Migrating Enterprise Legacy Source Code to Microservices

On Multitenancy, Statefulness, and Data Consistency

Andrei Furda and Colin Fidge, Queensland University of Technology

Olaf Zimmermann, University of Applied Sciences of Eastern Switzerland, Rapperswil

Wayne Kelly and Alistair Barros, Queensland University of Technology

// Microservice migration is a promising technique to incrementally modernize monolithic legacy enterprise applications and enable them to exploit the benefits of cloud-computing environments. This article elaborates on three challenges of microservice migration: multitenancy, statefulness, and data consistency. //

MODERNIZATION OF LEGACY enterprise systems is a challenge faced by many organizations that want to exploit the new cloud-computing technologies to meet high-scalability and high-availability needs. Assuming that the legacy system's source code is available and maintainable, microservices are a promising solution in which centralized services are reimplemented as multiple independent services[1,2] (see Figure 1). Microservices support incremental modernization, leading to highly scalable systems[3] with high availability through redundancy of service instances and reduced costs. Microservices also facilitate the low-risk, small-scale incremental modernization that is often preferred to large-scale approaches.[4]

In this article we explain three closely related challenges of microservice migration: multitenancy, statefulness, and data consistency. (For a synopsis of these three challenges, see the sidebar.)

Multitenancy

A multitenant software-as-a-service application fulfils the needs of multiple groups of users, organizations, or departments. The application-level multitenancy model allows application instances to be shared by multiple tenants. These instances can be configured to meet the tenants' requirements.[5,6] While the application instances are shared, the tenant data must be separated and must be accessible only by the tenant that owns it[7] (see Figure 2). In addition to the separation of tenant data, a multitenant environment should ensure that computing resources are equally distributed between the tenants. Performance separation ensures that increased demand for resources by one tenant does not negatively affect other tenants.[8,9]

0740-7459/18/$33.00 © 2018 IEEE

CHALLENGES OF MICROSERVICE MIGRATION

Multitenancy is a system's ability to fulfil the requirements of multiple groups of service consumers, organizations, and even competitors in an industry. It enables organizations that demand complete autonomy in the administration of their users and associated data to share access to the same physical instances of the system and to the application instances, while keeping their data strictly separated and ruling out any superuser having access to and control over it. Multitenant applications must be highly configurable with tenant-specific settings. Legacy enterprise applications have often been designed for single organizations and operate in single-tenant mode.

Statefulness in the context of microservices is the ability to retain state information that was generated previously. The response of a stateful microservice may depend not only on the most recent service request but also on the retained state from previous interactions. Ideal microservices are stateless, but monolithic legacy code is often stateful.

Data consistency is enforced by a system's ability to affect data stored in a shared repository only in allowed ways. Data consistency defects can be introduced by mistake when migrating sequential legacy code that accesses a centralized data repository to microservices. Microservices are designed to be scaled out, allowing multiple microservice instances to operate in parallel. To improve performance and scalability, microservices rely on decentralized data repositories, which leads to data consistency challenges when synchronizing the data. Concurrent database access is rarely supported in sequential legacy code.

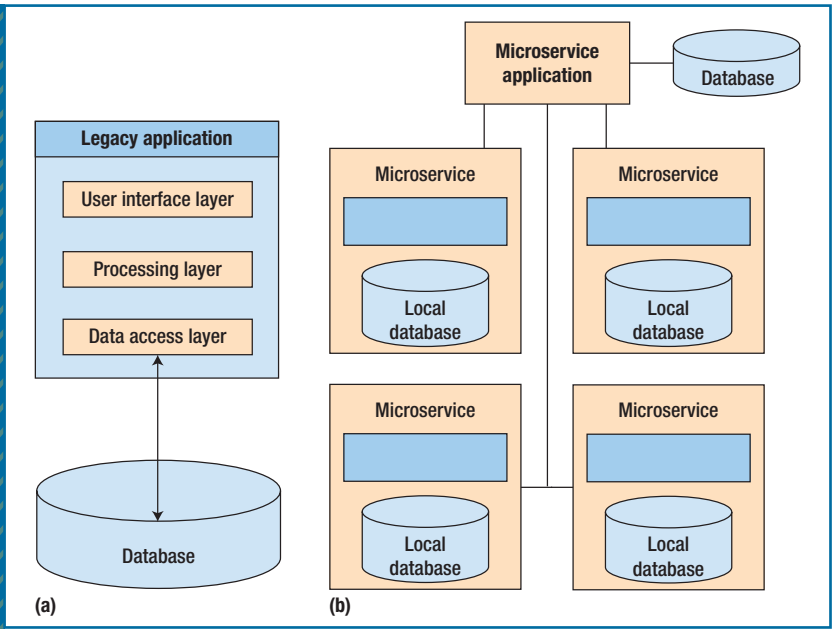

FIGURE 1. Comparing (a) a legacy (e.g, monolithic) architecture and (b) a microservice architecture.

Single-Tenancy and Multitenancy Challenges in Legacy Source Code

Legacy enterprise applications often operate in single-tenant mode. They were not developed for modern multitenant cloud environments, and they usually do not support multitenancy. Single-tenant legacy applications access the data repository of a single organization.

It is essential that the migration of single-tenant legacy code to microservices also considers multitenancy. Multitenant microservice instances can be shared by multiple tenants, configured to meet individual requirements, while strictly separating the tenants' data.

To ensure tenant data isolation, the source code needs to ensure that the data associated with tenant-aware input channels is tagged with the correct tenant ID, and that this tenant tag is maintained unmodified until the data reaches a tenant-aware output channel. At the source code level, tenant-aware business objects need to be tagged with the correct tenant context as follows (see Figure 3).

Procedures that process tenant-aware input data (e.g., service requests, user interface inputs, and database read operations) need to retrieve the tenant ID from the runtime environment (`env`) and assign it

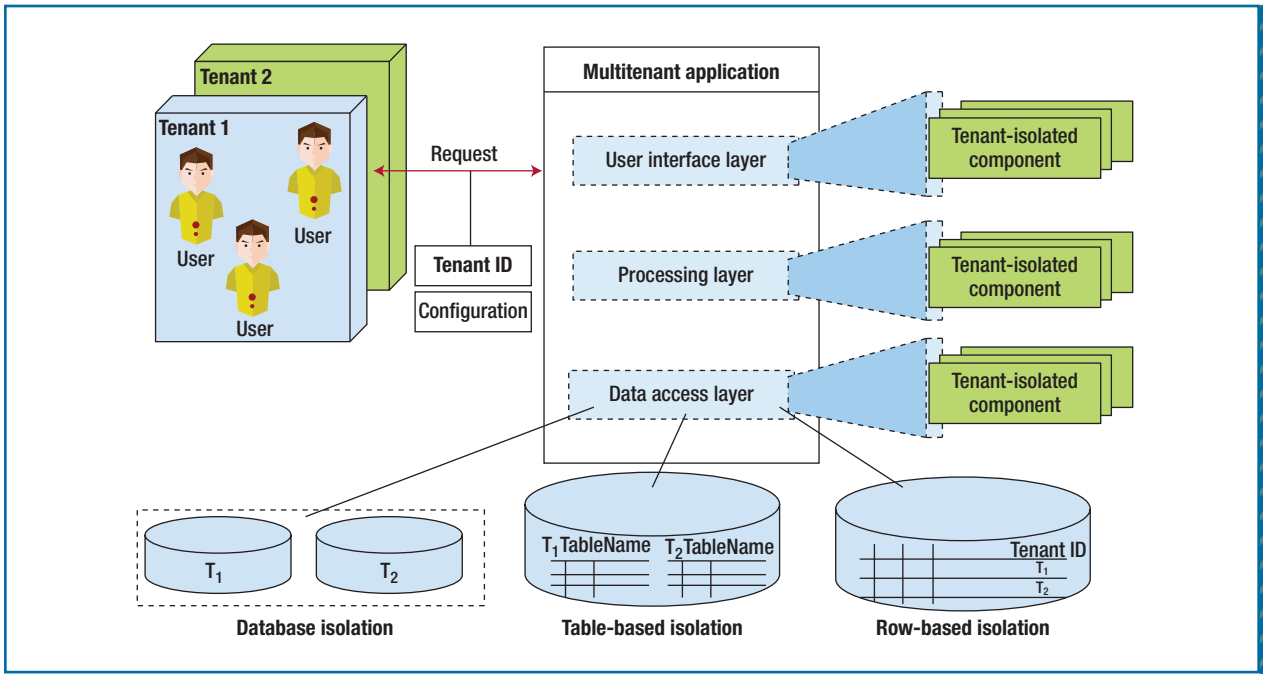

FIGURE 2. A multitenant enterprise software-as-a-service architecture.[6,7]

to the input channel. For example, if the multitenant microservice is hosted by the Google App Engine, the tenant ID is retrieved from the Google Apps domain.[10]

Similarly, procedures that process tenant-aware data outputs (e.g., service responses, user interface outputs, and database write operations) retrieve the tenant ID and assign it to the output channel. For example, Windows Azure requires setting the tenant context before calling the storage API.[10]

In summary, the multitenant source code first retrieves the tenant ID (tenant context) before receiving data from input channels, sets the tenant context for all required tenant-aware business objects, and sets the tenant context for all data that is sent to output channels.

Pattern-Based Microservice Migration of Single-Tenant Legacy Source Code

Multitenancy challenges can be decomposed into subproblems that can be solved using a combination of existing architectural patterns. In previous work, we have shown how to enable multitenancy by applying architectural patterns for enterprise applications and architectural refactoring.[6] We focused on enabling multitenancy in components; however, the techniques and architectural patterns can also be applied to multitenant microservices.

The microservice's data access component for a multitenant database can be implemented using the Two-Level Data Mapping Gateway pattern,[6] a combination of the Data Mapper pattern and the Table Data Gateway pattern[11] (see Figure 4). The first (Data Mapper) stage maps domain objects to the database gateway and implements basic CRUD (create, read, update, delete) operations, while the second (Gateway) stage loosely couples the access to a multitenant data repository and ensures the separation of tenant data.

```
processInputChannel ( TenantAwareBO input ) {
  //1. retrieve the tenant context
  tenantID = env.getTenantID();
  //2. set the tenant context
  input.setTenantID(tenantID);
  ...
}

processOutputChannel ( TenantAwareBO output ) {
  ...
  //1. retrieve the tenant context
  tenantID = env.getTenantID();
  //2. set the tenant context
  output.setTenantID(tenantID);
  sendResponse(output);
}
```

FIGURE 3. Multitenancy through tenant-aware I/O channels.

The operational-logic component of a multitenant microservice can be implemented using the Strategy pattern. This pattern lets you modify or extend tenant-specific logic implementations without affecting other parts of the implementation[6] (see Figure 5).

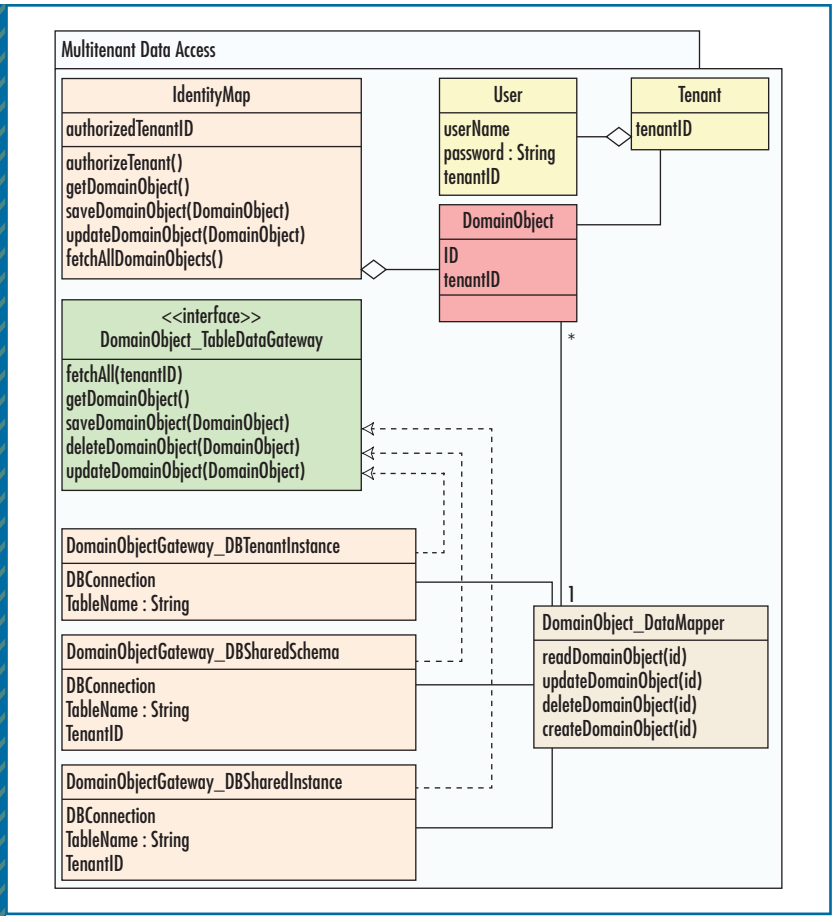

FIGURE 4. A multitenant MVC (model–view–controller) model.

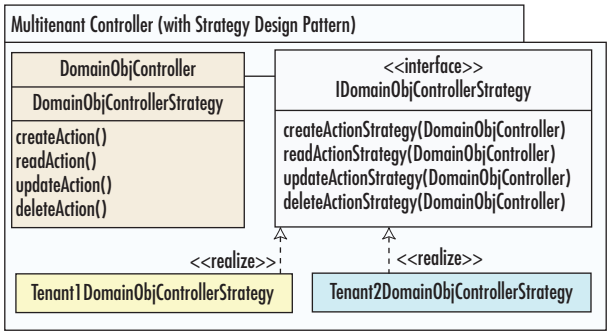

FIGURE 5. A multitenant MVC controller.

The user interface component of a multitenant microservice can be implemented using the Two-Step-View pattern.[6] This pattern facilitates the decoupling of user interface data from the layout. This allows the flexible implementation of tenant-specific user interface layouts using the same user interface data (see Figure 6).

Statefulness

A stateful system produces outputs that depend on the state generated in previous interactions and conversations. For example, an e-commerce application is stateful if it remembers previous shopping activities and past visits to the online shop.

Why Stateless Microservices Are Ideal

In many cases, stateless microservices are ideal because they exploit the benefits of cloud computing, such as on-demand elasticity, load balancing, high availability, and high reliability through redundancy.[12] Stateful microservices, on the other hand, require a more complex logic for managing the state, are less scalable, and do not facilitate high availability and high reliability.[13] For example, state information has to be made persistent and synchronized in hot standby configurations and recreated in failover scenarios.

Even without failure, statefulness leads to session affinity, which can decrease throughput and increase latency due to the need to wait for specific stateful instances. On the other hand, stateless microservices can decrease the average response time by distributing the load among the available microservices, without the need of complex state management and state synchronization techniques.

In a typical microservice architecture (see Figure 7), service consumer requests are directed to a load balancer that routes them to available microservice instances. System availability is defined as

$$Availability = \frac{MTBF}{MTBF + MTTR},$$

where MTBF is the mean time between failures and MTTR is the mean time to repair.[14] Assuming that all microservice instances have a similar MTBF, the system availability can be increased by reducing the MTTR. This can be achieved by allowing the load balancer to stop routing incoming consumer requests to failed microservice instances and to reroute them to operating instances instead. The load balancer is able to immediately route requests to already deployed redundant microservices only if these are stateless.

Stateless microservices also achieve a higher system reliability through redundancy. The reliability of a system is defined as the ratio between the number of successful responses to the total number of requests:[14]

$$Reliability = \frac{Successful\ responses}{Total\ requests} * 100\%.$$

If the services are stateless, the load balancer can increase the probability of receiving a successful response from one of the available microservices by sending the same request to multiple available microservices, without the need to synchronize or replicate state in a session database.

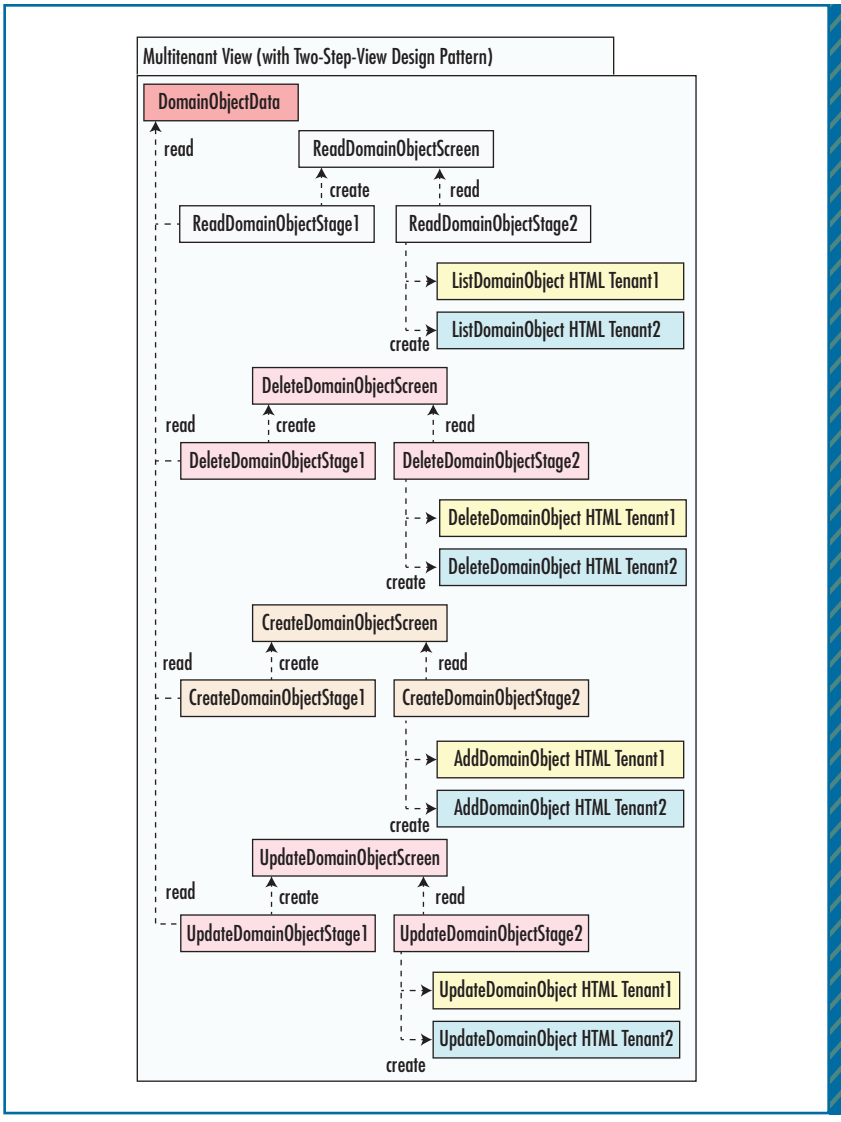

FIGURE 6. A multitenant MVC view.

Statefulness in Legacy Code

In the context of extracting microservices, legacy source code is stateful if it is capable of retaining values (i.e., its state) between invocations and is able to generate outputs that depend not only on the current input parameters but also on the previously retained state (see Figure 8).

Statefulness in the context of microservices. When analyzing the statefulness of legacy code for the purpose of microservice extraction, the inclusion or exclusion of state variable definitions (i.e., in-memory session state) determines whether or not the analyzed code is stateful with respect to a specific state variable.

Figure 8 depicts this in an example. State variable s_1 is defined within the analyzed code, a second state variable s_2 is defined outside the analyzed code (this could be either a session or domain state), and a third domain state value s_3 is stored in a database. Microservice option A includes only state variable s_1; therefore, microservice A is stateful only with respect to s_1 (writing to s_2 or s_3 can be considered an output). Microservice option B is stateful with respect to s_1 and s_2, and microservice option C (which includes the database) is stateful with respect to s_1, s_2, and s_3.

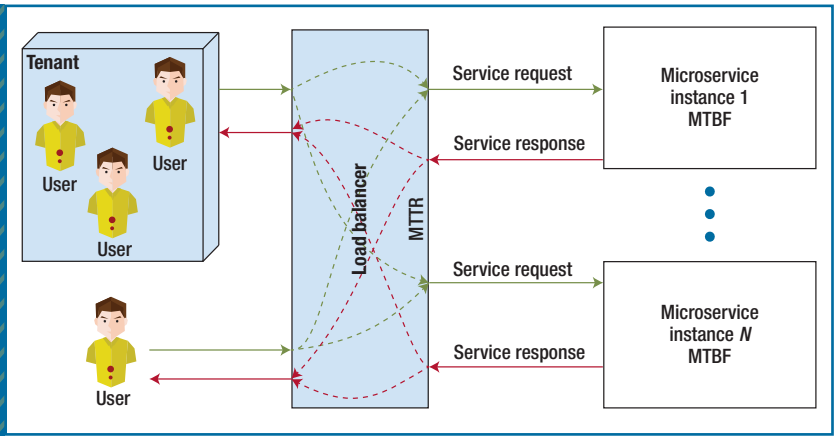

FIGURE 7. Availability and reliability through redundant (stateless) microservices.

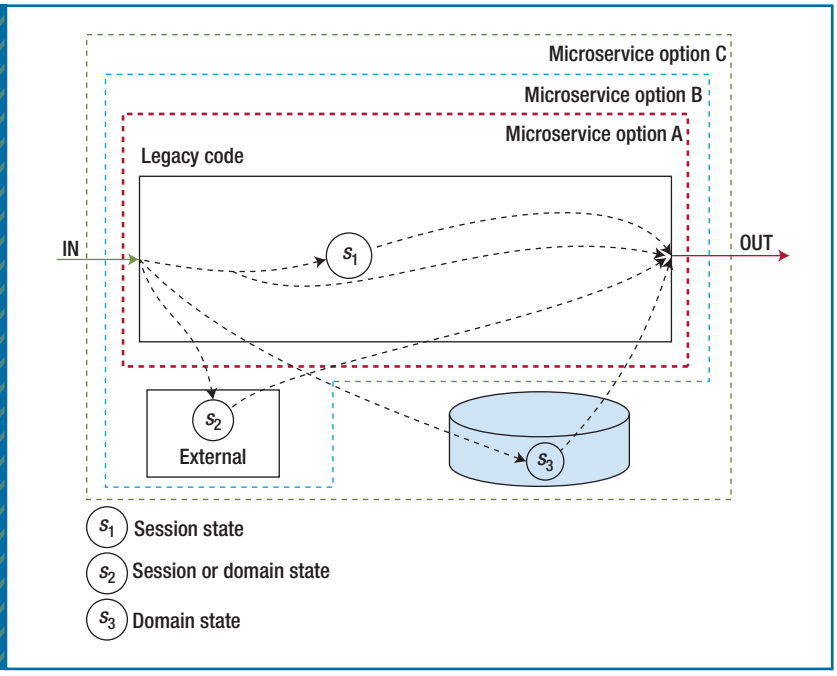

FIGURE 8. Statefulness in the context of microservice extraction.

Stateful procedures and objects. Legacy source code can be stateful at the procedure level, object level, or component level. A stateful procedure returns values that depend on variables whose lifetime exceeds the procedure execution; i.e., variables that are not reset between the invocations of the procedure. In PHP, for instance, a stateful procedure can be implemented by declaring a static variable in a procedure (see Figure 9, `function p1`) or, as in most languages, by declaring a global variable outside the procedure (see Figure 9, `function p2`). A procedure can itself become stateful by invoking and returning values of other stateful procedures (see Figure 9, `function p3`).

The most common form of legacy code statefulness is found in objects (class instances). An object can be stateful by instantiating a class that implements stateful procedures or one with (state) properties (see Figure 10).

State properties can be static or nonstatic. Static properties lead to class-level statefulness that affects all instances of a class when modified (see Figure 10, `class StatefulStatic`). In this case, the state can be changed only for all class instances at once, not for individual ones. Nonstatic properties, on the other hand, affect individual class instances (see Figure 10, `class StatefulNonStatic`).

Stateful components. Stateful legacy components may include stateful procedures or stateful objects. For migrating such stateful components to microservices,[12] it is important to distinguish between the microservice's public API and its configuration API. The public API is accessible by service consumers, and therefore the ideal microservice is stateless with respect to the public API.

The microservice configuration API, on the other hand, allows the automatic deployment and automatic reconfiguration of microservice instances. For example, the configuration API may expose procedures for setting specific multitenancy-related properties, such as a database connection setting. Therefore, the configuration API does not necessarily need to be stateless.

Microservice Migration of Stateful Legacy Source Code

We observe that migrating legacy code to microservices is either a top-down,

bottom-up, or meet-in-the-middle approach involving code-level refactoring decisions[15] of the legacy code and architectural decisions[16,17] of the desired microservice solution. The following service-oriented architecture (SOA) patterns address statefulness and are applicable in microservice architectures.[18]

The Stateful Messaging pattern delegates internal state data to microservice messages—i.e., the service request and service response. Stateful legacy procedures can be made stateless by replacing the local state variables with additional parameters and returning values that allow setting and retrieving the state. These additional state parameters and return values are then linked to the microservice request and response, respectively. Stateful objects and components can also be refactored in the same way.

The Partial State Deferral pattern is an option if the microservice can remain partially stateful, and the goal is to reduce its memory consumption.

The State Repository pattern defers the state of a microservice to a dedicated state repository. By sharing the state repository among the microservice instances, these can be refactored to be entirely stateless for the purpose of increased availability and reliability. This pattern corresponds to option B in Figure 8.

The Stateful Service pattern defers the state of a microservice to a set of stateful utility services whose only purpose is to manage state information. This pattern isolates the problem of statefulness and defers it to other stateful services that have the same negative impact on scalability, availability, and reliability.

Data Consistency

Data consistency is a property of a distributed shared data storage system. It specifies the allowed behavior with respect to data access operations. Sequential (single-threaded) legacy source code does not encounter data consistency issues when accessing data in a repository, database, or shared memory. However, when migrating such sequential code to microservices, multiple instances of a microservice interact in parallel with a data repository, creating data consistency challenges.

Identifying Data Consistency Issues in Legacy Code

Legacy source code often contains a mix of read/write operations from or to a data repository such as a database, file system, or other persistent storage. We observe that data consistency issues occur, however, when multiple instances of the same code (i.e., microservices) interact with a shared data repository. Therefore, before migrating such code to microservices, data access operations should be grouped into read-only or read/write operations.

Read-only operations only retrieve data from the data repository; they do not update, delete, or create new data. Read-only data is, for example, a multitenancy configuration setting that is read at start-up, and the local copy is never updated during the runtime of the microservice. Read/write operations update, delete, or create new data in a repository.

Restructuring Legacy Code into Data-Consistent Microservices

Microservices include their own data repository, such as a dedicated database instance, dedicated database schema in a shared database, or dedicated database tables. This leads to consistency challenges when the microservice data is synchronized with a centralized database.

The SOA pattern Service Data Replication replicates data in a service database.[18] This pattern can be safely applied for read-only operations. However, read/write operations need additional data consistency checks.

```
function p1() {
  static $static state = 0;
  $static_state++;
  return $static_state;
}

$glbl_state ;
function p2() {
  global $glbl_state ;
  $glbl state++;
  return $glbl_state ;
}

function p3() {
  return p1();
}
```

FIGURE 9. Examples of stateful procedures.

```
class StatefulStatic {
  private static $state;
  public static function setState($s) {
    self::$state = $s;
  }
  public function getState() {
    return self::$state;
  }
}

class StatefulNonStatic {
  private $state;
  public function setState($s) {
    $this->state=$s;
  }

  public function getState() {
    return $this->state;
  }
}
```

FIGURE 10. Stateful classes using a static state variable.

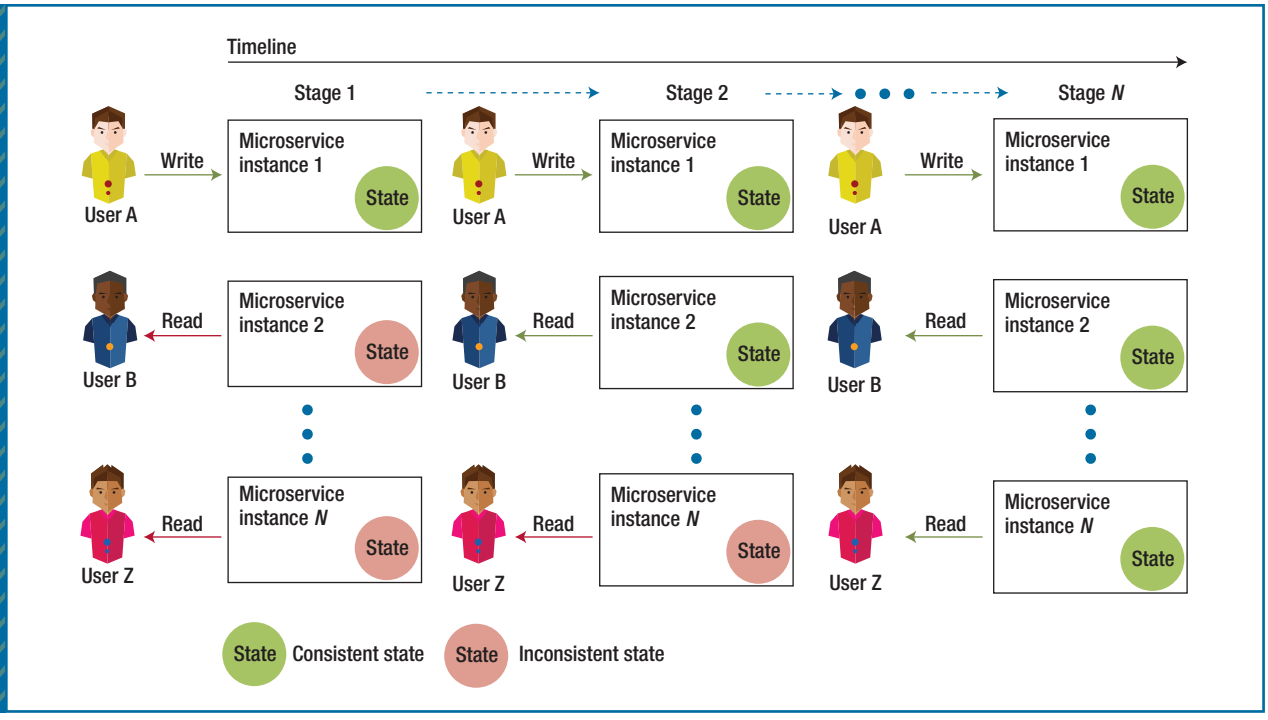

FIGURE 11. The Eventual Consistency pattern. Inconsistencies are corrected through multistage synchronization of the replicas.

The cloud-computing patterns Strict Consistency and Eventual Consistency address the consistency problem in cloud storage solutions.[12] The Strict Consistency pattern allows a variable number of data replicas to be read from and written to. For example, in a system consisting of n data replicas, a write operation might access w replicas, while a read operation accesses r replicas. For each operation it is ensured that $n < w + r$, and strict consistency is guaranteed through the number of read or written replicas by making sure that each operation accesses at least one most current data version.[12]

The Eventual Consistency pattern is applied for unreliable or limited-bandwidth networks, or high data volume, where simultaneously accessing multiple replicas is not feasible[12,19,20] (see Figure 11). Instead, only one replica is accessed for read/write operations, resulting in temporary data inconsistencies for the benefit of increased availability and performance. The inconsistencies are eventually corrected through synchronization of the data replicas. Data inconsistencies can persist if data entries in different replicas have been modified at the same time. In such a case, rules specify how such issues are resolved—for example, by simply dropping one of the modified versions.[12] We are of the view that this rule works only in a non-data-critical system, while a critical enterprise system would require a rollback or compensation activities.

We have described three basic challenges for migrating legacy code to microservices. A best-practice solution is to develop a microservice iteratively, focusing on

- eliminating statefulness from the extracted legacy code,
- implementing multitenancy functionalities, and
- solving potential new introduced data consistency challenges.

It is important to note that while these three challenges are interrelated, they are created by different

ABOUT THE AUTHORS

ANDREI FURDA is a lecturer in the Queensland University of Technology's Science and Engineering Faculty, where he teaches enterprise systems and mobile-app development. His research interests include the modernization of legacy software systems for cloud computing. Furda received a PhD from Griffith University's School of Engineering. Contact him at andreifurda@gmail.com.

COLIN FIDGE is a full professor in the Queensland University of Technology's Science and Engineering Faculty, where he teaches computing fundamentals. His research interests include safety-critical, security-critical, and mission-critical software engineering. Fidge received a PhD in computer science from the Australian National University. Contact him at c.fidge@qut.edu.au.

OLAF ZIMMERMANN is a professor and institute partner at the University of Applied Sciences of Eastern Switzerland, Rapperswil. His areas of interest include web-based application and integration architectures, service-oriented architecture and cloud design, and architectural knowledge management. Zimmermann received a doctorate in computer science from the University of Stuttgart. He's on the *IEEE Software* editorial board.

WAYNE KELLY is a senior lecturer at the Queensland University of Technology's School of Electrical Engineering and Computer Science, where he teaches enterprise software architecture, software engineering, and compilers. His research interests include static analysis of program code, particularly for parallelization and detecting dataflow security violations. Kelly received a PhD in computer science from the University of Maryland, College Park. Contact him at w.kelly@qut.edu.au.

ALISTAIR BARROS is a professor and the Head of Services Science Discipline at the Queensland University of Technology's School of Information Systems. His research interests are software and enterprise architecture, business process management, technical software, and service analysis methods. Barros received a PhD in computer science from the University of Queensland. Contact him at alistair.barros@qut.edu.au.

types of requirements. Multitenancy is driven by the economic advantages resulting from sharing resources and infrastructure, while taking into consideration the tenants' data privacy needs. Statefulness is a characteristic that influences nonfunctional requirements with respect to scalability, reliability, and availability of the modernized system. Finally, data consistency is a functional requirement that defines the

correct operation of the system and ranges from relaxed eventually consistent systems to strictly consistent ones.

Acknowledgments

This research was supported in part by ARC-DP grant DP140103788.

References

1. O. Zimmermann, "Microservices Tenets," *Computer Science—Research and Development*, vol. 32, nos. 3–4, 2017, pp. 301–310.
2. C. Pautasso et al., "Microservices in Practice, Part 1: Reality Check and Service Design," *IEEE Software*, vol. 34, no. 1, 2017, pp. 91–98.
3. I. Gorton, "Hyper Scalability—the Changing Face of Software Architecture," *Software Architecture for Big Data and the Cloud*, Elsevier, 2017, pp. 13–31.
4. S. Johann, "Dave Thomas on Innovating Legacy Systems," *IEEE Software*, vol. 33, no. 2, 2016, pp. 105–108.
5. J. Kabbedijk et al., "Defining Multi-tenancy: A Systematic Mapping Study on the Academic and the Industrial Perspective," *J. Systems and Software*, Feb. 2015, pp. 139–148.
6. A. Furda et al., "Reengineering Data-Centric Information Systems for the Cloud—a Method and Architectural Patterns Promoting Multitenancy," *Software Architecture for Big Data and the Cloud*, Elsevier, 2017, pp. 227–251.
7. F. Chong, G. Carraro, and R. Wolter, "Multi-tenant Data Architecture," Microsoft, 2006; ramblingsofraju.com/wp-content/uploads/2016/08/Multi-Tenant-Data-Architecture.pdf.
8. V. Narasayya et al., "SQLVM: Performance Isolation in Multi-tenant Relational Database-as-a-Service," *Proc. 6th Biennial Conf. Innovative Data Systems Research* (CIDR 13), 2013.
9. R. Taft et al., "STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments," *Proc. 7th ACM Symp. Cloud Computing*, 2016, pp. 388–400.
10. S. Walraven, E. Truyen, and W. Joosen, "Comparing PaaS Offerings in Light of SaaS Development," *Computing*, vol. 96, no. 8, 2014, pp. 669–724.
11. M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
12. C. Fehling et al., *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*, Springer, 2014.
13. M.F. Gholami et al., "Cloud Migration Process—a Survey, Evaluation Framework, and Open Challenges," *J. Systems and Software*, Oct. 2016, pp. 31–69.
14. E. Bauer and R. Adams, *Reliability and Availability of Cloud Computing*, Wiley–IEEE Press, 2012.
15. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
16. O. Zimmermann, "Architectural Refactoring: A Task-Centric View on Software Evolution," *IEEE Software*, vol. 32, no. 2, 2015, pp. 26–29.
17. O. Zimmermann et al., "Architectural Decision Guidance across Projects—Problem Space Modeling, Decision Backlog Management and Cloud Computing Knowledge," *Proc. 12th Working IEEE/IFIP Conf. Software Architecture* (WICSA 15), 2015.
18. T. Erl, *SOA Design Patterns*, Prentice Hall, 2009.
19. M. Fowler, "Eventual Consistency," *Microservice Trade-Offs*, 2015; martinfowler.com/articles/microservice-trade-offs.html#consistency.
20. W. Vogels, "Eventually Consistent," *Comm. ACM*, vol. 52, no. 1, 2009, pp. 40–44.