Point to Point Responses to Comments

January 30, 2023

Comment	Response
The authors identified a list of microservice patterns to research, and validated those with interviews. According to the discussion chapter, there are 3 major types of feedback that is relevant IMHO - it just does not come across that way yet: 1. Type 1 -patterns that are already recognised as being relevant and actually used by the industry experts (like API gateway and anti-corruption layer) 2. Type 2 -patterns that the experts might be interested in checking 3. Type 3 -patterns that the experts actually recommend against (like CQRS) Recommendation: it would be nice to see a table that summarises this information as part of this article's conclusion or discussion. Recommendation: for future work, types 2 and 3 are probably most interesting to focus on, besides confirming the applications of type 1.For instance, checking the applicability of type 2 patterns, and/or challenging the inapplicability of type 3 patterns by proving the critics wrong?	This is a valuable feedback. We've added the summary table in the discussion section and elaborated a bit about the nature of the feedbacks in regards to various patterns. We've also added a sentence at the end of the discussion section about future work.

Microservices (and patterns) are one type of architecture, mostly driven by the need for independent teams to deploy independently (either in release cadence or technologies employed). Microservices make this more flexible at the expense of some overhead (both coordination and network / performance as mentioned in the text). There are other patterns that may be interesting, but not necessarily "services" deployed independently but rather "intra-service" modules that do, for instance, input data validation. It would be interesting to see future work also incorporate that, since microservices are not necessarily the best architectural style. Many people and teams actually go against it, and Fowler himself said that many successful microservice projects started off as monoliths - refactored along the way. So I would suggest "architectural patterns" as a broader scope for future research work. On the side: refactoring is (in my experience) the best way to apply patterns since this ensure that patterns are only used to simplify existing code - rather than being a holy grail from the outset. I have had many worries over team members that learned a pattern in some training and started using it all over the code base the weeks after (alas, also in cases where it made things worse).	As we digged deeper into this study, we stumbled upon many cloud-based and intra-module patterns (inversion of control for instance) that could be applied to our work. Majority of our interviews went over time, because we had some interesting software engineering conversations going on in regards to patterns. While a lot of other patterns and ideas could be incorporated for this work, we had to define a boundary. This paper has been the result of six months of work, and has grown from 20 pages initially to 48 pages at the end. While we appreciate this feedback, we think keeping this study focused only on microservice patterns and big data suffices for this paper.
Some typos identified (I probably missed some): -page 17: "gonna" -¿ should be "going to" -p11: "to sketch" -¿ should be "to a sketch" -p1: "descriebd" -¿ should be "described"	All typos are now fixed. Thanks

A (CQRS style) event store could be an ideal way to refactor BD architectures afterward, since one can re-init a new system based on the historical "replay" of all past events.	Indeed. In fact, event sourcing and CQRS are usually going together. While this can be initially very useful, as time goes one, event sourcing itself can introduce myriad of challenges. We have been maintaining a large scale Kafka cluster, and we have faced several challenges after adopting event sourcing and CQRS. These challenges are complexity, performance (storing and retrieving events can be expensive, in Kafka things are actually stored on the disk), debugging and troubleshooting, data consistency, storage (as time goes, even more challenging), and event versioning (managing the evolution of our schema has been a challenge). Nevertheless, we do not posit that event sourcing is not an ideal way to refactor BD architectures. We think this pattern can be beneficial in specific scenarios.
Competing consumers don't really need a circuit breaker in my understanding?	The competing consumers pattern is a design pattern used to handle large volumes of messages in a message-driven architecture. The circuit breaker pattern, on the other hand, is a design pattern used to prevent failures from cascading and bringing down the entire system. By using these two patterns together, you can build a robust and scalable system that can handle large amounts of traffic and recover from failures gracefully. The competing consumers pattern provides the processing power to handle the traffic, while the circuit breaker acts as a guard against failures, preventing failures from cascading and bringing down the entire system. For example, if a service that is being consumed by multiple consumers starts to experience failures, the circuit breaker will trip, causing subsequent requests to fail fast. This helps to prevent further failures from cascading and bringing down the entire system. At the same time, the competing consumers pattern ensures that there are multiple consumers processing messages from the shared message queue, allowing for increased processing power and scalability. In summary, the combination of the competing consumers pattern and the circuit breaker pattern can help ensure high availability and performance for your application, allowing it to handle large amounts of traffic and recover from failures gracefully.

I strongly recommend the authors look at the "event storming" approach to identify and structure DDD / CQRS / microservice systems.	Thanks for the suggestion. We are aware of event storming, and had several event storming workshops in our company. Event storming can be particularly useful in the context of software development, as it helps to provide a shared understanding of the business domain and to identify potential pain points that may need to be addressed in the design and development of software solutions. We have even used some of the workshops to identiy the perimeters of our organizational domains. Event storming, however, is a technique and not a design pattern. As a result, it falls outside the scope of our work.