Deloitte.

Digital modernization 2021

A tech accompaniment piece
May 2021

A successful digital modernization incorporates business drivers, team topologies, and technical aspects. Thus, any modernization journey immediately becomes a multifaceted, cross-organizational effort that affects anyone within an enterprise. In this report, we discuss some of these aspects and detail the important role they play in digital modernization. We selected topics ranging from team organization and application portfolio management to software architecture and deployment to provide a holistic view of modernizing an enterprise's application landscape.

This paper discusses selected aspects from digital modernization. For a more general point of view see www.deloitte.com/us/digitalmod-report-2021.

Autonomous teams

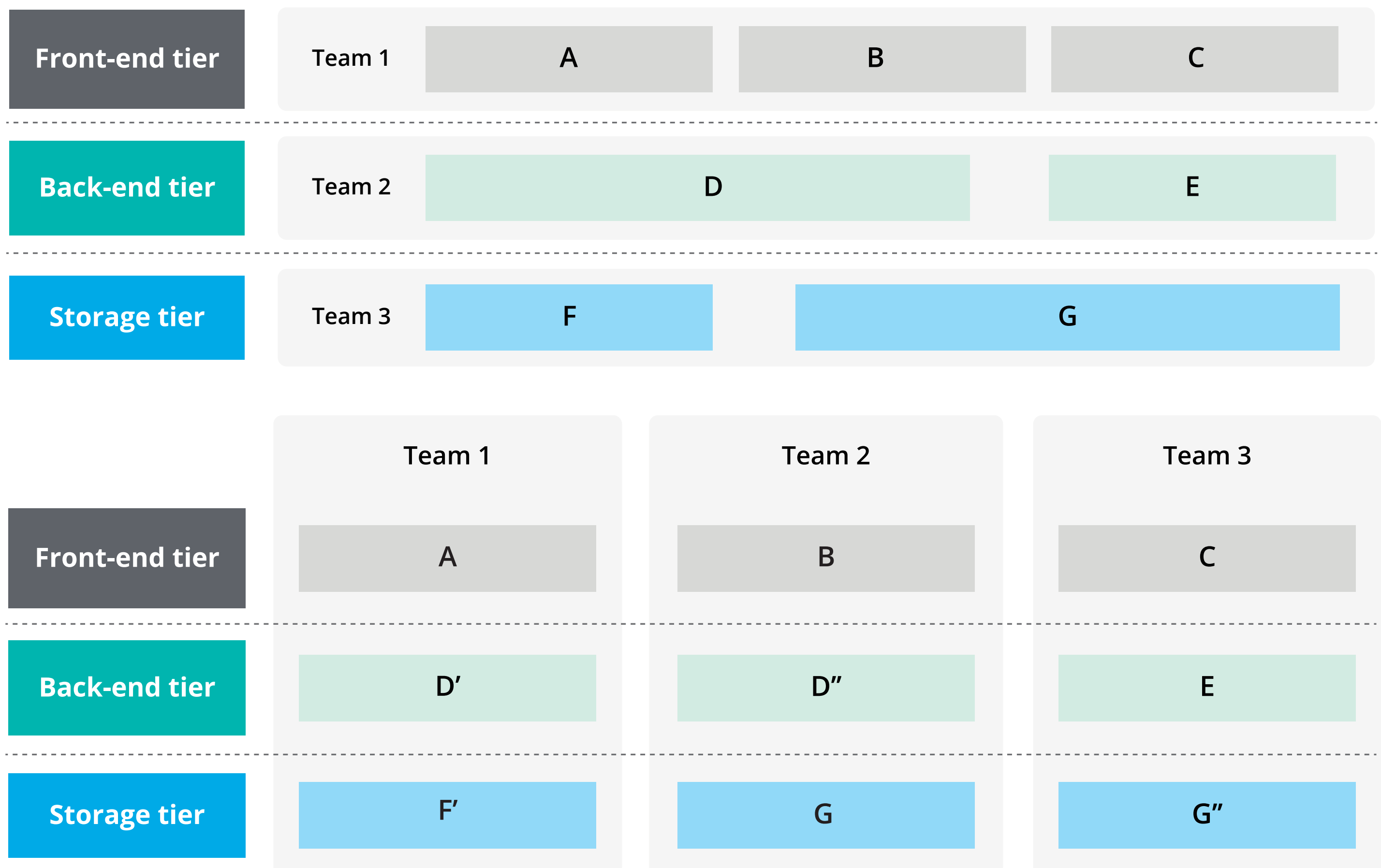
Team structure is key to successful software development. Conway's Law from 1968 states that you will find a copy of an enterprise's communication structure in its application code. More than 60 years later, this still holds true. Recently, Skelton and Pais[1] have shown that modern team structures immediately lead to sustainable software architectures: Autonomous teams create so-called team APIs, keeping complexity within the domain, the team, and the code, away from the rest of an organization.

This change in organizational culture—organizing teams vertically along business functionality rather than horizontally by technical expertise, as shown in figure 1—fosters business value as well. There are several examples demonstrating the benefits of matching software architecture and team structure, such as at Amazon AWS[2] or Spotify.[3]

Adersberger and Siedersleben[4] come to the same conclusions. They state that developing cloud-native applications based on microservices leads to organizational, methodological, and technological changes for a client, which can be a major roadblock if done incorrectly. However, this can be mitigated by actively investing in development and management know-how.

The relationship between modernizing applications and implementing organizational changes might not be visible initially. However, it is a crucial part of modernization projects that must be addressed in early phases. Otherwise, acceptance of a project may suffer, and willingness to contribute may degrade. The software industry has developed powerful tools to introduce modern development methodologies in legacy mainframe environments as well, offering the benefits of a modern development environment even in the early phases of a modernization process. Many successful implementations of these tools are found throughout client installations Deloitte has delivered.

Figure 1. Horizontal (top) and vertical (bottom) team structures[5]



Application Portfolio Matrix

One key aspect of developing a client's journey to the cloud is determining how valuable applications are. For this, Deloitte uses the Application Modernization Cloud Adaption Framework (figure 2). It evaluates how certain parts of a client's application portfolio can be transitioned into cloud-native environments, keeping or even increasing the value of an application. An important stage of the journey is to determine the contribution to business value (both day-to-day and strategic) using the Cloud Adaption Framework.

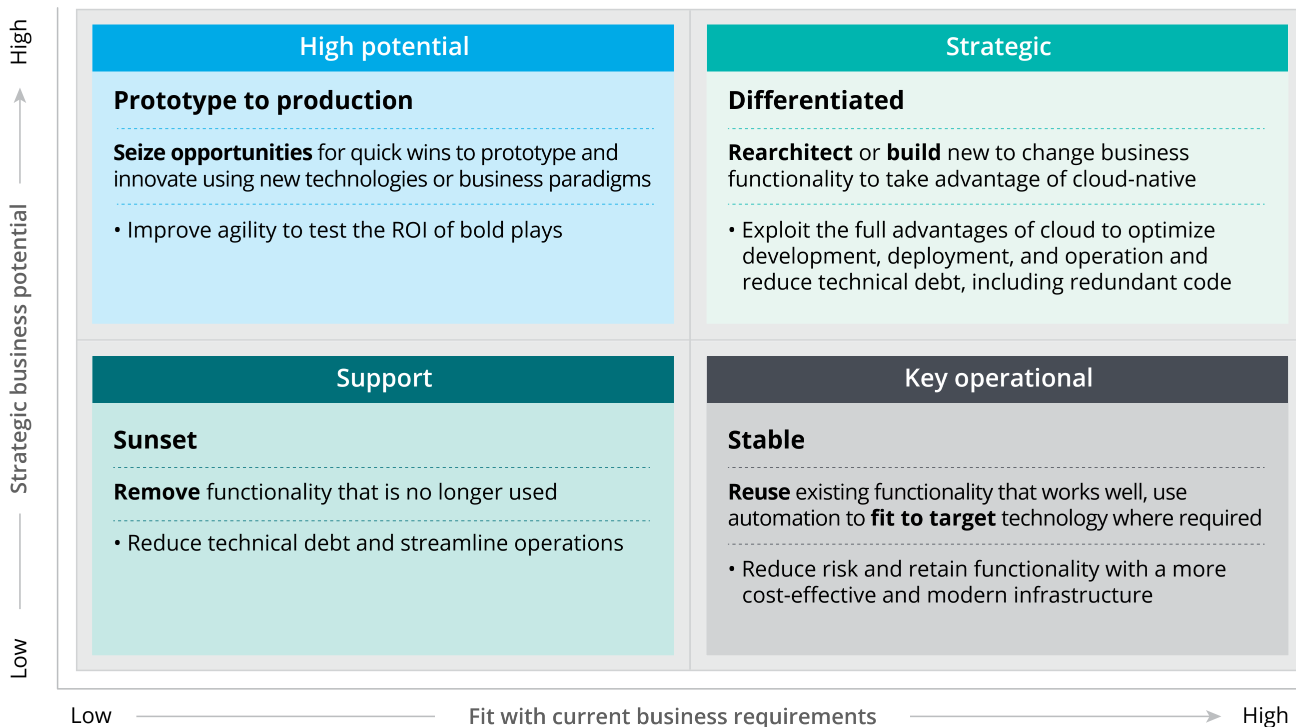
The foundation for this framework is the Application Portfolio Matrix (figure 3) as introduced by Ward and Peppard.[6] Based on the individual contribution, an application can be mapped to four quadrants:

- Strategic applications (high strategy, high value contribution) are the essential drivers of the current and future business.
- The high potentials (high strategy, low value contribution) play a pivotal role in the future of the business. Often, these applications are merely experimental and thus are considered tryout projects, internal proofs of concept, or in beta. Some of them may also be dead-end.
- Key operational applications (low strategy, high value contribution) keep the current business going. All business functions, such as billing systems, ERP systems, or CRM systems, must work reliably, as they are responsible for the foundation of the enterprise.
- Supporting applications (low strategy, low value contribution) are those supporting non-differentiating business, such as payroll. Applications in this quadrant must be efficient and cost-effective.

The matrix helps determine the overall business value of an application. It provides a first impression of how the value of the overall application can be increased. Ward and Peppard suggest the following procedures based on the quadrants—also as found in the Deloitte framework:

- Applications in the support quadrant are candidates for divestment, cost reduction, or increasing efficiency. Relevant functionality can be removed, and the remaining functionality is either no longer required (and thus subject to divestment) or can be replaced by commercial off-the-shelf (COTS) software.
- High potentials are also candidates for cost reduction; however, their potential must be evaluated on a regular basis. If the potential is still high, there must be a transformation toward strategic, or the application is divested. As an example, such a transformation can consolidate the valuable part of the application into an existing, strategic application.
- For all strategic applications, the focus is on maintaining or increasing technical health (for example, by refactoring or rearchitecting the application). Typically, strategic applications are constantly

Figure 2. Application Modernization Cloud Adaption Framework

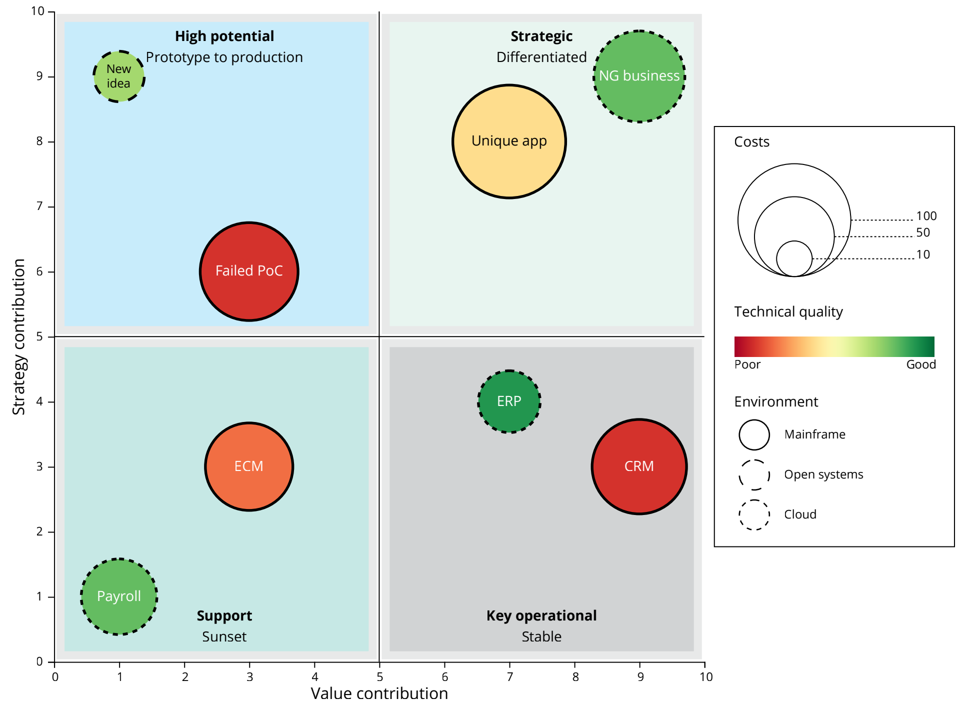


enriched with new functionality or are targets for ingesting functionality from other quadrants.

- Finally, key operational applications should focus on reliability and availability, and are thus candidates for moving toward a resilient and scalable infrastructure, as well as maintaining technical health.

In the field, a few approaches exist to assist in creating this matrix. Some automatically capture the metrics required for deriving the technical health of an application. Deloitte's solution, Application Modernization powered by innoWake™, is an automated one that has been successfully employed at an enterprise scale.

Figure 3. Application Portfolio Matrix



Underrated aspects of modernization

The relevance of some aspects of application modernization (for example, feature usage and UI modernization) tends to be underestimated. In our engagements, we also take care of those aspects when designing and implementing a cloud road map, as follows:

Reassess feature usage. When looking at custom-developed software, individual functionality constantly changes over time. However, we often see clients add capabilities without ever retiring anything unused. This leads to code bloat. Also, business processes (or certain branches thereof) can still be present, but no longer actively used (e.g., calculation routines for insurance policies with only terminated contracts). A similar experience has been shared recently by ThoughtWorks' Tech Radar.[7] They state that, according to a report published by Standish Group,[8] the majority of functionality (about 50%) is unused. For modernization efforts, they suggest giving up on feature parity between legacy and modernized architecture and investing the effort in understanding end users' needs by "conducting user research and applying modern product development practices rather than simply replacing the existing [functionality]."[7]

The role of UI modernization. A well-designed user interface (UI) contributes to the success of an application. A UI acts as the access point for an application's functionality and is hence a defining piece of the overall user experience.[9] When it comes to internal applications, however, companies all too often erroneously reduce the UI to its colors and do not see the need to invest in them, since none of its clients will ever see it. The truth is, however, that well-crafted UIs are far more than merely pleasant to look at. When done well, UIs render applications highly usable, thereby increasing employees' productivity, efficiency, and overall satisfaction.[10] The UI shapes how users perceive the application, including its ease of use, usefulness, or quality. And the more positive these perceptions are, the higher users' initial acceptance, current use, future use, and overall satisfaction with the application, all of which contribute to realizing the anticipated business benefits.[11, 12, 13] With a well-designed UI, an application becomes self-explanatory, thereby reducing the need for and associated costs of training, documentation, and support. It also reduces the need for special onboarding, as new employees usually know how to interact with recent applications. Higher satisfaction with an application (and thus with the workplace) further lowers employee turnover and, correspondingly, hiring costs.[14]

Service decomposition

Clients receive valuable insights when analyzing their applications using mining tools. An example is the so-called social graph, a color-coded visualization of dependencies between application parts. Figure 4 shows a social graph whose colors denote which software modules belong to a given application or business functionality: a codebase containing three different applications (yellow, green, and blue), as well as some technical aspects (red—in this case, a database abstraction layer).

This graph gives valuable insights into how to start decomposing into services, a crucial step of any modernization to microservices. During this step, the existing monolithic application is decomposed into business domains (i.e., fully functional units handling exactly one business). Lilienthal[5] states that, for a successful decomposition, domain-driven design (DDD) is important, as it provides significant guidance for decomposing an application into functional domains. In DDD, the functional domains are typically referred to as bounded contexts. The functional decomposition and the relationship between these contexts are the fundamental building blocks of a microservice architecture.

Within one bounded context, the notion of a concept is identical, whereas across contexts, this may vary. For example, picture the concept of a customer object across the sales, booking, and marketing departments of a travel agency. In each of these contexts, a customer may have distinguishing properties (beside some shared properties). Thus, a customer object is more a lead object in the sales context, a passenger object in the booking context, and a subscriber object in the marketing context.

Once bounded contexts are established, architects must handle cross-cutting concerns between contexts by establishing appropriate APIs (and teams). Software solutions exist to help architects and developers derive this information in well-integrated packages by using previously mined data, as well as by suggesting candidates for contexts and how service interfaces may be established between contexts. Figure 5 shows an example. Deloitte has been successfully using its innoWake™ tools to achieve these goals.

Figure 4. Social graph

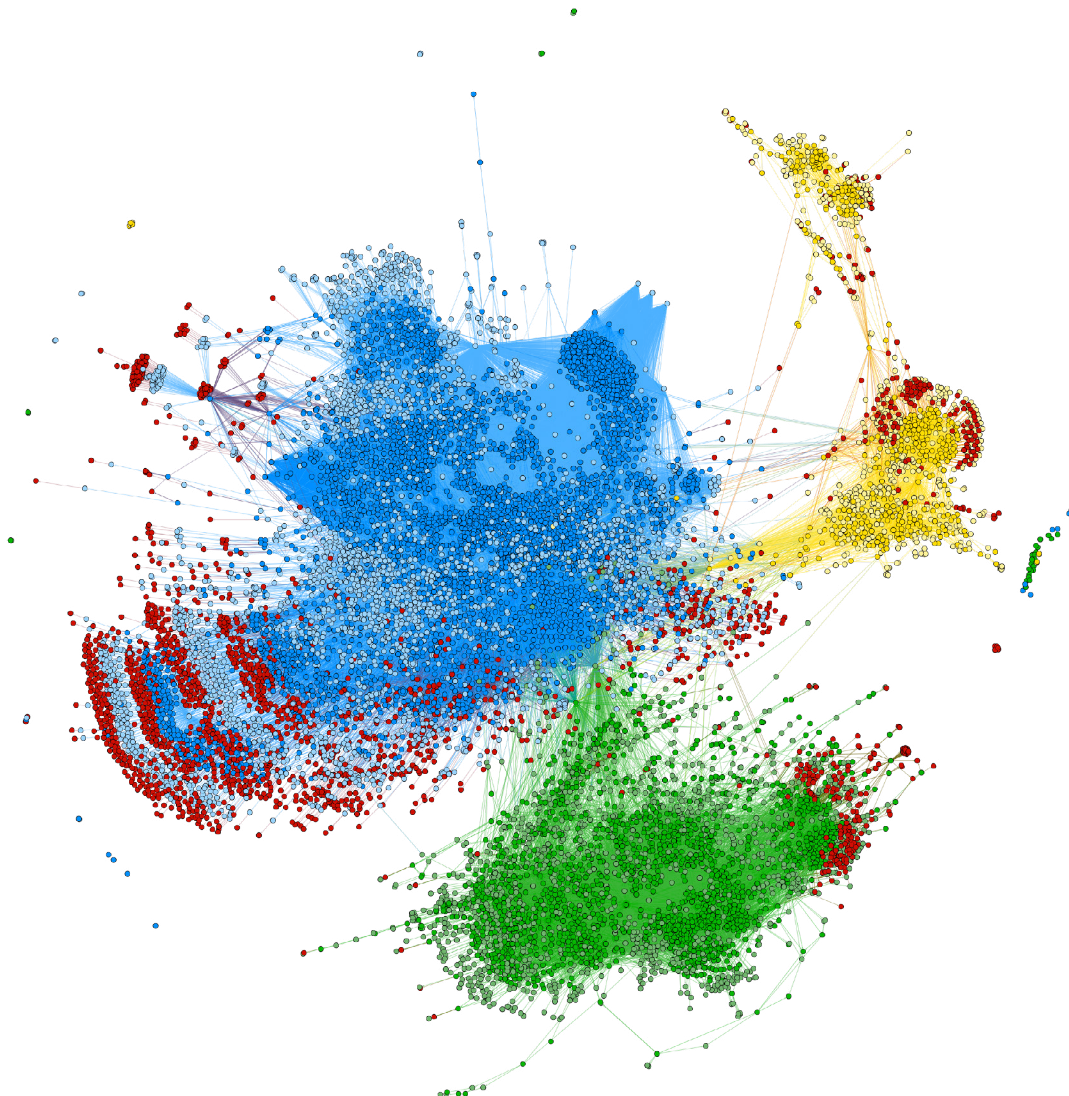
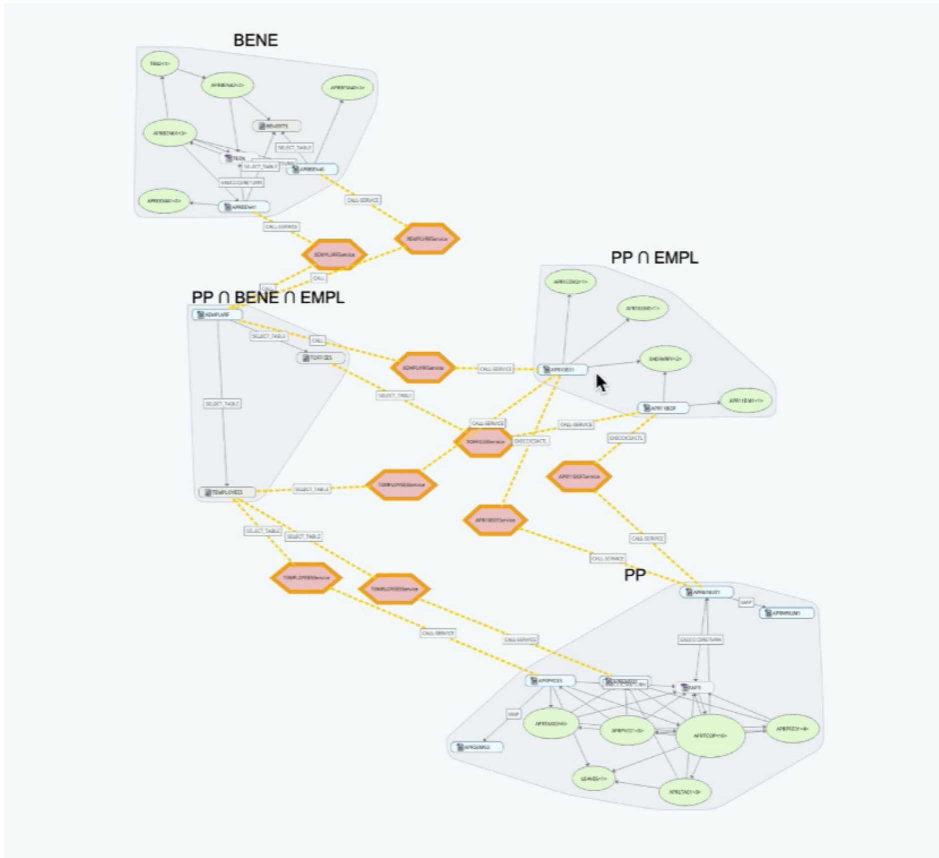


Figure 5. Monolith cutter screenshot



Decentralized data management

Because of functional decomposition, all affected data requires decomposition as well. In a typical microservice architecture, access to the data owned by the service is hidden behind the service API. Any functionality related to persistence, including the data, schema, and database access, is completely encapsulated by the service.

Database modernization incurs schema changes, data redistribution, data migrations, and other long-running operations. Furthermore, during transition to a decomposed database, there may be issues with data synchronization, transactional integrity, latency, and more. If mainframe data sources are part of the journey, complexity may increase, particularly if data models uncommon outside the mainframe world, like IMS/DB, must be migrated. Moreover, program logic in stored procedures can also be a pitfall, and there may be recommendations to remove database logic and pull the functionality out and into the service.

Experience shows that data migration can be a cumbersome task. We agree with Richardson,[15] who states that the number of data migrations should be kept as low as possible, especially with incremental modernization. His mantra, "functionality first, data last," is an integral part of any data decomposition strategy. Rather than assigning physical databases to microservices, data virtualization techniques support creating a virtual database. The physical reorganization of the data is done when the modernization reaches a steady state, justifying the efforts of costly data migration.

Micro-frontends

Further applying vertical decomposition (by also including the user interface) results in the micro-frontend architecture. Rather than having a monolithic frontend, as is typically found with single-page applications (SPA), the functional decomposition is also visible in the frontend, which forms a composite UI, as shown in figure 6.

This fosters encapsulation of service internals, resulting in an even more manageable service, as the codebase is further decoupled. Jackson[17] defines micro-frontends as "an architectural style where independently deliverable frontend applications are composed into a greater whole." He states that this style results in "smaller, more cohesive and maintainable codebases [and] the ability to upgrade, update, or even rewrite parts of the frontend in a more incremental fashion." Individual micro-frontends are integrated as parts of a user-facing container application; the integration with the backend services is done using API gateways. Figure 6 shows various solutions for implementing micro-frontends as suggested by Jackson.[17]

Software solutions provide rapid UI development frameworks that embrace the micro-frontend pattern and even allow for integration with refactored applications still employing "green-screen terminal" semantics. Deloitte has successfully refactored applications using this approach.

Event-driven microservice architecture

Microservice architectures have seen increasing adoption in recent years. According to a study by Lightstep[18] in 2018, microservices have become mainstream in enterprise applications, even though there are challenges. The main driver, according to 82% of the interviewed development stakeholders in the study, is agility. However, nearly everyone surveyed (99%) experiences daily challenges with microservices, especially troubleshooting (seen as 73% harder). The benefits and challenges of a microservice architecture have been extensively discussed by Newman.[19] He concludes that microservices are not a silver bullet, as complexity increases compared with monolithic applications.[20] However, modern frameworks like Micronaut may successfully mask this complexity.

Davis[21] considers event-first architectures as the most prominent cloud-native pattern, as they change the way software is developed. Rather than focusing on a request-response paradigm, event-driven architectures rely on the event-reaction paradigm. Here is an analog: With request-response, when you enter a room, you flip the switch (request), and a light turns on (response). In an event-first world, you enter a room, generate a "room entered" event (using a presence detector), and the light switches on (reaction). This inversion of responsibility is a fundamental paradigm shift. It provides developers and business experts a representation of the real world and allows them to model use cases for how humans think, not how technology forces them to think.

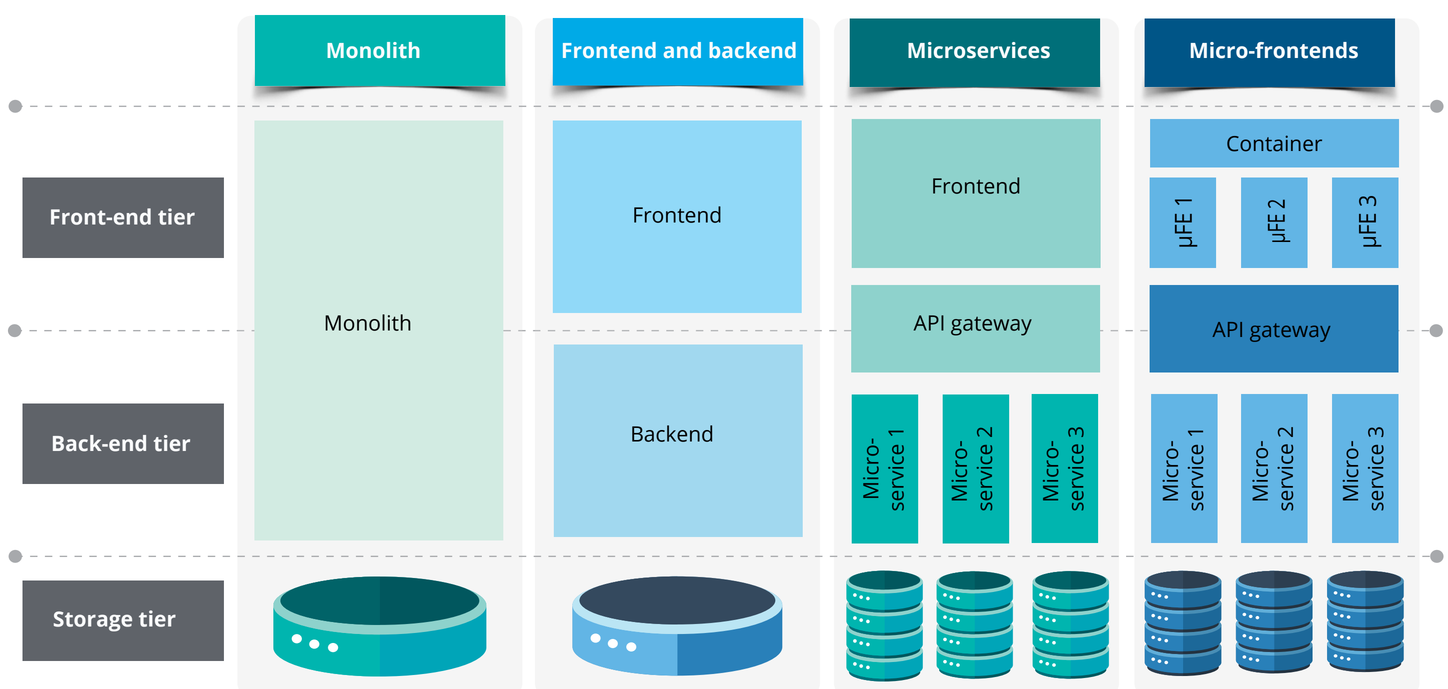
For a modernization project, the evolutionary aspects of a microservice architecture are very interesting. As service internals, including the data, are completely hidden behind a service API, the implementation can change over time. Developers can initially deploy legacy code, wrapped within an anticorruption layer or service façade, and subsequently refine the code. With such an evolutionary approach, a single service can be very quickly exposed for a short time to market.

We are embracing this evolutionary aspect in our modernization engagements. Our standardized, yet highly customizable approach uses event-driven architecture to provide the technical foundation for an iterative modernization.

Figure 7 shows an example of such an architecture. In this example, various services, each in its own architectural flavor, coexist with refactored legacy applications, as well as existing mainframe applications.

The event-driven backplane is responsible for transporting events across applications. With events being first-class citizens of such an architecture, the backplane can be used to capture an audit trail across all applications. This can be useful for regulatory compliance (an aspect that is typically spread across the whole codebase in legacy applications), as well as for logging and tracing for troubleshooting issues. Like a write-ahead log in relational database

Figure 6. Iterations of a modernization[16]



systems, the audit trail can be used to reproduce the application state at a given time, allowing for time travel across a whole enterprise application. As a side effect, services can recover from outages by replaying the audit log appropriately.

Event-driven architectures are inherently designed for integration with other applications. In the previous example, the mainframe uses change data capture (CDC), a noninvasive technique for adapting existing applications to the event-driven world.

Richardson[15] recommends using technologies like command/query responsibility segregation (figure 8) or event sourcing for modernizing monoliths to microservices. Both architectural patterns prescribe a strict split within a microservice and suggest isolating the command and query components from each other. As a rule of thumb, commands change the internal state of a microservice, whereas queries retrieve the internal state. Queries are always free of side effects; thus, they must not change the state.

The capabilities of CQRS and event sourcing can be exploited during application modernization. First, the existing monolith can still be the command part of the application, containing the business rules changing the application state. However, the query part can be implemented by a new microservice. Second, the business rules for an application are collocated within the command handler. The query part can evolve independently, as new query functionality can be added without touching the business rule. Third, when releasing a new version of the application, the internal state can be built up by replaying the event store rather than migrating data. This is handy if the persistence schema changes within two subsequent versions of a service.

Industry specialists have developed solution portfolios that contain several tools to facilitate the implementation of event-driven architectures. Deloitte's innoWake Legacy DevOps provides middleware and adapters for integrating mainframes. With innoWake Discovery and Mining solutions, we use automated diagnostic tools to understand and capture information, while innoWake Transformation provides solutions for rapidly moving legacy code to modern programming languages.

Serverless deployment for microservices

Castro et al.[22] define serverless as follows: "Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running." They also state that developers using serverless computing can get cost savings and scalability without requiring a high level of cloud computing expertise. It allows developers to focus on developing business logic and lets the cloud provider scale the computing and storage resources. The cost savings are incurred as payment is only required for consumed computing power, in contrast to preallocated virtual machines. A study by Kilcioglu et al.[23] claims significant overprovisioning in such preallocated environments, resulting

Figure 7. Architecture overview

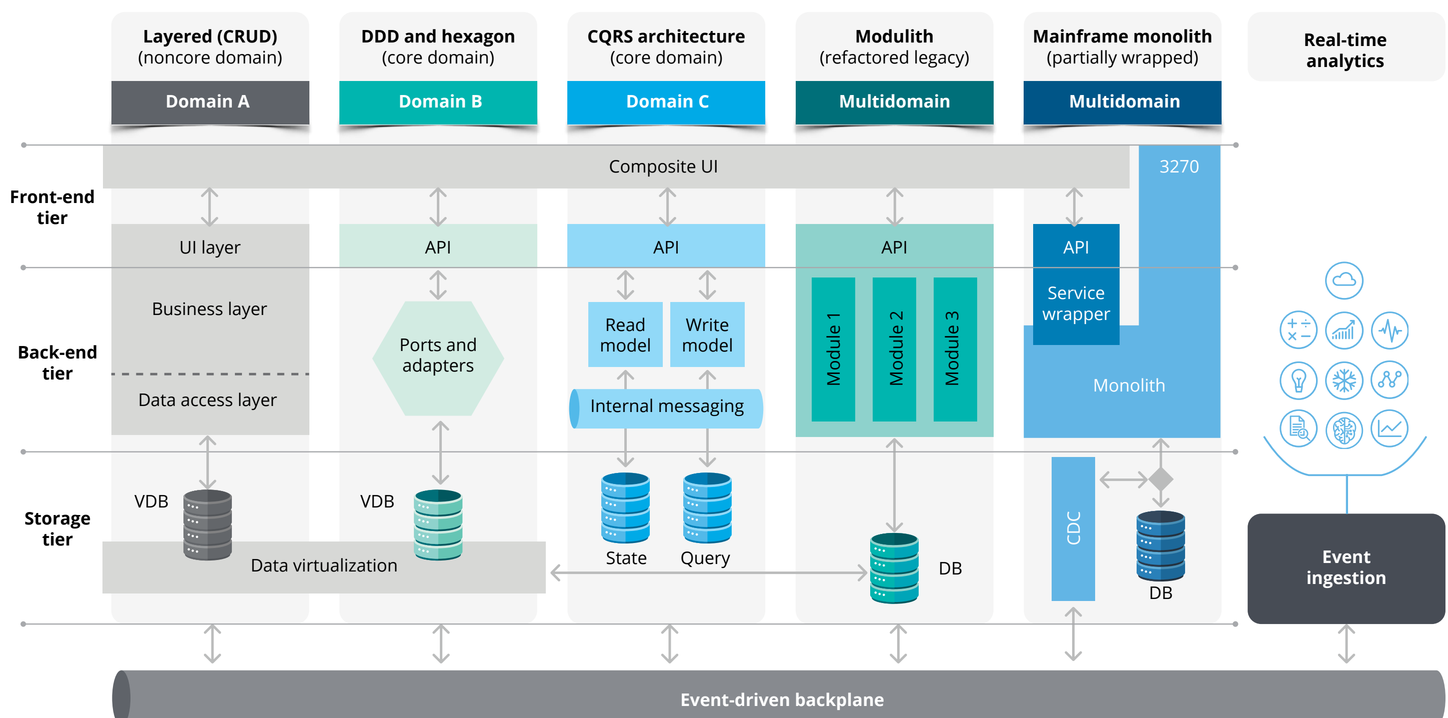
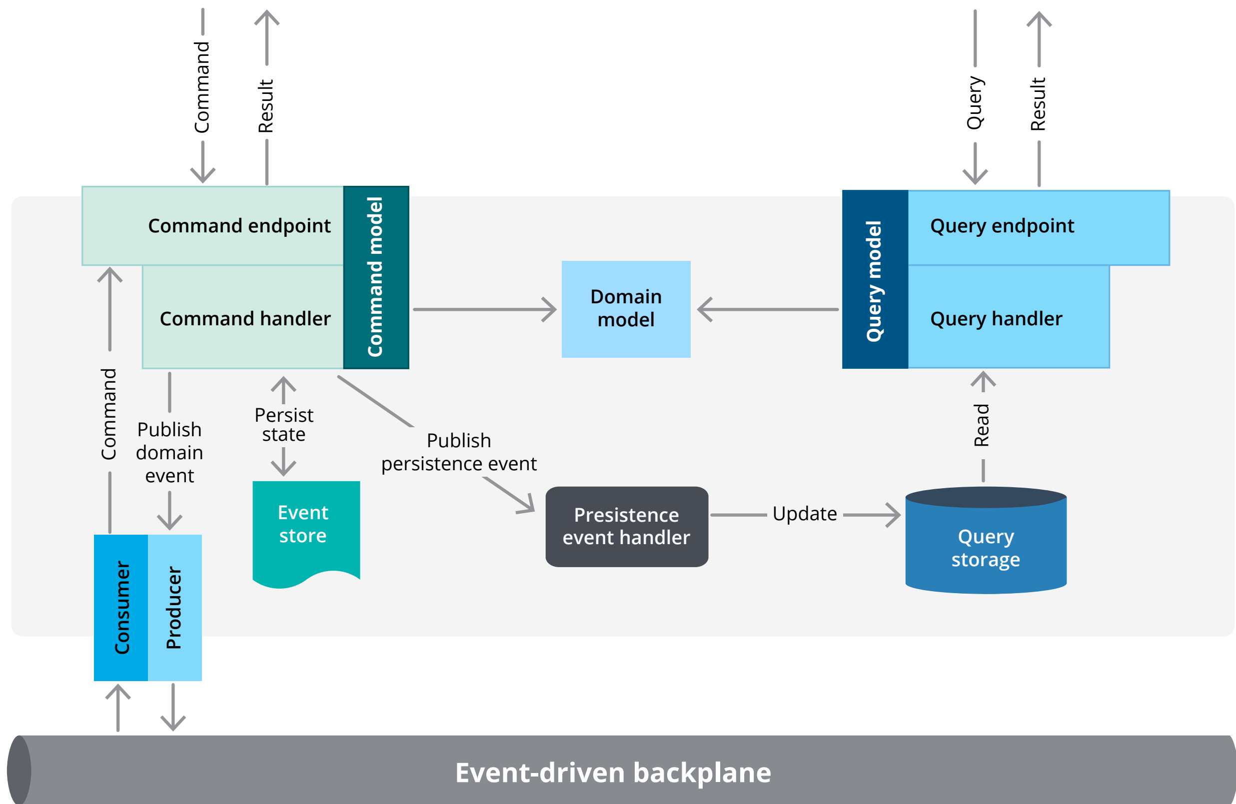


Figure 8. Microservices with CQRS and event sourcing



in unused, but paid computing power. Serverless computing allows for true pay-as-you-go pricing, where currently unused applications do not incur any costs.

In other words, serverless deployment is one of the prominent techniques of a cloud-native technology stack. According to Adersberger and Siedersleben,[4] a decent cloud-native technology stack "abstracts away the complexity of a cluster by making it look like one single, huge machine." Thus, cloud vendors provide instant resource availability based on workload demands. The workloads can be distributed based on expected resource utilization, leveraging elastic environments that allow for dynamic adaption to increasing or decreasing resource requirements. Elastic environments are still a form of preallocated environment, as they are still initially scaled for the average load. As computing instances for serverless applications are dynamically scheduled by cloud providers, the microservices must be truly stateless—a best practice already devised by Hamilton[24] in 2007.
With the advent of serverless computing, a real pay-per-use model can be employed. If an application is currently not required, it immediately gets "scaled to zero," incurring no cost for computing power. Castro et al.[22] compiled a list of projects where the transition from virtual-machine-based environments to cloud-native, serverless computing significantly drops the overall costs of an application; the highest cost savings achieved was 94%.

The serverless paradigm is currently extended to other cloud-managed services; the majority of cloud providers are providing serverless storage and serverless databases as integrated parts of their cloud-native technology stack.

Conclusion

Digital modernization is not a sprint, but a marathon. A multitude of aspects must be addressed, and a lot of decisions must be made. Thus, knowing the pitfalls and addressing them early reduces the overall risk of a modernization journey. This paper points out some common elements of large modernization, but as each environment is unique, specialties must be known and considered. Consequently, the most crucial step of a modernization story is to carefully assess the existing environment and plan a sound modernization strategy. This way, you can burn the phoenix of legacy applications so it may rise from the ashes as a revival of your business in the cloud.

Endnotes

1. M. Skelton and M. Pais, *Team Topologies* (Portland: IT Revolution, 2019).
2. K. Lane, "The Secret to Amazon's Success: Internal APIs," Apievangelist, January 12, 2012.
3. H. Kniberg and A. Ivarsson, *Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds*, October 2012.
4. J. Adersberger and J. Siedersleben, "The Cloud Native Stack: Building Cloud Applications as Google Does," in *Digital Marketplaces Unleashed* (Berlin: Springer, 2018), pp. 711–713.
5. C. Lilienthal, "From Monoliths to Modular Architectures and Microservices with DDD," JAX London, September 30, 2019.
6. J. Ward and J. Peppard, *Strategic Planning for Information Systems* (Chichester: John Wiley & Sons Ltd, 2002).
7. ThoughtWorks, Inc., "ThoughtWorks Radar Vol. 21," November 2019.
8. The Standish Group International, Inc., *Exceeding Value*, August 25, 2014.
9. D. Norman and J. Nielsen, "The Definition of User Experience (UX)," NN Group, accessed April 2020.
10. J. Anderson, J. McRee, and R. Wilson, *Effective UI: The Art of Building Great User Experience in Software*, O'Reilly Media, Inc., 2010.
11. F. D. Davis, R. P. Bagozzi, and P. R. Warshaw, "User Acceptance of Computer Technology: a Comparison of Two Theoretical Models," *Management Science* 35, no. 9, (1989): pp. 982–1003.
12. W. H. Delone and E. R. McLean, "The DeLone and McLean Model of Information Systems Success: a Ten-year Update," *Journal of Management Information Systems* 19, no. 4 (2003): pp. 9–30.
13. L. Deng, D. E. Turner, R. Gehling, and B. Prince, "User Experience, Satisfaction, and Continual Usage Intention of IT," *European Journal of Information Systems* 19, no. 1 (2010): pp. 60–75.
14. J. Ross, *The Business Value of User Experience*, Infragistics, January 2014.
15. C. Richardson, *Microservices Patterns* (Shelter Island: Manning, 2019).
16. F. G. Diaz, "Micro-frontends Using Vue.js, React.js, and Hypernova," Medium, February 28, 2019.
17. C. Jackson, "Micro Frontends," Martin Fowler, June 19, 2019.
18. Lightstep, "Global Microservices Trends Report: A Survey of Development Professionals," Lightstep, April 2018.
19. S. Newman, *Building Microservices: Designing Fine-Grained Systems* (Sebastopol: O'Reilly, 2015).
20. S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* (Sebastopol: O'Reilly, 2019).
21. C. Davis, *Cloud Native Patterns: Designing Change-Tolerant Software* (Shelter Island: Manning, 2019).
22. P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The Rise of Serverless Computing," *Communications of the ACM* 62, no. 12 (2019): pp. 44–54.
23. C. Kilcioglu, J. M. Rao, A. Kannan, and R. P. McAfee, "Usage Patterns and the Economics of the Public Cloud," in Proceedings of the 26th International Conference on World Wide Web, Perth, 2017.
24. J. Hamilton, "On Designing and Deploying Internet-Scale Services," USENIX Association, November 11–16, 2007.

Contacts

Thorsten Bernecker
Principal
Deloitte Consulting LLP
+1 512 226 4418
tbernecker@deloitte.com

Stefan Aulbach
Lead architect
Deloitte Consulting LLP
+49730792190139
saulbach@deloitte.com

Arne Gerhard
Senior manager
Deloitte Consulting LLP
+1 214 840 1887
agerhard@deloitte.com

This publication contains general information only and Deloitte is not, by means of this publication, rendering accounting, business, financial, investment, legal, tax, or other professional advice or services. This publication is not a substitute for such professional advice or services, nor should it be used as a basis for any decision or action that may affect your business. Before making any decision or taking any action that may affect your business, you should consult a qualified professional adviser.

Deloitte shall not be responsible for any loss sustained by any person who relies on this publication.

About Deloitte

Deloitte refers to one or more of Deloitte Touche Tohmatsu Limited, a UK private company limited by guarantee ("DTTL"), its network of member firms, and their related entities. DTTL and each of its member firms are legally separate and independent entities. DTTL (also referred to as "Deloitte Global") does not provide services to clients. In the United States, Deloitte refers to one or more of the US member firms of DTTL, their related entities that operate using the "Deloitte" name in the United States, and their respective affiliates. Certain services may not be available to attest clients under the rules and regulations of public accounting. Please see www.deloitte.com/about to learn more about our global network of member firms.

Copyright © 2021 Deloitte Development LLC. All rights reserved.