About Bounded Contexts and Microservices

Alberto Brandolini•Focus On•Jun 11, 2020

If you've been anywhere around enterprise software architecture these years, chances are high that you've run into questions like *"what is the right granularity of Microservices?"* or *"Are Microservices and Bounded Contexts the same thing?"*

In the next paragraphs, I'll try to clarify as much as I can.

Definitions first

The first issue with my clarification attempt is that both concepts have a somewhat *fuzzy* definition. It's already hard to define what the concepts are by themselves, so combining the two easily drives the conversation into ambiguity.

Bounded Contexts are built around the language

Eric Evans, the inventor of Domain-Driven Design is a person that can spend days looking for the perfect word to convey a specific meaning, so when he picks one ...he probably meant it. So let's look to the original definition of **Context**, and then of **Bounded Context** too (the source is here):

Context*: The setting in which a word or statement appears that determines its meaning*

Which is basically a *linguistic* term. I have endless stories and jokes about the different meanings of the word *coffee* around the world, but this is just the nature of any language. Terms progressively get used outside their original scope, a metaphor that initially sounded weird becomes part of conversational language, and so on.

Languages are not fixed, they *evolve.*

Languages are not flat, and contexts are necessary to achieve precisions.

But linguistic can only help us to a given extent: while writing software we need *precision*. The very same ambiguity that made some conversation sound like bad jokes, will be a dangerous source of bugs when developing software where multiple contexts are involved.

What is a Bounded Context then?

In the software world, we need to take the responsibility of setting boundaries for this language consistency. And this is where the **Bounded Context** concept appears.

***Bounded Context**: The delimited applicability of a particular model. BOUNDING CONTEXTS gives team members a clear and shared understanding of what has to be consistent and what can develop independently*

Here the definition is closer to the software development world. There are a few implicit assumptions, let's try to make them explicit.

- We are talking about **multiple specialized models**: non-trivial software will need different models and different styles to cooperate. Every problem will have a model that would be the *best fit* for that specific problem.

- There is a strong link between a **team** and a **bounded context**: a team with a well-defined purpose will be very likely to develop a precise jargon around their specific problem space.
- **One-size-fits-all standard architectures are not in the cards**. It would be just like saying that *"We have to win Formula 1 championship"* and at the same time stating that *"every employee, including drivers, should be paid the same."*

In practice, Bounded Contexts use the language as a *canary in a coal mine:* language is extremely sensible to variations coming from different sources (team members, stakeholders, team distribution, underlying technologies, time and so on.). Inconsistencies will manifest in the language, so we'll keep an ear on it.

Preserving the boundaries becomes an interesting design activity since it implies pushing translations at the boundaries, by implementing *adapters* or Anti-Corruption Layers, in order to keep the internal model as simple as possible, or better *focused on the only relevant concerns for this specific model*.

What are Microservices then?

When it comes to understanding exactly what a Microservice is, the situation can be even tougher since there is not a single central definition to refer to. However, Martin Fowler's Bliki can still be an authoritative source of information.

The term is strictly architectural, but interestingly the implications are spawning across the whole *Socio-technical stack.*

- **Componentization via Services** → This is the most obvious part. It's also the one that resembles SOA the most. Being *independently deployable* is a key feature.
- **Organized around Business Capabilities** → Cross-functional teams are serving a specific business purpose.
- **Products not Projects** → the team is responsible for the whole lifecycle, including production and maintenance.
- **Smart endpoints and dumb pipes** → putting the logic in the communication channel ultimately benefits the vendor (with horrible lock-in scenarios), the idea is not to repeat the mistake that happened with ESB vendors in SOA.
- **Decentralized Governance** → Local choices are more informed about local problems: so local implementation should have more freedom. (Yep sounds a lot like what we just said)
- **Decentralized Data Management** → Microservices should own their own persistence. No surprise Fowler's article directly refers to the DDD concept of Bounded Contexts.
- **Infrastructure Automation** → Continuous delivery is a must. For quality, faster feedback loops, and so on.
- **Design for Failure** → Applications need to tolerate service failures, so part of the Microservice approach involves making sure failure doesn't affect the user experience. Netflix is the most notable case, so far. And it led to interesting ideas like Simian Army and Chaos Monkey.
- **Evolutionary Design** → Parallel independent evolutions bring also the idea of smaller, possibly uncoordinated, releases. Components are independently replaceable and may have different lifespans.

Here I tried to visualize the overlapping of the key traits of the two concepts.

Language consistency	Mandatory, by definition	Implicit and recommended	No problem!
Organized around Business Capabilities	Implicit in the idea of vertical Ubiquitous Language shared between business stakeholders, developers, testers, and data engineers.	Cross-Functional Teams around a specific purpose.	Perfect Match!
Componentization via Services	Orthogonal	Key property	No problem
Products not projects	Orthogonal, recommended for deeper learning	Key feature	Why Not?

Smart endpoints and dumb pipes	Orthogonal, recommended as a strategic pattern	Key feature, we won't repeat SOA mistakes again.	Go for it!
Decentralized Governance	A model fitting its purpose.	Local choices win over enterprise standards.	Perfect Match!
Decentralized Data Management	Private persistence is fundamental for language consistency, and for safe evolution of the model	Private persistence per services	Perfect Match!
Infrastructure automation	DDD says nothing about it, but whatever is making the team feel safer and ready to experiment sound like a great idea.	It's core to Microservices and getting progressively better supported by the available tools	Go for it!

Design for Failure	Orthogonal	Key feature	No problem
Evolutionary design	Good fit with DDD approach. Bubble context can be seen as a good example.	Key feature	Perfect Match!

Feels like the two concepts are different but highly *compatible*.

Looking for the perfect size

However, they're not the same thing:

- obviously, you can have Bounded Contexts inside a different architecture: they predate Microservices by roughly a decade.
- you can have multiple bounded contexts inside the same deployment unit (like a monolith with a clean internal separation), as long as the safety recommendations are fulfilled.
- **Bounded Contexts** are built around the ***purpose*** of the model, **Microservices** around the ***deployment boundaries***. The driving forces can be similar in some scenarios, but they're not the same.

In a young startup, development speed is very important since we're building a lot of components from scratch, however physical separation of the different components can slow everything down and in a quickly growing environment, boundaries may be blurry for a while. A common strategy might be to go with a logical separation inside a monolith and start evolving the architecture only when hitting given numbers.

In a mid-sized company, different teams can be already working with well-defined departments. Purpose-driven boundaries can overlap with units of deployment, and with business capabilities too.

In a few scenarios, the value delivery can be achieved by loosely coupled components, with multiple independent modules. In such a scenario - made famous by Netflix - components are way smaller than the Bounded Context, and integrity of the overall behavior is preserved with radical approaches like Chaos Engineering (with Chaos Monkey and Simian Army being famous examples), to boost the resiliency of the whole system.

A possible rule of thumb

I like the idea of a progressively emerging architecture, instead of big design up front. A vision is fine, a death march is not.

Bounded Contexts should emerge around purpose and language boundaries. A Big Picture EventStorming is a very effective way to highlight and gather these boundaries.

Microservices are usually finding the best size for the independently deployable unit around the *team size*, and the *frequency of change*. Splitting components that end up being deployed together tends to backfire as cost of coordination, so autonomous teams tend to find their own perfect balance.

The cost of mistakes

The dynamics of a wrong size are different: too fine-grained **Bounded Contexts** can easily be mixed while separating BCs that have been mixed is way more expensive.

Microservices expose the opposite risk structure: too finely grained services will increase the costs of keeping the overall structure under control, while splitting a service into a more articulated structure, possibly preserving the front API, could be done on-demand, after analyzing the real pressure.

The combination for this opposite forces, usually calls for something like: ***keep your boundaries logically clean, enforce physical separation when it's worth the cost.***

Anti-patterns

Data-driven splitting (turning database tables into microservices without a behavioral inspection and redesign) also backfires spectacularly: the new system turns out as coupled as its monolithic ancestor, but possibly slower, and with skyrocketing refactoring costs.

Seeing beyond technology

Most practitioners are worried about the perfect size, and the technology stack. But the really interesting properties are in the *social* part of the Socio-Technical stack.

Bounded Contexts provide an isolated space for experimentation. Isolation is fundamental to provide **safety**. If your model is shared with too many parties, people would not run experiments, afraid of breaking someone else's code.

A good Bounded Context provides a safe space and a virtuous responsibility loop.

We are the only people responsible for this piece of code. If there's a bug in there, we can't blame anyone else. But we're good! So, we're going to make sure there will be no bugs at all in our code.

This little trick of not being dependent on other teams or other technology made all the difference from a lame team to a juggernaut. Well-thought Microservices seem to be going in the same direction, adding some acceleration in the feedback loop so that domain learning and business learning too are quickly validated by the impact in production.

What is really striking a chord for me is the alignment of both Bounded Contexts and Microservices with the principles of motivation coming from Daniel Pink's Drive: **autonomy**, **mastery** and **purpose**.

- Being as independent as possible from other teams constraints in terms of planning, technology and mental models. The more independency, the fewer possibilities for blame games and finger-pointing.
- Being committed to deliver high-quality software and to suffer the consequences of bad quality, is a great setting for improving the quality of software where it really matters.
- The direct link to a given business capability will simplify interaction with the business representatives, bringing shorter feedback loops, more opportunities for learning and ultimately opening the possibility of a more experimental and possibly creative conversation.

Put in another way: we keep looking on the technology and engineering side, because that's what we're good at, and we might forget that we're just working to establish a healthier workplace for software professionals.

https://blog.avanscoperta.it/2020/06/11/about-bounded-contexts-and-microservices/