Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 1, Issue 2

Open Access | PP: 42-46

DOI: https://doi.org/10.70315/uloap.ulete.2024.0102007

Methodology for Building Scalable Microservice Architectures on Go for High-Load E-Commerce Platforms

Ruslan Tsyganok

Head of Development Team, ecom.tech.

Abstract

The article describes a methodology for building scalable microservice architectures in the Go language for high-load e-commerce platforms. The basic principles of decomposition by "bounded contexts," the choice of communication protocols (REST vs. gRPC), as well as the use of stability patterns (Circuit Breaker, Bulkhead, Retry) and built-in Go capabilities (context. Context, error wrapping). The infrastructure of containerization (Docker), orchestration (Kubernetes), CI/CD pipelines, and Blue/Green & Canary deployment processes is described. The following scaling strategies have been investigated: horizontal/vertical, automatic (HPA, Cluster Autoscaler), load balancing, asynchronous processing via Kafka/RabbitMQ, caching (Redis) and profiling (pprof). Prototypes and load testing (k6, JMeter) confirmed a close to linear dependence of throughput on the number of replicas. The proposed methodology ensures reliability, fault tolerance, and an economical auto-scaling model with increasing user and transactional loads. The information described in this article will be in demand by leading researchers and architectural engineers of distributed systems engaged in the methodological justification and practical verification of scalable microservice solutions in the Go language for highly loaded e-commerce environments.

Keywords: *Microservices, Go, E-Commerce, Scalability, Devops, Kubernetes, Performance.*

INTRODUCTION

With the rapid expansion of e-commerce, the volume of user requests and transactions continues to grow exponentially, placing increased demands on system reliability, performance, and scalability. Traditional monolithic architectures no longer meet the requirements for peak-load handling and fast iterations, leading to substantial financial losses and a degraded user experience.

The **objective** of this article is to examine the methodology for designing scalable microservice architectures using the Go programming language, ensuring system reliability, performance, and flexibility under peak e-commerce workloads.

The **scientific novelty** of this work lies, first, in the introduction of a unified integration framework combining Go's concurrency model, service mesh (e.g., Istio), event bus (Kafka/NATS), and CI/CD automation to achieve near-linear scalability of microservices. Second, it formalizes the criteria for choosing communication protocols (gRPC vs. REST) and caching strategies (Redis, Memcached) based on cloud infrastructure cost-efficiency.

The **author's hypothesis** is that the use of the CSP paradigm in Go, combined with containerization (Docker), orchestration (Kubernetes), and an event bus, can ensure near-linear scalability and reduced response time under increasing load.

MATERIALS AND METHODS

In the scientific literature on distributed systems, microservices have firmly established themselves as a key pattern for building flexible and scalable applications. In the first group of works, J. Lewis and M. Fowler [4] provide a classical definition of a microservice as an autonomously deployable component that manages its own state and interacts with the rest of the system through lightweight inter-process interfaces. H. Peter [3] compares the monolithic and microservice models in the context of AI-integrated applications. The author emphasizes that microservices allow for better scaling of computational resources to meet machine learning demands, while also introducing additional complexity in managing interservice communication and data consistency.

The second group of studies focuses on ensuring the reliability and scalability of large-scale systems. C. Lattner et al. [1] emphasize that high availability in distributed systems requires both load balancing and service mesh integration. G. Nookala [2] analyzes architectural patterns for managing microservice environments. I.C. Donca, O.P. Stan, L. Miclea [10] propose a clustering model for microservices based on dynamic resource allocation and metric-driven autoscaling.

A key role in ensuring rapid delivery and release stabilization is played by CI/CD practices and orchestration tools, as reflected in the third group of studies. M. Di Carlo et al. [5] describe the experience of implementing CI/CD in the SKA astronomy

project. The present work emphasizes containerization, automated testing, and canary deployment. R. Wang et al. [6] propose a deployment approach for an archiving service on Kubernetes for the CAFe scientific equipment project using operators and Helm charts, which helps minimize manual intervention. In turn, S. K. Mondal et al. [8] explore the use of Kubernetes and serverless computing in IT administration, highlighting the issues of "cold start" latency and challenges related to stateful services.

Migration strategies are addressed by Y. Lee and Y. Liu [7], who describe a step-by-step transition from REST applictions to gRPC services. An automated step-by-step process for transforming contracts is described, along with a comparison of performance and overhead between the binary protocol and JSON-over-HTTP.

Special attention should be given to research in the area of security, particularly the study by S. Park et al. [9], which proposes a mashine learning - based DDoS detection system for 5G core networks. Despite the specialized context, the methodology for building distributed detection modules is applicable to microservice ecosystems in e-commerce, where request volumes and the threat landscape are similar.

Thus, the analysis of related studies reveals a wide range of approaches—from foundational definitions and architectural comparisons to specific CI/CD practices and clustering.

Moving to the theoretical discussion of the topic, it should be noted that, in accordance with the concept of Domain-Driven Design (DDD), each microservice must correspond to a clearly defined "bounded context"—for instance, order management, product catalog, payment, or user management. This kind of separation allows one to:

- Simplify the development process, as teams work on autonomous services without touching shared code;
- Improve fault tolerance, since a failure in one service does not lead to the failure of the entire system [3].

Microservice decomposition is based on the concept of bounded contexts. According to DDD, each service is responsible for its own business domain—order management, catalog, payments, users, etc. This simplifies team workflows (each team is responsible for its own service) and increases the fault tolerance of the system [1,3]. A comparison of two main approaches to interservice communication in Go is presented in Table 1.

Table 1. Comparison of HTTP/REST and rpc for Go microservices [3, 5, 6].

Indicator	HTTP/REST	gRPC
Protocol	HTTP/1.1	HTTP/2
Data format	JSON (text-based, human-readable)	Protocol Buffers (binary, compact)
Performance	Medium	High (low latency)
Typing	Absent	Strong (defined via .proto schemas)
Streaming	Not supported	Supported (bidirectional streaming)
Tooling	Any HTTP client	protoc-gen-go and Go standard libraries

To protect against partial failures and "dirty" peak loads, classical design patterns are applied:

- Circuit Breaker (Tink for Go)
- Bulkhead (resource isolation)
- Retry with exponential backoff

The following is an example of implementing retries with exponential delay [4].

```
func Retry(ctx context.Context, operation func() error) error {
  var err error
  for i := 0; i < 5; i++ {
    if err = operation(); err == nil {
      return nil
    }
    select {
    case <-time.After(time.Duration(math.Pow(2, float64(i))) * time.Second):
    case <-ctx.Done():
      return ctx.Err()
    }
  }
  return err
}
```

Following the described principles of responsibility separation, use of high-performance protocols, and built-in Go resilience mechanisms, it is possible to build a microservice architecture capable of scaling reliably and nearly linearly under increasing e-commerce platform load.

Go applications are compiled into statically linked binaries, which enables the creation of minimal Docker images [3]. Below is an example of a multi-stage Dockerfile for a Go service:

```
# Build stage
FROM golang:1.19-alpine AS builder
WORKDIR /app
COPY go.mod ./
COPY go.sum ./
RUN go mod download
COPY *.go ./
RUN CGO_ENABLED=0 GOOS=linux go build -o service .

# Execution stage
FROM scratch
COPY --from=builder /app/service /service
EXPOSE 8080
ENTRYPOINT ["/service"]
```

CI/CD pipelines based on GitLab CI/CD or GitHub Actions include build, testing, code quality analysis, image creation and publishing, as well as deployment via kubectl or Helm. To mitigate risks when releasing new versions, Blue/Green and Canary deployment strategies are used.

To ensure modularity and clear separation of concerns, the domain is decomposed into bounded contexts. Based on the analysis of business rules and data flows, bounded contexts are identified, each containing an independent data model and API. This approach minimizes external dependencies and facilitates parallel development of services.

When designing inter-service communication within the cluster, preference is given to the gRPC protocol due to its low latency, strong typing, and built-in streaming mechanism. At the same time, HTTP/REST continues to be used for external public APIs, as it simplifies integration with client applications and third-party systems [7].

To improve service fault tolerance, Circuit Breaker and Bulkhead patterns are implemented using ready-made libraries (e.g., Tink). Retry and timeout policies are standardized via the context package. A unified strategy for error wrapping and tracing is based on the use of the fmt. Errorf("...: %w", err) construct, which ensures logging and simplifies diagnostics.

Containerization is performed using multi-stage builds to minimize image size, and orchestration is handled through Kubernetes (Deployment, StatefulSet, HPA, Cluster Autoscaler). All configurations are tailored to meet the requirements for fault-tolerant deployment and horizontal scaling [8].

CI/CD processes are fully automated from commit to production, including build, test, and deployment stages. To reduce the risk of failures when releasing new versions, Blue/Green and Canary deployment strategies are implemented using Argo Rollouts or Istio VirtualService.

Metrics collection and visualization are organized using Prometheus and Grafana. Go runtime indicators and application-level business metrics are exposed, with alerts and dashboards configured accordingly. Centralized logging is implemented via the EFK stack or using Loki + Fluentd/Promtail. Distributed tracing is enabled through the OpenTelemetry SDK, allowing for end-to-end request analysis [9].

Adaptive scaling is performed using HPA and Cluster Autoscaler with thresholds based on CPU, memory, and custom application metrics. At the network stack level, L4/L7 load balancing is configured with support for canary rules and failover. Peak loads are mitigated using event-driven queues, and hot data is cached in Redis and processed asynchronously [10]. Regular profiling with pprof and load testing using k6/JMeter, in combination with Prometheus metric correlation, allows timely identification of bottlenecks and performance optimization [2].

The comprehensive application of the described methods forms a unified methodology for designing and operating scalable microservice architectures in Go, capable of delivering high performance, fault tolerance, and flexibility under increasing load on e-commerce platforms.

RESULTS AND DISCUSSION

Effective organization of computing resources requires consideration of both horizontal and vertical scaling, as well as automation of these processes, client request balancing, asynchronous processing, data caching, and detailed code profiling.

Horizontal scaling implies adding new microservice instances, which in Kubernetes is implemented by increasing the number of Pods via the Horizontal Pod Autoscaler based on CPU, memory, or custom metrics. This approach ensures virtually unlimited system growth and high fault tolerance through load distribution; however, it complicates state synchronization between services and increases the volume of network communication. Vertical scaling—i.e., increasing CPU and RAM resources on a single node—remains the preferred choice when load parallelization is limited.

Autoscaling automation combines the capabilities of Horizontal Pod Autoscaler and Cluster Autoscaler. The former dynamically adjusts the number of Pod replicas based on CPU, memory, or custom metrics, which can be integrated, for example, via the Prometheus Adapter. The latter adds or removes nodes in the cloud cluster in response to resource shortages or surplus. This combination allows for timely response to workload changes while conserving resources during idle periods. However, it requires configuring threshold values to avoid instance count fluctuations and delays when launching new virtual machines [3]. Table 2 provides a comparison of existing autoscaling mechanisms in Kubernetes.

Table 2. Comparison of autoscaling mechanisms in Kubernetes [3].

Mechanism	What is scaled	Trigger metric	Advantages	Disadvantages
HPA	Pod replicas	CPU, memory, custom metrics	Fast response to load	Threshold fluctuation, potential "jitter"
Cluster Autoscaler	Cluster nodes	Pod resource unavailability	Cost savings on idle nodes	Delay in launching new VMs (~2–5 min)

Incoming traffic distribution among microservice instances is handled via L4 (TCP) and L7 (HTTP/gRPC) load balancers. L4 balancing, based on round-robin or least-connections methods, provides simple and fast routing at the transport protocol level, while L7 approaches allow routing based on URL paths, headers, and support for canary deployments via Ingress controllers or service mesh solutions like Istio. Proper configuration of load balancing significantly improves system resilience and reduces average response time.

To offload synchronous requests and increase throughput, it is advisable to use asynchronous processing via message brokers (RabbitMQ, Kafka). In this model, the frontend service publishes events, and background systems process them independently of arrival time—this "smooths out" peak loads and increases service resilience. However, it also introduces the need to address eventual consistency and debug distributed operation scenarios.

Caching the results of frequently repeated queries in an in-memory cache (Redis, Memcached) reduces database load and minimizes data retrieval latency. Common strategies—such as read-through, write-behind, TTL (time-to-live), and LRU eviction policies—require balancing data freshness with memory allocation and implementing tools for monitoring and cleaning up stale entries.

Load testing and profiling of Go-executed modules help identify bottlenecks and optimize hot code paths. The built-in pprof package allows CPU and heap profiling, and load testing tools (k6, JMeter) simulate both short bursts and prolonged peak loads. Combined monitoring in Prometheus and visualization in Grafana enable real-time adjustment of autoscaling thresholds and service resource profiles.

The synergy of horizontal and vertical scaling, automatic resource expansion, traffic balancing, asynchronous architecture, deliberate caching, and regular profiling ensures near-linear performance growth of Go-based microservices even under extreme loads.

The analysis of the methodology demonstrates that the combination of Go and modern DevOps practices (containerization, CI/CD, monitoring) ensures high fault tolerance and rapid time-to-market for new features. Distributed API-driven communication via gRPC and Kafka enables horizontal scaling while maintaining predictable performance and low latency. In the future, it is advisable to expand the methodology through automatic analysis of business performance metrics (A/B testing, ROI analysis) and the implementation of GitOps practices for full infrastructure-as-code management, allowing technical KPIs to be directly linked to financial and product goals of the company.

CONCLUSION

The paper presents a methodology for designing and implementing scalable microservice systems in Go, combining architectural practices, infrastructure provisioning, and optimization strategies. The approach is based on decomposing the system into bounded contexts and selecting communication protocols that ensure both performance and type safety: gRPC is used for internal service interactions, while REST is reserved for external APIs, providing a balance between response speed and transmission reliability.

The infrastructure component of the methodology includes the creation of lightweight multi-stage Docker images and the use of Kubernetes controllers (Deployment, Horizontal Pod Autoscaler, Cluster Autoscaler, StatefulSet), enabling effective cluster state management and automated application scaling. CI/CD pipelines with integrated Blue/Green and Canary deployment strategies allow updates to be delivered without downtime and new service versions to be tested directly in production environments.

To ensure high system performance and resilience, scaling strategies were developed: combining horizontal and vertical scaling, autoscaling and load balancing, along with asynchronous message processing through Kafka and RabbitMQ, and distributed caching using Redis. System profiling via pprof helps identify bottlenecks and supports near-linear throughput growth as the number of replicas increases. The described development and operational patterns are easily adaptable to various domains, enabling the timely delivery of new features while maintaining service maintainability.

At the same time, the study identifies certain limitations: the economic aspects of cloud resource usage under conditions of extreme autoscaling were not analyzed, nor was the impact of network latencies in multi-regional deployments on overall performance. Further development of the methodology may involve the introduction of AI-based load forecasting systems, the expansion of service mesh capabilities (Istio, Linkerd) for traffic management, and the implementation of seamless data store migrations with zero downtime. The integration of Go's capabilities, advanced DevOps practices, and proven design patterns enables the creation of high-performance, fault-tolerant, and cost-effective microservice architectures that optimally meet the requirements of modern e-commerce applications.

REFERENCES

1. Lattner C. et al. MLIR: Scaling compiler infrastructure for domain specific computation //2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). – IEEE, 2021. – pp. 2-14.
2. Nookala G. Microservices and Data Architecture: Aligning Scalability with Data Flow //International Journal of Digital Innovation. – 2023. – Vol. 4 (1). – pp. 1-9.
3. Peter H. Exploring Cloud-Native Modular Architectures for AI-Driven Sales Pipeline Optimization: Opportunities and Challenges. – 2023. – pp. 1-14.
4. Lewis J., Fowler M. Microservices: a definition of this new architectural term //MartinFowler. com. – 2014. – Vol. 25 (14-26). – pp. 12.
5. Di Carlo M. et al. Ci-cd practices at SKA //Software and Cyberinfrastructure for Astronomy VII. – SPIE. - 2022. – Vol. 12189. – pp. 28-41.
6. Wang R. et al. A new deployment method of the archiver application with Kubernetes for the CAFe facility // Radiation Detection Technology and Methods. – 2022. – Vol. 6 (4). – pp. 508-518.
7. Lee Y., Liu Y. Using refactoring to migrate REST applications to gRPC //Proceedings of the 2022 ACM Southeast Conference. – 2022. – pp. 219-223.
8. Mondal S. K. et al. Kubernetes in IT administration and serverless computing: An empirical study and research challenges //The Journal of Supercomputing. – 2022. – pp. 1-51.
9. Park S. et al. Machine learning based signaling ddos detection system for 5g stand alone core network // Applied Sciences. – 2022. – Vol. 12 (23). – pp. 1-9.
10. I.C. Donca, O.P. Stan, P., Miclea L. Proposed model for a Microservices Cluster //2020 21th International Carpathian Control Conference (ICCC). – IEEE. - 2020. – pp. 1-5.

Citation: Ruslan Tsyganok, "Methodology for Building Scalable Microservice Architectures on Go for High-Load E-Commerce Platforms", Universal Library of Engineering Technology, 2024; 1(2): 42-46. DOI: https://doi.org/10.70315/uloap.ulete.2024.0102007.