# THE POWER OF TEN – RULES FOR TESTING HTTP APIS

**BY LIUDAS JANKAUSKAS**

[First version: September 2026]
[qaontime.com]
pdf-version: [PDF]
www-version: [WWW]

## ABSTRACT

Explicitly inspired by Gerard J. Holzmann's The Power of Ten – Rules for Developing Safety-Critical Code [1], this paper applies the concept of a small, memorable, and enforceable rule set to API testing. It proposes ten implementation-agnostic rules derived from recurring failure patterns observed during nearly two decades of testing API-driven systems across multiple industries.

The rules address critical areas frequently underrepresented in conventional API testing, including input validation, authorization, response semantics, predictable failure handling, protocol behavior, operational constraints, and lessons learned from production incidents.

Rather than defining an exhaustive testing methodology, The Power of Ten – Rules for Testing HTTP APIs establishes a compact set of principles that can guide both manual and automated testing. Its objective is to help teams detect systemic weaknesses earlier and improve the reliability, security, robustness, and long-term maintainability of APIs.

## INTRODUCTION

Postman's 2025 State of the API Report found that 82% of surveyed organizations had adopted some level of API-first development, illustrating how central APIs have become to modern software systems [7]. API testing is often considered sufficient once an endpoint accepts a valid request and returns the expected business data. However, this confirms only a narrow part of its behavior. An API also defines how a system responds to invalid input, insufficient permissions, unsupported methods, repeated requests, concurrent operations, large result sets, dependency failures, and increasing load. These behaviors are not secondary technical details. For an API consumer, they are part of the product.

Many serious API defects survive conventional testing precisely because the expected business scenario works. The weakness appears outside that scenario: a forbidden operation is reported as an authentication failure; an unsupported method returns a misleading response; malformed input reaches an unhandled code path; one nonexistent identifier returns 404 while another causes 500; or an unbounded query remains unnoticed until the volume of data increases.

Effective API testing must therefore examine more than whether the returned data is correct. It must evaluate response semantics, authorization boundaries, input handling, consistency of failure, resource constraints, state transitions, and operational behavior. These concerns apply whether testing is performed manually, through automated test suites, or with exploratory testing tools [3].

The ten rules presented in this paper are intended to guide that work. They do not prescribe a particular tool or attempt to enumerate every possible test case. Instead, each rule identifies a class of risk that should be considered whenever an API is designed, tested, changed, or investigated after a production incident. Together, the rules help teams move beyond endpoint-level happy-path validation and evaluate the API as a complete, observable system.

## RULE SELECTION AND SCOPE

The initial rules were derived from recurring API failure patterns observed during 18 years of professional testing across e-commerce, high-volume advertising platforms, banking, blockchain-based applications, government systems, and mission-critical environments. They were further refined through systematic exploration of API behavior beyond expected business scenarios.

Each candidate rule was evaluated against five criteria. It had to:

- Address a recurring rather than project-specific problem.
- Represent a significant functional, security, reliability, or operational risk.
- Remain applicable across HTTP API implementations, architectures, and domains.
- Describe a concern that can be tested and supported by observable evidence.
- Be sufficiently distinct from the other rules to avoid conceptual overlap.

Candidates that were too narrow, technology-specific, difficult to verify, or already covered by a stronger rule were removed or merged. Conceptual overlap was assessed according to the underlying risk and testing objective, not merely the observable signal used as evidence. Two rules could therefore involve the same response element while testing fundamentally different properties.

The remaining candidates were cross-checked against established HTTP semantics, API security guidance, and publicly documented industry failure patterns. This comparison was used to confirm their broader relevance and sharpen the boundaries between individual rules.

The final set was limited to ten following the conceptual precedent established by Gerard J. Holzmann's *The Power of Ten*. The limit served as a prioritization constraint: narrower, lower-impact, or overlapping concerns were excluded in favor of rules with broader applicability, greater potential consequences, and clear testing outcomes.

## 1. EVERY ENDPOINT MUST BE TESTED AS AN INDEPENDENT ATTACK SURFACE

**Explanation**

Every API endpoint should be treated as an independent attack surface regardless of its intended audience or deployment scope. The distinction between "public" and "internal" APIs is often temporary and operational rather than technical.

A common assumption in software projects is that internal endpoints require less scrutiny because they are not directly exposed to external consumers. In practice, this assumption frequently proves incorrect. Internal APIs are routinely consumed by additional teams, third-party integrations, mobile applications, partner systems, automation tools, and future services that were not part of the original design.

If an endpoint can receive requests from another system, application, or user, it should be considered externally reachable from a testing perspective. The fact that an endpoint is currently used by only one client does not reduce the potential impact of security, authorization, performance, or data exposure defects.

For this reason, every endpoint should be evaluated as an independent entry point into the system and tested accordingly.

**Example**

Many organizations initially develop APIs exclusively for their own web or mobile applications. Over time, external consumers discover that these applications communicate through well-defined HTTP endpoints and begin interacting with the APIs directly.

This pattern has been observed repeatedly across the industry. APIs originally intended for internal application use eventually become unofficial integration points for customers, partners, automation scripts, or third-party services. Once this occurs, assumptions about trusted consumers quickly become invalid.

A common anti-pattern is the statement: "This endpoint is internal, therefore it does not require the same level of testing."

History shows that internal endpoints often become external dependencies without any architectural changes to the API itself.

**Testing Guidance**

Each endpoint should be tested independently for:

- Authentication requirements
- Authorization boundaries
- Input validation
- Error handling
- Rate limiting
- Sensitive information disclosure
- Supported HTTP methods
- Response code correctness

Testing should not assume that an endpoint is safe merely because it is currently consumed by a trusted application.

Similarly, client-side controls such as CORS policies should never be considered a security boundary. While CORS may restrict browser-based requests, it does not prevent direct server-to-server communication, automation tools, scripts, or custom clients from interacting with the endpoint.

From a testing perspective, if a request can reach the endpoint, the endpoint must be treated as part of the attack surface.

## 2. EVERY ENDPOINT MUST BE TESTED WITH INVALID INPUT

**Explanation**

An API endpoint that has only been tested with valid input has not been adequately tested.

Most software defects do not occur when users provide expected data under expected conditions. They occur when systems receive malformed, unexpected, incomplete, oversized, or otherwise invalid input that was not anticipated during development.

A properly designed API should clearly define what input is accepted and reject all input that falls outside those boundaries. Invalid requests should be handled gracefully and consistently without causing application failures, data corruption, service degradation, or unpredictable behavior.

Input validation is not merely a usability concern. It is a reliability, security, and operational stability requirement. Invalid data accepted into a system often becomes significantly more expensive to correct once it propagates through databases, integrations, reporting systems, analytics pipelines, or downstream services.

For this reason, API testing should focus not only on proving that valid requests succeed, but also on proving that invalid requests fail safely [5].

**Example**

Consider an endpoint responsible for creating a user account:

```
{
"name": "John Smith",
"email": "john@example.com",
"phone": "+971501234567"
}
```

A successful test confirms that valid data is accepted and stored correctly.

However, significantly more information can be obtained by submitting invalid requests, such as:

- Invalid email formats
- Missing mandatory fields
- Unexpected properties
- Incorrect data types
- Oversized payloads
- Empty strings
- Null values
- Malformed JSON

A common anti-pattern occurs when invalid requests trigger internal server errors (HTTP 500). In such cases, malformed user input is capable of causing application failures rather than being rejected through proper validation.

Another frequently observed failure occurs when extremely large values are accepted without restrictions. For example, an API may successfully store a user name containing several megabytes of text. While the initial request succeeds, the oversized data can later disrupt reporting systems, user interfaces, search functionality, mobile applications, integrations, or other dependent services.

The resulting failure often appears far away from the original source of the problem, making diagnosis significantly more difficult.

**Testing Guidance**

Every endpoint should be tested with invalid and unexpected input, including:

- Missing required fields
- Invalid formats
- Invalid data types
- Null values
- Empty values
- Boundary values
- Oversized strings
- Oversized payloads
- Malformed JSON structures
- Unexpected properties

The expected outcome should be a controlled client error response, typically within the 4xx range, accompanied by a meaningful validation message.

Invalid input should never result in:

- HTTP 500 Internal Server Error
- Application crashes
- Database corruption
- Service instability
- Resource exhaustion
- Unbounded memory consumption

From a testing perspective, the objective is not to prove that valid data works. The objective is to identify how the system behaves when assumptions about the input are violated.

## 3. RESPONSE CODES ARE PART OF THE FUNCTIONALITY

**Explanation**

HTTP response codes are not implementation details. They are part of the observable behavior of an API and therefore part of its functionality.

An API communicates not only through response bodies, but also through status codes. These codes define whether a request succeeded, failed due to client input, failed due to authentication, failed due to authorization, failed because a resource does not exist, or failed because the server encountered an unexpected condition [2].

Incorrect response codes create ambiguity for both humans and systems. Developers integrating with the API may debug the wrong problem, client applications may handle errors incorrectly, monitoring systems may classify incidents inaccurately, and security weaknesses may be exposed unintentionally.

For this reason, response code correctness should be tested as deliberately as response body correctness.

**Example**

A common anti-pattern occurs when an unauthenticated request returns 403 Forbidden instead of 401 Unauthorized. In such a case, the client may incorrectly assume that authentication succeeded but the user lacks permission. Developers can then spend unnecessary time debugging authorization rules when the actual problem is missing or invalid authentication.

Another common failure occurs when authorization behavior leaks resource existence. For example, assume a user requests another user's resource:

```
GET /users/12345/orders
Authorization: Bearer valid-token-for-different-user
```

If the API returns 403 Forbidden, it may confirm that the resource exists but the requester is not allowed to access it. If the same endpoint returns 404 Not Found for a non-existing resource, an attacker can distinguish between existing and non-existing identifiers. This creates a resource enumeration risk.

Incorrect method handling is another frequent anti-pattern. If a client sends an unsupported HTTP method and receives 400 Bad Request, the response incorrectly suggests that the request syntax or body is invalid. The correct response should normally indicate that the method itself is not allowed, such as 405 Method Not Allowed. Without this distinction, developers may waste time debugging request payloads when the actual problem is the HTTP method.

**Testing Guidance**

API testing should verify response codes for both successful and unsuccessful scenarios.

At minimum, each endpoint should be tested for:

- Successful requests
- Missing authentication
- Invalid authentication
- Insufficient authorization
- Non-existing resources
- Access to resources owned by another user
- Invalid input
- Malformed request bodies
- Unsupported HTTP methods
- Unexpected server failures

Expected behavior should be explicit and consistent. Common expectations include:

- 200 OK for successful retrieval
- 201 Created for successful resource creation
- 204 No Content for successful operations without a response body
- 400 Bad Request for malformed or invalid client input
- 401 Unauthorized for missing or invalid authentication
- 403 Forbidden for authenticated users who are not permitted to perform an action
- 404 Not Found when a resource does not exist or when revealing its existence would create a security risk
- 405 Method Not Allowed when the HTTP method is not supported
- 409 Conflict for state conflicts
- 429 Too Many Requests when rate limits are exceeded [8]
- 500 Internal Server Error only for unexpected server-side failures

The important point is not that every API must use every status code in the same way. The important point is that the API must use response codes deliberately, consistently, and as part of its contract. From a testing perspective, an incorrect response code is a functional defect.

## 4. AUTHORIZATION MUST BE TESTED SEPARATELY FROM AUTHENTICATION

**Explanation**

Authentication and authorization are closely related concepts, but they address fundamentally different questions.

Authentication determines who is making a request.

Authorization determines what that authenticated entity is allowed to do.

In practice, many testing efforts focus heavily on authentication mechanisms such as login flows, token generation, session management, or identity verification, while authorization receives significantly less attention. As a result, systems may successfully verify user identities while still exposing data or functionality to unauthorized users.

From a testing perspective, proving that a user can authenticate successfully does not provide any assurance that authorization controls are implemented correctly.

Authorization failures are often among the most severe defects discovered in production systems because they can result in unauthorized access to sensitive information, privilege escalation, modification of protected resources, or complete compromise of business functionality [4].

For this reason, authorization must be treated as an independent testing concern and verified separately from authentication.

**Example**

Consider an API that allows users to retrieve their profile information:

```
GET /users/12345/profile
Authorization: Bearer valid-token
```

The endpoint correctly requires authentication and returns the profile data belonging to the authenticated user.

A common testing mistake is to verify only that the user can successfully access their own profile.

The more important question is what happens when the same user attempts to access another user's profile:

```
GET /users/67890/profile
Authorization: Bearer valid-token-for-user-12345
```

If the API returns another user's data, the authentication mechanism may be functioning perfectly while authorization is completely broken.

The defect is not related to identity verification. The defect is that the system failed to enforce ownership boundaries.

Another frequently observed anti-pattern is the assumption that authentication automatically provides security. Development teams often verify that users can log in, tokens are generated correctly, expired tokens are rejected, and invalid credentials are denied. At the same time, insufficient effort is spent validating whether authenticated users can access, modify, delete, or enumerate resources belonging to other users.

A similar problem occurs when authorization testing is performed only through the user interface. Since APIs can be called directly, UI restrictions provide little assurance that backend authorization controls are functioning correctly.

In many real-world incidents, attackers did not bypass authentication. They simply used valid credentials to access resources they should never have been permitted to access.

**Testing Guidance**

Authorization testing should be performed independently of authentication testing.

For each endpoint that exposes user-specific, tenant-specific, or protected data, testers should verify:

- Access to owned resources
- Access to resources owned by other users
- Access using different user roles
- Access using elevated roles
- Access using restricted roles
- Cross-tenant access attempts
- Resource modification attempts
- Resource deletion attempts
- Enumeration attempts using predictable identifiers

Testing should focus on answering the following question: Can an authenticated user perform actions beyond those explicitly permitted?

A useful practical approach is to create multiple test accounts with different roles, permissions, ownership relationships, and tenancy boundaries, then systematically repeat the same requests using each identity.

Authentication testing proves that users are who they claim to be.

Authorization testing proves that users can do only what they are allowed to do.

From a testing perspective, a successfully authenticated user should never be considered a trusted user.

## 5. EVERY COLLECTION ENDPOINT MUST HAVE BOUNDED RESULTS

**Explanation**

Every API endpoint that returns a collection of objects should enforce explicit limits on the amount of data returned in a single request.

While returning all available records may appear acceptable during early development or testing, unbounded collection endpoints inevitably become a scalability and reliability problem as data volumes grow. An endpoint that performs adequately with hundreds of records may become unusable when the dataset grows to thousands, millions, or tens of millions of records.

Bounded result sets protect both the server and the client. They reduce memory consumption, database load, network traffic, response times, and the risk of service degradation under normal or abusive usage patterns.

For this reason, collection endpoints should implement mechanisms such as pagination, result limits, cursor-based navigation, or other approaches that explicitly constrain the amount of data returned in a single response.

**Example**

Consider an endpoint that returns customer records:

```
GET /customers
```

During initial development, the system may contain only a few hundred customers, and returning the entire dataset appears harmless.

Over time, however, the number of records may grow significantly. The same endpoint can eventually return hundreds of thousands or millions of objects in a single response. What was once an acceptable implementation can become a source of excessive database load, increased memory consumption, network congestion, and poor user experience.

A common anti-pattern is the assumption that current data volumes represent future production conditions. As a result, endpoints are released without pagination, limits, or filtering capabilities.

Another frequently observed anti-pattern occurs when APIs technically support pagination but do not enforce reasonable limits. In such cases, a client may simply request:

```
GET /customers?limit=1000000
```

effectively bypassing the intended protection mechanisms.

In production environments, unbounded collection endpoints often become attractive targets for resource exhaustion attacks because a single request can trigger disproportionate consumption of computational resources.

**Testing Guidance**

For every endpoint that returns a collection of objects, testers should verify:

- Pagination support
- Maximum page size limits
- Default page size behavior
- Cursor-based navigation, where applicable
- Filtering capabilities
- Sorting behavior
- Performance under large datasets
- Response times near configured limits
- Protection against excessive page sizes

Testing should include attempts to:

- Request unusually large page sizes
- Omit pagination parameters entirely
- Retrieve complete datasets repeatedly
- Combine pagination with complex filters and sorting
- Access collections containing large numbers of records

The objective is not only to verify that pagination exists, but also to confirm that the API remains predictable, performant, and resilient as data volumes increase.

From a testing perspective, an endpoint that can return an unbounded dataset should be considered a scalability defect waiting to become a production incident.

## 6. APIS MUST FAIL PREDICTABLY

**Explanation**

Failure is an inevitable characteristic of all software systems. Network interruptions, unavailable dependencies, exhausted resources, database outages, invalid requests, software defects, and unexpected runtime conditions cannot be eliminated entirely.

The objective of API testing is therefore not to prove that failures never occur, but to verify that failures are handled predictably, consistently, and transparently.

A predictable failure provides sufficient information for clients to understand what happened, while preventing unnecessary disclosure of internal implementation details. An unpredictable failure creates ambiguity, complicates troubleshooting, and may expose weaknesses that can be exploited intentionally or unintentionally.

From a consumer's perspective, an API should behave deterministically. Similar failure conditions should produce similar responses regardless of when, where, or how often they occur.

**Example**

Consider an endpoint that normally returns customer information.

When the requested customer does not exist, the API returns:

```
404 Not Found
```

When the request is malformed, the API returns:

```
400 Bad Request
```

When the client exceeds a configured rate limit, the API returns:

```
429 Too Many Requests
```

These responses clearly communicate the nature of the problem and allow the client to respond appropriately.

A common anti-pattern occurs when fundamentally different failures all result in generic server errors:

```
500 Internal Server Error
```

For example:

- Invalid input returns 500
- Missing resources return 500
- Unsupported operations return 500
- Validation failures return 500

In such situations, clients cannot distinguish between user mistakes, temporary conditions, and actual system defects.

Furthermore, excessive generation of 5xx responses can create unnecessary operational load and may expose opportunities for resource exhaustion attacks, where malformed requests repeatedly trigger expensive error-handling paths.

A server error should indicate that the server failed to fulfill a valid request. It should not become the default response for expected or controllable conditions.

**Testing Guidance**

API testing should verify not only successful behavior but also failure behavior.

For each endpoint, testers should evaluate:

- Validation failures
- Missing resources
- Authorization failures
- Authentication failures
- Unsupported operations
- Rate limit violations
- Dependency failures
- Timeout conditions
- Unexpected internal errors

Testing should verify that:

- Similar failures produce consistent responses
- Appropriate status codes are returned
- Error messages are meaningful and actionable
- Sensitive implementation details are not exposed
- Error responses follow a consistent structure [9]
- Clients can distinguish between different categories of failures

Particular attention should be given to 5xx responses.

Any reproducible request capable of consistently generating a 5xx response should be investigated as a potential defect, regardless of whether the request itself is valid.

From a testing perspective, predictable failure handling is a feature, not an implementation detail.

## 7. STATE-CHANGING OPERATIONS MUST BE TESTED UNDER RETRY, REPLAY, AND CONCURRENCY

**Explanation**

Any API operation that creates, modifies, transfers, or deletes state must be tested under repeated and overlapping execution. A state-changing request may work correctly when executed once and still produce serious defects when retried, replayed, or processed concurrently.

In distributed systems, a client cannot always determine whether an operation succeeded. A connection may close after the server has committed a change but before the response reaches the client. The client, SDK, gateway, background worker, or message consumer may then repeat the request.

Retry, replay, and concurrency represent related but distinct conditions:

- A **retry** occurs when a request is repeated because the result of the original attempt is unknown or considered unsuccessful.
- A **replay** occurs when a previously submitted request is sent again after the original operation has already completed or outside its intended execution context.
- **Concurrency** occurs when multiple operations affecting the same state overlap before that state has stabilized.

Without deliberate handling, these conditions can produce duplicate payments, repeated refunds, duplicate orders, lost updates, oversold inventory, exceeded limits, invalid state transitions, or partially completed business operations.

HTTP method semantics provide useful expectations, but they do not prove that an implementation behaves correctly. PUT and DELETE are defined as idempotent, yet an implementation may still trigger unintended side effects each time the request is received. Conversely, a POST operation may be designed for safe retry through idempotency keys, request deduplication, unique constraints, or another application-level mechanism.

Testing must therefore evaluate the actual business outcome and resulting system state, not infer safety from the HTTP method or response code alone.

The correct behavior depends on the API contract. A repeated request may return the original result, report that the operation has already completed, or reject the request as a conflict. A concurrent update may be serialized, merged, or rejected because it was based on a stale state. The implementation mechanism may differ, but repetition and concurrency must not silently corrupt state or apply an operation more times than intended.

**Example**

Consider a payment API:

```
POST /payments
Idempotency-Key: payment-order-7842
{
"orderId": "7842",
"amount": 100,
"currency": "EUR"
}
```

The server processes the payment successfully and stores the transaction. However, the network connection closes before the client receives the response. Because the outcome is unknown, the client sends the same request again.

If the API creates a second transaction, each request may appear valid when examined independently, but the resulting business behavior is incorrect: the customer has been charged twice for the same order.

When the same idempotency key and request are repeated, the API should recognize the previously processed operation and avoid creating an additional payment. The returned status and response body may depend on the API contract, but the payment must not be applied twice.

The same protection must work when both requests arrive concurrently. An implementation that checks for an existing idempotency key and only later records it may still allow two service instances to process the payment simultaneously.

Concurrency creates a different variation of the problem. Suppose two clients retrieve the same account or order, modify different fields, and submit their updates at nearly the same time. If both updates are accepted without version or conflict detection, the later write may silently overwrite the earlier one. Each request succeeds, but one valid change is lost.

**Testing Guidance**

Testing should begin by identifying every operation that can change a persistent or externally observable state. This includes not only database updates, but also payments, messages, notifications, audit records, file creation, calls to external systems, and asynchronous jobs.

For each state-changing operation, testing should include:

- Repeating the identical request immediately after a successful response.
- Retrying after a simulated timeout, connection loss, or interrupted response.
- Replaying the request after the original operation has completed.
- Sending identical requests sequentially and concurrently.
- Sending different updates concurrently against the same resource.
- Updating a resource using an outdated version, timestamp, or entity tag.
- Repeating operations across multiple service instances where applicable.
- Interrupting processing between internal or external side effects.
- Executing related operations in an unexpected order.
- Verifying behavior when a completed operation is replayed after an idempotency or deduplication window has expired.

Where idempotency keys are supported, tests should verify:

- Reuse of the same key with the same request.
- Concurrent requests using the same key.
- Reuse of the same key with different request data.
- Missing, malformed, expired, and previously consumed keys.
- Consistency of the response returned for repeated requests.
- Whether deduplication covers all business and external side effects.

Where versioning or conditional updates are supported, tests should confirm that stale operations cannot silently overwrite newer state. Depending on the contract, conflicts may be reported through responses such as 409 Conflict or, when HTTP preconditions are used, 412 Precondition Failed.

Assertions must extend beyond the immediate HTTP response. Tests should inspect the final resource state, related records, external transactions, emitted events, audit history, and asynchronous processing results. Race-condition tests should be executed repeatedly because timing-dependent failures may not appear in every run.

A state-changing operation satisfies this rule only when repeated and overlapping requests produce a defined outcome without unintended duplication, lost updates, partial execution, or broken business invariants.

## 8. EVERY API MUST DEFEND ITSELF AGAINST ABUSE

**Explanation**

API design should assume that clients will eventually behave in unexpected, inefficient, or abusive ways.

Not all abusive behavior is malicious. Poorly implemented integrations, software defects, retry storms, misconfigured automation, excessive polling, and accidental misuse can generate traffic patterns that place significant stress on a system.

A common design mistake is to assume that all API consumers will behave responsibly. In practice, APIs operate in environments where client behavior cannot be fully controlled. As a result, every API should contain mechanisms that prevent individual consumers from consuming disproportionate amounts of shared resources.

The objective is not to prevent all abuse. The objective is to ensure that excessive consumption by one client cannot significantly degrade service availability for others.

For this reason, abuse resistance should be considered a fundamental API quality attribute rather than a purely operational concern.

**Example**

Consider an endpoint that performs an expensive database query.

Under normal usage, the endpoint may receive a few requests per minute and perform adequately. However, a malfunctioning integration repeatedly retries failed requests without delay and begins sending hundreds of requests per second.

Although each request is technically valid, the cumulative effect may exhaust database connections, consume excessive computational resources, increase response times, and reduce availability for other consumers.

A similar situation can occur when clients repeatedly request large datasets, aggressively poll for updates, or perform resource-intensive operations without restriction.

A common anti-pattern is the assumption that valid requests cannot be harmful.

Development teams often focus on validating request structure, authentication, and authorization while overlooking the cumulative impact of request volume. If a request is technically valid, it is frequently assumed that no additional protections are required.

For example, an API may allow clients to repeatedly execute expensive search operations, generate large reports, or retrieve substantial amounts of data without any meaningful restrictions. While each individual request succeeds as designed, the aggregate resource consumption can become unsustainable.

Another common anti-pattern is the belief that abuse prevention is solely an infrastructure responsibility. Development teams may rely entirely on load balancers, API gateways, web application firewalls, or network controls while the API itself contains no meaningful protections against excessive resource consumption.

In practice, abuse resistance is a shared responsibility between infrastructure and application design.

In each case, the system fails not because individual requests are invalid, but because the aggregate volume of requests exceeds the assumptions made during design.

The resulting outage may be operationally indistinguishable from a denial-of-service attack even when no malicious intent exists.

**Testing Guidance**

Testing should evaluate how the API behaves under conditions of excessive or abnormal usage.
This includes verifying:

- Rate limiting mechanisms
- Request quotas
- Retry behavior
- Resource-intensive operations
- Large-volume data retrieval
- Concurrent request handling
- Protection against resource exhaustion
- Fair resource allocation between consumers

Testing should also verify that protective mechanisms respond predictably and communicate clear failure conditions to clients.
The objective is not merely to confirm that limits exist, but to verify that the system remains available, stable, and predictable when consumer behavior exceeds normal operating assumptions.
From a testing perspective, an API that assumes all consumers will behave correctly is an API that has not been fully tested.

## 9. TEST THE PROTOCOL, NOT ONLY THE BUSINESS LOGIC

**Explanation**
Many API testing efforts focus almost exclusively on business functionality.
Testers verify that users can create accounts, retrieve data, update records, submit transactions, and perform other business operations. While these activities are essential, they represent only one aspect of API behavior.
An API is not merely a collection of business functions. It is also an implementation of a communication protocol. HTTP methods, status codes, headers, content negotiation, caching directives, authentication mechanisms, and idempotency guarantees are all part of the API contract.
When protocol behavior is not tested, APIs may appear functionally correct while still violating client expectations, interoperability requirements, performance assumptions, or industry standards.
For this reason, API testing should validate both business behavior and protocol behavior.
**Example**
Consider a customer management API that successfully supports creating, retrieving, updating, and deleting customer records.
From a business perspective, all functionality appears correct.
However, protocol-level defects may still exist:

- Unsupported HTTP methods return incorrect status codes.
- OPTIONS requests are not handled correctly.
- Content-Type validation is missing.
- Invalid Accept headers are ignored.
- Cache-Control directives are absent or inconsistent [10].
- Authentication failures return incorrect responses.
- Idempotent operations behave inconsistently when repeated.

A common anti-pattern is to consider protocol behavior the responsibility of frameworks, gateways, or infrastructure components and therefore exclude it from testing activities.
As a result, testing focuses exclusively on business workflows while protocol-level defects remain undiscovered until they impact integrations, client applications, automated systems, or production environments.
The API may satisfy business requirements while simultaneously violating the protocol expectations of its consumers.
**Testing Guidance**
Protocol-level testing should be performed alongside business functionality testing.
Testing should include verification of:

- HTTP method handling
- OPTIONS requests
- HEAD requests
- Content-Type validation
- Accept header handling
- Response headers
- Authentication protocol behavior
- Caching directives
- Content negotiation
- Idempotency guarantees
- Error response consistency
- Protocol compliance across endpoints

Testing should also verify that the same operation behaves consistently regardless of how many times it is executed when protocol specifications require such behavior.
The objective is not merely to confirm that business functionality works. The objective is to verify that the API behaves as a predictable and standards-compliant protocol implementation.
From a testing perspective, a business function that works correctly but violates its underlying protocol contract should still be considered defective.

## 10. EVERY DEFECT FOUND IN PRODUCTION SHOULD BECOME A REUSABLE API TEST

**Explanation**
No testing strategy can guarantee the discovery of every defect before release.
Despite code reviews, automated testing, manual testing, monitoring, and quality assurance processes, some defects will inevitably escape into production environments. The critical question is not whether defects will occur, but whether the organization learns from them.
A production defect represents evidence that a previously untested assumption existed within the system. Once such an assumption has been exposed, allowing the same defect category to reappear represents a failure of the testing process rather than a failure of the software itself.
For this reason, every significant production defect should result in a reusable test that permanently verifies the associated behavior [6].
The objective is not merely to fix defects. The objective is to continuously expand the system's ability to detect similar defects in the future.
**Example**
Consider an API that unexpectedly returns a 500 Internal Server Error when receiving a malformed request.
The immediate response is to identify the root cause, implement a fix, deploy the correction, and restore normal operation.
A common anti-pattern is to stop at this point.
The defect is fixed, the incident is closed, and the organization moves on without introducing any mechanism to detect the same failure in future releases.
Months later, a seemingly unrelated change reintroduces the defect. Because no automated or repeatable verification exists, the issue reaches production again.
A similar pattern can be observed with authorization defects, protocol handling issues, performance failures, pagination problems, input validation weaknesses, and security-related incidents. The underlying defect may be corrected repeatedly while the organization fails to preserve the knowledge gained from the original failure.
In such environments, the same categories of defects tend to reappear over time despite previous remediation efforts.
**Testing Guidance**
For every significant production defect, testing activities should include:

- Identification of the triggering conditions
- Creation of a reproducible test scenario
- Verification of the implemented fix
- Integration of the test into the regular testing process
- Preservation of the test for future releases

Particular attention should be given to defects involving:

- Authorization failures
- Input validation failures
- Protocol violations
- Performance incidents
- Resource exhaustion conditions
- Security-related defects

- Data integrity issues
- Integration failures

Testing teams should periodically review production incidents to identify recurring defect categories and determine whether appropriate test coverage exists.

The objective is not merely to verify that a defect has been fixed. The objective is to ensure that the same defect cannot silently return.

From a testing perspective, a production defect that is fixed but never transformed into a reusable test remains only partially resolved.

## CONCLUSION

Modern software systems increasingly depend on APIs as their primary mechanism for communication, integration, and data exchange. As a result, API failures often have consequences that extend far beyond individual applications and directly impact reliability, security, scalability, and business operations.

This paper presented ten practical rules for API testing derived from recurring failure patterns observed across real-world systems. The rules are intentionally concise, implementation-agnostic, and applicable to both manual and automated testing approaches.

The objective of these rules is not to replace existing testing methodologies, but to highlight a small number of testing principles that consistently expose high-impact defects. While no set of rules can guarantee software quality, systematic application of these principles can significantly reduce the likelihood of common API failures reaching production environments.

Most API incidents are not caused by unknown problems. They are caused by known problems that were never tested.

[Liudas Jankauskas homepage: qaontime.com]
[More research: qaontime.com/research]

## REFERENCES


[1] G. J. Holzmann, “The Power of 10: Rules for Developing Safety-Critical Code,” *Computer*, vol. 39, no. 6, pp. 95–99, June 2006. doi: https://doi.org/10.1109/MC.2006.212

[2] R. Fielding, M. Nottingham, and J. Reschke, “HTTP Semantics,” STD 97, RFC 9110, Internet Engineering Task Force, June 2022. doi: https://doi.org/10.17487/RFC9110

[3] ISO/IEC/IEEE 29119-1:2022, *Software and Systems Engineering — Software Testing — Part 1: General Concepts*, 2nd ed., International Organization for Standardization, 2022. Available: https://www.iso.org/standard/81291.html

[4] OWASP Foundation, *OWASP API Security Top 10 — 2023*, 2023. Available: https://owasp.org/API-Security/editions/2023/en/0x00-header/

[5] Z. Hatfield-Dodds and D. Dygalo, “Deriving Semantics-Aware Fuzzers from Web API Schemas,” in *Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings*, pp. 345–346, 2022. doi: https://doi.org/10.1145/3510454.3528637

[6] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. ISBN 978-0-321-60191-9.

[7] Postman, *2025 State of the API Report*, 2025. Available: https://www.postman.com/state-of-api/2025/

[8] M. Nottingham and R. Fielding, “Additional HTTP Status Codes,” RFC 6585, April 2012. doi: https://doi.org/10.17487/RFC6585

[9] M. Nottingham, E. Wilde, and S. Dalal, “Problem Details for HTTP APIs,” RFC 9457, July 2023. doi: https://doi.org/10.17487/RFC9457

[10] R. Fielding, M. Nottingham, and J. Reschke, “HTTP Caching,” STD 98, RFC 9111, June 2022. doi: https://doi.org/10.17487/RFC9111