

# 11 Myths About In-Memory Data Grids

In-memory data grids (IMDGs) can be quite handy for embedded developers, but a number of misconceptions swirl about the technology.

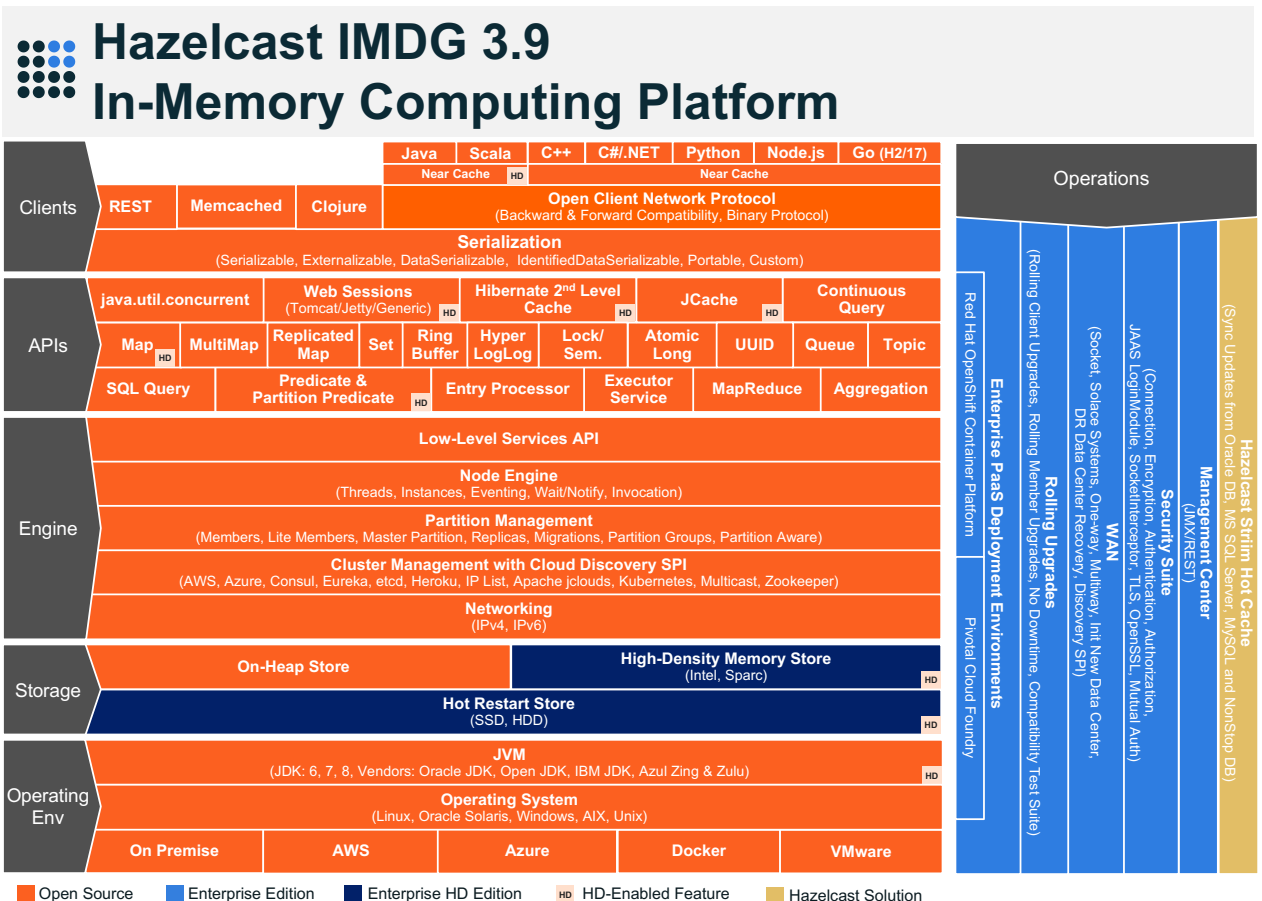
After years of working for an in-memory data-grid (IMDG) vendor, I find myself trying to correct the same set of very common clichés or misconceptions on IMDGs over and over again. Based on the facts that most

IMDGs are designed in a very similar way or based on similar technology seems to only stir the fire.

Looking at the recent past and that there's a growing need for IMDGs, I wanted to take the chance to dispel the most common myths and correct misun-

derstandings.

The given order below isn't an indicator about the importance of any particular myth (all of them are equally important and wrong), but might give an indication as to how often I have heard them and why they came to my mind in that order.



1. This IoT gateway inside a dust chamber is being tested per the IEC 60529 IP5X dust test.

### **1. YET ANOTHER NEW TECHNOLOGY!**

Probably most the common false fact; I often get told that IMDGs are yet just another new technology and that's why you should avoid using them.

This is wrong on two different levels. Let's start with the latter part of the claim. Even if it was new technology, we should still consider it. Our industry changes rapidly, systems need to provide access to more users each and every day, and in general technologies evolve way faster than they did 10 or 20 years ago.

So, is it just another new piece of technology? Looking at the history of IMDGs, it all began with a smallish company called Tangosol and its similarly named product in early 2000. In 2007, Tangosol was acquired by Oracle. That makes the technology almost old enough to be able to drink alcohol, at least in Germany with a legal age of 18.

Over the years the industry grew and today there are many different vendors, mainly in the Java space but also in .NET, C++ and Python. Most probably there are even more written in programming languages I'm not familiar with.

### **2. IT IS JUST ANOTHER NAME FOR NOSQL.**

Given the previously stated fact that IMDGs have been around since the early 2000s, a term that was created as a general concept in 2009 can hardly be the same.

However, the concept of key-value stores, which is a big part of the NoSQL movement, has a lot of similarities to an IMDG. One of the most common use cases for an IMDG is built around distributed caching. Since data is stored in memory, and most IMDGs have some kind of key-value store often highly optimized for network transport, it's the perfect caching solution.

On top of the common storage technology, IMDGs offer additional functionality like data structures (e.g., queue, list, ring-buffer), distributed computation layers, data aggregations, or general scalability solutions. Those additional features separate the IMDG space from pure caching solutions.

### **3. RAM IS TOO EXPENSIVE, FLASH MEMORY IS FAST ENOUGH.**

Another frequent claim is that RAM is much more expensive than solid-state disks (SSDs)—not even mentioning hard-disk drives (HDDs). Normally, the user wants to store gigabytes or even terabytes of data. Unfortunately it's hard to argue this, as it's obviously correct. But, if you need fast operation on stored data, there's a tradeoff for you to consider: runtime vs. cost factor.

So, to put this myth in perspective, prices for RAM are falling, and the decrease in value is substantial each year. Can you remember your first computer with 640-kB RAM? I do!

### **4. IMDG IS JUST ANOTHER NAME FOR (IN-MEMORY) DATABASES.**

Coming back to all of those "just another name" statements, here's yet another. Over the years, I heard so many different

views on IMDGs versus IMDBs, but many of these conversations miss an important point: Databases provide persistent storage.

I admit that in-memory databases are an interesting take on "hybridizing" data grids and databases, but still provide some internal way to persist data; IMDGs don't. Data grids might offer a way to integrate with external storage systems (e.g., RMDBS, mainframe, NAS), but don't persist on their own.

### **5. CLUSTERING COMES AT BIG COST.**

This is an interesting one, and these days I would consider it to be a myth of the past. A few years ago, the network deployed in buildings and even data centers was fast, but not that fast. That means latencies were a big burden whenever you had to cluster. The overhead, especially of wrong protocols, became a cost nobody wanted to pay.

Thankfully, the times of CORBA and SOAP are over, but JSON/HTTP1.1 APIs aren't that much better. Therefore, most IMDG implementations have their own, highly optimized binary network protocols. In the best-case scenario, the cluster communicates over persistent TCP connections or the UDP protocol and retrieves data with a single network hop.

Network interfaces and deployed network infrastructure have been increasing speed and decreasing latency for years. Most data centers have multiple 10G networks—maybe even 40G—deployed, and latencies are in the low one-number range. In reality, there's nothing to worry about anymore, except if you have a requirement for nanosecond latency.

### **6. SERIALIZATION IS GOING TO KILL YOU.**

This is one of my personal favorites, just because it's typically a homegrown problem. People often use the easiest possible solution without thinking about the drawbacks. That said, every time I go on-site and a user is complaining about speed, the first thing I check, and try to fix, is serialization. In the Java space, use of Java Serialization is one of the most common mistakes when it comes to any kind of transmission between different system endpoints, right before XML or JSON.

Given there are fast alternatives, which require little work on your data structures, Java Serialization really has no use case, except for prototyping—and please don't put prototypes into production! So let's assume that this issue has been solved years ago; it just hasn't hit the user yet.

### **7. I NEED ACID AND TRANSACTIONS.**

This misconception comes up quite often. When you look at a typical application, locking or transactions often isn't needed. By "no transactions needed," I mean no pessimistic locking transactions.

In most cases, relational databases already try to optimize transactions as much as possible, but nothing is as fast as no transaction at all. And let's be honest, nothing is more satisfying

than removing code that uses a transaction at least once a day.

So what would be an alternative to transactions? First we need to understand that not everything needs to be written in a durable fashion. If something fails, the user will just retry. A common example I use for those situations is YouTube. Imagine you upload a video but for some reason the database fails to write the entry and the video “disappears.” What would you do? Well you might think that’s “weird,” but you’ll go on and just retry.

Another solution that frequently works is optimistic locking. Not in the sense of a database transaction, but in terms of compare-and-swap (CAS) operations. The underlying idea is simple: I get the current value and keep it in memory; I create the updated value and pass the original and new value to the CAS function. The system will now compare the original value with the one in the system, and if they still match, the new value is stored. However, if the value has changed, the system will ask you to retry. In most situations, with low contention on data, retries are uncommon and the operations will just go through right away.

#### **8. IMDG IS IN-MEMORY ONLY—THIS ISN’T RELIABLE.**

With the name IMDG, users believe that you store information in RAM only. No persistence, as mentioned before. I also stated that most systems have some way to pass data through onto an external resource, like a relational database.

That alone, however, won’t dispel the myth. What happens if a node goes down—the data will be lost and need to be retrieved from the external resource? Wrong.

Information stored in memory usually has backups on other machines. Based on the implementation, these backups are hopefully split into multiple pieces and reside on more than one machine.

What is true, though, is that if you lose more machines than there are backups, some information will be lost one way or another. The number of backups is up to you, however—it’s your tradeoff between availability of data and available memory to store. But, and I think this is an important point, depending on why you use an IMDG, requirements change. Simple caches might not need backups at all. Don’t design your system for the worst-case scenario, because this will never actually scale. Instead, find meaningful settings for your use case.

#### **9. SETUP AND OPERATION IS COMPLICATED.**

This myth is unfortunately hard to argue. But, I would never support this claim. It depends on the solution you’re going to use. Hazelcast, for example, has a five-minute rule. If you can’t set up a Hazelcast cluster in five minutes, something’s very wrong.

Obviously going into production is more complicated than just starting a server for prototyping, but again, it all depends on the solution. So in one way, I’m happy to accept the myth, but I’m also happy to prove it wrong.

#### **10. WE’RE MOVING TO MICROSERVICES, AND THEY’RE STATELESS ANYWAY.**

That’s one hell of a claim. Microservice architectures and more specifically, REST APIs, are meant to be designed in a stateless manner. But let’s be blunt, what sounds like a great idea and simple to do often tends to be more difficult than first thought, especially for public-facing APIs.

Why? You need user authentication, you need rate limiting, you need other things. And while this can be shifted to other layers of the infrastructure (like gateway services), it’s still state and therefore somehow related to the microservice.

To scale those kinds of systems, information has to be shared between different instances for load-balancing reasons. IMDGs are a perfect fit, especially if they’re easy to set up, which is a great bridge from the previous claim.

I’m not saying that a stateless microservice or stateless architecture doesn’t exist, but remember there’s state somewhere. Moving it to different layers will maybe keep you off of it, but not others.

#### **11. ALL CLUSTER TECHNOLOGIES HAVE COMPLICATED APIS.**

This last myth is one of the claims I often use, too. It’s actually correct for most solutions. A lot of the cluster technologies out there have extremely complicated APIs. It sometimes feels like they do it on purpose to get cash from consulting, since almost nobody else is able to understand it. Otherwise, the wonderful statement of “it was hard to write, it must be hard to use” is true.

But similar to complicated operations and set up, this really depends on the solution in use. For instance, Hazelcast utilizes the Java Collections and Java Concurrency APIs and extends the interfaces where meaningful. For users, it feels like they just use common Java patterns most of the time. If there’s a need for power to get deeper into the system, the extensions offer all of the required functionality.

To conclude the myths and this myth in particular, I really think simplicity is the most important factor. For everything. So let’s not make things too complicated!