**1) Cohesion Chaos**

We developed a service to get the customer information designed to pull the customer policy information, personal information and the plan that they enrolled in. Over a time, it started to do more than getting the customer information. As new requirements came in, this service went through frequent changes and deployments. It was unable to scale and meet the required availability. It became the proverbial “Big ball of mud”.  How did it get there? For starters, there was no governance around functional separation of concern. If an influential consumer asking to put unrelated logic in this one service to reduce round trips, that function got slapped on without question.  Perhaps a gateway or a BPM layer could have avoided this scenario, but there was no time for that…just time to crank out another business function point.

The preventative cure is to govern business functionalities that are not relevant to the service. Services must align clearly to a business capability and should not try to do something outside of their boundary. Functional separation of concern is vital for architecture to govern otherwise it will destroy the agility, performance, and scalability and ended up in establishing a tightly coupled architecture, resulting in delivery entropy and cohesion chaos.

**2) Not taking Automation Seriously**

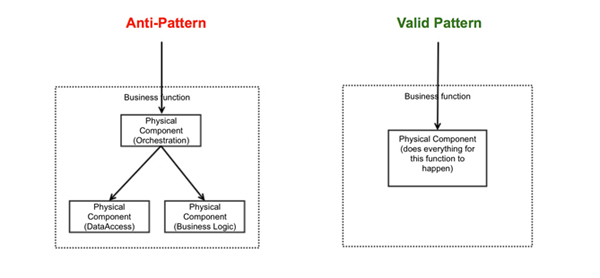
We didn't have a strategy for automated deployment and ops monitoring of services (runtime QoS metrics). It obviously increased operational expenses and manual errors during deployment. Several times production deployments caused outages due to configuration errors. The services were always deployed in HA mode and so the number of containers was 3x to the total number of services. The operations team was unable to handle the configuration for each service manually. After a certain time, ops started to complain that the architecture was inefficient as they were not able to handle the increased number of containers.

What is the vaccine for this? The recipe has multiple ingredients.  Continuous deployment, if you have not done so, is a must investment and a cultural change that every enterprise should aim for. At least, if you don't have a way to automatically test and deploy – do not do micro-services. Microservices are aiming to drive agility, with the speed we need to change; quality assurance involves each service having automated unit, functional, security and performance testing. Service Virtualization is another powerful concept when we develop services that are integrated with services outside of our control.

**3) Layered Services Architecture**

One common mistake people made with SOA were misunderstanding how to achieve the re-usability of services. Teams mostly focused on technical cohesion rather than functional regarding reusability. For example, several services functioned as a data access layer (ORM) to expose tables as services; they thought it would be highly reusable. This created an artificial physical layer managed by a horizontal team, which caused delivery dependency. Any service created should be highly autonomous – meaning independent of each other.

Creating multiple, technical, physical layers of services would only cause delivery complexity and runtime inefficiency. We ended up in having wrapper services, orchestration services, business services and data services. These service models served technical concerns. Individual teams formed to manage these layers and ended up having business logic sprawl, no single owner for a capability, lost the efficiency and there was always a blaming game.



Logical separation of layers within a service is fine, however, there should not be any out of process calls. Try to look at a service as one atomic business entity, which must implement everything to achieve the desired business functionality. The self-contained services are more autonomous and scalable than the layered services. It's perfect to re-write some common code across multiple services, that's fine and it's a good trade-off to keep the autonomy level. The bottom line is that don't have services separated by technical concerns instead they must be separated based on the business capability. The concept of containerization is thriving because of this character.

**4) Relying on Consumer Sign-off**

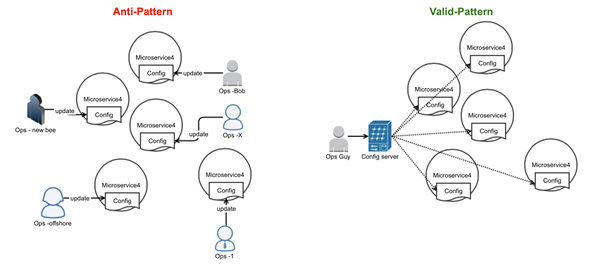
We had a service consumed by multiple applications from three different channels i.e. agent, the web, and voice. Agent channel was our primary, so the services had to wait to get their sign-off before they can go into production. It delayed the voice and web application production releases. What bound those three channels together so tightly?

The service was not a loosely coupled when it came to channel specific functionality. Give independence to your services. Every service that you deliver must have a test suite, which should cover all the service functionality, security, performance, error handling, and consumption driven testing for every current and future consumer. This must be included as part of the build pipeline for automated regression testing.

**5) Manual Configurations Management:**

As we started to do a larger number of services (and the inevitable sprawl due to lack of service lifecycle governance manifested itself) managing the configurations for each service went out of control. Most of our production deployment was not smooth because of configuration failures like the bad password, wrong URL, incorrect values. It became harder and harder to manage these manually. If we had only used application configuration management tools as part of a PaaS or CD…but we didn’t.

(Click on the image to enlarge it)

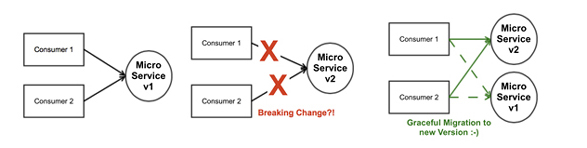
[](https://res.infoq.com/articles/seven-uservices-antipatterns/en/resources/fig21_large.jpg)

**6) Versioning Avoidance:**

Naively, we thought it would be only need one version of the service. Then we started to add major, minor versions to accommodate multiple consumers and frequent changes. Eventually, every release had to be a major release since the services were relying on consumer sign off. As a result, the number of containers increased very fast and it became a huge pain to manage them. Lack of runtime governance was another aspect that contributed to this issue. Some enterprises foolishly try to avoid versioning. Services need to be architected assuming that change is inevitable.  Have a strategy to manage the forward compatible service changes and allow your consumers to upgrade gracefully. Otherwise, it will lead to having consumers tightly bound to a service version and break when there is a change.

The complexity grows as the number of services grows which the microservices world expects. Have a versioning strategy that can allow the consumers a graceful migration and assure providers can transparently deploy changes without affecting anyone. Limit the number of side-by-side major versions in the production and govern them.

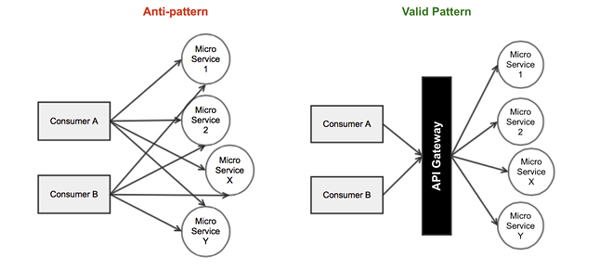
(Click on the image to enlarge it)

[](https://res.infoq.com/articles/seven-uservices-antipatterns/en/resources/fig31_large.jpg)

**7) Building a gateway in every service**

We didn't have an API gateway and we didn't have runtime governance (we didn’t know who was consuming what and at what rate at what time). We started to implement end-user authentication, throttle, orchestrate, transform, and route etc. in each service. It added complexity to each service and we lost consistency of implementation from service to service, so we had no idea who implemented what and where. On top of it, some of our services were built to satisfy one consumer non-functional requirements, but not another’s. If we had a gateway, applying some data filtering and enrichment patterns could have done it. If only.

Invest in API Management solutions to centralize, manage and monitor some of the non-functional concerns and which would also eliminate the burden of consumer's managing several microservices configurations. API gateway can be used orchestrate the cross-functional microservices that may reduce round trips for web applications.



**Conclusion**

The goal of microservices is to solve the three most common problems i.e. improve customer experience, highly agile to the new requirements and drive down cost by delivering the business functions as fine grained services. This is not a silver bullet and requires a disciplined platform, where delivering the services in an agile fashion with high quality is possible. Learn from other’s mistakes (mine) and avoid the above listed patterns in the architecture and delivery process. This is the first baby step before we can even talk about containerization, cloud adoption etc. I hope this article gives you something to think about for your enterprise and work towards resolving these anti-patterns before you weave them into your architectures. Most of the items will drive cultural changes within the organization and cannot be done just by yourself, ensure partnership with your executive and senior leaders.