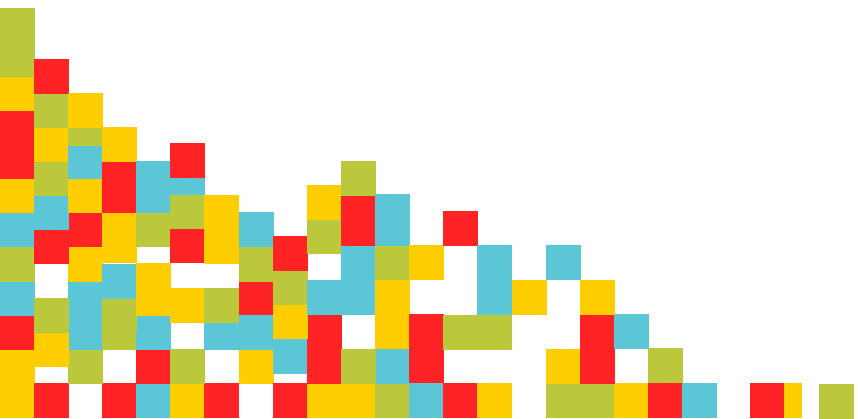
6 Pitfalls of Microservices:

How to Mitigate Them in your Applications



AppNeta®
Performance in Production

A microservices architecture is a style which uses small, independent services which communicate across well-defined APIs as the basic building blocks. The idea of service-oriented architectures isn't new, but combined with the popularity of DevOps in larger teams, the adoption of public IaaS providers like Amazon, and tooling that enables Continuous Integration / Delivery, the idea of re-tooling existing applications with microservices has clear advantages:

- Small services are easier to debug, because they have fewer use cases and a team that understands the service.
- Small services are easier to scale, as they can typically be deployed as an expandable set of nodes behind a single load balancer.
- Each service can use the right technologies for its use case, including different languages.

While compelling, these benefits aren't free. Creating a system where all of these services can be robustly designed, developed, tested, deployed, and monitored isn't trivial. When designing a system using microservices, implementing human and software processes which addresses the weaknesses of this approach will make the entire system easier to maintain and easier to add features to.

To understand these processes, let's look at top 6 challenges associated with microservices architectures, and best practices to address them.

Debugging Though APIs

While single services are developed in isolation, production requests will typically touch many different services owned by different teams. This means that requests will interact with a variety of APIs, and each of those APIs is owned by a different team. When trying to debug specific requests, the APIs add an additional level of complexity. Is the problem the team that's using the API, or is it internal to the service?

To manage this problem, monitor applications as they're used in production, and test them continuously there. Testing services for expected behavior from the outside validates that the service performs correctly for valid inputs, and monitoring live requests ensures that there aren't malformed requests to the service. In most cases, invalid requests will have a different performance profile, as well: either quickly returning an error or trying to return 100x the normal amount of data. Most APIs specify uptime requirements, even in the face of bad inputs, so tracking these requests individually is crucial.

Pro Tip: Some problems may span multiple services, influencing calls that are 2 or 3 layers removed. Make sure you can track requests through many API calls.

Testing Overhead

Testing independent microservices gets easier as they get smaller, but integration testing gets more complicated. QA typically forms part of a microservice team (sometimes owning several services), but integration testing isn't a part of any single service team's charter. Since bugs caught in production are typically 10x more expensive than bugs caught before deployment, assuming everything will work based on API contracts can be an expensive mistake.

To avoid the cost of integration bugs in production, integrate and test all services in a single environment before deploying anything. Many teams will maintain backwards compatibility on APIs, to help with phased upgrades across different services, and a full-scale test will make sure that necessary versions of everybody's APIs are working as expected. This integration test is typically owned by either operations (as they'll be first to notice if it fails) or a dedicated QA team. This is an additional cost on top of per-service testing, but as the number of services grows, the saved downtime and poor end user experience is well worth the additional step.

Pro Tip: Make sure your integration QA team is using the same tools as service teams. Fixing bugs is faster if everybody is speaking the same language.

Operations Overload

Operations teams are already familiar with the pain of fixing problems they didn't create, and growing the number of applications under their control from 2 to 200 doesn't help. Many ops teams have figured out what backchannels to use to get devs in the loop for the most critical applications, or how to get the product owner to prioritize a certain bug. Commonly, these practices don't scale to tens or hundreds of services, overwhelming either the ops teams or single developers.

To help with this human scaling problem, create defined contracts between development and operations teams to set expectations on triage, prioritization, SLAs, and level of detail in bug reports. Without this, many ops teams will find themselves overloaded, either hiring more people to throw at the problem or allowing the user experience to degrade while they fight the most visible fires. Neither is an effective long-term solution, and many times will lead the entire organization to feel like a microservices approach "simply doesn't work." Focus on scaling effective communication channels while eliminating areas that are one-off.

Pro Tip: Specifics make everything better. Include performance trends, error rates, and individual anomalous requests in every triage report.

Long Tail Performance

Any system will have a few requests where the response time is abnormally long, but in a request that touches many different services, the change of hitting one of those "long time" events is much higher. For example, if a system touches just 6 different services, 1 in 4 requests will be impacted by a service call that's in the bottom 5% of performance. This could mean end-user performance impact of 50-100% (multiple seconds, for many sites) which can sharply drive up abandonment rates or even cause lost revenue.

Instead of measuring average or even 90 percentile, microservices performance means measuring the 99th percentile or higher. The volume of requests to each microservice tends to be at least comparable with the volume of the site itself and in some cases an order of magnitude more (i.e. low-level data services that are aggregated before being presented to the user). Depending on the site traffic, measuring to this level might mean looking at only a handful of requests per hour. To really understand the long tail, it's important to both trend this data and to check abnormal requests individually.

Pro Tip: Building intuition for the long tail is an underrated skill. Periodically looking at strange requests can give ops a gut feel for where to look first during an emergency.

Technology Divergence

In theory, the flexibility to use the right tool for each service is great. In practice, different frameworks, coding styles, and even languages can quickly turn into an unmaintainable mess. Even with simply different styles developers will find it difficult to understand services outside of their own. If different services are written in different languages the entire organization risks losing services if even a single developer leaves.

To prevent this, pick a standard set of technologies and only diverge from that where necessary. Sometimes it makes sense to write an API router in node.js, or an admin interface in Ruby, but having a default simplifies decision-making and makes it easier for developers to move between project quickly. Beyond languages and libraries, standardizing on other tooling has a similar effect. Using the same APM solution, logging store, and infrastructure tooling reduces the technical complexity of the entire team, allowing any engineer to pitch in and come up to speed quicker in any situation.

Pro Tip: Pick tools that can be used by multiple teams. Fewer tools overall means new engineers have less to pick up, and their knowledge is transferable across teams.

Cross Cutting Changes

Arguably the biggest weakness of loosely coupled architectures is trying to deal with changes that are tightly coupled across the entire application. Microservices reduce the number of these changes drastically, and strategies like implementing backwards compatibility and independent deployments can mitigate some of the impact. Unfortunately, there are certain changes that must be deployed across everything in the application at the same time, frequently for business reasons. Product releases tied to real-world events (like a conference announcement) can mean many services changing at once, and security updates are rarely the concern of a single service.

The only way to deal with this is to build a release and monitoring system that gives deployment teams the information they need to notice when something has failed and roll it back seamlessly. Development, IT operations and DevOps teams are all ultimately trying to serve the customer--that means paying attention to the end-user experience. Many cross-cut changes can and should be staged and tested at the integration level, though (as discussed above), that comes with its own set of challenges. Ultimately, there's no substitute for a development cycle that involves small, incremental updates to production and extensive full-stack monitoring for regressions in releases.

Pro Tip: Make sure all teams are using monitoring tools that include end user experience monitoring. If there are resource issues deep in the application, the first question should be 'How many users are affected?

Conclusion

Like any architectural choice, microservices aren't a silver bullet. They have their own tradeoffs to consider, and many of the weaknesses can be mitigated if carefully considered ahead of time. Think through the specific needs of your team, your product, and your users.