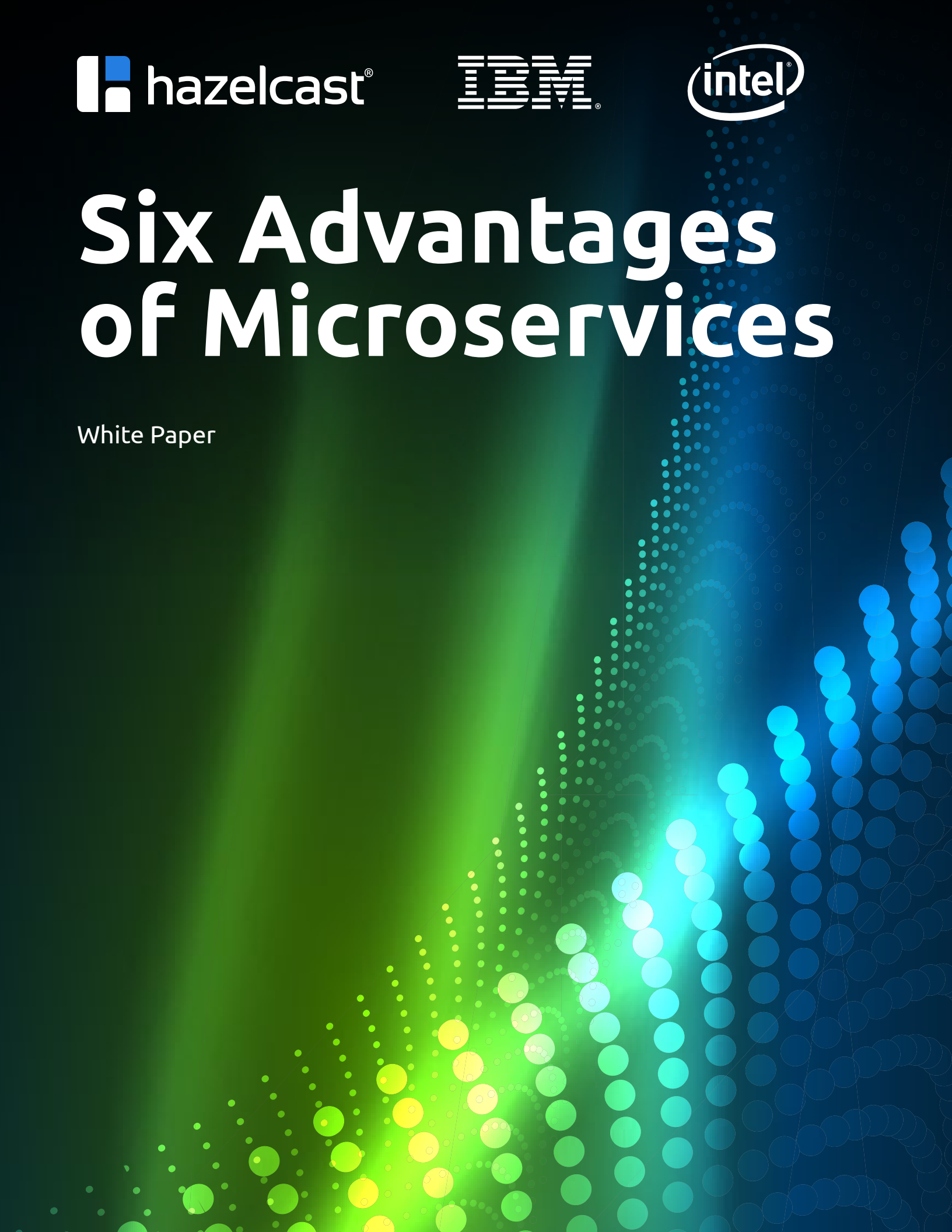
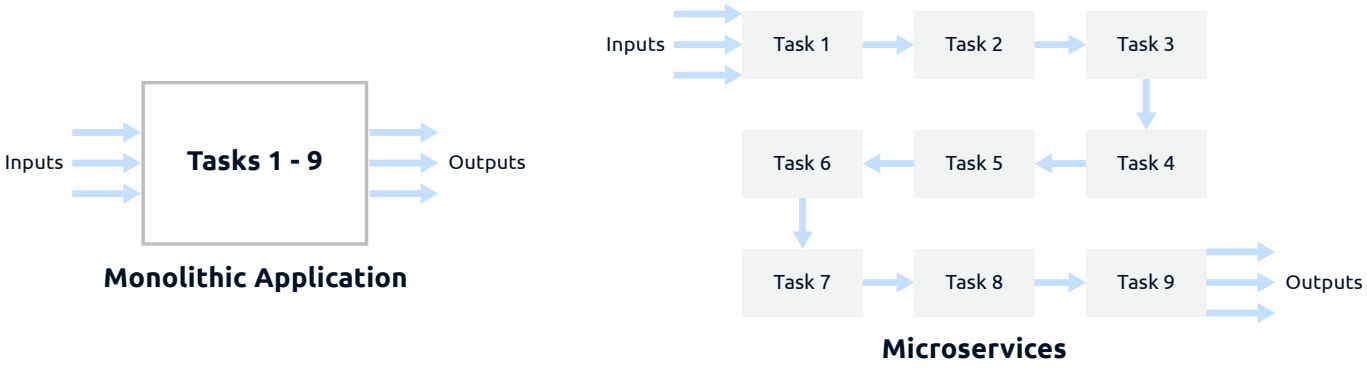

hazelcast
IBM
intel
Six Advantages of Microservices
White Paper

Six Advantages of Microservices

The topic of microservices continues to get significant buzz as businesses build more and more complex solutions. There are many advantages to microservices, and this paper aggregates them into six buckets to discuss why a microservices architecture can work well for you.

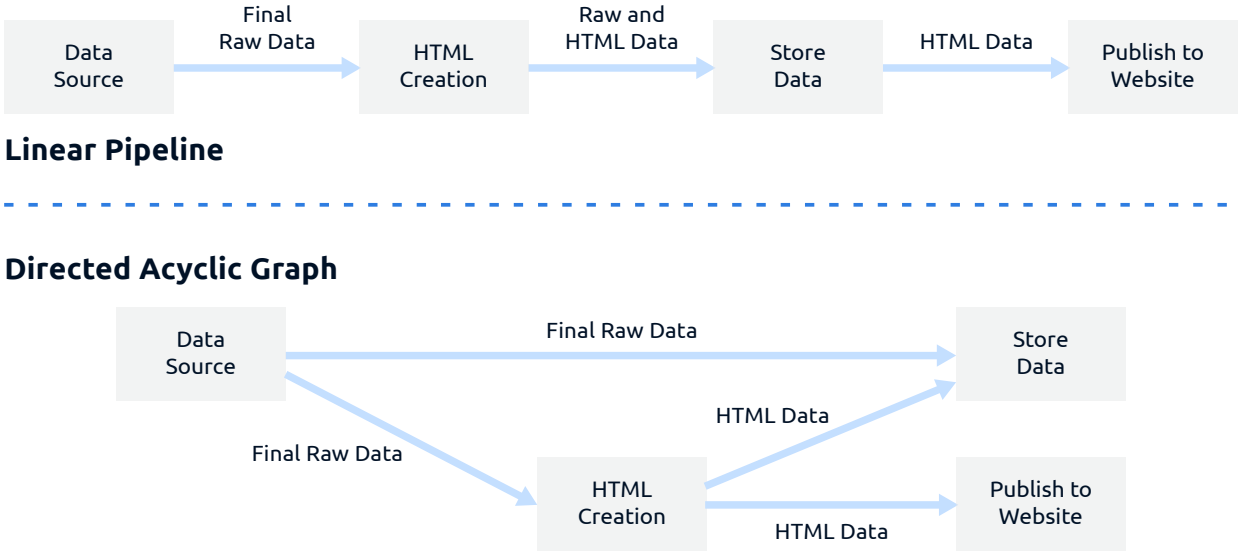
Introduction

Microservices are simply a set of software applications designed for limited scope that work with each other to form a bigger solution. Each microservice, as the name implies, has minimal capabilities for the sake of creating a highly modularized overall architecture. A microservices architecture is analogous to a manufacturing assembly line, where each microservice is like a station in the assembly line. Just as each station is responsible for one specific task, the same holds true for microservices. Each station/microservice has expertise in the respective responsibilities, thus promoting efficiency, consistency, and quality in the workflow and the outputs. Contrast that to a manufacturing environment in which each station is responsible for building the entire product itself. That is analogous to a monolithic software application that performs all tasks within the same process.



Microservices are individual components that work together to deliver an overall solution.

To be clear, the assembly line analogy does not imply a single linear flow, since microservices and assembly lines do not have to run in a strictly serialized order. With microservices, data can easily be copied and then distributed to multiple locations as part of the data pipeline, and thus can take multiple paths and be processed in different ways as in a directed acyclic graph (DAG). This gives you more flexibility in how you define the data pipeline, and also simplifies the effort in expanding your pipeline in case you want to create more outputs in the flow.



The data flow in a microservices architecture can be represented by a DAG.

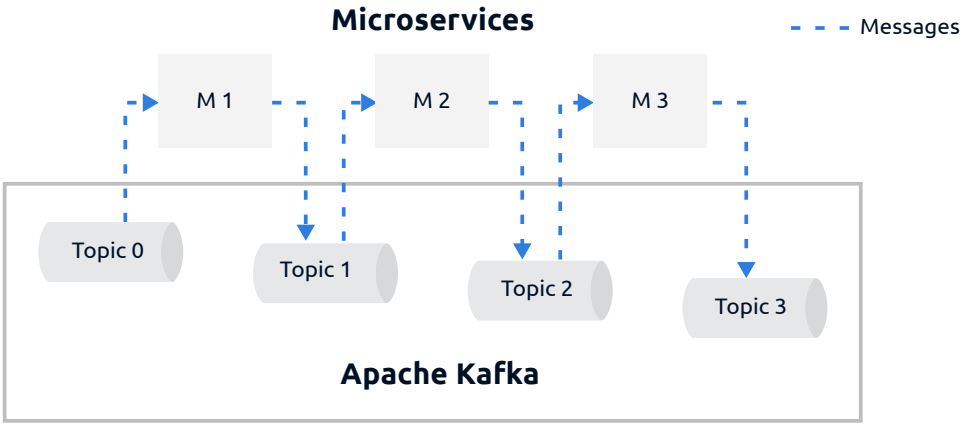
Although microservices have recently grown in popularity, the concepts behind them are not new. Topics like modular programming, separation of concerns, and service-oriented architecture (SOA) all have principles that align with the objectives of a microservices architecture. This means that microservices are based on best practices that we have used over the years, so their use can be easily justified. Some development teams may have already adopted a microservices architecture without necessarily calling it that. Big data pipelines typically process data one step at a time, which aligns well with a microservices approach. If you have a streams-based architecture (more on this later), you likely are running microservices within that framework. By having a more deliberate approach to microservices using the techniques you have already practiced, you can potentially get even more value out of your deployments. And microservices are a great fit for cloud environments, so if you have a Kubernetes cluster and/or are running in the public cloud, microservices represent a great architecture to take advantage of cloud capabilities.

Microservices Architecture Requirements

Building your applications and solutions in a microservices architecture does not require any specialized knowledge or experience. Much of the strategy will sound familiar, and again leverages concepts like modularity that you have likely practiced before. Certainly, inter-microservice communication is a key part of the architecture, and REST services are commonly used for data exchange. However, an increasingly popular design pattern for data exchange that might be new to you is the use of a central, lightweight messaging bus in an event-driven architecture. Since each microservice is related to one another as part of a larger solution, there needs to be an easy way to track its state and pass data between individual microservices. Since there are advantages of using a messaging bus with microservices over other means of data exchange, this paper will focus on that design pattern.

One option for the messaging system is Apache Kafka. Apache Kafka is a publish/subscribe (pub/sub) messaging bus that stores data in separate channels called "topics." A topic is a data structure that tracks data elements (or "messages") in order. Topics are flexible and can store messages in almost any format. From a coding perspective, it is easy to read and write messages to topics. And since this approach to messaging is decoupled from the microservices code, you get more flexibility in how you can exchange data versus having microservices communicate directly with each other.

You can set up a series of topics as sources and destinations for your microservices to make up the messaging system for your deployment. Each microservice will retrieve ordered data from one or more of its designated source topics, and after processing the data, will write the output in an ordered way to its designated destination topic (or writes no messages if it is the last microservice in the series). The output message is sent in a format that makes the most sense for the given microservice, and it is then up to the next microservice in the flow to know what that message format is, and how to consume it. In general, any microservice only needs to write to (at most) one topic, since multiple downstream microservices can independently read from that same topic without conflict.



Microservices can use Kafka topics to pass data to the next microservice.

To continue with the assembly line analogy, the messages are like the work-in-progress products, and each microservice performs more work on those products. Topics are like the conveyor belts that transport products between assembly line stations.

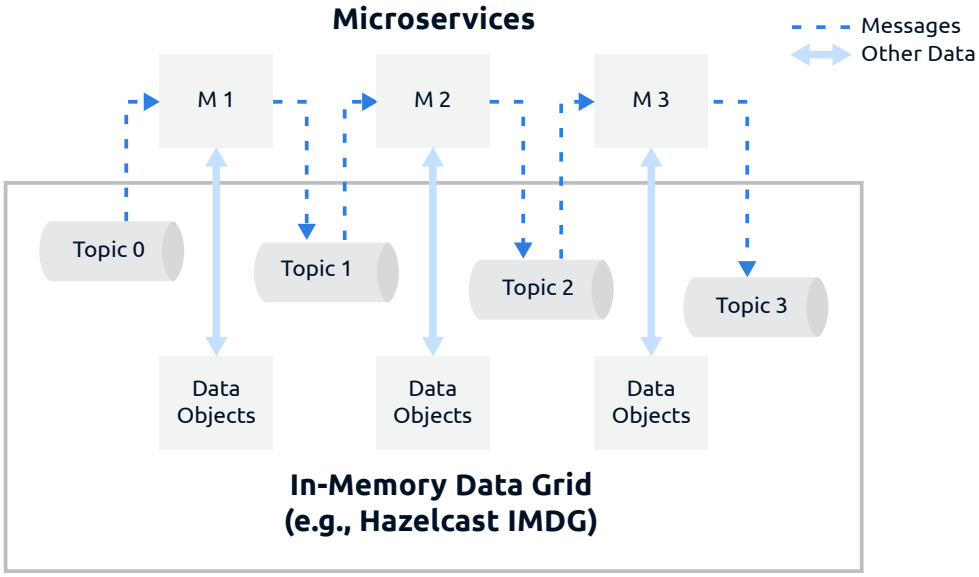
Since Kafka is often used for microservices architectures, it makes sense that terms like "streaming architecture" or "streams-based architecture" are closely related to (and sometimes used interchangeably with) "microservices architecture." Although those different terms imply a different focus of the architecture, the basic foundation is the same.

Another option for the messaging system is an in-memory data grid (IMDG), which is a distributed computing system that lets you store and process data with extreme speed due to the emphasis on RAM and parallelization. An IMDG stores data structures in memory, unlike Kafka, which relies on hard drives or solid-state drives (SSD). This means an IMDG lets your microservices access data at much higher speeds, which significantly accelerates the performance of your overall solution.

IMDGs also include topics as data structures, similar to Kafka, which you can use to pass messages between microservices. There are other data structures that can be used for messaging in IMDGs, such as maps (i.e., data structures that act like a key/value store), but for the sake of simplicity, our discussion will center on topics. In addition to speed, another advantage for using IMDGs for microservices is that you can leverage more data structures for your microservices to store and retrieve data beyond messages. The IMDG acts like a database that provides lookup data for stream enrichment, and also can store data to track state for your microservices.

By using a message bus as the messaging layer in a microservices architecture, it makes sense to also use a stream processing engine (like Hazelcast Jet) to build the microservices. This is only one example of a technology you can use, as microservices are actually great for supporting any technology for coding the business logic, as we'll discuss later in this paper.

Other technologies can be used for microservices messaging but are far from ideal. File systems and databases (either RDBMS or NoSQL) can be used, which are sometimes implemented as REST services, but they suffer from a significant performance disadvantage versus the technologies described above. Plus, the coding and administrative overhead to use these systems as a messaging system make them far from ideal for an otherwise agile-focused microservices architecture. Certainly, these technologies have been used for earlier generations of microservices, but application developers are recognizing the need for better suited technologies to ensure the performance levels and maintainability they seek.



In-memory data grids provide fast messaging and data storage/retrieval.

Why Microservices?

Microservices are ideal for modern deployments that use popular technologies like cloud (private, public, hybrid, multi), cloud object storage, Docker containers, Kubernetes, and RAM alternatives like Intel® Optane™ DC Persistent Memory. Below are some of the technical use cases that benefit from microservices:

- Operationalizing machine learning models (machine learning inference)
- Internet of Things data analytics and processing, from edge to cloud
- Fraud/anomaly detection
- Large-scale extract-transform-load (ETL)
- Transaction monitoring and processing (such as online payments and e-commerce)

The common theme among those use cases is that there are huge volumes of data, service-level agreement (SLA) requirements that need to be met, and a significant amount of code. Rather than trying to tackle such use cases with a monolithic approach, consider the following six reasons why microservices provide an advantage.

1. Microservices Are Easier to Build and Enhance

Microservices enable higher quality code by encouraging more focus. Since individual microservices, by definition, are smaller than monolithic applications, they have less scope and less code. This makes experimentation and testing with incremental code updates much easier. While the overall amount of code for an entire microservices-based solution may be similar to a functionally comparable monolithic application, the enforced separation of code makes each part easier to manage. Less code at a microservice level means less complexity, lower testing effort, easier unit testing, and lower risk for problems. And all of those traits mean it is easier to maintain and enhance the code, thus giving you greater agility and higher quality. It is easier for new members of the development team to understand the objectives of each microservice and to more easily contribute, and there is a lower risk of applications becoming "black boxes" in which the source code is too complex to dive into.

Thinking in a microservices context can help reinforce good development practices. If you can define your solution as a series of small tasks, then you can focus more on getting each task right with less concern on how each task will affect the next task in the process. This is where a decoupled messaging system like Kafka or Hazelcast IMDG will also help simplify the microservice development process. Each microservice can write to Kafka/IMDG in its own standardized format that the next microservice in the flow can understand. There is no strict messaging format that the microservices have to adhere to. For example, if a microservice only needs to send a comma-delimited list of integers, it can do so, and the next microservice is responsible for reading that format to do its own processing. This decoupling encourages experimentation and A-B testing to compare different versions of code in the same environment since it is easy to add in new code to the data pipeline.

The simplicity and modularity of each microservice also contributes to reusability. If a microservice is responsible for a generalized task such as converting CSV into JSON, then that module can be easily plugged into any microservices deployment that needs that conversion. This helps with rapid development and expansion for all subsequent microservices-based solutions. Of course, the microservice needs to be written to handle any type of CSV input, so you simply need to keep generalization and reusability in mind during development, which tends to be easier to do when coding limited-scope tasks.

A microservices architecture is not necessarily an all-or-nothing endeavor. You can incrementally add microservices to an existing monolithic application if you want to add more capabilities to that application. You can also incrementally migrate a monolithic application into a microservices architecture to gain all the advantages described in this paper. While this may sound like SOA, the key difference is that a microservices architecture makes each of your applications small and manageable, while SOA was largely about making large applications communicate with each other. With an SOA mindset, you would be able to share data in a less complicated way, but you would still deal with complicated applications that were not necessarily easier to build.

A microservices architecture also enables you to take advantage of a wider talent pool, because each microservice can be based on almost any programming language and environment, independent of all the others. There is no requirement on how each microservice is built. You can choose the most appropriate technology stack for each microservice. Separate application development teams with different skills can work together on a microservices architecture without requiring commonality on technologies and skill level. This also means that you are better able to incorporate newer technologies into your architecture in the future. Instead of struggling with technology standardization across the board, you can incrementally leverage new technologies to update only the microservices that can be made faster and/or more efficient. This gives you an opportunity to experiment with new technologies on a controlled basis to see how well they may work in other capacities.

2. Microservices Are Easier to Deploy

Microservices are easier to deploy than monolithic applications because they are smaller and thus have fewer environmental dependencies. One common problem with deploying monolithic applications is the unknown discrepancies between development environments and production environments. There are dependencies on OS versions, library versions, the amount of RAM, etc., and if you have a complex application with many dependencies, you may encounter deployment problems that are difficult to pinpoint. If you deploy microservices in an incremental manner, any issues you encounter with each individual microservice will be easier to track down since their smaller scope means they have fewer dependencies.

The reduction of dependency discrepancies from development to production is also a key advantage of a containerized architecture. The minimalist nature of individual microservices makes them even easier to deploy in a container and with other virtualized technologies, and doing so lets you further reduce the potential for dependency conflicts. Combining a container-based deployment with Kubernetes as the orchestration system further simplifies the deployment by efficiently allocating microservices to available resources.

Microservices are easy to deploy in a variety of deployment options, again because they are designed to be simple. They can be deployed on-premises, in the cloud, at the edge, in virtualized environments (especially Docker containers), in serverless environments, on one node, or on a cluster of nodes. In fact, microservices can be deployed in several different locations as part of the same overarching applications to take advantage of any of your available resources.

You can also deploy new versions of microservices without waiting for updates to other microservices. As long as your messaging format remains intact, you can redeploy updated versions of microservices without negatively impacting the rest of your system. If a problem occurs with your new microservice, you can just swap in the prior microservice to get the system fully running again.

3. Microservices Are Easier to Maintain, Troubleshoot, and Extend

Microservices more seamlessly enable ongoing maintenance and fault tolerance, which are major challenges in any large-scale software environment. Troubleshooting is difficult when a problem arises in a monolithic application since it is not always obvious how data is processed within the internal workflow. Isolating the source of the problem requires a good understanding of the entire code base. On the other hand, since each microservice has its own set of outputs, it is easier to identify which output has an issue, and therefore, which microservice needs attention. For example, if you discover that the final calculations in a microservice-based system are incorrect, you can trace back the lineage of the data within the system, one microservice at a time, to find the problematic calculation. The limited scope of each microservice, plus the fact that microservices make it easy to run automated, standardized test scenarios, makes them easier to debug and thus easier to create higher quality code.

The distributed nature of microservices across multiple computer servers and resources helps with fault tolerance strategies to enable 24x7 deployments. Any given microservice can be deployed redundantly so that there is no single source of failure. Each microservice performs independently of the other instances, and if any fail, the others will pick up the slack. This is another area where the decoupled messaging layer is valuable, as each microservice acts as an independent consumer, so a microservice failure will not disrupt the messaging

system. Deploying microservices in Docker containers in a Kubernetes cluster also simplifies the maintenance, as Kubernetes can manage some of the fault tolerance requirements and restart failed microservices.

Making incremental updates to your application is easy and relatively low risk due to the highly modular architecture. You can either plug in updated code to improve existing functionality, or you can add new code to extend your system to provide more outputs. The decoupling of the messaging layer reduces the risk of losing data (or other dramatic failures) if the new updates are found to be problematic. Even if you need to make sweeping changes to your application, you can do so with far less risk than in a monolithic application. For example, if you change your messaging format in an updated microservice, you can send out both the old format and the new format at the same time to separate topics. The existing microservices that are next in the pipeline will continue using the old format from the original topic, and in the meantime, you can write an updated microservice that will leverage the new format from the newly added topic. This lets you run old code alongside new code as an extra defensive measure against problems.

And should an unforeseen error occur later, you can shut down a given task which will temporarily stop the flow of data, but the messages will be simply queued up in the messaging layer and will be read once the task in question is restored. The message bus ensures all data is preserved until all microservices are operational again.

4. Microservices Simplify Cross-Team Coordination

One of the challenges of any large-scale software development effort is the risk of over-complicating the integration points. In a microservice, the internal workflow can be as simple as reading data from a source, performing an action on the data, and then sending outputs to a destination. There is no need for extensive planning for what data should be shared on which integration points, as was the case with SOA. Microservices typically deal with small amounts of data at a time, so it is easier to manage and share the output once the data is processed. This simplicity in the processing and the data scope leads to simplicity in the handoff.

To support this simplification, microservices should leverage a lightweight messaging system to let microservices communicate, which simplifies the coordination across development teams. Each microservice can send messages/data in whatever format you want, which naturally encourages you to adopt the simplest format possible. There is no requirement around a universal messaging standard, which ironically tends to complicate future communication. Once each format is documented, the development team of the next microservice in the pipeline can use that format to accept messages. And since each microservice performs a limited task on the data, there is no need for complicated outputs. The complicated messaging formats of years ago are not needed since a comprehensive message does not need to be shared. This simplicity in messaging helps when the type of information changes. The updates in the format will likely be simple, thus making it easier for the next microservice to accommodate those updates.

This lightweight messaging pattern helps to avoid overly complicating the communications topology. Risks such as those expressed in Conway's Law, which states that your designs will become copies of your team's communications structures, still exist, but only if you retain a monolith mindset. For example, if you build a single large team where each developer believes his/her modules need to integrate with all the others, you end up with a complex architecture with many integrations, APIs, and data structures throughout. On the other hand, if an architect can define the components of a microservices architecture, and then have the team leaders define the team structures to reflect the architecture, you will simplify the effort to implement the system's communications.

5. Microservices Deliver Performance and Scale

The distributed architecture of microservices opens opportunities for increasing performance and scaling out. Just as each microservice can be run redundantly for fault tolerance, this redundancy also enables greater parallelism that adds performance and scale. More microservice instances mean more computing resources allocated to the respective tasks. At the same time, this approach enables greater scale, so more data can be processed via increased parallelism. This is essentially a divide-and-conquer approach to processing your data, where you spread out work across more microservices to take advantage of more CPU power and RAM. Whenever a bottleneck in the pipeline arises, more instances can be deployed (either manually or with Kubernetes in a containerized environment) to provide load balancing. Again, the decoupled messaging layer helps in the easy incremental deployments of additional microservices.

The ability to add performance and scale is not merely about meeting SLAs with future growth in mind. It is also about having the extra headroom to experiment with new capabilities. This is especially critical in production machine learning deployments where experimentation is inherent to the deployment lifecycle. A machine learning model is rarely considered good enough, and the accuracy can only be fully assessed by running the models on live data. With the ability to experiment, you can keep your day-to-day operations running reliably while simultaneously exploring new opportunities for improvement.

Of course, the performance and scale advantages only apply to code that is specifically part of the microservices deployment. In other words, if there are external resources like a remote, third-party API that is called as part of the pipeline, then a microservices architecture will not necessarily help you there. You need to make sure that the external resource can scale as well so that your redundant microservices can leverage it in a parallelized manner.

6. Microservices Simplify Real-Time Processing

The demand for immediacy continues to grow, and businesses need to respond more quickly than before to maintain a competitive advantage. That is why streaming architectures and real-time processing are hot topics today, and those align well with microservices. Real-time processing introduces many challenges around performance, scale, reliability, and maintainability, and the microservices approach can help alleviate the difficulty.

Real-time processing typically starts with a continuous flow of data, like from a number of Internet of Things devices. The incoming data typically has to go through several processing tasks like enrichment, aggregation, and filtering. Each of these steps can be done by a microservice to form a clear separation of tasks that hand off data to the next. The messaging layer helps to facilitate this real-time flow, and perfectly aligns with the incoming flow of streaming data.

In some cases, the processing must be done immediately at the edge, where there is typically limited physical space that requires smaller footprint hardware. Since these hardware systems tend to be less powerful than servers in data centers or in the cloud, lightweight, efficient software packages are necessary to run with the reduced computing power. Technologies like Hazelcast Jet were designed for a wide variety of deployments, including at the edge, to let you offload some processing to edge computers rather than trying to transmit all data to a centralized data center. This deployment model is essentially a widely distributed microservices architecture, where some of the tasks are performed close to the data source, and then data is delivered to the cloud or other data center to perform additional processing on higher-powered computers.

Conclusion

As discussed in this paper, there are several good reasons to adopt a microservices architecture. Now that we are dealing with more data than ever before, thinking more strategically about how to handle that data is an important priority today. Investigating the new technology innovations today will further help your journey with microservices.

Innovation is not limited to software technologies, as we earlier alluded to the Intel Optane technology which introduces us to a new level of hardware performance. Leveraging in-memory RAM has, in the past, been an expensive proposition, so with Optane, in-memory speeds are more economically accessible. In volatile memory mode, Optane behaves like RAM with comparable speeds (depending on the workloads and data models), but at approximately half the cost. This encourages more businesses to turn to in-memory technologies to accelerate application performance. Not all in-memory technologies can take advantage of Optane off-the-shelf, but you should consider the Hazelcast suite of technologies (Hazelcast IMDG and Hazelcast Jet) which are certified to use Optane (along with other Intel technologies) for much higher-speed processing compared to your disk- or SSD-based systems.

Relevant Technologies for Microservices

As discussed in this paper, microservices architectures can benefit from in-memory, cloud, and streaming technologies. Hazelcast, IBM, and Intel are working together to offer solutions that deliver on today's demanding requirements around data. Performance, scale, security, reliability, and agility, especially in cloud-native environments, are the key components to deploying successful, business-critical systems. With the in-memory computing platform from Hazelcast, the innovative hardware from Intel, and the edge-to-cloud enterprise-wide solutions from IBM, data-driven businesses have solid technology choices for their current and future digital strategies.

The Hazelcast In-Memory Computing Platform

Hazelcast delivers the industry-leading in-memory computing platform that provides Global 2000 enterprises with ultra-high performance for time-sensitive, cloud-native applications.

The Hazelcast In-Memory Computing Platform comprises Hazelcast IMDG, the most widely deployed in-memory data grid, and Hazelcast Jet, the industry's most advanced in-memory stream processing solution. This technology is uniquely designed to allow you to gain computing insights faster, enable actions within shorter durations, and engage new data at the speed with which it is arriving. In addition, a distributed caching architecture allows you to scale up to hundreds of terabytes and scale out for maximum efficiency when dealing with remote data or edge processing.

Built for ultra-fast processing at extreme scale, Hazelcast's cloud-native in-memory data grid and event stream processing technologies are trusted by leading companies such as JPMorgan Chase, Charter Communications, Ellie Mae, UBS, and National Australia Bank to accelerate business-critical applications. The world's largest e-commerce sites rely on the Hazelcast Platform for sub-millisecond response times to support massive volume spikes associated with Black Friday, Cyber Monday, or Singles' Day.

Intel® Optane™ DC Persistent Memory

Since many of the performance requirements are dependent on in-memory processing, the one big hurdle that emerges is the cost of random-access memory (RAM). In many cases, the investment in more RAM-heavy hardware servers is justifiable, and as RAM prices continue to decrease, the use of in-memory processing becomes more accessible.

Recent innovations make the adoption of in-memory processing even more practical. The Intel Optane DC Persistent Memory technology offers two ways in which in-memory processing can be more cost-effective. The first way is in volatile memory mode, in which Optane chips act as an alternative to RAM and run at nearly the same speed but at a much lower cost and much higher capacities. This lets businesses more easily justify in-memory technologies and thus take advantage of the performance benefits that in-memory processing offers.

The second way in which Optane supports in-memory technologies is in the persistent mode. In this mode, Optane can be used as a faster alternative to solid state drives (SSDs). For example, Hazelcast provides a hot restart capability in which in-memory data is persisted in non-volatile memory so that if a node goes down temporarily, it can be restored quickly by reading data from the hot restart store. If the hot restart data is stored in Optane in persistence mode, recovery of that node can be up to 3.5x faster than using SSDs.

IBM Cloud Paks and Edge Application Manager

As more businesses pursue cloud strategies around public, private, multi, or hybrid cloud, the right foundational software will help on that journey. Cloud deployments are just one part of an overall distributed computing system that includes computing at the edge, making system deployment more complex.

IBM has key initiatives around solving edge-to-cloud challenges to help businesses get more value out of their distributed data. With their six Cloud Paks (Applications, Data, Integration, Automation, Multicloud Management, Security), IBM provides a faster, more secure way to move core business applications to any cloud environment through an enterprise-ready, cloud-native software stack. Built with Kubernetes on Red Hat OpenShift, the Cloud Paks give customers flexibility and agility to drive digital transformation strategies. Hazelcast is included in the Cloud Paks to provide in-memory speeds in a cloud-native framework to all applications deployed in the Paks.

The IBM Edge Application Manager (EAM) extends the reach of business data by enabling computing at the edge. Since edge computing typically entails remote, hard-to-access locations with limited physical space and thus limited computing power, some of the most important factors for a successful deployment are efficiency, reliability, and security. Together with IBM EAM, Hazelcast provides portable and lightweight yet powerful in-memory and stream processing technologies to enable edge computing and let businesses process, analyze, aggregate, and/or filter data where it is created.

2 West 5th Ave., San Mateo CA 94402 USA
Email: sales@hazelcast.com Phone: +1 (650) 521-5453
Visit us at www.hazelcast.com

@hazelcast
https://www.linkedin.com/company/hazelcast
https://www.facebook.com/hazelcast