WHITE PAPER

The Upward Spiral Growth of Advanced Architectures: From Microservices to Serverless Computing

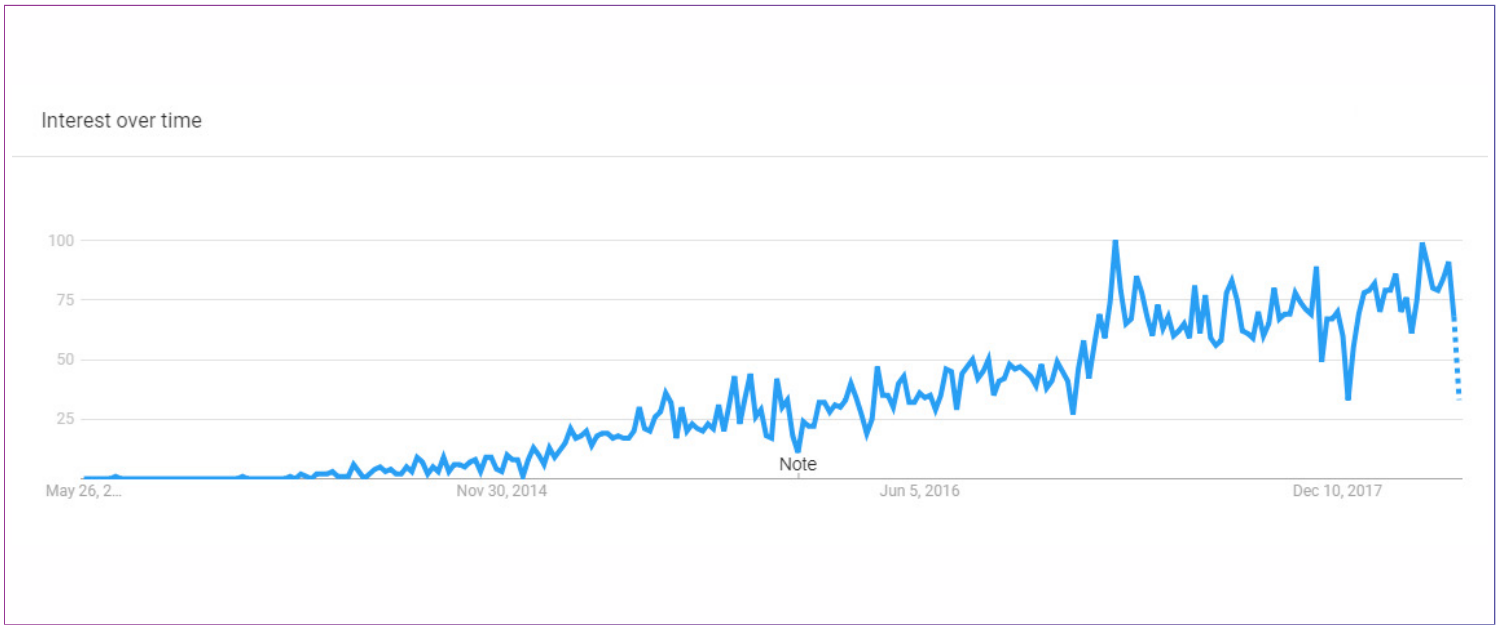
Practice Head
Bala Krishnamurthy
Sr. Manager

Author
Kavin Elango
Research Analyst

aspire SYSTEMS
attention. always.

INTRODUCTION

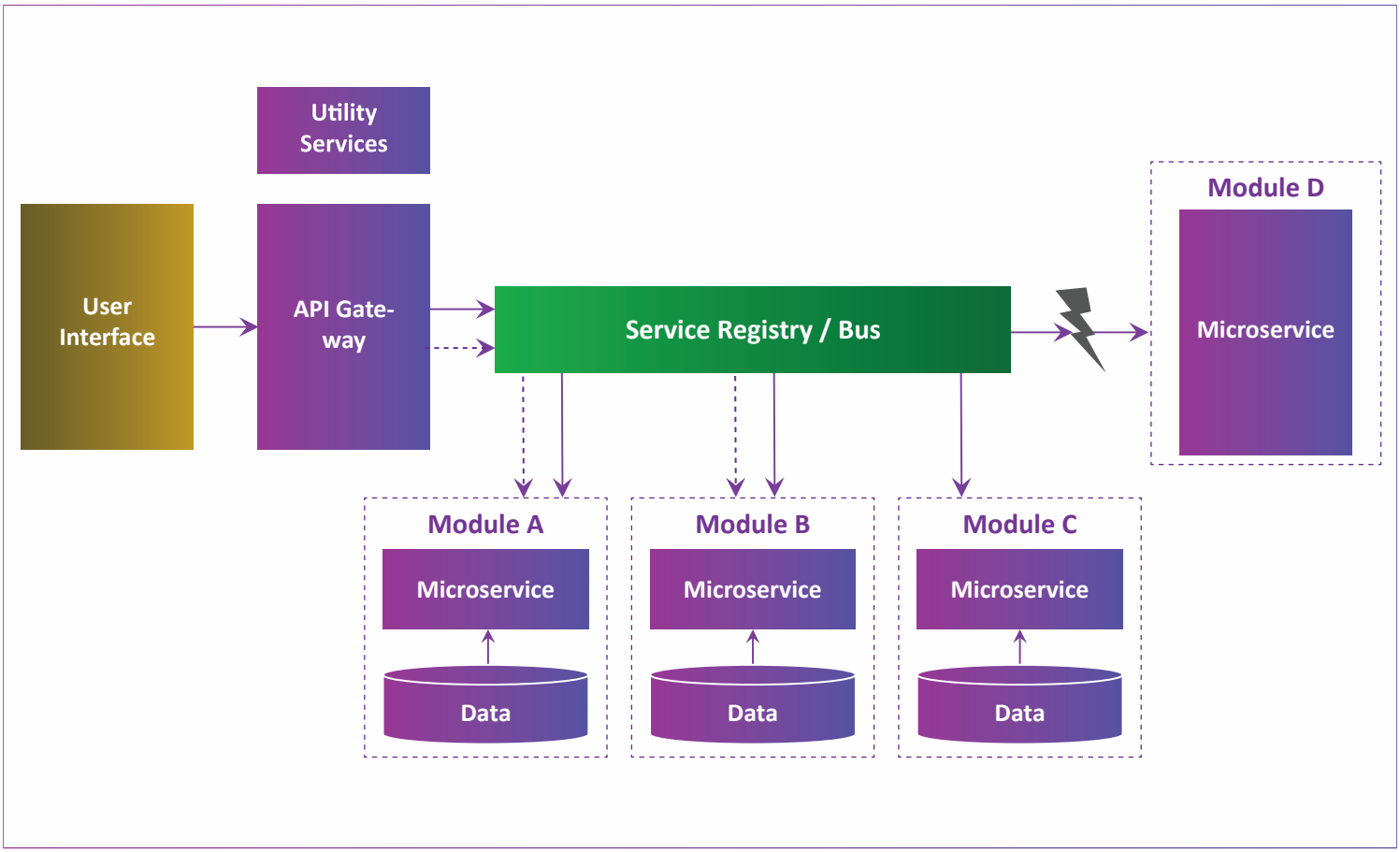
In recent times, the term Microservices has gained immense popularity. It has found advocates in big players like Amazon, Netflix, Facebook, Google, NGINX to name a few. These companies have adopted a common set of DevOps ideologies and in a lot of cases, they've started out as monolithic applications, growing bigger and smaller with a more advanced architectural approach in Microservices. Agile development is built upon the foundation of speed and scalability, to fuel the engine of Continuous Delivery. Microservices serves this need and recent Google trends (as shown in the graph) for the interest levels in Microservices over the past 5 years also corroborate the same. Let's first understand what Microservices really are.



So, what are Microservices? The term "Microservices" was coined to define an architectural style which proposes developing and maintaining an application as multiple small independent deployable units, specifically as services, that communicate via APIs.

There are no hard and fast rules as to how many individual services your application should be made of or how the breakdown should happen. Neither is there a specification as to the size of each unit. The only criterion is that each service should handle one responsibility and do it well. This is in unison with the core philosophy of UNIX "Make each program do one thing well".

The following diagram illustrates the skeleton of a basic Microservices architecture.



RECENT TRENDS IN MICROSERVICES

Incorporation of Microservices in already built applications

The usage of Microservices, in the past, was considered limited to only the to-be-developed applications. However, the advent of Microservices in the recent past has been so great and capable that organizations across the globe have started to implement Microservices in order to re-build their existing applications. According to the Red Hat 2017 Microservices Survey, "According to 67% Middleware customers and 79% Openshift customers, Microservices are being used to re-architect existing applications as much as for brand new projects". This re-iterates the fact that the organizations have started to implement Microservices more extensively than before.

Legacy Systems Modernization

Modernization of legacy systems involves updating the obsolete software functionalities and strengthening its potential to cope up with the modern, agile IT business requirements. It can give the competitive edge needed for sustenance by providing decreased complexity, increased efficiency and compliance and decreased costs. Microservices have in them a variety of features that make them a widely preferred tool for modernization. Not only do Microservices enable independent scaling of elements, but they also provide flexibility to choose multiple languages for different services. Besides supporting phased implementation, it is also feasible with independent service deployment. Owing to the aforementioned factors, Microservices are considered as an effective tool for modernization of legacy systems.

Event-Driven Architecture (EDA)

An upsurge of event-based architecture is one of the major recent trends in Microservices. A "push" is a terminology in EDA wherein the observer container waits for a message sent by a service. In this type of architecture, the central server of the system commences the request of a transaction as opposed to "pull" architecture in which the client initiates the transaction request. The key benefit of EDA is that is particularly well-suited for loosely coupled structures as the components can be used repeatedly by various networks. This also eases the workload of the developers since it gives them a liberty in designing their services.

Migration from REST to GraphQL

GraphQL, an open-source ecosystem, is a data query language for APIs. GraphQL basically acts as a medium between the server and client and aids in their intercommunication. The issues of Over-fetching and Under-fetching that persisted in REST can be solved by GraphQL as the latter sends only a single query with all compact data requirements to the GraphQL server. As the request is raised only to a single endpoint, GraphQL enables separation of data owners.

BENEFITS OF MICROSERVICES

Scalability

Selective scalability is one of the inherent advantages of the Microservices architecture. As mentioned earlier, there might be certain operations that can spike the load of the system and make your entire product slow or sometimes even make it non-responsive. If you were to pull it out as a separate service that is hosted in a dedicated server, rest of the product doesn't have to bear the brunt of the super intensive consumption of resources by this operation.

Maintainability

Modularity is easier to achieve when you build your product as a suite of smaller units or services each designated for a clearly defined responsibility. Each service can be guided by a set of principles best suited to the kind of problem it is meant to address. Over a period of time, even if a couple of services cease to be relevant or need a complete replacement, the rework would be restricted to those services alone which would be lot cheaper, less risky and less time consuming than a monolith.

Availability

The fault isolation that is achieved by implementing the modules as services guarantees high availability. Also, the flexibility to scale up your services selectively on demand is an added advantage.

Technology Upgrade

Microservices provide greater flexibility in terms of technological choices than a typical monolith. Based on the type of service you are building, you can choose a technology stack that best suits its needs. Technology migration can be achieved incrementally as well.

Rapid Application Development

Business demands sometime require a product to be built and available in the market in quick turnaround time. The increasing nature of such demands is what gave birth to frameworks like Ruby on Rails which aid in Rapid application development. The usage of Microservices gives an extra edge in this aspect. You can develop and release a subset of features developed as services. One might argue that this process known as iterative/agile model can be and have been followed for products built as a Monolith as well. True. However, Microservices takes this one step further by providing the ability to add or remove a feature or a module without actually disturbing the rest of the application and that too at the necessary "Rapid" pace. In short, the whole process becomes more streamlined and effective.

MICROSERVICES - IMPLEMENTATION STRATEGIES

Memory Segregation

In order to avoid over-allocations and heavy memory consumptions, the optimum size of the independent Microservices should be ensured. Since the independent services are segregated, each service requires an additional spacing to be allocated in its memory. However there is a challenge that lies ahead in this process: If it is not done methodically, it might end up over-allocating the memory than a standard Monolithic application.

Network Usage

Owing to the architectural design of the independent Microservices, there is a lot of network traffic that actually goes through between the services. However, it is a common typical challenge since every network communication has a latency attached to it. This issue can be avoided by ensuring that the data exchanges are faster and co-located within the same service.

Data Management

With an implementation pattern where the completion of one request may span multiple service calls and each of them issuing its own transaction, comes the inherent need to implement distributed transactions. Distributed transactions are not easy to implement; they are costly and can cause a lot of trouble if not implemented right.

The preferred option to handle this problem is an approach that is called "Eventual Consistency". Eventual Consistency is defined as " a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value". For example, the user action may require two requests to be completed by two different services. The action isn't complete until both the requests are processed. Say that the first request is completed and the second request is queued. So till the second request gets completed, the end user may be seeing some incomplete, incorrect data. This is allowed with the expectation that it will get resolved "eventually".

Security

In the traditional approach, when a service is defined, there is a security configuration for that particular service. When the element of security is added, it adds up to the processing overhead. So every request takes an additional few milliseconds to perform the security validation. When it comes to Microservices, there is a significant amount of process that happens with multiple services. Each service adds some overhead to it; so the cumulative overhead will quantify to a sizeable number. Hence, for an architect, it is crucial to define and establish what level of security needs to be applied to which particular service and if that particular service should be exposed outside or not.

Monitoring Mechanisms

Any application that involves interactions through the network should be prepared for the probable failure scenarios in one or more services.

1. If there is an integration service involved. In this case, there should be a disaster monitoring mechanism that has the potential to differentiate between the functional errors and the system/ process errors.

2. If the integration service fails, everything is brought to a grinding halt. Hence Microservices place a lot of importance in ensuring that the system is equipped to handle and recover from such cases. Building monitoring mechanisms is recommended so that failures can be identified and addressed as soon as they occur.

Asynchronous Communication

In Synchronous Communication, it does not make much difference between the usage of a Microservice or a standard service, because regardless of what is being used, it would have the same impact from end-user's perspective. As this is not the case with the implementation of Asynchronous Communication, it trumps the former. For instance, if the third party system involved becomes unavailable for sometime and consequently this leads to the failure of other applications due to the lack of inter-communication, the system can actually re-try after a specific pre-determined time period and ensure recovery from downtime. So, asynchronous communication should be preferred to synchronous communication during the adoption of Microservices. The traditional asynchronous communication systems require manual orchestration, but when Microservices are used, self-orchestration is achieved and the process becomes autonomous in nature.

Versioning vs Tolerant Services

One of the most critical aspects while developing services is the contract that you establish and share with the consumers of the service. How do you ensure that the change to the service contract or the implementation does not necessitate a change in all the consumers?

For example, consider that you enhance a particular service to meet the demands of one particular consumer. Will you request all the other consumers to update their implementation even though they derive no discernible use from the change? You have to unless you use one of the two options - Versioning or Tolerant services. Versioning involves retaining the old version(s) while you roll out the updated versions and have the consumers use a version of the service based on their need. This of course adds an additional overhead of deploying and maintaining multiple versions of the same service. The preferred approach is to develop your services as what is termed as "Tolerant" way. In this approach, the schema is designed with minimal restrictions and validations. The consumers are meant to extract data that is required and ignore what is not.

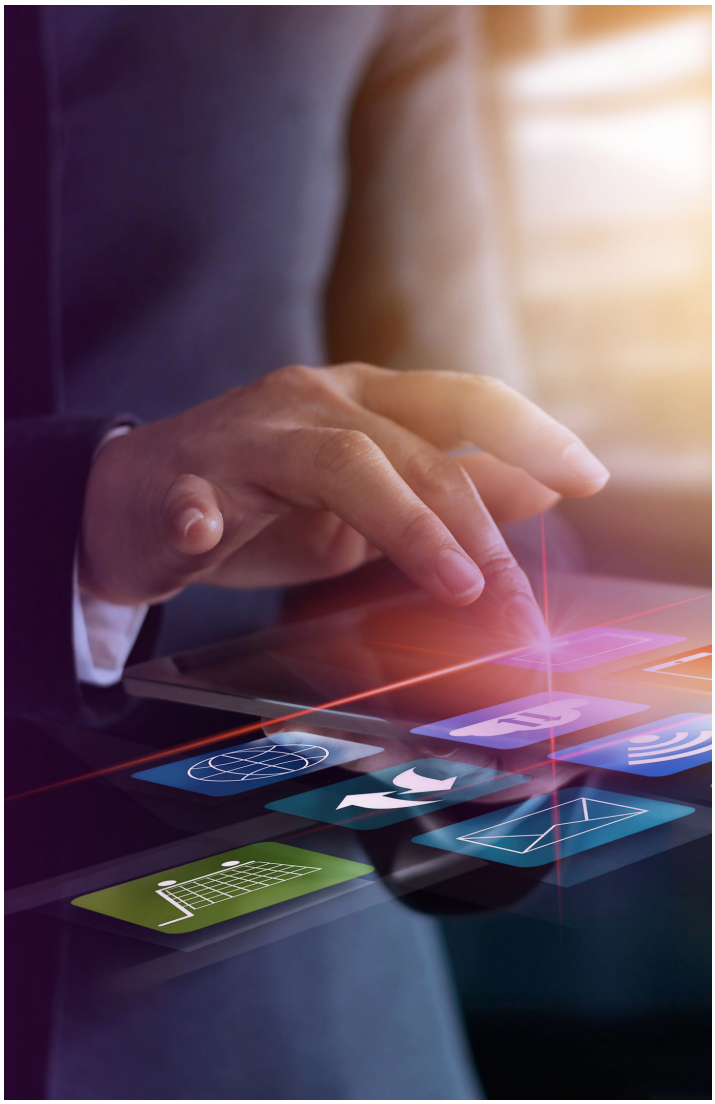
DevOps

Successful execution of Microservices requires the entire build and deployment to be streamlined and a good collaboration be established between the development and the operations team. The build and deployment processes should be automated using Continuous Integration and Deployment tools. Using the likes of Amazon Web Services, the infrastructure management can be simplified as well. The more the automated deployment and server processing tools are involved with the development team, the more the benefits.

Testing

Any distributed system demands more rigorous testing as the collaboration among multiple components opens the possibility for more failures. Microservices is no exception to this.

Unit Testing - most important, yet most neglected aspect of software development cycle. It is imperative that unit tests are implemented and their execution is part of your continuous integration strategy. Over and above this, contract testing and integration testing need to be performed. Contract testing is done to verify if the actual direct and indirect contract of a Microservice is in adherence with its intended functions. It can either be consumer-driven in which the end-users of the Microservice are dealt with or provider-driven wherein the emphasis is on the entity that provides the service. Integration testing is adopted when the multiple individual Microservices can be consolidated before deployment. The purpose of Integration testing is to ensure that the system works together flawlessly and that there is reliable inter-dependency among the services. However, a solution will require an additional overhead of ensuring the entire mesh is tested rather than just the joints.



Progressive Advancement of Microservices – The Serverless Architecture

Serverless Architecture is a software paradigm wherein a third-party service hosts the applications. Eventhough the applications still run on servers, the workload of the developers is significantly reduced as they typically don't manage the servers. This consequently narrows the areas of focus of developers and help them concentrate only on the individual functions in the given application code. There are 2 broad adaptations of Serverless Architecture: Function as a Service (FaaS) and Backend as a Service (BaaS).

Function as a Service (FaaS)

Incorporation of FaaS into applications is considered as the nucleus of Serverless Architecture. In FaaS, the application is basically constituted by several automated, individual functions. The FaaS provider hosts each of these functions and the scaling is done automatically based on the frequency of the function call. The 3 vital elements of FaaS are Functions, Events and Resources. The main benefits of FaaS are faster development and deployment times, high level of scalability and cost-efficiency.

Backend as a Service (BaaS)

BaaS is a service model that basically acts as a middleware for App developers for linking their products with backend cloud storage and processing. BaaS uses tools like Software Developer Kit (SDK) and Application Interface (API) for linking the apps to backend services. BaaS typically provides a vast range of services like search functionality, social integration, mobile application management etc. Some of the key benefits of BaaS are un-necessary stack development stoppage, multi-usability, flexibility, etc.

CONCLUSION

The evolution in the sphere of advanced architecture from Monoliths to Serverless Computing has been remarkable. With the emergence of Serverless Architecture, the long-time dream of developers not having to worry about server management has been achieved to an extent. In this age of agile computing, Microservices and Serverless Computing should make IT decision-makers create roadmaps for the same in the near future. Perhaps, in the future, contriving solutions to currently faced key challenges in implementing Microservices like integration management, high initial investment of time and money, complexity in finding the root cause of problems etc. would further increase the rate of adoption and implementation.

www.aspiresys.com

aspire SYSTEMS
attention. always.

ABOUT ASPIRE

- Global technology services firm with core DNA of Software Engineering
- Specific areas of expertise around Software Engineering, Digital Services, Testing and Infrastructure & Application Support
- Vertical focus among Independent Software Vendors and Retail, Distribution & Consumer Products
- 2400+ employees; 150+ active customers
- CMMI Maturity Level 3, ISO 9001:2015, ISO 270001: 2013 certified
- Aspire has extended its global presence across North America, Europe, APAC, and the Middle East
- Recognized 9 consecutive times as "Best Place to Work for" by GPW Institute

SINGAPORE +65 3163 3050 | NORTH AMERICA +1 630 368 0970 | EUROPE +44 203 170 6115 | INDIA +65 3163 3050 | MIDDLE EAST +971 50 658 8831

For more info contact
info@aspiresys.com or visit www.aspiresys.com