ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article

Open Access

Designing and Building Large-Scale Distributed Enterprise Applications for Commerce

Mahidhar Mullapudi[1]*, Satish Kathiriya[2] and Rajath Karangara[3]

[1]Senior Software Engineer, Microsoft, WA, USA

[2]Software Engineer, CA

[3]American Express, U.S

ABSTRACT

Designing a large-scale enterprise application for e-commerce that can manage multiple millions of requests and manages multi-million dollars in revenue has lot of complexities. The critical part of developing and maintaining such systems needs a thoughtful design that can be scalable, reliable and flexible enough to handle changes as the product evolves over time. Many companies have developed their own architecture and systems for this use case; we propose a system that's designed and developed a few years ago which can handle millions of requests per day, rigorously tested with time on production and served various use cases and scenarios.

There are several components and services in this system that must work seamlessly, and decisions must be made while designing a system. In this paper, we discuss the overall architecture, components involved and important design decisions that affect their ease of use, flexibility, performance, scalability, reliability and fault-tolerance. We also discuss other approaches that can be used while making design decisions.

We then illustrate how our decisions and systems satisfy the requirements that can be used for designing and developing these large-scale enterprise applications for commerce stack.

***Corresponding author**

Mahidhar Mullapudi, Senior Software Engineer, Microsoft, WA, USA.

Received: June 06, 2023; **Accepted:** June 13, 2023; **Published:** June 20, 2023

Introduction

Commerce has transformed the global retail and online purchase industry. In 2023, more than three billion people purchased goods or services online across different platforms like Amazon, Walmart, Microsoft (online services), Etsy, Shopify etc., Global e-commerce retail sales surpassed $6.3 trillion in 2023, accounting for 20.8% of all global retail sales – up from 7.4% in 2015. That share increased to 22% in 2023 and projected to increase to 24% by 2026 [1-4].

Given the impact and the scale of the problem, there are some important qualities that play a critical role in designing large-scale enterprise applications.

Ease-of-Use: how complex are the requirements for inputting data into the system? What is the authoring and review process for the input? Does the system need to maintain versions of data for consistency and testing?

Flexibility: how often do the requirements change and how complex are those changes? Are there a variety of types of services and offerings in the system? Are there any services dependent on this data generated by this system? Is this design language agnostic?

Performance: how much latency is, ok? Seconds? Or minutes? How much throughput is required, per machine and in aggregate for each of these services? [5]

Scalability: how scalable should the system be, would the traffic increase two times every year or so? Can data be sharded and re-sharded to process partitions of it in parallel? How easily can the system adapt to changes in volume, both up and down? [1]

Reliability: does the system perform the function that the user expected? Can it tolerate the load and data volume?

Fault-Tolerance: what kinds of failures are tolerated? What semantics are guaranteed for the number of times that data is processed or output? How does the system store and recover in-memory state?

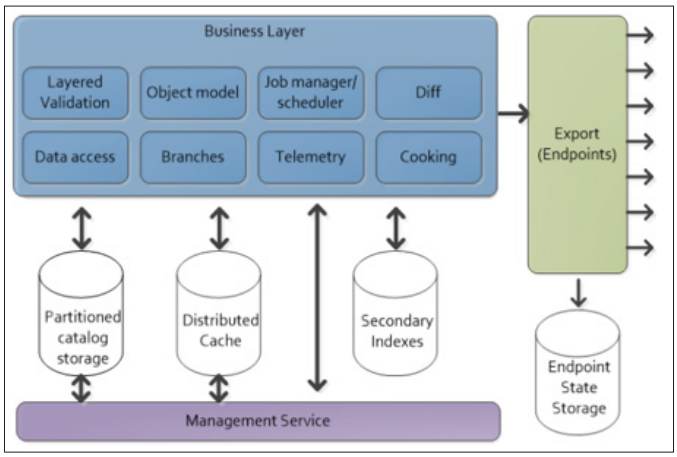
In this paper, we outline the overall architecture, overview of the components involved, design of the system and try to adhere to best practices and answer the above questions while making design decisions.

For any commerce application, the product would be a common point of transaction, needs distinct options or configurations in that

product, would need to maintain information about the prices per region, language or localization, tax codes, financial instructions, currency configuration based on the region that the service or product is being bought, promotions etc., To be able to configure these options, system needs a user interface that is simple enough to interact with and flexible enough to be scaled across different service families or product types. So, defining a schema for the model and having all the behaviors and dimensions to the model is of utmost importance. The data can change at any time and the system should provide an effortless way to evaluate and isolate revisions would be particularly important. The system should have review and manual approval process with auditing, as there is revenue impact for this data. Also, the system should easily transform and ingest this data into different downstream systems, so having a generic transformation and publishing pipeline which can validate and save the state of the publishing data is essential. We have discussed generic requirements for a commerce application and designing that system on a global scale would be a complex task. We present these requirements and design decisions in more detail after we describe the components in our system. We then reflect on lessons we learned over the last few years as we built and rebuilt these systems. One lesson is to place emphasis on ease of use: not just on the ease of writing applications, but also on the ease of testing, debugging, deploying, and finally monitoring hundreds of applications in production [6,7].

This paper is structured as follows. In Section 2, we provide an overview of the architecture and components involved in the system. In Section 3, we dive deep into some of the important components in the system. Then in Section 4, we reflect on lessons learned while designing and building large-scale distributed commerce applications. Finally, we conclude in Section 5.

Systems Overview



There are multiple systems involved in designing and building large scale enterprise applications for commerce. We present an overview of architecture and various components involved in the commerce eco-system.

Figure 1 illustrates the overall architecture of the system, individual components, or services and how data is modeled, transported across various components, and fed to dependent systems for processing. Main components include data Inputs, Modelling, Endpoints, Ingestion to downstream systems, and Business Layer that abstracts away validation, data management, job orchestration, diff and telemetry.

Data Intake

Intake form is a term that refers to how the data is captured to be inserted or updated into the system. This form can be any type of user interface – web application or just an excel with different columns or a web service that provides this data in Json format that can represent the intent for commerce. This data should be in a normalized format that can be used by business planners and other non-technical folks to convey the intent.

Modelling

Modelling is the core of the system that defines the schema to capture intent from the input data, other configurations that are required to maintain the products/services and transform this data to a generic format that can be transferred to downstream services. Here we discuss some key components that come under modelling.

Product

In the realm of commerce, a product is a fundamental entity that an online business offers for sale. Let's delve into some key aspects:

Product Types

Physical Products: These are tangible items that customers can touch, feel, and use. Examples include clothing, electronics, books, and household goods.

Digital Products: These are intangible goods delivered electronically. Examples include e-books, software licenses, music downloads, and online courses.

Services: Although products are not always considered, services (such as consulting, web design, or subscription-based services) are also part of the e-commerce landscape [8].

Some of the key properties in the product are:

Product Name: This can be defined as a localized string to maintain the name of the product.

Product Description: Localized string

SKU (Stock Keeping Unit): Unique identifier that helps keep track of inventory [9].

Region Configuration: Region Configuration includes data about the list of regions this product is available in.

Currency Configuration: Currency conversions for different market and regions.

Language Configuration: Language Configuration defines the list of languages and region-specific conversions.

Pricing: The cost associated with purchasing the product, often inclusive or exclusive of taxes and shipping fees.

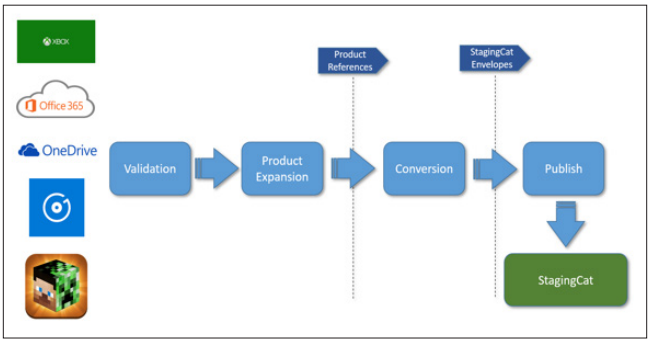
Availability: It indicates whether the product is in stock, in backorder, or out of stock.

Related Products: Recommendations for other products that are related or complementary to the one being viewed.

Product Lifecycle: It maintains information about the product's lifecycle, including state transitions: *Test* > *GA* > *Preview* > *Live*.

Financial Instructions: It gives information regarding the *revenue Skus*, *financial tax codes* etc.,

The above properties can be included when defining the model for *product*. This schema as you see is generic and normalized as this allows simplicity and ease of use to interact with and develop user interfaces. We will discuss how we can extend this model and data flow between different components



Cooking/Augmentation: In the intuitive model, data is modeled to reduce redundancy and express relationships that align to a business view. The process of Augmentation is a process of de-normalizing this data into a set of all unique permutations & combinations [10,11].

Conversion: The process of taking the unique permutations created by Augmentation and coercing them into a format understood by a downstream system.

Model Reference: a unique permutation of data yielded by process called Augmentation, which we will discuss later in this paper.

Dimension: An intrinsic property of a model reference. An option in Product. It functions as a measurement, a standalone value. Dimensions cause proliferation. There are several ways to go about this, from adding the interface to a dimension type.

Data Component: information about a model reference which does not cause proliferation and could be a standalone value or a set of values. Commonly known as "dumb data".

Behavior Component: the interface which allows a class to participate in the augmentation process by providing options and/or data [10].

Behavior Component Descriptor: informs sorting logic for augmentation process. Defines what behavior needs and provides that. Behaviors are then sorted according to these requirements.

Composite Behavior: participants contribute to other behaviors are composite behavior. In general, they do not have any logic executed during augmentation by themselves but can express conditions to process other behaviors.

Conditional behavior: A behavior which only applies under certain conditions defined by criteria.

Runtime Behaviors: Generic (as in Generic Type) behaviors which encapsulate common behavior patterns.

Criterion: The means by which participants in Augmentation are allowed to be conditional.

Applicability Criterion: behavior can be made conditional. When it is conditional, applicability criterion is defined and used as the criteria to determine if behavior should be included in the Augmentation process.

Item Criterion: item criterion, like applicability criterion, allows behaviors to be conditional. This form of criterion is used for composite behaviors. The difference is in the use case.

Applicability criterion says, "I am applicable in the following situations" whereas Item criterion says, "The behaviors I contribute are applicable in the following situations."

Business Layer

Business Layer can include various components that take the data model from user interfaces or data files or REST APIs from different partner services, process and capture the intent in the form defined in the schema and stored. We will discuss how this data is managed and stored in the next sections.

Here we talk about components that are generic and schema agnostic for most cases.

Layered Validation: validation framework which takes in the list of validators and performs checks to ensure the consistency and reliability of data at various stages in the transformation or augmentation process starting from the data inputs to all the way before landing at intermediate model.

Smart data diff: module that can smartly differentiate between different versions of data based on outputs that these versions generate and therefore allow users to identify the diffs of the changes in inputs.

Data Branches: to maintain different versions of data, we choose a concept of data branching which is like git branches and allows for modularizing data and provides fine grained control over the data that we are committing. This also allows us to maintain a review/approval process before publishing the data that has huge impact. We propose a concept of snapshotting and watermarks to maintain copies of data using references and pointers rather than maintaining entire copy [10].

Job orchestrator: to perform operations at scale while dealing with large set of products and option permutations, we need to distribute these cooking/augmentation processes across multiple nodes which needs a scheduler or an orchestration process to distribute load, allow the processes to run in isolation, maintain history or progress, track the jobs to retrieve and segregate the results [5].

Localization Import/Export: localization is to capture and apply different languages for some of the metadata about the product like title, description, Sku names, search words and other properties that need to be updated as per the area that is being sold in.

Cooking/Augmentation: process of Augmentation is de-normalizing the input data into a set of all unique permutations and combinations based on various configurations that are defined, using the real-time data to generate an intermediate model that can be stored or transformed to be ingested to different down streams. We will dive deep on this in later sections [11].

Telemetry: ease of debugging is the utmost priority while designing and building large-scale distributed systems so having an advanced framework to read and write telemetry/logs plays

a crucial role. We will discuss different options in later sections.

Data Management and Storage

In this section we talk about how to store and manage data for simplicity, reliability, consistency, and performance. The most important thing is to abstract away any model and business intent from the data management layer. So, we store the data as objects in memory and we refer to them as entities.

Entities

Entities can be thought of as addresses, pointers, or the keys to data. The entity object itself can be serialized, and then Base36 encoded to give a string-based address. This serialized entity as a string key is commonly passed around in Urls and scripts and de-serialized back to the entity object itself. An entity itself never exists or doesn't exist, in the same way you can't say a pointer exists or not in native languages.

Data Transfer Objects (DTOs)

DTOs serve as atomic units of data storage, serialized as a single block of data. They are stored and retrieved using the entity as the address in the underlying storage. Accessing DTOs is achieved through properties on entities generated by the Entity Designer. Entities may consist of zero or more DTOs, with DTO fields exposed through designer-generated data access properties. Entities lacking DTOs have only immutable key properties. The primary DTO, which has ACLs, determines the existence of data, and the Exists () method [12].

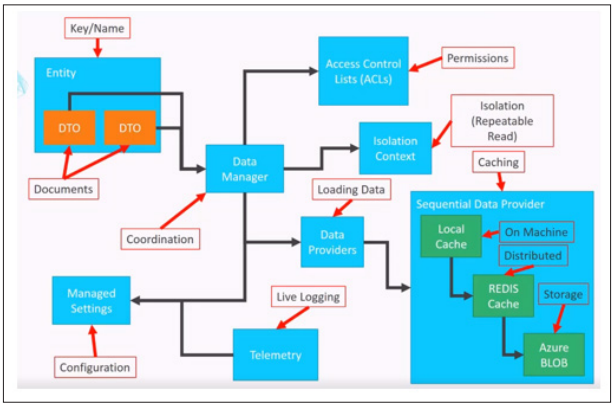
Direct Entities vs Isolation Context Entities

Entities are categorized as Direct (or System Entities) and Isolation Context (or 'Regular' Entities). Direct Entities: Manage their modification state independently. Changes made to properties can be persisted using *Save*() or discarded with *DiscardChanges*() [13].

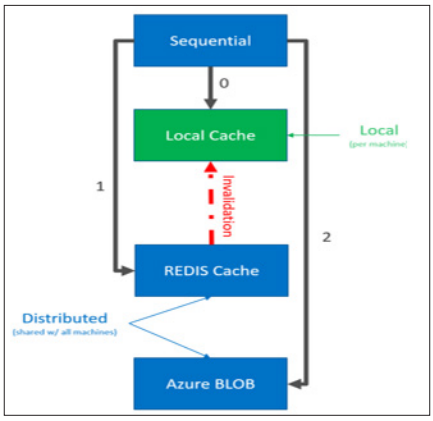
Isolation Context Entities

Modifications are held in an Isolation Context, lacking a direct Save() or *DiscardChanges*() interface. Changes can be persisted by calling *CommitChanges*() on the Isolation Context. Since modifications are not in the entity reference itself, discarding changes by reconstructing the entity reference will not work, and changes are visible across references [13].

Data Provider



Data Providers in commerce system manage the loading and saving of DTOs, offering per-DTO configuration. Sequential Data Providers, commonly employed in commerce systems, consist of an ordered list of providers, often including local cache, distributed cache, and blob storage. Reads in Sequential Data Providers iterate through each provider, returning the first result (typically from local cache), updating previous providers if needed. Writes cascade through providers until completion or encountering an error.



In a standard 3-provider setup, local cache optimizes fast access, with distributed cache handling off-machine calls and triggering local cache invalidation upon saves. For example, if machine A writes to the distributed cache, machine B is notified to invalidate its local cache, ensuring the next read on machine B fetches the updated data from the distributed cache.

Endpoints Management

Once we have the de-normalized data after Augmentation referred to as intermediate model, we can transfer the data to different systems in a specific format that is compatible with those downstream applications. We propose a custom solution to abstract many internal details and business specific logic from the endpoint management using the proposed design below

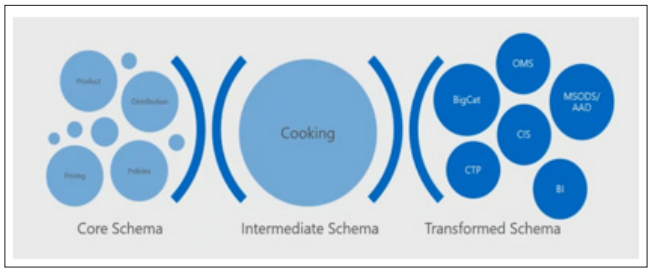
Environment: Environment maintains the list of endpoints that the system needs to export/publish the data to and the owner for this environment so that each separate business can have its own environment configuration and publish data.

Publish Endpoint: The endpoint contains all the properties and behaviors to transform and transmit data to downstream systems and save the state of data to be ingested. Basic code template for the Publish endpoint would look something like below

class IPublishEndpoint:
def PublishEntities(self, entities, catalog)
def ResetStateAsync(self)
def ValidateEntities(self, entities, catalog)
def IsPublishable(self, catalog)

Above is a basic template that can be used to build on to validate, transform and publish the entities as well as maintain state of the publish or ingestion.

Deep Dive
Model Transformation

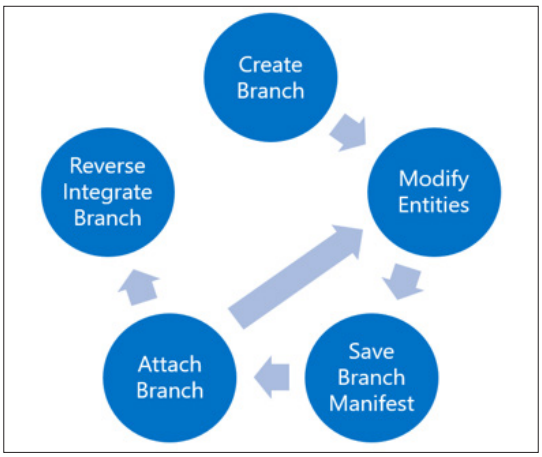


Partner/Business Planner Model: This is persistent, and the source of truth for partner intent and expressed as scenario-focused intent fragments.

Core Model: This is also persistent and well-factored across all scenarios. This is a well-defined normalized schema and will be generic to serve various businesses and services. This core model contains all required schema for building a commerce application.

Intermediate/Augmented Model: After applying the configurations and processing various permutations, we arrive at this model that can be processed and converted into various formats supported by different applications. Also, this data can be fed to Business Intelligence and Machine Learning pipelines to extract valuable insights from trends and identify any anomalies with payment information or tax codes.

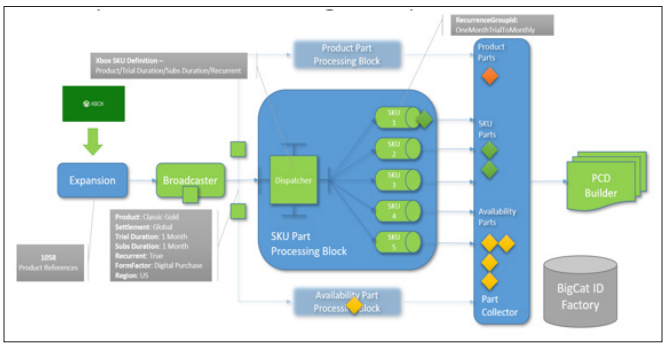
Data Versioning/Branching



The above figure demonstrates the lifecycle of data branching/versioning. This acts like source-control for the data changes in the commerce system. Any changes to the data or modifications to the entity are stored under the scope of a data branch. Maintaining revisions of data is helpful for consistency, audit, and traceability. Data branching and versioning can be achieved by creating pointers/references to the data using a key and stored on the blob. So, any modifications on the data are applied to a data branch version and those changes are written to a specific version of the reference in blob storage [9]. Once the changes and review process are complete, the data gets committed to the main branch and is published to the downstream systems.

Cooking/Augmentation

In the commerce system, data is stored efficiently to represent business intent, avoiding duplication, and ensuring correctness. Augmentation is a process where product configurations, specifying choices and dependencies, generate all permutations based on the options and dimensions.



For instance, a shirt may have configurations for color and size, defining options and dependencies. Augmentation systematically explores the configurations, allowing the creation of various product instances [5-6].

Key Classes and Methods

Product: represents the entry point to Augmentation, storing configurations.

Model Reference: built product with metadata.

Product Reference Builder: executes behaviors with satisfied dependencies.

Behavior: defines dimension option groups, dependencies, and provides data components.

Model Dimension Option Group: defines options for a property.

Dimension: an instance of an option.

Model Reference.Get All Permutations Parallel: performs parallelized build and permutation steps.

High-level Steps of Augmentation

Discover all behaviors on the product.

Sort behaviors are based on the configuration and dependency information.

Build: run behaviors with satisfied dependencies, providing option groups.

Permutate: select options for groups without a selection, repeating the process.

Detailed Steps of Augmentation

Create the initial reference, discovering and sorting behaviors and create a builder, passing an empty product reference.

Use the builder to build a new product reference with behaviors.

Build: Run the Apply method of each behavior with satisfied dependencies.

Dependency Graph: Enumerate products, create a dependency graph, and sort behaviors.

Options for Add Requires: Define requirements for dependency graph behavior placement.

Augmentation optimizes the representation of business data, allowing scalable and flexible product configurations in the commerce system.

Lessons Learned

While designing and building these large-scale distributed applications there are different things that come into play which become more complex than the actual business logic implementation itself. Providing the same level of experience for the customers at an exponential level of growth involves thinking about the underlying infrastructure and horizontal scaling of resources to serve the needs.

Latency: The system should be able to scale horizontally to maintain similar latency over the years for exponential growth in data and complexity in business needs. So, designing the system to be able to parallelize and run across multiple nodes and be able to get consistent results will be important [1].

Ease of Debugging: In traditional applications, iterative development is facilitated by storing data and rerunning queries as needed. However, in this normalized stream processing system, where data is maintained in different formats, iterating becomes challenging, and updating operators may yield different results on new data streams. So, maintaining different versions of data and be able to use that to test the outputs that the transformation pipeline generates helps a lot with debugging for any scenarios.

Ease of Deployment and Rollback: Ease of deployment and rollback is crucial in distributed applications. The feature development and serialization processes need to consider forward and backward compatibility to be able to achieve frequent deployments and rollbacks.

Ease of Monitoring and Operations: Monitoring and operation of deployed apps are vital. We use alerts to detect processing lag, ensuring timely adjustments. Creating dashboards and monitoring key metrics to check the overall health and functionality of the system.

Stream Processing vs Batch Processing: The choice between streaming and batch processing is not binary. A hybrid approach combines streaming and batch processing for optimal efficiency. Streaming-only systems can provide authoritative results without compromising accuracy and helps in dynamic scalability by better utilization of resources with low latency [5].

Conclusion

In recent years, large-scale distributed commerce applications with real-time processing have seen widespread adoption, with the development of multiple independent yet composable systems. These systems collectively form a versatile platform catering to diverse needs. This paper explored various design decisions and their impact on ease of use, performance, fault tolerance, scalability, and correctness [1].

Firstly, a crucial design decision prioritized targeting seconds of latency over milliseconds. This choice, deemed sufficient for supported use cases, allows the utilization of persistent storage for data transport. This transport mechanism facilitates fault tolerance, scalability, and multiple correctness options in data processing and conversions.

Secondly, emphasizing ease of use is paramount. Ensuring systems have manageable learning curves that enable rapid development and testing. Simplifying debugging, deployment, and operational monitoring significantly boosts system adoption rates, a trend we aim to enhance further.

Thirdly, recognizing a spectrum of correctness needs, our approach offers choices along this spectrum. Application builders can opt for ACID semantics if required, accepting additional latency and hardware costs. However, for many use cases measuring relative proportions or directional changes rather than absolute values, faster and simpler applications are enabled. Also, this paper introduced and deep dived into some advanced concepts that can be utilized while building a distributed commerce system and designed those components for greater flexibility.

References

1. Kleppmann M (2017) Designing Data-Intensive Applications. O'Reilly Media https://www.oreilly.com/library/ view/ designing-data-intensive-applications/9781491903063/.
2. (2020) Commerce Best Businesses. Forbes https://www.forbes.com/sites/quickerbettertech/2020/03/01/ these-are-the-top-business-apps-for-2020and-other-smallbusiness-tech-news/?sh=47a7730c41dc.
3. (2022) Forbes e-commerce transforming global trade. Forbes https://www.forbes.com/sites/danikenson/2022/06/13/ thee-commerce-revolution-is-transforming-global-trade-andbenefitting-the-us-economy/?sh=66e547fd22fa.
4. Marius K (2023) Largest e-commerce companies. Markinblog https:// www.markinblog.com/largest-ecommerce-companies/.
5. Janet LW, Shridhar I, Anshul J, Ran L, Guoqiang JC (2016) Realtime Data Processing at Facebook. SIGMOD 1087-1098.
6. Low latency system design. Kayzen https://kayzen.io/blog/ largescale-low-latency-system-design.
7. Yang M (2022) Designing A High Concurrency, Low Latency System Architecture. Medium https://medium.com/@ markyangjw/ designing-a-high-concurrency-low-latency-systemarchitecture-part-1-f5f3a5f32e36.
8. Ajeet Khurana (2019) Defining the Different Types of E-Commerce Businesses. Liveabout https://www.liveabout.com/ ecommerce-businesses-understanding-types-1141595.
9. Allie Decker (2023) What Is a SKU? Shopify https://www.shopify. com/retail/what-is-a-sku-number.
10. Commerce Business Model. Shopify https://www.shopify.com/blog/ business-model.
11. Nick Handel (2021) Denormalization and Cooking: Metrics are recipes to make your data useful Cooking data (normalization/ denormalization). Medium https://towardsdatascience.com/ denormalization-andcooking-metrics-are-recipes-to-make-your-data-useful100a6933b47e.
12. DTO (Data Transfer Object). Baeldung https://www.baeldung.com/ java-dto-pattern.
13. Transaction Isolation Levels. Microsoft https://learn.microsoft.com/ en-us/sql/t-sql/language-elements/transaction-isolationlevels?view=sql-server-ver16.
14. Anna Baluch (2023) Forbes e-commerce statistics. Forbes https:// www.forbes.com/advisor/business/ecommerce-statistics/. 15. What is a DTO? https://stackoverflow.com/ questions/1051182/ what-is-a-data-transfer-object-dto.