The American Journal of
Engineering and Technology

ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 230-236
DOI 10.37547/tajet/Volume07Issue03-20

Check for updates

Problems and Solutions in Building Highly Loaded Software

Volodymyr Kozub

Senior Software Engineer at Korn Ferry Houston, Texas, USA

OPEN ACCESS

SUBMITED 26 January 2025
ACCEPTED 27 February 2025
PUBLISHED 27 March 2025
VOLUME Vol.07 Issue03 2025

CITATION

Volodymyr Kozub. (2025). Problems and Solutions in Building Highly Loaded Software. The American Journal of Engineering and Technology, 230–236. https://doi.org/10.37547/tajet/Volume07Issue03-20

Abstract: High-load software systems are pivotal in today’s digital landscape, where organizations must handle ever-growing user volumes, data transactions, and real-time interactions. This article explores the core challenges and corresponding solutions in designing, deploying, and maintaining high-load software applications. Emphasis is placed on architectural scalability through microservices, optimal database management (including sharding and replication), and effective use of caching and load balancing techniques. In addition, the study outlines asynchronous processing methods that enhance system responsiveness by offloading resource-intensive tasks to background queues. A dedicated focus is also given to monitoring, logging, and fault tolerance approaches, showcasing how a combination of redundancy, automated failover, and chaos testing procedures can ensure uninterrupted service delivery. The conclusions are drawn from both the author’s previously published concepts and recent academic insights. By integrating these proven practices—from containerized deployment to distributed tracing—software engineers can more effectively address performance bottlenecks, guarantee high availability, and support real-time scalability. The primary contribution of this article is a consolidated framework, illustrating how modern load handling strategies and robust monitoring pipelines can optimize throughput, lower latency, and reduce operational risks in high-load environments. The solutions proposed are adaptable to diverse technology stacks, with special attention to NoSQL options, microservices orchestration, and automated testing/verification protocols. This consolidated perspective underscores the necessity of proactive design choices, continuous testing, and rigorous observability practices to achieve resilient, scalable software systems in the face of volatile market demands.

Keywords: high-load software, microservices, sharding, caching, load balancing, fault tolerance, asynchronous processing.

Introduction:

The evolution of the modern digital society is accompanied by a rapid increase in processed data volumes, a growing number of users and transactions, and the rising complexity of business logic [1]. High-load software has become an integral component of online services, e-commerce systems, social platforms, and financial applications, wherein even a brief outage or malfunction can lead to considerable financial losses and reputational harm.

Intensifying market competition imposes stricter requirements on scalability and availability: it is no longer sufficient merely to "add more" server resources—it is necessary to implement a comprehensive strategy encompassing microservice architecture, distributed databases, and fault-tolerance mechanisms [2]. At the same time, the dynamic nature of load changes and the proliferation of cloud technologies reveal new bottlenecks with critical effects on performance and reliability [3]. A more recent review [4] likewise underscores the importance of an integrated approach to balancing, testing, and monitoring in distributed environments.

In this article, we build on the results presented in [5], which featured architectural patterns and methodological recommendations for designing systems that can endure sudden load spikes without incurring critical degradation in service quality. However, an open question remains: how to consolidate practical techniques (e.g., sharding, caching, asynchronous queues) within a unified methodology that also accounts for network-layer components (such as SDN-based load balancing) and the proper integration of microservices (including Node.js)?

A portion of existing research focuses on localized issues, such as the transition from a monolith to microservices [1] and splitting databases (both SQL and NoSQL), while another part addresses networking solutions (Software-Defined Networking, intelligent routing) [3, 4]. Nevertheless, many such works lack a fully integrated model that can combine tools at various layers (application–database–network) into a cohesive strategy for high performance and fault tolerance.

The purpose of this study is to systematically analyze the primary challenges and existing solutions for building high-load software (with attention to microservice patterns, database optimization, and networking aspects) and to present the author's approach—extending ideas from the already published monograph [5] —demonstrating the added value of combining these solutions under a single framework.

- We summarize approaches for constructing high-load systems, including microservice architecture, database sharding, and asynchronous load balancing.
- We illustrate how the author's concept (as outlined in [5]) can be applied to real-world projects, ensuring high fault tolerance and the ability to withstand traffic peaks.
- We provide recommendations for integrating the above-mentioned methods into a unified architectural model, allowing for the effective elimination of bottlenecks across different tiers (application, database, and network).

By merging microservice architecture, intelligent balancing mechanisms (potentially involving SDN), and a well-designed data storage scheme (sharding, replication), the system gains a significantly higher capacity for handling sudden surges in load, thereby reducing the risk of performance degradation and maintaining the stability of the overall service.

1. Key challenges in building high-load software

A fundamental consideration when designing high-load software is choosing the architectural style: a monolithic approach or a microservices pattern. In the traditional monolith, all modules and business functions are deployed as a single process [1]. This approach initially simplifies development and deployment, yet when the load grows significantly, major drawbacks quickly emerge:

- Scalability Complexity: In a monolithic setup, it is often difficult to selectively scale an individual module; the entire application must be scaled as one unit, leading to inefficient resource use.
- Tight Coupling: Any change in one module may require recompiling and redeploying the entire application, thereby increasing the risk of regression and system downtime [6].
- Longer Build and Test Cycles: As the functionality expands, so does the codebase, complicating CI/CD pipelines.

By contrast, the microservices style partitions the application into a set of small, autonomous services, each deployed and scaled independently [1, 2]. These services communicate via lightweight protocols (HTTP, gRPC, message queues). Such an approach makes it possible to:

- Scale only the specific services under peak load.
- Accelerate the development and release of updates without impacting the entire product.
- Flexibly choose technology stacks for different services.

However, microservices also present notable challenges: they require sophisticated orchestration (e.g., Kubernetes, Docker Swarm), introduce complexities in unified monitoring/logging, and raise concerns regarding network latency and reliability in service-to-service communication [4].

For clarity, Table 1 compares key characteristics of monolithic vs. microservice architecture.

Table 1. Comparison of monolithic and microservice architecture (based on [1]).

Criterion	Monolithic Architecture	Microservice Architecture
Scalability	Entire application scales as one unit; potentially costly/inefficient	Granular scaling of only the needed services
Development Complexity	Initially simpler; single codebase and repo	Requires well-thought-out DevOps practices; multiple repos
Coupling	High—modules are tightly coupled	Low—each service is autonomous
Deployment & Updates	Any change triggers a rebuild of the entire application	Services are deployed independently; supports Blue-Green and Canary strategies
Resilience	A critical failure in one module can bring down the whole system	Failures are isolated to a single service; additional failover detection is required
Testing	One overall test cycle; high risk of regressions	Each service can be tested individually; end-to-end testing is more complex
Technology Choice	Single stack (language, framework, database)	Each service can opt for the most appropriate stack

Shifting from a legacy monolith to microservices in practice is non-trivial and involves:

- Service Boundaries and Decomposition: Determining appropriate boundaries for each microservice while avoiding over-splitting [7].
- Security and Distributed Transactions: In a monolith, transactions are typically ACID-compliant; however, microservices call for saga patterns or CQRS for transaction management [2].
- Ensuring Consistency: Independent databases for multiple services increase the risk of conflicting writes and data drift.
- Team Readiness: Requires upskilling DevOps teams to work with Kubernetes or equivalent orchestration systems, plus introducing centralized logging and distributed tracing [3].
- Pitfalls in Designing the Distributed Model: Mistakes made at this stage are expensive to correct and might only surface under high load [1].

Within high-load scenarios, the “cost of an error” escalates exponentially for two primary reasons:

- Financial Loss: Downtime in e-commerce services can amount to tens of thousands of dollars per minute.
- Reputational Risk: Frequent failures drive users toward competitors, and regaining trust can be nearly impossible [8].

Likewise, a poorly planned architecture (e.g., a non-sharded monolith confronted by massive traffic growth) can render the entire service unreachable during peak hours [4].

Historically, large-scale systems have relied on heavyweight relational DBMSs (Oracle, PostgreSQL, MySQL). When data volumes grow vertically, several issues arise:

● Difficulty of Horizontal Scaling: Traditional relational models are geared toward “scale up” (beefing up a single node), eventually hitting hardware limits [8].

● Locking and Contention: High concurrency leads to an increased risk of lock contention and delays, which degrade throughput.

● High Cost of Licensing and Hardware: Scaling “upward” can be prohibitively expensive [9].

The popularity of NoSQL solutions (MongoDB, Cassandra, Redis) stems from their original design for horizontal scalability (scale out). A key consideration is the CAP theorem (Consistency, Availability, Partition Tolerance): it is impossible to achieve both strong consistency and high availability simultaneously under network partitions [10]. In high-load software, therefore, eventual consistency is often chosen, enabling asynchronous data synchronization while keeping the system online even if some nodes fail.

● NoSQL Advantages: Distributed architecture (ring topology), easy scale-out by adding servers, high write throughput.

● Drawbacks: Lack of ACID transactions, complexity for analytical queries, extra layers for inter-service consistency [8].

Sharding is a technique for dividing data into logical partitions (shards) potentially residing on different nodes. This is particularly effective for:

● Continuous Growth in Data Volumes: new shards can be added “on the fly,” distributing load [11].

● Geographically Distributed Systems: shards located closer to regional users reduce latency [8].

However, choosing the wrong shard key (e.g., one that is poorly correlated with usage frequency) leads to load imbalances (the “hot shard” problem) [3]. Concurrently, replication (master-slave or master-master) is typically essential for fault tolerance: if one node goes down, another can still handle requests. Many architectures adopt a hybrid strategy, storing some data in a relational DB and some in NoSQL [12].

Large-scale network infrastructures for Internet services often come under peak load from legitimate traffic and from DDoS (Distributed Denial of Service) attacks, in which hundreds of thousands of malicious requests attempt to saturate bandwidth [4]. Under such conditions:

● Any “single point” (authentication server, API gateway) can quickly be overwhelmed.

● Local failure in a load balancer or firewall can cause full service unavailability.

Continuous availability (High Availability) at the network layer is therefore critical, along with a flexible traffic-filtering algorithm (e.g., rate limiting to block IPs that exhibit suspiciously high activity).

In high-load scenarios, a traffic asymmetry often arises: one service might experience a surge in requests, while another remains underutilized. Solutions include:

● Load Balancing: distributing incoming requests across a server pool using round-robin, least connections, or load-based algorithms [2]. At the network layer, SDN technologies can dynamically reconfigure routing [3].

● Asynchronous Queues (RabbitMQ, Apache Kafka): offloading critical services by running tasks in the background. This is especially valuable for operations (e.g., report generation) that do not require an immediate response [8].

● Caching (Redis, Memcached, CDN): storing frequently accessed data in memory or at edge locations, drastically reducing database load [4].

An appropriate combination of these techniques minimizes latency and protects services against temporary overloads.

Finally, monitoring and logging in a distributed microservices architecture—often with dozens of services and numerous servers—becomes another major challenge. In monolithic systems, collecting logs centrally is often sufficient. Yet under high load and service sprawl:

● End-to-End Tracing to follow a request through multiple services is difficult.

● Brief load spikes can generate an overwhelming number of log events, potentially saturating the logging system. Missed logs hamper incident investigations [1].

● Tools such as Prometheus+Grafana for metrics and Elastic Stack (ELK) for text-based logs must themselves be clustered and scaled [3].

In summary, the key issues discussed in this section affect all layers of high-load systems—ranging from the application architecture (monolith or microservices) to database and network interactions, encompassing DDoS protection and load balancing configurations. Subsequent sections will address methods to mitigate these obstacles, demonstrating how the author’s integrated approach (see [5]) effectively tackles these challenges in combination.

2. Existing solutions and optimization approaches

Microservice architecture involves breaking an application into a set of loosely coupled, independent services [1]. The most commonly used patterns include:

● Service Discovery: Registers services in a central registry, simplifying automatic request routing.

● Circuit Breaker: Prevents cascading failures by "breaking the chain" when persistent errors are detected.

● API Gateway: Serves as a single entry point for external requests, routing traffic to specific services while providing authentication functions.

According to [5], these patterns ensure isolation and enable flexible scaling of components experiencing the highest load. It is critical to avoid tight coupling between services by minimizing direct calls and mutual data storage.

Step-by-step recommendations for service decomposition and technology selection:

● Domain modeling. It is recommended to follow the Domain-Driven Design methodology by defining bounded contexts and independent functional areas [5]. Each area becomes a separate microservice with its own data storage.

● Protocol selection. Internal service communication can utilize gRPC or AMQP, whereas public APIs typically remain on HTTP/REST [1].

● Infrastructure foundation. A basic orchestration layer (Docker, Kubernetes) and a configuration server (HashiCorp Vault, Consul) should be in place. This approach ensures that each service is deployed independently and managed through centralized tools [4].

Such decomposition reduces the risk of monolithic bottlenecks as traffic increases and allows for more flexible resource allocation.

Caching server-side data (e.g., storing the results of computationally expensive queries) can reduce the load on the central database and improve response times. Two main approaches are commonly used in practice:

● Redis: Supports data structures (hashes, lists, sets), allows TTL configuration, and enables cluster replication for availability.

● Memcached: Designed for simple key-value storage, providing high-speed access but lacking persistence support [8].

In [5], it is noted that proper use of TTL and regular checks for "expired records" significantly reduce the number of redundant queries to the database. For example, the following Node.js code (using Redis) demonstrates caching query results:

```
const redis = require('redis');
const client = redis.createClient();

function setCache(key, data, ttl = 3600) {
  client.setex(key, ttl, JSON.stringify(data));
}

function getCache(key, callback) {
  client.get(key, (err, reply) => {
    if (err) callback(err, null);
    else if (reply) callback(null,
JSON.parse(reply));
    else callback(null, null);
  });
}
```

Here, **setex** sets the time-to-live (TTL) in seconds, allowing Redis to automatically remove outdated data.

Load balancing across multiple nodes ensures high availability. The most widely used solutions are Nginx and HAProxy, which support Round Robin, Least Connections, and Weighted Round Robin algorithms [1]. Additionally, [5] emphasizes the need to implement:

● Horizontal scaling: Automatic server addition managed by Kubernetes orchestration or AWS Auto Scaling in response to increased traffic [11].

● CDN (Content Delivery Network): Static resources (images, JS/CSS) are distributed across globally deployed nodes (e.g., Cloudflare, CloudFront), reducing latency and offloading the primary backend.

To minimize response time under high loads, long-running operations (such as PDF generation or mass notifications) should be offloaded to background queues (RabbitMQ, Kafka) [4]. In [5], the following advantages are highlighted:

● Asynchronous processing: The API immediately responds with "task received", while actual processing occurs in the background.

● Resilience: If one worker fails, messages remain in the queue, allowing other workers to continue processing.

● Worker scalability: As message volume increases, additional worker processes can be added.

For example, in Node.js, the Bull library can be used:

```
const Queue = require('bull');
const emailQueue = new
Queue('emailQueue');

emailQueue.process(async (job) => {
  // Simulating email sending
  await sendEmail(job.data.email);
});

emailQueue.add({ email:
'user@example.com' });
```

This approach improves load distribution and simplifies the implementation of event-driven mechanisms.

For high-load systems, ensuring automatic failover to backup nodes is critical. This can be implemented as:

● Active-Active: All nodes are available simultaneously, and the load is distributed; in case of failure, traffic is redirected to the remaining nodes.

● Active-Passive: A standby node is activated only when the primary node fails [11].

Previous research [5] highlighted that multi-region deployment further reduces the risk of global downtime. It is essential to test the failover mechanism and periodically simulate failures (Chaos Engineering) to verify the correctness of the recovery plan [8].

Modern monitoring practices involve collecting metrics (CPU usage, response latency, number of active connections) and logs from all system components (microservices, databases, load balancers, etc.). Recommended tools include:

● Prometheus: Stores time-series metrics and allows alert configuration.

● Grafana: Provides real-time metric visualization.

● ELK Stack (Elasticsearch, Logstash, Kibana): Centralized log collection and analysis with convenient search and aggregation capabilities (Smith & Brown, 2021).

For request correlation in microservice architecture, trace IDs are assigned to all service logs [4]. [5] describes how end-to-end tracing simplifies bottleneck identification and anomaly analysis.

Load testing (Apache JMeter, Locust) provides insights into system behavior under extreme conditions [1]. It is recommended to:

● Simulate an increase in users or transactions within a short time frame.

● Analyze response time, throughput, and error rates.

● Implement chaos testing (Chaos Engineering) by randomly shutting down containers to verify the ability of remaining nodes to handle the load [8].

A systematic approach to testing improves reliability and enables early detection of technical issues.

Thus, the solutions described in this section form a comprehensive optimization strategy for high-load systems at all levels—from microservice architecture to fault tolerance mechanisms and testing.

CONCLUSION

High-load software development entails more than simply adding servers or scaling existing infrastructure. Rather, it requires a holistic approach that incorporates architectural design, database optimization, asynchronous task processing, and robust monitoring. Microservices emerge as a central paradigm, enabling independent scaling and minimizing cross-service dependencies. Caching strategies and load balancing significantly improve throughput and reduce mean response times, while background queues handle resource-intensive or time-consuming processes asynchronously.

Equally vital is the implementation of redundancy and failover techniques to mitigate downtime and protect against partial failures, as well as continuous observability through metrics and centralized logging. These strategies, drawn from both established best practices and systematic analyses [5], offer a comprehensive roadmap for practitioners aiming to build or refine high-load environments.

Future research may investigate the potential of machine learning for predictive scaling, explore advanced distributed consensus models beyond traditional CAP-theorem constraints, and delve deeper into security concerns, including automated anomaly detection to counter sophisticated distributed attacks. Incorporating these developments promises to advance the field further, ensuring that high-load software remains flexible, fault-tolerant, and capable of sustaining peak traffic demands under evolving conditions.

REFERENCES

Newman, S. (2019). Monolith to Microservices. O'Reilly.

Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., & Weber, T. (2017). Microservices: How to Make Your Application Scale. In R. Sellami & B. Vogel (Eds.), Microservices: Science and Engineering (pp. 10–30). Springer.

Neghabi, A. A., Jafari Navimipour, N., Hosseinzadeh, M., & Rezaee, A. (2018). Load Balancing Mechanisms in the Software Defined Networks: A Systematic and Comprehensive Review of the Literature. IEEE Access, 6, 14159–14178.

Belgaum, M. R., Musa, S., Alam, M. M., & Su'ud, M. M. (2020). A Systematic Review of Load Balancing Techniques in SDN. IEEE Access, 8, 98612–98630.

Kozub V. Best practices for the development of high-load systems / Volodymyr Kozub - K.: Gutenberg, 2024. - 116 p. ISBN 978-617-95387-2-8

Bass, L., Weber, I., & Zhu, L. (2015). DevOps: A Software Architect's Perspective. Addison-Wesley.

Evans, E. (2004). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly.

Stonebraker, M., & Hong, S. (2018). What To Do About Growing Data Volume and Variety? Communications of the ACM, 61(8), 34–39.

Brewer, E. (2012). CAP Twelve Years Later: How the ''Rules'' Have Changed. Computer, 45(2), 23–29.

Harrison, G., Huang, B., & Zhang, Y. (2019). Improving the Scalability of Sharded Databases with Adaptive Load Balancing. International Journal of Database Management Systems, 11(3), 41–56.

Patel, A., Chauhan, S., & Sen, D. (2019). A Hybrid Approach for Data Storage in Distributed Environments. Journal of Cloud Computing, 8(1), 10–15.