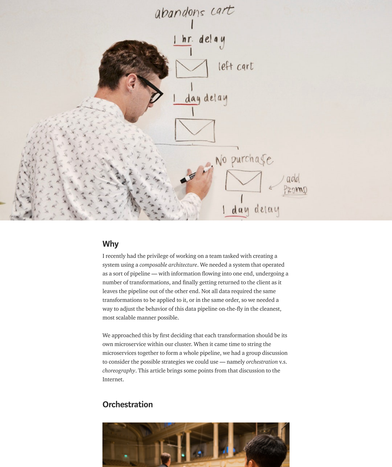
Microservice Coordination at a High Level

Jacob See Follow
Dec 5, 2018 · 5 min read

December 4, 2018

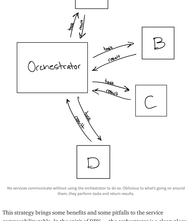


Why

I recently had the privilege of working on a team tasked with creating a system using a *composable architecture*. We needed a system that operated as a sort of pipeline — with information flowing into one end, undergoing a number of transformations, and finally getting returned to the client as it leaves the pipeline out of the other end. Not all data required the same transformations to be applied to it, or in the same order, so we needed a way to adjust the behavior of this data pipeline on-the-fly in the cleanest, most scalable manner possible.

We approached this by first deciding that each transformation should be its own microservice within our cluster. When it came time to string the microservices together to form a whole pipeline, we had a group discussion to consider the possible strategies we could use — namely *orchestration* v.s. *choreography*. This article brings some points from that discussion to the Internet.

Orchestration



When you go to a concert house to listen to your local symphony orchestra, arguably the most important role played in the performance is that of the *conductor*. Each member of the orchestra acts in a synchronized manner because they're all receiving instructions from the same point of reference. Not only does the conductor synchronize the sounds of all the musicians while you're listening, the conductor shaped the music itself in rehearsals *before you ever arrived*. Adjusting timings here and volumes there, what you are hearing is tuned by the conductor to bring you *their interpretation* of the best possible performance.

Orchestration in the context of microservices is very similar! Just as a violinist wouldn't look to the woodwinds to learn what they should be doing, one service does not need to have any knowledge of the others to play its part in the grand scheme of the application. All microservices pay attention to one system (the *orchestrator* — generally another service on the same cluster) that is charged with determining what needs to happen, and directing the microservices under its control to perform whatever tasks are necessary in order to reach its desired outcome. The orchestrator is the centralized authority for everything that happens within its scope of control. Other services can simply be thought of as its workers.

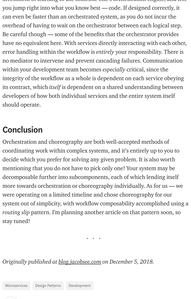
No services communicate without using the orchestrator to do so. Oblivious to what's going on around them, they perform tasks and return results.

This strategy brings some benefits and some pitfalls to the service composability table. In the spirit of [SEE] — the orchestrator is a clean place to put workflow logic. Since the orchestrator has the high-level understanding of what should be happening within the system, it can provide advanced progress monitoring and error handling right out of the box. It's not all good news for the orchestration pattern though…
Orchestration inherently sets up a structure with a single point of failure. If the orchestrator becomes unable to delegate tasks for any reason — it's as if the conductor of the symphony orchestra dropped dead in the middle of a performance. All progress within the system stops, and unless the fault is of a type which can be automatically detected and healed within your environment, your on-call engineer is not going to have a good night.

For further reading on orchestration in practice, or to try it out yourself, you may want to check out Uber's Cadence or Netflix's Conductor orchestration engines. Conductor, the system I'm more familiar with, defines workflows using a JSON based DSL that looks like this:

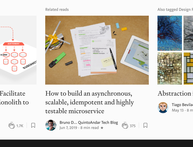
```
{
  "name": "encode_and_deploy",
  "description": "Encodes a file and deploys to CDN",
  "version": 1,
  "tasks": [
    {
      "name": "encode",
      "taskReferenceName": "encode",
      "type": "SIMPLE",
      "inputParameters": {
        "fileLocation": "${workflow.input.fileLocation}"
      }
    },
    {
      "name": "deploy",
      "taskReferenceName": "d1",
      "type": "SIMPLE",
      "inputParameters": {
        "fileLocation": "${encode.output.encodedLocation}"
      }
    }
  ],
  "outputParameters": {
    "cdn_url": "${d1.output.location}"
  },
  "schemaVersion": 2
}
```

Choreography



To continue with the performing arts analogies, and because the names of these strategies align so well with these examples, imagine you're attending a ballet performed by skilled dancers. There is no conductor leading them — dancing is a visual art and that would block the audience's view! The dancers are synchronized and move fluidly around each other because they've gone through rigorous training to be able to do exactly that. The absence of an orchestrator during their performance is supplanted by the fact that they each know what should be happening and when, and maintain an awareness of their surroundings and other dancers to make sure that the performance goes as planned.

This is also a valid strategy in microservice coordination. Choreography is a design pattern in which there is no orchestrator present. Each service must know not only how to do its job, but also when and how to interact with other services in the same system. All components are autonomous, and typically agree to adhere to a contract, but otherwise are fully self-managed and do not receive direction from an external source.



No orchestrator is present. Services do not receive instructions to perform tasks, rather, they understand the result they are given, and know what to do with it next.

Compared to orchestration, choreography is (in my opinion) quicker to get started with for a new project. It does not require the upfront time investment of learning to use someone else's orchestration engine, and lets you jump right into what you know best — code. If designed correctly, it can even be faster than an orchestrated system, as you do not incur the overhead of having to wait on the orchestrator between each logical step. Be careful though — some of the benefits that the orchestrator provides have no equivalent here. With services directly interacting with each other, error handling within the workflow is *entirely your responsibility*. There is no mediator to intervene and prevent cascading failures. Communication within your development team becomes *especially* critical, since the integrity of the workflow as a whole is dependent on each service obeying its contract, which *itself* is dependent on a shared understanding between developers of how both individual services and the entire system itself should operate.

Conclusion

Orchestration and choreography are both well-accepted methods of coordinating work within complex systems, and it's entirely up to you to decide which you prefer for solving any given problem. It is also worth mentioning that you do not have to pick only one! Your system may be decomposable further into subcomponents, each of which lending itself more towards orchestration or choreography individually. As far as — we were operating on a limited timeline and chose choreography for our system out of simplicity, with workflow composability accomplished using a *routing slip* pattern. I'm planning another article on that pattern soon, so stay tuned!

. . .

Originally published at blog.jacobsee.com on December 5, 2018.

Microservices Design Patterns Development

WRITTEN BY
Jacob See
Follow
Consulting Engineer at Red Hat, Passionate about solving problems in simple, elegant ways through the use of innovative and emerging technologies.