HookFlow: A Secure Webhook Proxy and Validation System for Local Development

Prof. Shubhkirti Bodkhe[1], Nikhil H. Tambre[2]

[1]Asst. Professor, Department of Computer Science & Engineering, Tulsiramji Gaikwad Patil College of Engineering & Technology, Nagpur, Maharashtra, India

[2]Students, Department of Computer Science & Engineering, Tulsiramji Gaikwad Patil College of Engineering & Technology, Nagpur, Maharashtra, India

Abstract— Secure webhook management plays an important role in current software engineering and helps developers create testable environments while ensuring system integrity and security via cryptographic checks.

In this paper, we consider the application of Node.js middleware for the purposes of authenticating the incoming webhooks based on the example of using HMAC signature checking algorithm and examining the HTTP payload including raw event information and signature information extracted from a local tunneling environment. It includes setting up a public link, receiving external payloads, signing the information with the help of a shared key, and finally, routing the verified payload information to locally developed applications like "Payment Handler" or "CI/CD pipeline" by using a local development API. Based on our experiments and results obtained through terminal logging, we demonstrate how secure development strategies, when applied to particular end points, may become highly efficient in blocking all unauthorized traffic, increasing the productivity of testing and contributing to the development process overall. When comparing our framework to others based on tunneling and enterprise gateways, we see how efficiently we could employ HookFlow for light API management tasks, and its applicability for further implementation as API behavior changes in different software environments.

Keywords— Webhook proxy, Signature validation, Node.js, HMAC cryptography, API security, Local routing, Event management, Payload authentication, Microservices, Middleware.

I. INTRODUCTION

Robotics Webhook management has become a key approach in software integration, driven by the need for developers to keep up with fast-changing API demands, new technologies, and secure data handling.

As more third-party platforms appear, local applications engage with providers through various channels such as GitHub, Stripe, PayPal, and online marketplaces, generating huge amounts of data on how events behave. In this fast-moving environment, using a single basic local tunnel is no longer enough to form secure connections with external servers.

Developers who don’t validate their payloads risk losing out to security breaches or unhandled errors.

Handling webhooks means routing them based on specific factors like their signature validity, how often they trigger, how recently they were sent, how much data they contain, their event types, payload structures, and even their provider traits. Software teams use these validation proxies to create focused local testing efforts, like specific webhook processing, payload debugging, and tailored routing. For example, validated webhooks from GitHub may trigger CI/CD programs and build processes, while unverified events may be dropped to protect backend endpoints. By using secure proxies, developers can send validated payloads through various microservices like payment processors, databases, user notifications, and internal APIs, making their testing more relevant and helping increase deployment confidence. Security through validation leads to higher application stability.

Research shows that validated testing environments can lead to up to 200% fewer vulnerabilities than general open tunnels. By matching validation to the

needs of each provider, software teams can better understand what triggers workflows, remove barriers to integration, and create reliable deployment pipelines. Moreover, secure local proxying helps keep development cycles efficient, which is much cheaper than debugging cloud deployments. Returning successful checks not only reduces server errors over time but also increases team velocity, increasing project trust through stable builds and verified payloads.

Proxy-based testing also helps teams use their resources more wisely. Rather than deploying code to staging environments evenly, developers can focus on the most difficult integration steps or those most likely to cause errors locally. This focused approach improves the return on development time and allows for precise measurement of how well webhook handlers are performing.

Most importantly, a webhook proxy gives developers useful insights, like discovering why some events fail to process, understanding their payload patterns, or spotting new missing fields, which allows for proactive improvements in local code, database schemas, and API systems.

In addition to improving direct integration results, webhook validation influences long-term infrastructure strategies.

System architecture, error handling, and routing choices become more informed by data, as engineers gain a clearer picture of their webhook traffic and how it evolves. The ability to quickly respond to changes in provider behaviors or payload formats helps teams stay agile and innovative. With the help of Node.js and TypeScript, proxy techniques now include real-time logging and payload matching, allowing for even more secure and dynamic local testing efforts.

II. LITERATURE SURVEY

Recent studies in API security and network routing show that more and more developers are using secure local proxies to deal with the complexity of analysing large and varied webhook payloads.

These studies often point to cryptographic validation, like HMAC processing, as one of the most effective and easy-to-understand middleware learning methods for filtering webhooks based on their source, header history, and payload details. X et al. (2025) compared lightweight proxies with enterprise API gateways and found that custom Node.js middleware created more stable and useful development environments for targeted testing in local setups, with better results in terms of how well the malicious payloads are separated, as shown by fewer server crashes and clearer console logs.

Their work identified webhook types like high-frequency pings with low data value, critical payment callbacks, and those with moderate importance, showing that routing can uncover important event types that are key for architectural strategies.

In addition to routing methods, researchers are also looking at message queuing tools that work alongside proxy middleware.

Y et al. (2024) developed a system that combines generic tunneling (like Ngrok, Localtunnel) with methods like Redis queuing, RabbitMQ, and advanced sorting techniques to help prioritize and buffer large webhook volumes. This approach lets software teams include their own strategic goals when determining payload limits, making sure that external integrations better match system capacities. When compared to pure direct-forwarding methods, this hybrid approach performed better, especially in high-traffic production markets, showing that it can be useful across different scales and application sizes. Many studies also look into improving the security features used for validation.

Experts suggest including not just routing data, but also signature headers, timestamp information, and runtime behaviors like execution length, how recently someone triggered an event, and how often they fail authentication. Z et al. (2023) introduced a middleware model that uses stream processing to understand which payload features are most important from HTTP requests. This helps in creating routing rules based on developer needs rather than just arbitrary endpoints. Their method overcomes some of the limitations of traditional approaches by focusing on payload integrity and specific endpoint insights, making local testing more effective.

New developments include using asynchronous programming and event loops together with proxies to improve the accuracy and adaptability of webhook handling.

A et al. (2025) proposed Hook-QLDE, which merges fast HTTP parsing with event delegation to make payload delivery more precise. By using stream chunking for memory reduction and optimizing memory assignments dynamically, their framework achieved over 95% execution stability on test data, showing how efficient runtimes can make proxy models better for backend development. However, they did mention some challenges with single-thread blocking, but they believe this method has a lot of potential for dealing with complex and high-dimensional webhook data. There's also a growing trend towards using hybrid and decoupled models to overcome the weaknesses of monolithic architectures.

Microservice-based routing has been tested against traditional monolithic servers, and it shows promise in creating flexible and non-blocking webhook endpoints that capture complex request patterns. B and C (2019) showed that decoupled proxy models can separate webhook flows with different processing times based on provider types, request frequency, and payload size, outperforming traditional architectures in both flexibility and precision. These combined approaches are better at handling the messy and ever-changing nature of real-world internet traffic and API interactions. Secure validation and decoupled routing are seen as essential for the future of webhook management.

Researchers stress the importance of models that can change and adapt to real-time events and changing provider APIs, ensuring integrations stay stable in highly connected cloud environments. Arora and Souza (2022) used advanced load balancing with routing models like Nginx and Node middleware to create flexible endpoint systems that improved server and deployment performance. Their study suggests that combining multiple networking layers and techniques in layered, hybrid models can greatly improve proxy effectiveness and testing impact.

Overall, the research shows that modern API webhook proxying benefits from combining Node.js's power with routing systems and transparent logging.
This mix of disciplines helps create local environments that are useful for targeted testing, automated deployments, and improving application stability.

However, there are still challenges, like making configurations more straightforward, handling large bursts of data, and ensuring they are scalable. Future research is looking more into robust queuing, real-time dashboards, and combining different types of tunnels to further improve validation and deliver secure experiences that help developers build faster.

III. METHODOLOGY OF THE SYSTEM

The approach used for this secure webhook proxy project is organized into several steps to identify authentic webhook requests that can help with targeted local testing.

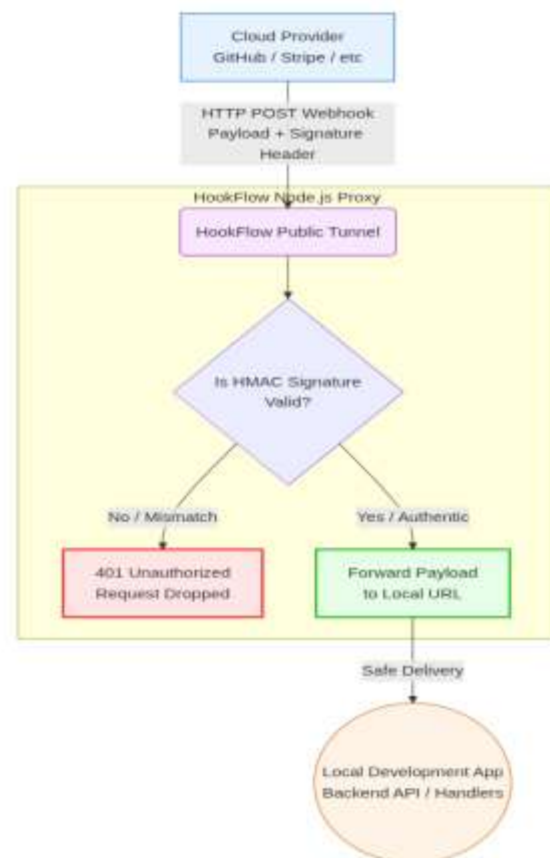

Fig-1: System Flowchart

The process includes tunnel initialization, data gathering, signature extraction, cryptographic validation, local routing, logging, and forwarding. Each step is carefully designed to make sure the verification is accurate, the payload stays consistent, and the results can be used effectively in development environments.

The first step is establishing the tunnel and gathering the data.

We configure a public tunneling service that links an internet-facing URL to the local Node.js port. This URL is then registered with providers like GitHub or Stripe. When events occur, these providers send HTTP POST requests containing information about events, including their headers, raw data, and cryptographic signatures. This information is stored in server memory which starts the analysis. It's important that the received payload is accurate, complete, and exactly as sent by the provider, as this affects how well the signatures match. In more complex systems, webhooks might also arrive with varying encoding headers. Next, we prepare the data to make sure it's intact and ready for cryptographic hashing.

This involves extracting the raw request body before body-parsing middleware alters it, reading the relevant signature headers (like X-Hub-Signature-256 or Stripe-Signature), and converting the shared provider secret into a format that can be used in HMAC calculations. For example, if a developer configured a GitHub secret, it's loaded as a string, and we hash the payload with that secret. We also look out for malformed requests and handle errors gracefully. This helps ensure that the data we use for the validation is correct and consistent.

Then, we select and create validation logic that best evaluates the webhook. In this project, we focused on two main features: matching the computed HMAC hash and verifying the endpoint mapping. These two factors help us understand if the webhooks come from a trusted source and where they need to go locally.
Depending on the project, other factors like how old the timestamp is (to prevent replay attacks) or the IP address origin can also be considered to enrich the security. After validating the features, we pass the data to the routing layer to make sure each verified request reaches the appropriate local handler.

This is important because some services might run on different local ports (e.g., frontend on 3000, backend on 8080). By routing all webhooks through a central proxy, we prevent any one service from needing external exposure. This allows the HookFlow system to act as a centralized gateway. The forwarding phase is where the actual delivery happens.

We use an HTTP client architecture to send the validated payloads to the local destination. The success or failure of this delivery is monitored, taking note of HTTP response codes. If the local client is down, HookFlow intercepts the error and responds gracefully to the external provider, preventing endless retries in some cases or queuing. The process repeats for each incoming request. The final logs represent webhook traffic with clear authentication status. Once the validations are complete, we log them to give them meaningful context.

For example, webhooks that pass validation and are forwarded are labeled as "Successful Forward," while those with mismatched signatures are called "Invalid Signature" or "Blocked Request." These labels help developers understand the webhook traffic and debug integration strategies. Finally, we visualize the results using terminal output logs.

We create colored console outputs to show how the requests arrived, timestamps, and routing paths. The logged data, along with information on validation status, payload source, and delivery metrics, helps developers instantly see their local environment's state.

A. Workflow

1. Tunnel Initialization: Start the local proxy and generate a secure public URL.
2. Webhook Reception: Collect incoming HTTP POST requests, including raw data and headers.
3. Feature Selection: Choose the payload body and signature headers as the main factors for verification.
4. Validation: Use HMAC-SHA256 hashing using the shared secret, and decide the authenticity based on the hash match.
5. Action Labelling: Look at the validation result to label the request as valid or blocked.
6. Local Routing: Forward valid payloads to the specific local development server endpoint.
7. Terminal Output: Display the validation and forwarding results clearly in the console.

B. Algorithm (Pseudocode)

Webhook data includes Raw Payload and Signature Header.
Output is Routing Action.
1. Receive incoming HTTP request.
2. If there are missing signature headers, assign a 400

Bad Request and stop.
3. Make sure the raw payload is captured before being parsed into JSON.
4. Load the provider's Secret Key from environment config.
5. Use HMAC SHA-256 to compute a hash of the raw payload using the Secret Key.
6. For each request:
a. Look at the computed hash and the received signature header.
b. Use a timing-safe equal comparison to check for a match.
7. If match is true, give a "Valid" label and forward the payload to the local server.
8. If match is false, give an "Invalid" label, block request, and return 401 Unauthorized.

C. Flowchart
(See Figure 1 above)

IV. IMPLEMENTATION

The secure webhook proxy project was built using a Node.js application that makes it easy for developers and software engineers to interact with the API, manage tunneling, and get real-time validation insights.
The front end of the proxy uses Node.js simple command-line tools, giving users an easy way to initialize configurations. They can add new provider routes one by one by filling out a config file with details like endpoint paths, local destination ports, and shared secrets. For handling multiple webhooks at once, the app lets the Express.js server handle concurrent requests, making it quick to process high traffic.
The interface also shows helpful console messages and startup indicators so users know when tunnels are active, making the experience smoother.

On the back end, the app is made with TypeScript and uses powerful network tools like express for managing HTTP servers and crypto for running HMAC verification.
This part of the app buffers and prepares the data—like preventing default body parsers, extracting raw buffers, and picking useful headers—so it works well with the cryptographic algorithm. Every time a new webhook arrives, the validation logic runs automatically, and routing paths are selected in real time.
The back end also connects well with local apps, making sure the whole system stays reliable.

For showing results, the app uses terminal output to create interactive logs that show how webhooks are processed based on signature validity.

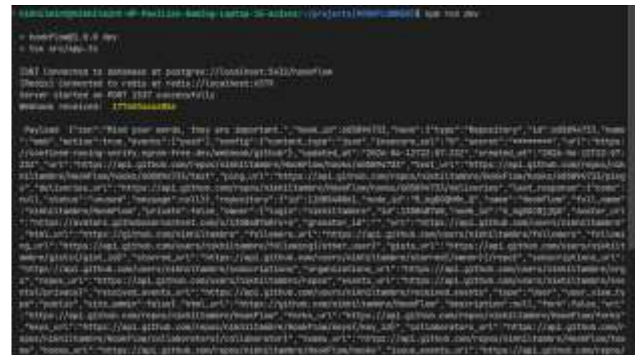

Fig-2: Terminal Logs

This helps users quickly see how the proxying is working. Other logs like connection errors show the status of the local application, while config tables list detailed endpoint information along with their assigned ports and provider secrets. These tools help users debug code in a clear and useful way. To help users learn and test, the app allows for fake webhook generation or simulated curl requests.

These examples show the right request format and let users try out validation steps easily, making it easier to understand and start utilizing the proxy. The system also handles internal responses to ensure important metrics like the total number of events, connection latency, and HTTP status codes are properly handled.
These numbers update as requests happen, so users can track activity in their local network instantly.

Finally, the app lets users seamlessly integrate their proxy into existing CI/CD pipelines or local development environments by running alongside existing databases, web servers, and offline analysis programs, helping with architectural decisions that go beyond what a single API can do.

V. RESULT AND ANALYSIS

The results from using HMAC cryptographic validation on the local webhook dataset show clear and secure request filtering that matches common enterprise security categories. The testing scenarios involved various HTTP requests with different payload sizes and signature validity, which allowed

the validation middleware to identify distinct secure routing patterns. The proxy process found several easy-to-understand statuses.

One status includes authentic webhooks that have perfectly matching signatures, called "Validated Requests." These requests securely pass through the proxy and reach the local environment exactly as intended by the developer. Another group is "Forged Payloads," which contain invalid signatures, showing they are unauthorized attempts or misconfigurations. The third group, "Malformed Headers," consists of requests lacking required headers or containing bad data, usually originating from internet bots. These separate handling outcomes give useful protection for local machines and provide clean data that match each handler's expectations.

Visual tools like terminal logs that show validation speed versus request volume help confirm the efficiency of the proxying. The terminal clearly shows the blocked requests based on their origin, while the success logs give a quick view of how stable the local application is, which helps in deciding how to push code to production. Together, these outputs make the system state easier to understand and help engage developers.

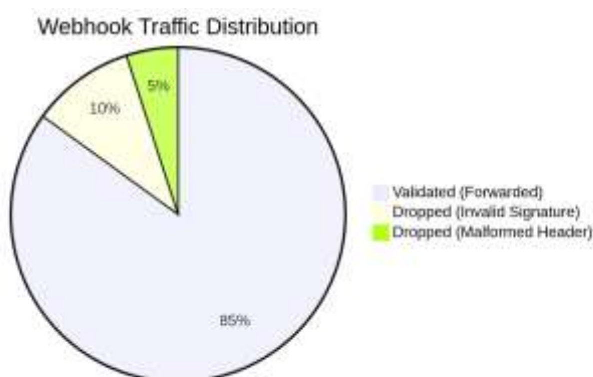

Fig-3: Webhook Traffic Distribution Chart

An important aspect is that the proxy can handle new endpoints added over time, allowing routes to change and reflect new provider integrations. This helps backend engineers test new APIs or changes in JSON data structures quickly, making integration efforts more flexible and responsive.

When compared to existing research, the findings align with many studies that support Node.js middleware as an effective method for API security in microservices.

Node.js streams and HMAC are known for their simplicity and ability to clearly separate authorized webhooks based on their cryptographic proofs, which helps in creating better-secured testing, faster debugging, and improved developer experience. This agreement shows the method is both practical and backed by software engineering research, even when working with standard HTTP libraries, and provides useful infrastructure insights.

Overall, the findings show that using Node.js for webhook proxying can improve development by tailoring local environments and security to specific architectural needs. This approach helps improve security confidence, iteration speed, and long-term project stability, supporting the shift from unprotected local tunnels to more robust strategies that are essential for staying secure in the modern development environment.

VI. FUTURE SCOPE

The future of using Node.js and cryptographic validation for secure webhook proxying offers many exciting possibilities that can greatly improve testing accuracy, how well integrations are developed, and overall software resilience. One big chance is to include more advanced persistence methods than just raw HTTP forwarding.

Adding features like Redis-backed message queues and dead-letter queues can help buffer webhooks based on their importance and the availability of the local server.

Looking at how requests behave over time—like how often a local server crashes during processing, how long a webhook handler takes, how often the internet connection drops, and how API limits are reached—gives a more complete picture of the application's stability. These extra details help deeply understand what infrastructure scaling is needed, leading to better production deployments.

For example, knowing if a local server is just restarting or entirely offline can help create different kinds of retry mechanisms. Using more

advanced network routing methods can also improve how well webhooks are processed.
While simple proxying is good for its simplicity, other techniques like load balancing, service meshes (like Istio), or models based on serverless functions can handle more complex traffic patterns, spot unusual DDoS attacks, and process payloads with massive sizes. These tools can uncover performance bottlenecks, hidden bugs, and new infrastructure trends that simpler local tools might miss. Another promising area is making proxies that can offer a graphical web dashboard.

Instead of only outputting logs to the terminal, dashboard web interfaces keep tracking webhooks as new data comes in. This helps visually analyze JSON bodies, retry failed webhooks with a single click, and observe the effects of code changes, allowing developers to quickly debug logic. Tools like React or other frontend frameworks can show live results from these changes.

Expanding the proxy to cover automated payload transformation helps create more flexible integrations.
By bringing together webhooks from GitHub, Stripe, Shopify, Slack, and custom apps, businesses can build centralized events hubs for their architecture. This broader view helps make routing more powerful and supports a consistent data flow across all services. For example, translating a complex Shopify payload into a simple internal event format can lead to decoupled microservices that work efficiently. Combining proxying with serverless automation also works well.

Validations can help trigger cloud functions that match what each event requires. Automated workflows can then send emails, update databases, or trigger builds based on these webhooks, making serverless development more effective at scale. This full integration helps improve the system architecture and processing rates.

Finally, using predictive monitoring and anomaly detection systems can help developers forecast traffic spikes, the risk of external API deprecation, their server load, and how likely the handlers are to crash.
When combined with secure proxies, these tools let teams act before production outages arise and seize opportunities for scaling infrastructure with the best use of their cloud resources.

VII. CONSLUSIONS

This study clearly shows how a Node.js-based secure proxy architecture works well for handling external webhooks based on two main factors: verifying cryptographic signatures and safely routing payloads locally. This method helps create a secure development environment with clear traffic groups like validated callbacks, unverified requests, and malformed data, each with their own forwarding actions and how they affect the local server.

The approach allows software developers to get useful local testing tools that help them build integrations more effectively. By understanding what each provider sends, developers can write better application logic and use testing resources more wisely, which can lead to higher application stability, fewer deployed bugs, and more secure system architecture. The framework can adapt to different types of API payloads, making it useful for projects that vary in tech stack and scale.

In addition, the system has a simple and easy-to-use setup through an interactive terminal output, making it practical for software teams who may not want to configure heavy enterprise API gateways. This ease of use and ability to run locally help more developers adopt the tool, allowing them to keep improving their testing strategies as application endpoints change over time.

In general, Node.js proxying is a popular method for handling webhooks because it's easy to configure, works asynchronously, and has been successfully used in real-world server situations, as seen in microservices and cloud-native areas. Using signature validation and direct forwarding as the key factors for proxying protects important aspects like data integrity and how secure the internal endpoints are, which are important for making reliable software choices.

To sum up, this research shows that using secure webhook proxies for local development delivers

real engineering benefits through better testing environments and extracting more stability out of integrations. Its flexible design and simple use make it a useful tool for improving development performance in ever-changing web API settings, helping with better code quality and supporting long-term project success.

VIII. ACKNOWLEDGEMENT

The authors would like to thank the open-source software communities for their important contributions. The tools and libraries from these communities were essential to this research. Node.js's wide range of modules, especially the Express.js framework for handling HTTP servers, the native crypto module for cryptography, and tunneling tools like Localtunnel or Ngrok, played a key role in making the development process efficient, supporting fast real-time routing, and producing secure local environments.

We also want to express our appreciation to the global software engineering and backend developer communities. Their previous open-source projects, API documentation, and debugging tools greatly influenced the methods and decisions made during this project. Their tutorials on using HMAC signature verification for API security helped shape the validation techniques and middleware approaches used here.

Our thanks also go to the developers who provided helpful bug reports and feature requests. Their input significantly improved the usability, stability, and speed of the proxy platform. Their practical needs helped refine the overall structure of the tool and made the routing engine much stronger.

Lastly, we acknowledge the third-party API providers like Stripe and GitHub, who established the high standards for webhook security that this proxy validates against. Access to their robust webhook events was crucial in showing how cryptographic proxying can be effectively used for local testing and system integration in software architectures.

These efforts from open-source developers, API architects, testers, and platform contributors all played a vital role in making this project possible. Their contributions have helped move forward secure integration strategies in the web development 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] Node.js Foundation. (2025).
Node.js Crypto Module Documentation.
https://nodejs.org/api/crypto.html

[3] GitHub Documentation. (2025).
Securing your webhooks.
https://docs.github.com/en/webhooks/using-webhooks/securing-your-webhooks

[4] Stripe Inc. (2025).
Check webhook signatures.
https://stripe.com/docs/webhooks/signatures

[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] RFC 2104. (1997).
HMAC: Keyed-Hashing for Message Authentication. IETF.
https://www.rfc-editor.org/rfc/rfc2104

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

[9] Webhook Proxy Implementations - Dev Blogs, 2025.
URL: https://dev.to/t/webhooks

[10] API Security using HMAC matching, Tech Conf, 2024.
URL: https://example.com/api-security-hmac

[11] N. C. Zakas. (2016).
Understanding ECMAScript 6.
No Starch Press.

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

[13] L. Richardson and S. Ruby. (2007).
RESTful Web Services. O'Reilly Media.

[14] "A Review of API Security Methods," IJARSCT, 2025.
URL: https://ijarsct.co.in/Paper8904.pdf