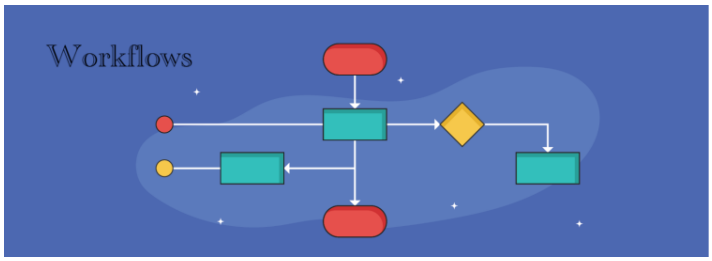


Workflow-Oriented Software Architectures

Tamer Khraisha, Ph.D.
Software and Data Engineer | Cloud Computing | Python | Machine Learning |...
13 articles + Follow

August 23, 2022

Open Immersive Reader

As a natural part of the growth journey of software-based companies, the technological stack tends to increase in size and complexity, generating many interdependencies and interactions between the different components of the system. For this reason, the software development community has identified the need for architectural designs and tools that organize **software transactions** into **workflows** that can be defined, coordinated, managed, monitored, and scaled reliably following business logic.

In this article, let's look at some of the main elements of workflow-oriented software architectures, their range of applications, as well as the frameworks and tooling available for implementing such patterns.

The components of a workflow-oriented architecture

- **Microservices**

In the old days, companies used to build applications that advocate a central **Business Process Management** (BPM) that acts as the workflow engine, which then communicates with the services via a messaging system often referred to as **Enterprise Service Bus** (ESB). This software pattern belongs to a more general style of software development

called **Software-Oriented Architecture (SOA),** which has been pioneered by big software vendors. However, due to its shortcomings (centralized hierarchy, lack of CI/CD integration, vendor lock-in), the SOA pattern is increasingly becoming the kind of solution companies want to avoid, and this is where microservices come into the scene. In a few words, the idea of microservices is that the application and business logic are broken up into a set of small, lightweight, autonomous services that work together. Microservice architectures are designed with the goal of achieving **low coupling**, **high cohesion**, **efficient use of resources**, and **focused scope**.

- **Distributed Systems**

A distributed system is a design pattern that involves a collection of software components (often storage and compute) located on a different number of networked machines (potentially across multiple regions) that communicate and coordinate their actions to complete a job and achieve the same result. Distributed systems allow for **scalability**, **high-performance**, **load balancing**, and **fault tolerance**, while at the same time presenting challenges like **consistency**, **concurrency**, and **clock synchronization**. With the growth of cloud services, building distributed systems has become much easier as the managed offerings of cloud providers have removed much of the configuration setup and security measures required for building distributed systems.

Distributed storage systems can be implemented with different design patterns and using a variety of technologies, for example with HDFS clusters, Cloud Storage (S3, Google Storage, Azure File System), and distributed databases (e.g. Cassandra). Similarly, distributed compute systems can also be implemented with a Kubernetes cluster that orchestrators containerized applications, managed container services, and distributed computing frameworks like Apache Spark, MapReduce, and Apache Storm.

- **Communication Patterns**

In a coordinated workflow system, the components of the system are supposed to be able to communicate with each other. In this regard, two communication patterns are worth mentioning: First is the **Synchronous Blocking Pattern**, where one component makes a call to another component

and blocks its operation while waiting for a response. Systems designed using **HTTP** and **RESTful** APIs are the most common realization of this pattern. Second, in an **Asynchronous Non-Blocking Pattern**, a component emits a call to another component but it does not expect to receive a response (except maybe a confirmation message), therefore it can carry on processing other tasks; Systems designed with message brokers like **PubSub** or **Apache Kafka** are examples of tools that can implement this communication pattern.

- **Coordination Logic**

In a multi-component system such as workflow-based systems, components interact with each other following a well-defined logic that should make sense from **technical** and **business** logic points of view. Oftentimes there is a tradeoff between the amount of business logic and tthe echnical reliability of workflow systems. The more complex the business logic that one desires to implement in a workflow, the more likely that constraints such as **operational control**, **inconsistency**, **concurrency**, and **traceability**, will emerge.

- **Programming Paradigm**

A programming paradigm is often chosen to represent the style used in the implementation and organization of a workflow solution. Among the most popular workflow paradigms is the **Dataflow** paradigm, which belongs to the declarative paradigm and models a program as a **Directed Graph** or **Directed Acyclic Graph** (DAG) of operations. In the dataflow model, a graph is created to represent the execution topology where nodes are operations that can represent a microservice, a bash operation, database operation, or HTTP request, and edges design the order and topology of execution of the different operations in the flow. The dataflow model differs from classical more general software sequential models in that it emphasizes the **movement of data** as a sequence of input/output from one step to another.

Types of workflows

- **Extract, Transform, Load (ETL) Workflows**

ETLs are perhaps the most well-known type of workflows. An ETL is a three-phase process or pipeline where the data is extracted from one or more sources, transformed following a business or structural logic, and finally stored in a target destination. ETL workflows are common in Data Engineering and they are mostly batch workflows, meaning that they run according to a trigger or schedule and not continuously.

A variety of tools exist for ETL workflow management, including Commercial Enterprise tools (e.g. IBM DataStage and Oracle Data Integrator), Open-Source tools (such as Apache Airflow), Cloud-Based tools (e.g. Amazon Glue), and custom tools.

A number of features are often sought-after when choosing the right tool for ETL pipelines including ease of use, graphical interface, flexible workflow design (different flow structures), advanced dependency modeling (intra-flow and inter-flow), plugins and operators, reliability, and deployability, among others. In this regard, **Apache Airflow** has emerged as one of the most popular open source solutions, which is also offered as a managed service by cloud providers like **Google** and **Amazon**. The main advantage of Apache Airflow is that it's an **infrastructure-as-code** tool, where workflows can be defined as DAGs written in Python and using a wide range of operators, with flexible flow design (**flow structure** and **Cross-DAG dependencies**), monitoring tool, and operational control options. On the other hand, the main drawback of Apache Airflow is the lack of a feature to pass data from one task to another, thus not properly implementing the dataflow model. An alternative, called **Prefect**, endeavored to address this issue by allowing tasks to take input and produce output that can be passed to other dependencies, thus **conforming** to the dataflow model.

- **Microservice Workflows**

Microservice workflows are a more complex form of workflows, where different microservices that incorporate different business logic are coordinated together to perform a series of operations towards an end goal. For example, in an online store, one microservice might be responsible for acknowledging the arrival of an order, another checks the payment, and the next reserves the stock until the order is fulfilled. Microservice workflows are in many cases deployed

at a large scale and therefore require high performance, scalability, and resiliency standards.

Microservice workflow solutions try to achieve reliability by decoupling workflow definition from task implementation, meaning that the structure of the workflow (the tasks that it executes and the order and dependencies between them) are defined in a workflow definition (often JSON or YAML) and the execution of the tasks happen by calling the related microservice or application.

A variety of tools exist for microservice workflows, where the most popular one is perhaps **Netflix Conductor**, a tool created by Netflix that implements a rich set of features and abstractions that allow for the definition, monitoring, and execution of microservices. On the cloud side, tools such as **Google Workflows** and **Amazon Step Functions** are offered as managed workflow management tools that integrate easily with other cloud services such as cloud functions, APIs, storage, and databases.

- **Machine Learning Workflows**

In Machine Learning (ML), projects are often structured as a set of sequential steps which apply different operations on an input dataset to produce predictions. Examples of machine learning operations are data loaders, transformers, estimators, and predictors. Such design is what powers machine learning software like **scikit-learn**, Spark MLib, Keras, and many others. ML software libraries offer the option to create ML workflows via the concept of **pipeline**, which assembles together a set of steps that perform different transformations, validations, and predictions.

For production purposes, a variety of machine learning tools have been developed, mostly available as open source. Examples of such tools are **Metaflow** (a Python library based on the dataflow model), **Kubeflow** (for deploying ML projects on Kubernetes), **Tensorflow Extended** (an end-to-end platform for deploying production ML pipelines), and **MLflow**. Among the desired features of ML flow systems are **deployment**, **reproducibility**, **model artifact registry**, **experimentation**, **speed**, and **ease of integration** with external services such as databases and compute clusters.

- **Stream Processing Workflows**

In a streaming data workflow, data is assumed to be flowing in real-time and each data arrival is handled separately as an event in a continuous stream of events. Stream workflow management solutions emphasize scalability, large volume streams, in-memory processing, even-driven patterns, and asynchronous data processing. Technologies involved in stream workflows often belong to the big data family of tools; this includes messaging systems such as Apache Kafka or Amazon Kinesis, real-time computing frameworks such as Apache Storm or Apache Spark, and distributed databases such as Cassandra.

Multiple distributed real-time computation frameworks offer the flexibility to model the execution of a set of operations following a topology or structure that developers can build. For example, Apache Spark allows the creation of complex job structures and executing them in a Directed Acyclic Graph (DAG). Apache Storm is another interesting tool for real-time computing which allows for designing an application as a "**topology**" in the shape of a DAG with **spouts** and **bolts** acting as the graph vertices and data transformation steps as the directed edges.

Workflow Coordination: Orchestration and Choreography Patterns

In workflow management systems, and in particular microservice-oriented systems, the concepts of orchestration and choreography are often encountered as they are used to describe how coordination is achieved.

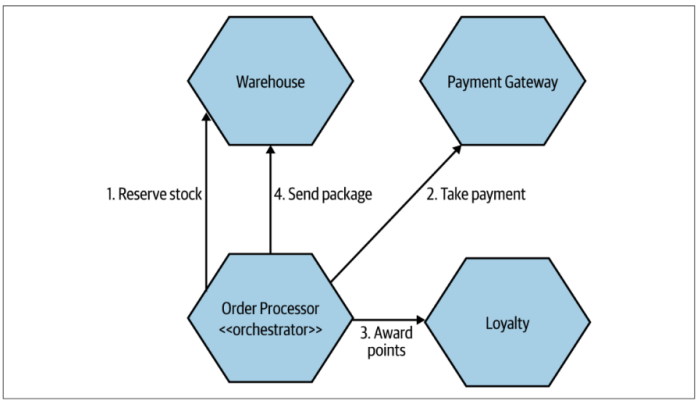
The main high-level difference between choreography and orchestration is that orchestration revolves around **commands** while choreography is built around **events**. A component or service can emit an event or a command. An event does not know who picks it up or why, it simply doesn't care. If instead the emitting component wants something to happen, what it sends is not an event, but a command. It is important to note that commands and events are characterized by their **semantics** only, not the technical protocol. This means that the same technology (e.g. messaging broker) can be used to emit events and commands (both being messages).

- **Orchestration**

Orchestration can be defined as a **command-driven communication** system that coordinates services. We can think of orchestration as a command-and-control approach: one or more **orchestrators** control what happens and when, and with that comes a good degree of visibility and traceability of what is happening at the system level. Whenever a service or component coordinates one or more services through commands, then we have orchestration.

Another important aspect of orchestration is that it **does not need to be central**. Orchestration simply means commanding (or coordinating) another component. Every component can do this; it is not about having a central unit of orchestrator. Nevertheless, implementing a central coordinator (orchestrator) is the most practical and common practice in orchestration. The orchestrator is a component or service that defines the order of execution and triggers any required compensating action. **The involved services do not "know" (and do not need to know) that they are involved in an orchestration** process and that they are taking part in a higher-level business process. Only the central coordinator of the orchestration is aware of this goal, so the orchestration is centralized with explicit definitions of operations and the order of invocation of the services.

For example, as shown in the figure below (source Newman, 2021), we could have a central Order Processor, playing the orchestrator role, to coordinate a fulfillment process. The orchestrator knows what services are needed to perform the operation, and it decides when to make calls to those services. If the calls fail, it can decide how to handle errors. In general, orchestrated systems tend to make heavy use of request-response interactions between services: the Order Processor sends a request to services (such as a Payment Gateway) and expects a response to let it know if the request was successful and provide the results of the request.



- **Choreography**

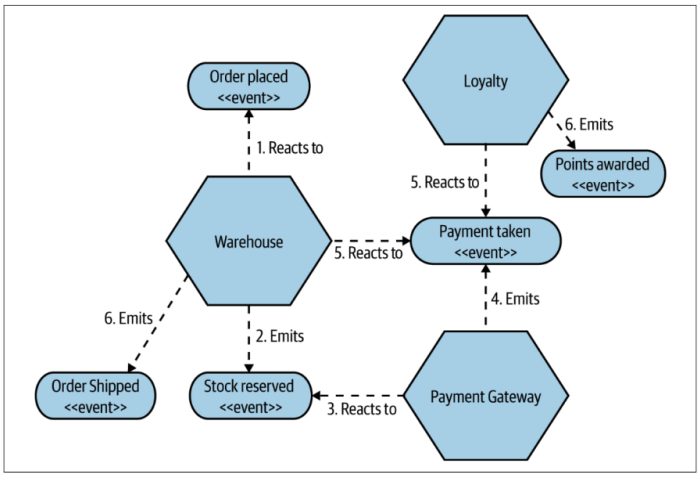
In choreography, services interact **directly** with each other following an **event-driven** communication style in order to achieve a common goal. An important aspect of choreography is that it focuses on a **single communication link** and not on the system as a whole. This means that it rarely makes sense to say that one designed a "choreographed system".

Each service in a choreography may be seen as an orchestrator of the collaborating services. All services in the choreography need to be aware of the business process, the services with whom to interact, operations to execute, messages to exchange, and the timing of message exchanges. Choreography is a collaborative effort focusing on the exchange of messages in public business processes.

A choreographed architecture aims to distribute responsibility for various tasks among multiple collaborating services. If orchestration is a command-and-control approach, choreographed systems represent a **trust-but-verify** architecture. Choreographed architectures will often make heavy use of **events** for communication between services.

An example is illustrated in the figure below (source Newman, 2021). When the Warehouse service receives that first Order Placed event, it knows its job is to reserve the stock and send an event once that is done. If the stock couldn't be received, the Warehouse would need to raise an error event (an Insufficient Stock event, perhaps), which might lead to the order being rejected or canceled. When the Payment Taken event is produced by the Payment Gateway, it causes reactions in both the Loyalty and Warehouse microservices. The Warehouse responds by

dispatching the package, while the Loyalty microservice reacts by awarding points.



References

Newman, S. (2021). *Building Microservices*. " O'Reilly Media, Inc.".

Report this

Published by

Tamer Khraisha, Ph.D.
Software and Data Engineer | Cloud Computing | Python | Machine Learni...
Published • 1y
13 articles
+ Follow

A short introduction to software patterns and tools for workflow-oriented software architectures #software #workflows #orchestration #pipelines #dataflow

Like Comment Share 17 2 comments

Reactions

+5

2 Comments

Most relevant

Add a comment...

Renato Dinis • 3rd+
Cyber Security Specialist na Next Gate Tech
1y

This will help me

Like · 4 | Reply · 1 Reply

Tamer Khraisha, Ph.D. • 3rd+
Software and Data Engineer | Cloud Computing | Python | Machine Learning | Financial Technology
1y

If it helps I'll write often then

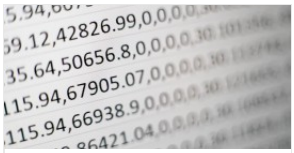
Like · 1 | Reply

Tamer Khraisha, Ph.D.

Software and Data Engineer | Cloud Computing | Python | Machine Learning | Financial Technology

+ Follow

More from Tamer Khraisha, Ph.D.



Database Engine Ranking and Popularity in 2023

Tamer Khraisha, Ph.D. on Li...

Commoditization of Artificial Intelligence: AI-as-a-Service

Tamer Khraisha, Ph.D. on Li...

Top Artificial Intelligence and Machine Learning Scientific Journals

Tamer Khraisha, Ph.D. on Li...

See all 13 articles