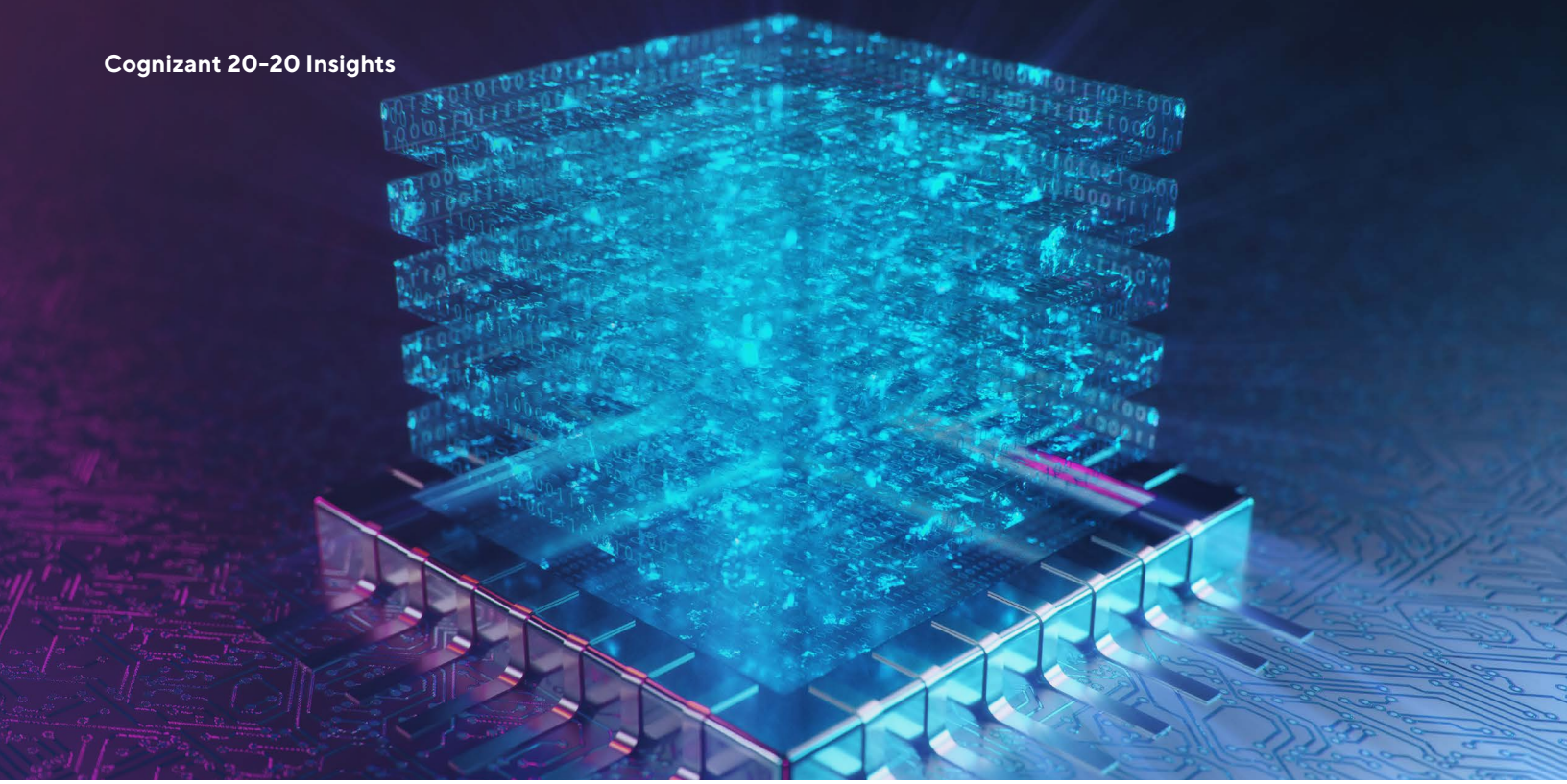


Cognizant Digital Systems & Technology

Pitfalls & Challenges Faced During a Microservices Architecture Implementation

Microservices are the *de facto* design approach for building digital applications. However, issues highlighted in this paper can and do lead to implementation challenges and even failures. Here are a few strategies to avoid and overcome them.

Executive Summary

Organizations across industries are at various stages of their journey toward adopting a microservices architecture style.[1] Some have been successful in delivering real business benefits, while others are still experimenting.

While the benefits delivered by a microservices architecture such as agility, selective scalability and availability still hold true, we are dismayed by the various suboptimal implementations of microservices architectures that have emerged since our initial take on the topic.[2]

This white paper provides readers with guidance on fundamental design decisions required to properly implement a microservices architecture to realize all the benefits available to organizations willing to take the plunge.

February 2020

Cognizant

Some teams tend to think of the "micro" in microservices as a lever to design smaller-scoped systems, while others think of micro as the lever to deploy and operate manageable systems.

Microservices architecture pitfalls

A large number of the teams developing systems based on microservices architecture have some form of experience in service-oriented architecture (SOA) that was heavily influenced by the middleware vendor products such as enterprise service buses (ESBs) or Web standards such as Simple Object Access Protocol (SOAP) and Web Service.

While SOA experience helps in developing service-oriented thinking, we have found that this background can also have a negative influence, resulting in pitfalls such as sub-optimal granularity of service, representational state transfer (REST)-only mindset, excessive service calls (chatty services) and database as shared resource mindset.

Continuous technological improvements have caused some teams to fall into a technology-only thinking trap, causing even more complexity when these systems experience high-volume production traffic. These pitfalls manifest in multiple forms. (Read on to see what we have observed across projects.)

The "micro" in microservices architecture

Application architects have struggled with the granularity dilemma for a long time. What's the right level of granularity for a system component to be scoped? How we can maximize reuse for a system component or service in its functional form? Basically, identifying the right boundaries and granularity are two of the biggest challenges for a solution designer, especially as the selection depends greatly on the domain and context of the solution space.

Application developers using microservices style face similar challenges. Some teams tend to think of the "micro" in microservices as a lever to design smaller-scoped systems, while others think of micro as the lever to deploy and operate manageable systems. We see three major variations of this issue in microservices scoping (see Figure 1, next page).

- **Entity scope**: Data entity-oriented design tends to cause inflexibility at the individual microservices layer and change complexity at the overall system level.
- **Process scope:** Designing microservices at this level of granularity tends to result in fragile and volatile services.
- **Utility scope:** The notion of utilities treated as microservices causes the provisioning of operational sophistication without reaping adequate business benefits since such utilities are often non-differentiating to most enterprises.

Microservices granularity levels

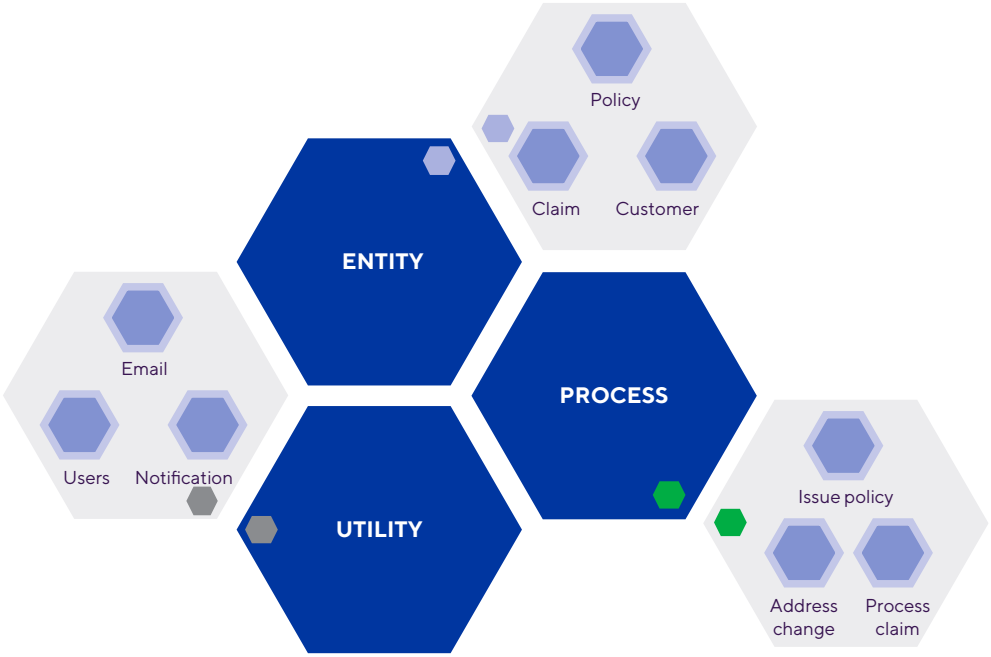


Figure 1

While all granularity levels are expected to be served in a system, the main benefits of such architecture design can easily be missed; that is, balancing agility (speed of change) with resiliency (safety of change). Two issues are typically observed:

- It's difficult to get business buy-in to the program that's delivering such services.
- Operational complexity increases without real benefits.

Finally, as IT organizations embrace a system paradigm across all non-trivial business domains, event-oriented collaboration patterns have become a mainstay of such microsystem architecture design. These pitfalls tend to become a big bottleneck that thwarts event-oriented thinking on the journey toward evolutionary system design.

To avoid these pitfalls, IT organizations must ensure that no microservice is designed without an explicit alignment and traceability with business capability of the domain, as advocated by bounded context strategy of domain-driven design.[3] In addition, the microservice designers must pay close attention to the context-dependent interactions that the microservice needs to participate in. This requires the designers to think in terms of loosely coupled, event-oriented boundaries and the context mapping.

The database monolith

Microservices adoption moved quickly from an emerging concept to the *de facto* design pattern for application architecture. However, as with any over-hyped technology, the design patterns and best practices were not very well established and understood, which led to poor implementations.

Among the major obstacles are large databases used as a persistent store for many – or all – of the microservices. This database can lead to a monolith at the data persistence layer, resulting in several challenges such as:

- **Performance bottlenecks.** One of the key drivers for microservices architecture is the ability to scale horizontally and dynamically.

Quick Take

Advancing Innovation & Time to Market in Consumer Lending

One of our APAC clients in the consumer lending domain that wanted to enable innovation and improve speed-to-market transformed a single monolithic application into microservices with little or no focus on the design aspect of microservices.

The result was 500-plus microservices with extreme complexity leading to performance bottlenecks due to chatty inter-service communication.

We have recommended a domain-driven rationalization to address this situation. This not only helps address performance challenges, but also ensures evolution of the application in a completely autonomous manner.

However, with a monolithic database, the scaling of microservices puts additional load on the database, creating a performance bottleneck.

- **Coupling between microservices.** Microservices architectures offer agility in that they are loosely coupled and independently deployed. However, if multiple microservices are tied to the same tables in the database, then any change in the schema will result in cascading changes in other microservices, which defeats the core purpose of microservices.

There are many reasons we have seen that lead to this anti-pattern; two key ones are risk averseness to move away from the monolithic database and database designers who are not fully skilled in newer patterns like microservices, leading to traditional database design.

If multiple microservices are tied to the same tables in the database, then any change in the schema will result in cascading changes in other microservices, which defeats the core purpose of microservices.

To avoid these pitfalls, IT organizations must ensure that the database design complements the microservices by following best practices:

- **Database per service in a no-share model.** Each microservice needs to have full ownership of the data it requires. This does not mean a separate physical database but ownership of the data it masters.
- **Polyglot persistence model.** Making a relational database a default storage of all types of data leads to poor results, hence all types of databases (e.g., NoSQL, Graph, in-memory, etc.) must be used.
- **Adopt CQRS pattern to use read replicas.** Command query response segregation (CQRS) is an architecture pattern but it can be applied to microservices database design to answer the biggest question: Can data be shared between microservices?
- **Break distributed transactions with a saga pattern.**[4] The division of database objects into groups of logical schemas in bounded context of the wider transaction boundaries require multi-phase commits. Avoid this with saga patterns that keep intermediate states.

Quick Take

Overcoming Centralized Database Shortcomings

One of the new cloud-native features of a consumer lending solution implemented in pure microservices style was undermined by a single centralized relational database.

This design led to the database becoming a performance bottleneck and hindrance to system scalability and resilience.

We addressed this issue by refactoring the database into multiple domain-centric physical instances with microservices-based logical separations. This allowed domain-based isolation, which enabled resiliency, scalability and performance improvements.

Breaking the data monolith through bounded context and polyglot design

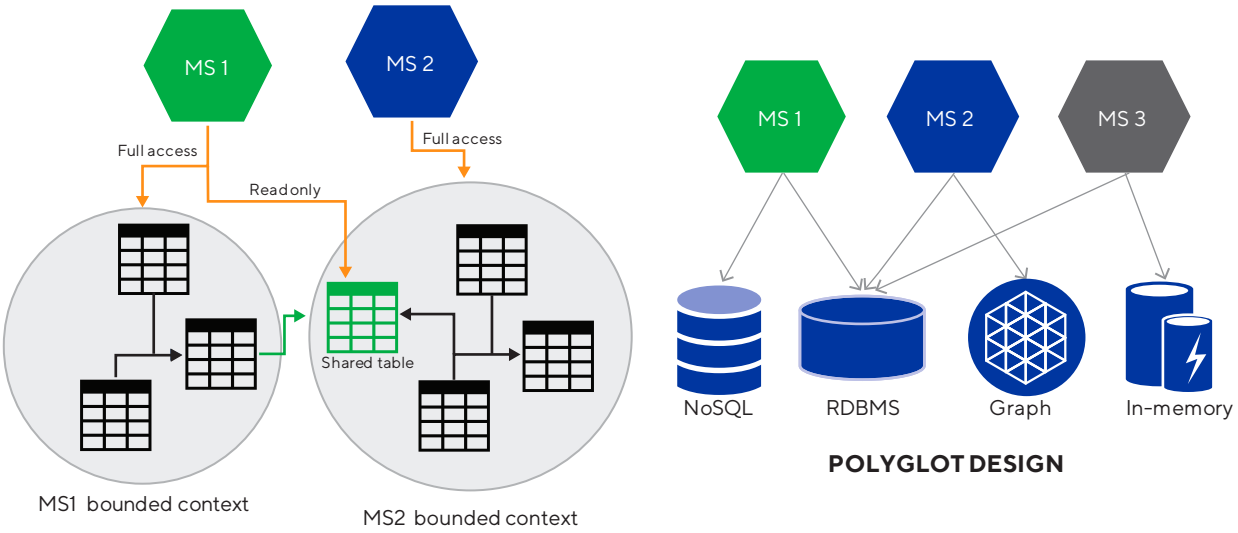


Figure 2

The REST compulsion

The microservices architecture grew during the decline of traditional SOA. As most of the traditional SOA was built on HTTP-based SOAP, the natural evolution for microservices was to expose its functionality through HTTP-based REST services.

The pitfall here is that most developers consider HTTP REST as the default way to expose the functionality of a microservice rather than considering asynchronous messaging alternatives. This leads to various challenges of performance and complex transaction management.

The key reasons why HTTP REST is used as the default for microservices implementation:

- REST API has become very popular and developers are comfortable with its implementation.
- Most developers are not experienced in reactive patterns and hence avoid implementing services based on asynchronous messaging.
- Reactive services need additional messaging infrastructure to implement besides microservices, which is not generally available.

To avoid these pitfalls, we recommend the following best practices as shown in Figure 3 (see next page).

- **Understanding the distinction between API microservices and core microservices.** During the microservices design, there should be a clear distinction between API microservices that expose their functionality to the external world against core microservices that are used by other microservices. The core microservices should be implemented on a reactive pattern.
- **Developer awareness of reactive system architecture is key.** Developers should understand reactive system architecture and associated benefits such as responsive and resilient microservices.

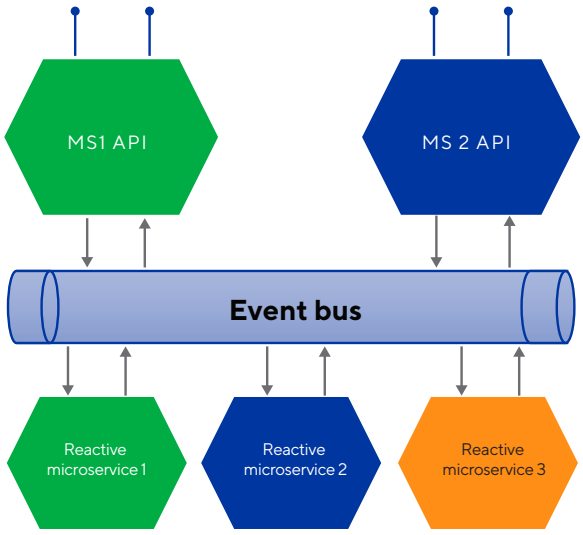
Quick Take

Overcoming REST Challenges

One of our large banking clients has implemented a core platform that offers both message-based and service (REST) interfaces. However, in pursuit of REST-only integration, the company implemented a REST interface on top of messaging that resulted in interface complexity, transaction losses and latency issues due to the increased processing pipeline and number of components engaged.

This is now addressed by offering APIs as either messaging or services. Inter-service communication is now being handled through messages directly across components, helping with both loose coupling and high reliability.

Reactive microservices and integration



For any microservices-based implementation, the architecture should have a provision for core messaging infrastructure. For more complex architectures, a more elaborate event bus architecture should be considered.

Figure 3

Overall, while REST is a very useful protocol to expose functionality for microservices, indiscriminate use can lead to challenges that can be avoided by using asynchronous patterns.

Chatty services

A chatty application is one that relies on numerous services to fulfill a request or a process. In most cases, chattiness is a result of designing microservices that are too fine-grained and break the bounded context and independent business capability principle.

While calling other services to fulfill a request is often considered as normal and acceptable, chatty services incur numerous overhead such as:

- Network latency, disk reads, database queries, etc. on both the calling service and the service being called.
- Runtime dependency between the microservices, resulting in a distributed monolith. All dependent microservices need to be available and operational at the same time.

Quick Take

Embracing an Event-Driven Architecture

A large bank had embarked on a transformation program to modernize its payment landscape using a microservices architecture style.

A centralized service orchestration model was implemented for payments processing that led to chattiness between orchestrator and functional services. This affected both latency and throughput of the platform. Additionally, the orchestrator became a scalability bottleneck for the system.

Our recommendation was to consider an event-driven architecture and implement long-running processes using service choreography since it results in a highly scalable and performant design due to loose coupling and non-blocking.

- Testing, which can become challenging as all the dependent microservices need to be available.

If such issues are observed, then first revisit the microservices design to ensure that the essential principles of domain-driven design have been followed.

It is also worth analyzing the dependency on other microservices. If the dependency is on the data provided by another microservice, then the IT team should consider replicating that data to avoid the remote call. This also allows the service to transform and store the data in a way that is optimal for a given microservice. Additional benefits result beyond an avoided remote procedure call. For example, within customer management and know your customer (KYC) bounded contexts, a customer snapshot remains persistent, which avoids the need for a remote call. See Figure 4.

Sharing data across microservices

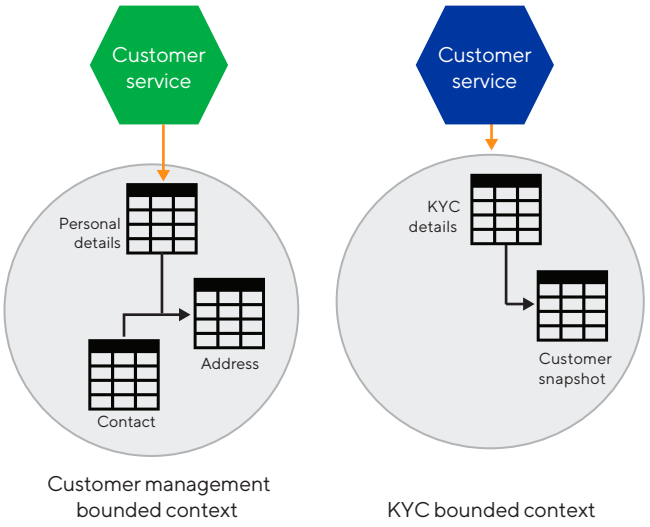


Figure 4

Loose coupling through asynchronous communication

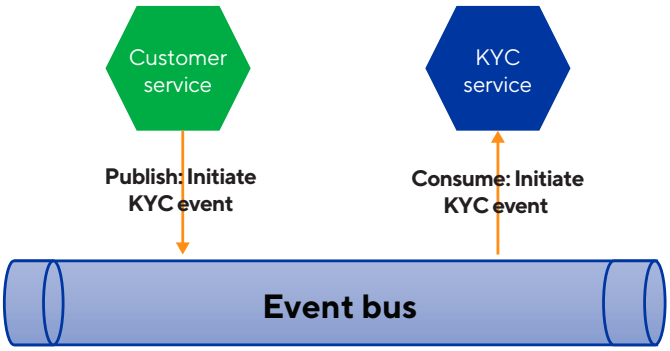


Figure 5

In some cases, one microservice might need to call another microservice to trigger the business logic it contains. In these situations, one service cannot avoid communicating with the other, but the IT team can implement it as efficiently as possible via asynchronous communication protocols (see Figure 5).

Technology-only thinking

While designing microservices, we often see development teams acting in complete isolation with business stakeholders. IT teams tend to think that once the business has provided its project requirements, design is an IT-only activity and sub-system decomposition, and defining microservices and release mechanisms are the technology team's agenda. This turns out to be a major shortcoming in projects where business subject matter experts (SMEs) are not involved in microservice design systems that are misaligned with business objectives. In addition, the evolution of these microservices tends to be influenced by the technologists.

In the majority of cases, this snag is a result of problems on both sides. The business thinks topics like microservice design are too "techie" and thus believes it has no role to play. IT thinks business doesn't have any useful know-how to contribute to microservices design and thus shouldn't be invited/consulted. System designers create application boundaries that do not align with the business capabilities and the experience design that the business requires. The result: the microservices identified for such projects do not reflect the business domain (both problem and solution) and instead are filled with technical services.

Our experience suggests that, more than technology, it's the culture and systems thinking that influences how well we deal with these pitfalls in microservices-based systems.

To avoid this pitfall, solution architects must play a major role. They need to ensure that business has a seat at the table while designing microservices and both business and IT are able to offer reasons about design for delivery, lifecycle management and evolution of business capability in scope.

Another major issue in this category tends to be rooted in the accelerating evolution of technology. Every few months (sometimes even weeks) a new shiny technology is released by vendors who talk up their use of microservices. While continuous technology evolution is a good thing, it tends to stimulate interest in the technology by business and IT teams, who steer their project in this direction without really understanding the implications.

An example is Docker and Kubernetes. Statements like "We are designing our microservices using Kubernetes" or "These performance problems will now be gone as we are structuring our system as microservices on Docker" are becoming commonplace. We believe this pitfall is one of the biggest issues facing microservices adoption today.

Technology is a strong enabler to get microservices right, but that's not the endgame. Teams should not let these technologies steer their efforts toward microservices. Once they have designed the right level of microservices that are aligned to the business being automated, technology innovation should be used to deliver the promise of high quality and agility. Some of the areas where microservices will really benefit from technology evolution include elements such as automated deployment, release, scaling, secure communication, high performance, availability, monitoring and provisioning. In fact, our experience suggests that, more than technology, it's the culture and systems thinking that influences how well we deal with these pitfalls in microservices-based systems.

Confronting the challenges

A microservices implementation is no easy task and there are many challenges that test a team's abilities.

While the pitfalls can be attributed to speed of change and a lack of perspective (this is just new SOA technology), there are some genuine challenges that teams face while adopting the microservices architecture style.

Although some of these challenges are related to how systems are designed, others concern how these systems are operated and managed in production. For example, architects and developers at one of Europe's major banks, whose operations pivot around a large mainframe and database-rich landscape, had a hard time grasping the design of business-aligned microservices using domain-driven design.

In another case, one of our clients implemented a series of microservices but had a hard time achieving production stability of these services

or obtaining a good view of service integration across applications. In addition, we observed that teams faced real challenges in terms of people skills and capabilities and teams have spent substantial efforts and money to carry out workforce re-skilling. We'll look at intricacies of these challenges next.

Design challenge

Organizations struggle consistently with microservices design challenges such as determining the optimal boundaries between the microservices, size of microservices, integration between microservices, etc. Microservices architecture design challenges include:

- **The team only consists of IT specialists or technology architects**. Defining capability-aligned service boundaries requires domain experts. Fundamentally, this should be a combined exercise independent of technology used.
- **Architects or technologists always consider data as the most important thing and use a data-centric view when modeling a problem domain.** Without logic, the data is meaningless. Hence, architects and technologists should start with context (business capability) and logic instead of data.
- **At times, UI screens are used as guidelines for identifying data ownership and service boundaries.** User interface (UI) doesn't help here as data matters only when it is involved in some business logic – not when it is just displayed.
- **There is a tendency to focus on database transactions instead of business transactions or business processes**. Focus should be on real-world processes, such as actions, their outcomes and compensating for the failed actions if failures occur. A properly designed bounded context modifies only one aggregate instance per transaction.

Quick Take

Going Domain Driven

One of our clients faced challenges in correctly modeling the microservice aggregate – the cohesive core model of any microservice. The team fell into a trap of designing for compositional convenience and the resulting aggregates were too large, with data consistency problems.

We recommended a domain-driven design approach to discover the aggregates in a business operating construct aligned with boundaries called bounded contexts. This approach resulted in a domain-aligned, coherent model with true invariants that addressed the data consistency problems.

Quick Take

A Decoupling Approach

One of our financial services clients faced outages on its online portal as result of resource starvation by a system performance monitoring solution. The agents on the tool exhausted resources required for business logic processing, causing the portal to go down.

Not isolating the system software from business software was one of the reasons for the failure, and detecting and isolating it was a difficult job.

The recommendation to decouple the system policy concerns from functional software was applied through proper runtime-decoupling to resolve this issue. Additionally, the monitoring and resiliency test practices were enhanced to detect such dependencies through DevOps continuous integration/continuous delivery (CI/CD).

- **Developer-centric terms such as create, read, update and delete are too technical and have no specific business meaning**. The team must always think from the business's point of view, and give a clear context to it.
- **Building a large organization's system of microservices is difficult and requires building a view, context by context.** A starting point for representing a system of microservices can be a context map.
- **Consistent use of unambiguous language is missing, which leads to a lack of domain understanding.** The technical and business obstructions in the language may not discover the vital concepts hidden or assumed by domain experts. For example: a customer enrolls using a social profile. "Enroll" here has a technical or business obstruction because of various questions about issues, such as what happens during enrollment? Is the customer enrolled for any product/s or is the customer enrolled only to create his or her profile?

Resiliency challenge

Microservices bring a host of benefits – primarily enhanced operational agility; however, a microservices implementation increases complexity by its decomposition of application functionality into many independent deployable units. One challenge that emerges with this complexity is of resiliency due to the following factors:

- **The distributed nature of request processing.** In a microservices environment, most requests are processed by multiple microservices, which increases the dependency on network and infrastructure services, thus increasing the probability of failure.
- **Challenges in failure detection.** In a traditional monolithic system, failures are simple to detect due to fewer probable causes and failure of application as a whole. In a microservices environment, the failure can be of many causes such as the microservice itself, the container

it is running on, and the network that is interconnecting the microservices.

- **Recovery after failures.** As the failure usually results in complex intermediate states, it is often difficult to recover from. While respective microservices can be restarted, the transactions that were in-flight must be recovered from their failure state, which is often difficult.

Some strategies that we have found helpful to address the challenge of resilience in a distributed microservices environment include:

- **Observability as a key architecture concern.** For a microservices environment to be highly resilient it needs observability implemented at each level (i.e., infrastructure and application). The capability to log, monitor and trace requests across the network is key to providing resiliency.
- **Design for recoverability more than failures.** While any well-implemented system is defined for failures, it is recoverability more than failures that address the concern of resilience. The application should have the ability to recover from a failure automatically at the container (e.g., restarting a container), a microservice (e.g., reinitiating a connection pool) and application state level (e.g., maintaining consistent state after recovery).
- **Design for idempotency.** One of the key features to be implemented at the microservice level to enable flawless recovery is idempotency. Idempotency is the feature to retry the same request without impacting the state. With idempotency, each of the inflight transactions can be retried without compromising the overall system state.
- **Delegation of intercommunication to a service mesh.** While some of the microservices implementations use patterns like a circuit breaker, in any complex microservices-based application circuit breaking is not enough. Implementation of a full-service mesh (e.g., Istio[5]) or simpler side car proxy (e.g., Envoy[6]) can take away the complexity of intercommunication.

In summary, resiliency is a big challenge in microservices-based applications and unless dedicated architecture and design focus is given, the desired outcomes will not be achieved.

Complexity challenge

Complexity reduction through well-defined bounded contexts and communication patterns is one of the critical benefits that microservices provide. If done right, microservices offer excellent support for autonomous evolution of business capabilities. Typically, such microservices are also business-capability driven and therefore act as the common vocabulary used by both business and IT teams, resulting in effective evolution of business capabilities.

Complexities introduced by microservices architecture

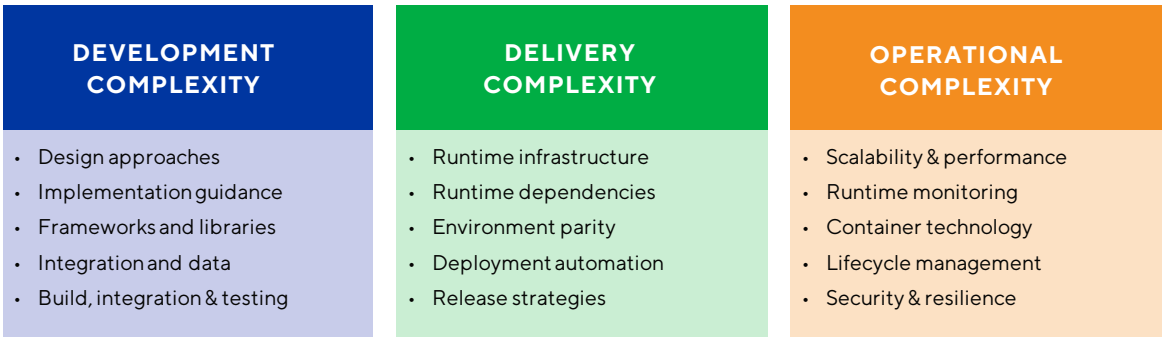


Figure 6

However, microservices come in systems, which means that often any non-trivial enterprise system tends to have dozens or hundreds of these systems (each developed as a microservice). As each of these systems are narrowly focused, single-purpose microservices, any business process or user interaction tends to result in invocation and interaction of multiple microservices, resulting in challenges across multiple phases of projects:

Development complexity
Microservices development requires teams to think in terms of distributed application design and interaction patterns. Applying concepts such as CQRS, functional interfaces, CAP,[7] BASE,[8] and sagas[9] in contemporary programming languages and configuration is not something that developers are used to.

Moreover, this is a new area of complexity for many teams. Data persistence and integration requires them to not just understand aspects such as polyglot persistence,[10] persistence ignorance[11] or event-driven messaging,[12] but also poses challenges in terms of frameworks, libraries and programming languages to choose from.

Finally, how to build and generate these polyglot microservices as a coherent whole through the complex continuous integration/continuous delivery (CI/CD) pipeline is a major concern that many teams must overcome. Last but not least, testing microservices is still an evolving area and poses a major challenge to the majority of teams.

Delivery complexity
With the advent of CI/CD and automated infrastructure provisioning through APIs, deployment and release practices have evolved over the last few years. In addition, the use of containers and managed cloud environments in the platform-as-a service (PaaS) model requires teams to work with constructs provided by these technologies/services.

While developing microservices in this environment, deployment and release is typically coded in the form of YAML or JavaScript Object Notation (JSON) scripts and these artifacts have become a first-class citizen of the code repositories for microservices. Technology diversity, a lack of standards and a plethora of agility-oriented release mechanisms (canary release, blue-green deployment, push-to-

Quick Take

Going Serverless

In one of our serverless microservices-based applications consisting of 100-plus serverless services, we recommended using log streaming for application logs, infrastructure logs and security logs to a central log store.

With log streaming, we were able to add various log processors and generate a variety of real-time metrics such as configured memory vs. memory used. Ultimately, this technique helped us to establish a centralized log store for monitoring application, infrastructure and security events in a unified manner.

green release, dark release, "dogfooding," etc.) are among the areas of complexity that teams must deal with.

Finally, the element of release environments with necessary dependencies required for a microservices-based system to be deployed, tested and operated is another area of complexity that many large enterprises have to solve.

Operational complexity

Once released into production, microservices present challenges in the areas of capacity utilization, scaling, failures and monitoring – to name a few. Operating an infrastructure with tens or hundreds of microservices requires sophisticated tooling for automated provisioning in a secure and resilient manner.

Technologies such as Kubernetes provide the necessary foundation for this but require operational procedures and practices beyond technology. In the case of monitoring, tools such as ELK,[13] Grafana,[14] Jaeger,[15] Prometheus,[16] or cloud providers (e.g., AWS CloudWatch) help deal with this complexity, but orchestrating all of these tools at scale is not a trivial task. If the microservices are operated in container or serverless runtime, then these complexities are amplified as the infrastructure is quite dynamic.

Being able to achieve production stability and a real-time view of the hosted services is anything but trivial. Security of interactions and APIs across bounded contexts and partner systems is another area of complexity. A good API management, messaging infrastructure and monitoring approach is essential to overcome these challenges.

We strongly suggest that teams consider these challenges and set up the right infrastructure, processes and practices to deal with them proactively.

Observability challenge

Systems can be understood only if they are observable. Observability, monitoring and analysis are in a symbiotic relationship, which is depicted as a pyramid (see Figure 7).

Key challenges faced by microservices implementation teams include:

- **Observability vs. monitoring.** Observability is about making the data available from within the system to be monitored; monitoring is the task of collecting and displaying that data. Microservices teams focus on monitoring tools without making the software observable.

The observability pyramid

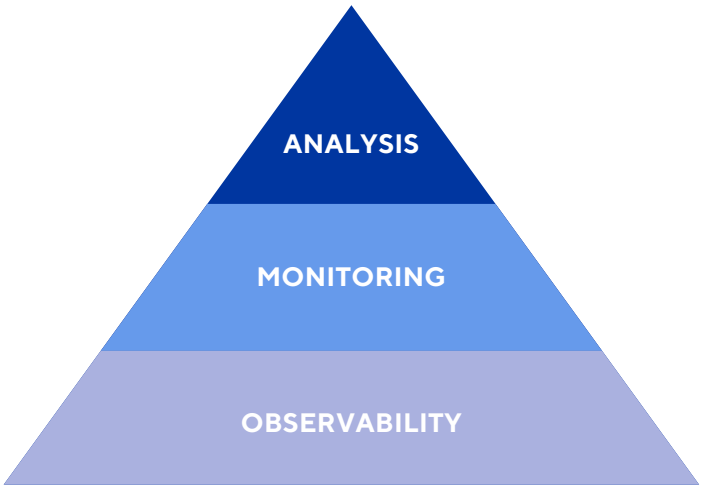


Figure 7

Quick Take

Working Around Event-Driven Obstacles

In one of our banking client's event-driven straight through processing (STP) applications, the team was struggling to monitor the event processing.

We proposed a mechanism in which each functional module published an event before and after processing. We also defined application logging standards, log aggregation and monitoring approaches, which helped the team to enable effective observability and monitoring.

As a result, incident resolution improved and business KPIs provided real-time insights to the business.

- **Distributed tracing.** The most difficult part of observability is distributed tracing with and between application services. A great deal of work is involved in implementing distributed tracing. Microservices teams lack the underlying principles of tracing requests that flow between services. Organizations typically need a cultural overhaul to embrace observability as part of the development process.
- **Contemporary tools.** Modern application delivery has shifted to containerization, microservices, and polyglot environments, which cause problems for monitoring tools. The speed of deployment has increased along with the number of software components that are deployed. The existing monitoring tools or production profilers are finding it difficult to keep pace. Additionally, these tools have trouble identifying and connecting dependencies between microservices, especially at the individual request level.
- **Automating observability.** The practicalities of implementing observability can be quite significant, which puts pressure on developers. This results in developers spending time on writing instrumentation for monitoring rather than the functional code.

Guidelines that we have found helpful to address the observability challenges include:

- **Establish standards and governance.** For observability, it is critical to establish standards and governance. Without standards, the collection and correlation of event logs and metrics becomes highly challenging, compromising the outcomes. IT organizations must establish the following standards.

  Logging standards:

  > Standards such as, log format, log level usage and logging frequency.

  > Business data standards: Logging of business data (masking and hiding).

  > Technology standards: Clear guidelines on tools usage for dev engineers.

Governance standards:

- > Tollgates to validate right logging and tracing done in the functional code.
- > Awareness and enablement of dev engineers for knowledge and the capability to implement the right tooling.
- > Feedback loop: Incidents should result in changes in log/trace implementation.

- **Tooling guidance:** Frameworks that automate logging, tracing and tools for monitoring and analyzing the metrics.
  - > Logging and tracing: Istio, Jaeger, Sleuth, Zipkin, Dynatrace.
  - > Log aggregation: Logstash, Elastic Search, Graphite.
  - > Monitoring and analysis: Prometheus, Kibana, Grafana.

People challenge

Microservices-style architecture has forced changes across all three dimensions of system development – design, operations and delivery practice.

As outlined in Figure 8, these systems possess internet scale characteristics that require developers to (re)think the way we design applications. In addition, the environment in which we operate enterprise applications is changing dramatically to improve quality of service. Finally, the practices and processes to deliver software to business is now strongly influenced by techniques favoring agility in software delivery.

Given these developments, microservices developers must evolve their skills and thinking. Architects and designers need to develop skills across multiple areas such as evolutionary architecture, distributed design, polyglot data architecture, event-centric integration, service meshes, domain-driven design, CQRS, resiliency engineering, etc.

Systems engineering and operations need to develop newer skills in disposable, immutable and ephemeral infrastructure design, technologies such as Cloud Foundry, OpenShift, Docker, Kubernetes, etc. Most of these technologies also require the operations teams to pick up new skills in systems-oriented declarative programming as many rely on software-defined infrastructure models.

In terms of practices, all teams need to ensure that there is collective broad-based capability on Lean Product Development (LPD), continuous delivery and reliability engineering practices. These practices have become the mainstay of high-quality software delivery on an ongoing basis. It is safe to assume that these are prerequisites for all non-trivial projects based on microservices architecture.

In our experience, microservices efforts in enterprise IT organizations are still in their nascent stage. To move forward, IT management must invest in people capabilities in addition to

Three dimensions of system development

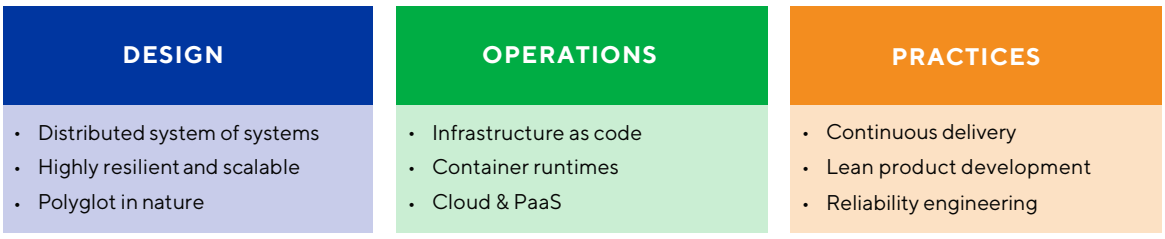


Figure 8

Quick Take

Team Building

One of our clients decided to modernize its IT landscape using microservices architecture style. Since the domain was complex and the landscape was large scale, we worked with the client to enable a diverse workforce on microservices design, cloud-native infrastructure, SRE disciplines and DevOps practices in a dedicated infrastructure setup.

This enablement and competency development led to a strong team of developers across locations delivering microservices-based projects in a consistent manner.

technology infrastructure modernization. These efforts may include a wide variety of measures such as formal training, certifications (for specialized technologies), hiring experts to augment existing teams, creating community of practices, collaborating with partners for pilot projects and providing developers with broad industry exposure by participating in industry conferences, hackathons, cloud sandbox environments, specialized labs, etc. Engaging an external specialist or consultant for a short time to handle capability enhancement challenges is not a scalable option.

Looking forward

Microservice architecture style is one of the prominent and relevant design approaches for developing cloud-native systems. We strongly consider that our clients apply this style in distributed systems development. However, the pitfalls and challenges presented in this paper tend to result in sub-optimal applications of this architecture style, leading to an environment in which teams start to think that microservices is yet another adopted fad.

The additional complexity in operations of and troubleshooting these systems creates further issues. Hence, we highly recommend keeping an eye on these pitfalls and proactively addressing the challenges so that microservices benefits can be effectively achieved. Wherever any of these pitfalls or challenges result in tactical decision-making, we should ensure that such exceptions are handled though a proper technical debt-management mechanism.

Finally, the extent to which enterprises can address these issues depends on infrastructure maturity and design competency within IT and alignment of business and IT overall. Try to evolve toward that state instead of trying to address all of these pitfalls and challenges upfront, learning and adapting your approaches along the way.

Endnotes

1 James Lewis, "Microservices," Martin Fowler, March 25, 2014, http://martinfowler.com/articles/microservices.html.

2 Cognizant, "Overcoming Ongoing Digital Transformation Challenges with Microservices Architecture," November 2015, https://www.cognizant.com/InsightsWhitepapers/Overcoming-Ongoing-Digital-Transformational-Challenges-with-a-Microservices-Architecture-codex1598.pdf.

3 Domain Driven Design – Bounded Context, https://martinfowler.com/bliki/BoundedContext.html.

4 A saga is a sequence of local transactions where each transaction updates data within a single service. The first transaction is initiated by an external request corresponding to the system operation, and then each subsequent step is triggered by the completion of the previous one (Rosa, 2018).

5 Istio, https://istio.io/.

6 Envoy, https://www.envoyproxy.io/.

7 CAP Theorem and Distributed Database Management Systems, https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e.

8 BASE – database transaction processing, https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/.

9 Saga pattern, https://microservices.io/patterns/data/saga.html.

10 http://www.martinfowler.com/bliki/PolyglotPersistence.html.

11 https://deviq.com/persistence-ignorance/.

12 https://en.wikipedia.org/wiki/Event-driven_messaging.

13 ELK – Elasticsearch, Logstash and Kibana, https://www.elastic.co/what-is/elk-stack.

14 Grafana, https://grafana.com/.

15 Jaeger, https://www.jaegertracing.io/.

16 Prometheus, https://prometheus.io/.

References

- A. Brandolini, *Strategic Domain Driven Design with Context Mapping*, November 25, 2009, Retrieved from infoq.com: https://www.infoq.com/articles/ddd-contextmapping.
- M. Fowler, *BoundedContext,* January 15, 2014, Retrieved from martinfowler.com: https://martinfowler.com/bliki/BoundedContext.html.
- D. Rosa, *Saga Pattern,* January 10, 2018, Retrieved from blog.couchbase.com: https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/.

About the authors

Dinkar Gupta

Senior Director & Chief Architect, Cognizant Banking and Financial Services

Dinkar is a Senior Director & Chief Architect with Cognizant's Banking and Financial Services (BFS) Technology and Architecture Office. Based in Switzerland, Dinkar leads the practice for Europe and the UK region. He has a post-graduate degree in computer science and applications from National Institute of Electronic and Information Technology, India, and has over 20 years of architecture and technology experience across diverse industry segments with a strong focus on financial services. Dinkar can be reached at Dinkar.Gupta@cognizant.com | LinkedIn: www.linkedin.com/in/dinkargupta/.

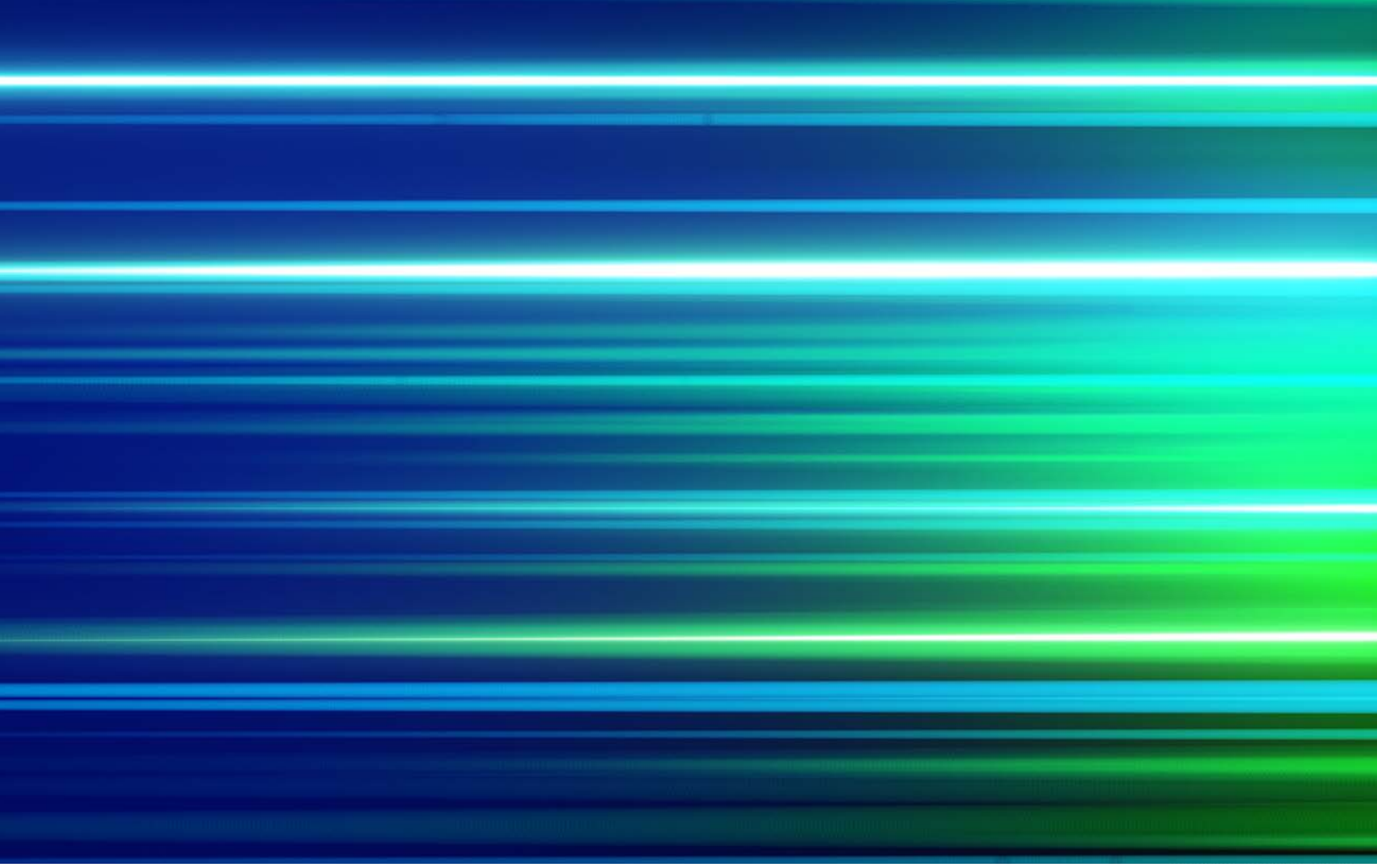
Mrudul Palvankar

Former Principal Architect, Cognizant Banking and Financial Services

Mrudul is a former Principal Architect in Cognizant's Banking and Financial Services (BFS) Technology and Architecture Office, where she previously headed the architecture and technology team assisting a global BFS client. She has a post-graduate degree in business administration from Institute of Management Development and Research, India, and has 19 years of software development experience across multiple industry segments. She is also TOGAF certified and is actively engaged in architecture transformation consulting and solution delivery for BFS customers. Mrudul can be reached at LinkedIn: www.linkedin.com/in/mrudul-palvankar-593398/.

Acknowledgments

The authors would like to thank their former colleague Vivek Kant (www.linkedin.com/in/vivekkant/) for his contributions to this white paper.



About Cognizant Banking and Financial Services

Cognizant's Banking and Financial Services business unit, which includes consumer lending, commercial finance, leasing insurance, cards, payments, banking, investment banking, wealth management and transaction processing, is the company's largest industry segment, serving leading financial institutions in North America, Europe, and Asia-Pacific. These include six out of the top 10 North American financial institutions and nine out of the top 10 European banks. The practice leverages its deep domain and consulting expertise to provide solutions across the entire financial services spectrum, and enables our clients to manage business transformation challenges, drive revenue and cost optimization, create new capabilities, mitigate risks, comply with regulations, capitalize on new business opportunities, and drive efficiency, effectiveness, innovation and virtualization. For more, please visit www.cognizant.com/banking-financial-services.

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 193 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.

Cognizant

World Headquarters
500 Frank W. Burr Blvd.
Teaneck, NJ 07666 USA
Phone: +1 201 801 0233
Fax: +1 201 801 0243
Toll Free: +1 888 937 3277

European Headquarters
1 Kingdom Street
Paddington Central
London W2 6BD England
Phone: +44 (0) 20 7297 7600
Fax: +44 (0) 20 7121 0102

India Operations Headquarters
#5/535 Old Mahabalipuram Road
Okkiyam Pettai, Thoraipakkam
Chennai, 600 096 India
Phone: +91 (0) 44 4209 6000
Fax: +91 (0) 44 4209 6060

APAC Headquarters
1 Changi Business Park Crescent,
Plaza 8@CBP # 07-04/05/06,
Tower A, Singapore 486025
Phone: + 65 6812 4051
Fax: + 65 6324 4051

Codex 5066