	Relational Database Management System (RDBMS)	Distributed data store	None
Supported Service Endpoints	Many protocols and human interactions	Many protocols, but REST and Kafka focus	Many protocols
Design Tools	Graphical	Code-based	Code-based
Process Duration	Long (minutes to years)	Short to medium (minutes to days)	Short (seconds to minutes)
Process Analytics	Detailed and aimed at business users	Limited and aimed at developers	None
Target User	Process modeler	Service developer	Integration developer

Feature	*BPM*	*Microservices Orchestration Framework*	*Integration Framework*
Preferred Runtime	Large virtual machine (VM)	Container	Any
Operations Tools	Graphical Tools	Scripts and APIs	Scripts and APIs

Source: Gartner (September 2021)

There can be exceptions to the above general characteristics — for example, Zeebe includes graphical design tools.

Use Patterns When Applying Microservices Orchestration Frameworks

You can choose from several common patterns when implementing microservices orchestration. Which patterns you use will depend on the resilience, agility and performance needs of your processes. This section examines four common patterns:

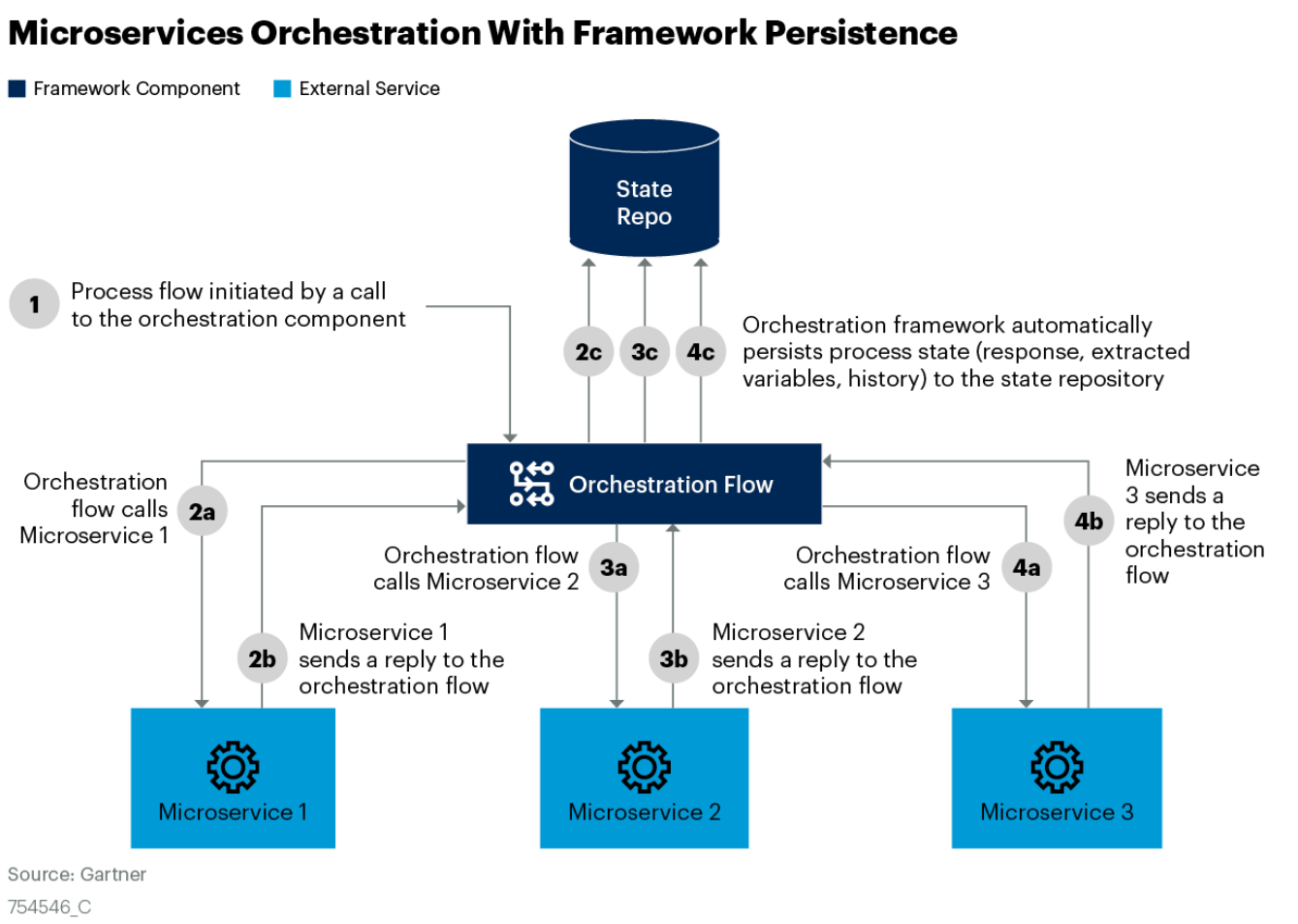
- Orchestration with framework persistence
- Orchestration with message queue persistence
- Orchestration with manual persistence
- Hybrid Orchestration

Orchestration With Framework Persistence

In this pattern, you delegate the persistence of the process state to the framework. The state of a process includes all information required to allow the framework to resume the process in the case of a failure, interruption or just an asynchronous call. Process state includes request and response data, any variables that have been extracted for use in logic decisions, and the results of any logic decisions. Delegating persistence to the framework is the most common way orchestration frameworks are used. (Not having to build a system for persisting process state is the main reason people will adopt an orchestration framework.) However, process state persistence can generate a large amount of load in heavily used applications, and, although microservices orchestration frameworks have minimized the scalability bottleneck, it still adds latency to each microservice execution. Therefore, you may wish to use a different pattern or not

use persistence if you have very high performance requirements. (The pattern is illustrated in Figure 4.)

Figure 4: Microservices Orchestration With Framework Persistence



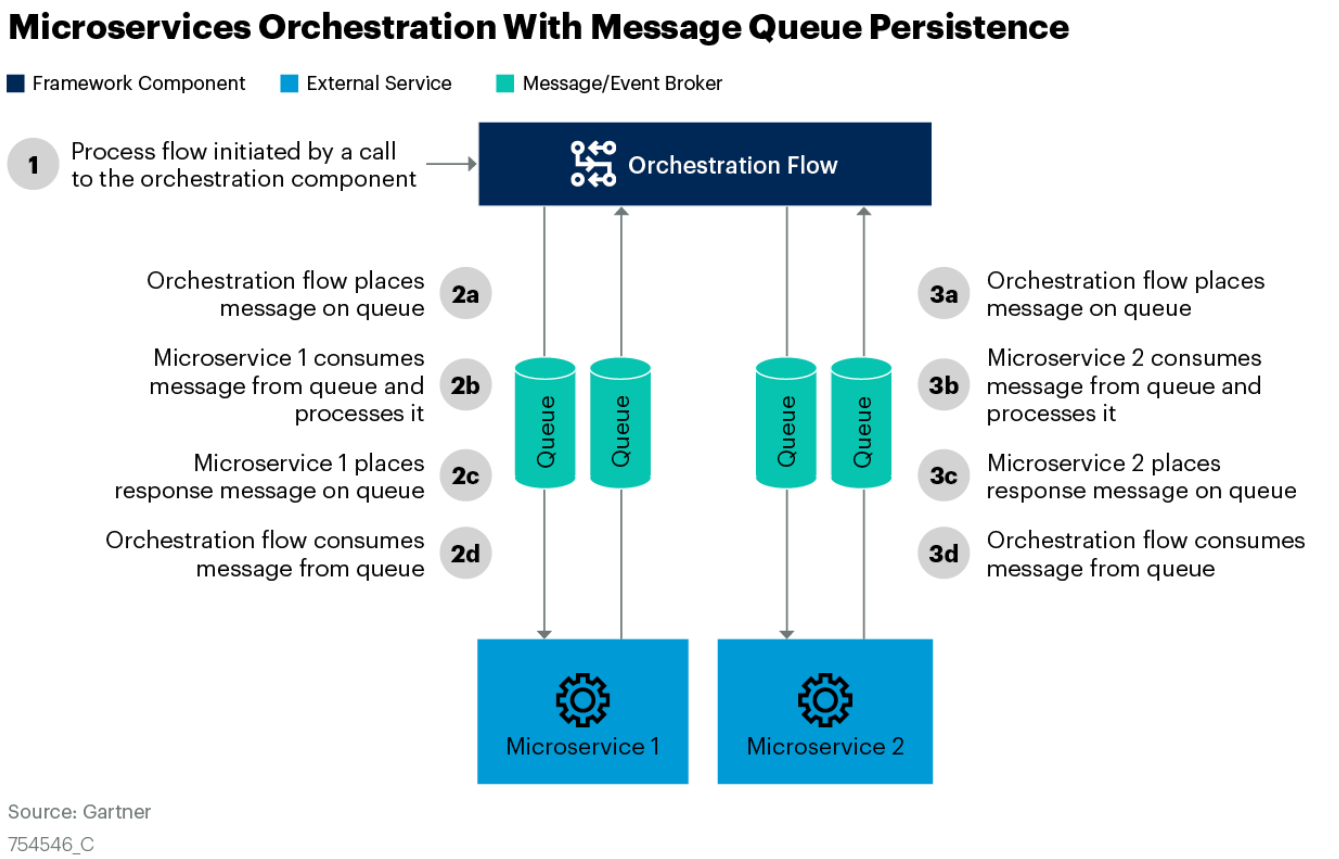
To use this pattern, you must configure the MSOF to automatically persist process state at each step. You initiate the process by calling the orchestration framework (using an API or client library) and telling it which type of process to start. The framework loads the relevant process definition and begins executing the required steps.The framework ensures that the state repository is updated whenever the process state changes (due to the successful execution of another step). If the framework fails to successfully execute a service, then it is able to automatically recover or take a compensating action, according to how you have configured the worker connector and overall process flow. If the process orchestration framework runtime fails, and is restarted, the process instances will resume from the last stored state.

Orchestration With Message Queue Persistence

An alternative to having the framework take care of your state persistence is to build your process using messages that carry the current state of the process instance (see Essential Patterns for Event-Driven and Streaming Architectures). In this approach, the state of each process instance is retained because the messages are persisted in your queues (or topics). If a microservice or the orchestration flow fails, messages will remain in the queue until the service becomes available

again. Your microservices will need to deal with the quality of service (QoS) provided by your message broker, which may include having to handle duplicates or out-of-order messages. (Figure 5 illustrates this pattern.)

Figure 5: Microservices Orchestration With Message Queue Persistence



Communication among microservices may follow either an event-driven or a request-reply pattern. However, in either case, the communication occurs via a stateful messaging platform. This approach is not choreography, because there is still an orchestration component sitting between the microservices. The orchestrator receives the messages and makes decisions about the next service to execute, and generates the next event or message. Once the orchestration framework has placed the request message on the queue, it is the responsibility of the message-oriented-middleware (MOM) to ensure that the microservice executes successfully. If the orchestration framework runtime fails and is restarted, any messages that have not been acknowledged will be redelivered, allowing the process instance to resume from where it was.

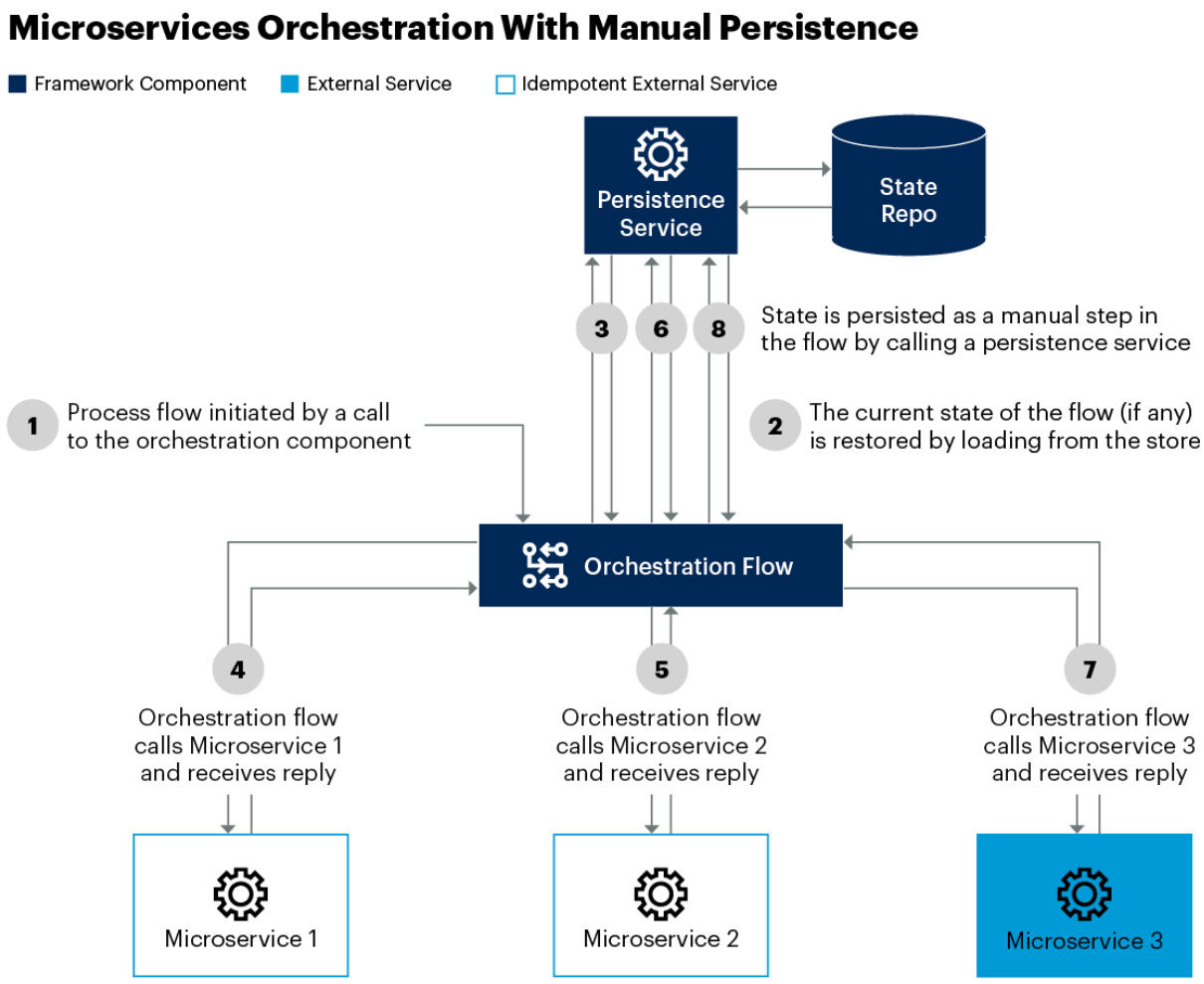
Orchestration With Manual Persistence

If you have high-throughput performance requirements that make persisting all of the process state at every step impractical, and don't want to use message queues for persistence, then you can manually persist the process state to get more control. In this approach, you define your own state store and schema. You then manage state by performing the following tasks in your flow:

- Disable automatic state persistence in the framework.
- Whenever a request is received relating to a process flow, you must check for existing state data and restore the process state if required.
- At key points in your process flow, you write updated information regarding the process to your store with a manually configured step, such as a call to an API.

Figure 6 illustrates the manual persistence of process state.

Figure 6: Microservices Orchestration With Manual Persistence



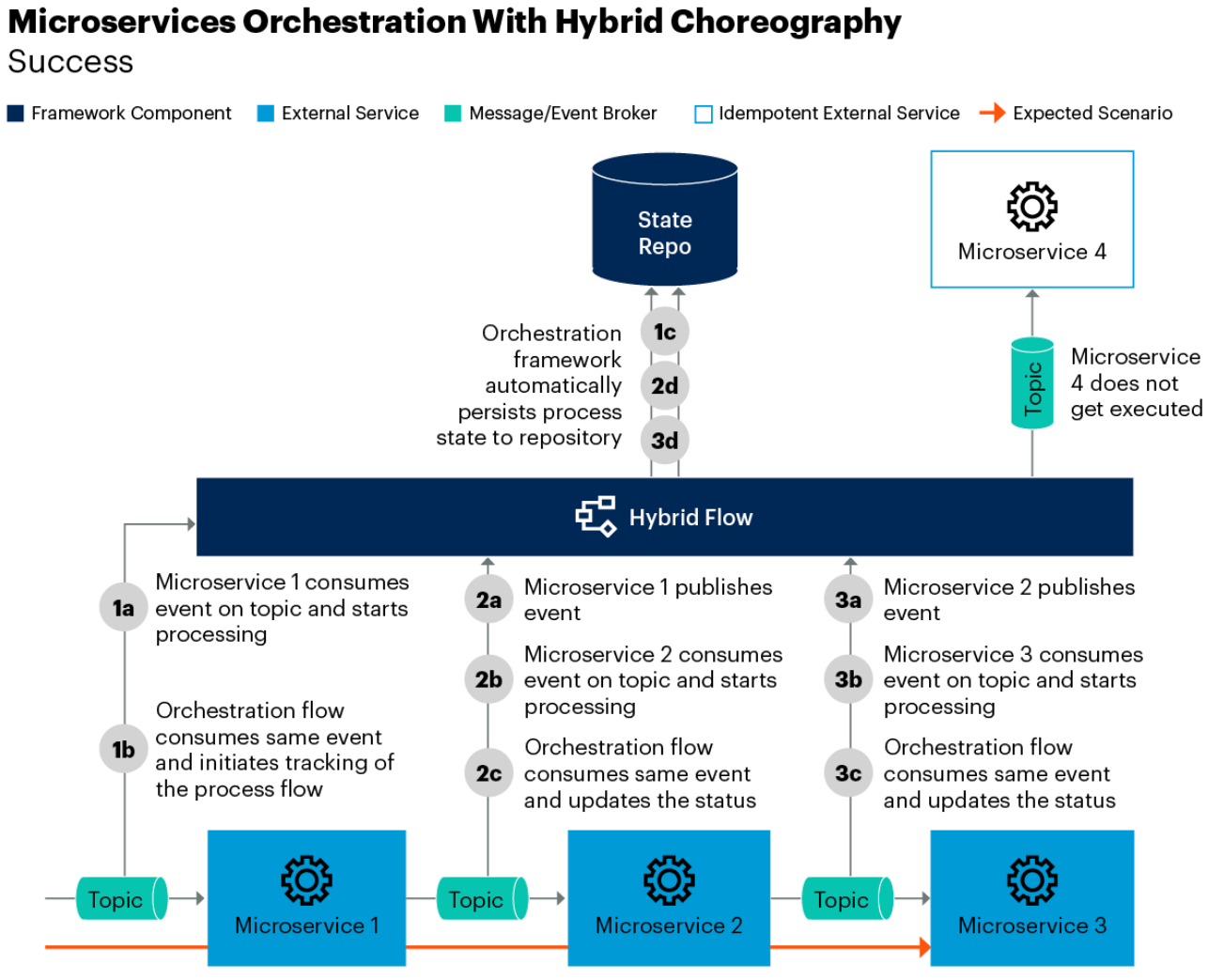
This is quite a bit of work to perform yourself. Still, if you only have a few situations in which state persistence is necessary, then, by manually persisting state, you have more control over how often state is stored and loaded. Therefore, you can balance state persistence and performance, according to your requirement. For example, if you have many steps in your process flow that are idempotent, such as performing calculations, you do not need to persist the state after each of them. If a failure occurs, then you will need to perform the calculations again; however, by not persisting at every step, you can improve performance.

Hybrid Orchestration

The hybrid orchestration pattern combines the visibility of an orchestrated approach with the scalability of choreography. With hybrid orchestration, you still define the process flow in your orchestration framework, but the orchestrator observes most of the execution of the flow, rather than driving it. This observation will often be in the form of consuming events from a topic, but can include observing data changes in databases or anything else the framework can detect.

To create a hybrid orchestration flow, you define the "happy path" process execution in terms of the sequence of things that you expect to observe, together with timings. If the framework observes the right things in the correct sequence, it records state changes in an audit log, and the process executes via choreography. However, suppose the framework observes a known error state or does not observe an anticipated event during the expected time frame. In that case, the orchestrator can initiate an action, such as a compensating transaction or initiate an alternative flow. Figure 7 illustrates the hybrid orchestration pattern in a scenario where the flow proceeds as intended, and the orchestration engine stores only state and metrics.

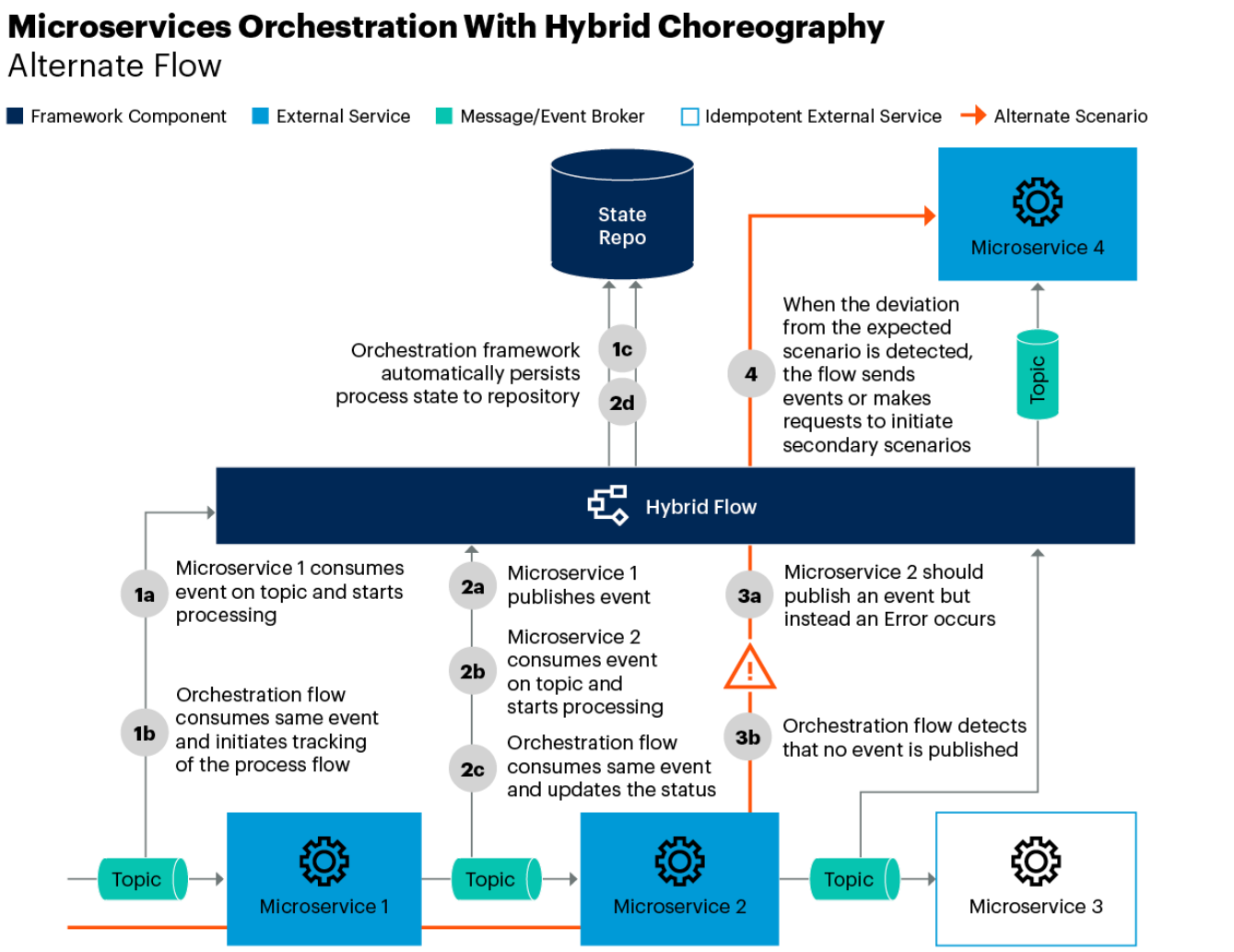
Figure 7: Microservices Orchestration With Hybrid Choreography — Success



Source: Gartner
754546_C

Figure 8 shows the same example, but, this time, with a scenario in which an error occurs that means the flow does not proceed as expected. The orchestration component identifies the failure and calls Microservice 4 to initiate an alternate flow.

Figure 8: Microservices Orchestration With Hybrid Choreography — Alternate Flow



The hybrid orchestration model combines many of the advantages of orchestration (the central visibility and the opportunity to initiate alternate flows or compensating actions) with much of the scalability of orchestration. The latter includes the orchestration component not being within the execution path for the most common scenario, so it doesn't become a bottleneck. However, you also have some of the drawbacks of orchestration, such as:

- A tighter coupling between services, because the orchestrator defines which services are executed in which order.

- The need to define your process flow upfront, and software changes to the process flow that require testing and deployment.
- Events emitted by microservices components that must be understandable to the orchestrator.

Strengths

Microservices orchestration frameworks are new, but they build on the well-established principles of process orchestration and integration. Using a microservices orchestration framework delivers the following benefits:

- **Central visibility of process definition, status and metrics —** The orchestration framework can capture detailed information about each executed process instance, and make that available for analytics. This allows you to answer questions about specific instances (such as "Where is my order?"), as well as analytical queries (e.g., "When was the peak in new orders during the past seven days").
- **Microservices aligned with your enterprise process automation strategy (such as a hyperautomation strategy) —** By modeling your applications as processes, it is easier to understand how your microservices applications fit into a wider set of enterprise business processes. Treating your set of microservices as a process improves your understanding of where a microservice fits into a value chain, and allows you to feed metrics "upstream" for process optimization initiatives.
- **Scalable orchestration on cloud-native platforms —** Microservices orchestration frameworks are specifically optimized for use cases associated with a large number of independent services coordinating transactions across service boundaries.
- **Fit with a developer-centric workflow —** Created by developers for developers, microservices orchestration frameworks fit well into existing development pipelines that include continuous integration/continuous delivery (CI/CD), as well as test and deployment automation.

Weaknesses

The disadvantages of adopting a microservices orchestration framework are:

- **The high overhead of building and operating a production orchestration framework environment —** The modeling processes and the managing of processes as services also produces high overhead, if you only have a few processes to orchestrate.
- **The need to invest more time in building and deploying a production-ready platform than traditional orchestration technologies, such as ESB and BPMS —** You must have the expertise to design a production-ready deployment of a microservices orchestration framework. Otherwise, you need to use a vendor product. The open-source projects are packaged to optimize for developer experience (getting up and running quickly). The documentation and

community support for establishing and sustaining operational stability or security using these OSS frameworks is lacking.

- **A fast-changing, immature market for products that means today's popular choices may not remain popular** — The relative popularity of frameworks is changing rapidly, and new generations of frameworks may arrive that provide a smoother adoption and operations experience.
- **Few vendor support options** — Cadence and Conductor have been created by companies at the leading edge of microservices architecture (Uber and Netflix), then contributed as open-source projects. However, these companies do not offer support or professional services options on these frameworks.

Guidance

To successfully adopt a microservices orchestration framework, follow the guidance below.

When to Use a Microservices Orchestration Framework

Microservices orchestration frameworks offer a compelling solution to the common problem of coordinating stateful processes that span multiple service boundaries. However, they are an emerging technology, and deploying a microservices orchestration framework in production requires that you have an experienced platform operations capability (see Using Platform Ops to Scale and Accelerate DevOps Adoption). This is because a microservices orchestration framework operates as a shared runtime used by many distinct microservice development teams. You must also be able to design and deploy resilient services for state persistence, automated build and deployment, and operational governance.

If at least three of the following statements are true, then you should implement a microservices orchestration framework:

- You have multiple microservices that must be invoked as part of a coordinated process to deliver a business outcome, but there are multiple pathways in which the services may be invoked (governed by some business logic that defines the order).
- You need to be able to track and monitor the progress of individual instances of the process flow as (and after) they execute.
- You have a mature platform operations capability, and can deploy and implement the required platform components.
- You are happy to accept a slight increase in coupling (because your sequence of services is defined in your orchestration flow, rather than being dynamically choreographed) to achieve improved governance and monitoring of your flow.

When to Use Another Framework

Microservices orchestration frameworks solve a specific class of problems in a way that works well for organizations with strong platform ops and microservices skills. There are several strong indicators that microservices orchestration is not a good fit or necessary for your requirements:

- If you are looking for a platform for enterprise process automation of wide-scoped business processes, you should use an intelligent business process management suite (iBPMS).
- If you do not have a mature platform ops team with experience designing and building production-grade containers and distributed data store clusters, you should wait for the technology to mature).
- If you have a small number of microservices that just need to execute a simple logical flow, then you should create a simple orchestration service using custom code and use log aggregation across all services to get visibility.
- If you want to build a user experience (UX) on top of a small set of service calls, then you should build a custom JavaScript user interface (UI) or use a low-code application platform (LCAP).

Many of these cautions are based on the relatively low maturity of microservices orchestration frameworks and the amount of work required by you to deploy a production-ready environment and configuration (see the How to Build an Operational Orchestration Platform section). As such, waiting for the technology to mature before adopting a microservices orchestration framework will negate several of these pitfalls.

How to Build an Operational Orchestration Platform

The current generation of microservices orchestration frameworks provide the core requirements for microservices orchestration. However, if you want to use them in a consistent way across your enterprise, then you must build additional support structures around them. You will need deployment architectures, scripts and tools, and a set of support procedures and approaches that bring operating the framework in line with other parts of your microservices platform. This may include creating your container images, deployment scripts or environment configurations, as well as designing an architecture that supports failure and disaster recovery, and secures access to the platform and process instances.

> **Microservices orchestration frameworks lack the critical governance and management capabilities necessary for production environments.**

Although some microservices orchestration frameworks are available in hosted (such as Camunda Cloud) or commercially supported (e.g., Temporal and Camunda Cloud self-hosted

edition) distributions, the open-source streams are not distributed ready for operational deployment. For ease of testing and development use, they are often distributed in a Docker image or set of images without security controls and optimized to get you using the tool quickly, rather than securely and at scale. You must perform the following tasks to create a production-ready deployment of a microservices orchestration framework:

- Build data store environments that will scale to meet your current and anticipated needs, using tools like Redis or Memcached.
- Create and tune process runtimes to ensure they are optimized for production use and configured to connect to your data store environments.
- Ensure communication between process runtimes, data store environments, and operations and management tools are secure.
- Define a process and create tools that allow operators to identify and resolve process failures.
- Define process versioning policies and implement an approach to zero-downtime process deployments.
- Create a set of operations scripts to deploy, remove, retire and version process flow definitions.
- Create a set of reports to gather process analytics.
- Create a set of scripts (or even a UI) for operational support and debugging of failed process instances.
- Create test harnesses and stubs that allow you to test process flows with the microservices stubbed-out and the actual microservices.

Select Your Orchestration Framework-Based Developer Skills

One of the primary differentiators among the current microservices orchestration frameworks is the language in which developers define their processes. For example, Zeebe uses BPMN 2.0, Cadence uses a flowing DSL in a choice of programming languages, and Conductor uses a JSON-based language. The purpose of using an orchestration framework is to simplify the process of creating and maintaining orchestration flows. You need to match the process definition language to the skills and preferences of your process engineers. If your processes will be created by microservices developers who are used to programming in Go, you may find they prefer Cadence (or Temporal). However, if your teams include business analysts, or your teams spend time sketching out process flows before implementing them, then Zeebe might work better.

Treat Orchestration Flows as (Micro)services

A common pitfall of microservices architecture is that each team concentrates only on its own service, but no one owns the end-to-end business outcomes. This can lead to a situation in which there are no defined requirements and no tests for the only thing the business users care about.

Each implemented orchestration flow is itself a composite service, and may (if it follows the principles of microservices architecture) be a microservice. This means your orchestration flows have their own life cycles, product ownership and development backlog, and are owned by a specific team, rather than being an afterthought once all the low-level microservices have been delivered. These orchestration microservices must be owned by teams responsible for the end-to-end flow. This ownership includes the requirement to define and execute tests across the flow and be responsible to update and operate the flow, resolving process-related issues in production.

Recommended by the Author

Comparing Digital Process Automation Technologies Including RPA, BPM and Low-Code

Decision Point for Process Automation Platforms

Designing Services and Microservices to Maximize Agility

Solution Path for Applying Microservices Architecture Principles to Application Architecture

Solution Scorecard for Microsoft Azure Cloud Application Platform Services

Solution Scorecard for Amazon Web Services Cloud Application Platform Services

Essential Patterns for Event-Driven and Streaming Architectures

Recommended For You

Decision Point for API and Service Implementation Architecture

Essential Patterns for Event-Driven and Streaming Architectures

Working With Data in Distributed and Microservices Architectures

Decision Point for Mediating API and Microservices Communication

Designing Services and Microservices to Maximize Agility

Supporting Initiatives

Integration Architecture and Platforms for Technical Professionals

Track