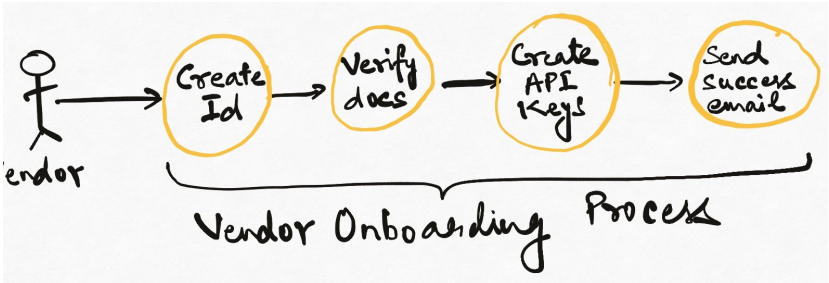
Architecture Pattern: Orchestration via Workflows

Kislay Verma · December 13, 2020

—

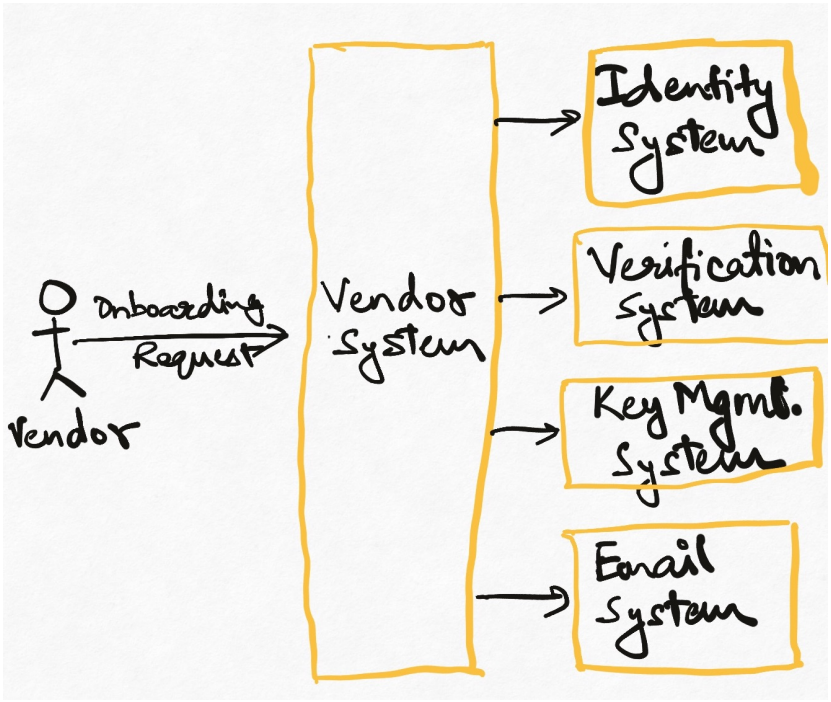
The problem in modelling business processes

Most business processes involve doing multiple things across multiple systems. Imagine onboarding a new vendor onto a B2B platform. When a vendor is onboarded, we might have to create an identity for it, trigger some sort of verification process, issue new API keys and credentials to it, and so on. All of these steps typically happen across multiple teams and technical modules.



Or imagine something with a shorter life span like an item getting ordered on an eCommerce site. An order has to be created, payment accepted, inventory blocked, and a confirmation email sent – arguably in an all-or-nothing manner.

How should we implement such systems? One way to do this in the object-oriented/REST style is to identify a *primary entity* (Vendor/Vendor Service in the former example, Order/Order Service in the latter) which owns the entire operation. This primary entity invokes all the other components involved to make sure that the process runs end to end without any problems. If the process is asynchronous in some way (e.g. if verification of vendor documents is handled manually offline), the primary entity carries the state necessary to make sure that it can continue the process from where it was paused.



In many cases, this is a perfectly acceptable design. It keeps things simple. However, as systems get larger and business processes get more complex, some problems start emerging. One technical problem that can already be seen is that the business logic in the primary entity is very tightly coupled to all the other entities. As any part of the

order taking process changes, the logic in the primary entity has to be updated continuously. The team which owns the primary entity is now forever in the path of every team which wants to modify any part of the business process. Roadmap/bandwidth negotiations abound.

Another problem is imposed by microservice and other distributed architecture patterns. Not only are vendor/order services logically coupled to inventory, verification, email and other services, they are also obliged to handle the vagaries of communication over the network and the various kinds of error handling and performance overheads that distributed systems (https://kislayverma.com/tag/distributed-systems/) are prone to. Orchestration of operations across multiple services (https://kislayverma.com/programming/design-review-checklist-for-distributed-systems/) can be a very challenging task.

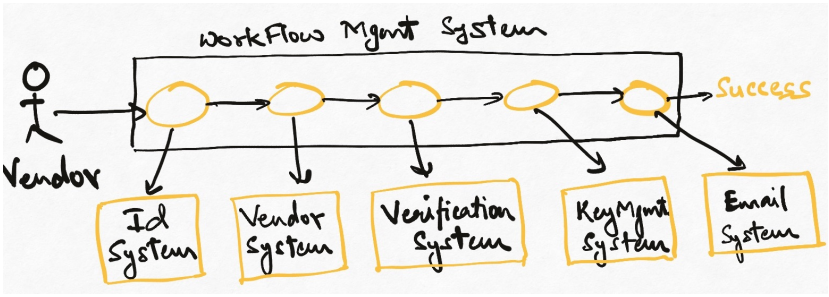
In large scale systems, these two problems are enough overhead that the primary entity soon feels like it's riddled with technical boilerplate and debt. It often becomes more and more dangerous to modify it, and shipping velocity suffers.

We need a better way to implement long-running, multi-component processes than the object-oriented paradigm offers.

Enter the **workflow** design/architecture pattern.

The workflow pattern

The workflow pattern is a powerful means of modelling business processes. We create a workflow management system/component whose primary responsibility is to model each action in the business process as a step. This series of steps constitute the entire business logic are executed one after another. A workflow can be thought of as a directed graph that invokes multiple components at each node/step to achieve the system objective.



This is different from the object-oriented model where processes are abstracted as object behaviours and their specifics typically hidden behind object APIs. The workflow pattern explicitly turns this inside out by talking about the steps of the process as first-class constructs. While workflows are graphs first and logic later, in the object-oriented style, we typically encapsulate the graph inside objects and methods as an implementation detail.

Using the workflow pattern means moving all orchestration responsibilities out of the core system components and into a separate component which only deals with the definition and execution of workflows. Executing all the steps, handling errors,

retries and breaches of SLAs are primary system objectives of the workflow management component. The core components expose APIs offer only some core capabilities and have no/minimal context of when and from where they are being called. Most of the “business” logic goes into the workflow component, which can now be used to compose the capabilities offered by different core components in whatever manner the business demands.

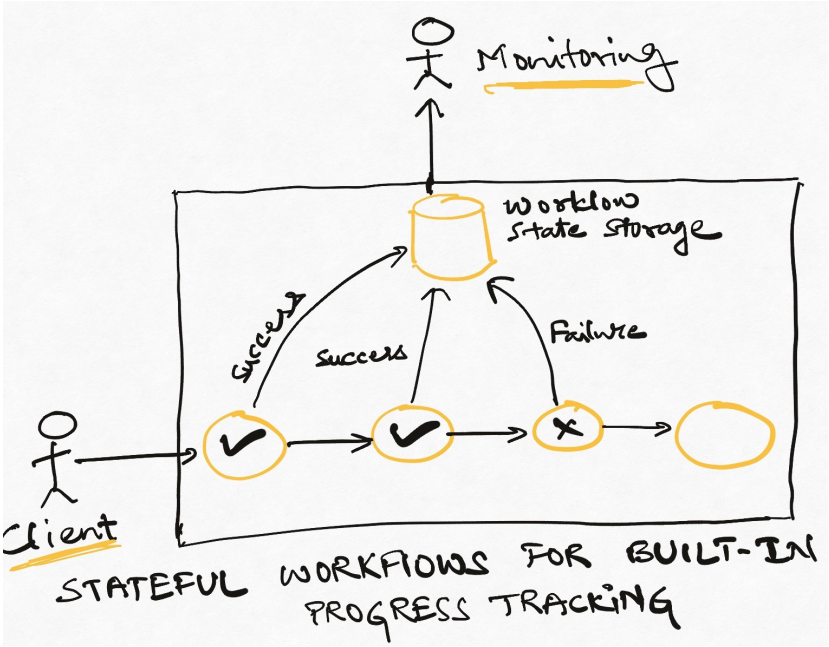
This is especially useful in distributed architectures because orchestration is a critical part of the system and no single component is often equipped to deal with this heavyweight activity without getting too deeply coupled with the other components. Extracting all of the orchestration (persistence, state, error handling etc) out of all the components which are getting orchestrated results in a nice layered, “smart pipes” architecture.

This style is sometimes called by some other names like Saga pattern (in the context of transaction handling across multiple systems).

Stateful Orchestration

Most uses of a workflow based design are in cases where we require the workflows to be long lived and have heavy synchronization overhead. In such scenarios, it usually (though not necessarily – remember that a workflow is an abstraction layer – the implementation details are relatively less

important) makes sense to make our workflow management stateful. This means that the workflow implementation itself remembers how far it had run, whether it succeeded or failed and whether things can be retried in case of failure.

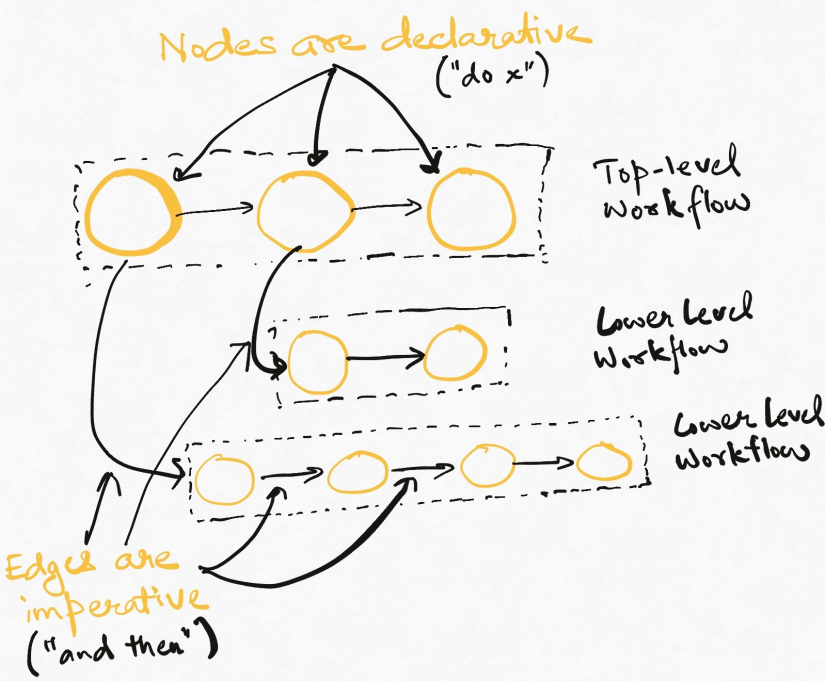


By putting all of the workflow tracking information inside the workflow management system, we are making it heavier and more capable while making the core components simpler. To know whether we generated a new vendor's API keys or not after they were successfully verified we ask neither the verification system nor the API management system, we ask the workflow management system about the state of the vendor onboarding workflow.

Another benefit of having stateful workflows is that it is easy to tell how far each business process has run so far. In a way, status updates come for free!

Mixing declarative and imperative to create layered abstractions

Workflows allow us to compose declarative and imperative styles of programming in a layered manner. Seen as a sequence of steps, a workflow is an imperative expression of how a problem should be solved. Seen as a single unit, each step in the workflow is a declarative expression – “Do X”. However, doing X can be a whole workflow by itself. So a workflow-driven system weaves a combination of declarative and imperative styles into a layered architecture.



Layers are good. It reduces the cognitive effort required to understand any component. If a component seems to be doing too many things, then this is often a good indicator that we are missing some kind of layering opportunity that will

separate core functionality from the use of that functionality (https://kislayverma.com/platform-thinking/platforms-and-dogfood-everywhere/).

This is, of course, general observation and can be applied at any level of code. If we are working inside a single component that has multiple smaller modules inside it, we can apply this pattern to compose these components (https://kislayverma.com/platform-thinking/platform-nuts-bolts-flexible-decision-making-with-rule-engines/) effectively. At an architectural scale, we can use this same construct to build layered domains and architectures (https://kislayverma.com/software-architecture/layering-domains-and-microservices-using-api-gateways/) which can give huge productivity gains if done right. Workflow-based design, with its mix of declarative and imperative paradigms, offers us a way to do exactly this.

Each layer of the emergent architecture provides higher-order functionality which is composed of the capabilities of the layers underneath. A workflow allows end-users to be hidden from the numerous steps involved in doing a logical business activity, and hence serves as an abstraction layer above the core components. As a result, what we get is not just technically layered architecture, but also layering from a business and organization perspective. Business processes can also build on top of each other – this can be a powerful tool in designing an effective organization.

Workflows as Products

As I said above, workflows are ad-hoc compositions of lower-order capabilities into higher-order ones. What drives the choices of what capabilities to compose and in what order? Obviously the requirements of the business. While lower-order capabilities like sending mail, creating order may be relatively generic, they can be composed in ways which are specific to use-cases or specific business scenarios.

If we consider the superset of all the business requirements as an aggregate, what we get is a Product. A product is a specific set of choices (https://kislayverma.com/platform-thinking/products-are-not-platforms/) made to create a specific user experience. So a different way of looking at a workflow-based design is to say that each workflow represents a part or whole of a product, while the underlying core components _may_ constitute platform building blocks. In this sense, it is technically acceptable to have multiple workflows for similar things like order placement or vendor onboarding since each would represent a specific business process or use-case. From an organizational perspective, however, having different processes to do similar things might constitute wasted effort and overhead. The ability to easily identify the major workflows in the product helps us improve organizational processes.

In the early days of any system, we often see a lot of changes happening because the end-user experience changes often and this seems to affect the entire stack over and over. The core methods and APIs of the system seem to be getting modified again and again with if-else conditions. While this flux is not ideal, it is actually pointing towards the fact that our system seems to have fewer layers than required to address the numerous use cases of the business. A good way to deal with this problem is to identify each of these use cases as a workflow, and then identify the common pieces one or more of them need, and so on. Each of these levels of recursive is a logical layer in our system's design. Of course, if we can do this before we implement the system, then it will save us all a lot of refactoring. However, I often find it technically acceptable to ship quickly so that we cover a good number of real use-cases before we try to identify too many unnecessary commonalities and abstractions.

Caveat Emptor

Calling workflows a separate architectural pattern is only a change in perspective. No matter what kind of design approach we take, we will always have to do all the things that make up the business process. Focussing on the process as a first-class concept puts the definition and management of the steps in the spotlight instead of brushing it inside an API boundary and considering it an implementation detail. Like most things in software

architecture, this is a trade-off. If the workflows are many and complicated, the workflow pattern can bring them out in the open and make them easy to understand and manage, while keeping core components relatively simple. On the other hand, if the workflows are short/simple/few, using workflow can introduce unnecessary overhead and make the code difficult to understand.

Adopting a full-blown workflow management system like Camunda (https://camunda.com/) is overkill if you don't have long or very complicated workflows. In my experience, the best uses of workflow style emerge as an artefact of refactoring done on object-oriented systems to reduce the accumulating "tech-debt" related to orchestration. As I have mentioned earlier on in this article, it is perfectly acceptable to start out by modelling a workflow as the behaviour of a primary entity.

A lesser concern is that the workflow management system now becomes the hub of all action and we have to careful in making sure it is fault-tolerant, scalable, and resilient to problems in the downstream components. I say this is a lesser concern because these are well-understood problems and careful analysis of the design and implementation choices can help us keep them at bay.

Read Next: Combining rule engines and workflows in platform architectures (https://kislayverma.com/platform-

thinking/platform-nuts-bolts-flexible-decision-making-with-rule-engines/)

If you liked this, subscribe to my weekly newsletter It Depends (https://kislayverma.com/newsletter-archive/) to read about software engineering and technical leadership

Email Subscribe

Post Views: 9,718

Software Architecture (https://kislayverma.com/category/software-architecture/)

Architecture Pattern (https://kislayverma.com/tag/architecture-pattern/), Distributed Systems (https://kislayverma.com/tag/distributed-systems/), System design (https://kislayverma.com/tag/system-design/), Workflows (https://kislayverma.com/tag/workflows/)