



Migrating a University Registration Platform to a Microservice Architecture: A Review of Published Lessons and Migration Patterns

Manon Iyer^{1*}, Hala Haddad², Bongani Ngo³

¹Department of Computer Science, Tamiraparani Graduate Institute of Computer Science, Slovakia

²Department of Computer Science, Han River Research Institute of Computer Science, Peru

³Department of Computer Science, Tagus University Centre for Computer Science, Philippines

Article Info

Volume 1, Issue 1, 2026

[doi: 10.62715/ijcser.2026.v1.i1.pp13-18](https://doi.org/10.62715/ijcser.2026.v1.i1.pp13-18)

Received: 08 February 2026

Accepted: 29 March 2026

Published: 27 May 2026

Abstract

University registration platforms are old, monolithic and subject to a load pattern that few other systems share: almost idle for most of the term and overwhelmed in the hours when enrolment opens. Microservices are frequently proposed as the remedy. This narrative review examines what the published literature on migrating monoliths to microservices has to say to an institution considering that step. Accounts devoted to university systems are few, so the review draws on the general body of migration research, including practitioner surveys, experience reports from other sectors, grey literature reviews and the standard books on the architecture, and reads it against the particular demands of course registration. It covers the reasons organisations give for migrating and how well they apply, the decomposition of the registration domain into services, incremental migration by the strangler fig approach, the handling of data and consistency where a limited number of seats must never be oversold, the operational capabilities that have to be in place first, and legal duties concerning student records. The lessons that recur across sources are collected in a table. The review concludes that the literature supports a cautious, incremental migration for those institutions with a clear scaling or delivery problem, and that evidence from higher education itself remains thin.

Keywords: *Microservices, Legacy system migration, Student information systems, Course registration, Strangler fig pattern, Software architecture*

© 2026 Manon Iyer et al. This is an open access article under the CC BY license (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

1. Introduction

A registration platform lets students search the course catalogue, checks that they may take what they choose, and records their enrolment against a limited number of places. In many universities it is a module of a student information system that also holds admissions, grades, fees and timetables, built around one relational database and extended over many years by local customisation. The architecture is the classic monolith: one deployable unit, one schema, one release cycle.

* Corresponding author: Manon Iyer, Department of Computer Science, Tamiraparani Graduate Institute of Computer Science, Slovakia. E-mail: manon.iyer@tamiraparani.edu

© 2026 Manon Iyer et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Microservices divide such a system into small services that are deployed independently, own their data and communicate over the network (Lewis and Fowler, 2014; Newman, 2015). The style grew out of service-oriented architecture and the operating practices of large web companies (Dragoni et al., 2017), and it promises exactly what registration seems to need: the ability to scale the part under pressure without scaling everything, and to change one function without redeploying the rest.

Whether the promise holds for a university is less obvious. The sector has small development teams, fixed academic calendars and strict duties towards student data. This review asks what the published literature on monolith-to-microservice migration teaches, and how far those lessons carry over to a registration platform.

2. Scope and Approach

The article is a narrative review. It considers peer-reviewed studies of microservice migration, notably interview and survey research with practitioners, published experience reports, a systematic review of grey literature, and the books that practitioners themselves cite. It does not report a migration carried out by the author and describes no particular institution. Published accounts that deal specifically with university registration are scarce, and that scarcity is itself a finding. The general literature is therefore read with the registration domain in mind, and the reasoning that connects the two is made explicit so that readers can judge it.

3. What Makes Registration Distinctive

Three properties of the domain shape any architectural decision. The first is the load profile. Demand is concentrated in short windows at the start of each enrolment period, when a large part of the student body arrives at the same minute and competes for the same popular sections. The second is a hard invariant: a section has a capacity, and the number of students enrolled must not exceed it. Prerequisites, timetable clashes, credit limits and holds on a student's account are further rules that must be evaluated at the moment of enrolment, and they draw on data owned by other parts of the system.

The third property is regulatory. Student records are protected by law, in the United States by the Family Educational Rights and Privacy Act (FERPA, 1974) and in the European Union by the General Data Protection Regulation (European Union, 2016). An architecture that copies student data into many service databases multiplies the places where access must be controlled, audited and, where the law requires, erased or corrected.

4. Reasons for Migrating and How Well They Apply

Practitioners interviewed by Taibi et al. (2017) named maintainability and scalability as their main reasons for moving to microservices, and admitted that some migrations were undertaken because others were doing the same. The industrial survey of Di Francesco et al. (2018) found a similar mix of technical and organisational motives. The grey literature review by Soldani et al. (2018) sets the reported gains, among them independent deployment and fault isolation, against pains that include harder testing, more demanding monitoring and greater consumption of infrastructure.

For registration, the scalability argument is real but narrow. The load peak falls on a few operations, catalogue search and seat allocation, and those could in principle be scaled by other means. The maintainability argument depends on team structure. Conway (1968) observed that a system's structure mirrors the communication structure of the organisation that builds it, and microservices pay off when several teams need to release independently. A university with one small team gains little from independent deployment and inherits all of the operational cost. Fowler (2015) advises starting from a well-structured monolith for this reason.

5. Decomposing the Registration Domain

The step that practitioners report as hardest is deciding where the service boundaries lie (Di Francesco et al., 2018). The principle is older than the style. Parnas (1972) argued that modules should hide design

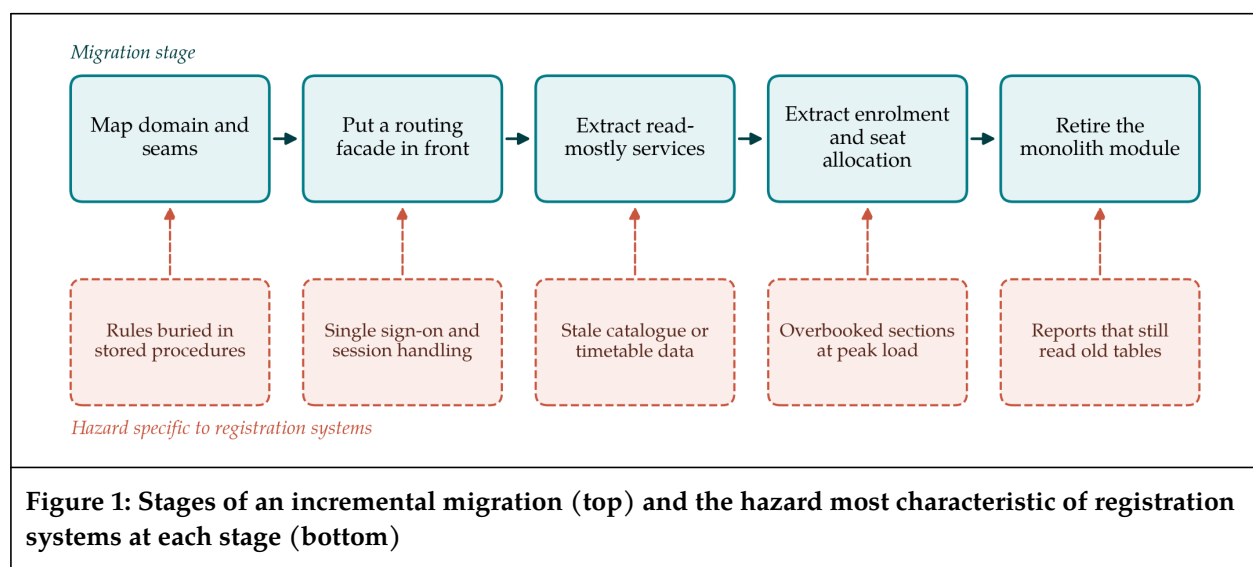
decisions likely to change, and domain-driven design supplies the working tool: the bounded context, a part of the domain within which a model and its vocabulary are consistent (Evans, 2003). In registration, the catalogue, the timetable, student eligibility, enrolment and waitlists, and fees are natural candidates. The word "course" means different things in each, which is the usual sign of a context boundary.

Automated support exists. Mazlami et al. (2017) derive candidate services by clustering classes according to how they change together in version history, and the classification by Fritzsche et al. (2019) groups refactoring approaches by the inputs they need, from static code analysis to workload data. Such techniques assume a codebase with meaningful structure and history. Much of the logic of an older student system sits in database procedures and vendor customisations, where these tools reach poorly, so domain analysis with registrar staff is likely to do more of the work.

6. Migrating Incrementally

The literature is close to unanimous that migration should be incremental. The strangler fig approach places a routing layer in front of the old system and moves functions behind it one at a time until nothing is left to route to the monolith (Fowler, 2004). Newman (2019) catalogues the supporting patterns, among them branch by abstraction, parallel running of old and new implementations, and change data capture to keep two stores in step. The experience report of Balalaie et al. (2016) describes a migration organised as a sequence of such steps and stresses that the delivery pipeline was built before services were split out. Knoche and Hasselbring (2018) report a modernisation that first defined service facades over the legacy system, moved clients onto them, and only then replaced the internals.

The academic calendar gives this advice particular force. Enrolment periods are fixed and public, so a failed cut-over cannot be postponed. Each increment has to be released and proven in a quiet part of the term, and the order of extraction should follow risk. Figure 1 sets out a plausible sequence, pairing each stage with the hazard that the domain makes most likely. Read-mostly functions such as catalogue search come early because an error there is visible and recoverable. Seat allocation comes late, once the team has operated simpler services through at least one peak.



7. Data Ownership and Consistency

Splitting the database is the part of migration that reports describe as most painful (Taibi et al., 2017). The pattern literature recommends a database per service, with other services obtaining data through interfaces or published events, not shared tables (Richardson, 2018). Once data is divided, an operation that used to be one transaction spans several services. The usual answer is the saga, a sequence of local transactions each paired with a compensating action that undoes it, an idea introduced for long-lived database transactions by Garcia-Molina and Salem (1987).

Enrolment tests this answer severely. Under a network partition a distributed system must give up either consistency or availability (Brewer, 2012), and for seat counts consistency cannot be given up. The practical consequence drawn in the design literature is to keep the invariant inside one service: the component that owns a section's capacity should also own the enrolments counted against it, so that the check and the update remain a single local transaction (Kleppmann, 2017). Eligibility rules that depend on other contexts can be evaluated against replicated data, provided the staleness is bounded and understood. Prerequisites change slowly. A financial hold placed a minute ago may not.

8. Operational Prerequisites

A microservice system has many more moving parts than the monolith it replaces. Automated build, test and deployment are therefore preconditions, not refinements (Humble and Farley, 2010). So are centralised logging, tracing across service calls and monitoring. Calls that were in-process become remote and can fail or stall, and Nygard (2018) describes the timeouts, circuit breakers and bulkheads that stop one slow dependency from exhausting its callers. Fan-out makes matters worse at the peak: Dean and Barroso (2013) showed that when a request depends on many components, the slow tail of each one comes to dominate overall response time. An enrolment request that consults five services is exposed to five tails at precisely the moment each is busiest.

Testing changes character as well. A monolith can be exercised end to end in one process, whereas a set of services needs contract tests at each interface and rehearsals of the whole system under load. For registration the rehearsal that matters is the opening of enrolment, and it can be approximated by replaying recorded request patterns from an earlier term against a staging environment before each release. Running the old and new implementations side by side on the same requests and comparing their answers, the parallel run described by Newman (2019), is well suited to eligibility checks, where a silent difference in a rule would otherwise surface only as a complaint from a student.

Table 1 gathers the lessons that recur across the sources reviewed and states what each implies for a registration platform.

Table 1: Recurrent lessons in the microservice migration literature and their implication for a registration platform		
Lesson	Sources	Implication for registration
Migrate in small steps behind a facade	Fowler, 2004; Newman, 2019	Release each step in a quiet part of the term
Build the delivery pipeline first	Balalaie et al., 2016	Automate deployment and rollback before the first extraction
Boundaries follow the domain	Evans, 2003	Model catalogue, eligibility, enrolment and fees as separate contexts
Splitting data is the hardest part	Taibi et al., 2017	Keep section capacity and enrolments in one service
Expect remote calls to fail	Nygard, 2018	Set timeouts and degrade search before enrolment
Architecture mirrors organisation	Conway, 1968	A single small team may be better served by a modular monolith
Operational cost rises	Soldani et al., 2018	Budget for monitoring, tracing and on-call cover at peak

9. Gaps and Future Work

The clearest gap is evidence from the sector. The studies cited here draw largely on commercial software companies and enterprise information systems. Peer-reviewed accounts of registration or student-record

migrations, with enough detail on load, team size and outcome to support comparison, would allow the reasoning of this review to be tested. A second gap concerns alternatives. The literature compares microservices with an unreformed monolith far more often than with a modular monolith whose peak-load path has been tuned or scaled separately, which may be the realistic competitor for a small institution.

Third, most universities run vendor systems. Migration then means surrounding a product with services through its interfaces, not decomposing source code, and that situation is little studied. Finally, the effect of data distribution on compliance with records legislation is asserted more often than it is examined, and would repay joint work by software engineers and data protection specialists.

10. Conclusion

The migration literature does not say that a university should move its registration platform to microservices. It says what such a move costs and how to reduce the risk: migrate behind a facade in small steps, establish automated delivery and observability first, draw boundaries from the domain, and keep the seat-allocation invariant within one service. It also says that the benefits depend on having several teams and a real scaling problem. Institutions that meet those conditions will find well-tried patterns to follow. Those that do not may be better served by restructuring the monolith they have, and the sector as a whole would benefit from publishing what it learns either way.

Conflict of interest

To be declared by the author(s) before publication.

Funding

To be declared by the author(s) before publication.

References

- Balalaie, A., Heydarnoori, A. and Jamshidi, P. (2016). [Microservices architecture enables DevOps: Migration to a cloud-native architecture](#). *IEEE Software*, 33(3): 42-52.
<https://doi.org/10.1109/MS.2016.64>
- Brewer, E. (2012). [CAP twelve years later: How the "rules" have changed](#). *Computer*, 45(2): 23-29.
<https://doi.org/10.1109/MC.2012.37>
- Conway, M.E. (1968). [How do committees invent?](#) *Datamation*, 14(4): 28-31.
- Dean, J. and Barroso, L.A. (2013). [The tail at scale](#). *Communications of the ACM*, 56(2): 74-80.
<https://doi.org/10.1145/2408776.2408794>
- Di Francesco, P., Lago, P. and Malavolta, I. (2018). [Migrating towards microservice architectures: An industrial survey](#). *Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA)*, 29-38.
- Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2017). [Microservices: Yesterday, today, and tomorrow](#). In: Mazzara, M. and Meyer, B. (eds), *Present and Ulterior Software Engineering*. Springer, Cham, 195-216.
https://doi.org/10.1007/978-3-319-67425-4_12
- European Union (2016). [Regulation \(EU\) 2016/679 of the European Parliament and of the Council \(General Data Protection Regulation\)](#). Official Journal of the European Union, L 119: 1-88.
- Evans, E. (2003). [Domain-Driven Design: Tackling Complexity in the Heart of Software](#). Addison-Wesley, Boston.
- FERPA (1974). [Family Educational Rights and Privacy Act of 1974](#). 20 U.S.C. § 1232g; 34 CFR Part 99. United States.
- Fowler, M. (2004). [StranglerFigApplication](#). martinowler.com (online article).

- Fowler, M. (2015). [MonolithFirst](#). martinowler.com (online article).
- Fritzsche, J., Bogner, J., Zimmermann, A. and Wagner, S. (2019). [From monolith to microservices: A classification of refactoring approaches](#). In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment (DEVOPS 2018)*. Lecture Notes in Computer Science 11350. Springer, Cham, 128-141.
- Garcia-Molina, H. and Salem, K. (1987). [Sagas](#). *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 249-259. <https://doi.org/10.1145/38713.38742>
- Humble, J. and Farley, D. (2010). [Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation](#). Addison-Wesley, Upper Saddle River, NJ.
- Kleppmann, M. (2017). [Designing Data-Intensive Applications](#). O'Reilly Media, Sebastopol, CA.
- Knoche, H. and Hasselbring, W. (2018). [Using microservices for legacy software modernization](#). *IEEE Software*, 35(3): 44-49. <https://doi.org/10.1109/MS.2018.2141035>
- Lewis, J. and Fowler, M. (2014). [Microservices: A definition of this new architectural term](#). martinowler.com (online article).
- Mazlami, G., Cito, J. and Leitner, P. (2017). [Extraction of microservices from monolithic software architectures](#). *Proceedings of the 2017 IEEE International Conference on Web Services (ICWS)*, 524-531.
- Newman, S. (2015). [Building Microservices: Designing Fine-Grained Systems](#). O'Reilly Media, Sebastopol, CA.
- Newman, S. (2019). [Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith](#). O'Reilly Media, Sebastopol, CA.
- Nygard, M.T. (2018). [Release It! Design and Deploy Production-Ready Software, 2nd edition](#). Pragmatic Bookshelf, Raleigh, NC.
- Parnas, D.L. (1972). [On the criteria to be used in decomposing systems into modules](#). *Communications of the ACM*, 15(12): 1053-1058. <https://doi.org/10.1145/361598.361623>
- Richardson, C. (2018). [Microservices Patterns: With Examples in Java](#). Manning, Shelter Island, NY.
- Soldani, J., Tamburri, D.A. and Van Den Heuvel, W.-J. (2018). [The pains and gains of microservices: A systematic grey literature review](#). *Journal of Systems and Software*, 146: 215-232. <https://doi.org/10.1016/j.jss.2018.09.082>
- Taibi, D., Lenarduzzi, V. and Pahl, C. (2017). [Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation](#). *IEEE Cloud Computing*, 4(5): 22-32. <https://doi.org/10.1109/MCC.2017.4250931>