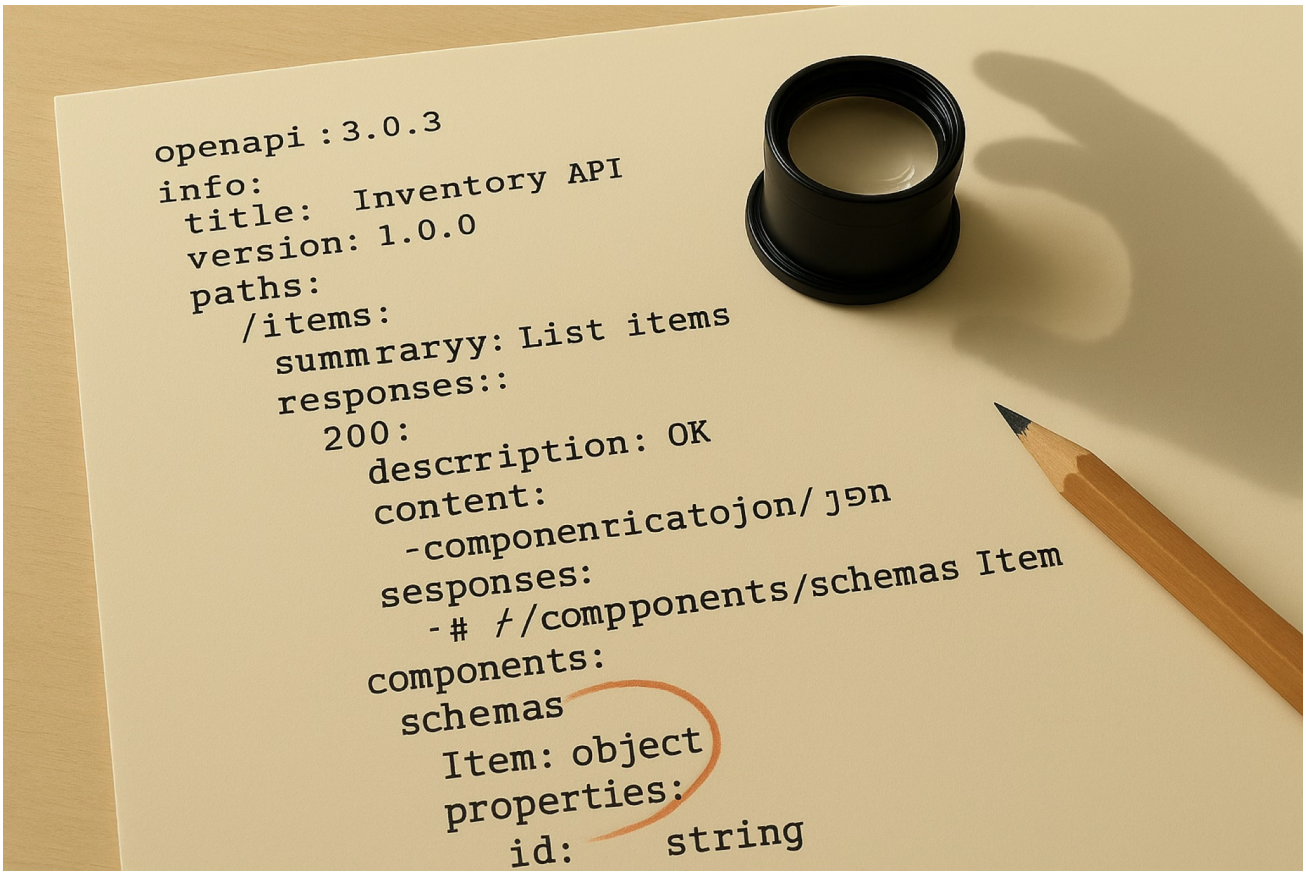

Why I built OpenAPIDoctor

"The given data was not valid YAML."

The first time I saw this error, the YAML was fine.

I had pasted ten kilobytes of perfectly formed OpenAPI into a generator, watched it crash, and squinted at a stack trace that very confidently told me my document was malformed. It wasn't. The document was a well-indented, schema-valid spec that someone had hand-edited the week before. The actual problem was a single stray key on a `Tag` object, eleven levels down, which `OpenAPIKit` had quietly rejected and which `Yams` had then wrapped in a `DecodingError.dataCorrupted` carrying a debug message that had nothing to do with the actual cause. The real error lived in `Context.underlyingError`, three more layers in.

I lost an afternoon to that error. Then I lost a morning to it again, a week later, on a different spec. Then I built OpenAPIDoctor.

What spec-first Swift gets you, and what it costs

When you generate Swift code from an OpenAPI document, the document is no longer documentation that lags behind the code. It *is* the code. Apple's `swift-openapi-generator` reads the spec, runs it through `mattpolzin/OpenAPIKit`, and emits typed Swift clients and servers from what it finds. Spec parses cleanly, the generated code compiles. Spec has any structural issue OpenAPIKit doesn't like, the generator either fails outright or emits code that won't compile.

That contract is what makes spec-first Swift worth doing. It is also what makes the spec a load-bearing artifact that nobody can afford to hand-edit carelessly. Which, of course, everybody does.

OpenAPIKit is a strict parser by design. It does not silently round-trip unknown keys, because that would let bugs through into your generated client. The trouble is that real-world specs accumulate small violations the moment more than one person edits them, or the moment anyone copy-pastes a fragment from another tool. A short, honest taxonomy:

> A `Tag` object carrying a `slug:` field. Not part of the spec. Strict parser rejects it.
>
> A `Parameter` with stray `email:` and `phone:` keys left behind by a half-finished refactor. Strict parser rejects them.
>
> `nullable: true` on a spec that has been migrated to OpenAPI 3.1, where the keyword is no longer valid. Strict parser rejects it.
>
> A response body declaring `application/x-www-form-urlencoded`, which the Swift OpenAPI runtime cannot serve. Parser accepts it; the runtime explodes downstream and writes you a different kind of letter.

Each violation is small. None is the sort of structural problem a code review catches. All of them surface as `InconsistencyError` from OpenAPIKit, which is the right thing for OpenAPIKit to do. The wrong thing is what tends to happen next: the caller gets a stack trace, fixes the first key by hand, runs the generator again, hits the next key, fixes that, runs again. On a single-file spec it is tedious. On a spec split across folders with shared schemas it is a half-day exercise that involves opening seventeen files and remembering which one you were in.

There are excellent OpenAPI linters in the JavaScript world. They are also not OpenAPIKit. If I lint with one tool and generate with another, I am postponing the failure, not preventing it. The validator had to be OpenAPIKit, end of debate.

What I wanted

The shape was small and specific:

1. Validate using the exact parser the generator uses. No second source of truth for "valid."
2. Surface every failure as a typed value, not a stack trace.
3. Auto-repair the mechanical violations, with an audit trail of what changed.
4. Stop on the violations that need human judgement, and say clearly which ones.
5. Handle multi-file specs cleanly. Everything serious lives in more than one file.
6. Exit with codes a shell pipeline can use.

That list looks obvious in hindsight. It mostly is. The interesting part is what stands between you and it.

What OpenAPIDoctor does

OpenAPIDoctor is a Swift package and CLI built on `OpenAPIKit` and `Yams`. Three pieces, in increasing order of how much fun they were to write:

A **Validator** that decodes the spec through OpenAPIKit and categorises every result.

A **SpecLoader** that handles multi-file specs.

A **Repairer** that runs validate, fix, revalidate in a loop until the spec is clean or it hits something it cannot touch.

Every result comes back as a `DiagnosisKind`. Six cases. Only one is auto-repairable:

`.ok`: spec parses cleanly. Go home.

`.vendorExtensionPrefix(codingPath, invalidKeys, subjectName)`: an extensible OpenAPI object carries keys that are not in the type's documented set and do not start with `x-`. The diagnosis carries the exact coding path and the exact keys to strip. `isAutoRepairable == true`.

`.inconsistency(codingPath, details, subjectName)`: every other `InconsistencyError` from OpenAPIKit. Unsupported `openapi:` version, malformed `$ref:` pointing at a component that does not exist. Needs a human.

`.decodingError(codingPath, details)`: Foundation-level type mismatch or missing required field. No `title:` on `info:`. Wrong type for `version:`. Needs a human.

`.fileError(details)`: the file is not there or cannot be read. Needs a different human.

`.unknown(details)`: escape hatch. Nobody is happy about this case existing, but the alternative is lying.

Having these as typed values changes how you write callers. You can pattern-match. You can ask `diagnosis.kind.isAutoRepairable` and route accordingly. From the CLI it looks like this:

```
$ openapi-doctor my-spec.yaml
{"status":"ok"}
```

```
$ openapi-doctor stray-tag-keys.yaml
{"codingPath":["tags","Index
0"],"invalidKeys":["slug","timezone"],"kind":"vendor-extension-prefix","status":
"inconsistency","subject":"Vendor Extension"}
```

One line of JSON on stdout, always. Exit code 0 for clean, 1 for "fixable, try `--fix`", 2 for "you need to edit this, or the file is missing." Nothing to parse, nothing to grep through.

The seams of the abstraction

If you have read this far, you might enjoy the parts that didn't come out as cleanly as the API suggests.

OpenAPIKit tells you which keys are invalid in English. The error message contains a sentence like `Invalid properties: [slug, timezone]`. There is no `[String]` accessor for those keys; they live inside the human-readable `details` string, sitting between two brackets, separated by commas. So OpenAPIDoctor parses them back out:

```
guard let openRange = details.range(of: "Invalid properties: [") else { return
[] }
let after = details[openRange.upperBound...]
guard let closeRange = after.range(of: "]") else { return [] }
```

```
return after[..<closeRange.lowerBound]
    .split(separator: ",")
    .map { $0.trimmingCharacters(in: .whitespaces) }
    .filter { !$0.isEmpty }
```

This is honest engineering, not clever engineering. If OpenAPIKit ever exposes the rejected keys as structured data, this disappears in one commit. Until then, the seam is here and visible.

Same applies to the classification. A vendor-extension-prefix violation is identified by `details.contains("vendor extension property")`. A substring match against an error message. I am aware of how that looks. I am also aware that it has worked perfectly across 594 spec files and counting.

OpenAPIKit reports array positions as strings. A coding path looks like `["tags", "Index 0", "slug"]`. The `"Index 0"` segment is a literal string, not an integer wrapped in `Any`. The walker that descends into the YAML to delete keys has to parse that string back into an `Int`:

```
static func parseIndex(_ segment: String) -> Int? {
    guard segment.hasPrefix("Index ") else { return nil }
    return Int(segment.dropFirst("Index ".count))
}
```

Small, dumb, real. If you have ever shipped a tool that runs against someone else's API, you know that almost all of the work is small and dumb and real.

The 3.0 / 3.1 sniff. OpenAPIKit splits its Document types: `OpenAPIKit.OpenAPI.Document` is 3.1, `OpenAPIKit30.OpenAPI.Document` is 3.0. If you point the 3.1 decoder at a 3.0 spec, you get the immortal error `"Failed to parse Document Version 3.0.x as one of OpenAPIKit's supported options."` So OpenAPIDoctor scans the raw YAML for the `openapi:` line, picks a decoder, and only parses once. Cheap, deterministic, no second source of truth for what version the spec is.

Yams.dump doesn't preserve key order. A comment in the Repairer is candid about it: *"preserves enough of the structure for our auto-repair use case (the spec is regenerated on every round anyway). Stable key ordering isn't guaranteed."* In the auto-repair flow this doesn't matter, because the source of truth after a repair is the repaired YAML, and you read what was just written. In a workflow that diffs the result against the original, you'll see ordering churn. That tradeoff is documented in the code, not hidden in a footnote.

Repair, round by round

The Repairer is my favourite part. It runs as a loop, capped at thirty rounds: validate, fix, revalidate, until either the spec is clean or it hits a diagnosis that is not auto-repairable. Each round removes exactly the keys OpenAPIKit reported, at exactly the coding path it reported them. No regex matches against the YAML, no guessing. A recursive walker descends to the target, deletes the listed keys from the dict at the leaf, and rebuilds the tree from the bottom up.

```
$ openapi-doctor --fix multi-stray.yaml
# stderr: one JSON per round, streamed as it happens
# {"removedKeys":["slug"],"round":1,...}
# {"removedKeys":["timezone","region"],"round":2,...}
# stdout: aggregate result
{"finalDiagnosis":"ok","roundsApplied":2,"status":"repaired","totalRemovedKeys":
```

```
3,...}
```

The 30-round cap is not arbitrary, but it is generous. The worst-shaped spec I have thrown at it cleared in five. The cap is there so that if the validator and the YAML ever disagree about shape and the loop starts looking like progress that isn't, the tool stops with a real diagnosis instead of running forever.

For a destructive-free survey of how broken a spec is, there's a dry-run mode that uses the same loop but discards the repaired YAML, keeping only the list of diagnoses found along the way:

```
$ openapi-doctor --all my-spec.yaml 2>/tmp/diagnoses.jsonl
{"diagnosesFound":3,"fixableDiagnoses":2,"status":"ok","terminalKind":"ok"}
```

Internally, `--all` and `--fix` share the strip implementation. The Validator's `collectAll` reaches into `Repairer.stripKeys` directly. One source of truth for "how do you remove a vendor extension key from a YAML tree." That is the kind of taste that doesn't show up in the README, which is exactly why I am putting it here.

`--output <path>` writes the repaired YAML to a side file and leaves the source untouched, useful for diffing before-and-after in CI. `--corpus <dir>` walks a directory of specs and reports per-file results with an aggregate summary, useful when you have inherited a folder of half-maintained specs and want a triage report before you touch anything.

The 594-file corpus

Multi-file specs are where the interesting bugs live. A serious API spans folders: a `finance` service that pulls `../core/parameters/id.yml`; an `analytics` service that pulls `../core/schemas/apiError.yml`; a `ledger` service that references both. `OpenAPIKit` expects one document. Without help, every external `$ref` would dangle.

The loader delegates to Stitcher, a separate library I published in November 2025 for exactly this problem. OpenAPIDoctor followed in May 2026 and took Stitcher as a dependency rather than re-implementing the resolver, which is the right division of labour: cross-file `$ref` resolution is a generic problem; OpenAPI validation is a specific one. Two libraries, two responsibilities, one composes the other.

To prove the two pieces actually agree about what "valid" means on the shape of spec I care about, the test suite drives an anonymised real-world corpus through the validator: **594 YAML files across 22 services**, of which **146 contain at least one cross-folder `../` reference**, all OpenAPI 3.0.3, the kind of layout you only get when humans have been editing for a year. The test asserts every entry point validates cleanly. When it does, I know Stitcher's merged output and OpenAPIKit's strict parser agree, in detail, across the whole tree, every commit.

You can flip Stitcher off with `--no-resolve-refs`, which loads the spec file as-is. Useful when the referenced files are not on disk, or when you want to isolate a problem to a single file and stop wondering if the resolver is the one lying to you.

Where it fits

OpenAPIDoctor is one stage in a longer code-generation pipeline I run. Specs come in. Validation happens. Generation happens. Swift compiles. There is nothing exotic about that flow except the requirement that every stage produces something the next stage can use, without surprises.

OpenAPIDoctor's contract is the simplest in the chain. After it runs, the spec is either clean, or the caller knows exactly what is wrong with it, whether a tool can fix it, and where in the document the problem lives. That contract is what lets the rest of the pipeline be boring. Boring pipelines are the only kind that ship.

Where to find it

codeberg.org/Mihaela/OpenAPIDoctor, tagged at `1.0.0`. MIT licensed. Library for embedding in your own pipeline, CLI for the shell:

```
.package(url: "https://codeberg.org/Mihaela/OpenAPIDoctor", from: "1.0.0"),
```

If you generate Swift from OpenAPI on anything bigger than a toy spec, you probably want this somewhere upstream of the generator. And if you ever see *"The given data was not valid YAML."* in a stack trace, you now know two things: it is lying to you, and somebody already wrote the tool that tells you the truth.