Next-Generation Microservices Architecture: Design Principles and Operational Strategies for Digital Transformation

Nov 2023

INDEX

dnext-technology.com

Nov 2023

Seattle
USA

İstanbul
TÜRKİYE

Dubai
UAE

London
UK

Tirana
ALBANIA

1. Introduction and Executive Summary

1.1. About this White Paper

Considering the digital transformation journey of Communication Service Providers (CSP aka Telecom Operators), from maintaining monolithic, legacy software solutions, towards managing nimble, cloud-based Lego capabilities that can scale, most operators have already invested or been investing in BSS/OSS products that assert providing the right, pragmatic and operationally manageable architectural approaches.

One of the big buzz words often used that telecom managers and decision makers come across is **Microservices**; a lot said and written about and probably one of the most selling IT words of this decade.

In this context, this White Paper provides details about DNext's top key Microservices Design principles for applying Microservices in the right way to its products; in particular **DNext**.

This document shall be considered as complimentary extension to referenced [1], [2], and [3] TMFORUM documents which lack these principles.

1.2. About DNext

DNext (the Next) is the Cloud-Native Customer Engagement, Order Orchestration and Catalog/Inventory Management Platform by DNext that comprises suite of Lego-stye modules where each modules fulfills a specific functionality. DNext enables transformation from the present mode of silo-based BSS/OSS application systems to **the Next** future mode of platform-based execution; aligned to TMFORUM's (ODA) Open Digital Architecture (Ref [1]), which offers an industry-agreed blueprint and common terminology providing pragmatic transformation paths from monolithic, legacy software solutions, towards scalable, cloud-based platforms that can be orchestrated with artificial intelligence-driven analytics.

DNext product suite is positioned to play a vital role within the Digital Transformation strategies of the next generation Communication Service Provider (aka Telecom Service Operator) with its industry standards-compliant, cloud-native, and ready to scale Lego-style components, with each component exposing TMFORUM Open APIs (Ref [2]) resulting in a model-driven Lego-style set of capabilities ready to be orchestrated and integrated in line with the business processes of the service operator.

1.3. What is "Microservices"?

One of the most repeatedly referred definitions of Microservices is:

"… an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API…", Martin Fowler.

Even though correct, the above definition is not complete and moreover may even be misleading resulting in wrong assessment, choice of systems or even in wrong decision making. This is where a well-defined set of principles are critical while applying the concept.

1.4. Why correct assessment and application of “Microservices Design Principles” are vital?

When done properly, and applied in the right context, Microservices is the collection of practices and approaches, ***that impacts not only software development processes,***

but impacting all of; organization, architecture, software development, testing and deployment aspects of the application development and delivery lifecycle.

Consider the following cases of various applications/systems each claiming to have applied microservices. The critical Microservice principles to be detailed in next section are highlighted as **“pink”** in the following:

Focus on one Task & Aligned with a bounded Context

- An application composed of independent Microservices (modules) with **Focus on one** Task and that communicate over well-defined APIs, will still not scale **businesswise or transaction wise** if decomposed solely from technical perspective.
- Applying even simple business changes can become challenging and excessively expensive for applications broken down into Microservices that are not aligned with a bounded context. This implies that for effective implementation, these Microservices should correspond to **a single cohesive part of a larger business context**.
- As an example; **Applications that are designed as a suite of many fine grained technical services, cannot be called microservices driven since benefits do not yield and a simple change on business scope will not fit the shape of the product structure resulting in slow delivery and high costs.**
- When Microservices (modules) are properly aligned with the business's bounded contexts, they can be managed by small, autonomous teams. This alignment enables executive managers and businesses to scale their functional delivery throughput according to business demands by adding more independent teams. However, this approach alone may not always be sufficient.

Autonomous, Independently deployable & Loosely Coupled

- Even when above listed aspects are addressed, assuming each module exposing its functionality via services over well-defined APIs and aligned with a bounded context, if the modules are not properly designed/developed to decouple/translate internal model of things from external API exposed model of things, if modules lack independent testing and deployment capabilities, this results in a situations where still development and/or operations wise teams and modules become dependent on each other. So modules, must also satisfy the rules of engagement considering these principles; Autonomous, Independently deployable & Loosely Coupled.

Error Handling Design for Failure, Logging and Monitoring Metrics

- Still, in spite of all the above points considered, it is extremely critical to consider that Microservices approach results in a **distributed systems architecture** way more complex than non-distributed/monolithic systems. A system that cannot be monitored, that cannot be operationally fully kept under control with quickest possible root cause analysis of real-life problems cannot be launched. Thus, modules must also satisfy the principle; **Error Handling Design for Failure, Logging and Monitoring Metrics**

Unfortunately, often management realizes all or some of these problems at a a very late stage and sometimes even after deploying apps into production that may result in refactoring, rework, loss of time and money.

1.5. Wrap-Up

In a nutshell,

- Yes, “microservices” is probably the most popular organizational, architectural approach to developing applications and software today that was put forward based on the lessons learned (out of real life bad examples based on purely service oriented architectures/approaches to software development that did not consider other aspects of IT and enterprise organizations, which microservices intent to do)
- Yes, it's extremely effective, but only when applied properly
- Yes, it's the approach used by many of the most successful companies in work in the world, particularly the big web companies
- And, as a result many enterprise organizations and teams attempt to adopt it.

Microservices are a type of **architecture used in distributed systems**, which are inherently more complex than traditional, non-distributed (monolithic) systems.

In addition to the cost overhead of developing distributed systems, with multiple teams, there are all sorts of problems that organizations automatically buy into when they start distributing computing across multiple different devices, developed by different teams and apply microservices architectural breakdown approaches in the wrong ways.

Thus, when microservices applied wrong, the result in operational and organizational reality may easily turn into:

- miss both the value and the cost benefits of microservices
- end up doing something that really isn't microservices at all, plus not even as agile as monolithic
- and even not operationally manageable as monolithic

2. Microservices Design Principles

The following is a list of Microservices Design Principles applied by DNext and to be detailed in this section:

- Focus on one task and do it well (aka Single Responsibility)
- Aligned with a bounded context
- Autonomous
- Independently deployable & Loosely Coupled
- Error Handling Design for Failure, Logging and Monitoring

Important Information:

Even though typically, Microservices term is typically used to denote the distributed systems architectural style and approaches to design applications as a suite of loosely coupled, independently deployable modules.

In this section, Microservice term is used from a more tangible operational perspective to denote a single independently deployed module running in its own process, exposing its functionalities over lightweight communication mechanisms; via http-REST web services or via reactive message driven approaches. Each of these Microservices fulfill a specific business capability and constitute the application.

2.1 Focus on one task and do it well (aka Single Responsibility)

From the outside, the Microservice must accomplish one task. In this context, task shall not be misinterpreted as if it is a single/technical low-level atomic functionality. Otherwise, the result will be overly coarse grained or overly atomic Microservices:

- **The Good:** As an example task may refer to “party management”. In the technical details, a party may be individual or organization. In this case, a relevant **party management microservice** will expose (when viewed from outside):
  - create, search&list, get by id, update, delete **individual**
  - create, search&list, get by id, update, delete **organization**

- **The Bad-Coarse Grained:** Referring to the previous task “party management”, a bad example would be to have a **party management microservice** that implements all of the following, resulting in a coarser grained Microservice. The danger is pollution of the module and a rotting design risk as a single module contains non-cohesive entities, but still implying risk of instability to other non-related entities service logic; violation of separation of concerns.
  - ✦ create, search&list, get by id, update, delete **individual**
  - ✦ create, search&list, get by id, update, delete **organization**
  - ✦ create, search&list, get by id, update, delete **customer**
- **The Bad-Fine Grained-Too Atomic:** Referring to the previous task “party management”, a bad example would be to have multiple microservices; one for individual and one for organization, which will result in finer grained Microservices. The danger is too atomic, more difficult to be used/deployed modules, and code duplication due to cohesive entities being separated.

It shall be noted that implementation logic regarding the single task of concern for the microservice may involve many other concerns, internal checks, validations, data representations, transformation of exposed API contract entities to internal entities, getting services from other microservices in the ecosystem and such, but from the outside the microservice is focused on accomplishing one task/business function and doing it well.

2.2 Aligned with a bounded context

Most teams misinterpret this idea, probably due to not knowing that the roots of the original idea comes way before Microservices became popular. “Aligned with a bounded context” idea is based on Domain Driven Design by Eric Evans, that describes modeling/breaking down the problem domain as an approach to designing software, which would naturally yield to components aligned each with a bounded business context. Thus, when Microservices aligned with a bounded context, particular terms and rules apply in a business consistent/aligned way, (which is part of the problem domain) and thus makes it a cohesive unit.

- **The Good**: when Microservices aligned with a bounded/business context, magically this acts like as if each Microservice boundary becomes a natural fire break in the forest (metaphor for the full application). Since, software aligned with such fire breaks, Microservices will end up in naturally being less coupled and easy to change, because the problem domain (aka business) is also less coupled in those borderlines. And since it is the business lines generating change requests, when a change request comes, less modules will be impacted. In other words; the Microservices will have one reason to change.
- **The Bad:** if the application is designed to break a problem into a collection of fully reusable small modules each exposing web services (to optimize reuse of technical, non-business meaningful capabilities) that doesn't really qualify as Microservices since not aligned with a bounded context. In such applications; each new business request will become harder and more costly to implement, because the scope of business change will not align/match the shape/structure of the software application. (Please refer to [4] section 2 for details). Thus a simple scope of change by the business, may end up in unexpected high number of modules being impacted.

2.3. Autonomous

Autonomous principle ensures making any changes and verifying these changes to Microservices independently.

One of the greatest values of Microservices approach based systems is that the teams that maintain and implement changes on Microservices can progress alone without the need to interact with other teams (aka without depending on other Microservices)

Most teams also partially misinterpret this idea and claim they can change the implementation of their Microservices without needing to coordinate with other teams provided that either the new exposed API is backward compatible and extends the existing or the new change impacts the internal service logic and not the interface contract. So, they claim this as proper development of Microservices.

But still, what seems OK on paper is not enough. Autonomy must include all aspects of Microservice development including the autonomy of unit /build testing.

- **The Good:** when Microservices depend on other Microservice(s) as part of fulfilling its service logic or other Microservice(s) depend on the changed Microservice(s), when Behavior Driven Development (BDD) properly applied, with all automated unit tests properly developed not compromising autonomy, then this means, the module can be BOTH developed and verified autonomously, making the module autonomous considering continuous integration and continuous deployment.

- **The Bad:** Teams are developing Microservice(s) independently, but due to lack of properly applied BDD, lack of proper independent unit testing procedures being in place, teams are build testing together.

2.4. Independently deployable & Loosely Coupled

Whereas Autonomous principle ensures making any changes and verifying these changes to Microservices independently, Independent deployable principle ensures deploying these changes independently.

This principle, also requires relevant organizational support as designing and implementing changes in such ways, keeping backward functionality, without any assumptions on the timing of deployment of components etc. is one of the biggest extra cost factors of Microservice approach based system design and development.

- **The Good:** Consider a product such that; Microservices are developed taking the separation of external and internal representation of entity models extremely seriously; time, effort and cost invested to keep the separation clean including when interacting with the dependent Microservices as part of impacted Microservice fulfilling its service logic. Changes impacting new APIs developed (unless a major independent product version) extending old API versions in a backward compatible manner with additional techniques applied such as API versioning. Interactions with dependent microservices developed always via transforming internal representation of entity models to that dependent API with required capability to support the also multiple versions of that Microservice with proper error handling. Then, such Microservices will be Loosely Coupled and it will be possible to deploy them independently.

- **The Bad:** Consider a product such that; Microservices are not developed taking the separation of external and internal representation. Any change that is required to fulfill an internal service logic must be reflected to external API since same entity model objects used both for external API and internal service logic. Dependent Microservice APIs are called without any adapters and entity transformation with poor error handling. Then, such Microservices will not be Loosely Coupled and it will not be possible to deploy them independently and even in some cases deployment must ensure system downtimes and a certain sequence of deploying of these new Microservices

2.5. Error Handling Design for Failure, Logging and Monitoring Metrics

Microservices is a distributed systems architecture, and the fact is distributed systems are way more complex than non-distributed/monolithic systems.

In addition to the cost overhead of developing distributed systems, with multiple teams, there are all sorts of problems that organizations automatically buy into when they start distributing computing (deploying suites of Microservices) across multiple different devices, developed by different teams.

In such cases, even though all the above listed principles are properly applied, it crucial to have the proper common standards while implementing Microservice components considering Error Handling Design for Failure, Logging and Monitoring Metrics.

- **The Good:** Even though microservices running in their own process and independently deployed, assume a product such that; Microservices are developed with proper error handling, generating relatable Logs that provide the necessary metadata such; application, channel, session-id, transaction-id, business-interaction-id etc. and metrics data such as; Min/Max/Avg transactions/ service calls per sec, Min/Max/Avg response times, Alerts when certain thresholds reached etc. Then the following are ensured by design:

- **Proper Error Handling Design For Failure**

  A. **Resilience:** Effective error handling ensures that a failure in one service does not lead to a cascading failure across the entire system resulting in end2end system resilience when unexpected conditions arise or when new deployments/updates being made on a live system

  B. **User Experience:** Well-designed error handling provides users with clear and actionable feedback, enhancing the overall user experience.

  C. **Troubleshooting and Debugging:** Proper error handling is the prerequisite with proper logging making it easier to identify and diagnose issues, allowing for quicker resolution and root cause analysi

- **Proper Correlated Logging**

  A. **Live System Debugging:** Logs are invaluable for identifying and diagnosing issues in a distributed system. Correlated Logs, provide a trail of execution over multiple Microservices involved in the business context that can be used for troubleshooting and root cause analysis.

  B. **Auditing and Compliance:** Logs can be used for auditing purposes to track user actions or system events for compliance with regulatory requirements.

  C. **Performance Monitoring:** In some specific cases, in addition to Monitoring Metrics, Log data can be used to monitor specific system performance, identify bottlenecks, and optimize resource allocation of specific functionality used as part of processing logic of the Microservice.

- **Proper Monitoring Metrics**

  A. **Performance Optimization:** Metrics provide insights into the performance of individual Microservices, allowing for optimization and resource allocation.

  D. **Capacity Planning:** Metrics help in capacity planning by providing data on resource utilization, which is essential for scaling Microservices as needed.

  E. **Alerting and Automation:** By setting thresholds and alerts based on metrics, teams can be notified of potential issues before they escalate into critical failures.

- **The Bad:** An application comprised of many Microservices without a common binding skeleton of Error Handling Design for Failure, Logging and Monitoring Metrics, will NOT be by any means operationally manageable and when deployed under load and while maintaining the lifecycle of the application in-production will result in a catastrophe.

3. Verdict

When microservices applied wrong, without considering the principles listed in previous section, the result is missing both the value and the cost benefits of microservices and even not operationally manageable as monolithic.

When Microservices Design Principles detailed in previous section applied properly, organization will have the following benefits:

- Horizontal Scalability of Business Value Delivery
- Horizontal Up/Down Scalability of Independent Development Teams
- Modularity
- Resilience
- Performance Scalability
- Availability and Redundancy
- Faster Time to Market - Continuous Integration/Deployment
- Fast Fault Isolation/Recovery and Quick Root Cause Analysis
- Cost Effective Maintenance and Change Management

References

[1] TMFORUM, IG1167 ODA Functional Architecture, Ver.5.0.1, Jan 31, 2020, [PDF], Available: https://www.tmforum.org/resources/ [Accessed 30 May 2020].

[2] TMFORUM, Open API Table, "TMFORUM Open APIs Production Versions", [Online]. Available: https://projects.tmforum.org/wiki/display/API/Open+API+Table [Accessed 20 June 2020].

[3] *TMFORUM, TMF630 REST API Design Guidelines Part 1 to 6 Ver.4.0.0, May 2020, [PDF], Available: https://www.tmforum.org/resources/ [Accessed 30 May 2020].*

[4] *Clean Architecture, Robert C.Martin, 2018*