KushoAI

AI Agent Benchmark: API Bug Detection

A black-box evaluation of how AI-generated tests find functional bugs in live APIs.

20	7	97	7	5
Live API scenarios	Application domains	Planted functional bugs	Agents and models	Workflow modes

Evaluated using APIEval-20 v1.0, an open benchmark contributed by KushoAI. Scoring is execution-based: a generated test either triggers a planted functional bug in the live reference API or it does not. All reported scores are the mean of five independent runs per scenario.

Executive Summary

AI coding tools can generate API tests quickly. The harder question is whether those tests find bugs.

This report evaluates that question across 20 live API scenarios spanning seven application domains, with 97 planted functional bugs across three complexity tiers. Each system receives only a JSON schema and one valid sample payload, then must generate API test cases that expose failures in a live reference API. No source code. No documentation beyond the schema. No hints about where failures are planted.

The evaluation uses APIEval-20 v1.0, a black-box benchmark contributed by KushoAI. Because KushoAI is also one of the evaluated systems, this report includes the methodology, workflow definitions, repeated-run setup, and robustness checks so readers can understand where the performance difference comes from.

Seven systems were compared across three groups: general-purpose LLMs, coding agents, and KushoAI. The report also compares workflow modes engineering teams commonly try in practice: one-shot prompting, structured test-strategy prompting, prompt chaining, native coding-agent workflows, and native API test generation.

- **Simple structural bugs are no longer a meaningful differentiator.** Most systems can generate missing-field, null, empty-array, and wrong-type tests. These tests are useful, but they are also the easiest class of failures to discover from the schema alone.

- **Prompt engineering improves parameter coverage, formatting, and field-level negative tests.** It makes suites broader, more explicit, and easier to parse, but it does not consistently make general coding tools reason about cross-field business states.
- **The gap opens on complex bugs.** KushoAI detects 76% of complex planted bugs in this evaluation, compared with 53% for the strongest coding-agent workflow and 34% for the strongest general-purpose LLM.

KushoAI ranks first on the primary score and across all bug complexity tiers. The largest margin appears on the metric most tied to production risk: cross-field and business-logic bug detection.

Key Findings

1. Plausible-looking test suites can still miss bugs.

Several systems generated suites with readable names, valid payloads, and broad field coverage. The difference appeared only after running those suites against live APIs with known planted failures.

2. Simple schema-level tests are now table stakes.

Most systems can generate tests for missing fields, null values, empty arrays, and wrong types. These tests are useful, but they are not enough to evaluate whether a tool can find production-relevant failures.

3. Prompting helps breadth more than depth.

Structured prompts improved parameter coverage, JSON validity, and field-level negative tests. They did not consistently produce cross-field business-logic tests.

4. Complex bugs separate field mutation from API test design.

The hardest bugs required combining individually valid fields into invalid states, such as invalid refund state, role hierarchy violations, conflicting recurrence rules, or notification channels enabled before verification.

5. Test composition matters more than test volume.

Coding-agent workflows often generated many tests. The gap came from whether those tests explored meaningful field interactions.

6. Consistency matters for CI/CD adoption.

KushoAI showed the lowest run-to-run variance among all evaluated systems. For teams integrating generated tests into automated pipelines, output stability matters as much as peak performance.

1 Why API Bug Detection Needs a Different Evaluation

Most API test generation comparisons ask whether a tool can produce tests. That is too low a bar. Any current LLM can generate a list of plausible tests from an API schema.

The test names may sound comprehensive, and the payloads may be syntactically valid, but that does not tell an engineering team whether the suite actually reduces risk. Traditional coding benchmarks usually measure properties like code correctness, task completion, or whether generated tests execute. API testing has a different core objective: finding behavior that violates the intended contract of a live service.

A more useful evaluation question is narrower: Given only the request schema and one valid sample payload, with no source code, no documentation beyond the schema, and no hints about planted failures, can an AI system generate tests that trigger planted functional bugs in a live API?

That is the task evaluated in this report. It evaluates end-to-end behavior: the agent reads the schema and sample, constructs a test suite, the suite is executed against live reference implementations, and scoring is determined by which planted bugs are triggered.

This black-box constraint reflects a common practitioner reality. Teams often receive an OpenAPI schema or request payload examples before they have complete documentation, test data, or implementation context. In that setting, a useful testing agent has to infer likely constraints from field names, data types, descriptions, nested structure, and the operation being performed.

The benchmark contains 20 scenarios across e-commerce, payments, authentication, user management, scheduling, notifications, and search/filtering. Across those scenarios, it contains 97 planted functional bugs: 28 simple, 35 moderate, and 34 complex.

The benchmark does not try to reproduce every production condition. It isolates one capability that matters in production: whether an AI system can generate high-signal API tests from limited request-shape context. That makes the comparison controlled, repeatable, and easier to inspect.

2 Methodology

Every system received the same two inputs per scenario: a JSON schema and one valid sample payload. No implementation code, response schema, logs, changelog, production examples, or planted-bug hints were provided.

Each system had to produce a JSON array of test cases. Each case included a `test_name` and a complete request `payload`. No expected outcomes were required; the evaluator determines whether a test triggers a planted bug by running it against the live API.

The schema and sample payload together represent the minimum useful context a tester might have. The schema tells the agent what fields exist and what constraints are explicit. The sample payload shows how the API is normally used. The benchmark intentionally withholds everything else so that systems cannot rely on implementation leakage or hand-written documentation that points directly at the failure modes.

This keeps the task focused on test generation rather than assertion writing. A system is not rewarded for writing a confident expected outcome unless the request payload actually reaches a planted bug.

Category	Systems	How they were used
General-purpose LLMs	GPT-5, Claude Sonnet 4.6, Gemini 2.5 Pro	API/chat mode with a structured JSON-output prompt.
Coding agents	Claude Code, Cursor, GitHub Copilot	Native agentic workflow with schema files and prompt instructions.
API testing agent	**KushoAI**	Native API test generation workflow.

The systems evaluated here update frequently. Results should be read as a point-in-time evaluation of the specific models, product modes, prompts, and workflows used during this study.

Workflow Modes Compared

The workflow comparison is included because teams rarely stop after a single prompt. In practice, engineers try a one-shot prompt, then make the prompt more explicit, then ask the tool to review its own gaps, then build local scripts around the process.

Workflow mode	Description	Systems included
One-shot prompt	Generate tests from the schema and sample payload in a single pass.	General LLMs and coding-agent baseline runs
Structured strategy prompt	Adds explicit instructions for required fields, invalid types, formats, enums, boundaries, and negative cases.	General LLMs and coding agents
Per-scenario prompt chain	One prompt to infer the strategy, one to generate tests, one to review gaps, and one to emit final JSON.	Coding agents
Native coding-agent workflow	Agent reads local scenario files, writes suites to disk, and revises after format validation.	Claude Code, Cursor, Copilot
KushoAI native workflow	Purpose-built API testing generation with internal field analysis and cross-field candidate construction.	**KushoAI**

For each non-KushoAI system, the main leaderboard reports the strongest workflow observed across the tested modes. This gives general LLMs and coding agents the benefit of structured prompting and iteration rather than comparing KushoAI only against one-shot outputs.

Bug Complexity Tiers

Tier	Definition	Examples
Simple	No semantic domain understanding required.	Missing required field, `null`, wrong type, empty array.
Moderate	Requires understanding field meaning or documented constraints.	Invalid currency code, malformed email, out-of-range rating, invalid enum.
Complex	Requires reasoning about relationships between fields or operation semantics.	Mutually exclusive fields, refund amount greater than original transaction, date range where end precedes start.

Scoring Formula

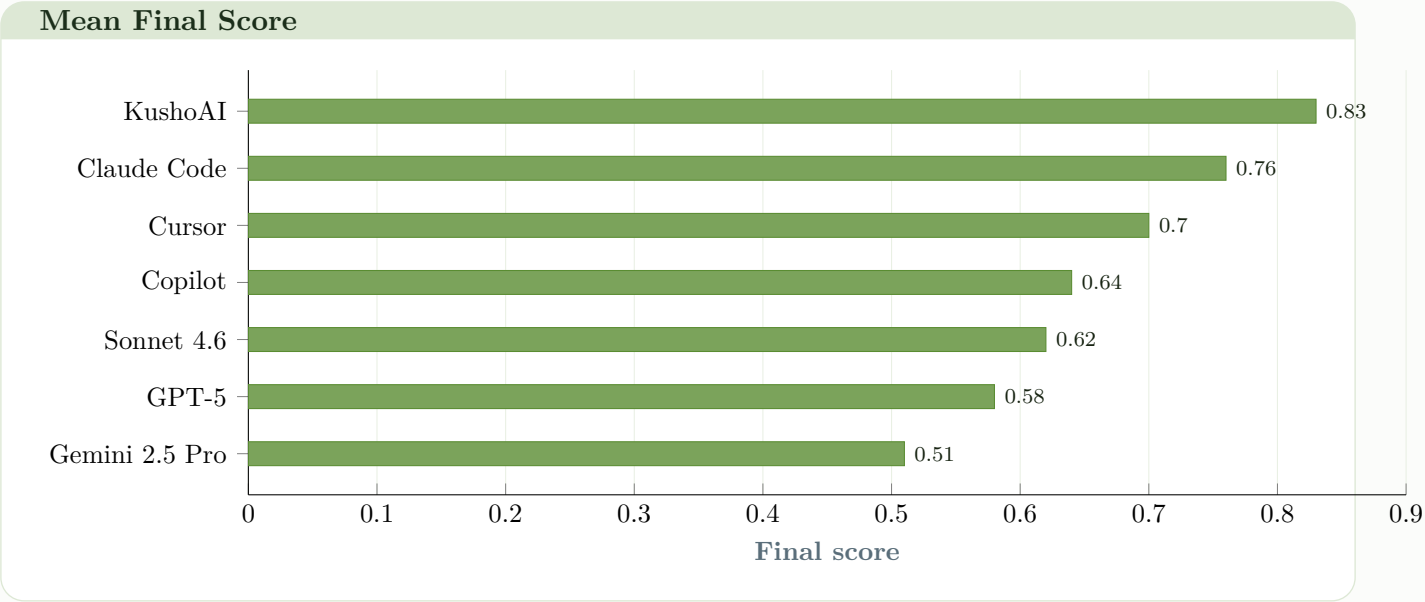
```
Final Score =
  0.70 x Bug Detection Rate
+ 0.20 x Coverage Score
+ 0.10 x Efficiency Score

Bug Detection Rate = bugs_triggered / total_planted_bugs
Coverage Score = param_coverage
Efficiency Score = min(1, bugs_found / number_of_tests)
```

Bug detection is weighted most heavily because tests that do not find bugs have limited engineering value, even if they look broad. Coverage rewards suites that exercise each top-level schema field at least once. It is intentionally simple and should not be read as a proxy for edge-case depth, business-logic coverage, or bug-finding quality. Efficiency penalizes suites that bury a few useful cases inside a large amount of redundant noise.

3 Overall Results

Rank	System	Category	Best workflow	Bug detect rate	Coverage	Efficiency	Final score	Std dev across runs
1	**KushoAI**	API testing agent	Native KushoAI	0.89	1.00	0.14	0.83	+/-0.03
2	Claude Code	Coding agent	Prompt chain	0.76	0.98	0.18	0.76	+/-0.05
3	Cursor	Coding agent	Prompt chain	0.70	0.95	0.16	0.70	+/-0.07
4	GitHub Copilot	Coding agent	Structured prompt	0.64	0.92	0.14	0.64	+/-0.08
5	Claude Sonnet 4.6	General LLM	Structured prompt	0.60	0.90	0.20	0.62	+/-0.09
6	GPT-5	General LLM	Structured prompt	0.56	0.88	0.18	0.58	+/-0.08
7	Gemini 2.5 Pro	General LLM	Structured prompt	0.49	0.82	0.17	0.51	+/-0.10



Coverage is near-saturated across the leading systems, so the leaderboard should not be read as a coverage story. In this report, Coverage measures whether generated suites exercise top-level schema fields. It does not measure edge-case depth, cross-field reasoning, or business-logic coverage. The separation comes from bug detection, complex-bug detection, and run-to-run consistency. KushoAI has the highest bug detection rate, the strongest complex-bug rate, and the lowest standard deviation across runs.

KushoAI achieved full Coverage (1.00) across all 20 scenarios, meaning its generated suites exercised every top-level schema field in every scenario. This reflects the native workflow's schema traversal approach rather than selective field targeting.

One metric worth contextualizing is Efficiency, defined here as the ratio of bugs found to tests generated. KushoAI's Efficiency score (0.14) reflects that its native workflow generates more tests per scenario than general LLMs, which increases overall exploration but lowers the bugs-per-test ratio. Because the final score weights bug detection at 0.70, this tradeoff is intentional: finding more bugs with more tests is preferable to finding fewer bugs with fewer tests. Teams optimizing for CI runtime can apply deduplication or suite trimming after the initial generation pass.

Coverage and bug detection diverge. A model can touch many fields and still miss the failure. For example, a suite may test currency with an empty string and amount with zero, but never test the combination where each field is individually valid and the overall payment state is invalid. The evaluator rewards the latter because that is what reveals the planted bug.

Coding agents outperform raw chat models because they handle local files, format correction, and iterative generation better. Their scores reflect real workflow advantages. The remaining gap is between general software engineering agents and a system built specifically for API bug detection.

For engineering teams, consistency matters as much as peak score. A tool that produces a strong suite in one run and a weak suite in the next creates review overhead. Lower variance means fewer manual retries and a more reliable path into CI.

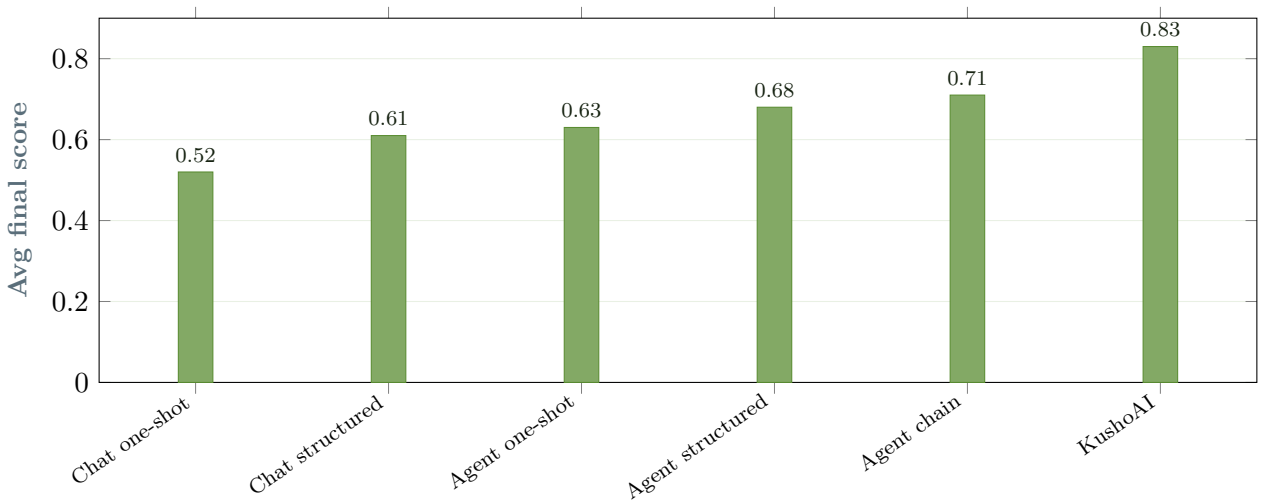
4 Prompting Helps, But It Does Not Close the Gap

The practical question for engineering teams is whether better prompting can make general AI coding tools competitive with a purpose-built API testing agent. It helps. It does not close the gap.

This section matters because "just write a better prompt" is the default reaction to weak AI-generated tests. Better prompts do improve output. They make the model enumerate more fields, include more boundary values, and return cleaner JSON. But in this benchmark, the main weakness is not lack of instruction-following. It is the ability to infer meaningful invalid states from the shape of the API.

Workflow	Avg bug detection rate	Avg coverage	Avg final score	Human setup and review
Chat LLM, one-shot	0.48	0.82	0.52	5-10 min per scenario, manual copy-paste
Chat LLM, structured prompt	0.58	0.90	0.61	15-25 min per scenario, manual cleanup
Coding agent, one-shot	0.62	0.89	0.63	10-15 min per scenario
Coding agent, structured prompt	0.68	0.93	0.68	20-35 min per scenario
Coding agent, prompt chain	0.71	0.95	0.71	35-50 min per scenario
KushoAI native	0.89	1.00	0.83	**Single upload/run**

Workflow Comparison: Average Final Score



Human setup and review estimates are based on observed evaluation workflow time and exclude API execution time. They should be read as directional, since review time varies by team, endpoint complexity, and acceptance criteria.

Structured prompting increases coverage and produces more required-field tests, invalid-type tests, basic boundary tests, and format tests for emails, currency codes, phone numbers, dates, and enums. Prompt chaining adds another improvement because the agent can first infer a test strategy, then generate tests, then review its own gaps.

The remaining gap is concentrated in complex tests: fields that are individually valid but invalid together, optional fields whose validity changes depending on another field, and business states that require combining arrays, nested objects, and operation semantics.

In the coding-agent workflow, the prompt chain generally produced the best non-Kusho result. The first pass identified fields and likely risk areas. The second generated tests. The third asked the agent to find missing categories such as boundary values and nested object combinations. This improved coverage, but it also increased human setup time and review effort.

The reason the gap remains is visible in the generated suites. Prompted coding agents become more exhaustive along a field-by-field axis: more missing-field tests, null tests, wrong-type tests, and boundary tests. Those are useful, but they are still mostly independent mutations. The harder API bugs live in relationships: start and end time, role and permission, refund and original transaction, gift flag and price visibility, channel activation and verification state.

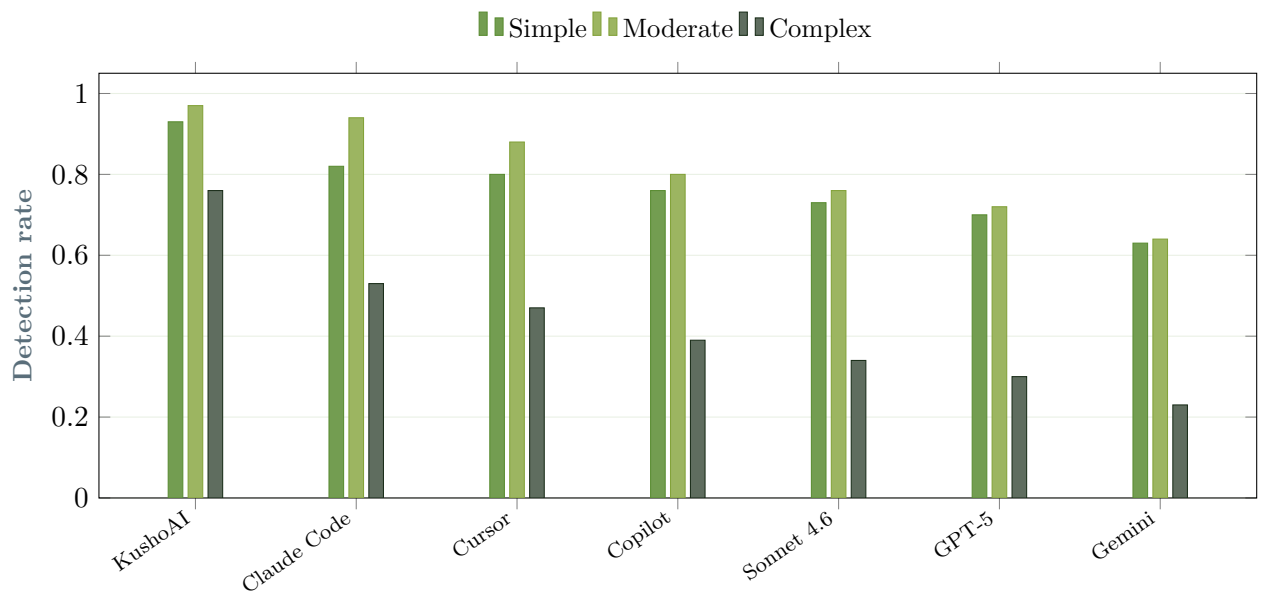
5 The Complexity Cliff

The overall leaderboard shows who wins, but the complexity split shows why. Simple, moderate, and complex bugs represent different kinds of test design. Simple bugs come from schema mutation. Moderate bugs require field-level semantics. Complex bugs require field interaction.

The simple-to-complex drop shows how much performance holds up when the task moves from basic schema-level negative testing to cross-field and business-logic reasoning. This is the important comparison. A system can perform well on simple and moderate bugs and still miss the bugs that emerge only when valid fields interact in invalid ways.

System	Simple	Moderate	Complex	Simple-to-complex drop
KushoAI	0.93	0.97	0.76	0.16
Claude Code	0.82	0.94	0.53	0.29
Cursor	0.80	0.88	0.47	0.33
GitHub Copilot	0.76	0.80	0.39	0.37
Claude Sonnet 4.6	0.73	0.76	0.34	0.39
GPT-5	0.70	0.72	0.30	0.40
Gemini 2.5 Pro	0.63	0.64	0.23	0.40

Bug Detection by Complexity Tier



Key observation: KushoAI has the smallest simple-to-complex drop in the evaluation. Its detection rate moves from 0.93 on simple bugs to 0.76 on complex bugs, a 0.16 drop. The strongest coding-agent workflow drops from 0.82 to 0.53, while general-purpose LLMs drop more sharply. The gap is not in basic schema traversal. It is in maintaining bug-finding performance when tests require cross-field reasoning.

Simple bugs are schema-mutation tests: omit required fields, send `null`, use the wrong type, empty an array. The weakest system still detects 63% of simple bugs, which is why this tier should not be overinterpreted. Basic schema traversal is now table stakes.

Moderate bugs require the agent to understand what a field means. A currency string is not just any string. A rating has a valid range. A locale should look like a locale. A `page_size` field may have a documented maximum. Structured prompts and coding agents perform relatively well here because they can be instructed to look for formats, enums, boundaries, and common standards.

This is why moderate scores are sometimes higher than simple scores in the table. Moderate bugs often have visible cues in the schema or field descriptions. When a field says currency, email, locale, enum, date, or `page_size`, a well-prompted model can generate useful tests by applying known conventions. That is a real capability, but it is still mostly field-level reasoning.

Complex bugs require combining fields. This is where most systems still fall sharply. Four of seven systems detect fewer than 40% of complex bugs in this benchmark. The strongest coding-agent workflow, Claude Code with prompt chaining, detects 53%. KushoAI detects 76%. The 23-point gap between the best coding-agent workflow and KushoAI is the central result of this evaluation.

This is the production-relevant cliff. The payload is well-formed. Required fields are present. Types are correct. The failure appears because the combination is invalid: a refund exceeds the captured amount, a recurring event rule conflicts with an exception date, a role assignment violates hierarchy, or a notification channel is enabled before verification has completed.

KushoAI's smaller simple-to-complex drop suggests that its native workflow is not only finding schema-level and field-level issues. It is also preserving more of its bug-finding ability when the test needs to reason about relationships between fields. That is the capability API testing agents should be evaluated on.

6 Example: When Valid Fields Create an Invalid State

The `PUT /api/v1/users/{user_id}/notification-preferences` scenario contains nested channel settings for email, SMS, and push, plus frequency, category preferences, quiet hours, digest scheduling, and localization fields. It has five planted bugs: one simple, two moderate, and two complex.

This scenario is useful because the request is not a flat validation problem. Many fields are individually valid booleans or strings, but their meaning changes when they are combined. One complex bug involves enabling SMS notifications while the SMS channel is still unverified.

Simple structural test

```
{
  "test_name": "invalid channel enabled flag type",
  "payload": {
    "user_id": "usr_4821",
    "channels": {
      "email": { "enabled": "yes", "verified": true, "address": "alice@example.com" },
      "sms": { "enabled": false, "verified": false, "address": "+14155550101" },
      "push": { "enabled": true, "verified": true, "address": "device_token_abc123" }
    },
    "frequency": "realtime",
    "categories": { "transactional": true, "security": true }
  }
}
```

This catches a structural validation issue: `channels.email.enabled` is supposed to be a boolean, not a truthy string. It is useful, but it still changes one field in isolation.

Moderate field-semantics test

```
{
  "test_name": "invalid notification frequency",
  "payload": {
    "user_id": "usr_4821",
    "channels": {
      "email": { "enabled": true, "verified": true, "address": "alice@example.com" },
      "sms": { "enabled": false, "verified": false, "address": "+14155550101" },
      "push": { "enabled": true, "verified": true, "address": "device_token_abc123" }
    },
    "frequency": "hourly",
    "categories": { "transactional": true, "security": true }
  }
}
```

This requires understanding the allowed frequency values. The payload is structurally valid JSON, but the semantic value is outside the supported notification cadence.

Complex cross-field test

```
{
  "test_name": "sms enabled while sms channel is unverified",
  "payload": {
    "user_id": "usr_4821",
    "channels": {
      "email": { "enabled": true, "verified": true, "address": "alice@example.com" },
      "sms": { "enabled": true, "verified": false, "address": "+14155550101" },
      "push": { "enabled": true, "verified": true, "address": "device_token_abc123" }
    },
    "frequency": "realtime",
    "categories": {
      "marketing": false,
      "transactional": true,
      "security": true
    },
    "quiet_hours": { "enabled": false, "start": "22:00", "end": "07:00" },
    "language": "en-US"
  }
}
```

Every individual value is plausible. The invalid state emerges from the relationship between channel activation and verification state. If SMS is enabled while the SMS channel is unverified, the API should reject the configuration or force verification before accepting it. In the reported run, KushoAI generated this test and Claude Code did not.

This is the kind of case that separates field mutation from API test design. The test is not created by making a field empty or assigning the wrong type. It is created by noticing that one valid setting should be conditional on another valid setting.

This example illustrates why field count is not a proxy for test quality. The notification preferences endpoint has fewer than 15 distinct fields. The two complex bugs are invisible to a field-by-field mutation strategy and only detectable to a system that models conditional relationships between fields.

7 Implications for Engineering Teams

For teams using chat LLMs

Chat LLMs are a reasonable starting point for simple structural checks. They are not enough for API testing coverage that needs to catch business-logic failures. Better prompts improve breadth, but the workflow remains manual and output quality varies across runs.

The practical limitation is not only quality; it is workflow. Chat interfaces require manual copy-paste, manual file creation, and manual cleanup when JSON is malformed or when a suite mixes explanation with data. For a single endpoint this may be acceptable. Across 20 scenarios, or across a production spec with hundreds of endpoints, the friction compounds quickly.

For teams using coding agents

Coding agents are meaningfully better than chat LLMs because they can operate on files, fix invalid JSON, and run iterative passes. With prompt chains, they can produce useful internal API testing suites.

The tradeoff is ownership. Teams must maintain scenario splitting, prompt templates, validation, retries, deduplication, review, and reruns when the API changes. This can be a reasonable path for teams that want to build their own internal testing workflow, but the engineering team owns the surrounding system.

For teams evaluating API testing agents

Do not evaluate on a simple CRUD endpoint. Evaluate on an endpoint where correctness depends on combinations: payments, refunds, subscriptions, permissions, scheduling, inventory, onboarding, or state transitions.

The inspection question is direct: does the generated suite only mutate fields independently, or does it construct invalid business states from individually valid fields? A suite that only does the former will miss the class of bugs most likely to reach production.

A useful vendor or internal-tool evaluation should include at least one endpoint with nested objects, optional fields, conditional behavior, and business-state constraints. Count tests after deduplication. Separate Coverage from bug detection before drawing conclusions. Inspect the complex tests manually: they are often the easiest way to tell whether the system is reasoning about the API or just expanding a checklist.

For engineering and platform teams building on AI infrastructure

The results in this benchmark reflect a broader pattern in applied AI: general-purpose models and agents reach a ceiling on tasks that require domain-specific reasoning about structured data. The remaining performance gap is not addressed by prompting alone. It requires systems that model the semantic relationships within an API schema, a capability that sits at the boundary between software testing and API intelligence.

8 Scope and Limitations

APIEval-20 is a functional API testing benchmark. Security testing is outside the scope of this report. Authentication scenarios in this benchmark test functional correctness of request handling, not vulnerability discovery.

The benchmark input constraint is intentionally narrow: schema plus sample payload only. This makes the task comparable across systems and prevents tools from exploiting implementation details.

The benchmark also does not claim to represent every production API condition. It does not evaluate authentication bypass, authorization flaws, injection attacks, performance, concurrency, multi-step workflows, or stateful end-to-end journeys. Those are important testing domains, but they are outside this report.

Limitations

This benchmark isolates one task: black-box functional bug detection from a request schema and one valid sample payload. It does not measure every dimension of API testing.

The benchmark uses planted functional bugs in live reference implementations. Planted bugs make scoring repeatable, but they may not capture the full distribution of production failures in every domain.

The systems evaluated here update frequently. Results should be read as a point-in-time evaluation of the specific models, product modes, and workflows used during this study.

The results are most useful for comparing one capability: whether a system can generate high-signal request payloads that expose functional bugs from limited request-shape context.

About This Report

This report was conducted by KushoAI in 2026 using the APIEval-20 v1.0 methodology and the comparative workflow structure used in prior KushoAI research. The benchmark scenarios and scoring methodology are published by KushoAI.

For questions or feedback, reach out at contact@kusho.ai.

KushoAI 2026