Broken Contracts? Towards Automated Conformance Testing for HTTP Semantics

Matthias Keckl[1][a], Lorenz Gruber[2][b] Peter Müller[1][c], Andreas Tennert[1], Markus Ende[1][d], Peter Braun[3][e] and Nikolas Herbst[2][f]

[1]*qupaya Technologies GmbH, Nürnberg, Germany*

[2]*Department of Computer Science, Julius Maximilian University, Würzburg, Germany*

[3]*Faculty of Computer Science, Technical University of Applied Sciences Würzburg-Schweinfurt, Würzburg, Germany*

{*matthias.keckl, peter, andreas, markus*}*@qupaya.com,* {*lorenz.gruber, nikolas.herbst*}*@uni-wuerzburg.de,*
peter.braun@thws.de

Keywords: HTTP, Conformance Testing, API Testing, Contract Testing.

Abstract: The Hypertext Transfer Protocol (HTTP) serves as the implicit contract governing the semantic interactions of modern web services. Despite the prevalence of explicit interface descriptions like OpenAPI, adherence to the underlying HTTP specifications remains inconsistent. This divergence creates significant barriers for API usage, while simultaneously introducing security vulnerabilities and performance degradation. Existing conformance tools often lack comprehensive coverage, focusing narrowly on specific subsets of the protocol or omitting specific architectural roles. To address this gap, we present a novel framework designed to translate HTTP specifications into executable, architectural roles specific rules. By extracting 390 rules from the HTTP Semantics specification, we demonstrate that entire protocol specifications can be translated into a unified, executable model. By applying it to a microservice reference implementation, we show that semantic HTTP conformance testing can be effectively automated, identifying 225 runtime violations. Furthermore, we utilized the framework to analyze 3.916 publicly available OpenAPI documents, uncovering over 2.8 million violations. Notably, only approximately 1% of the validated documents were completely free of rule violations. These findings reveal a systemic gap between protocol specifications and real-world implementations, underscoring the necessity for automated conformance testing and specialized tooling throughout the API lifecycle.

1 INTRODUCTION

The Hypertext Transfer Protocol (HTTP) serves as the fundamental application protocol of the World Wide Web. Over the last more than 30 years, it has evolved from a simple system designed for document retrieval into the backbone of complex distributed systems and modern web services. To clearly define the interfaces of these HTTP-based services, developers use description formats like OpenAPI[1]. These descriptions establish an explicit contract between the client and the server, including, among others, available resources, parameters, and data schemas. However, we do not see these explicit descriptions as the only contract that governs communication between client and server. The HTTP protocol itself represents an implicit but equally important contract, including its semantics such as the correct use of methods, status codes, and headers. Violations of the protocol are not merely a disregard for best practices. It can lead to serious issues, such as usability problems, security vulnerabilities, or faulty caching mechanisms that degrade performance (Attia et al., 2025; Rautenstrauch and Stock, 2024; Nguyen et al., 2019).

Especially in this age of autonomous AI agents, we see usability as an increasingly significant problem. Whereas humans can deduce the cause of a rejected request from contextual cues, AI agents rely strictly on precise protocol semantics to autonomously resolve client-side errors. For example,

[a] https://orcid.org/0009-0002-6084-1357
[b] https://orcid.org/0009-0004-4343-6864
[c] https://orcid.org/0009-0007-1788-004X
[d] https://orcid.org/0009-0009-4691-3837
[e] https://orcid.org/0000-0002-3164-3063
[f] https://orcid.org/0000-0003-3462-6426

[1] https://spec.openapis.org/oas/

a compliant 429 (Too Many Requests) response explicitly triggers an agent's back-off strategy, whereas a non-conforming 400 (Bad Request) leaves the agent unable to distinguish a rate limit from a syntax error.

Consequently, there is a systematic disconnect in the API tooling landscape. While there are tools for isolated checks, there is a lack of a unified approach that combines the explicit contract of API descriptions with the implicit semantic requirements of the HTTP specification across all architectural roles.

To address this gap, we introduce *Thymian*[2] which is designed to translate HTTP specifications into executable, role-specific rules. *Thymian* enables developers to automatically verify conformance of both API descriptions and actual HTTP implementations. By formalizing these requirements, we aim to bridge the gap between abstract specifications and practical enforcement. We leverage our framework to investigate how the HTTP specifications can be translated into an automated approach to HTTP conformance testing. Furthermore, to evaluate the potential discrepancy between explicit API contracts and the implicit requirements of the HTTP specification, we analyze the current state of publicly available OpenAPI documents.

RQ1: How can HTTP specifications be translated into an executable rule model to verify both API descriptions and their implementations?

RQ2: Can the process of HTTP conformance testing be automated?

RQ3: To what extent do publicly accessible API descriptions contradict the semantic requirements of HTTP?

2 BACKGROUND AND RELATED WORK

This section provides an overview of the required background. In Section 2.1, the history of the HTTP protocol and its current state of specification are briefly summarized. Section 2.2 shortly outlines the OpenAPI format, and Section 2.3 discusses related work in the area of HTTP conformance testing.

2.1 The Implicit Contract: HTTP and Its Specifications

The Hypertext Transfer Protocol (HTTP) is a stateless, client-server application-layer protocol. Since the publication of HTTP/1.0 in 1996 with RFC 1945 (Berners-Lee et al., 1996), the protocol has undergone continuous evolution to meet the demands of the modern web. The current specifications strictly separate the semantics of HTTP (Fielding et al., 2022b; Fielding et al., 2022a) and the specific transport syntax used *on the wire*, such as HTTP/1.1 (Fielding et al., 2022c) or the most recent version *HTTP/3* (Bishop, 2022). This separation ensures that the protocol's semantics remain consistent and are not affected by changes in the underlying HTTP version used for communication.

Communication within HTTP is defined by different roles. Since it is a client/server protocol, there are the roles *client* and *server*. An HTTP client establishes a connection to a server for sending one or more HTTP requests. An HTTP server accepts connections in order to serve requests by sending HTTP responses. These roles are further differentiated. A *user agent* is a more specific client that initiates a request. It could be a web browser, a human, or an autonomous AI agent. *Origin servers* are specific servers that provide authoritative responses for a resource. Furthermore, there are also *intermediaries*, like *proxy*, *gateway*, and *tunnel*. These participants are placed between the user agent and the origin server. These intermediaries can act, among others, as clients, servers, or even as an origin server in specific situations. Another role in this HTTP message processing chain is the *cache*, which is a local storage for previously sent messages.

2.2 The Explicit Contract: OpenAPI

To formalize the interface of HTTP-based services, developers increasingly rely on machine-readable description formats, with OpenAPI being the de facto standard. OpenAPI acts as an explicit contract by providing a structured, declarative specification of an API's capabilities that is intended to be shared between providers and consumers. This contract is *explicit* because it defines the technical boundaries of the interaction, such as resources, data schemas, and interaction expectations **and** applies only to the API described in this document. This stands in contrast to the HTTP protocol, which applies to virtually any API that uses HTTP.

OpenAPI itself is a JSON[3] or YAML[4] based document format that describes the available resources, supported methods, and expected data schemas for requests and responses. Listing 1 shows an exemplary and incomplete OpenAPI document that defines a single GET endpoint. Its only documented response is a

[2]https://github.com/thymianofficial/thymian

[3]https://www.json.org/json-en.html

[4]https://yaml.org/

```yaml
openapi: 3.1.1
paths:
  /courses:
    get:
      responses:
        "200":
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Listing 1: Examplary OpenAPI Document.

200 OK response that includes a JSON-encoded array of strings in the response body.

2.3 Static Analysis of Web APIs and HTTP Conformance

Tools such as Spectral (Stoplight, 2025) lint and analyze API descriptions such as OpenAPI documents against predefined rule sets. Using this concept of statically analyzing OpenAPI documents, the tool *RESTRuler* (Bogner et al., 2024) is specifically designed to identify RESTful API design flaws. By implementing 14 specific design rules, they successfully analyzed over 2300 OpenAPI documents sourced from the *apis.guru* repository. Their study revealed a total of 169,061 rule violations, highlighting significant design problems in public API specifications.

Beyond static description analysis, other tools focus on conformance to the HTTP protocol itself. The test suite from (Rautenstrauch and Stock, 2024) contains rules extracted from various specification documents. Their extraction process relies on strictly defined keywords like *MUST* and *SHOULD*, as discussed in Section 2.1. Each extracted rule is assigned a *level* based on its keyword or whether it defines syntax; rules that define syntax are classified as *ABNF*. Thereby, the test suite only includes rules that are declared with keywords that do not mark the rule as truly optional (e.g., *MAY*). Guided by these keywords, they extracted 106 rules in total. Their empirical analysis of nine web servers and 9900 live web hosts shows that more than half of the rules were violated at least once, with most targets breaking at least one rule.

Based on this work, two further studies have been conducted to extend the scope of HTTP conformance testing to middleboxes. Their research specifically investigates how intermediate network components influence protocol conformance. In the first study (Benhabbour et al., 2025), they tested 12 popular proxy implementations with 47 conformance tests and showed that none of them is fully compliant with the relevant RFC. In a second study (Attia et al., 2025), the authors extend the previous study, which focused on *HTTP/1.1*, to *HTTP/2*. A test suite of 156 tests, 12 proxies, and 3 cloud proxies was evaluated. As in the previous study, none of these implementations is fully compliant with the specification.

Further research has focused on the implementation of caches. In two papers (Nguyen et al., 2019; Nguyen et al., 2018), 397 rules for web caches were extracted and implemented in a web caching testing tool. This tool was then used to investigate 7 distinct shared caching systems (Nguyen et al., 2019). In another work (Nguyen et al., 2018), 6 web browser cache implementations were tested. Both results show that none of the tested implementations were fully RFC-compliant.

While the discussed approaches address isolated aspects of HTTP conformance, a comprehensive solution for verifying HTTP conformance across both static specifications and actual implementations is still missing. Current methodologies lack a unified approach to test the semantic correctness of HTTP interactions for arbitrary components, regardless of their specific role as a cache, proxy, or origin server. *Thymian*, which is presented in detail in the following section, addresses this gap by introducing role-aware so-called *HTTP rules*. These rules provide a dedicated interface that allows the underlying logic to be defined once and applied consistently to both static documents and live HTTP implementations.

3 THYMIAN

We now introduce the *Thymian* framework, focusing on its unified rule representation and the specialized execution contexts used for conformance testing. Therefore, we first define the formal structure of *HTTP rules* and examine how they interface with both static API descriptions and live HTTP implementations. To enable a unified definition of rule logic across these diverse environments, *Thymian* introduces a specialized Domain-Specific Language (DSL) for HTTP. Before the systematic process to extract these rules from official HTTP specification documents is described, this work's approach for request data sampling is discussed.

3.1 HTTP Rules

The central element of our approach is the *HTTP Rule*. Conceptually, an HTTP Rule encapsulates a single, falsifiable constraint derived from the protocol

specification. Each rule is defined by a set of properties: a unique name, a summary, a reference to the source specification, the rule's severity, and the architectural role(s) it applies to. In addition to this metadata, it contains executable verification logic which is defined against an **ApiContext**. This abstraction models the execution environment of the rule and represents a combination of the specific API under test and the available context-specific testing capabilities. By providing access to three distinct contexts, the framework allows each rule to validate the same constraint through static analysis, active testing, and the analysis of recorded traffic and therefore to automate the process of HTTP conformance testing throughout the entire development cycle:

LintContext: This context operates on the static API description formats. It does not work directly on the documents, such as OpenAPI, but on an internal used format into which the documents have been previously converted. It allows the rule to verify the API description's conformance without interacting with a live server.

LiveApiContext: In contrast, the *LiveApiContext* works on actual HTTP implementations and is further split into two contexts, namely, **TestContext** and **AnalyzeContext**. The *TestContext* interfaces with running HTTP servers. It allows rules to actively generate traffic probes, execute requests, and verify the server's runtime behavior. The *AnalyzeContext* provides access to a repository of recorded HTTP traffic. This traffic does not necessarily have to be recorded by *Thymian* itself, but can be provided by any third-party tool. This approach allows for performing post-hoc analysis on real-world usage data to detect violations that occurred during operation.

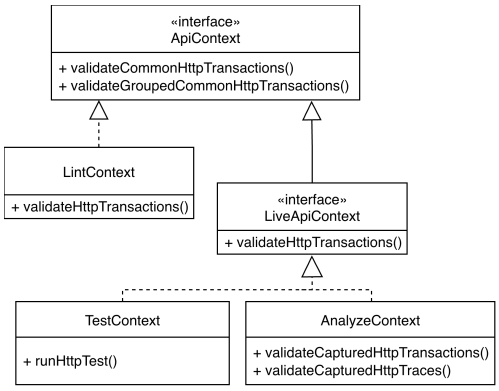

Figure 1: Class diagram of the different rule contexts.

Figure 1 illustrates a class diagram showing the relationship between these execution contexts. As depicted, the *ApiContext* interface provides two core methods, which must be implemented by all concrete implementations. If an HTTP rule is implemented utilizing one of these methods, it can be automatically translated into the three concrete context implementations. For this automated translation to occur, the rule logic must be implemented using a specially designed Domain-Specific Language (DSL), which is explained in detail in Section 3.3. This abstract DSL representation is subsequently utilized and adapted by the three distinct execution contexts to perform their specific conformance checks.

The *LintContext* takes the DSL representation and directly translates it into executable functions that run on the internal *Thymian* format. Because this process is purely static, the translation is straightforward. Furthermore, it provides the fastest user feedback as it does not require sending actual network requests or analyzing large volumes of recorded data.

```
httpRule('rfc9110/server-should-send-
    validator-fields')
 .severity('warn')
 .type('static', 'test', 'analytics')
 .appliesTo('origin server')
 .url('https://...')
 .summary('...')
 .rule((ctx) =>
  ctx.validateCommonHttpTransactions(
   and(
    or(
     method('GET'),
     method('HEAD')
    ),
    statusCode(200)
   ),
   not(
    or(
     responseHeader('etag'),
     responseHeader('last-modified')
    )
   )
  )
 );
```

Listing 2: Declarative definition of an HTTP Rule.

The *TestContext* requires, in addition to the DSL, another novel concept introduced by *Thymian*: API-agnostic tests. For rules to be executed against arbitrary APIs, a mechanism is needed to describe the test logic at a more abstract level. We solved this challenge using a pipeline-based approach. At the start of this pipeline are all HTTP transactions defined within the internal API description format. These transactions can then be filtered and/or grouped before being mapped to specific test cases. These test cases can

be multi-step, containing an arbitrary chain of dependent requests. Finally, they are executed against the live API, and the results are analyzed. To successfully execute these requests, semantically and syntactically valid request data is essential. Since this is not a trivial task and has already been studied in related works (Atlidakis et al., 2019), we will discuss it in more detail in Section 3.4.

Finally, the *AnalyzeContext* takes the DSL representation and translates it into a corresponding SQL query. To facilitate this, the context allows previously recorded HTTP traffic from any third-party source to be provided to *Thymian*. This captured traffic is then structured and written into an SQLite database, enabling the translated queries to efficiently search for historical conformance violations.

3.2 HTTP Rule in Detail

Before taking a closer look at the DSL, we will first explain an exemplary HTTP rule and demonstrate how the three contexts translate the DSL definition. Listing 2 shows an exemplary rule extracted form RFC 9110 (Fielding et al., 2022b). This rule is, just like *Thymian*, implemented in TypeScript and states that an *origin server* **SHOULD** send validator fields in responses with status code *200* to requests with method *GET* or *HEAD*. How such rules are extracted and implemented systematically is given in Section 3.5.

The definition of the rule begins with assigning metadata, including a unique name, a reference to the corresponding section of the specification, and the severity level, here *warn*. Furthermore, the rule explicitly declares its applicability via the *types* method, indicating in this case that it can be verified across all three contexts. A crucial aspect of *Thymian* 's design is the role-based assignment. The rule is tagged specifically for *origin servers*. This ensures that violations are reported only if the target under test is indeed an origin server, avoiding false positives when analyzing intermediaries like proxies or gateways.

The implementation logic itself is encapsulated within the *validateCommonHttpTransactions* method. Using this method signals that the rule is developed against the high-level **ApiContext** interface. This abstraction restricts the rule to access only those transaction properties available in all three contexts, such as method names, status codes, and the presence of header fields, while abstracting away context-specific details like concrete header values, which might not be present in static descriptions.

```
SELECT t.id AS id
FROM http_transaction t
JOIN http_request req
 ON req.id = t.request_id
JOIN http_response res
 ON res.id = t.response_id
WHERE
 (
  req.method = 'get' COLLATE NOCASE
  OR req.method = 'head' COLLATE NOCASE
 )
 AND res.status_code = 200
 AND NOT (
  EXISTS (
   SELECT 1
    FROM http_response_header resHeader
    WHERE
     resHeader.response_id = res.id
    AND resHeader.name = 'etag' COLLATE
        NOCASE
   )
   OR
    EXISTS (
     SELECT 1
     FROM http_response_header
         resHeader
     WHERE
      resHeader.response_id = res.id
     AND
      resHeader.name = 'last-modified'
          COLLATE NOCASE
    )
  )
```

Listing 3: Generated SQL query for the AnalyticsContext.

The *validateCommonHttpTransactions* method accepts two arguments defined via DSL expressions. The first argument acts as a filter, selecting the subset of HTTP transactions to which the rule applies. In the concrete example of Listing 2, all successful transactions with the request method GET or HEAD are filtered. The second argument defines the actual validation logic, specifying the condition that triggers a violation. Or, in the concrete example, all transactions whose responses do not include an *ETag* or *Last-Modified* header.

Internally, *Thymian* translates these abstract expressions into context-specific execution plans. In the **LintContext**, the expressions are compiled into a traversal function that validates the static API description structure for the required definitions. In the **TestContext**, the filter logic is used to identify relevant transactions from the API description. *Thymian* then actively executes these requests against the running service and analyzes the live responses. In the **AnalyzeContext**, the DSL is transpiled into SQL queries. These queries are executed against the repository of recorded traffic to identify matching transactions that

violate the rule. Regardless of the underlying execution mechanism, all three contexts produce a unified format of violation reports. The overall architecture and the data flow of this translation mechanism are visualized in Figure 2.

While generating a validation function for the *LintContext* is relatively straightforward, translating it for the active testing and analytics environments is significantly more complex. For the *TestContext*, the provided filter is translated into a filter function and used to get all relevant transactions from the *Thymian* format. These transactions are subsequently executed using the approaches described earlier. The received responses are then checked with a validation function that is generated based on the DSL expression. In contrast, the *AnalyzeContext* acts as a transpiler rather than iterating over objects. It compiles the declarative DSL expressions directly into SQL queries. This translation enables the analysis of larger amounts of traffic data in the first place. Listing 3 demonstrates the generated SQL query for the rule defined in Listing 2. The DSL filter expression is mapped to the `WHERE` clause to select only relevant transactions, and the validation logic is appended to identify rows that violate the requirement.

3.3 A DSL for HTTP

Since HTTP represents a relatively small and strictly defined domain consisting of finite elements such as methods, paths, headers, parameters, and status codes, it is suited to define a declarative modeling approach for it. *Thymian* introduces a specialized Domain-Specific Language (DSL) for this purpose, allowing developers to describe HTTP requests, responses, and complete transactions without needing to focus directly on concrete implementation details. The DSL is fundamentally JSON-based, but *Thymian* provides TypeScript helper functions to support the structured construction of these expressions. Every expression within the DSL is an *HttpFilterExpression* and is specifically instantiated as either a *RequestFilterExpression*, a *ResponseFilterExpression*, or a *LogicalExpression*.

LogicalExpression: These provide standard logical operators such as *and*, *or*, *not*, *xor*, and *constant*, enabling developers to construct complex chains of conditions.

RequestFilterExpression: *RequestFilterExpressions* address the discrete components of an incoming HTTP message. Developers can use them to query properties such as the request method, origin, or parameters. Furthermore, the DSL supports recursive and context-specific evaluations, such as the *hasResponse* or *isAuthorized* expressions, which enable the analysis of authorization requirements and cross-transaction relationships.

ResponseFilterExpression: These serve as the counterpart for outgoing messages, and target specific parts of the HTTP response, such as the status code or response headers.

An exemplary usage of this DSL is shown in Listing 2. The *validateCommonHttpTransactions* method accepts two DSL expressions. The corresponding JSON representation of the first expression is shown in Listing 4. It shows how multiple operators are combined to describe HTTP requests and responses using two logical operators *and* and *or*.

```
{
 "type": "and",
 "kind": "logic",
 "filters": [
  {
   "type": "or",
   "kind": "logic",
   "filters": [
    {
     "type": "method",
     "kind": "request",
     "method": "get"
    },
    {
     "type": "method",
     "kind": "request",
     "method": "head"
    }
   ]
  },
  {
   "type": "statusCode",
   "code": 200,
   "kind": "response"
  }
 ]
}
```

Listing 4: Examplary HTTP DSL expression.

3.4 Request Data Sampling

To actively verify conformance rules in the *TestContext*, the framework must trigger specific API states. This requires semantically valid request parameters that static descriptions often lack. This is implemented with a two-step deterministic sampling strategy. First, synthetic dummy request data is generated for every unique transaction. As a second step, a user-in-the-loop approach is used. Recognizing that fully

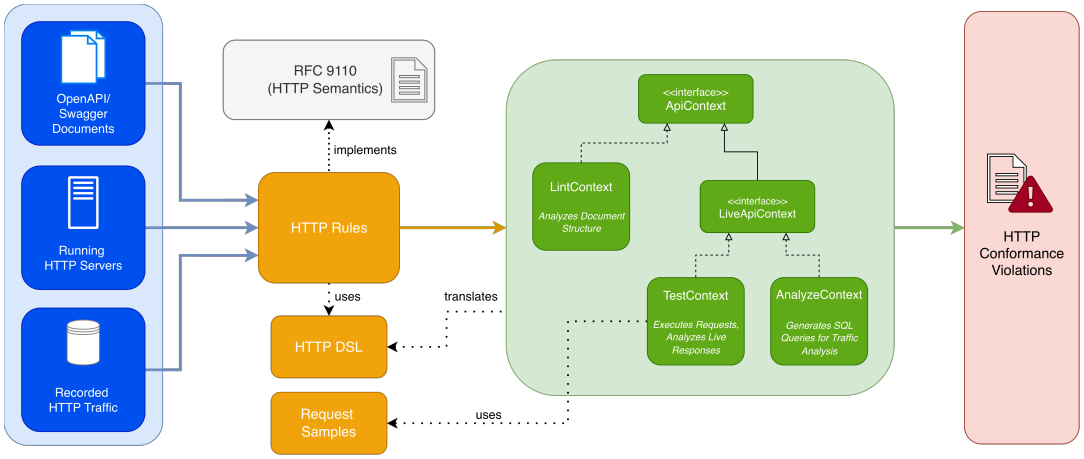

Figure 2: Conceptual Approach of *Thymian*.

automatic generation of request data to trigger all possible states of an API is an open research question (Kim et al., 2022), the framework provides a feedback loop. Users can manually refine the generated samples to ensure they can trigger the specific states. By combining automated structural synthesis with manual semantic refinement, the framework ensures that every documented HTTP transaction can be executed deterministically to verify its protocol conformance.

3.5 Rule Extraction Method

To populate the framework with verifiable rules, we applied a systematic extraction methodology. Our process comprises three stages: *identification*, *classification*, and *context mapping*. For the identification and classification stages, we followed the approach from Rautenstrauch et al. (Rautenstrauch and Stock, 2024) and manually scanned the specification documents for the keywords defined in (Bradner, 1997). Identified rules were translated into *HTTP Rules* and assigned a *severity* and *role(s)*, along with metadata such as a unique name, summary, description, and a URL pointing to the respective specification section. For the severity, we map *MUST*/*MUST NOT* requirements to *ERROR*, as these represent absolute conditions for conformance. *SHOULD*/*SHOULD NOT* requirements are mapped to *WARNING*, indicating strong recommendations. *MAY* or *OPTIONAL* aspects are categorized as *HINT*. For the role, we explicitly tag each rule with the participant role(s) responsible for the requirement. This contrasts with previous work (Rautenstrauch and Stock, 2024), which intentionally omitted rules that only apply to certain communication participants.

The context mapping of our method includes determining **how** a specific rule can be verified. Not every rule is falsifiable in every context. Therefore, for each extracted rule, we performed an analysis to specify its applicability to the *Lint-*, *Test-*, and *AnalyzeContext*.

Rules are mapped to the *LintContext* if their constraints can be sufficiently verified based solely on the interface description. While this context primarily targets *server* and *sender* roles, we also include specific *client* and *user Agent* rules. Including these allows *Thymian* to proactively inform API designers about possible client behaviors that must be accounted for. For example, the rule “For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request” which refers to *302 Found* responses and is defined in RFC 9110 (Fielding et al., 2022b).

The *TestContext* is applied to rules that can be verified via actively generated traffic. Its unique advantage lies in coverage. It allows for the verification of rules even if the functionality is missing from the static description or has not yet occurred in recorded traffic. For example, *Thymian* can actively test if a general-purpose server supports the methods *GET* and *HEAD* as specified in RFC 9110 (Fielding et al., 2022b) with “All general-purpose servers MUST support the methods GET and HEAD.”.

Finally, the *AnalyzeContext* is the only context capable of verifying conformance across **all** architectural roles. Since every transaction recorded in this context preserves the role of the communication participant, rules can be applied dynamically to any specific role. This enables verifying actors that are invisible to static analysis or difficult to isolate in active testing, such as *Intermediaries* or specific *Cache* implementations.

If a rule is not applicable to one of these contexts, it is declared as an *informational* rule. An informa-

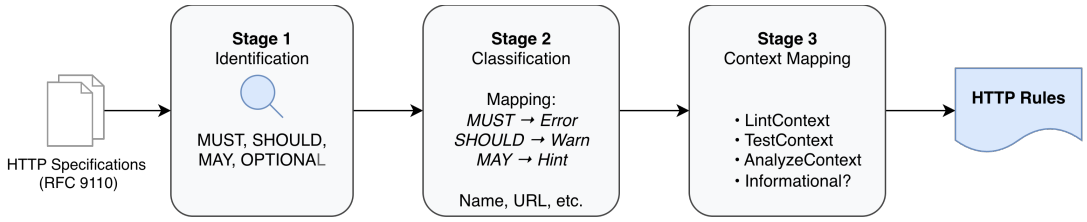

Figure 3: Rule extraction process.

tional rule cannot be verified in any way within the framework, but is still extracted for several reasons. First, for the sake of completeness, so that a specification is fully translated. Furthermore, they are extracted because they may be tested in the future. For example, by implementing a new context or by extending an existing one. The overall extraction process is illustrated in Figure 3.

3.6 Rule Execution

To provide a concise overview of the framework's approach, it can be summarized in the following five steps which are also shown in Figure 2:

1. **Contract Ingestion:** Depending on which context should be applied, the framework ingests OpenAPI documents for linting and testing or recorded HTTP traffic for analysis.
2. **Rule Integration:** These data sources are given to the extracted HTTP rules and are accessible via the corresponding contexts.
3. **Rule Logic Definition:** Each rule defines its semantic constraints using a declarative HTTP DSL against these contexts.
4. **Context-specific Translation:** The DSL is then translated based on the specific context by compiling it into traversal functions for the static checks, transpiling it to SQL for traffic analysis, or utilizing the request samples to actively run test cases.
5. **Unified Reporting:** Regardless of the execution environment, all three contexts generate HTTP conformance violations in a standardized way.

4 MIXED-METHOD STUDY

To evaluate the framework, a mixed-method empirical study was conducted. To do this, the extraction process described in Section 3.5 is first used to extract HTTP rules from RFC 9110 (Fielding et al., 2022b). Following that, a qualitative study is conducted using an origin server implementation to determine whether conformance testing for this API can be automated. Finally, a large-scale quantitative analysis of OpenAPI documents is performed.

4.1 RFC 9110 Rule Set

Applying the extraction methodology of Section 3.5 to RFC 9110, **390** rules were extracted and implemented. To ensure transparency regarding the scope of the analysis, we categorized these rules based on their context suitability and severity. Of the **390** rules, **96** are applicable to the *LintContext*, **58** to the *TestContext* and **155** to the *AnalyzeContext*. The exact classification of the rules and their severities is shown in Table 1. For the following two studies, the rules that can be verified with *Lint-* or *TestContext* are of particular interest.

Table 1: Overview of the extracted rules and their types.

Rule Type	Hint	Warn	Error
Static	20	39	37
Test	11	21	26
Analytics	40	50	65
Informational	63	52	100

4.2 Conformance Testing of TeaStore

To verify if the conformance testing can be automated, a study was conducted using a concrete implementation of an origin server. We selected TeaStore (Von Kistowski et al., 2018) as the reference system for this analysis. TeaStore is a microservice reference and test application for benchmarks and tests. It emulates a basic web shop for tea and tea supplies. While the TeaStore architecture consists of six distinct microservices, we specifically focused our analysis on the persistence service. This service acts as the central data handler for resources such as products and orders, making it of special interest for HTTP conformance.

As *Thymian* relies on an explicit contract to generate valid traffic, and the TeaStore implementation does not provide one by default, we first manually generated an OpenAPI document from the service's source code. Following the application's deployment

instructions in a local environment, *Thymian* was executed against the running persistence service.

4.3 Analyzing Web API Descriptions

For the quantitative study of API descriptions, we utilized the *openapi-directory*[5] from *api.guru*[6]. This repository contains **4138** Swagger and OpenAPI documents and is a widely recognized repository of publicly available definitions, as related work (Bogner et al., 2024) used it as a data source. In the first step, we downloaded the repository from GitHub. Subsequently, *Thymian* was invoked for each document in the repository. This step includes the following substeps. First, the document is loaded from the specified local file path. Since the collection contains OpenAPI documents with different version numbers (*2.0*, *3.0.x*, and *3.1.x*), an upgrade step is required. As a result, each document is upgraded to the *3.1* version of the OpenAPI specification as next step. This is required to enable *Thymian* to transform the document into the internal *Thymian* format. As a last sub-step, the upgraded document is finally dereferenced to check if the document contains any invalid reference. The upgraded OpenAPI document is then transformed into the Thymain format. Based on this format, all applicable rules are applied to the static context. The results of this analysis are saved in a CSV files. The result for each OpenAPI document is separate CSV file containing all rule violations.

5 RESULTS AND DISCUSSION

The following section presents the results of the empirical studies along with a discussion of the research questions. It is shown how the approach of our novel frameworks allows for testing HTTP conformance in an automated manner. The analysis of the API description documents reveals a systematic gap between the HTTP specification and the descriptions of its implementations.

5.1 Role-Aware HTTP Rules

Our first research question *RQ1* addresses the feasibility of translating HTTP specifications into a unified, executable model. For this purpose, our framework introduces the concept of role-aware *HTTP Rules*. These rules encapsulate a single, falsifiable constraint derived from the protocol specification. Through the development of a DSL for HTTP, it is possible to describe the logic of the rule in a declarative way. On the one hand, this simplifies the development of the rules, and at the same time, it allows the rules to be applied in different contexts automatically. They access the API under test in different ways. While the *LintContext* only validates the description of the API, the *TestContext* generate real traffic and validates the responses of the server. The *AnalyzeContext* takes another approach and analyzes given arbitrary traffic. If a rule implements its logic against a common abstraction layer, called *ApiContext*, the DSL representation can be translated into the respective context approaches automatically. By using this unified abstraction, *Thymian* ensures that the semantic interpretation of an HTTP requirement remains consistent, whether it is checked against a static API description file or a live production server.

5.2 Automated Conformance Testing

RQ2 addressed the question of whether testing for HTTP conformance can be automated. To answer that, we must first note that HTTP rules can test for HTTP conformance in different ways depending on the context. While the *Lint-* and *AnalyzeContext* simply need to analyze given data, the *TestContext* must run probe requests for its analysis. So the question of automation focuses on the automation capability of the *TestContext*. For this purpose, a study as described in Section 4.2 was conducted. After setting up, the required request data, all *58* rules that are applicable to the context could be run successfully. It revealed a total of **225** runtime violations, comprising **87** warnings and **138** hints. These finding were produced by **9** distinct rules. So **49** of the **58** test rules did not find any issues. Thereby, **3** rules could not be checked because the origin server implementation was not implemented accordingly.

Although the significance of the number of violations is limited, since only one API was examined, it was nevertheless demonstrated that HTTP compliance testing can be automated based on a description of the API. Once the framework is provided with the necessary request data to execute all relevant API transactions, conformance testing can be automated. While the required request parameters may necessitate manual adjustment to ensure semantic validity, the approach constitutes automated HTTP conformance testing because the systematic verification of protocol constraints is executed entirely without human intervention. The automation focuses on the systematic translation of complex protocol specifications into executable logic that can independently verify ev-

[5]https://github.com/APIs-guru/openapi-directory
[6]https://apis.guru/

ery API transaction across different contexts.

5.3 HTTP Conformance of API Descriptions

Our final research question *RQ3* evaluates to what extent static API description documents comply with the HTTP specification. A violation here means the documented contract does not declare conformance to the requirement, which indicates, but does not prove, an actual runtime violation, unlike the results of the previous section, which were verified against a live server. Thereby, **3916** of the **4,138** Swagger/OpenAPI documents from the *api.guru* repository could be processed. The remaining **222** files were excluded from the analysis because they were not valid Swagger/OpenAPI documents and did not comply with the relevant specification, or because they could not be processed due to internal exceptions.

Across these documents, *Thymian* found **2,874,773** violations in total. Disaggregating these findings by severity revealed that the majority were hints (**1,774,943**), followed by warnings (**750,710**), and errors (**349,120**). Of the **96** implemented lint rules, **29** rules were not violated at least once. In terms of overall conformance, **42** documents had no findings. Only **5** documents do not have at least one rule violation with a severity of *error*. However, these five documents have at least one violation with severity *warn*. While the specification of the *SHOULD* keyword states that there may exist valid reasons to ignore specific items (Bradner, 1997), it also does not mean that "the server is free to ignore a requirement because it doesn't feel like honoring it" (Nottingham, 2018). Meaning the chance is high that an implementation is not HTTP-compliant if it emits any violation with severity level *warn*.

The most violated rule with **320,170** violations is "rfc9110/origin-server-with-clock-must-generate-date-for-2xx-3xx-4xx" which simply states that an origin server MUST generate a *Date* header for specific responses insofar as the origin server has a clock. While the *Date* header is crucial for caches to calculate the freshness of cache entries, it is added automatically by most frameworks used to implement origin servers. Moreover, because the *Date* header belongs to the protocol layer rather than to an API's documented semantics, providers have little reason to declare it in an OpenAPI document even when their running servers emit it correctly. That is why other rules are now being considered. That is why other rules are now being considered.

A significant part of the remaining findings, specifically related to the rules "rfc9110/server-should-send-validator-fields", "rfc9110/origin-server-should-send-etag", and "rfc9110/origin-server-should-send-last-modified", concern the absence of validator fields. For these rule, RFC 9110 states "In 200 responses to GET or HEAD, an origin server SHOULD send any available validator fields (Section 8.8) for the selected representation, with both a strong entity tag and a Last-Modified date being preferred." (Fielding et al., 2022b) The specification also defines what validator fields are and the use cases they fulfill. Validator fields can be used as preconditions for conditional requests. Such requests include one or more request header fields that indicate a precondition. These preconditions are tested before the corresponding request method is applied to the target resource. One use case of conditional requests is conditional GET requests. As the specification states, this is the most efficient mechanism for HTTP cache updates. Another use case mentioned in the specification is the application of this mechanism to state-changing methods, such as PUT or DELETE. This can prevent so-called "lost update" problems, in which a client accidentally overwrites another client's content. While the rule only requires sending **any** validator field, it states that strong *entity tag* and *Last-Modified* date are preferred. For this reason, the corresponding HTTP rule in *Thymian* does only check for these two header fields. Non-conformance to this rule and failure to send validator fields mean that the most efficient mechanism for cache updates cannot be used, and the "lost update" problem cannot be prevented.

Another frequently violated rule, with **29,788** violations, is "rfc9110/server-should-send-error-representation-for-4xx-responses". Responses with a *4xx* status code indicate that the client has erred. The specification states on this that "Except when responding to a HEAD request, the server SHOULD send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition." (Fielding et al., 2022b). If the client does not receive any information about the issue or how to fix it, human interaction is required to resolve the error. That this is a fundamental problem is also demonstrated by specification RFC 9457 (Nottingham et al., 2023), which defines a JSON-based problem format for HTTP APIs that carries machine-readable details of the error(s).

Overall, the results of the static HTTP conformance analysis reveal a large gap between the HTTP specification and the descriptions of its implementations. Only **42** or **1.01%** of the successfully validated documents had no violations that could lead to non-HTTP-compliant implementation. Also, the

1,774,943 hints found in the document reveal another issue. Because features, declared with keywords like *MAY* or *OPTIONAL*, are, as specified, truly optional, clients have no way to know whether they are supported by the current origin server implementation if they are not documented in the API description. This means that there are two benefits to static analysis of descriptive documents for HTTP conformance. First, it reveals potential performance and usability issues, informs developers about optional HTTP features, and points out that these should be documented in the API description.

6 THREATS TO VALIDITY

In the following section, the threats to validity (Wohlin et al., 2012) are discussed.

Conclusion Validity. A primary threat is the potential for bugs and errors within the *Thymian* framework itself. If the HTTP rules are implemented incorrectly, the reported violation counts would be inaccurate. To minimize this risk, as many rules as possible were implemented using the DSL for HTTP, as described in Section 3.1. By working at a higher level of abstraction, fewer development errors can be made, as the rules do not have to be implemented manually, but only need to be described via the DSL. Another threat is the quality of the data set. For example, if the set of OpenAPI documents intentionally consists solely of documents with non-HTTP-compliant descriptions, this considerably reduces the significance of the results. To mitigate this potential issue, a well-maintained repository of OpenAPI documents was chosen, which has already been used in other works (Bogner et al., 2024).

Internal Validity. The extraction of rules from RFC 9110 involves translating natural language text into executable code. There is a risk that ambiguous phrasing in the standard was misinterpreted during implementation, leading to results that reflect the authors' bias rather than the protocol's intent. We mitigated this threat by adhering strictly to the normative keywords defined in RFC 2119 (Bradner, 1997). Only requirements explicitly marked with these keywords were translated into rules. The only exceptions to this are rules for status codes marked as deprecated or reserved. However, these rules are very simple and allow little scope for interpretation. Another mitigation is the usage of the HTTP DSL. This DSL allows only existing aspects of the specification to be referenced and described, leaving less room for interpretation.

Construct Validity. We measure conformance using a set of HTTP rules. However, the HTTP specifications are far more extensive. A system that passes all *Thymian* checks is not necessarily fully compliant with RFC 9110 or HTTP at all, but only with the subset implemented. Therefore, we explicitly frame our results as violations rather than as correctness certificates. The detected 2,874,773 violations are definitive evidence of non-conformance, whereas the absence of violations is treated only as a lack of evidence for specific defects, not as proof of total validity. In the TeaStore analysis, *Thymian* generates synthetic traffic based on the OpenAPI definition. This traffic might not reflect the complexity and sequence of real-world user behavior, potentially missing edge cases that occur only under specific usage patterns. We acknowledge that *Thymian*'s focus is not on simulating realistic user data. The real-world usage of the service can therefore differ significantly from the tests conducted of *Thymian*. However, at the protocol level, it makes no difference what content is actually transmitted in a request/response; only the semantics matter, carried by the request method and the response status code, among other things.

External Validity. A potential threat of our findings is the use of the TeaStore microservice as our primary subject for the qualitative study. The TeaStore is a microservice benchmark, not a production system operating at scale. However, our objective in Section 4.2 was not a definitive assessment of microservice conformance in general, but rather a feasibility study of *Thymian*'s automated testing pipeline.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented a novel framework that translates HTTP specifications into executable *HTTP rules* for verifying protocol conformance. A key feature of its approach is the capability to explicitly define rules for specific roles within the HTTP communication model. These rules enable an automated verification process covering HTTP API descriptions, running HTTP implementations, and arbitrary recorded HTTP traffic.

To validate our approach, we extracted 390 rules from RFC 9110 (Fielding et al., 2022b) and conducted a mixed-method study. By applying the framework's approach to the Teastore microservice, we showed that HTTP conformance testing can be automated by uncovering 225 rule violations. Further, we successfully analyzed 3,916 OpenAPI documents, identifying a total of 2,874,773 rule violations and revealed a large gap between the HTTP specification and the descriptions of its implementations.

In future work, we will expand the rule set to cover additional HTTP specification such as RFC 9111 (Fielding et al., 2022a). Additionally, we plan to leverage a large-scale qualitative study to examine the HTTP conformance of HTTP implementations. The tool allows for systematic testing not only of origin servers but also of proxy, gateway, and cache implementations.

ACKNOWLEDGEMENTS

This research was mainly funded by qupaya technologies GmbH.

REFERENCES

Atlidakis, V., Godefroid, P., and Polishchuk, M. (2019). RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, Montreal, QC, Canada. IEEE.

Attia, M., Benhabbour, I., and Dacier, M. (2025). The Developer, the RFC, and the Middlebox: An HTTP/2 Compliance Story. In *Proceedings of the 2025 ACM Internet Measurement Conference*, pages 258–273, Madison WI USA. ACM.

Benhabbour, I., Attia, M., and Dacier, M. (2025). HTTP Conformance vs. Middleboxes: Identifying Where the Rules Actually Break Down. In Testart, C., Van Rijswijk-Deij, R., and Stiller, B., editors, *Passive and Active Measurement*, volume 15567, pages 155–181, Cham. Springer Nature Switzerland. Series Title: Lecture Notes in Computer Science.

Berners-Lee, T., Fielding, R., and Frystyk, H. (1996). Hypertext Transfer Protocol – HTTP/1.0. Technical Report RFC1945, RFC Editor.

Bishop, M. (2022). HTTP/3. Technical Report RFC9114, RFC Editor.

Bogner, J., Kotstein, S., Abajirov, D., Ernst, T., and Merkel, M. (2024). RESTRuler: Towards Automatically Identifying Violations of RESTful Design Rules in Web APIs. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pages 123–134, Hyderabad, India. IEEE.

Bradner, S. (1997). Key words for use in RFCs to Indicate Requirement Levels. Technical Report RFC2119, RFC Editor.

Fielding, R., Nottingham, M., and Reschke, J. (2022a). HTTP Caching. Technical Report RFC9111, RFC Editor.

Fielding, R., Nottingham, M., and Reschke, J. (2022b). HTTP Semantics. Technical Report RFC9110, RFC Editor.

Fielding, R., Nottingham, M., and Reschke, J. (2022c). HTTP/1.1. Technical Report RFC9112, RFC Editor.

Kim, M., Xin, Q., Sinha, S., and Orso, A. (2022). Automated test generation for REST APIs: no time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 289–301, Virtual South Korea. ACM.

Nguyen, H. V., Iacono, L. L., and Federrath, H. (2019). Mind the cache: large-scale explorative study of web caching. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 2497–2506, Limassol Cyprus. ACM.

Nguyen, H. V., Lo Iacono, L., and Federrath, H. (2018). Systematic Analysis of Web Browser Caches. In *Proceedings of the 2nd International Conference on Web Studies*, pages 64–71, Paris France. ACM.

Nottingham, M. (2018). How to Read an RFC.

Nottingham, M., Wilde, E., and Dalal, S. (2023). Problem Details for HTTP APIs. Technical Report RFC9457, RFC Editor.

Rautenstrauch, J. and Stock, B. (2024). Who's Breaking the Rules? Studying Conformance to the HTTP Specifications and its Security Impact. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 843–855, Singapore Singapore. ACM.

Stoplight (2025). stoplightio/spectral. original-date: 2018-08-15T18:43:18Z.

Von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., and Kounev, S. (2018). TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236, Milwaukee, WI. IEEE.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg.