WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

Request Lab: A Visual API Monitoring and Failure Diagnosis Platform for Automated Workflows

Varun D, Vishnu Varshat K, Sathiya Seelan G, Sanjay C, Dhinakaran S

Sri Shakthi Institute of Engineering and Technology

ABSTRACT

Modern applications rarely work in isolation. A single business process may depend on a messaging service, a backend webhook, an AI service, and an order-management API. If one of these parts stops working, the whole process can be delayed or fail. Finding the cause is not always straightforward, especially when the information is spread across separate dashboards or expressed in technical error messages. This paper introduces Request Lab, a proposed visual tool for monitoring API-based workflows and helping users understand integration failures. It displays services as connected nodes, shows their health status, identifies downstream steps affected by a failure, and provides plain-language troubleshooting suggestions. The prototype is designed to use simulated responses and failure cases first, so its main features can be demonstrated without live credentials. The paper describes the system design, monitoring approach, and a plan for evaluating it. Real provider integrations and automated recovery are treated as future work rather than completed features.

Keywords: API monitoring, automated workflows, failure diagnosis, dependency analysis, incident management.

I. INTRODUCTION

APIs allow applications to exchange information and work together. They are used in everyday processes such as receiving an order, sending a customer notification, updating inventory, and storing a transaction. These tasks often involve several services, with each step depending on the output of the one before it.

When one service fails, the effect can spread through the workflow. For example, an expired access token may stop a request from reaching an order system, while a timeout can leave the next step waiting for a response. The first error may be small, but the user may only notice that the overall task did not finish. Diagnosing the issue can require checking logs, API responses, and separate provider dashboards.

Request Lab is intended to make this process easier to follow. It presents an automated workflow as a visual pipeline, with a status shown for each service and connections indicating dependencies. When a check fails, users can inspect the error, see which later steps may be affected, and review suggested actions. The initial version focuses on demonstrating these ideas with simulated data before connecting to real services.

The project has five practical goals: show workflow structure clearly; check service health and expected responses; separate actual failures from steps that are blocked by another failure; explain common problems in accessible language; and keep enough incident history to help users confirm whether a fix worked.

II. PROBLEM STATEMENT

Many API tools are designed to test one request at a time. This is useful for developers, but it may not answer the broader questions that arise during automation: Which part of the workflow failed? What depends on it? Which steps never ran because of that failure? Has the workflow recovered after a change?

Request Lab addresses this gap by combining a visual dependency map with service checks, incident details, and guided troubleshooting. A key design choice is to distinguish a service that has actually failed from a downstream step that is simply unable to run. This avoids presenting every affected step as a separate root-cause failure.

III. OBJECTIVES

Request Lab aims to show APIs and services as connected nodes; check service health and validate responses; distinguish healthy, warning, failed, and blocked states; identify downstream effects; explain common errors in plain language; maintain incident history; and confirm recovery through new checks.

IV. RELATED WORK

API testing tools such as Postman help users send requests, inspect responses, and define assertions. Monitoring and observability platforms collect metrics, logs, and traces to support operational diagnosis. Distributed tracing can show how a request moves through several services, while workflow editors use nodes and connections to explain the order of tasks.

Request Lab draws on these established approaches, but its emphasis is on bringing the workflow view, health state, incident context, and next-step guidance together in one interface. It is not intended to replace existing testing or monitoring products. Instead, it explores how their core ideas can be presented in a way that is easier to follow when troubleshooting a multi-step integration.

V. PROPOSED SYSTEM

The system is organised around a visual dashboard and a monitoring engine. The dashboard shows the pipeline, summary status, selected-service details, and incident history. A user can select a node to review its latest check, the observed error, likely causes, and recommended troubleshooting steps.

For the prototype, React and TypeScript can be used to build the interface, with React Flow providing the canvas for nodes and connections. A simulated monitoring engine will return predefined healthy and failing responses. This allows the team to demonstrate how the interface behaves when a service times out, returns an error, or recovers after a retry, without needing API keys or modifying real systems.

A later version could use a Python/FastAPI backend to run actual checks and store workflow definitions and incident records. PostgreSQL could hold durable history, while a scheduler and background worker could support recurring checks. These are planned extensions; they are not assumed to be present in the mock-data prototype.

VI. SYSTEM MODEL AND WORKFLOW

Request Lab represents a workflow as a directed graph. Each node stands for an API, service, or processing step, and each directed edge shows that one step depends on another. A representative e-commerce flow is: WhatsApp Business API → backend webhook → OpenAI API → Amazon Selling Partner API → order confirmation.

Each node uses a clear status: green for healthy, red for a confirmed failure, yellow for a warning, and gray for a blocked or not-yet-tested step. If the AI service fails, the order-submission and confirmation steps may be shown as blocked because they cannot proceed. The upstream services should remain healthy if their own checks passed.

The monitoring process has five stages. First, the user configures or imports a workflow. Second, the system runs health checks and validates the responses. Third, it updates the visual status and identifies downstream effects. Fourth, it records the incident and presents troubleshooting suggestions. Finally, the user can rerun checks to see whether the workflow has recovered.

VII. HEALTH CHECKS AND RECOVERY

A check should reflect what the service is expected to do, not just whether it responds. Depending on the integration, the check may consider reachability, response time, HTTP status, authentication outcome, and required response fields. A successful HTTP response can still contain missing or invalid data, so response validation is important.

The first version can use predefined diagnostic rules for common cases. For example, an authentication error can prompt the user to review the token and permissions; a timeout can prompt a check of service availability and network conditions; and a rate-limit response can suggest reviewing request frequency and provider limits. The system should avoid claiming certainty when the evidence is incomplete. For instance, an HTTP 429 response indicates rate limiting or a related usage restriction, but does not by itself prove that an account has run out of credits.

Each incident can record the affected service, time detected, error category, status, and resolution notes. These records help users compare incidents and see whether a later check confirms recovery. AI-generated explanations may be considered in a future version, but the initial prototype can remain transparent by using explicit rules.

Retrying a request is not the same as proving that the workflow is fixed. A service may respond again while still returning incomplete data or failing a later business step. Request Lab should therefore update a node to healthy only after its required checks pass. Where dependencies matter, the relevant downstream steps should also be tested in the correct order.

Retries should be controlled rather than unlimited. A production implementation would need safeguards such as retry limits, backoff, clear user confirmation for actions that could create duplicate transactions, and audit logs. The prototype can demonstrate the status transition using simulated outcomes without performing real external actions.

VIII. EVALUATION PLAN

The prototype should be tested with repeatable scenarios, including a fully healthy workflow, an authentication failure, a timeout, an invalid response, a rate-limit warning, a downstream blocked step, and a successful recovery. For each scenario, the expected status of every node should be defined before the test is run.

Useful evaluation measures include whether the system identifies the correct failing node, correctly marks dependent steps as blocked, displays an appropriate explanation, preserves incident history, and reports recovery only after checks pass. A small usability study could ask participants to identify the cause of a failure and choose a next step using Request Lab, then compare their performance with a text-only log view. No measured results are claimed in this paper; these tests are proposed work.

IX. LIMITATIONS AND FUTURE WORK

The mock-data prototype can demonstrate the user experience and status logic, but it cannot establish how well the system performs against every real provider or production workload. Actual integrations will need secure credential storage, access controls, provider-specific error handling, rate-limit awareness, and careful treatment of sensitive data. Monitoring frequency and network conditions may also affect the accuracy of health checks.

Future work could add real API connectors, scheduled monitoring, alerts, team-based access, richer incident analytics, and integrations with existing observability systems. Any automated repair feature should be introduced gradually, with clear permissions and safeguards for actions that may affect customer data or create business transactions.

X. CONCLUSION

Request Lab proposes a practical way to understand failures in workflows that depend on multiple APIs and services. By showing service health, dependency impact, incident context, and recovery checks in one visual view, it aims to make troubleshooting less fragmented and easier to follow. A simulated prototype provides a manageable first step: it lets the team test the core interactions before taking on the complexity and risks of live integrations. The next stage is to implement the prototype, evaluate it with defined failure scenarios, and use those findings to guide further development.

REFERENCES

[1] Postman, “Postman Documentation.” https://learning.postman.com/docs/

[2] OpenTelemetry, “Documentation.” https://opentelemetry.io/docs/

[3] OpenAI, “API Error Codes.” https://platform.openai.com/docs/guides/error-codes

[4] Amazon, “Selling Partner API Documentation.” https://developer-docs.amazon.com/sp-api/

[5] React Flow, “React Flow Documentation.” https://reactflow.dev/

[6] FastAPI, “FastAPI Documentation.” https://fastapi.tiangolo.com/