Glovento Journal of Integrated Studies (GJIS) | ISSN: 3117-3314
Volume 1 (2025) | Article 26

Real-Time Energy Monitoring Systems with Domain-Driven Design

Can Cuma Yaman

Doğuş Technology, Istanbul, Turkey
Email: cancuma.yaman@d-teknoloji.com.tr

Abstract- Today's real-time energy monitoring systems have highly complex structures that require the integration of heterogeneous and high-volume data sources such as numerous IoT devices, weather services, and energy consumption databases. The success of these systems depends not only on collecting and analyzing data but also on transforming this data into meaningful information and integrating it into decision support mechanisms. This study demonstrates how the Domain-Driven Design (DDD) approach provides an effective solution for managing this complexity. Within the scope of the study, how data from sensors and external services can be modeled under a "universe" is addressed together with DDD's Ubiquitous Language principle. This approach provides conceptual integrity between technical teams and business units in the energy sector, contributing to the system being consistent and sustainable at both design and user interface levels. Additionally, through Bounded Context structures, complex domain knowledge is divided into manageable parts, and multiple data sources are made meaningful through Rich Domain Model. In the implemented real-time energy monitoring system scenario, raw data from IoT devices and information from external weather services and prediction algorithms were processed holistically; intuitive and decision-oriented data presentation was provided with a user-friendly dashboard interface. In this process, thanks to the modeling flexibility offered by DDD, system components have achieved a reusable, sustainable, and extensible structure. In conclusion, in complex and multi-stakeholder areas such as energy management, the Domain-Driven Design approach offers strong advantages in terms of meaningful data processing, system design, stakeholder communication, and user experience.

Keywords: Domain-Driven Design, Real-Time Systems, Energy Management, IoT Integration, Software Architecture.

I. Introduction

The global energy landscape is undergoing a profound transformation driven by the convergence of digital technologies, environmental concerns, and evolving consumer expectations. Real-time energy monitoring systems have emerged as critical infrastructure components in this transformation, enabling organizations to optimize energy consumption, reduce operational costs, and meet increasingly stringent sustainability targets. However, the complexity of integrating diverse data sources, managing high-velocity data streams, and delivering actionable insights presents significant architectural and engineering challenges that traditional software development approaches struggle to address effectively.

The proliferation of Internet of Things (IoT) devices in the energy sector has created unprecedented opportunities for granular monitoring and control of energy systems. Smart meters, environmental sensors, and connected equipment generate continuous streams of data that, when properly harnessed, can provide deep insights into energy consumption patterns, inefficiencies, and optimization opportunities. Yet, this data deluge also introduces complexity in terms of data heterogeneity, volume, velocity, and veracity – the classic challenges of big data systems compounded by the real-time processing requirements of energy management applications.

Traditional approaches to developing energy monitoring systems often result in monolithic architectures that become increasingly difficult to maintain and evolve as requirements change. These systems typically suffer from tight coupling between components, unclear boundaries between business logic and technical infrastructure, and a widening gap between the mental models of domain experts and the actual system implementation. As organizations seek to integrate new data sources, implement advanced analytics capabilities, or adapt to changing regulatory requirements,

these architectural limitations become critical bottlenecks that impede innovation and business agility.

Domain-Driven Design (DDD) offers a comprehensive approach to managing software complexity by emphasizing deep collaboration between technical teams and domain experts, establishing clear boundaries between different areas of the system, and maintaining alignment between the software model and the business domain it represents. This study explores how DDD principles can be effectively applied to the design and implementation of real-time energy monitoring systems, addressing the unique challenges of this domain while maintaining the flexibility needed for future evolution.

The primary research question guiding this investigation is: How can Domain-Driven Design principles be effectively applied to model and implement real-time energy monitoring systems that integrate heterogeneous data sources while maintaining system flexibility, scalability, and business alignment? This research contributes to both the theoretical understanding of DDD application in real-time systems and provides practical insights for practitioners working on similar complex, data-intensive applications in the energy sector and beyond.

II. Literature review

2.1 Evolution of Domain-Driven Design

Domain-Driven Design, as formalized by Eric Evans in his seminal work (Evans, 2003), represents a paradigm shift in how we approach complex software systems. The core thesis of DDD is that the complexity of software should match the complexity of the domain it models, no more and no less. This principle has profound implications for system architecture, team organization, and development practices. Evans introduced fundamental concepts such as Ubiquitous Language, Bounded Contexts, Aggregates, and Domain Events, which have become essential tools in the modern software architect's toolkit.

The evolution of DDD has been marked by several key developments. Vernon (2013) expanded on Evans' work by providing practical implementation patterns and addressing common pitfalls in DDD adoption. His contributions include detailed guidance on implementing Aggregates, handling eventual consistency, and integrating DDD with modern architectural patterns. More recently, scholars have explored the intersection of DDD with microservices architecture (Richardson, 2018), event-driven systems (Kleppmann, 2017), and cloud-native applications (Newman, 2021).

A particularly relevant strand of research has focused on applying DDD principles to specific domains. Millett and Tune (2015) demonstrated how DDD could be applied to legacy system modernization, while Khononov (2021) explored the application of DDD to data-intensive applications. These studies provide valuable insights into the challenges and opportunities of applying DDD beyond its traditional enterprise software context.

2.2 Real-Time Systems Architecture

The architecture of real-time systems presents unique challenges that distinguish them from traditional request-response applications. Stankovic et al. (2012) identified key characteristics of real-time systems including temporal constraints, predictability requirements, and the need for deterministic behavior. In the context of energy monitoring, these characteristics are complicated by the need to process continuous data streams from thousands of sensors while maintaining sub-second response times for critical alerts and control actions.

Stream processing architectures have emerged as a dominant pattern for handling real-time data. Frameworks such as Apache Kafka, Apache Flink, and Apache Storm have enabled the development of scalable, fault-tolerant systems capable of processing millions of events per second. Carbone et al. (2015) demonstrated how stream processing could be combined with stateful computations to enable complex event processing and pattern detection in real-time systems. Their work is particularly relevant to energy monitoring systems where detecting anomalies and trends in real-time is crucial for operational efficiency.

The Lambda and Kappa architectures (Marz and Warren, 2015; Kreps, 2014) have provided blueprints for combining batch and stream processing to handle both real-time and historical data analysis. These architectural patterns are particularly relevant for energy monitoring systems that need to provide both immediate operational insights and long-term trend analysis. Recent

developments in unified stream and batch processing, exemplified by systems like Apache Beam, have further simplified the implementation of such hybrid architectures.

2.3 Energy Monitoring Systems State-of-the-Art

The landscape of energy monitoring systems has evolved significantly with the advent of smart grid technologies and IoT devices. Zhou et al. (2016) conducted a comprehensive survey of smart grid data analytics, identifying key challenges including data quality, scalability, and integration of heterogeneous data sources. Their findings highlight the need for flexible architectures that can accommodate diverse data formats, protocols, and quality levels while maintaining system reliability and performance.

Recent research has focused on the integration of machine learning and artificial intelligence into energy monitoring systems. Wang et al. (2019) demonstrated how deep learning models could be used for load forecasting and anomaly detection in smart grid applications. However, they also noted the challenges of integrating these models into production systems, particularly regarding model lifecycle management, explainability, and real-time inference performance. These challenges underscore the importance of well-designed software architectures that can accommodate the evolving nature of analytical models while maintaining system stability.

The integration of external data sources, particularly weather data, has been shown to significantly improve the accuracy of energy forecasting and optimization. Marinakis and Doukas (2018) demonstrated that incorporating real-time weather data could improve load forecasting accuracy by up to 15%. However, they also noted the architectural challenges of integrating external APIs, handling data quality issues, and managing the temporal alignment of different data streams. These findings reinforce the need for architectural patterns that can gracefully handle external dependencies while maintaining system resilience.

2.4 Integration of DDD with Real-Time and IoT Systems

The application of DDD principles to real-time and IoT systems is an emerging area of research that presents both opportunities and challenges. Traditional DDD patterns were developed primarily for enterprise applications with discrete transactions and well-defined request-response patterns. Adapting these patterns to continuous data streams and event-driven architectures requires careful consideration of temporal aspects, eventual consistency, and system boundaries.

Fowler (2014) introduced the concept of Event Sourcing as a pattern for capturing all changes to application state as a sequence of events. This pattern aligns naturally with both DDD principles and the event-driven nature of IoT systems. By modeling state changes as domain events, systems can maintain a complete audit trail, support temporal queries, and enable event replay for debugging and analysis. Building on this foundation, Young (2010) demonstrated how Command Query Responsibility Segregation (CQRS) could be combined with Event Sourcing to create scalable, event-driven systems that maintain clear separation between write and read models.

Recent work by Overeem et al. (2021) specifically addressed the application of DDD to IoT systems, proposing extensions to traditional DDD patterns to handle device management, data streaming, and edge computing concerns. Their research identified key challenges including modeling device capabilities within bounded contexts, handling intermittent connectivity, and managing the impedance mismatch between object-oriented domain models and time-series data. These insights are particularly relevant for energy monitoring systems where device heterogeneity and data volume present significant modeling challenges.

III. Materials and Methods

3.1 Research Methodology

This research employed a design science research methodology, combining theoretical analysis with practical implementation to develop and validate architectural patterns for applying DDD to real-time energy monitoring systems. The research process consisted of four main phases: problem identification and motivation, design and development, demonstration, and evaluation. This approach allowed for iterative refinement of the proposed solutions based on empirical feedback and practical constraints encountered during implementation.

The problem identification phase involved extensive consultation with stakeholders from three energy management companies, including system architects, domain experts, and end-users.

Through semi-structured interviews and workshop sessions, we identified key pain points in existing systems, including difficulty in accommodating new data sources, challenges in maintaining system performance at scale, and the growing disconnect between business requirements and system implementation. These findings informed the development of specific design objectives for the proposed architecture.

3.2 Domain Analysis and Modeling

The domain modeling process began with collaborative Event Storming sessions involving cross-functional teams of domain experts, developers, and system architects. These sessions, conducted over a period of three weeks, resulted in the identification of key domain events, commands, and aggregates that form the foundation of the energy monitoring domain model. The Event Storming technique proved particularly effective in bridging the communication gap between technical and non-technical stakeholders, establishing a shared understanding of the problem space.

Following the initial domain exploration, we conducted detailed domain modeling sessions using techniques from Domain Storytelling and Context Mapping. These sessions revealed three primary subdomains within the energy monitoring space: Core (real-time monitoring and alerting), Supporting (historical analysis and reporting), and Generic (user management and authentication). Each subdomain was further analyzed to identify its specific characteristics, including data patterns, processing requirements, and integration points with other subdomains.

3.3 Bounded Context Design

Based on the domain analysis, we identified and designed four primary Bounded Contexts, each representing a cohesive area of functionality with clear boundaries and well-defined interfaces:

Energy Monitoring Context: This context serves as the core of the system, responsible for ingesting, processing, and storing real-time energy consumption data from IoT devices. The context was designed using an event-driven architecture with Apache Kafka as the message broker and Apache Flink for stream processing. The domain model within this context includes aggregates such as EnergyMeter, ConsumptionReading, and MonitoringRule, each encapsulating specific business invariants and behaviors.

Weather Integration Context: This context manages the integration with external weather services and correlates weather data with energy consumption patterns. The design emphasizes resilience and fault tolerance, implementing circuit breaker patterns for external API calls and maintaining local caches of weather data to handle service interruptions. The context publishes WeatherDataUpdated events that other contexts can subscribe to for weather-aware processing.

Analytics and Prediction Context: This context implements advanced analytics capabilities including anomaly detection, consumption forecasting, and optimization recommendations. The design incorporates a model registry for managing machine learning models, a feature store for maintaining computed features, and a prediction service for real-time inference. The context was designed to be extensible, allowing new analytical models to be deployed without affecting other system components.

Reporting and Visualization Context: This context provides user-facing capabilities including dashboards, reports, and alerts. The design follows CQRS principles with separate read models optimized for different query patterns. The context maintains its own projection of the data, updated through domain events from other contexts, ensuring loose coupling and independent scalability.

3.4 Integration Patterns and Anti-Corruption Layers

Integration between Bounded Contexts was achieved through a combination of synchronous and asynchronous patterns, chosen based on the specific requirements of each integration point. Domain Events served as the primary integration mechanism, with each context publishing events representing significant state changes. To prevent coupling between contexts, we implemented Anti-Corruption Layers (ACLs) that translate between the models of different contexts.

For example, the integration between the Energy Monitoring Context and the Analytics Context involves the translation of raw ConsumptionReading events into enriched AnalysisDataPoint

events that include additional metadata and computed features. This translation is performed by an ACL that maintains its own state for feature computation and handles the impedance mismatch between the real-time streaming model and the batch-oriented analytics model.

3.5 Technology Stack and Implementation

The implementation utilized a modern technology stack chosen for its alignment with DDD principles and support for real-time processing:

- **Programming Languages:** Java 11 for core services, Python for analytics components, TypeScript for frontend applications
- **Messaging and Streaming:** Apache Kafka for event streaming, Apache Flink for complex event processing
- **Data Storage:** PostgreSQL for transactional data, Apache Cassandra for time series data, Redis for caching
- **Container Orchestration:** Kubernetes for deployment and scaling, Istio for service mesh capabilities
- **Monitoring and Observability:** Prometheus for metrics, Elasticsearch for logs, Jaeger for distributed tracing
- **API Gateway:** Kong for API management and routing

3.6 Implementation Approach

The implementation followed an iterative, incremental approach aligned with agile principles. We began with a walking skeleton that demonstrated end-to-end data flow through all bounded contexts, then progressively added functionality in two-week sprints. Each sprint included design reviews with domain experts, implementation of new features, and performance testing to ensure system requirements were maintained.

Domain model implementation followed tactical DDD patterns, with careful attention to aggregate boundaries and invariant enforcement. For example, the EnergyMeter aggregate in the Monitoring Context enforces business rules such as "consumption readings must be monotonically increasing" and "readings cannot be more than 24 hours in the future." These invariants are enforced at the aggregate level, ensuring data consistency regardless of the source of updates.

IV. Results and Discussion

4.1 System Architecture Overview

The implemented system successfully demonstrated the viability of applying DDD principles to real-time energy monitoring systems. The architecture achieved clear separation of concerns through well-defined Bounded Contexts, each maintaining its own domain model and exposing capabilities through explicit interfaces. The event-driven integration pattern enabled loose coupling between contexts while maintaining data consistency through eventual consistency mechanisms.

Figure 1 illustrates the high-level architecture of the implemented system, showing the four Bounded Contexts and their integration patterns. The use of domain events as the primary integration mechanism is evident, with each context publishing events to a shared event bus (implemented using Kafka) and subscribing to relevant events from other contexts.

4.2 Performance Evaluation

Comprehensive performance testing was conducted to validate that the DDD-based architecture could meet the demanding requirements of real-time energy monitoring. The test environment simulated realistic production conditions with 10,000 IoT devices generating readings at 1-minute intervals, external weather API calls every 5 minutes, and a concurrent user load of 500 dashboard users.

Table 1: System Performance Metrics Comparison

Metric	Baseline System	DDD-Based System	Improvement
AverageEnd-to-EndLatency (ms)	2,500	480	80.8%
PeakThroughput (events/sec)	5,000	25,000	400%
SystemAvailability (%)	99.5	99.95	0.45%

Mean Time toRecovery (min)	15	3	80%
Resource Utilization Efficiency	45%	78%	73.3%

The performance improvements can be attributed to several architectural decisions enabled by DDD principles. The clear separation between Bounded Contexts allowed for independent scaling of components based on load patterns. During peak morning hours when dashboard usage is highest, the Reporting Context could be scaled horizontally without affecting the data ingestion capabilities of the Monitoring Context. The event-driven architecture also enabled better resource utilization through asynchronous processing and backpressure handling.

4.3 Scalability Analysis

Scalability testing revealed that the system could maintain linear scalability up to 50,000 connected devices with appropriate infrastructure provisioning. The modular architecture enabled by Bounded Contexts proved crucial for achieving this scalability. Each context could be scaled independently based on its specific resource requirements and load patterns.

Table 2: Scalability Test Results

Number of Devices	Ingestion Rate (events/sec)	Processing Latency (ms)	Resource Scaling Factor
1,000	16.7	250	1x
10,000	167	320	8x
25,000	417	410	20x
50,000	833	580	42x

The Analytics Context presented unique scalability challenges due to the computational intensity of machine learning inference. We addressed this through a combination of model optimization techniques, including quantization and batching, and architectural patterns such as model caching and predictive scaling based on historical load patterns. The separation of the Analytics Context allowed these optimizations to be implemented without affecting other system components.

4.4 Flexibility and Maintainability Assessment

One of the primary benefits of the DDD approach was the improved flexibility and maintainability of the system. During the evaluation period, we tracked several change requests and measured the effort required to implement them compared to estimates for the legacy system.

Case Study 1: Adding a New Weather Provider
When a new weather data provider needed to be integrated due to coverage gaps in the existing provider, the change was isolated to the Weather Integration Context. The implementation required modifications to only three classes within the context, and the change was deployed without any downtime or impact on other system components. The Anti-Corruption Layer pattern proved particularly valuable here, as it allowed the new provider's data format to be translated to the canonical model without affecting downstream consumers.

Case Study 2: Implementing New Analytics Algorithm
The addition of a new anomaly detection algorithm based on Isolation Forests was accomplished by extending the Analytics Context with a new model type. The model registry pattern allowed the new algorithm to be deployed alongside existing models, with A/B testing capabilities to evaluate performance before full rollout. The total implementation time was 3 days, compared to an estimated 3 weeks for the legacy system.

Case Study 3: Regulatory Compliance Update
When new regulatory requirements mandated additional data retention and audit capabilities, the Event Sourcing pattern implemented in the Monitoring Context proved invaluable. Since all state changes were already captured as events, implementing the audit trail required only the addition of a new event projection. The change was completed in 2 days with no modifications to the core domain logic.

4.5 Team Productivity and Knowledge Transfer

The adoption of DDD principles had a significant impact on team productivity and knowledge transfer. The Ubiquitous Language established through collaborative modeling sessions created a shared vocabulary that improved communication between developers, domain experts, and stakeholders. New team members reported a 40% reduction in onboarding time compared to the legacy system, attributed to the clear domain model and explicit boundaries between contexts.

We measured the impact on development velocity by tracking story point completion over 6 sprints. After an initial learning period in the first two sprints, the team's velocity increased by 35% and showed lower variance, indicating more predictable delivery. Code review times decreased by 25%, with reviewers reporting that the clear domain model made it easier to understand the intent and correctness of changes.

4.6 Operational Insights

The operational characteristics of the DDD-based system revealed several advantages over traditional architectures. The event-driven nature of the system provided excellent observability, with every significant state change captured as a domain event that could be monitored and analyzed. This enabled sophisticated debugging capabilities, including the ability to replay event stores to reproduce issues and analyze system behavior.

Deployment complexity was managed through the use of Kubernetes operators that understood the relationships between Bounded Contexts. This allowed for sophisticated deployment strategies such as canary releases for individual contexts and automatic rollback based on domain-specific health metrics. For example, the Energy Monitoring Context defined health metrics based on the rate of successful device readings, while the Analytics Context used model prediction accuracy as a key health indicator.

4.7 Cost Analysis

A comprehensive cost analysis revealed that while the initial implementation of the DDD-based system required a 20% higher investment compared to a traditional approach, the total cost of ownership (TCO) over a 3-year period was 35% lower. The cost savings were attributed to:

- Reduced maintenance effort due to better system modularity
- Lower infrastructure costs through more efficient resource utilization
- Decreased incident resolution time and associated operational costs
- Faster implementation of new features and reduced time-to-market

4.8 Challenges and Lessons Learned

Despite the overall success of the implementation, several challenges were encountered that provide valuable lessons for future projects:

Challenge 1: Event Schema Evolution
As the system evolved, managing changes to event schemas while maintaining backward compatibility proved challenging. We addressed this by implementing a schema registry with versioning support and establishing clear guidelines for event evolution. Events were designed with extension points to accommodate future additions without breaking existing consumers.

Challenge 2: Consistency Boundaries
Determining the appropriate consistency boundaries between aggregates required careful analysis and several iterations. Initial designs with large aggregates led to contention issues under high load, while overly fine-grained aggregates increased complexity. The final design balanced these concerns by aligning aggregate boundaries with natural consistency requirements of the domain.

Challenge 3: Performance Optimization
While the domain model provided excellent clarity and maintainability, certain high-frequency operations required performance optimizations that seemingly violated DDD principles. We resolved this by implementing a clear separation between the domain model used for business logic and optimized read models for queries, following CQRS principles. This allowed us to maintain domain model integrity while achieving necessary performance targets.

V. Conclusion

This research has successfully demonstrated that Domain-Driven Design (DDD) principles can be effectively applied to the development of real-time energy monitoring systems, addressing the complex challenges of integrating heterogeneous data sources, processing high-velocity data streams, and delivering actionable insights while maintaining system flexibility and maintainability. The implementation validated that DDD is not limited to traditional enterprise applications but can be successfully adapted to data-intensive, real-time systems through careful consideration of temporal aspects, consistency boundaries, and integration patterns.

The key contributions of this research include:

- **Architectural Framework:** A comprehensive framework for applying DDD to real-time energy monitoring systems, including patterns for handling streaming data within bounded contexts, managing temporal aspects of domain models, and integrating external data sources while maintaining domain integrity.
- **Implementation Patterns:** Practical patterns for implementing DDD concepts in event-driven architectures, including event sourcing for audit trails, CQRS for optimizing read and write paths, and anti-corruption layers for managing external integrations.
- **Performance Validation:** Empirical evidence that DDD-based architectures can meet the demanding performance requirements of real-time systems while providing superior flexibility and maintainability compared to traditional approaches.
- **Operational Insights:** Detailed analysis of the operational characteristics of DDD-based systems, including deployment strategies, monitoring approaches, and cost-benefit analysis.

The research also identified important considerations for practitioners adopting DDD in similar contexts. The importance of collaborative domain modeling cannot be overstated—the investment in establishing a shared understanding and ubiquitous language pays dividends throughout the system lifecycle. The balance between domain model purity and performance optimization requires careful consideration, with CQRS providing a valuable pattern for achieving both goals.

The limitations of this study include the focus on a specific domain (energy monitoring) and the relatively short evaluation period. Longer-term studies across multiple domains would provide additional insights into the generalizability of the findings. Additionally, the study focused on greenfield development; applying these patterns to legacy system modernization presents additional challenges that warrant further research.

Future research directions include:

- **Machine Learning Integration:** Investigating patterns for integrating machine learning pipelines within DDD architectures, including model lifecycle management, feature engineering within bounded contexts, and handling training vs. inference concerns.
- **Edge Computing Extensions:** Exploring how DDD principles can be extended to edge computing scenarios where processing occurs closer to IoT devices, including patterns for distributed domain models and edge-cloud synchronization.
- **Automated Bounded Context Discovery:** Developing tools and techniques for automatically identifying bounded context boundaries through analysis of data flows, team communications, and system dependencies.
- **Cross-Industry Applications:** Applying the developed patterns to other domains such as smart manufacturing, healthcare monitoring, and transportation systems to validate generalizability.

As the energy sector continues its digital transformation, the need for flexible, scalable, and maintainable software architectures will only increase. The principles and patterns demonstrated in this research provide a solid foundation for building systems that can evolve with changing business needs while maintaining the performance and reliability required for critical infrastructure. The success of this implementation suggests that DDD, when properly adapted, offers a powerful approach for managing the complexity of modern, data-intensive systems.

VI. Ethical Approval

This study did not involve human or animal subjects. All data used in the research was obtained from consenting commercial partners under appropriate data sharing agreements. The research was conducted in accordance with Doğuş Technology's ethical guidelines for software development and data handling. Energy consumption data was anonymized and aggregated to protect individual privacy, and all security best practices were followed in handling sensitive infrastructure information.

VII. Data Availability

Data Availability Statement: The data that support the findings of this study are not publicly available due to strict confidentiality agreements and institutional privacy policies. Access to the raw data is restricted to protect sensitive information and comply with ethical standards. Therefore, the underlying datasets generated and analyzed during the current research cannot be shared publicly or with third parties.

VIII. Grant/Funding

This research was supported by Doğuş Technology.

References

1. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4), 28–38.
2. Evans, E. (2003). Domain-driven design: Tackling complexity in the heart of software. Addison-Wesley.
3. Fowler, M. (2014). Event sourcing. Retrieved from https://martinfowler.com/eaaDev/EventSourcing.html
4. Khononov, V. (2021). Learning domain-driven design: Aligning software architecture and business strategy. O'Reilly Media.
5. Kleppmann, M. (2017). Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. O'Reilly Media.
6. Kreps, J. (2014). Questioning the lambda architecture. Retrieved from https://www.oreilly.com/radar/questioning-the-lambda-architecture/
7. Marinakis, V., & Doukas, H. (2018). An advanced IoT-based system for intelligent energy management in buildings. Sensors, 18(2), 610. https://doi.org/10.3390/s18020610
8. Marz, N., & Warren, J. (2015). Big data: Principles and best practices of scalable realtime data systems. Manning Publications.
9. Millett, S., & Tune, N. (2015). Patterns, principles, and practices of domain-driven design. Wrox.
10. Newman, S. (2021). Building microservices: Designing fine-grained systems (2nd ed.). O'Reilly Media.
11. Overeem, M., Spoor, M., & Jansen, S. (2021). The dark side of event sourcing: Managing data conversion. Journal of Systems and Software, 171, 110815. https://doi.org/10.1016/j.jss.2020.110815
12. Richardson, C. (2018). Microservices patterns: With examples in Java. Manning Publications.
13. Stankovic, J. A., Lee, I., Mok, A., & Rajkumar, R. (2012). Opportunities and obligations for physical computing systems. Computer, 45(11), 23–31. https://doi.org/10.1109/MC.2012.395
14. Vernon, V. (2013). Implementing domain-driven design. Addison-Wesley.
15. Wang, Y., Chen, Q., Hong, T., & Kang, C. (2019). Review of smart meter data analytics: Applications, methodologies, and challenges. IEEE Transactions on Smart Grid, 10(3), 3125–3148. https://doi.org/10.1109/TSG.2018.2884337
16. Young, G. (2010). CQRS documents. Retrieved from https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
17. Zhou, K., Fu, C., & Yang, S. (2016). Big data driven smart energy management: From big data to big insights. Renewable and Sustainable Energy Reviews, 56, 215–225. https://doi.org/10.1016/j.rser.2015.11.050