International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2024: 7.101

Design of an Asynchronous Architecture Based on Django and WebSocket for Interactive Applications

Arley Iván Solis Zacapantzi[1], Juan Ramos Ramos[2], José Juan Hernández Mora[3], Blanca Estela Islas Flores[4]

[1, 2, 3, 4] Instituto Tecnológico de Apizaco, División de Estudios de Posgrado e Investigación,

[1, 2, 3, 4] Av. Instituto Tecnológico S/N, Conurbado Tzompantepec, C.P. 90300, Ciudad de Apizaco, Tlaxcala, México
[1]Email: *m23370008[at]apizaco.tecnm.mx*
[2] Corresponding Author Email: *juan.rr[at]apizaco.tecnm.mx*
[3]Email: *juan.hm[at]apizaco.tecnm.mx*
[4]Email: *m23370004[at]apizaco.tecnm.mx*

Abstract: ***This work proposes an asynchronous architecture based on the Django framework to enable real-time bidirectional communication using WebSockets and the ASGI standard. Unlike the traditional synchronous WSGI model, the proposed design supports multiple simultaneous connections, enabling interactive web applications such as messaging, monitoring, and notifications. It details the structural design, Django Channels configuration, and inter-process communication, demonstrating notable improvements in responsiveness, concurrency, and scalability—positioning Django as a strong option for modern real-time services.***

Keywords: ASGI, asynchronous architecture, Django, interactive applications, real-time communication, WebSockets

1. Introduction

The demand for interactive web experiences and real-time two-way communication has steadily increased over the past decade, driven by use cases such as messaging, monitoring, dashboards, and instant notifications [1]. In this context, synchronous models based on the request–response paradigm (e.g., WSGI) present limitations when managing user-dependent interactions that require low latency and high concurrency, often leading to polling strategies using AJAX or fetch, which result in excessive requests and performance degradation under peak load conditions [2].

Various platforms have addressed real-time communication through WebSockets and event-driven architectures, with ecosystems such as Node.js/Socket.io, Phoenix/Elixir, and ASP.NET Core SignalR standing out, as well as newer Python frameworks with native asynchronous support [3]. On the frontend, libraries like React or Vue have facilitated reactive patterns that integrate with low-latency backends [4]. However, Django—historically synchronous—remains preferred for its robustness, security, and extensive Python ecosystem (AI, IoT, vision, billing, scientific libraries), in addition to its clear and extensible MVT architecture that reduces integration complexity [5].

Although approaches exist to "simulate" asynchrony (e.g., short/long polling) or offload non-interactive tasks to Celery, these solutions do not fully address real-time two-way communication with concurrency and efficiency guarantees at scale [6]. The identified gap lies in the absence of an integrated and replicable architecture that transforms Django into an asynchronous core for interactive applications without abandoning its ecosystem and operational advantages.

This work proposes the *Design of an Asynchronous Architecture Based on Django and WebSockets for Interactive Applications*, supported by the ASGI standard, the Daphne server, Django Channels as the WebSocket protocol layer, and a Redis message channel for fan-out, load balancing, and decoupling of producers and consumers. The proposal eliminates polling dependency, enables true concurrency, and promotes an event-driven model while preserving Django's multifunctionality as a full-stack framework, pure backend, or "glue" across multiple technologies (AI/ML, IoT, billing services, and security modules) within a single framework.

2. State of the Art

The transition of web development toward real-time interactions has driven the adoption of persistent, bidirectional channels that reduce latency and request overhead. Early attempts generalized polling and long-polling via AJAX/Fetch to emulate immediacy, though scalability degraded under high concurrency [7].

WebSockets later enabled continuous message exchange over a single connection, improving responsiveness and efficiency. A widely explored path involved event-driven backends such as Node.js with Socket.IO, which supports reconnection, multiplexing, and fallbacks—establishing a foundation for collaborative and presence-based applications [8]. Alternatives consolidated efficient concurrency models for large-scale connections, but migration from Python/ Django systems often demands deep rewrites and data reengineering, increasing transition costs.

Within the Python ecosystem, offloading non-interactive work to queues like Celery with Redis or RabbitMQ became common practice [9]. While ideal for deferred jobs and heavy processing, this approach does not enable real-time bidirectional interaction during user sessions, as it focuses on asynchronous task handling rather than persistent messaging.

A key advancement was the introduction of ASGI, which standardized asynchronous execution for HTTP and WebSocket protocols, enabling coroutines and event loops [10]. Since Django 3.x, official ASGI support opened the door to real-time architectures without abandoning Django's ORM, MVT, and security stack [11]. Django Channels extended this model "beyond HTTP" through async consumers, ASGI routing, and WebSocket support, typically deployed with Daphne and a Redis-based message layer for pub/sub and process decoupling [12][13].

Technical literature emphasizes that while this stack achieves low latency and high concurrency, success depends on a well-defined architecture: separation of real-time and batch processes, proper backplane configuration, retry policies, and observability metrics such as active connections and latency [14].

In summary, current solutions fall into three families: polling/SSE for basic needs; event-driven stacks (Node.js/Phoenix/SignalR) for greenfield projects; and ASGI + WebSockets for preserving Django ecosystems with production-grade robustness. Despite existing resources on individual components (Channels, Celery, Redis), few works provide integrated, reproducible architectures for transitioning from WSGI to ASGI while maintaining Django's data and security integrity. The proposed model—Django (ASGI) + Channels + Daphne + Redis, complemented by Celery for offline workloads—addresses this gap, consolidating real-time communication with a structured and maintainable design.

3. Methodology

The research and technological development focus on designing and implementing an asynchronous Django architecture (ASGI, Channels, Daphne, Redis) to enable real-time bidirectional communication. A flexible, agile-based approach is adopted, integrating tools and practices suited to the project's needs. This aligns with the view that technological research not only analyzes phenomena but also creates new realities through innovation and design [15].

Figure 1 presents De la Cruz Casaño's methodological framework, which guides the project through proposal, theoretical foundation, methodology, implementation, and evaluation. Phase 4 incorporates the system's development process, unifying research and technological execution into a single, coherent workflow that connects problem definition, conceptual grounding, and field validation with the incremental design of the asynchronous architecture.

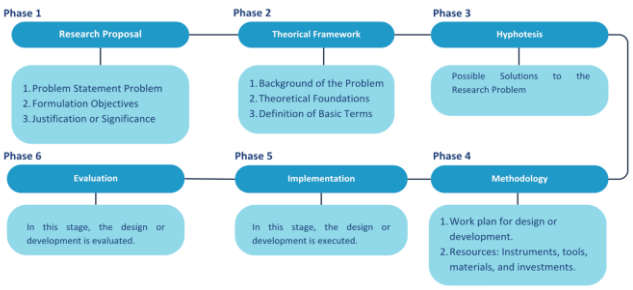

Figure 1: Research Methodology. Author's own work

Figure 2 illustrates the development methodology applied to this project, combining agile practices and tools to build and validate the architecture through real use cases (real-time notifications, monitoring panels, and messaging). Project planning follows sprint-based iterations with Kanban visualization, while quality and delivery rely on XP practices such as automated testing, peer review, and CI/CD pipelines. Security and observability guidelines address per-channel authentication, origin checks, and latency metrics. Phase 4 thus becomes an incremental, traceable cycle where each software increment is validated against acceptance criteria, ensuring the architecture evolves through empirical evidence.

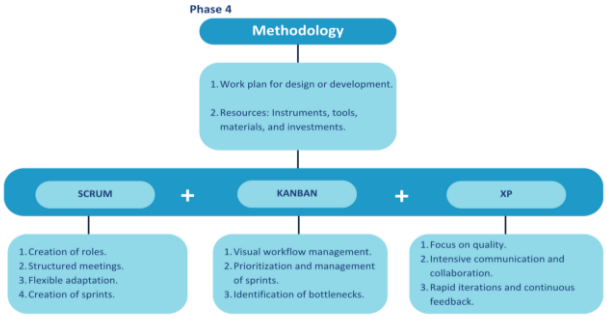

Figure 2: Project Programming Methodology. Author's own work

In summary, the reference technological methodology [19], reinterpreted with an agile and artifact-oriented lens, aligns research with development: the asynchronous architecture is designed, constructed incrementally with verifiable engineering practices, and validated through practical use cases, maintaining alignment between objectives, process, and outcomes.

4. Architecture Development

Figure 3 presents the proposed technological architecture for real-time communication in an asynchronous Django environment. It integrates client, server, and messaging layers under a modular structure that optimizes concurrency and bidirectional data flow.

The web client uses JavaScript and jQuery to update interfaces dynamically and exchange events through WebSockets, enabling low-latency interaction. On the server side, Django—running under the ASGI standard—efficiently handles multiple simultaneous connections through asynchronous processes managed by Daphne and Django Channels.

Redis acts as a message broker using the publish/subscribe pattern to synchronize events and support inter-process communication. Finally, all components are containerized with Docker, ensuring portability, isolation, and consistency across environments.

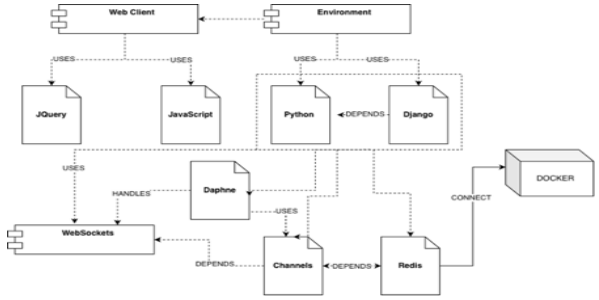

Figure 3: Asynchronous technological architecture based on Django, Channels, Daphne, and Redis

5. Results

The proposed asynchronous architecture enabled the system to evolve from a traditional synchronous server to a fully asynchronous environment, achieving real-time communication without polling or page reloads. Deployed and validated on AWS across five independent projects, it demonstrated stable performance and seamless user interaction.

A key implementation is the School Management System for the Universidad Politécnica Región Poniente, where real-time notifications were integrated into grade and payment modules, allowing instant updates for students and administrators. Another example is the Dental Management Module (Figure 4), where patients and dentists receive immediate notifications upon appointment confirmation, showcasing the efficiency of the Django Channels, Daphne, and Redis–based asynchronous model.

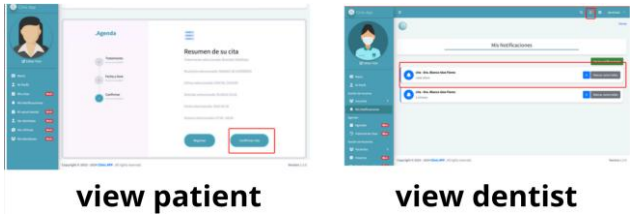

Figure 3: Real-time notification interface in the Clinic App: patient view (left) and dentist view (right)

The deployment on AWS—combined with containerized environments via Docker—ensured scalability, reliability, and consistent performance under concurrent connections. Each implementation verified the architecture's capacity to handle real-time events with low latency, validating the approach as a viable model for interactive, service-oriented web applications.

References

[1] J. D. A. Rincón, J. A. D. Lote, M. R. Galavís, y C. M. S. Loreto, "Sistema de notificaciones en tiempo real por proximidad basado en IoT para promover los servicios de la Vicerrectoría del Medio en la Pontificia Universidad Javeriana," *Pontificia Universidad Javeriana*, 2023.

[2] N. W. T. Mamani, "Modelo de API Gateway para mejorar la comunicación de los microservicios en aplicaciones empresariales," Universidad Nacional del Altiplano de Puno (Perú), 2023.

[3] H. V. P. Singh, S. R. Rizvi, y Q. H. Mahmoud, "Two architectures for real-time sensor data streaming for cloud applications," *International Symposium on Signal Processing and Information Technology*, pp. 133–138, 2015. doi: 10.1109/ISSPIT.2015.7394315.

[4] E. E. Sosa, "Librería de componentes para agilizar el desarrollo inicial de un proyecto," Universidad Nacional de La Plata, 2024.

[5] G. E. Paula, L. Quisaguano Collaguazo, A. C. Guaman, y S. Llambo Alvarez, "Frameworks del lado del servidor: caso de estudio Node JS, Django y Laravel," *593 Digital Publisher CEIT*, vol. 10, no. 1, pp. 403–414, 2025.

[6] K. J. Gracia Orejuela, M. B. Gorozabel Bazurto, y W. Chango, "Long polling, WebSockets y Server-Sent Events: comunicación para el envío de datos en tiempo real," *Mikarimin*, vol. 10, no. 3, pp. 101–120, 2024. doi: 10.61154/mrcm.v10i3.3272.

[7] E. Bozdag, A. Mesbah, A. van Deursen, *"A Comparison of Push and Pull Techniques for Ajax"*, WSE 2007, IEEE, 2007. doi:10.1109/WSE.2007.4380239. Esta compara los enfoques "pull" (polling) frente a "push" (Comet/long-polling) en aplicaciones AJAX y describe los compromisos de escalabilidad.

[8] R. Kannan, M. A. K. T., S. Vairachilai, and V. Ramshankar, "NodeJS and Postman for Serverless Computing," *Advances in Systems Analysis, Software Engineering, and High Performance Computing*, Apr. 2024, doi: 10.4018/979-8-3693-1682-5.ch012.

[9] S. Ganesh, R. George, R. Tejas, N. Badri, and K. Vinodha, "Queue Orchestration Using an In-Memory Broker," in *Proc. 2022 IEEE 2nd Int. Conf. on Mobile Networks and Wireless Communications (ICMNWC)*, Dec. 2022. doi: 10.1109/ICMNWC56175.2022.10031963

[10] M. Lathkar, *High-Performance Web Apps with FastAPI: The Asynchronous Web Framework Based on Modern Python*, Apress, 2023. doi: 10.1007/978-1-4842-9178-8

[11] M. M. Tkhabisimova, U. G. Baymuradov, and A. Sh. Izrailova, "Development and implementation of web applications in Python using the Django framework," *Èkonomika i upravlenie: problemy, rešenîâ*, vol. –, Jan. 2024. doi: 10.36871/ek.up.p.r.2024.12.08.007

[12] P. Soligo and J. S. Ierache, "Informe técnico, telemetría satelital de tiempo real sobre WebSockets y framework Django," *ReDDI: Revista Digital del Departamento de Ingeniería*, vol. 7, no. 2, pp. –, 2023. doi: 10.54789/reddi.7.2.5

[13] J. M. Claudiyap and P. O. N. Saian, "Implementasi sistem broadcast message menggunakan python dan redis pub/sub," *JIPI (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika)*, vol. 7, no. 3, pp. –, Aug. 2022. doi: 10.29100/jipi.v7i3.3014

[14] L. P. Carloni, "The Role of Back-Pressure in Implementing Latency-Insensitive Systems," *Electron. Notes Theor. Comput. Sci.*, vol. –, 2006. doi: 10.1016/J.ENTCS.2005.05.036

[15] C. De La Cruz Casaño, "Metodología tecnológica en ingeniería," *Revista Ingenium*, vol. 1, 2016, pp. 1–.