Step Up to Modern Cloud Development

Microservices and Containers: Patterns for Scalable Agility in the Age of Digital Disruption

By Boris Scholl, VP Development, Microservices, Oracle Cloud

Vol 1.0, July 2017

ORACLE

Digital disruption. You can't work in IT without hearing it often. But unlike the buzzwords that marketers throw around to sell you more stuff, this one was born of IT—from architects and developers, not unlike you and me. Amazon, Lyft, Netflix—each of these now-famous brands disrupted long-standing industries by delivering better customer experiences faster and cheaper than their competitors by leveraging—and often inventing—modern, digital technologies. And this is only the beginning. It is estimated that in less than ten years, three quarters of the S&P 500® will be made up of digital-native companies we haven't even heard of yet. If you're an incumbent, it's a sobering statistic (Figure 1).

Average lifespan of S&P 500 companies is on the decline

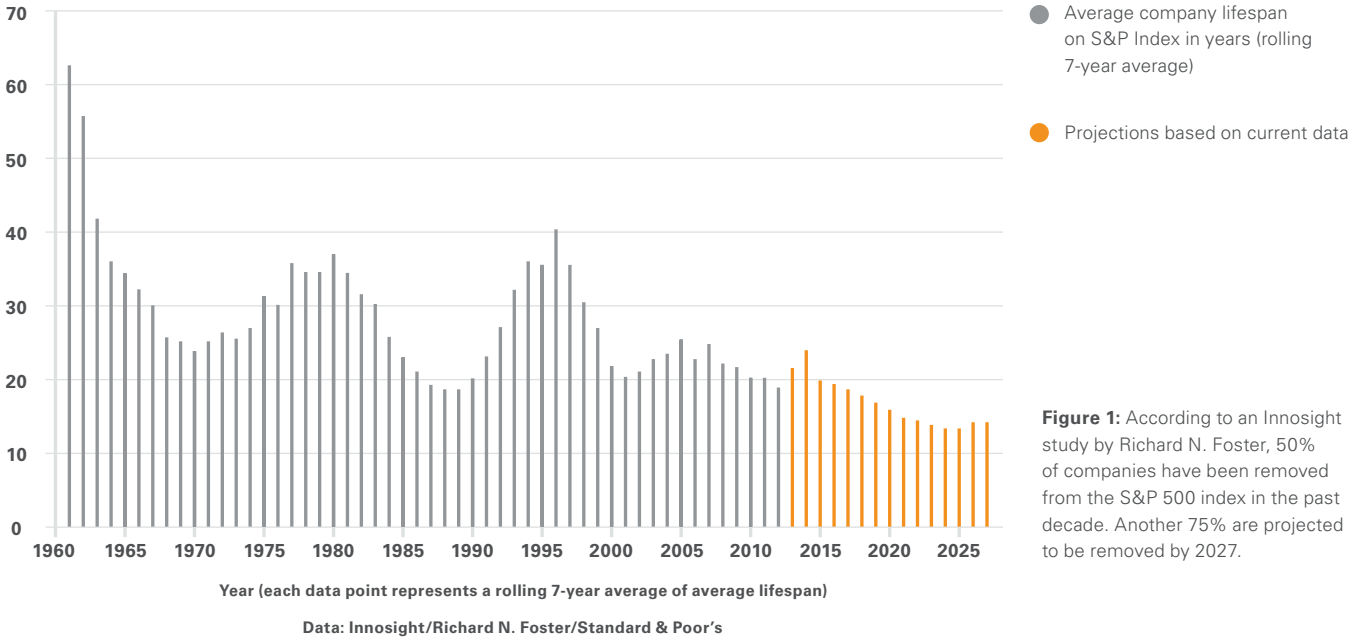

Figure 1: According to an Innosight study by Richard N. Foster, 50% of companies have been removed from the S&P 500 index in the past decade. Another 75% are projected to be removed by 2027.

Traditional application development, with its long release cycles and inability to react to rapidly changing requirements, is incompatible with today's agile, data-driven digital business. Modern IT departments have already embraced concepts such as DevOps, automation, software modularity, continuous development and deployment, and more frequent releases. However, rapid advancements in cloud computing, and new tools such as containers, orchestrators, and managed container services, are helping many forward-thinking teams meet the ever-increasing need for speed, flexibility, innovation, and scale by adopting a microservices approach to building and deploying applications.

My goal for this paper is to help you better understand microservices and containers, so you can thrive in this fast-paced world of distributed application development. I will outline the many challenges and complexities to consider when transitioning to microservices-based applications. Most importantly, I will share field-tested best practices and patterns I've encountered while working with customers to help solve some of the most common issues associated with adopting this modern style of application development. Shall we begin?

Why Microservices?

Microservices architecture is a modular, agile approach to software development. Unlike tightly coupled monolithic applications, where all components for the system are compiled together in a single codebase, microservices-based applications are created from a suite of smaller, loosely coupled services. Each service runs its own process and focuses on a specific function of the larger application, communicating with other services via lightweight APIs (Figure 2).

To fully appreciate the benefits of adopting a microservices architecture, let's compare it to the monolithic approach in three key areas: speed of deployment, ability to scale, and the ability to take advantage of the latest technology innovations.

Microservices-based application with three services

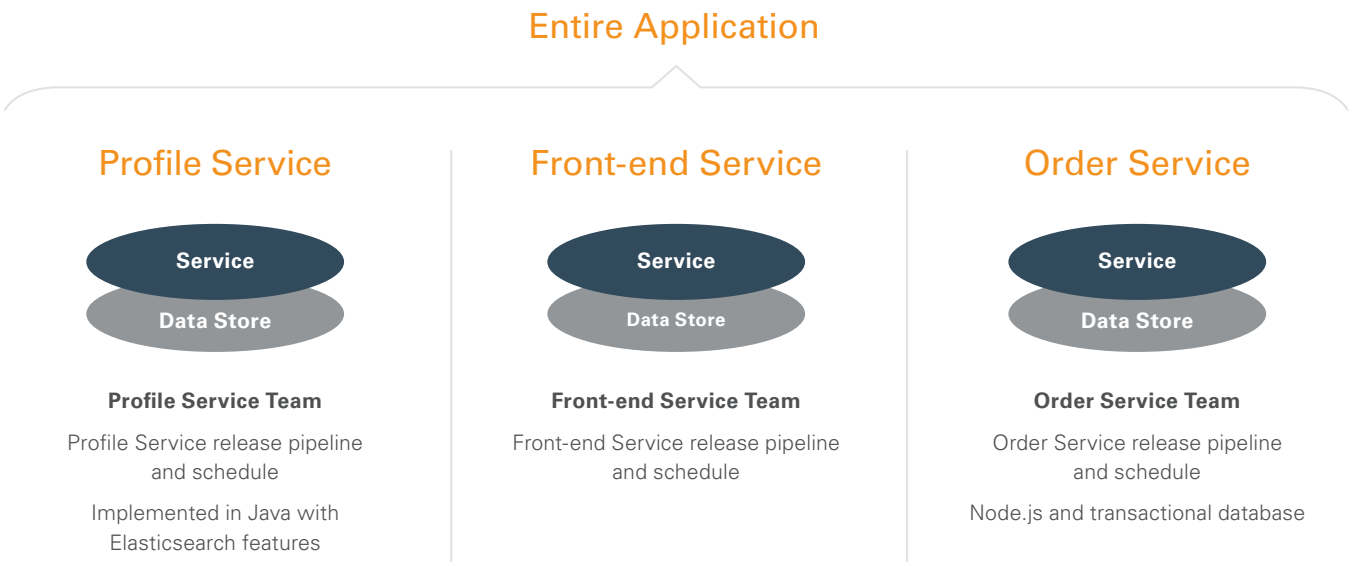

Figure 2: A simple microservices-based application built with three services. Each service is autonomous, using its own framework, programming language, libraries and data stores.

Speed of deployment

Because monoliths are built and deployed as a single application, fixing a single bug in one component means rebuilding, testing, and redeploying the entire application—including all the components that didn't require changes. All this to ensure that one small change didn't break any other component. Depending on the size of your application, and the processes and technologies used, this can take considerable time—not to mention risk. This is why teams managing large monoliths typically group enhancements and bug fixes into a handful of longer, planned releases. While more efficient in some ways, it makes it difficult to respond quickly to customer demands and rapidly changing market conditions.

In contrast, the modularity of microservices allows you to develop, test, and deploy individual components of your application without impacting other services and systems. Critical bug fixes or rollbacks can be handled immediately,

without impacting services in production. You can even assign smaller, agile teams that are best suited to support each service—all of which enables you to react faster to input from customers and market pressures, and deliver continually faster cycles of innovation.

Cost-efficient scalability

You scale applications by adding and load balancing machines or instances that are running the application. With a monolith, if just one feature requires an additional machine, the entire application will need to be replicated and load balanced. This is obviously an inefficient use of resources that adds unnecessary expense.

The modular nature of microservices allows you to scale each service as needed, using an instance that best meets its specific resource requirements. If a service requires more memory, you can deploy it to an instance with more memory. The ability to scale your application at the discrete services level allows you to optimize resources and increase your return on investment.

Ability to innovate with different technology stacks

Monoliths are typically developed in one language on one technology stack, often on one specific version of a stack. What happens when a new version or stack comes along, offering some cool new capabilities for your application? You already know the answer. If one component can't support the newer version, you can't adopt it—not without a substantial investment of time and money to rework, test, debug, and redeploy the entire application.

When services are independent and self-contained, you don't need to worry about dependency conflicts. Each service can leverage its own programming languages, different frameworks, libraries, data stores, different versions of these, even different operating system platforms. This allows you to pick the best technology for the feature, while avoiding stack version conflicts between features or libraries.

Microservices: Challenges and Considerations

While microservices have the potential to offer enormous advantages in application agility, innovation and scale, the shift to a microservices-based approach is not without challenges. Distributed computing adds a level of complexity that can sink a team that is not fully prepared, equipped, and organized to operate in a distributed fashion. Here are a few important differences to consider:

Increased complexity

Instead of one big deployment, you are now dealing with potentially hundreds of smaller service deployments that need to work together seamlessly. This will require service registry and service discovery capabilities for new and updated services to detect one another, and synchronous or asynchronous messaging for handling service requests. You will also need to rethink your load-balancing strategy and monitoring capabilities to ensure each service remains performant and is not running out of required resources.

Network congestion and latency

Because microservices communicate through standard protocols such as HTTP, network performance is critical. Considering there can be hundreds of services in one application, and requests can span multiple services, it's easy to see how overall application performance can suffer if the network is not taken into consideration. Data serialization and deserialization can also cause significant latency as the same data is repeatedly passed from one service to the next.

Data consistency and integrity

Since each microservice has its own state store, you will have to deal with the consistency and integrity challenges that come with decentralized data. For example, an order service needs to check availability of a product, so it references data in the catalog service. The same data now resides in both services. What happens when the data in

one of these services changes—if a product is pulled from the catalog, for instance? You will need a mechanism to ensure that the order service is made aware of this change.

Fault tolerance and resiliency

While individual microservices are particularly fault tolerant since each runs its own process or container (more on that in a minute), faults can be more prevalent with microservices-based applications due to network communications between services. For example, if a microservice takes too long to respond, exhausting all the threads in the calling service, it can cause cascading failures in the entire call chain. Not handling faults properly can hurt your application's uptime SLA.

Diagnostics

Logging and tracing in microservices-based applications require serious planning, as there can be hundreds, if not thousands, of microservices producing a huge number of logs for a single application. Requests typically span multiple services, so it's important to tag them in such a way that you can find and view the entire request across all services.

Versioning

In monolithic systems, the code consuming an interface is typically deployed along with the interface, so any breaking changes are usually caught at build or during integration testing. Because microservices tend to be on different update cadences, changes to an interface will not necessarily be handled by a consuming microservice right away. Therefore, it is critical to think through and agree on common service versioning techniques to ensure that all consuming services will still work as expected.

Containers: Why They're Good for Microservices

While often used interchangeably, microservices and containers are not the same thing. Microservices are an architectural approach to software development. Containers are encapsulated, individually deployable components running as isolated instances, provisioned with minimal resources required to do the job. A microservice can run in a container or as a process, but there are some very good reasons to use containers for easing both the development and operations side of microservices-based applications.

Containers provide isolation and speed to scale

Processes start fast, and dynamically share resources such as RAM. However, running one or more microservices per process can create "noisy neighbors." Poorly coded microservices running as a process can potentially compromise the integrity of the entire machine.

Running microservices inside a VM provides the necessary isolation, but at the cost of scalability due to lengthy boot times associated with the VM's embedded operating systems (OS). Containers, on the other hand, boot in seconds, thanks to OS-level virtualization where calls for OS resources are made via API. Because they package the service code, runtime, dependencies, and system libraries together with their own view of operating system constructs, containers offer the isolation microservices require at a suitable speed needed to scale (Figure 3).

Density and isolation levels compared

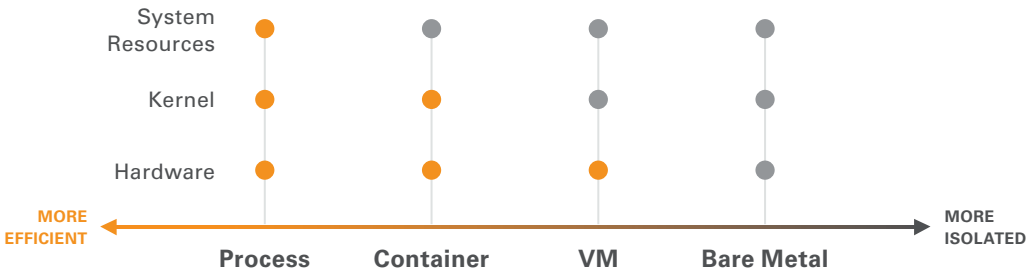

Figure 3: Containers provide a good balance between isolation and density for microservices deployments that need to scale quickly.

Containers simplify operations

As stated earlier, one of the benefits of microservices is the flexibility of choosing the best programming language and tech stack for each service. While great for development, it can quickly become an operational nightmare when it comes time to deploy all these different applications. By packaging microservices in containers, your ops team need only know how to deploy containers, and nothing about the different types of applications running inside them. You can think of containers as a bridge between dev and ops. You can also avoid potential service failures due to missing dependencies or mismatched versions on the host system, as the code, frameworks, and everything the service needs is packaged together in an immutable environment. By running one service instance per container, it's possible to tie system telemetry (CPU usage, memory, etc.) to the service itself. Containers further simplify operations by shielding developers from the need to concern themselves with machine and OS details. If your infrastructure team decides to switch the Linux distribution on the host, the application would not be affected as containers run on any Linux distro.

In the following example (Figure 4), an application consisting of a catalog service, a profile service, and two versions of an order service are each packaged inside its own container. If one service misbehaves and uses up available CPU or RAM, the other services are not affected because the encapsulated service can only use the resources assigned to the container.

Containerized services running on single host

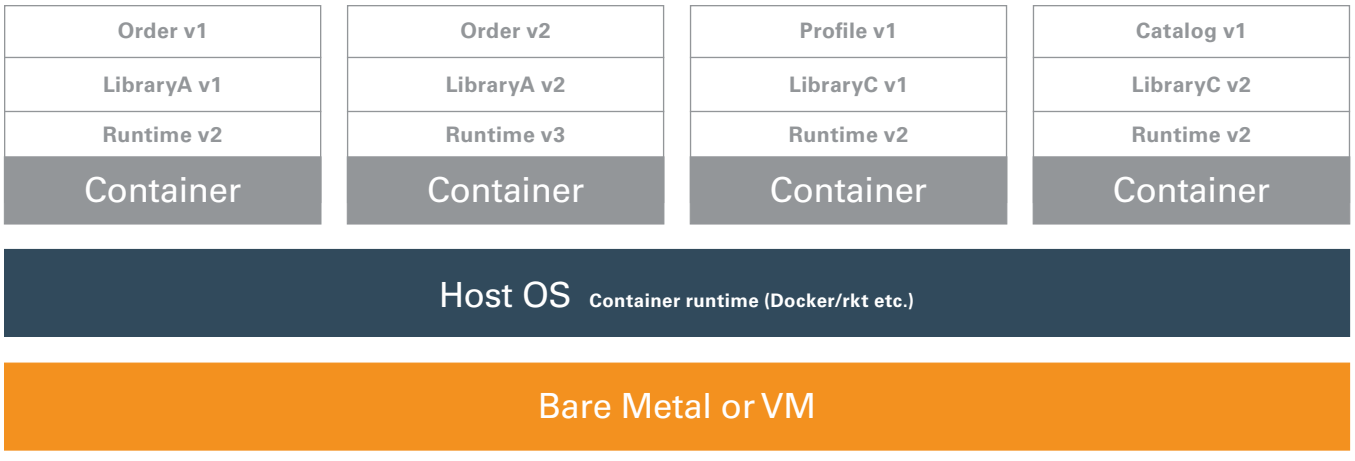

Figure 4: Multiple containerized services and versions of services can run on the same host, regardless of the differences in runtime and dependent components.

Other Challenges: Scheduling, Discovery, Routing

Container orchestrators—sometimes referred to as schedulers—can help you manage all the complexities associated with distributed, microservices-based applications. Because they're distributed, you'll be running your applications on a cluster—a series of networked computers (called nodes) viewed as a single system. While scheduling new services in a cluster is simple enough, orchestrators are invaluable for understanding all your services' unique requirements, their dependencies, available system resources, placement, and resource constraints—everything necessary to locate the best place for each service. A good container orchestrator can keep your application alive if a service fails, move services from failed nodes or from nodes being serviced, and enable rolling upgrades so that services are "always on."

The most popular orchestrators that you install and manage yourself are Kubernetes, DC/OS, and Docker Swarm. You can also use managed container services, such as the Oracle Container Cloud Service. The advantage of using a fully managed service is that you don't have to worry about the underlying infrastructure. From a core functionality standpoint however, all the container orchestration solutions, whether on-prem or managed, offer a similar set of functionalities.

Service registry and discovery

Now that you will be relying on an orchestrator to find the best place for each service, it's likely you won't know where each one is located within a cluster. Service registry and discovery solutions are used to ensure that new or updated services are easily discoverable by others within the system. Often referred to as east-west routing, service discovery solutions can also include additional services' metadata, providing the ability to maintain the state of each service instance with regular health checks.

Service Registry of microservices instances and endpoints

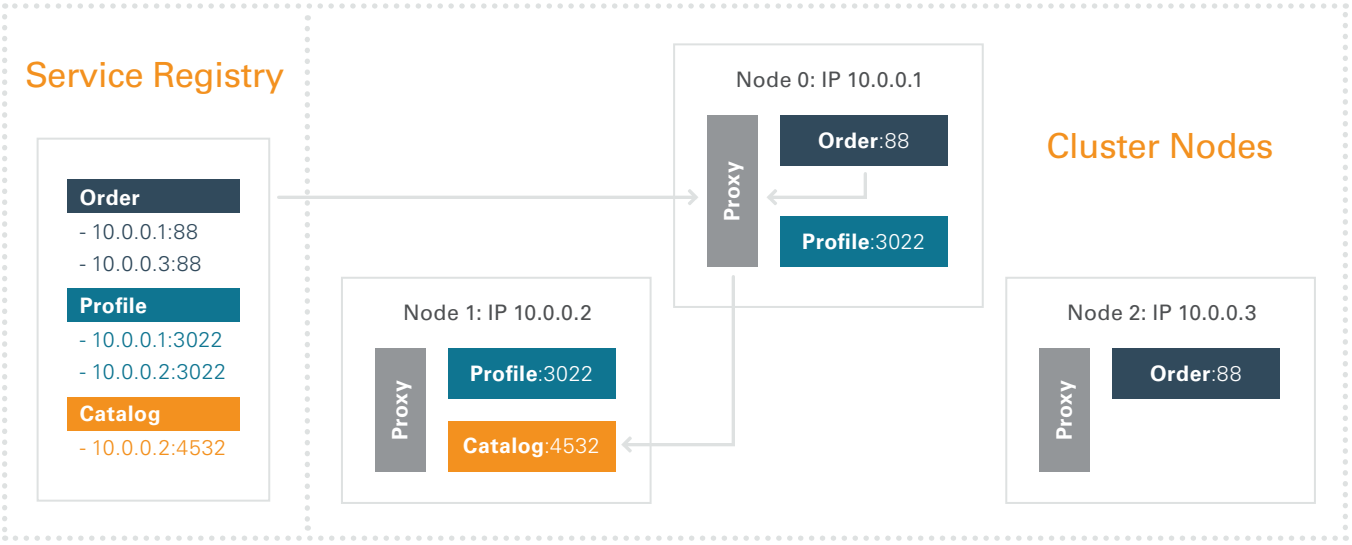

Figure 5: In this example, one instance of the order service is calling the catalog service. The order service calls a well-known endpoint on the node and the local proxy services handle the lookup for that service.

API gateways for external client requests

You will also need to consider how to route traffic from a client outside your cluster to services inside your cluster (referred to as north-south routing). The most common approach involves the use of an application or API gateway. In a microservices architecture, an API gateway is used to aggregate and route client requests to the appropriate services (Figure 6). Additional use cases for API gateways include authentication offload, SSL offload, quality of service throttling, and monitoring, among others.

Typical gateway deployment patterns include peer, dedicated, and standalone:

- **Peer gateway** routing is where you run the gateway on every node in the cluster. It is typically used with smaller cluster sizes.
- **Dedicated gateway** routing has gateway services placed on specific nodes in the cluster using placement constraints. The public load balancer is set up to direct traffic only to these nodes.
- **Standalone gateway** routing works similarly to dedicated, except the nodes hosting the gateway services are not part of the cluster scheduling.

API gateway aggregates and routes service requests

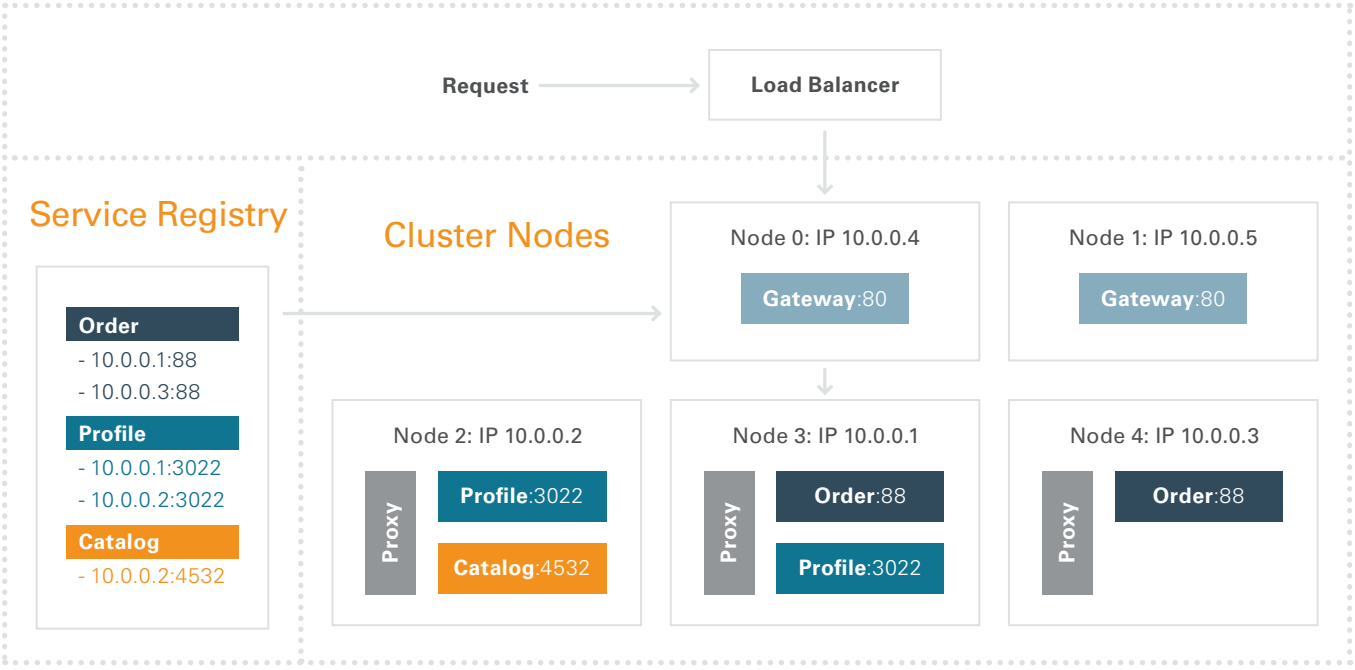

Figure 6: An external request is made to the order service. The load balancer directs traffic to one of the API gateway instances. The gateway routes the request using a path (/order) to one of the order service instances returned by the service registry.

In addition to service registry, service discovery, and gateway services, there are also infrastructure and networking issues to deal with, such as port conflicts, and assigning IP addresses per container. Fortunately, container orchestrators are evolving rapidly to take more of these infrastructure issues into consideration.

Eight Common Patterns and Best Practices for Microservices Implementation Success

As you can see, there is a lot that goes into successfully designing and deploying applications with containers and microservices. While there are likely as many approaches as there are applications for using a microservices architecture, we have collected the following best practices based on recent engagements with various customers. Here are seven field-tested patterns for dealing with some of the most common issues:

1. Design for stateless compute

In non-dynamic environments, state information, such as session, application, or configuration data, is typically stored with the service instance. With microservices, you never know where new instances get spun up, so state information stored in this manner will not be able to be used by any new instances. One way to solve for this is to consider pushing state data into highly available managed services.

2. Design for failure

Designing for failure means writing code to deal with fault scenarios in such a way that the application remains responsive and always returns something to the user. Here are some common fault scenarios and patterns for ensuring success:

Patterns	Fault Scenarios
Retry	A microservice that fails to receive an answer to a request—no matter how short lived—will go into failure handling mode without a retry policy in place. Retry policies offer enhanced failure handling by enabling the underlying infrastructure to retry requests without the requesting microservice's knowledge. Retry strategies include: • **Fixed interval**—retry at a fixed rate. Caution: choosing very short intervals could be interpreted as a denial-of-service attack. • **Exponential backoff**—using progressively longer waits between retries. • **Random**—establishing random retry intervals.
Circuit breaker	Circuit breakers prevent a service from performing an operation that is likely to fail. They are used to stop remote calls to a downstream service that is not functioning properly. This prevents network congestion and cascading failures due to failed retries from multiple services. Self-healing circuit breakers will check the faulty service at regular intervals and reset the breaker once the service begins functioning properly.
Bulkhead	Bulkheads can prevent a single component failure from taking down the entire system. Imagine a multi-threaded application where requests to one component start to hang. Eventually, the entire application will become nonresponsive as all request-handling threads, waiting for an answer, begin to hang. Bulkheads prevent this by assigning each component only a certain number of threads, so the system won't use up all available threads on a failed component.
Fallback	Fallback provides an alternative solution during a service request failure. For example, a movie service designed to serve up a list of popular movies could have a fallback service set to serve the last cached list, or a list of generic movies, in the event the first service fails.

3. Design for backward and forward compatibility

To ensure updates to one service don't break any services it communicates with, you should strive for backward compatibility. Never rename existing fields when updating a service with new functionality. New fields added to the API should be optional, or should have sensible defaults. Lastly, always test the new version of the API by passing old messages to ensure there are no compatibility issues with other services before releasing the new version.

Forward compatibility is less common, but essential if rollback functionality is required. To achieve forward compatibility, a service must be able to work as intended against an updated version of itself. This can be tricky, but you'll rarely go wrong by following Postel's law: "Be conservative in what you do, be liberal in what you accept from others." In other words, ignore any additional fields that are passed along and don't throw errors.

Documenting APIs and their version history is also a good practice. Semantic Versioning (major.minor.patch) is especially good for microservices, as it provides clear and standard communications to distributed consumers of your APIs, and simplifies dependency management. Swagger is a popular tool for designing, building, documenting, and consuming RESTful APIs. It also happens to be used by the Open API Initiative (OAI), which is focused on creating, evolving, and promoting a vendor-neutral API Description Format.

4. Design for efficient services communication

Earlier we discussed external service communication via the API Gateway, which allows one to route traffic from clients to microservices. Large microservices applications have hundreds and thousands of services, and the more services you have, the more communication and data exchanges will take place. As a result, the communication protocol selected becomes an important factor, and can impact performance. It is important to carefully consider the options not only for communication from a client to your service over the internet, but also the options for communication between internal services, to achieve optimal performance for each scenario. While HTTP (soon HTTP/2) is a natural choice for communication from a client to your service over the Internet, TCP/UDP protocols for communication between internal services are ideal for improved performance.

The second aspect that is often overlooked is how data serialization and deserialization can impact the overall performance, and in the worst case become a bottleneck. One way to avoid paying a penalty on data serialization and deserialization, besides choosing a good JSON serializer, is to consider whether you need to re-serialize the object if a downstream service works with the same object. Instead, you can just augment the de-serialized object and pass it on to another service in a form. Another option to improve performance is to use a binary format like Protocol Buffers.

5. Design for proper asynchronous messaging

Since each service instance in a microservices-based application is typically a process, services must interact using an inter-process communication (IPC) mechanism. Asynchronous messaging is a common approach to IPC because a service won't block while it waits for a response from another service. However, there are a few challenges to be aware of:

- **Message order**—Depending upon your messaging solution, the order messages are received by a service may not match the order in which they were sent. If your service requires a specific order to its messages, you will need to apply some logic to your service to address this.

- **Poisoned message**—Messages that are malformed or contain corrupt data can cause the receiving service to throw an exception. A good practice is to discard them by adding them to a specific queue as part of exception handling where they can be retrieved for diagnostic purposes.

- **Repeat messages**—To avoid processing the same request twice, it's critical to ensure that your service can detect and handle duplicate messages resulting from network congestion or retries. More on this in the next best practice, *Design for idempotency.*

Messaging systems such as Apache Kafka or RabbitMQ are readily available to help with these and other challenges associated with asynchronous messaging.

6. Design for idempotency

Whether dealing with messages or data, it is important to always design for idempotency. That is, any message, database write, service to service call, operation, etc., should always produce its intended result even if a bad retry policy, failed receiver or other unexpected incident might cause it to be issued multiple times. For optimal design, the receiver should handle incorrectly repeated messages in the same way the original one was handled. We call this designing for idempotency—that is, the repeated call produces the same result. For example, suppose that when a service seeks to add $100 to an account, the first operation fails due to network problems. We know it is good practice to implement retries, so the sender submits the message again, resulting in two equal messages. If both are processed, the account is credited $200 and not the correct $100. One way to ensure idempotency is to add a unique identifier to messages, and instruct the service to only process the message if the identifiers do not match. One should design operations to be idempotent, so that each step can be repeated without impacting the system.

7. Design for eventual consistency

Distributed computing has long operated by Eric Brewer's CAP theorem, which states: In the presence of a network partition, one must choose between consistency and availability. As most microservices applications are designed to be highly available, strong consistency will take a hit, which can be problematic since a single business transaction can span multiple services touching different data stores. Design patterns such as event sourcing and compensating transactions have proven to be efficient for achieving eventual consistency. Ultimately, the ideal solution will depend heavily on your specific application scenario.

8. Design for operations

Being able to monitor the entire system, including host machines, containers, services, and data stores—and successfully diagnose issues—is critical to operations. The key to this will be your ability to collect and track data through logging. As mentioned, establishing and adhering to a common log format across all services is essential. Opentracing offers a vendor-neutral open standard for distributed tracing that can be used in a polyglot microservices application.

You will also need to consider what to log. What's too much, and what is not enough? Again, the answer depends on your specific application scenario, but here's a good starting point based on many conversations with Oracle customers:

- **Log healthy state:** Logging how your system behaves in a healthy state is necessary to create a baseline with which to compare future anomalies. Data types to consider include service start events, and heartbeat.
- **Log what's not working:** Seems obvious, right? But in addition to error details and stack traces, you should also log semantic information, such as user requests, to help determine why something's not working.
- **Log performance data on critical path:** Aggregating performance metrics as a percentile is a great way to identify outliers and systemwide long-tail performance issues.

Finally, one of the most important best practices is to use a correlation or activity ID. This ID is generated when a request is made to the application and is passed on to all downstream services. This allows you to trace a request from beginning to end, even though it spans multiple independent services.

Distributed tracing solutions such as Zipkin have proven to be very efficient when it comes to monitoring and diagnosing microservices-based applications.

Final Thoughts

Adopting a microservices architecture can help you innovate faster, scale more efficiently, and be the disrupter—not the disrupted—in the digital economy. But it will only work if you're willing to look at your organization's culture, and make the necessary changes to succeed. Service teams need true independence. They must be given the freedom to use the languages, frameworks, products, and release cadences that best meet their unique objectives. Many organizations are simply not set up for this level of distributed autonomy.

The DevOps movement, with its focus on silo busting, shared responsibility, and team autonomy is well suited for microservices architecture. It's no coincidence that the two have grown in popularity along parallel lines, as they both seek to inject agility into the organization.

As you continue your own microservices journey, I encourage you to embrace these principles when designing your cross-functional service teams. If you found this paper useful, please look for a blog post from me on best practices for DevOps, coming Summer 2017, which will help you create a collaborative, trusting environment, free of finger pointing or fear of failure, and full of optimism for the future.