Executive Summary

Rapid digitization is causing tectonic shifts in customer expectations across industries. Many are struggling to adapt to this accelerated change. As a result, many businesses are aggressively pushing IT to cut delivery time, reduce cost and improve quality - all simultaneously.

Microservices architecture (MSA) has emerged as one way to address these challenges. MSA describes a new way of designing software applications as suites of small, loosely-coupled independent services. While there is no precise definition of this architectural style, these architectures encompass certain common characteristics around organization, business capability, automated deployment, intelligent service endpoints, and decentralized control of languages and data. Like their predecessors, they bring their own set of unique challenges that should be addressed up front to prevent the architecture from turning into a tangled mesh of point-to-point communication paths.

This white paper describes an approach to microservices design and development that is
based on adoption of RESTful services design best practices. Rather than conforming to a checklist of design and development guidelines and best practices that encompass our collective knowledge of the right ways of developing microservices, we recommend a tool-based approach that generates code compliant with microservices best practices and the chosen technology stack.

We believe that given their distributed nature and myriad moving parts, microservices can easily turn into an implementation nightmare, unless their design and processes are enforced and validated on an ongoing basis using best-in-class software engineering tools.

This paper offers a comprehensive microservices design and development technique that not only leverages rapidly evolving best practices but also supports and enforces them using a development workbench that we call Cognizant COSMOS. It builds on our thinking shared in previous works on the topic, “Overcoming Ongoing Digital Transformational Challenges” and “The Seven Waves of Change That Will Power, or Crush, Your Digital Business.”

**DEFINING MICROSERVICES ARCHITECTURE**

A microservices architecture is an architectural style for developing software applications as a suite of small, autonomous services that work together running in its own process and that communicate with other services using a lightweight communication protocol. The two protocols commonly used are HTTP request-response and lightweight messaging.

Microservices architectural styles are best understood by comparing them to traditional monolithic architectural styles - a style of application development where an entire application is deployed and scaled as a single unit. In monoliths, business logic is packaged in a single bundle and run as a single process. These applications are usually scaled by running multiple instances horizontally. Figure 1 depicts MSA’s architectural implications in comparison to monoliths.

**Microservices: Architectural Implications**

<table>
<thead>
<tr>
<th>Larger Service Count</th>
<th>A larger number of independently deployed services increases operational complexity. This shifts complexity from the application developers to the operations team.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast- Changing System Landscape</td>
<td>The system landscape is very dynamic, with services redeployed, replaced, deprecated and spawned on a continuous basis.</td>
</tr>
<tr>
<td>Remote Service Communication</td>
<td>Services communicate remotely with one another. Remote calls impact service performance and increase chattiness.</td>
</tr>
<tr>
<td>Larger Attack Surface</td>
<td>Modularity limits the privileges an attacker gets in a single attack. But it also increases the attack surface, as more services are exposed to the external world.</td>
</tr>
<tr>
<td>Frequent Service Failures</td>
<td>Higher-service modularity and remote calls increase the risk of failures while reducing the failure complexity and resolution time.</td>
</tr>
</tbody>
</table>

Figure 1
Each microservices implementation should not only be designed for failures but should have built-in fallback mechanisms to degrade gracefully with minimum impact to the service invocation chain.

**MICROSERVICES ARCHITECTURE**

**Implementation Challenges**

As IT organizations migrate to a microservices architecture, they typically contend with a number of challenges, most of which are not apparent in the monolithic world. As a result, IT organizations need a proven mitigation strategy and a reference architecture consisting of components that we believe are essential for implementing a microservices architecture from a design, delivery and operational perspective. Many of the challenges arise out of the very nature of the MSA paradigm.

Microservices use distribution to improve modularity. But distributed software has the following inherent challenges:

- **Performance:** Remote calls are slower compared to in-process function calls. While we may limit the number of remote calls, collaborating services need to be called at least once and in many cases as a chain of invocations. From an operational standpoint, a microservices implementation must have components for monitoring service performance and for tracing the service-call chain across processes and hosts.

- **Reliability:** Remote calls have a higher likelihood of failure than in-process calls. A large number of microservices thus means more potential failure points. Health monitoring of microservices and their hosts are key for ensuring system reliability.

Verification that all services are up and running is critical. Components for monitoring the service health and usage characteristics ensure a constant watch on service liveness by collecting run-time statistics and create a consolidated picture of service health. Besides service health, components to monitor the health of the virtual machines are also an essential part of the reference architecture.

For debugging, a component for centralized logging of all the services is needed; this can store log data in a central repository with search and dashboard capabilities.

- **Resiliency:** Each microservices implementation should not only be designed for failures but should have built-in fallback mechanisms to degrade gracefully with minimum impact to the service invocation chain. Patterns like Circuit Breaker or Bulkhead should be used to design services to avoid such cascading failures.

**Mitigation Strategies & Reference Architecture**

As Figure 2 (next page) shows, a microservices reference architecture must have components that address issues that arise due to its dynamic and distributed nature.

- **Service discovery:** Tracking microservices endpoints in a rapidly changing system landscape is a complex yet necessary task. Such rapid microservices configuration changes are hard to manage manually. Instead, IT organizations need service registration and
discovery functionality that enables microservices to self-register to a central service registry on startup. Service consumers use the discovery API to locate the requested microservice from the service registry. Additional load balancing components can determine which instance of the microservice to route the request to, assuming multiple instances are deployed for the requested service.

- **Configuration management:** Microservices architectures with their large number of deployed services make configuration management tedious and error-prone. Local file-based configurations are ineffective. Instead, a centralized configuration manager is a necessity. In addition, the configuration manager should provide APIs that the microservices can query to fetch configuration information.

- **Service access and routing:** A typical enterprise system landscape will contain many microservices. Some will be exposed externally over public networks, while others will be for internal consumption. All services, internal and external, are exposed through their respective (mostly RESTful) API models. While this unified approach to wrapping all business functionalities as services simplifies access, it demands additional infrastructure for access management.

To expose microservices externally and to prevent unauthorized access, IT organizations need an API gateway through which all external traffic is routed. A routing service, which acts as an entry point to the microservices environment,

---

**Microservices Reference Architecture**

---

Figure 2
may often double up as the API gateway. The routing service uses dynamic routing and load-balancing capabilities to route external requests to the selected service by looking up services from the service registry. In addition, an edge management portal may be required for administering routing rules.

To design the API model, with its constraints and design attributes, a developer workbench is needed to support the API modeling archetypes. The workbench should have features to publish the API model to an API store with support for the full API lifecycle. The API model also needs to be access-controlled using state-of-the-art security standards. From an API consumption perspective, we believe that a component in the microservices reference architecture is needed to support the definition of usage policies and its association with consumers.

- **Infrastructure automation:** DevOps is a key ingredient of a microservices reference architecture. Microservices, by nature, are agile and rapidly move from development to deployment. Continuous integration (CI) and continuous delivery (CD) automate testing and delivery, ensuring that the software is working as per expectation every time. (For more on CI and CD, read our white paper, “Patterns for Success: Lessons Learned When Adopting Enterprise DevOps.”

- **Data strategy:** One of the major challenges of migrating to a microservices architecture is to adopt the database-per-service design pattern. In monolithic service architecture, business transactions converge on the database layer, making it a shared service. This goes against the notion of an end-to-end loosely-coupled component, which is strongly advocated by microservices purists. The best solution to mitigate this challenge is to use an **eventually consistent database**.

### OUR APPROACH TO MSA EXCELLENCE

COSMOS is an open-standards-based solution for microservices design and delivery (see Figure 3)

**Peering into the COSMOS**

![Cognizant COSMOS: A unified solution for developing microservices; generating microservices governance components; modeling, publishing and managing the microservices; and building required artifacts for cloud-based and on-premises deployment.](image)

- **Supports multiple technology options and interaction**
- **Publishes API model to API store**
- **Generates deployment bundles for different deployment environments**
- **Generates operational governance components**
- **Generates business services components**
- **Generates critical infrastructure service components required for operational governance - routing service, discovery service, configuration service, performance monitoring and management components.**
- **Generates business services components**
- **Generates skeleton projects for the core and composite business microservices.**

---

*Accelerating Microservices Design and Development | 5*
Each microservices implementation should not only be designed for failures but should have built-in fallback mechanisms to degrade gracefully with minimum impact to the service invocation chain.

which implements the reference architecture described above. It also provides a unified workbench for the following activities:

• **Developing microservices:** Microservices characteristics include the organization of a logical block of functionality around a specific business capability, programming language agnosticism and decentralized control of data. Thus, each microservices project will have its API interface and implementation classes, its own data persistence and management module, client libraries required to efficiently communicate with other microservices and the operational governance infrastructure. Optionally, it may also include the client classes required to connect with back-end resources.

• **Generating components for operational governance:** The reference architecture recommends a group of critical infrastructure service components for operational governance. These may be broadly classified into the following:
  
  » A routing service that acts as a reverse proxy at the edge of the microservices network with support for dynamic routing rules.
  
  » A discovery service that allows individual microservices (and other supporting components) to register at startup, including client libraries to look up and discover the service instances from the service registry.

  » A configuration service that centrally stores and manages all configuration data across business and infrastructure services.

  » Performance monitoring and management components that monitor virtual machines, support log aggregation, visualization and analysis, ensure service resiliency through intelligent routing and enable distributed call tracing.

• **Modeling and publishing APIs for accessing microservices:** Our workbench provides an intuitive API designer interface to model RESTful APIs for the microservices. The modeling activity promotes the adoption of best practices and the typical constraints of the RESTful design pattern. It supports the use of various media types, including hypermedia constructs, in responses. Modeling activity is typically followed by automated code generation with the capability to generate a RESTful services interface and implementation classes using multiple technology options. The libraries
required by the microservices to connect with other microservices and to exchange data with the operational governance infrastructure is also injected into the skeletal classes. Custom service logic needs to be added to the generated code. Some microservices are exposed for invocation over the public network. The reference architecture recommends inclusion of infrastructure components for API management. These components typically consist of an API manager to manage, an API store to host and an API gateway to act as a single entry point for the public APIs.

We believe that API management infrastructure forms an integral part of the reference architecture for microservices implementation. The convergence of these related yet distinct technology paradigms is a critical success factor for microservices architecture. COSMOS has support for different API management solutions, both open-source and commercial.

**Building and deploying the artifacts to cloud or on-premises infrastructure:** The workbench supports service deployments to both on-premises infrastructures and cloud by packaging the developed artifacts (compiled code, configurations, etc.) into executable, deployable bundles. COSMOS can also be integrated with continuous delivery infrastructures (CI/CD) whereby any changes made to components may trigger the CI/CD build and deployment pipeline directly from the workbench.

**IMPLEMENTING MICROSERVICES VIA OPEN SOURCE**

One of the key characteristics of a microservices architecture is technology diversity and the support for it. There are a variety of libraries and frameworks available today that implement the various components of the reference architecture shown above. The variety exists not only in terms of programming languages, but also in terms of the various communication
patterns between these components. Synchronous interaction between microservices may not always be the optimum option. The exchange of data with operational components for performance monitoring or call tracing, for instance, typically adopts the asynchronous message exchange pattern through an underlying message broker.

COSMOS supports the open-source Spring Cloud Netflix stack as one of its technology options for implementation. Figure 4 depicts how components of the Spring Cloud Netflix stack map to our reference architecture. Components depicted in Figure 4 that directly map to the Spring Cloud Netflix, Spring Framework and other recommended open-standard, open-source stacks are represented in bold italicized font. COSMOS can also integrate with custom-built and third-party solutions for implementing components of the reference architecture as needed. Support for additional technology options are being continuously added to COSMOS. Apache Camel and Eclipse Vert.x support for implementing a microservices architecture are in the works and planned for release in the next version.

Springing Forward with Microservices

“One of the key characteristics of a microservices architecture is technology diversity and the support for it.”
LOOKING FORWARD

COSMOS offers a wide variety of technology options and communication patterns to implement microservices architecture. It is envisioned to be an extensible development tool in which support for newer libraries, frameworks and deployment platforms can be added incrementally through plug-ins as they mature and are ready for production use. We believe that a solution like COSMOS provides a number of compelling benefits, including:

- A wide array of technology options for implementing different components of a microservices reference architecture.
- Auto-generating the plumbing code based on the chosen technology stack, ensuring a high quality of coding standard compliance and consistency across implementations.
- Enforcing a common standard for API modeling across teams, business units and geographies.
FOOTNOTES


ABOUT THE AUTHORS

Dipanjan Sengupta
Chief Architect, Software Engineering and Architecture Lab

Dipanjan Sengupta is a Chief Architect within the Software Engineering and Architecture Lab of Cognizant’s Global Technology Office. He has extensive experience in service-oriented integration, integration of cloud-based and on-premises applications, API management and microservices-based architecture. Dipanjan has a post-graduate degree in engineering from IIT Kanpur. He can be reached at Dipanjan.Sengupta@cognizant.com.

Hitesh Bagchi
Principal Architect, Software Engineering and Architecture Lab

Hitesh Bagchi is a Principal Architect within the Software Engineering and Architecture Lab of Cognizant’s Global Technology Office. He has extensive experience in service-oriented architecture, API management and microservices-based architecture, distributed application development, stream computing, cloud computing and big data. Hitesh has a B.Tech. degree in engineering from University of Calcutta. He can be reached at Hitesh.Bagchi@cognizant.com.

Pijush Kanti Giri
Architect, Software Engineering and Architecture Lab

Pijush Kanti Giri is an Architect within the Software Engineering and Architecture Lab of Cognizant’s Global Technology Office. He has extensive experience in Eclipse plug-in architecture and developing Eclipse RCP applications, API management and microservices-based architecture. Pijush has a B.Tech. degree in computer science from University of Kalyani. He can be reached at Pijushkanti.Giri@cognizant.com.
ABOUT COGNIZANT

Cognizant (NASDAQ-100: CTSH) is one of the world’s leading professional services companies, transforming clients’ business, operating and technology models for the digital era. Our unique industry-based, consultative approach helps clients envision, build and run more innovative and efficient businesses. Headquartered in the U.S., Cognizant is ranked 230 on the Fortune 500 and is consistently listed among the most admired companies in the world. Learn how Cognizant helps clients lead with digital at www.cognizant.com or follow us @Cognizant.

© Copyright 2017, Cognizant. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the express written permission from Cognizant. The information contained herein is subject to change without notice. All other trademarks mentioned herein are the property of their respective owners.

TL Codex 2533