MICROSERVICES:

The software development approach with macro benefits potential

CAPCO

THE APPEAL OF THE MICROSERVICES APPROACH IS THAT IT ALLOWS THE ENTERPRISE TO PROTECT AND EVOLVE IMPORTANT EXISTING CAPABILITIES, WHILE ISOLATING AND RAPIDLY REPLACING OTHER, LESS DESIRABLE, LEGACY COMPONENTS.

MICROSERVICES AT A GLANCE

Microservices are making waves in service oriented architecture. If you are experiencing ever-increasing pressures to deliver agility and scalability from your IT estate, microservices are certainly worth learning more about. Their promise is real and the disciplines required to implement them effectively are achievable. The key to success is an informed approach, underpinned by key-decision quality at the outset and effective ongoing organizational and technology support.

CONTEXT

Ever-increasing demands for enablement of business agility are leading to software development expectations that are both evolutionary and revolutionary. Large enterprises are typically conservative, requiring technology continuity and a stable base. They tend to gravitate to the more evolutionary approach of adapt and change (natural selection) as opposed to the revolutionary mode of destroy and rebuild. The appeal of the microservices approach is that it allows the enterprise to protect and evolve important existing capabilities, while isolating and rapidly replacing other, less desirable, legacy components.

DEFINITIONS

A microservices-based architecture is a specific type of service oriented architecture (SOA) that addresses many business agility demand issues through modern-day solutions. It is true that SOA itself is a very broad term, covering many architecture styles. Microservices take a more targeted approach. This allows for greater flexibility in defining, evolving and deploying architectural components in the solution domain, and, as a result, helping achieve greater organizational agility.

The microservices architecture structures a software application as a suite of small services, each running its own process and communicating via lightweight mechanisms. These services – microservices – are built around business capabilities and are independently deployable via fully automated means[1].

SCALABILITY AND AGILITY ARE THE REWARDS

Defining the size and scope of individual microservices is a highly flexible exercise. For the purpose of this summary, our focus is on considering an architectural approach that has the potential to significantly increase scalability and agility. We will explore the detailed mechanisms later in this paper.

SCALABILITY

By running software with a microservice as the basic unit of deployment, and by having the necessary ecosystem ('scaffolding') to deploy units quickly and in an automated fashion, we can scale software solutions by deploying only those microservices that need to meet high volume or highly resource-intensive user loads.

AGILITY

In terms of agility, having microservices that can be deployed and configured to work together in many ways – even in previously unplanned ways – provides the software architect with a truly flexible toolset for engineering solutions that meet business user needs. (The term 'business user' is applied here in its widest possible sense. A business user could be a back-office employee, an internal knowledge worker, an external-facing customer, or an external system.)

DEMAND FOR MATURITY

With this approach, however, comes an increase in deployment complexity and a change in the workings of architectural governance. This in turn requires a higher degree of maturity than is typically found in many organizations.

A CLOSER LOOK

In the following sections we address the key drivers for adopting microservices. We also look at architecture and design attributes typical in a microservices approach. We address data and how it is managed. And we discuss – from a people and working styles perspective – the impact a microservices architecture approach has on an organization.

A SMOOTH TRANSITION TO MICROSERVICES
FOR GREATER AGILITY AND SCALABILITY

TECHNOLOGY AND BUSINESS DRIVERS

There are many technology and business drivers that support transition to a microservices-centric architectural approach. These include:

- The move to rich, dynamic, and highly interactive user experiences, across multiple platforms and form factors;
- The rising trend of a polyglot approach, with people building services in their language of choice and best fit, rather than standardizing on a single language;
- Flexible deployment options (public and/or private cloud, etc.);
- Independent component lifecycles;
- Frequent deployment of components, as often as multiple times each day;
- Improved ability to build new applications, based on compositions of new and existing microservices.

Each of these drivers reflects a specific demand to which software architecture agility can directly respond, ultimately leading to an overall increase in business agility.

ARCHITECTURAL AND DESIGN ATTRIBUTES

The architectural and design attributes of a microservices architecture resemble those of other architectural approaches that, as experience has shown, lead to the development of robust systems. They include:

- Low coupling between components;
- High cohesion within components;
- High levels of scalability achieved with little effort;
- Fault tolerance of components and interfaces.

At this point, we would refer readers to the so-called **SOLID** principles[2]. In the SOLID acronym, the letters denote the following principles:

S = SRP – single responsibility principle
O = OCP – open/closed principle
L = LSP – Liskov substitution principle
I = ISP – interface segregation principle
D = DIP – dependency inversion principle

The SOLID principles are applied most often to object-oriented systems design. With regard to microservices, several of them are very much worth adhering to strictly, in order to create a workable microservices architecture. Below, we explain the key principles in more detail.

SRP

The single responsibility principle, in microservices terms, means there should only be one reason for the service to exist, and a razor-thin reason for change (this aligns closely to the Unix philosophy of "do one thing well rather than multiple things poorly").

ISP

The interface segregation principle means that clients of a service should only depend upon the contracts they explicitly use (and not have any underlying dependencies on the services they use in turn), and they should strive to maintain as few service dependencies as possible.

DIP

The dependency inversion principle, applied to microservices, means that services should define and adhere to abstractions, as manifested in their interface contracts. These abstractions should not depend on details; rather, the details should depend upon abstractions. DIP is close to the idea of inversion of control (IoC) – allowing an external mechanism (e.g. Spring) to resolve run-time dependencies between microservices, rather than building the run-time dependencies into the components.

A SMOOTH TRANSITION TO MICROSERVICES FOR GREATER AGILITY AND SCALABILITY CONTINUED

In addition, and in its ideal form, the microservices architecture approach exhibits the following attributes:

- Intelligence in the endpoint-communications mechanisms is, by necessity, very simple;
- Typical use of two simple communications interaction styles are:
  - Synchronous call support (e.g., HTTP/REST),
  - Lightweight messaging support (e.g., AMQP or similar);
- Components are independently deployable;
- Explicitly defined and documented component interfaces are deployed;
- A 'shared-nothing' architectural approach is taken, whereby components do not share any technology dependencies (memory, data persistence, etc.) with any other component;
- There is increased reliance on, and usage of, build/test/deploy cycle automation and sophisticated infrastructure management techniques (continuous build and integration, devops, etc). In other words, you need to set up automation to unlock the full value of a microservices approach;
- Each component can be (and often is) implemented as an appropriately-scoped subset of the IT organization's technical architecture – one appropriate for the business requirements of the component.

ENTERPRISE ATTRIBUTES

Enterprises already committed to – or beginning a journey towards – a microservices-based architecture most likely intend to build large numbers of microservices, over time. Some of the architectural and design attributes discussed above will come to the forefront, when the building of a large suite of microservices is being considered. In addition, the following enterprise attributes are relevant.

Simple communication interaction styles

Common, lower-level application-layer communication APIs that abstract and homogenize the sending of data over the wire between services must be adopted. The microservices that form an application (or application suite) should all use this common set of communication APIs, in order to ensure interoperability and pluggability. The good news is that such APIs do exist. The less good news is that motivating an organization to adopt them universally is a challenge.

Cohesion and shared-nothing promote testability

We need the ability to test our components in isolation. If we can accomplish this, we can also carry out independent deployments. For microservices, we must have a flexible (ISP + DIP) mechanism that allows us to perform service testing. We can use the mock objects pattern to assist but, for microservices, we have to push this to the extreme. In general, at an enterprise level, we have to build scaffolding – again, think of common APIs and tool frameworks that enable isolated testing of components.

Defining microservice boundaries

While adhering to the single responsibility principle comes first and foremost as a consideration, microservice boundaries should be aligned with the business value chain. The concept of a business function within the value chain is an excellent way to think about how services are partitioned but, as a concept, it is still just a starting point. In practice, some microservices may be 'smaller' than a specific business function within the business value chain. Others may provide lower level, cross-business-functionality. Does aligning with business value chains in fact detract from business agility? Not really. While some enterprises may change their fundamental service offerings, and expand or contract their business value chains, our experience suggests that the value chains themselves are quite stable. We address this issue further in the section on key ingredients for success, since these necessarily influence how (micro-)services are partitioned.

Coupling. Solving the problem of 'chaining' microservices

Often, in order to perform a useful business task, it will be necessary to 'chain' (or compose) a set of microservices together.

Consider a service that allows a retail banking customer to transfer funds from one bank account to another entity (for example another account at the same bank, or to an account at another institution). One of the prerequisites for the transfer is that the source account has sufficient funds. In this scenario, at least two services are in play: (1) a service which returns and/or checks the current balance of the source account; and (2) a service which performs the actual transfer of funds from the source account to the target account.

Focusing on the 'TransferFunds' service, we can design it in at least two different ways:

1. TransferFunds takes as input the transfer amount, checks the validity of this amount (via the 'AvailableBalance' service), performs the transfer function (possibly by invoking other services), and returns a result.
2. TransferFunds takes as input both the transfer amount and the available balance, checks the validity of the transfer amount, performs the transfer function (possibly by invoking other services), and returns a result.

The point here is that there is either an explicit or implicit dependency between TransferFunds and AvailableBalance. In design alternative 2, the dependency is implicit; we would need a higher-level service to invoke both AvailableBalance and then TransferFunds. We have essentially shifted the interdependency to a higher-level service. Then, in either of these design scenarios, we have to wire some services together – the endpoints (addresses) have to be known at run-time. We can apply some sound OO (object-oriented) and EAI (enterprise application integration) patterns, such as adapter and factory, to aid us in following the dependency inversion principle. Ultimately though, we need some run-time configuration information to enable one service to invoke another.

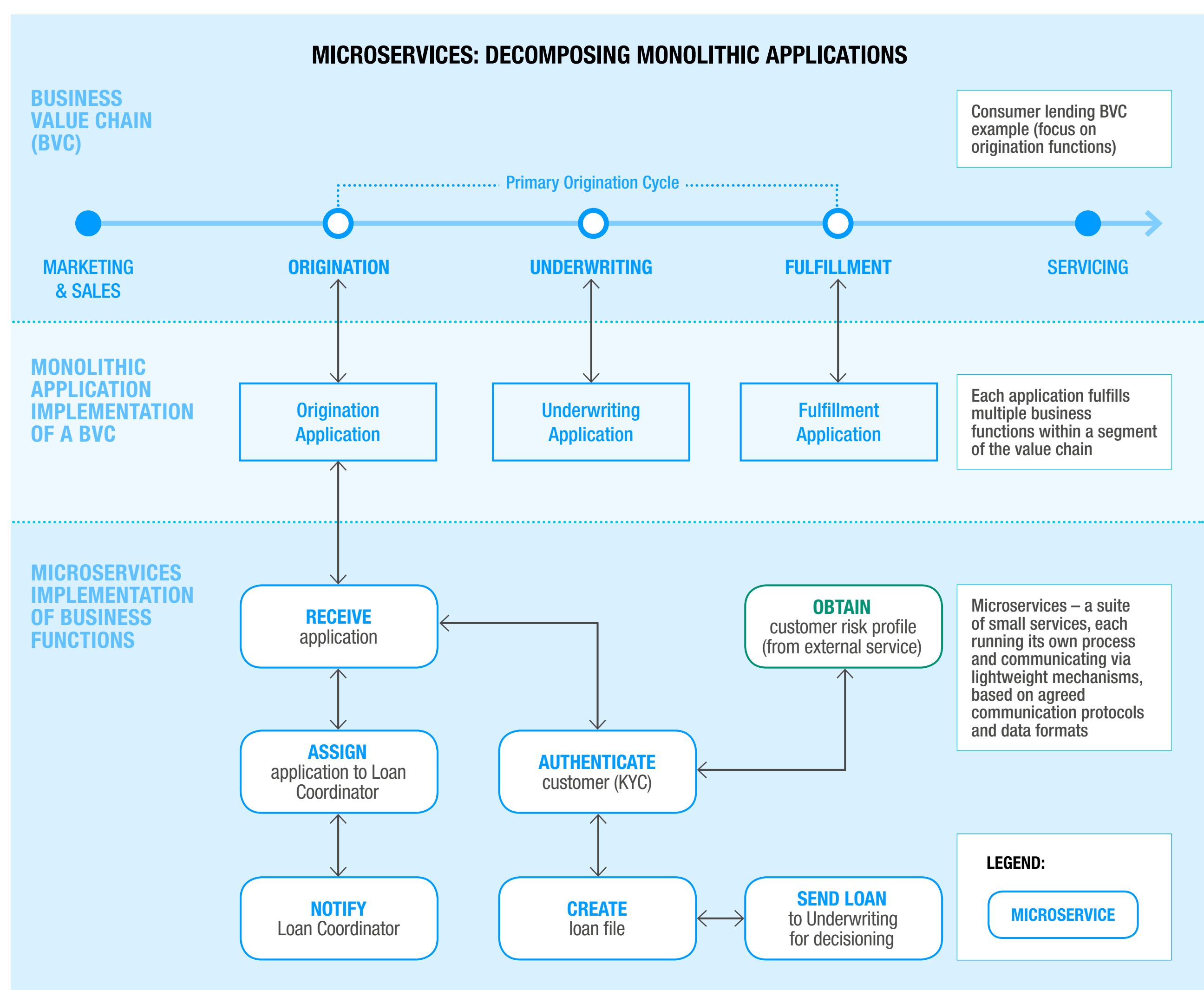
A SMOOTH TRANSITION TO MICROSERVICES FOR GREATER AGILITY AND SCALABILITY CONTINUED

The following best practices in run-time configuration management should be applied:

Implement a configuration management facility that deploys configuration information to all services in the infrastructure, so that whenever a microservice is (re-)deployed, its endpoint dependencies are deployed with it. There are good tools for achieving this, such as Puppet, Chef, and Ansible. If all endpoints (e.g. REST endpoints) can be placed behind a network load balancer, or other HA (high availability) configuration (e.g. JMS clustering), then this simple approach can work very well.

Set up a configuration service Off-the-shelf tools, such as Netflix's Eureka Server, are available to facilitate this approach. As individual service instances start up, they register themselves with the configuration service. Each service is a client of the configuration (location) service, asking for an endpoint address of any service that needs to be invoked. Replicating configuration service nodes across physical data centers (and using tools such as Spring Boot to configure business services to abstract details of the location services), provides a vast amount of scalability, while keeping service coupling to a minimum.

It must be noted that these approaches to sharing configuration and endpoint information across services do introduce some coupling between the microservices in architecture. In reality, 'shared-nothing' is more aspiration than achievable goal, once an enterprise starts creating entire application suites composed of microservices. Also, minimizing run-time endpoint dependencies does require more organizational effort – both in terms of time and money – as well as greater discipline.



MICROSERVICES INTEGRATION CONSIDERATIONS

MICROSERVICES ORCHESTRATION

Some organizations find that, over time, microservices implementing business processes have been orchestrated in an ad hoc manner, using a combination of pub/sub, making direct REST calls, and using a database to manage the state. Without a central orchestrator, as the number of microservices grows and process complexity increases, gaining visibility into these distributed workflows becomes difficult. Consequently, we would consider central orchestration as a best practice, but only for organizations that have scaled to a level of microservices adoption that may be experiencing the issues noted above.

One approach to orchestration is to use an enterprise service bus (ESB). The choice of ESB platforms should be made thoughtfully, with due regard paid to microservices fit and scale. Any platform that cannot support hundreds – or thousands – of independently-deployed microservices should be avoided.

Kai Waehner, technology evangelist for TIBCO, described use of the company's ActiveMatrix BusinessWorks as a wholly suitable platform for hosting microservices, as a result of its combination of scalable runtime and low hardware footprint[3]. Use of such a platform does carry some orchestration advantages, due to sophisticated tooling capabilities. Our advice here is that organizations which have already made investments in platforms such as TIBCO BusinessWorks should seriously consider their suitability for microservices.

Another approach is to use a relatively lightweight, general-purpose service orchestration framework. Apache Camel is a notable example implementing many of the enterprise integration patterns[4] and a few of these patterns are key to orchestrating microservices.

Yet another approach involves a custom orchestration framework. Prominent among these is Netflix Conductor[5] built as an 'orchestration engine' to address the following requirements:

- Blueprint base (a JSON DSL based blueprint which defines the execution flow);
- Tracking and management of workflows capability;
- Ability to pause, resume and restart processes;
- Visualization of process flows through a user interface;
- Ability to synchronously process all tasks when needed;
- Ability to scale to millions of concurrently running process flows;
- Back-up by a queuing service abstracted from the clients;
- Ability to operate over HTTP or other transports e.g. gRPC.

CANONICAL DATA MODELS

The term 'canonical data model' can conjure images of substantial data architecture effort to develop a comprehensive, governed data model for a large enterprise. Paradoxically, this would appear directly at odds with the development of a suite of independent, lightweight microservices. And indeed, if this traditional, resource-intensive image of a canonical data model were the only way to share a common vocabulary of data entities across microservices, it would be impractical.

Fortunately, industry efforts such as the schemas from Schema.org have recently made it much easier to posit a common vocabulary of data entities. These are in fact a set of 'types', each associated with a set of properties. The 'types' are arranged in a hierarchy. The core vocabulary at the time of writing consisted of 589 types ('classes'), 860 properties, and 114 enumeration values. While the current set of types in Schema.org is quite rich, a schema that attempts to name all objects, interactions, and concepts will not always serve all applications or domains equally well. In some cases, the organization's microservices architecture can leverage the available types and extend them in a private (organization-centric) way. In other situations, it may be worthwhile to submit extensions to the community process and make them a permanent part of the standard.

There is also an argument for avoiding canonical data models, which can be a valid tactical approach for microservices architectures and for enterprise architectures[6].

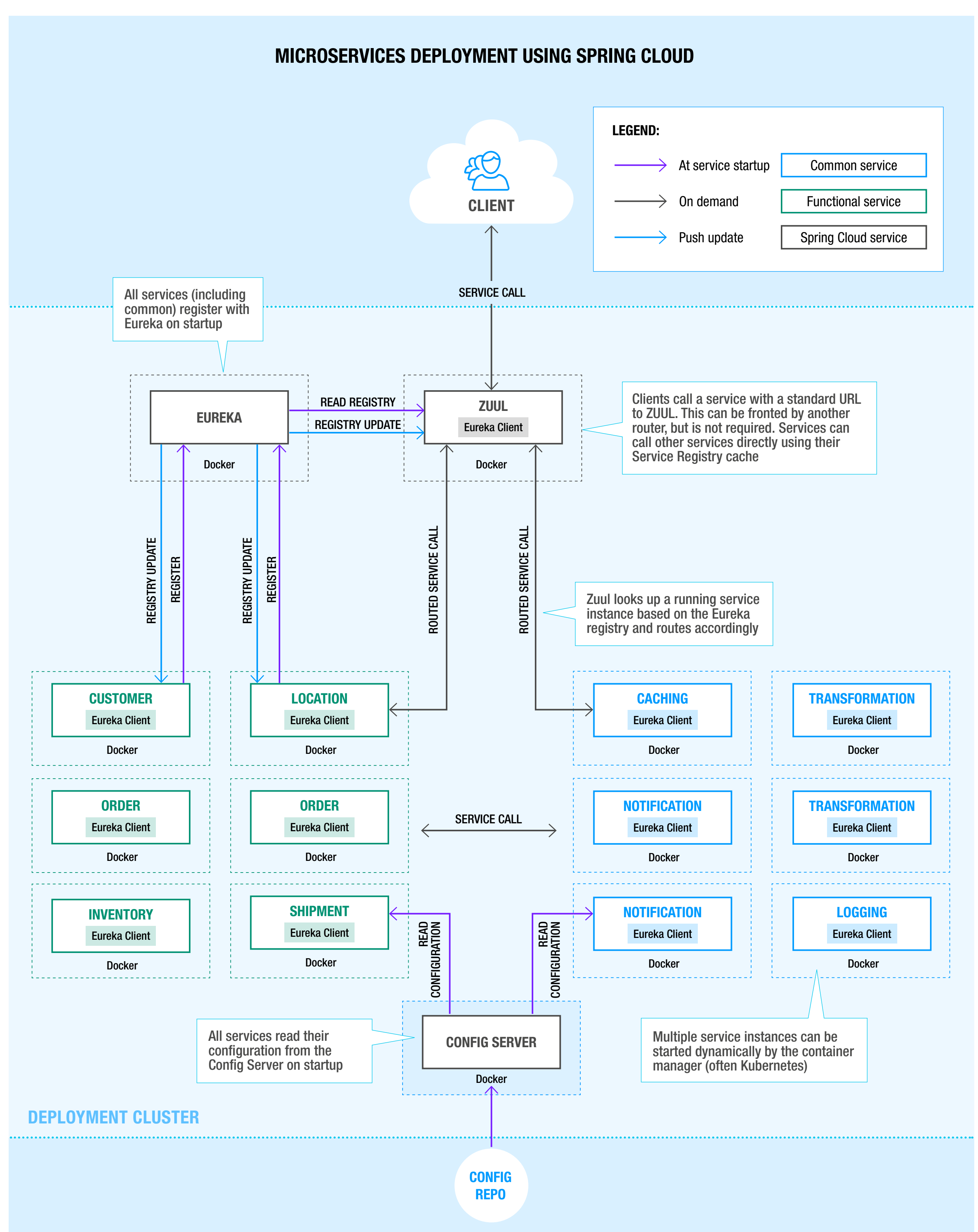
OTHER DATA CONSIDERATIONS

Data persistence and management are specific to each component. There are several approaches to practical implementation, but coupling at the data layer is a microservice antipattern. We recommend, during an architectural transition to microservices, to 'wrap' the data access and persistence interactions in an API that looks like separate, service-specific data setup. Once each data interaction is isolated behind this API layer, the data-stores themselves can be separated over time. (We anticipate however, that this final step will be lower priority than many other initiatives. In practice, 'technical debt' will likely be carried forward for a long time.)

It should be noted, that this transition state still leaves some level of coupling between microservices that should be entirely independent[7].

We also recognize the need to move traditional database layer responsibilities to the service implementation layer, for activities such as enforcing referential integrity – something that will require information architects to rethink how they structure and govern data in an enterprise.

MICROSERVICES INTEGRATION CONSIDERATIONS CONTINUED



KEY INGREDIENTS FOR MICROSERVICES SUCCESS

DEVELOPMENT TEAM STRUCTURE, FUNDING, AND GOVERNANCE

For an organization determined to successfully implement a microservices architecture, the following features are important.

- Teams are cross-functional within a suite of services and organized as deployable units (e.g. designers, developers, testers, and others are all part of the same team).
- There is a good fit between a microservices approach and Agile methodologies.
- Deployment units are treated more as products than projects (while the development team also provides production support and tends to stay with the product over its lifespan).
- Funding models are flexible and, rather than fund a large project with a fixed end-date, a team is funded with a run-rate cost commensurate with the current product backlog, business priorities, and desired pace of change.
- There is recognition that centralized governance is not as valuable in the world of microservices as it tends to drive monolithic solutions and standardized tech stacks. In fact, governance should be focused more pragmatically on such issues as service contracts and quality metrics, while leaving specific technology choices to the (largely autonomous) component teams.

The above features, particularly the last point, suggest that the role of enterprise architecture (EA) should evolve in organizations that have commenced a microservices architecture journey.

MICROSERVICES READINESS CHECKLIST

Be under no illusions: a microservices architecture does not remove the need for good architecture practices and discipline. On the contrary, these strengths are more important than ever. Before embarking on any journey to adoption of the microservices approach, organizations must assess where they stand today, in terms of the drivers and disciplines that will determine just how well this approach will work for them tomorrow. In key areas of the business, a number of important questions need to be raised and answered honestly.

☑ Business drivers' status

Is there sufficient appetite inside your business for a truly responsive software development agenda? Do scalability and agility really matter in business terms? Are their benefits understood, in the context of commercial competitiveness enablement and at senior level?

☑ Architecture discipline

Do you have the architecture discipline needed to sustain and grow a microservices ecosystem? (You must give the results of any assessment very careful consideration. Avoid at all costs just plunging into the next generation of service oriented architecture, without a clear understanding of current strengths and weaknesses.)

☑ Understanding of current approach

Do you have an accurate picture of the way you do things today? Your existing architecture principles and governance structures will need to be examined and modified, to align with microservices best practices. And many of the ways your people work will need to change.

☑ Data management policy

Do you have the flexibility and maturity in data governance? You will need to know about and be willing to accommodate revisions to data management policies and a move away from a centrally defined tech stack.

☑ Relevant expertise

Do you have the in-house technology capabilities to transform the theory of a microservices architecture into a working reality? Access to appropriate and proven external support can prove invaluable in the pursuit of successful delivery.

CONCLUSION

MICROSERVICES ARE NOT THE PANACEA, BUT THEIR VAST POTENTIAL CANNOT BE IGNORED

For a start, the microservices approach will not, on its own, fix a 'broken' organizational software delivery process. Yet the real benefits it brings – of scalability and flexibility – are eminently achievable. And, as organizations make the transition to a cloud-based infrastructure and the internet of things (IoT), structuring their application architecture using a microservices approach will indeed be a natural and logical fit.

MICROSERVICES, MACRO BENEFITS

For all its inherent challenges, including the churn and chaos that will be inevitable at the beginning of the journey, the benefits to be enjoyed at the destination will make the effort worthwhile.

Microservices are demonstrably capable of delivering macro benefits. Alone, they cannot make organizations work. With the correct understanding, expectations and disciplines in place however, organizations can most assuredly make microservices work for them, reaping the rewards of agility and scalability, and ultimately, profit and growth.

REFERENCES

1 http://martinfowler.com/articles/microservices.html
2 http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
3 http://www.kai-waehner.de/blog/2016/07/08/tibco-activematrix-businessworks-6-leading-integration-service-delivery-platform/].
4 http://www.enterpriseintegrationpatterns.com/index.html
5 http://techblog.netflix.com/2016/12/netflix-conductor-microservices.html
6 https://www.innoq.com/en/blog/thoughts-on-a-canonical-data-model/
7 Chris Richardson (creator of CloudFoundry and author) addresses both the shared database antipattern (http://microservices.io/patterns/data/shared-database.html) and the database per service pattern (http://microservices.io/patterns/data/database-per-service.html) to show the inherent difference. Another credible source is Arun Gupta's blog http://blog.arungupta.me/microservice-design-patterns/

AUTHORS:

Darrell Rials, Principal Consultant
Mick Smothers, Principal Architect

CONTRIBUTORS:

Luke Penca, Principal Consultant
Poorna Bhimavarapu, Managing Principal
Pete Grebus, Principal Consultant
Rajendra Konduru, Managing Principal
Matthew Markham, Partner

ABOUT CAPCO

Capco is a global technology and management consultancy dedicated to the financial services industry. Our professionals combine innovative thinking with unrivalled industry knowledge to offer our clients consulting expertise, complex technology and package integration, transformation delivery, and managed services, to move their organizations forward. Through our collaborative and efficient approach, we help our clients successfully innovate, increase revenue, manage risk and regulatory change, reduce costs, and enhance controls. We specialize primarily in banking, capital markets, wealth and investment management, and finance, risk & compliance. We also have an energy consulting practice. We serve our clients from offices in leading financial centers across the Americas, Europe, and Asia Pacific.

To learn more, visit our web site at www.capco.com, or follow us on **Twitter, Facebook, YouTube, LinkedIn** and **Xing.**

WORLDWIDE OFFICES

Bangalore
Bratislava
Brussels
Chicago
Dallas
Dusseldorf
Edinburgh
Frankfurt
Geneva
Hong Kong
Houston
Kuala Lumpur
London
New York
Orlando
Paris
Pune
São Paulo
Singapore
Stockholm
Toronto
Vienna
Warsaw
Washington, DC
Zurich

CAPCO.COM

© 2017 The Capital Markets Company NV. All rights reserved.