The Implementation Journey of CRUD in Jina

Implementing CRUD should be easy, right? Answer, right off the bat: no.

Cristi Mtr Follow
Feb 17 · 4 min read

We all know about CRUD [1]. Every app out there does it. It is essential to interact with the data those apps present to us after all. As a user, this all feels intuitive and expected. So implementing CRUD should be easy, right?

Answer, right off the bat: no. Answer, with a bit more nuance: *depending on the system*, no. Answer, in full this time: Jina is complex, so no.

OK, I'm going to assume you have some questions now. Like: why is Jina so complex? What about that complexity makes it hard to implement CRUD? What specific challenges did you face? Did you make any compromises? This blog post is about answering these questions.

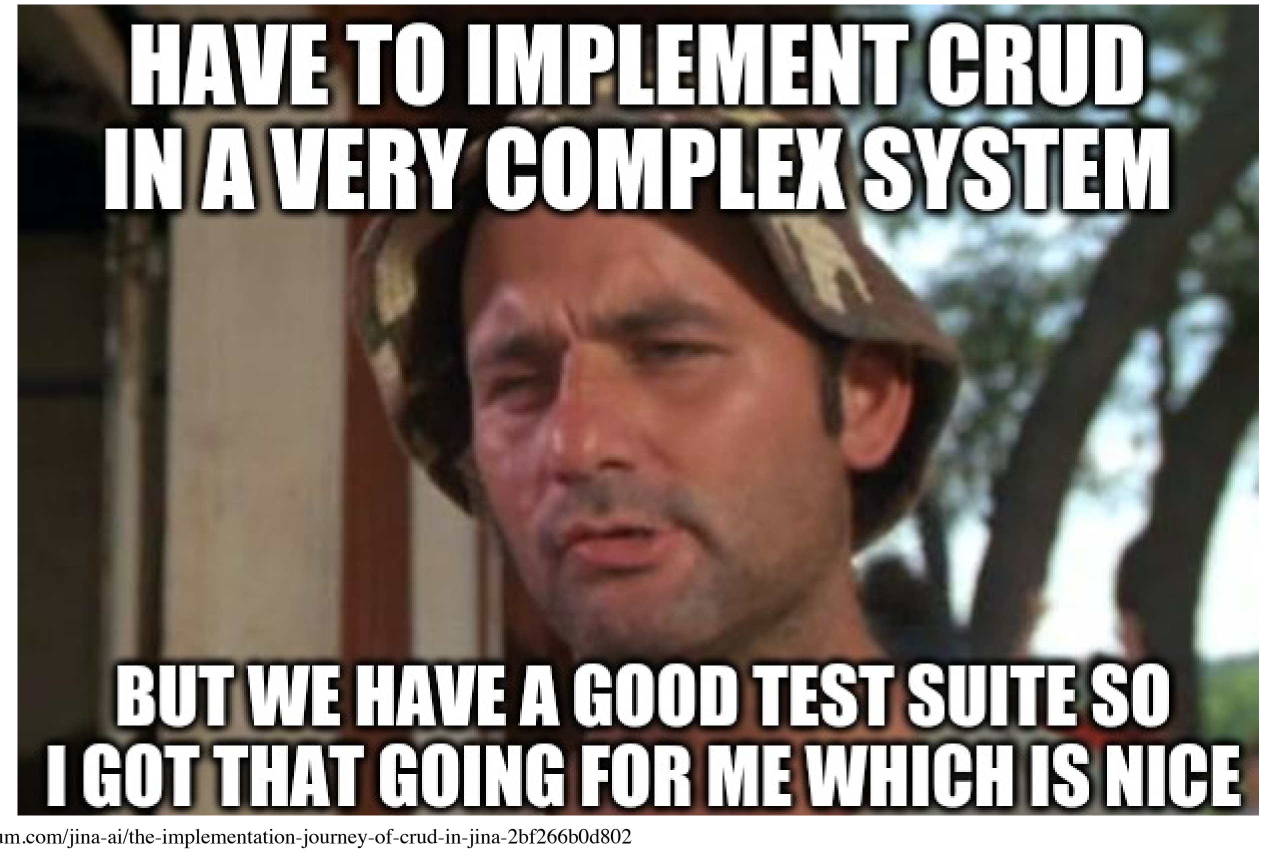
The Vision of Jina

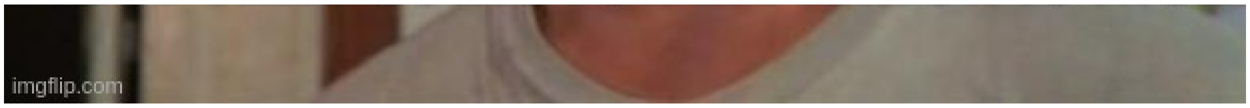
We built Jina from the ground up with the vision of being _the _design pattern for neural search. As part of this, we prioritized adding and searching data. We made these operations our top priority in terms of ease of use and performance. These are the core operations in Jina.

The unidirectional Flow

However, like any design decision in software development, everything is a compromise. At the time, we were focusing on building up Jina and were not considering the complications arising from specific future features.

This is what happened in November 2020, when we had started to focus on building the Update and Delete operations. Piece by piece, we were tracking down what components in our system would be problematic for this feature. We realized that it would be quite a few. Good thing we had tests.


HAVE TO IMPLEMENT CRUD
IN A VERY COMPLEX SYSTEM
BUT WE HAVE A GOOD TEST SUITE SO
I GOT THAT GOING FOR ME WHICH IS NICE



One of the key elements that made the process hard was our unidirectional Flow. In order to be as efficient and scalable as possible, the Flow in Jina only processes your requests in one direction. This means that the system can be easily scaled horizontally, by adding machines with replicas and parallelization. It also keeps the design clean, without the danger of circular calls between Pods.

However, this proves difficult in the context of Update and Delete. When you update a Document that has child Documents (as result of segmentation), you will need to first delete the old chunks yourself. Then you can send the new Document, and the new chunks will be created from that. If that Flow had been bidirectional, it would have been able to send a request back through its Pods to delete the specific IDs of the chunks. However, this would have led to possible endless loop problems.

Indexers and `mmap`

Another key element in Jina is our Indexers. We have written the code to be very efficient for indexing and querying. We wrote the classes to actually use `mmap` to map it to a file on the disk. We chose this in order to reduce RAM usage. However, this performance did come at the cost of flexibility. When adapting these classes to support Update and Delete, we noticed that it would be very inefficient to do an actual Update and Delete in the data structure itself. That would mean loading it all into RAM, editing it, and then writing it back… which would ruin the entire point of the initial performance design.

Thus we had to design a solution that would *mask* the old entries in order for them to be considered deleted. For updates, we would then insert the same ID with the new updated contents. This is not a perfect solution. Users need to rebuild their indices regularly, since they would be growing in size due to the old data still being stored (just unreachable).

Conclusion

TL;DR: Important to bear in mind that Jina is **not** a database engine. While we support some operations that databases usually exhibit, our main focus is on making Jina performant, reliable, and scalable for **search**. Traditional database engines are optimized for CRUD specifically. It's a matter of design goals.

Crud Neural Search Machine Learning Software Design Performance