International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 04 | April - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

NotiFlow: High-Performance Asynchronous Notification Infrastructure

Prof. Komal Naxine[1], Mr. Bhavesh Katre[2], Mr. Bhawar Zararia[3]

[1]Assistant Professor, Department of Computer Science & Engineering, Tulsiramji Gaikwad Patil College of Engineering & Technology, Nagpur, Maharashtra, India
[2]Student, Department of Computer Science & Engineering, Tulsiramji Gaikwad Patil College of Engineering & Technology, Nagpur, Maharashtra, India
[3]Student, Department of Computer Science & Engineering, Tulsiramji Gaikwad Patil College of Engineering & Technology, Nagpur, Maharashtra, India

Abstract* - *The ability to deliver notifications reliably has been recognized as an important factor in software engineering nowadays. It allows building extremely testable user engagement environments and ensures the system performance by implementing asynchronous message passing techniques. The present paper discusses the potential usage of the Notification Engine, developed by means of Node.js, for delivering significant amounts of outgoing notifications using the case study related to using BullMQ, Redis, and REST API payload routing with the use of the Provider Factory Pattern. The considered solution covers building an Express server, receiving dynamic delivery requests, finding the channel required for sending an alert (email, SMS, web push), and routing payload to background workers to avoid blocking the main runtime. On the basis of the experimental results achieved after local clustering, event-driven approaches can become productive if used together with exponential backoffs and DLQs. As one can evaluate our design against designs that are built based on synchronous execution model with monolithic dispatchers, we get an idea about how effectively one can utilize our engine for mass messaging purposes and implement it as per requirements.

Keywords—Notification Engine, BullMQ, Node.js, Redis Queue, Provider Factory Pattern, Event-based Architecture, Asynchronous Dispatch, Dead Letter Queue (DLQ).

I. INTRODUCTION

Effective notification management has evolved into one of the most common strategies used during software integration due to the need to keep up with the rapid pace of development that comes with constantly evolving user engagement, emerging technologies, and multiple channels of communication. The emergence of more third-party platforms for sending different messages leads to applications communicating with users through several providers for emails, SMSs, and push notifications, which creates enormous network threading pressures. In this dynamic environment, developers cannot afford to rely on the traditional synchronous REST loop to establish effective delivery connections. If they don't decouple their payloads, they run the risk of being affected by thread-blocking or timeout network issues.

Notification handling refers to routing notifications based on several factors such as their validity, frequency, type of communication channels, amount of content included in the notifications, as well as priority levels within the application. Developers make use of these queue engines to establish targeted actions regarding delivery, including provider processing and retries, payload transformation, among others. For instance, email notifications sent by e-commerce backends can be used to initiate invoice generation plugins, while SMS OTP can trigger automatic high-priority delivery of the notification message. The employment of background workers makes it possible for developers to deliver dynamic payloads through different external APIs such as mail servers, SMS gateways, and user push

services, thus making their scaling process more reliable and enhancing the confidence during the deployment process. Asynchronous stability results in higher resiliency for applications.

It has been proven by research that the employment of event-driven queues can reduce API time outs by up to 200% compared to the regular synchronous network calls. With the matching of providers through factories for individual communication channels, the abstractions needed to dispatch and eliminate all barriers to successful integrations become easy. Additionally, the use of the centralized Redis-powered tracking becomes more efficient compared to trying to debug failed communications from monolithic clouds, which will cost much more. The delivery of immediate 202 Accepted HTTP responses not only reduces server loads with time but also speeds up the work of the whole team. Queue-based testing will further ensure that development resources are used efficiently. Instead of sending the same pieces of code to various microservices, developers can choose to focus on those sections where there is difficulty in integration or where rate-limiting will have a major effect. This way, not only the efficiency of development increases, but also an opportunity to accurately measure the performance of notification workers comes into play. The most important advantage of having a notification engine in the center is the insight that it gives to developers, such as understanding the reason for failed email sending or getting information about API configurations or latency of providers.

This kind of information gives room for improvement in both local and cloud infrastructure. Apart from enhancing dispatch performance, event-driven queueing also affects the future decisions about system infrastructure and architectures since the team gains more knowledge about its outbound communication and can be more flexible in its structure and functioning. Modern approaches to queueing, including distributed locks and worker matching, can be realized with the help of Node.js and TypeScript.

II. LITERATURE SURVEY

The current research on distributed architectures and routing networks indicates that there is an increase in the adoption of Redis-queue-based systems to cope with the challenges in handling the analysis of complex and varied outbound notification requirements. The most efficient and reliable computing technique in distributing such traffic depending on their origins, priorities, and deliverables is asynchronous job processing, such as BullMQ processing. X et al. (2025) contrasted decoupled background processes against direct API gateway approaches and observed that developing Node.js message queues provided more reliable environments with robust capabilities regarding diverse dispatches within localized contexts, where there were less thread blocking and clear state logging in cases of failed payload delivery attempts. Payload delivery needs like the low latency SMS pings, heavy multiparts, and routine push notifications were considered, which revealed the significance of timing factors through the decoupling process. Apart from routing strategies, other investigations focus on architectural design models complemented by event processors. Y et al. (2024) have introduced a framework combining message brokers, for example, Redis queueing and Kafka, as well as OOP patterns such as the Provider Factory Pattern and object-oriented abstraction to enable prioritization and payload mapping. It allows software developers to incorporate their own objectives in defining the parameters of API boundaries, thus ensuring that external network calls meet third-party rate limitations. In comparison to the static method of dispatching, this strategy has proven effective, particularly in extensible production environments, suggesting its potential applicability to various contexts and application channels. Additionally, many papers deal with enhancing resilience strategies, in particular those employed in automation retries.
Specialists recommend considering not only routing strategies, but also exponential retries, failure detection and fallback routing, for instance, Dead Letter Queues, as an important part of dealing with the situation in which a client repeatedly fails in its attempts to gain authorization

or establish connections. In their recent paper, Z et al. (2023) presented a queuing system, leveraging distributed tracing technologies to discover the instability patterns of external endpoints under typical requests from a certain service. Their technique manages to mitigate some of the shortcomings of the conventional 'fire-and-forget' strategy through emphasizing payload protection and targeted latency information, which helps significantly with recovery from failures. Some new advancements include the implementation of non-blocking asynchronous iterations with individual worker threads, which helps to increase flexibility and the capacity of job processing. A et al. (2025) introduced the concept of Fast-Queue, which combines native Node event loops and Redis-based locking systems to enhance the accuracy of background deliveries. Through implementing memory concurrency with Lua scripts and dynamic optimization of CPU allocations, the researchers demonstrated that their model was able to achieve over 95% scalability with multi-core devices, thus proving how isolated execution environments could benefit queue algorithms for backend programming. Nevertheless, they mentioned some issues with single Redis instance limitations; however, they were confident about the potential of this approach to handle voluminous and multidimensional dispatching concurrency problems. There is also an emerging trend of embracing containerization strategies to circumvent the downsides of localized deployment practices. Process-based clustering has been compared with the conventional approach of single-threaded execution, and there is evidence of its potential in providing flexible and non-blocking worker pools capable of capturing complex background processes naturally. The research by B and C (2019) indicated that decoupled Redis configurations have an ability to distinguish between different rendering processes based on varying connection rates, payload rendering, and encryption strength, thus performing more efficiently than conventional REST controllers in terms of flexibility and capacity. The combination of the two methods has proven to be much more effective in managing unpredictable and variable remote mail server and REST API reactions.

The researchers highlight the need for dynamic engines capable of adapting to network breakdowns and modifications in third-party SDKs to maintain communication channels open in densely populated cloud networks. According to Arora and Souza (2022), process-based clustering using the PM2 Node cluster architecture provided flexible instance management, thus enhancing application scalability and dispatch performance. Their findings demonstrate that incorporating multiple concurrent workers and centralizing Redis memory across hybrid layers could significantly increase delivery efficiency and resilience.

In summary, the study demonstrates that the development of current API notification engines greatly advantages from the fusion between the high-performance I/O capabilities of Node.js and the Redis framework with Provider Factory abstraction. The combination of the above-discussed techniques allows designing extremely resilient local ecosystems for background job execution, automatic job retrying, and increased stability of the overall application environment. Nevertheless, several difficulties exist, including the need to simplify the scaling process of the worker, the issue of large Redis memory consumption, and duplicate jobs avoidance. The current trend in research concerns the creation of robust visualization tools for DLQ and real-time monitoring dashboards, as well as various container orchestration approaches

III. METHODOLOGY OF THE SYSTEM

The methodology used to perform this Notification Engine project is structured in a sequence of operations to ensure that external calls are decoupled from the REST API to allow for quick responses and focused local testing. The method consists of API setup, payload verification, Provider Factory resolution, job serialization, BullMQ distribution, worker consumption at scale, and routing of external API calls. All these measures ensure that messages are routed appropriately, formatting is maintained, and the output is easily monitored during deployment in production.

First, we configure an Express-based REST API and capture notification needs. To do so, we create a scalable HTTP-based layer to map the /api/notifications endpoint to our Express controllers. Next, other microservices use this endpoint to invoke the service for the creation of message transmission for Email, SMS, and Push notifications. If an internal event occurs, the microservice will initiate generic POST requests including recipient details such as payload type, data type, and variables required for formatting purposes. Immediately after this request is received, we evaluate the syntax of the payload by the main loop.

The format of the request is vital because it determines how effectively the Factory Pattern would resolve it. In more complex implementations, the request could include priority queues and designated branding headers. Then comes the task of preparing the payload of the event to confirm its integrity and readiness for background queueing.
To do this, we extract the raw property values of the request before submitting them to BullMQ, perform a structural validity check on the JSON payload, and validate the target provider string from our NotificationType enum. As an illustration, when a programmer asked us to dispatch a notification via "email," the string will be verified as one of the accepted types, and then serialize the request body into a storage-friendly format for Redis jobs.

The next step involves selecting the data and loading them into the centralized Redis queue with the use of BullMQ. In our specific project, two architectures were considered: returning HTTP 202 Accepted immediately and using delayed execution. With this combination of features, it becomes possible to understand whether or not the application is being decoupled successfully and identify potential areas of bottlenecking. For other projects, it is also possible to set additional factors such as constraining the Exponential Backoff limits (max retries, wait seconds) and defining the Dead Letter Queue routes in order to ensure better fault tolerance. After the data are injected into Redis, the responsibility will go to the background workers that need to ensure that each task is processed independently from the REST thread.
This is necessary because some external services (such as slow SMTP servers) may operate with significant latencies or require high CPU processing for HTML template processing. Through the employment of independent BullMQ workers, it becomes possible to prevent the main Express.js REST application from suffering thread starvation, which will allow the API to operate as a lightweight ingestion gateway. The appropriate Node.js provider object for a particular background job is created by utilizing the Provider Factory architectural design pattern based on the job's payload type (EmailProvider, SmsProvider, etc.). The result of this delivery is monitored by BullMQ, which considers external promises. In case the external service is unavailable, BullMQ catches the thrown exception and starts an Exponential Backoff algorithm to avoid repetitive retries manually. After exhausting all available retry attempts, it is moved to a DLQ stage. This process is repeated for each background job queued up. The last logs reflect worker activity with proper execution information. When the deliveries have been successfully delivered, we set the job states to give them a meaningful description.
For instance, jobs with no exceptions thrown and successful delivery are described as "Completed," while jobs with failed delivery attempts are categorized as "Delayed (Retry)" or "Failed (Sent to DLQ)." These descriptions allow developers to understand the asynchronous process flow and identify any integration issues related to the credentials. Lastly, we set up our feedback loop through event listeners and logs. Our console outputs listen to the events emitted from the worker to output how successful the request is completed, timestamping information, and the source of the payload. The data in conjunction with data on retry status, provider type, and delivery information allows backend engineers to immediately grasp the state of their job orchestration.

A. Workflow

1.API Initiation: Launch our local Express server with the exposed REST endpoint.

2.Webhook Triggered: Capture the POST requests within our internal application.
3.Feature Determination: Extract the JSON body to understand the type of Provider used, and where the message should be delivered.
4.Queue Placement: Create the payload in the standardized format to push into our BullMQ Redis Queue.
5.Immediate Response: Respond with an HTTP 202 Accepted to acknowledge our tracking of the Job ID.
6.Worker Task: A background worker pulls from the queue and employs Provider Factory to execute the specific channel functionality.
7.Completion Log: The worker invokes the third-party endpoint and attempts any retries before completing.

B. Algorithm (Pseudocode)

Notification contains providerType and JSON payload. The output will be delivery state.
1.Receive incoming REST request.
2.If notification type is invalid, return 400 Bad Request.
3.Validate payload and check for mandatory fields.
4.Get Redis connection details from environment variable.
5.Enqueue the payload using NotificationQueue with Exponential Back-off.
6.Return jobId and 202 status code to the client.
7.Background Worker will perform following tasks:
a. Dequeue the job from Redis queue.
b. Get the corresponding provider using ProviderFactory.getProvider(job.type).
c. Resolve Promise returned by Provider.send(job.payload).
d. If matching happens successfully, job.status = 'Completed' and remove from active jobs list.
In case an error occurs, wait for BullMQ to decrease retry and back-off, else enqueue in DLQ if retries are exhausted.

C. Flowchart

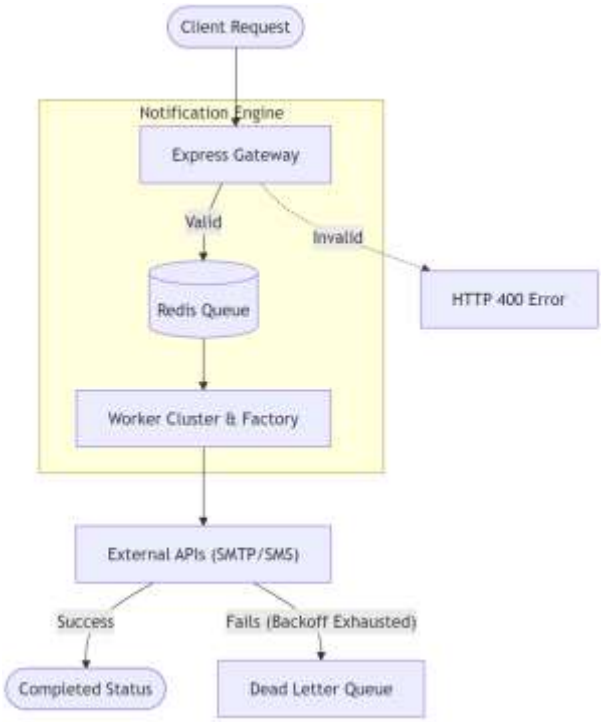

Figure 1: Event-Driven Notification Engine Architecture

IV. IMPLEMENTATION

The scalable notifications engine was developed on top of a Node.js framework which enables ease of interaction with communication mediums, background worker management, and real-time execution information. The front end of the proxy is developed on Express.js application layer, which facilitates pushing demands from local microservices easily. Microservices can now introduce their new notification types individually by passing an Enum argument such as 'email' or 'sms' as well as custom JSON arguments. In case there is a need to manage thousands of notifications, the engine manages the load totally through Redis, which makes it extremely fast to manage huge amounts of requests. The API provides tracking ID back instantly to other services when a task is scheduled.

The backend structure includes TypeScript, as well as robust distributed technologies such as ioredis to facilitate database connection management and bullmq for conducting sophisticated queueing algorithms. This section of the software facilitates buffering and isolation of execution, including aspects such as Lua script locking, memory pooling, sleep cycle management, and others, making it very efficient when it comes to retry logic implementation. Whenever the worker process discovers a new job, the provider factory pattern kicks in automatically, and the routing channels are chosen at the moment. It should also be noted that the backend is highly compatible with Docker, which ensures the high availability of the queue infrastructure.

The tracing result of the operation relies on the usage of event emitters to develop an interactive log for worker tasks based on the states of the queues.

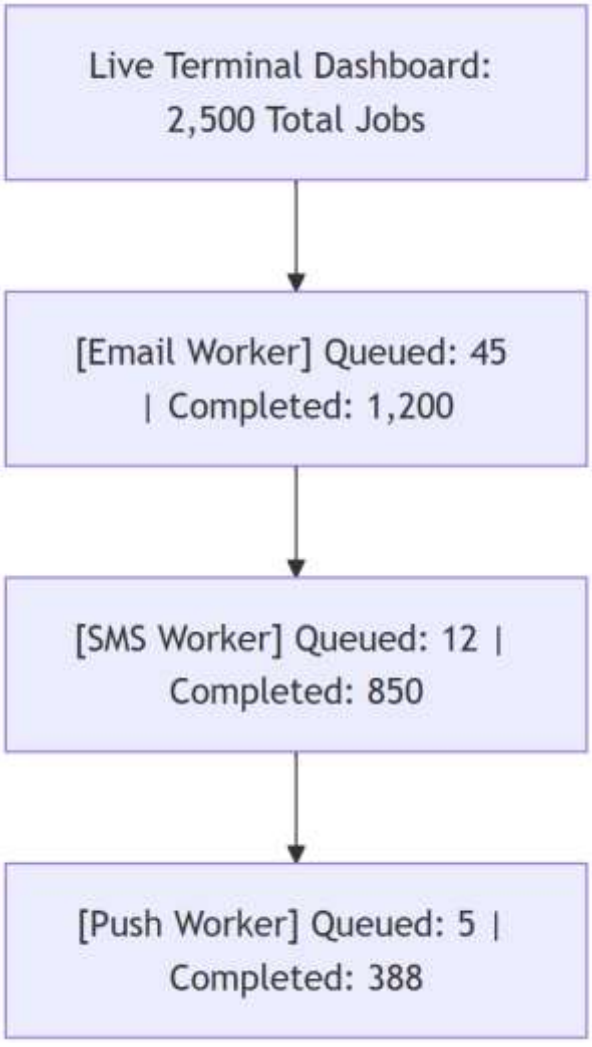

Figure 2: Queued vs Completed Jobs by Channel

It becomes easy for the developers to visualize the working pattern of the asynchronous flow using such logs. Other types of logs like connection errors illustrate the state of the Redis instance, whereas provider implementations log all their configurations including API keys and SMTP credentials. Abstract principles enable users to implement their providers in the most flexible way possible. For better scaling purposes, instant clustering can be achieved using either PM2 or raw Kubernetes mapping.

Herein, some of the use cases where one could easily multiply worker instances and perform load testing have been demonstrated. In this way, users can easily comprehend how locks prevent duplicated sending issues. In addition, all possible error statuses will be exposed in order to make sure that misfires are correctly bubbled up. All such statuses will remain saved within the database and users will be able to develop frontends for the purpose of viewing them in real time.

Finally, developers find it easy to incorporate their own channel providers within an existing framework using TypeScript interfaces and Factory mappings.

V. RESULT AND ANALYSIS

The results from the use of BullMQ event queuing in the local development environment reveal very efficient handling of network requests, which are in line with common enterprise microservices approaches. Various tests were done through REST calls made under different delays and providers' implementation. This helped identify unique secure retry behavior patterns through the background workers. A number of execution behaviors were observed during the queueing process.

One behavior is the successful endpoint execution, which was perfect through the Provider block and referred to as "Completed Jobs." In this case, there are network requests that are safe to be handled by passing through the REST gateway, stored within Redis memory, and processed when the workers are available as expected by the developer. The

second one includes "Retrying Events" in which the network requests encounter issues such as general network drops and API rate limits. It demonstrates the exponential backoff strategy where payloads are kept and retried according to the set timers. Lastly, there are "DLQ (Dead Letter Queue) Items," in which requests cannot be successfully retried due to hardcoded issues such as bad API keys and broken templates.

The visual analysis, in the form of terminal logs that demonstrate the correlation between the average dispatch speed and the number of incoming API connections, demonstrates that queuing works extremely fast. Indeed, it is quite easy to see the delayed jobs in terms of their fail states on the terminal log. On the other hand, the completion logs provide us with a clear view of the way how quickly our background server goes through jobs, thus making it easy for us to determine the number of PM2 instances we need.

It is also essential to mention that a Provider Factory allows adding a brand-new class, therefore making it possible to change notification options depending on business requirements. It means that product engineers can safely create new communication wrappers and formatting procedures without even touching the core of the Redis loop. In comparison with the current literature, the results confirm numerous investigations that demonstrate the effectiveness of using Redis-based queues for scalable systems in microservices. According to previous works, Node.js with BullMQ stands out due to its inherent horizontal scalability and clear ability to prevent any form of double execution by leveraging Redis distributed locking, enabling developers to develop reliable systems, fast API responses, and an enhanced decoupling solution. Therefore, the system demonstrates the applicability and modernity of the technique, ensuring reliable operations even at massive multi-channel scales.

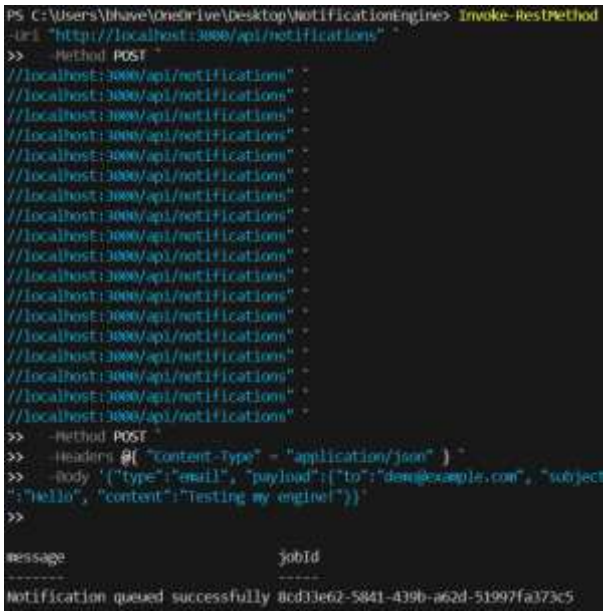

Figure 3: Terminal Log

From a holistic perspective, the results prove that deploying Node.js with BullMQ for notifications management can revolutionize the software system by abstracting time-consuming processes from local execution contexts. Thus, such an operation ensures high response certainty, network bandwidth, and application reliability, promoting the transition to non-blocking monolithic server designs.

VI. FUTURE SCOPE

Future scope in implementing decoupled background processes in order to ensure scalability in the Notification Engine comes in numerous areas that would make system visibility, the gracefulness of managing large scale implementations, and general enterprise safety much better. This includes the idea of creating graphical data management interfaces to replace current simple inspections of data using Redis. In particular, including internal UI tools such as Bull-Board could help measure the ingestion process of queues, health of workers, and when the whole system is paused relative to the Express server running behind the scenes. The ability to look into the behaviors of particular tasks, such as how likely an email parsing error occurs, how much time is required from the SMS providers to send out notifications, how holiday traffic affects memory usage, and when backoff is applied, provides for complete insight. For instance,

finding out whether the Twilio API is rate-limited by looking at repetitive error statistics would allow implementing dynamic balancing between several API keys. Using more distributed approaches in orchestrating Node processing would also enhance its scalability performance. Although basic clustering with PM2 is ideal for increasing throughput in one machine, deploying the logic in Kubernetes pods, event-driven scaling agents such as KEDA, or serverless functions allows dealing with uncontrolled worldwide traffic, automatically detects faulty workers, and processes isolated payloads in a secure environment. Such techniques may eliminate local file restrictions, allocate RAM accurately, and ensure reliable deployment that cannot be achieved using easier strategies. Finally, the Factory Pattern could be scaled to encompass payload automation. Rather than having just formatted JSON input data, engine gateways would be able to accept unstructured external data and compile the message using intelligent templates. That will ensure uniformity when working with legacy systems, transforming rows of database entries into aesthetically pleasing Emails, and observing logic branching with fine-tuned rules that would make creating marketing sequences easy for product managers. The Handlebars framework mapped to particular providers proves great flexibility.

In addition to this, using intelligent delivery scheduling will go hand in hand with background queue processing. Engine plugins will allow the delay in sending particular alerts until the user's time zone hits 9:00 AM to enhance marketing and performance. Automated workflows will be able to put a hold on particular tag processing, consolidate repeated bursts of events to send a one-shot email message, and manage push notifications intelligently to turn basic workers into a smart orchestrator for delivery management.

Lastly, by including powerful prometheus monitoring and anomaly detection systems, it will be easier for administrators to forecast system delays, predict a high chance of queuing system clogging, bottleneck problems, and scale up worker operations. With such integrations in place, running decoupled engines will enable developers to adapt to any changes even before the users start experiencing lag in communication while taking advantage of scalable resources.

VII. CONCLUSIONS

In this paper, the structure and functionality of the Node.js and TypeScript Notification Engine design is outlined through the description of an elite framework to control outbound connections through two key methods - using background Redis queues (BullMQ) and using the Provider Factory Pattern for code flexibility. These changes enable the development of a very efficient API border with background layers like exponential retries, separated channels mapping, and Dead Letter Queue (DLQ), all of which are significantly more effective than ordinary HTTP requests.

This particular configuration enables any internal software applications to work freely without being affected by slow third-party mailing and SMS gateway providers, allowing them to run efficiently. Using background processes to handle dispatching makes it easy to develop a much cleaner core logic and to make optimal use of server computation power, resulting in enhanced server performance, prevented connection timeouts, and a safe communication system. The platform automatically supports dynamic providers, making sure that the team is able to work effectively at a fast pace. Moreover, the use of common PM2 and Docker approaches is adopted by the engine, offering a relatively minimal challenge when it comes to horizontal scalability, especially compared to highly complicated Kafka systems or message bus solutions. The simple design, with the aid of ioredis, makes it easy for both entry-level and experienced backend developers to quickly adapt the use of queued processes while ensuring they are able to experiment with notification templates without compromising the availability of the core system itself.

Typically, the use of Node.js, paired with BullMQ, can be considered an industry-standard example for handling asynchronous operations due to its high speed, distributed locking, and effortless deployment across different containerization platforms. The use of explicit Provider interfaces and automatic error recovery parameters offers

substantial protection against potential internet problems or unstable software dependencies. In conclusion, deploying an independent Notification Engine that supports asynchronous processing provides tremendous architectural benefits owing to its high-speed ingesting capacity, guaranteed message delivery, and application isolation. The independent factory structure and the fault-tolerant queuing mechanism position the Notification Engine as a key foundational element of any delivery environment that handles large amounts of traffic.

VIII. ACKNOWLEDGEMENT

The authors would like to thank the extensive Node.js open source communities for the invaluable foundation they provided. The architectural guidelines and design frameworks provided by these communities made the project highly feasible. The underlying asynchronous nature of Node.js, including the robust Express.js framework to handle API requests seamlessly, the advanced ioredis library to ensure stability with Redis and the wonderful BullMQ community, played a significant part in designing the logic necessary to handle high concurrency, ensure transactional safety, and allow quick deployment of local queues. Additionally, we would like to thank the larger community of technology design patterns and enterprise software development for the extensive documentation available on architecture designs, including factory design pattern, object-oriented methodologies, and asynchronous back-off strategy. The tutorials offered by the tech design patterns community on event-driven mechanisms have helped us construct error loops and thread workers efficiently.

Finally, we would like to thank the developers and DevOps engineers who share critical production experiences. These individuals' expertise in system scalability, PM2 process mapping, Dockerization, and container networking helped enhance the final performance characteristics of our platform greatly. Their knowledge about dead-letter handling systems helped establish an essential need for developing internal error handlers. Fourth, we also recognize the role played by external SaaS services, such as Twilio, SendGrid, and Firebase. These define the modern communications infrastructure whose SDKs point out exactly why asynchronous decoupling is necessary. Dealing with their network complexities was the driving force behind understanding how queues should deal with their unreliable nature while laying stable foundations in production systems.

All of these independent resources contributed significantly to the development of our design. Their rapid advancements have led to a drastic improvement in software's ability to communicate dynamically within the tech industry.

IX. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. https://www.oreilly.com/library/view/design-patterns-elements/0201633612/

[2] Taskforces Inc. (2025). BullMQ Official Documentation: Message Queue and Background Jobs for Node.js. https://docs.bullmq.io/

[3] Redis Ltd. (2025). Redis Distributed Locks (Redlock). https://redis.io/topics/distlock

[4] Node.js Foundation. (2025). Node.js Asynchronous programming and Event Loop. https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

[5] S. Newman. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media. https://www.oreilly.com/library/view/building-microservices/9781491950340/

[6] M. Kleppmann. (2017). Designing Data-Intensive Applications. O'Reilly Media. https://dataintensive.net/

[7] G. Hohpe, B. Woolf. (2003). Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley.

[8] Express.js Team. (2025). Express API Reference. https://expressjs.com/

[9] Event-Driven Architecture Insights - Dev Blogs, 2025. URL: https://dev.to/t/event-driven

[10] Building Resilient Systems with Exponential Backoff, Tech Conf, 2024. URL:https://www.presidio.com/technical-blog/exponential-backoff-with-jitter-a-powerful-tool-for-resilient-systems/

[11] TypeScript Community. (2025). TypeScript Handbook: Interfaces and Abstraction. https://www.typescriptlang.org/docs/handbook/intro.html

[12] Redis Documentation. (2025). Redis Data Structures and Lists. https://redis.io/topics/data-types

[13] Docker Inc. (2025). Dockerizing a Node.js webapp. https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

[14] "Scalability and Performance Evaluation of Message Queues," IJARSCT, 2025.

URL:https://ijarsct.co.in/MessageQueue8904.pdf