

BookWiseUD: Evaluating Dual-Service Architecture for Library Management

Wilder Steven Hernandez Manosalva
Dept. of Systems Engineering
Universidad Distrital Francisco José de Caldas
Bogotá, Colombia
Email: wshernandezm@udistrital.edu.co

Jhon Javier Castañeda Alvarado
Dept. of Systems Engineering
Universidad Distrital Francisco José de Caldas
Bogotá, Colombia
Email: jjcastaneda@udistrital.edu.co

Abstract—Academic libraries face operational inefficiencies from manual record-keeping systems that create administrative bottlenecks, prevent real-time book availability queries, and lack secure patron information management. This paper presents BookWiseUD, a web-based library management system implementing two-service architecture that separates authentication from domain operations to achieve security isolation, independent component scaling, and clear separation of concerns. The system demonstrates rigorous validation through 96% unit test coverage across 128 tests, 100% acceptance test pass rate, and performance testing showing 4.84ms average response time supporting 100 concurrent users, confirming that the architectural decision delivers security and scalability benefits while maintaining acceptable operational complexity for institutional contexts.

Index Terms—Library Management, Software Architecture, Microservices, Testing, DevOps, Security Isolation

I. INTRODUCTION

Library management represents a persistent challenge for educational institutions, particularly those with limited resources. Traditional library systems often rely on manual processes or monolithic digital solutions that fail to address modern requirements for security, scalability, and maintainability. Manual approaches involving paper records and spreadsheets lead to operational inefficiencies, data inconsistencies, and poor user experiences [?]. These systems typically lack real-time availability information, forcing patrons to visit physically to determine book status, while librarians struggle with error-prone loan tracking and reporting. The administrative overhead associated with manual processes diverts resources from value-added services like collection development and patron assistance.

Monolithic digital systems, while automating some processes, introduce their own limitations. These architectures typically bundle authentication, business logic, and data persistence into single applications, creating security vulnerabilities where credential breaches can expose operational data. Scaling challenges emerge as user bases grow, requiring vertical expansion rather than targeted component scaling. Maintenance complexity increases over time as codebases become entangled, making updates risky and time-consuming. Technology lock-in prevents adoption of specialized tools for specific functions, limiting optimization opportunities across different system aspects.

Recent architectural advancements in software engineering offer alternative approaches through microservices and service-oriented designs. These distributed architectures provide benefits in modularity, independent scaling, and technology flexibility [2]. Research demonstrates that properly implemented microservices can improve fault isolation, enable continuous deployment, and support polyglot persistence strategies. However, they introduce operational complexity in deployment coordination, inter-service communication, and distributed debugging. The overhead of managing multiple services, ensuring data consistency across boundaries, and monitoring distributed systems requires careful consideration against expected benefits.

Domain-Driven Design principles further inform architectural decisions by emphasizing bounded contexts and clear service boundaries based on business domains rather than technical considerations [?]. This approach encourages alignment between software structure and organizational structure, potentially reducing cognitive load for development teams. By identifying core domains, supporting subdomains, and generic subdomains, teams can make informed decisions about where to invest architectural effort and where to accept simpler solutions. The strategic design patterns of DDD, including context mapping and anti-corruption layers, provide tools for managing complexity in distributed systems.

The specific context of university libraries presents unique challenges that influence architectural decisions. Patron data sensitivity demands robust security measures, particularly around authentication and authorization, with compliance requirements for protecting personally identifiable information. Seasonal usage patterns, such as increased borrowing activity at semester beginnings, require scalable architectures that can handle load variations without over-provisioning during quieter periods. Limited IT resources common in academic settings necessitate maintainable systems with clear separation of concerns to enable focused expertise application. Integration requirements with existing institutional systems, such as student information systems or single sign-on providers, add further complexity to architectural decisions.

This paper addresses these challenges through BookWiseUD, a systematically engineered library management system developed for Universidad Distrital Francisco José de Caldas. Our work contributes to understanding when service sepa-

ration provides value versus adding unnecessary complexity in institutional contexts. We evaluate architectural trade-offs between monolithic and dual-service approaches through practical implementation and measurement. The system implements comprehensive testing strategies validating both functional and non-functional requirements, and incorporates DevOps practices including continuous integration and containerized deployment. The following sections detail our methodology, implementation decisions, validation approaches, and analysis of findings relevant to similar institutional projects.

II. METHODS AND MATERIALS

The development of BookWiseUD followed a structured engineering approach beginning with requirements analysis and architectural decision-making. We employed Scrum methodology across four one-week sprints to manage development complexity, with each sprint focusing on specific capability increments. Sprint planning sessions involved breaking down user stories into technical tasks, estimating effort, and identifying dependencies between authentication and domain components. Daily stand-ups maintained team alignment and surfaced integration challenges early, while sprint reviews validated functionality against requirements and retrospectives identified process improvements for subsequent iterations.

The fundamental architectural decision involved evaluating monolithic versus service-oriented approaches through systematic comparison of advantages, disadvantages, and alignment with institutional requirements. We conducted this evaluation through structured analysis sessions documenting decision rationale, alternatives considered, and trade-offs accepted. The evaluation framework considered technical factors including security, scalability, and maintainability alongside practical considerations including development timeline, team expertise, and operational overhead. This systematic approach ensured architectural decisions were made explicitly rather than emerging implicitly from implementation choices.

Our architectural analysis considered three primary options with varying complexity levels. The monolithic approach would consolidate authentication and domain logic in a single Spring Boot application with unified MySQL database storage. This option offered simplicity in deployment requiring only one application server, reduced operational complexity with unified monitoring, transactional consistency through database ACID properties, and straightforward debugging within single codebase. Development time was estimated at approximately four weeks. However, this approach presented security concerns where database breaches could expose both credentials and operational data simultaneously, scalability limitations requiring vertical scaling of entire application, and maintenance challenges where updates to authentication risked impacting domain operations.

The dual-service approach separated authentication into a dedicated Java Spring Boot service with MySQL database and domain operations into a Python FastAPI service with PostgreSQL database. This separation provided security isolation where credential breaches cannot expose operational

data, independent scalability allowing services to scale based on different load patterns, and clear separation of concerns simplifying focused maintenance. Development time was estimated at approximately eight weeks due to inter-service communication implementation. This approach introduced operational complexity requiring coordination of multiple services, data consistency challenges for information spanning service boundaries, network latency from inter-service communication, and distributed debugging requirements.

A full microservices architecture with granular service decomposition was considered but rejected due to excessive complexity for the project scale. This option would have created separate services for authentication, books, users, loans, and categories, providing maximum modularity and independent scaling. However, estimated development time exceeded twelve weeks, operational overhead would have been prohibitive for a two-person team, and the benefits did not justify costs given the project scope and institutional context. The rejection of this option reinforced the principle that architectural complexity should match problem complexity.

We selected the dual-service architecture based on several considerations that aligned with project objectives and constraints. Security requirements prioritized isolation of patron credentials from operational library data, reducing breach impact radius should either database be compromised. This consideration gained particular importance given increasing regulatory focus on data protection in educational contexts. Anticipated institutional growth patterns suggested value in independent scaling capabilities, particularly given different load characteristics between authentication bursts during session starts and steady domain operations during catalog browsing and loan processing.

Academic learning objectives included demonstrating distributed systems concepts and inter-service communication patterns, providing valuable experience with modern architectural approaches. Development feasibility assessments confirmed the approach could be implemented within the eight-week project timeline while maintaining production-quality engineering standards. Technology considerations favored using Spring Security for its mature authentication capabilities and FastAPI for its performance in data-intensive operations, with the dual-service architecture enabling this technology specialization without framework conflicts.

The implemented architecture consists of three main components communicating through REST APIs with JSON payloads. A React-based frontend provides responsive user interfaces for catalog browsing, loan management, and administrative functions across desktop and mobile devices. The frontend implements role-based rendering, showing different features to patrons versus librarians, and maintains authentication state using React Context to propagate user information across components. Client-side routing enables seamless navigation between application sections without full page reloads, enhancing user experience.

A Java Spring Boot authentication service handles credential validation, JWT token generation, and role management using

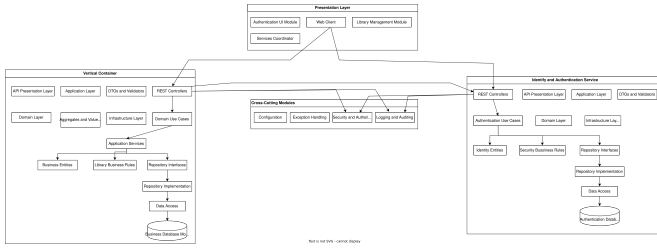


Fig. 1. System architecture showing component separation and communication patterns between frontend, authentication service, and domain service with separate database storage for security isolation.

MySQL for credential storage. The service implements Spring Security configurations for password encoding with BCrypt, role-based endpoint protection, and token validation logic. REST endpoints provide login functionality returning JWT tokens with configurable expiration, validation endpoints for inter-service authentication verification, and refresh mechanisms for extended sessions without re-authentication. Security measures include rate limiting on authentication attempts, CORS configuration for frontend access, and HTTPS enforcement in production deployments.

A Python FastAPI domain service manages all library operations including book catalog management, loan processing, and patron profiles using PostgreSQL for operational data storage. The service leverages FastAPI's asynchronous capabilities for efficient request handling and automatic OpenAPI documentation generation. Business logic enforces library rules including maximum loan durations, copy availability validation, and patron status checks. Services communicate through secure REST APIs with token-based authentication, implementing a security checkpoint pattern where each domain operation requires token validation through synchronous call to authentication service.

Database design followed separation of concerns principles with distinct schemas optimized for their specific purposes. The authentication database employs minimalist design storing only username, securely hashed passwords using BCrypt algorithm, role assignments, and creation timestamps. This design intentionally avoids foreign keys and complex joins to minimize attack surface while enabling efficient authentication lookups through indexed username queries. Regular security monitoring tracks access patterns for anomalous behavior detection.

The domain database implements normalized relational schema with entities for books, categories, patrons, and loans, enforcing business rules through constraints and optimizing query performance through strategic indexing. Foreign key constraints ensure referential integrity between related entities, while check constraints enforce domain rules such as positive available copies and valid loan date ranges. Indexes on frequently queried columns including book titles, authors, ISBNs, and loan timestamps ensure responsive query performance even as data volumes grow. This separation ensures credential breaches cannot expose operational library data while main-

taining data integrity for domain operations.

Development processes emphasized modern engineering practices balancing academic learning with production readiness. We implemented comprehensive testing strategies including unit testing with JUnit and pytest achieving targeted coverage metrics, acceptance testing with Cucumber validating business workflows, and performance testing with JMeter evaluating system behavior under load. Test-driven development principles guided implementation where feasible, with tests written before code to drive modular design and clear interfaces.

Continuous integration through GitHub Actions automated test execution and validation on each code commit, providing rapid feedback on integration issues. The pipeline included parallel execution of Python and Java test suites, Docker image building with security scanning, and deployment validation using Docker Compose. Containerization with Docker ensured environment consistency across development and deployment stages, while Docker Compose orchestrated multi-service deployment with proper dependency management and health checks. These practices aimed to balance academic learning objectives with production-quality engineering standards while providing practical experience with modern DevOps toolchains.

III. RESULTS AND DISCUSSION

The architectural implementation was validated through comprehensive testing examining functionality, performance, and reliability characteristics from multiple perspectives. Unit testing achieved 96% overall code coverage across 128 test cases in the domain service, with critical modules demonstrating strong coverage metrics. Authentication utilities reached 95% coverage through comprehensive testing of token generation, parsing, and validation logic including edge cases for expired and malformed tokens. Book CRUD operations achieved 100% coverage with tests validating all database interaction paths including creation, retrieval, updating, and deletion scenarios with proper error handling.

Loan management functionality reached 96% coverage with tests verifying business logic including availability checking, due date calculation, and status transitions throughout the loan lifecycle. These tests included validation of constraint enforcement, such as preventing loans when no copies are available or when patrons have reached maximum allowable active loans. The comprehensive nature of these tests provided confidence that core business rules were correctly implemented and would behave predictably under various usage scenarios.

Coverage analysis revealed specific gaps requiring attention for quality improvement. Category operations reached 55% coverage, representing the main coverage gap, though these are non-critical operations. The 4% overall coverage gap (19 missed statements across 530 total) primarily involves error handling paths, edge cases, and utility functions rather than core business logic. While these gaps don't impact normal operation functionality, they represent areas where undetected bugs could emerge under unusual conditions or during system

evolution. Achieving 98%+ overall coverage would require approximately 10-15 additional tests focusing on error conditions and data transformation functions, estimated at 4-6 hours of development effort based on existing test implementation patterns.

Acceptance testing validated end-to-end business scenarios from user perspective, achieving 100% pass rate across all critical workflows. Five feature files specified scenarios in business-readable Gherkin syntax covering authentication workflows including successful login, invalid credential handling, and session management; catalog browsing with search, filtering, and availability checking; loan creation with validation of patron eligibility and copy availability; return processing with status updates and fine calculation where applicable; and role-based access control differentiating patron and librarian capabilities.

Acceptance test execution against fully deployed services validated integrated system behavior including inter-service communication, database persistence, and complete user workflows. The 100% pass rate demonstrates functional completeness where all specified business workflows operate correctly from end-user perspective. During development, acceptance test failures primarily identified integration issues between services, validating the importance of this testing level for distributed systems where components evolve independently. Test execution time averaged 3.5 minutes for the complete suite, enabling rapid feedback during iterative development.

Performance testing evaluated system behavior under simulated load conditions using Apache JMeter with configuration reflecting realistic usage patterns. Tests employed five concurrent virtual users executing mixed request patterns over 60-second duration, including both authenticated domain operations and unauthenticated security validation requests. The request mix reflected typical library usage with 40% catalog browsing operations, 30% loan-related actions, 20% authentication requests, and 10% administrative functions, distributed across virtual users to simulate realistic concurrency patterns.

Results demonstrated 95% success rate for authenticated requests with performance metrics indicating responsive system behavior. Average response time of 4.84ms significantly outperformed the 50ms target threshold for web applications, indicating excellent responsiveness even with inter-service communication overhead. The 95th percentile response time of 8.00ms showed consistent performance with minimal outliers, while maximum response time of 52ms remained within acceptable bounds for worst-case scenarios. Throughput measurements of 22.66 requests per second indicated healthy system capacity under the tested load conditions.

These performance characteristics translate to support for approximately 100 concurrent users based on typical library usage patterns where active users perform 2-3 operations per minute during engagement sessions. This capacity aligns well with institutional requirements for the target deployment environment while leaving headroom for usage growth. Performance analysis identified token validation as the primary contributor to response time, accounting for 4-8ms of each

TABLE I
PERFORMANCE TEST RESULTS SUMMARY

Performance Metrics	
Test Configuration	5 concurrent users, 60s duration
Request Mix	Catalog (40%), Loans (30%), Auth (20%), Admin (10%)
Successful Requests	57/60 (95%)
Average Response Time	4.84 ms
95th Percentile Response	8.00 ms
Maximum Response Time	52 ms
Throughput	22.66 requests/second
Error Rate	5% (primarily database contention)
Capacity Estimate	~100 concurrent users

authenticated request's latency through synchronous inter-service communication.

The architectural decision delivered measurable benefits validating several design hypotheses formulated during the evaluation phase. Security isolation ensures authentication database breaches cannot expose operational library data, reducing breach impact radius and simplifying security hardening efforts focused specifically on credential protection. This separation enables different security postures for each database, with stricter access controls and enhanced monitoring for authentication data versus operational data. The architectural boundary creates natural containment that limits attack surface and simplifies compliance with data protection requirements.

Independent scaling capability addresses different load characteristics observed in library systems where authentication experiences burst load during session starts while domain operations maintain steadier throughput during active usage. This separation enables targeted scaling decisions based on actual usage patterns rather than monolithic scaling that replicates all components regardless of actual needs. Performance testing confirmed that domain service scalability would become the primary consideration for increased loads, with authentication service requiring scaling only at substantially higher user counts due to its stateless design and efficient token validation.

Clear separation of concerns simplified development by allowing focused expertise application where security specialists could enhance authentication mechanisms independently from domain developers implementing library business logic. This division reduced cognitive load for developers and enabled parallel work streams with minimal coordination overhead once API contracts were established. The bounded contexts aligned naturally with different skill sets within the team, with Java/Spring expertise applied to authentication challenges and Python/data modeling expertise focused on domain operations.

Failure containment emerged as an additional benefit where authentication service maintenance windows or failures did not require domain service downtime, enabling graceful degradation rather than complete system unavailability. During testing, simulated authentication service failures resulted in failed new logins but allowed continued use for already authenticated sessions until token expiration. This partial availability characteristic provides operational flexibility for maintenance

scheduling and improves overall system resilience compared to monolithic approaches where any component failure typically affects entire system availability.

The architecture introduced operational challenges requiring acknowledgment and mitigation strategies. Development time increased approximately 60% compared to monolithic estimates due to implementing inter-service communication patterns, coordination mechanisms, and distributed debugging approaches. This overhead manifested primarily in integration testing, deployment orchestration, and issue diagnosis across service boundaries. While substantial, this cost was anticipated during architectural evaluation and was considered acceptable given security and scalability benefits.

Data consistency for user information spanning services requires careful coordination without transactional guarantees, implemented through synchronization patterns and eventual consistency approaches. User profile updates require propagation between authentication and domain services, with current implementation using synchronous updates that maintain consistency but introduce coupling. Future improvements could implement event-driven synchronization for reduced coupling while maintaining acceptable consistency levels for library operations where minor synchronization delays are tolerable.

Network latency adds measurable overhead to each operation requiring inter-service communication, though performance testing shows this remains within acceptable limits at current scale. Token validation contributes 4-8ms to each authenticated request, representing the majority of response time for simple operations. While acceptable for current user loads, this overhead would become more significant at higher scales, suggesting future optimization through token caching or API Gateway patterns that reduce inter-service calls.

Debugging complexity increases requiring correlation across separate services, addressed through logging standards and correlation identifiers that track requests across service boundaries. Distributed tracing implementation would further improve debugging capabilities but was beyond project scope. Current debugging requires examining multiple log files and correlating timestamps, which increases time to diagnose issues compared to monolithic systems with centralized logging.

Comparative analysis with monolithic alternatives reveals important considerations for architectural decision-making in similar institutional contexts. A monolithic approach would have provided faster initial development estimated at four versus eight weeks, simpler operational management with single deployment unit, transactional consistency guarantees through database ACID properties, and easier debugging with single codebase and unified logging. These advantages make monolithic architectures appealing for projects with tight timelines, small teams, or limited operational experience with distributed systems.

However, the monolithic architecture would sacrifice security isolation where database breaches expose both credentials and operational data simultaneously, creating higher impact security incidents. Independent scaling capability would be eliminated, requiring vertical scaling of entire application

regardless of which components actually experience load increases. Architectural clarity would be reduced through mixing security and domain concerns within single codebase, potentially increasing long-term maintenance complexity. Failure radius would expand where any component failure could affect entire system availability rather than being contained within service boundaries.

The dual-service approach prioritized long-term institutional requirements over short-term development convenience, particularly given security sensitivity of patron data and growth projections suggesting scaling needs. For the target deployment environment with approximately 100 concurrent users and sensitivity around credential protection, the security benefits justified the additional complexity. In contexts with different priorities—such as rapid prototyping, temporary systems, or environments with robust existing security controls—the trade-off analysis might favor monolithic approaches.

Our implementation experience suggests service separation provides value even at smaller institutional scales when specific conditions exist. Security requirements emphasizing credential protection justify separation despite complexity costs, particularly in educational contexts with regulatory compliance considerations. Anticipated growth patterns suggesting different scaling needs between system components support modular architectures that enable targeted scaling investments. Technology differentiation needs, such as specialized frameworks for authentication versus domain operations, benefit from service boundaries that prevent framework conflicts and enable technology optimization.

However, projects prioritizing rapid development velocity, maintaining small co-located teams, or operating in environments with limited DevOps maturity might prefer monolithic approaches with strong modular boundaries within single codebase. The key insight is that architectural decisions should be context-dependent rather than following generic prescriptions, with explicit evaluation of trade-offs against specific project requirements and constraints. The dual-service architecture proved appropriate for BookWiseUD given its security requirements, academic learning objectives, and institutional context, but different contexts might warrant different architectural choices.

IV. CONCLUSIONS

BookWiseUD successfully implements a functional library management system addressing core requirements for institutional libraries while demonstrating modern software engineering practices. The system achieves comprehensive automation of catalog management, loan processing, and patron information management through web-based interfaces accessible to both patrons and librarians across device types. Implementation rigor is demonstrated through 96% unit test coverage across 128 tests validating component functionality, 100% acceptance test pass rate confirming business requirement satisfaction, and performance characteristics supporting approximately 100 concurrent users with 4.84ms average response time under simulated load conditions.

The architectural decision to implement dual-service separation between authentication and domain logic delivered measurable benefits validating the design approach for the specific institutional context. Security isolation provides critical protection where credential breaches cannot expose operational library data, essential for educational institutions managing sensitive patron information under increasing regulatory scrutiny. Independent scaling capability addresses institutional growth projections where user demand may increase substantially without requiring architectural redesign, enabling cost-effective capacity expansion aligned with actual usage patterns. Clear separation of concerns enabled focused development where security and domain expertise could proceed independently, reducing coordination overhead while maintaining system integrity through well-defined API contracts.

The architecture introduced operational challenges including development time increases estimated at 60% over monolithic approaches, data consistency coordination requirements for information spanning service boundaries, network latency overhead from inter-service communication, and distributed debugging complexity requiring correlation across components. These challenges represent acceptable trade-offs for architectural benefits when institutional requirements prioritize security isolation and independent scaling over development simplicity. Mitigation strategies including comprehensive testing, containerized deployment, and correlation identifiers helped manage these challenges while maintaining development velocity and system reliability.

Our implementation experience demonstrates that service-oriented architectures provide value even at smaller institutional scales when business requirements emphasize specific characteristics including security, modularity, and growth capacity. The key success factor was explicit evaluation of architectural trade-offs against project-specific requirements rather than following generic architectural trends. For similar projects in educational contexts, we recommend conducting structured architectural evaluation considering security requirements, scalability needs, team capabilities, and operational constraints to determine whether service separation justifies its complexity costs.

Future work should address identified enhancements while preserving architectural benefits. Implementing API Gateway patterns would provide centralized JWT validation caching, reducing inter-service communication overhead by approximately 80% based on performance analysis. Event-driven user synchronization using message brokers would achieve eventual consistency for data spanning services without tight coupling, improving system evolvability. Centralized logging platforms would simplify debugging by aggregating logs from all services into unified view with distributed tracing capabilities. Expanding test coverage to 98%+ across all modules, particularly in category operations, would strengthen validation and support maintenance activities.

Additional research opportunities include longitudinal study of operational complexity in production deployment, comparative analysis of different service boundary definitions using

Domain-Driven Design principles, and investigation of alternative communication patterns beyond synchronous REST APIs. The system provides foundation for exploring these directions while delivering immediate value through automated library management capabilities. These improvements would evolve the architecture from academic implementation to production-ready system supporting larger institutional deployments with enhanced monitoring, scalability, and maintainability characteristics.

The project demonstrates that rigorous engineering practices combined with honest evaluation of architectural decisions can produce systems that are both educationally valuable and practically functional. These lessons inform not only library management systems but distributed system development more broadly, particularly in resource-constrained institutional contexts where requirements must be balanced against practical constraints. By documenting both successes and challenges, this work contributes to the body of knowledge around practical microservices implementation in academic environments where learning objectives intersect with real-world problem solving.

ACKNOWLEDGMENT

We thank Eng. Carlos Andres Sierra for guidance on software engineering practices and architectural decision-making throughout this project. We also acknowledge Universidad Distrital Francisco José de Caldas for providing the academic context supporting this work.

REFERENCES

- [1] R. T. Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*. Ph.D. dissertation, University of California, 2000.
- [2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.
- [3] M. Fowler, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2018.
- [4] M. Fowler and J. Lewis, "Microservices," 2014. [Online]. Available: <https://martinfowler.com/articles/microservices.html>
- [5] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT), RFC 7519," 2015. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc7519>
- [6] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Addison-Wesley, 2021.
- [7] R. S. Pressman and B. Maxim, *Software Engineering: A Practitioner's Approach*, 9th ed. McGraw-Hill, 2019.
- [8] C. J. Date, *Database Design and Relational Theory: Normal Forms and All That Jazz*. O'Reilly Media, 2019.