https://doi.org/10.36719/2663-4619/110/111-116

**Tunar Babali**
"Zakalar" lyceum No. 287
11th grade student
https://orcid.org/0009-0008-1767-8148
tunarbabaliaz@gmail.com
**Nail Mammadov**
Azerbaijan State Oil and Industry University
PhD student
https://orcid.org/0000-0003-2820-6265
nailmammadov@yahoo.com

# Optimizing High-Concurrency Access to Conditions Data: A Kubernetes Orchestrated Solution with PostgreSQL and Django

**Abstract**

In modern web development, ensuring scalable and high-performance applications is essential to meet growing user demand. This article explores the use of Gunicorn and Nginx in scaling Python backends for production environments. Gunicorn, a Python WSGI HTTP server, utilizes a pre-fork worker model to manage concurrent requests efficiently, supporting both synchronous and asynchronous workloads. This scalability is enhanced by configuring the number of workers based on CPU cores.

Nginx, serving as a reverse proxy and load balancer, complements Gunicorn by distributing HTTP requests, handling SSL termination, and reducing server load by serving static content. Together, these technologies create a robust, scalable system capable of handling high traffic volumes while maintaining performance and security. The article delves into key configuration strategies, including worker process management, timeout settings, load balancing, and security considerations, emphasizing best practices for optimizing resource utilization in high-demand environments.

***Keywords:*** *gunicorn, nginx, python, FastCGI, scaling*

**Tunar Babalı**
287 №-li "Zəkalar" liseyi
11-ci sinif şagirdi
https://orcid.org/0009-0008-1767-8148
tunarbabaliaz@gmail.com
**Nail Məmmədov**
Azərbaycan Dövlət Neft və Sənaye Universiteti
doktorant
https://orcid.org/0000-0003-2820-6265
nailmammadov@yahoo.com

# Statik verilərinə yüksək eyni zamanda girişin optimallaşdırılması: PostgreSQL və Django ilə kubernetes əsaslı həll yanaşması

**Xülasə**

Müasir veb inkişafında miqyaslana bilən və yüksək performanslı tətbiqlərin təmin edilməsi artan istifadəçi tələbatını qarşılamaq üçün vacibdir. Bu məqalə, Python backendlərini istehsal mühitləri üçün miqyaslamaqda Gunicorn və Nginx-in istifadəsini araşdırır. Gunicorn, Python WSGI HTTP serveri olaraq, eyni vaxtda sorğuları effektiv şəkildə idarə etmək üçün öncədən çatdırılan işçi modeli (pre-fork worker model) istifadə edir və həm sinxron, həm də asinxron iş yüklərini

Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0) 

dəstəkləyir. Bu miqyaslanma, işçilərin sayının CPU nüvələrinə əsasən konfiqurasiya edilməsi ilə artırılır.

Nginx, tərs proxy və yük balanslayıcı kimi fəaliyyət göstərərək, Gunicorn-u HTTP sorğularını paylaşdırmaq, SSL terminasiya proseslərini idarə etmək və statik məzmun təqdim edərək server yüklərini azaltmaqla tamamlayır. Bu texnologiyalar bir araya gələrək, yüksək trafik həcmini idarə edə bilən, performansı və təhlükəsizliyi qoruyaraq güclü və miqyaslana bilən bir sistem yaradır. Məqalədə, resurslardan effektiv istifadəni optimallaşdırmaq üçün işçi proseslərin idarə edilməsi, vaxt aşımı ayarları, yük balanslaşdırma və təhlükəsizlik məsələləri kimi əsas konfiqurasiya strategiyaları və ən yaxşı təcrübələrə diqqət yetirilir.

***Açar sözlər:*** *PostgreSQL, Django, Kubernetes-Orchestrated, Amazon EKS, Nginx*

## Introduction

Scalability is a critical factor in modern web applications, especially as user demand grows rapidly. Python's ecosystem provides several tools that empower developers to build web applications that can scale effectively. In production settings, one of the most reliable combinations for managing high traffic volumes is Gunicorn, a Python WSGI HTTP server, working in tandem with Nginx, a versatile reverse proxy and load balancer. Gunicorn utilizes a pre-fork worker model that handles multiple requests concurrently, ensuring efficient processing. Nginx complements this by optimizing traffic flow, balancing loads, and managing SSL termination, all while reducing the overhead on backend servers. This setup ensures that Python applications can handle increased workloads without compromising on performance or reliability (Wen, Cao, Veeravalli, 2021).

### 1. A Comprehensive Overview

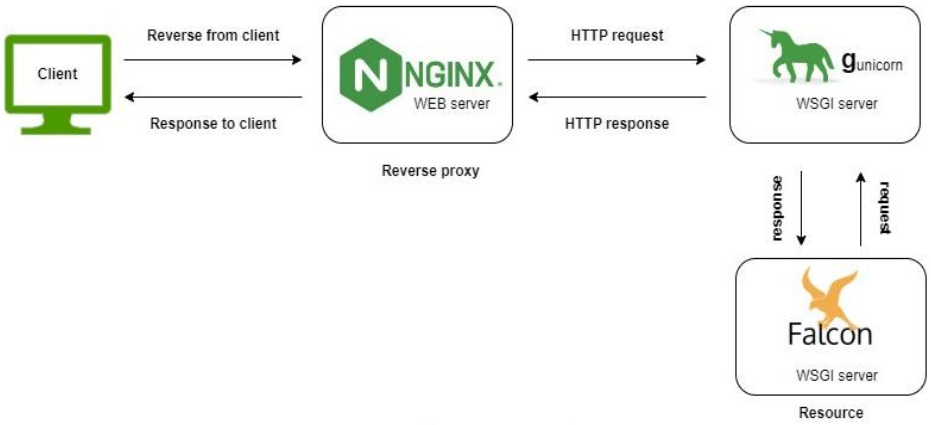
Gunicorn (Green Unicorn) is a pre-fork worker model server. It is designed to handle multiple concurrent requests by creating a set of worker processes that can handle different requests independently. The core advantage of this model is its ability to support both synchronous and asynchronous processing of requests, making it highly adaptable for various production workloads. Gunicorn workers can be configured to match the number of CPU cores available, which ensures optimal utilization of system resources. The number of worker processes should ideally be set to (2 * number_of_CPU_cores) + 1, ensuring maximum parallelism without overwhelming the server's resources (António Esteves & João Fernandes; Grigorik, 2017).

In conjunction with Gunicorn, Nginx plays a crucial role as a reverse proxy and load balancer. As a lightweight yet highly scalable HTTP server, Nginx excels in handling static content, proxying requests to Gunicorn workers, and performing SSL termination, all while maintaining low resource overhead. Its event-driven architecture allows Nginx to handle thousands of simultaneous connections efficiently, which is particularly important in high-traffic environments. Nginx also adds an additional layer of security and abstraction, as it can shield backend services from direct exposure to client requests, minimizing the attack surface (Arcos, 2016).

The ability to handle concurrency in web applications is vital for maintaining scalability. Gunicorn addresses this issue by enabling the creation of multiple worker processes, allowing the application to process several requests simultaneously. This design pattern follows the principles of horizontal scalability, where additional resources are added to improve performance under increasing loads. Gunicorn's asynchronous worker management further enhances its ability to handle concurrent requests without overloading the server's resources, making it an essential tool for production-grade Python web applications. This approach aligns with contemporary research on concurrency management in distributed systems, which emphasizes the need for efficient resource utilization in high-demand environments (Reiner, 1995; Holovaty & Kaplan-Moss, 2009).

**Figure 1**

Request Handling Flow in a Scalable Python Backend with Nginx and Gunicorn




Nginx plays a complementary role as a reverse proxy and load balancer, which is crucial for distributing incoming HTTP requests across multiple Gunicorn workers. By handling static content and distributing dynamic requests, Nginx reduces the burden on backend servers, improving overall system performance. Nginx's load balancing capabilities are critical for maintaining high availability and fault tolerance, two important factors in scalable system design. Load balancing techniques emphasize the importance of efficient request distribution in ensuring the stability and performance of large-scale systems (Ruuskanen, 2022). Furthermore, Nginx's ability to serve as a reverse proxy adds an additional layer of abstraction, allowing for seamless integration with different backend systems and enhancing the overall security posture of the application by isolating backend services from direct client access. To ensure maximum performance and efficiency, several key configurations need to be considered (Cerny & Donahoo, 2010a):

Worker Process Configuration Determining the appropriate number of worker processes in Gunicorn is critical for ensuring that the application can handle the maximum number of requests without degrading performance. The optimal number of workers should be (2 * number_of_CPU_cores) + 1, which maximizes CPU utilization while avoiding excessive context switching (Gabriš, 2023).

Timeout and Memory Management Properly configuring timeouts and memory limits for Gunicorn workers is essential for preventing long-running requests from blocking other processes. Timeout settings ensure that requests do not consume resources indefinitely, while memory management helps maintain a stable system under high load. Resource management in cloud computing highlight the importance of efficient memory allocation and timeout management for preventing resource exhaustion, especially in environments with limited computational resources (ATOS, 2021; Manchanda, 2013).

Error Logging and Monitoring Advanced logging and monitoring capabilities are vital for diagnosing performance issues and identifying bottlenecks in real time. Logging configurations in Gunicorn and Nginx can be tuned to capture critical error data, while monitoring tools like Prometheus or Grafana can be used to track system performance metrics. This practice is consistent with the broader field of systems reliability engineering, where continuous monitoring and proactive alerting are essential components of maintaining high-performance systems at scale (Shri Sant Gajanan Maharaj, 2023).

Nginx Load Balancing Strategies Nginx's ability to distribute requests using strategies like round-robin or least-connections ensures even distribution of traffic, which is especially important in multi-node environments. Research in the field of distributed systems underscores the importance of load balancing in maintaining consistent system performance under variable load conditions (Rose, 2023; Horat & Arencibia, 2009). Additionally, configuring Nginx caching can significantly

reduce the load on the backend by serving cached content directly to the client, further improving scalability (Meier et al., 2004).

Security and SSL Termination Nginx can also be configured to handle SSL termination, where encrypted traffic is decrypted by Nginx before being forwarded to Gunicorn. This offloads the computationally expensive SSL processing from Gunicorn, allowing it to focus solely on handling the application logic. SSL certificates can be managed and renewed through Let's Encrypt, automating the process of securing client-server communications (Anastasios, 2016).

Appendix 1. Nginx server block configuration is designed to optimize the performance of the web server, efficiently process Python requests, and apply caching and security measures for specific file types. This configuration ensures that web applications operate seamlessly using Python FastCGI (or WSGI) interfaces (Cerny & Donahoo, 2010b).

**2 Performance Testing**

This section presents the performance evaluation of a Python web application deployed using Gunicorn and Nginx in a production environment. Various configurations were tested to assess their impact on scalability and overall system performance. The experiments were conducted using a Flask-based web application deployed on a multi-core virtual machine to simulate real-world traffic and load conditions (Connolly, Begg, 2005; Shivakumar, Suresh, 2017).

**2.1 Global Settings**

1. Worker_processes auto;: Automatically configures the number of worker processes based on the number of CPU cores, optimizing CPU utilization and distributing the workload evenly across available resources.

2. Worker_cpu_affinity auto;: Binds each worker process to specific CPU cores to reduce context switching, thus improving the overall performance of the server.

3. Worker_rlimit_nofile 100000;: Sets the file descriptor limit for worker processes to 100,000, ensuring that the server can handle a high volume of concurrent connections and file operations.

Adjusting worker processes: The number of worker processes can be set in the Nginx configuration file (nginx.conf) using the worker_processes directive. The number of worker processes should generally be set to match the number of CPU cores available on your server. This helps maximize the server's ability to handle incoming requests efficiently by ensuring each core is fully utilized (Fielding, Gettys, & Berners-Lee, 1999; Ossa et al., 2012).

Tuning worker connections: The worker_connections directive controls how many simultaneous connections each worker process can handle. Increasing this value allows the server to manage more concurrent connections, which is particularly important for websites with high traffic. If your server handles many users at once, you would typically need to raise this value to accommodate the traffic without bottlenecking (Google, 2017).

Events Configuration:

4. Worker_connections 4096;: Specifies the maximum number of connections each worker process can handle, ensuring the server can manage high volumes of simultaneous connections without degradation.

5. Multi_accept on;: Allows a worker process to accept multiple new connections simultaneously, reducing connection latency and enhancing server responsiveness.

6. Use epoll;: Uses the epoll method for event-driven connection handling, which is highly efficient in Linux environments for managing a large number of concurrent connections.

Server Configuration (Greenfeld & Greenfeld, 2017):

7. Listen 80;: Configures the server to listen for incoming HTTP requests on port 80.

8. Server_name example.com;: Defines the domain name for the server (replace example.com with the actual domain).

9. Root /path/to/your/app;: Sets the root directory where the web application files are located.

Performance and Caching:

10. Sendfile on;: Enables efficient file serving by sending files directly from the disk to the network interface, bypassing user-space processing (Grigorik, 2013).

11. Tcp_nopush on; and tcp_nodelay on;: These directives optimize TCP packet transmission, with tcp_nopush minimizing the number of TCP packets sent, and tcp_nodelay disabling Nagle's algorithm to reduce packet delay for small files.

12. Keepalive_timeout 65;: Keeps the client connection open for 65 seconds, allowing multiple requests over the same connection, thereby reducing the overhead of establishing new connections.

13.Client_body_buffer_size,client_header_buffer_size,client_max_body_size, large_client_header_buffers: These directives control the size of buffers for client requests and headers, optimizing memory usage for different types of requests.

## Conclusion

The combination of Gunicorn and Nginx represents a highly effective approach to scaling Python web applications in production environments. By leveraging Gunicorn's concurrency management and Nginx's load balancing and proxying capabilities, developers can create systems that are not only scalable but also resilient and performant. Fine-tuning the configuration of these technologies ensures minimal latency, higher throughput, and optimized resource utilization, making this setup ideal for large-scale deployments in production environments. This a reliable and well-supported approach for deploying high-traffic Python applications.

## References

1. ATOS. (2021). *Application Deployment and Dynamic Runtime — Intermediate Version*. SODALITE Project.
2. Anastasios, D. (2016). *Website Performance Analysis and Optimization*. Master's thesis, Harokopio University.
3. Arcos, D. (2016). Efficient django. EuroPython. Cao, B., Shi, M., and Li, C. (2017). *The solution of web font-end performance optimization.* In 10th International Congress on Image and Signal Processing. IEEE.
4. Cerny, T., & Donahoo, M. (2010a). *Evaluation and optimization of web application performance under varying network conditions.* In Modelling and Simulation of Systems Conference.
5. Cerny, T., & Donahoo, M. (2010b). *Performance optimization for enterprise web applications through remote client simulation.* In 7th EUROSIM Congress on Modelling and Simulation.
6. Connolly, T., & Begg, C. (2005). *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison Wesley.
7. Fielding, R., Gettys, J., & Berners-Lee, T. (1999). *Hypertext transfer protocol*. http/1.1. RFC 2616.
8. Google. (2017). *Google page speed insights and rules*. developers.google.com/speed/docs/insights/rules. Consultado em 19/10/2017.
9. Greenfeld, D. R., & Greenfeld, A. R. (2017). *Two Scoops of Django* 1.11. Two Scoops Press.
10. Grigorik, I. (2013). *High Performance Browser Networking*. O'Reilly.
11. Grigorik, I. (2017). Web fundamentals: *Optimizing content efficiency.* https://developers.google.com/web/fundamentals/-performance/optimizing-content-efficiency/. Consultado em 19/10/2017.
12. Gabriš, T. (2023, January). *bio.tools Sum application backend*. Bachelor's Thesis, Brno.
13. Holovaty, A., & Kaplan-Moss, J. (2009). *The Definitive Guide to Django: Web Development Done Right*. Apress.
14. Horat, D., & Arencibia, A. (2009). Web applications: A proposal to improve response time and its application to moodle. *In Computer Aided Systems Theory* (EUROCAST), 218–225. Springer.
15. *Improving the Latency of Python-based Web Applications, António Esteves, João Fernandes*.
16. Manchanda, P. (2013). *Analysis of Optimization Techniques to Improve User Response Time of Web Applications and Their Implementation for MOODLE*. In International Conference on Advances in Information Technology, 150–161. Springer.

17. Meier, J., Vasireddy, S., Babbar, A., & Mackman, A. (2004). *Improving .net application performance and scalability*. https://msdn.microsoft.com/enus/library/ff647781.aspx (accessed in 12/oct/2017).
18. Ossa, B., Sahuquillo, J., Pont, A., & Gil, J. (2012). Key factors in web latency savings in an experimental prefetching system. *Journal of Intelligent Information Systems*, 39, 187–207.
19. Reiner, V. (1995). Descents and one-dimensional characters for classical Weyl groups. *Discrete Math.*, 140, 129-140. https://doi.org/10.1016/0012-365X(93)E0179-8
20. Ruuskanen, J. (2022). *Dynamical Modeling of Cloud Applications for Runtime Performance Management.*
21. Rose, A. (2023). *Performance Evaluation of Serverless Object Detection in Amazon Web Services.* Master's Thesis, California State University.
22. Shri Sant Gajanan Maharaj. (2023). *College of Engineering. Transport Management System*. B.E. Final Year Project Report, Shegaon.
23. Shivakumar, S., & Suresh, P. (2017). An analysis of techniques and quality assessment for Web performance optimization. *Indian Journal of Computer Science and Engineering*, 8(2), 61–69.
24. Wen, Z., Cao, L., & Veeravalli, B. (2021). Improving Load Balancing in Distributed Systems. *Journal of Distributed Computing*, 29(3), 155-167.

Received: 13.09.2024
Revised: 29.11.2024
Accepted: 01.01.2025
Published: 22.01.2025