International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Asynchronous processes and message queues in Ruby applications: efficiency analysis of Sidekiq and RabbitMQ

Anna Topalidi

Specialist degree, Moscow state university of geodesy and cartography, Moscow, Russia

Abstract

This article examines the role of asynchronous data processing in the architecture of modern Ruby applications. Particular attention is given to the use of background jobs and message queues as tools for offloading the main execution thread and improving server-side performance. The architecture of asynchronous processes is studied with a focus on the integration of tools such as Sidekiq and RabbitMQ within the Ruby ecosystem. It analyzes the specifics of their use, implementation approaches, and the principles of organizing queues and workers in the context of web applications. It explores the comparative efficiency of Sidekiq and RabbitMQ in handling background tasks, considering their strengths and weaknesses in terms of reliability, scalability, fault tolerance, and ease of operation.

Keywords: Ruby, Asynchronous processing, Background jobs, Message queues, Sidekiq, RabbitMQ, Performance

1. Introduction

Asynchronous data processing plays a pivotal role in ensuring the performance and scalability of modern web applications. Within the Ruby ecosystem, the offloading of resource-intensive operations from the main execution thread is commonly achieved through the use of message queues and background job processors. This architectural approach reduces server load, improves user interface responsiveness, and ensures reliable execution of tasks that do not require immediate completion. In practice, asynchronous workflows are especially valuable when interacting with external APIs, sending notifications, generating reports, and performing other operations involving latency or large data volumes.

Among the widely adopted tools for implementing asynchronous logic in Ruby applications are Sidekiq, which leverages Redis, and RabbitMQ, a fully-featured message broker system. Despite fundamental differences in architecture and intended use, both tools are actively employed for background processing. The choice between them is often guided by considerations of reliability, flexibility, performance, and ease of integration. A clear understanding of these differences is essential for effective application architecture design. The goal of this research is to provide an overview and comparative analysis of Sidekiq and RabbitMQ in the context of implementing asynchronous processes in Ruby-based applications.

2. Main part. Architecture of asynchronous processing in Ruby applications: approaches and tools

Amid increasing demands for scalability and fault tolerance in web applications, developers are increasingly turning to architectural strategies that offload resource-intensive or time-consuming operations from the main execution thread. Asynchronous data processing enables high interface responsiveness while simultaneously serving a large number of clients. Within the Ruby ecosystem, this approach has gained significant traction due to the availability of robust tools and libraries tailored to the language's characteristics and frameworks, particularly Ruby on Rails.

Asynchronous architecture is commonly implemented using message queues and background workers that execute tasks outside the primary HTTP request lifecycle [1]. A typical example involves a user submitting a form, which triggers a background process to send a confirmation email. Performing this action synchronously would delay the server's response and hinder the user experience. By delegating such tasks to a separate thread or process, the server can immediately complete the request while offloading the additional workload to a dedicated handler. Performance differences between these two processing models can be observed in a benchmark study [2], where response times were measured under varying user loads in a .NET environment for illustrative purposes (fig. 1).

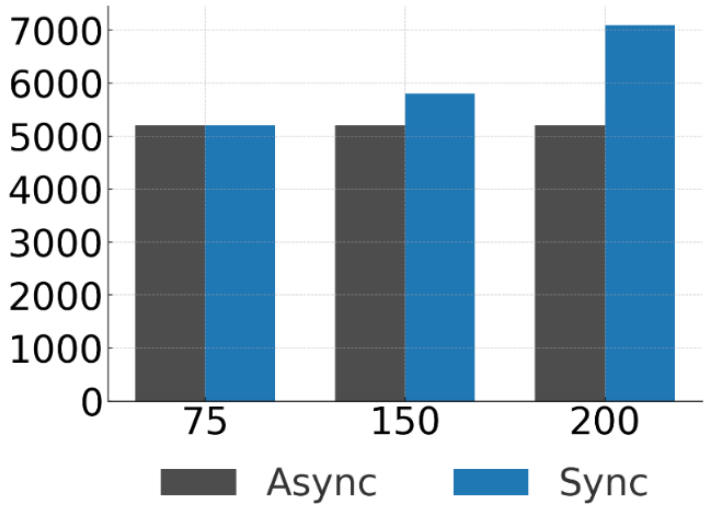

Figure 1: Median response time for synchronous and asynchronous processing under varying user load, ms

In Ruby development, two architectural solutions have received significant recognition for the effectiveness in handling asynchronous tasks: the Redis-queued process handling solution, Sidekiq, and the end-to-end message broker, RabbitMQ, based on the AMQP protocol. While both tools provide for postponed or distributed tasks, there are significant differences in message sending strategy, message sending capabilities, scalability, and monitoring in handling tasks. The choice between them is largely influenced by the architectural requirements of a given application, the presence of external services that must be integrated, and overarching project priorities, ranging from ease of configuration to demands for high availability and reliable message delivery.

Integration of asynchronous tools into Ruby applications is achieved through specialized libraries. Sidekiq is closely integrated with Ruby on Rails, offering a simple syntax and native support for ActiveJob, which makes it an accessible option for developers seeking minimal configuration overhead. In contrast,

RabbitMQ typically requires more explicit setup and is often favored in highly distributed architectures, microservice environments, or scenarios that involve complex message routing. Ruby developers can interact with RabbitMQ through libraries such as Bunny or March Hare, though in comparison to Sidekiq, incorporating RabbitMQ into Rails-based projects generally demands more deliberate architectural planning [3].

It is important to note that asynchronous processing in Ruby is not limited to the use of message queues. The language also provides low-level concurrency mechanisms such as threads and fibers [4]. However, in the context of server-side applications built with frameworks like Rails or Sinatra, external queues and background workers remain the most reliable and widely adopted approach. These solutions enable consistent, repeatable, and scalable task execution independent of the web server, while minimizing the impact of background operations on user-facing workflows.

Thus, the architecture of asynchronous processing in Ruby applications is fundamentally based on the separation of execution flows between the main application and dedicated task handlers, as well as the use of intermediary message delivery systems. Understanding the distinctions between the primary tools available, their capabilities, and implementation specifics is a critical step toward the effective design and maintenance of high-load systems.

3. Comparative analysis of Sidekiq and RabbitMQ in the context of background task optimization

Implementing background job processing requires selecting an appropriate tool that ensures reliable task delivery and execution without overloading the main application. Sidekiq and RabbitMQ are two widely adopted solutions within the Ruby ecosystem. Nevertheless, their architectural foundations, task-handling models, and operational characteristics differ significantly. These differences lead to distinct scenarios for optimal use and varying levels of effectiveness depending on the nature of the tasks being performed.

Sidekiq is designed as a lightweight, tightly integrated background processing system for Ruby on Rails applications, leveraging Redis as its queue storage backend. Its primary strength lies in its high throughput when handling large volumes of homogeneous tasks, all while requiring minimal configuration and maintenance effort [5]. Utilizing an internal multi-threading mechanism, Sidekiq can process numerous jobs concurrently within a single process, making efficient use of system resources. Built-in support for retries, delayed jobs, and a flexible monitoring interface via a web dashboard facilitates rapid adoption, even in projects with modest architectural complexity. However, in distributed environments or in cases where complex message routing is required, Sidekiq may present limitations in terms of flexibility.

RabbitMQ is a fully-featured message broker based on a broker-client model. It supporting various routing patterns including direct, topic, and fanout exchanges. Comparing to Sidekiq, RabbitMQ offers a higher degree of control over message delivery paths, acknowledgement mechanisms, and fault tolerance. This makes it particularly well-suited for microservice architectures, where precise coordination of task flows across multiple components is essential (table 1).

Table 1: Comparison of Sidekiq and RabbitMQ [6, 7]

Characteristic	Sidekiq	RabbitMQ
Processing model	Parallel threads within a process	Message brokering via external broker
Storage / broker	Redis	RabbitMQ (AMQP)

Retry support	Yes (built-in with exponential backoff)	Yes (configurable policies)
Message delivery reliability	Limited (depends on Redis durability)	High (acknowledgements, durable queues)
Routing flexibility	Limited	Flexible (direct, topic, fanout, headers)
Delayed job support	Yes (via set + wait)	Yes (native, with TTL and delay plugins)
Monitoring and UI	Built-in web dashboard	Via external tools (e.g., Management Plugin)
Rails integration	Full (integrated with ActiveJob)	Possible, but requires manual setup

The adoption of RabbitMQ typically involves more complex configuration and operational overhead, including queue management, exchange setup, and retry policies, which may be excessive for smaller or less complex projects. The performance of these two solutions also depends heavily on the nature of the tasks being executed. Sidekiq tends to deliver superior results when processing short-lived, high-frequency jobs such as sending emails, updating database records, or interacting with external API. However, it is less equipped to ensure message durability in the event of system-wide failures, as Redis does not provide persistent storage guarantees without additional configuration. RabbitMQ, by contrast, is designed to support reliable message delivery through acknowledgement protocols and the option for long-term queue persistence. This capability significantly reduces the risk of data loss, particularly in critical business workflows where each task must be processed at least once.

Scalability is also achieved differently in the two systems. Sidekiq scales vertically by increasing the number of threads within a process and horizontally by running multiple worker instances. However, the use of Redis introduces a potential single point of failure, particularly in the absence of a properly configured clustered environment. RabbitMQ, by contrast, enables scalability through broker clustering and the addition of independent consumers. This approach provides greater flexibility in distributed systems but comes at the cost of increased infrastructural complexity and operational expertise.

Reliability and fault tolerance likewise vary between the two tools. Sidekiq includes built-in retry mechanisms with exponential backoff for failed jobs, yet the persistence of task data depends entirely on Redis. Without a durable write configuration, Redis may lose data in the event of a failure [8]. RabbitMQ, on the other hand, offers a more robust reliability model, it supports message acknowledgements, durable queues, and recovery mechanisms that preserve state across restarts. These features make RabbitMQ quite suitable for systems with strict requirements for data integrity and guaranteed delivery.

In summary, Sidekiq and RabbitMQ are architecturally and functionally geared for different objectives. Sidekiq shines in traditional, monolithic Ruby applications where the integration and timely execution of scheduled tasks are the top priority. On the contrary, RabbitMQ would suit distributed systems where guaranteed delivery, routability, and independently scalable parts are the need of the hour. Hence, the selection of one of the two should solely depend on careful analysis of project-necessitated demands for dependability, performance, architectural flexibility, and operational practicability.

4. Conclusion

Handling asynchronous tasks is one of the central pillars in the development of contemporary web applications, enabling efficient distribution of the primary execution thread and responsiveness optimization for user interactions. There are quite a variety of tools for asynchronous process handling in the Ruby programming environment, with the two most widely used being Sidekiq and RabbitMQ.

Although both tools are designed for asynchronous support, the inherently disparate design implemented for message handling leads to diverse applications and pragmatic effects.

Sidekiq is designed particularly for ease of integration and throughput in monolithic applications, while RabbitMQ offers advanced route management and scalability features-specifically, features of particular focus in distributed system design. Therefore, the ultimate decision for the use of one tool or the other should depend on the architectural setup of the application, the reliability requirement for message sending, the business logic, and the budget for infrastructure upkeep.

References

1. Sidorov D., Kuznetcov I., Dudak A. “Asynchronous programming for improving web application performance”, ISJ Theoretical & Applied Science, 2024, 138 (10), 197–201.
2. Async vs Sync Benchmark (NET) / Medium // URL: https://medium.com/azlamps/async-vs-sync-benchmark-net-f1e752a57755 (date of application: 12.09.2025).
3. Johansson L., Dossot D. “RabbitMQ Essentials: Build distributed and scalable applications with message queuing using RabbitMQ”, Packt Publishing Ltd, 2020.
4. Ulanov A. “Utilizing Real Time Technologies in Ruby Web Applications”, Internafional Journal of Computer Science and Mobile Compufing, 2022, 11 (12), 34–45.
5. Bakker I.E. “Update to the Statistical Output of a Cybersecurity Monitor”, Fermi National Accelerator Laboratory (FNAL), Batavia, IL (United States), 2023.
6. Sangeetha E., Deny J. “Innovative Actuator Control in Smart Cities with the InterSCity Platform”, In2023 7th International Conference on Electronics, Communication and Aerospace Technology (ICECA), 2023, 342–350.
7. Ćatović A., Buzađija N., Lemes S. “Microservice development using RabbitMQ message broker”, Science, engineering and technology, 2022, 2 (1), 30–7.
8. Sreekanti V., Wu C., Chhatrapati S., Gonzalez J.E., Hellerstein J.M., Faleiro J.M. “A fault-tolerance shim for serverless computing”, InProceedings of the Fifteenth European Conference on Computer Systems, 2020, 1–15.