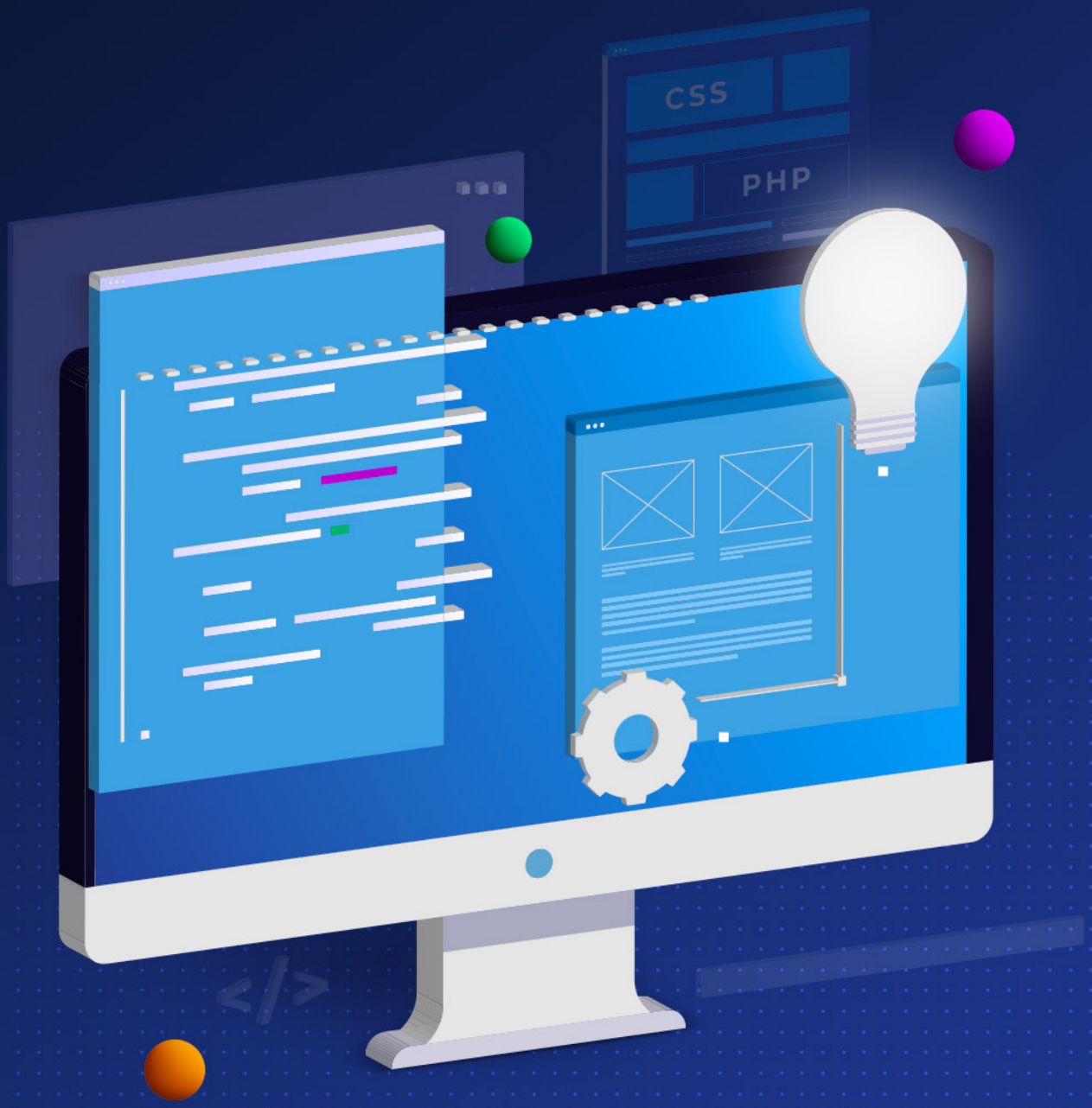
Stoplight

Leverage API-First Design and API Documentation to Drive Business Agility



stoplight.io

Are you ready to take your internal innovation and strategic partnerships to the next level? Do you want to allow developers to build on top of what you've built? Every piece of code is a bit unique, but there is a movement toward anyone being able to leverage the power of code. APIs are the backbone of this movement and of great digital agility in general. Today Stoplight guides you through some axioms of great APIs including:

API documentation is critical for API platform success.

If teams apply a more traditional code-first, bottom-up approach to design and documentation, they may miss critical business goals and fail to meet industry and regulatory requirements.

Taking a collaborative, API design-first approach accelerates the design, documentation, and delivery process and avoids costly mistakes.

Documenting your API should occur across the entire delivery process.

Stoplight supports an API design-first approach to help create a well-designed API with great documentation. We introduce you to our tool as a potential solution to bring all of the advice presented into action.

What are you waiting for? Let's get started building great APIs!

API Design Best Practices

Before we start talking about what is API-first and how it drives design, what makes for good API design in the first place?

First, we can't talk best practices without offering you examples of great API design, built with their customers in mind.

Some favorite examples of great API documentation are Twilio, Django, Stripe, and MailChimp. This is a good range of examples from both API-based business models and more traditional business software.

Of course, there great API design is more than just solid docs. Some other examples of stellar API design include Plaid for financial services, Okta for authorization and authentication, Trello for project management, Digital Ocean for cloud management, and Google for, well, everything.

Now that we know which are examples of beautiful, developer-delighting APIs, let's dive into some of the best practices.

1. The full API lifecycle must be tied to business goals.

Your API is a product within your whole business offering, so you need to first look at where it fits into that offering. Why are you designing that API? What business goals is it contributing to? What customer pain point does it solve? Good API-first design starts with defining your value proposition and identifying the stakeholders that it matters to. Only then can you serve that purpose through its design.

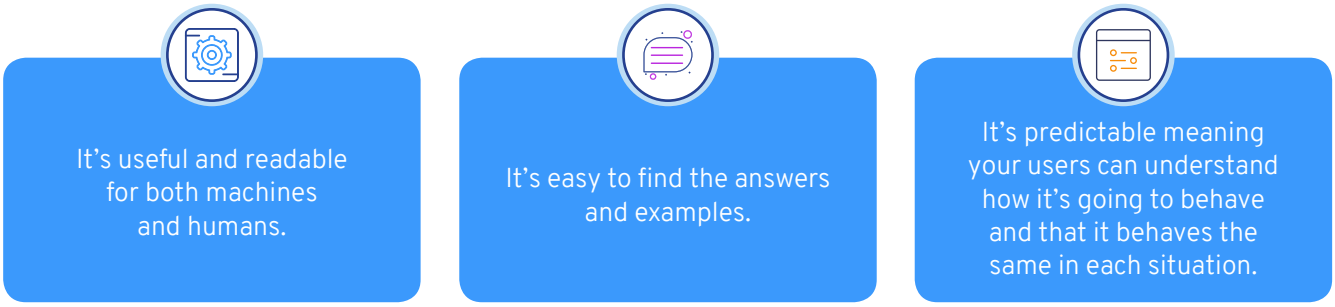
Lorinda Brandon, director of developer marketing at Capital One DevExchange, put it best:

"It starts with the idea: What's your idea for your API? What value do you believe it has? Before you even design the endpoints, before you start writing your Swagger definition, what do you do? Our group works with the product management teams in other areas of Capital One to help them hone their ideas to come up with the business implications of their APIs. Then it moves into design, implementation, support, marketing. When we say we do All The Things, we mean that we're interfacing with all the different functions across all the different businesses to help them deliver their API."

Core API design considerations also include the business drivers and the resources and constraints — like budget and your current infrastructure — which can be difficult for a lot of developers to pinpoint and focus on. However, it gets down to the crux of it all — **you need to know why you are designing a new API** and how it benefits your overall business objectives.

2. Good API design is easy to read and work with.

Good API design and good API documentation share the same ideals:



Good API documentation is an essential part of good API design. Certainly an API business that offers a fantastic example is the payment platform Stripe's API documentation. What makes the Stripe API so popular and a model for other businesses? According to user Logan Volkers, it comes down to four things:

1. **Simplicity -** really easy to get to Hello, World
2. **Docs -** the Stripe API is very self-service with both language-specific reference docs and high-level "How do I..." guides
3. **Power -** logical chains of events are strung inside one API call
4. **Dogfooding -** Stripe's internal tools were released publicly, so they use their own stuff before they release it on the world.

On the other hand, there are always products that are no loved by their customers, even when it may be necessary to integrate with them. One such unloved API is the Facebook API, which is notorious for deprecating features without giving any prior notification to users and one known for not documenting versions and changes.

3. Great API design offers simple, but complete documentation.

Yes this sounds contradictory, but nothing should be missing and nothing should be extra in your API's code or documentation. This takes experience and it takes being able to share your code with your potential users.

Of course, if your documentation is too complicated that's because your code is too complicated. One of the best benefits of API-first design is that you plan out your API by putting it all on paper (or more likely using Stoplight to model your complex APIs ahead) you are able to better visualize your API ahead and make sure that it's no more complicated than necessary — before you start implementing the API with code.

If you are looking for a tool to easily edit your APIs, try out Stoplight over other more raw editors. We of course use the market frontrunner OpenAPI Specification (OAS), but we like to think we make it more user-friendly by using basic forms and resource modeling — you get a more pleasing user interface and experience, while we generate complex (but not too complex) code behind the scenes. You get the benefits of it without having to edit raw, clunky OAS YAML files.

"We decided to go for Stoplight because we are convinced that the API must be held by the business people — the product owners, but the Swagger specification looks repulsive to them," said Nicolas Tisserand, an API architect at the Crédit Mutuel Arkéa French banking group.

He said that a lot of the Product Owners (POs) on his team are new to the position and come to it after designing processes or UI screens. Not all of them are ready to design APIs — in fact, often POs shy away from things like OpenAPI editors since they read YAML as "code", despite its intention to be more approachable. Therefore they try to pair product owner with developer, and they chose Stoplight for its very visual ease of use.

"This is why a graphical tool like Stoplight was a good opportunity — perhaps the only one — to involve them in the design of API. We recommend co-designing, with a product owner and a developer. The developer should know a bit about REST and Swagger in order to drive the product owner and let him focus on his job — the business," Tisserand continued. "On the other hand, the developer must be accompanied by a business user because he doesn't know all about the tricky parts of the business. It's particularly right in the financial domain, for example with all the legislation and specific business rules. So, it was the real first reason to go for Stoplight."

Stoplight was built to be understood by developers and non-developers. Product managers, partners, and even potential clients can have input into the design of your API, because they can see it clearly mapped out in this friendly UI. Stoplight was built to drive documentation with design in mind.

4. Don't forget to dogfood.

You want to design great APIs? Consume your own APIs! Nothing will tell you more about how complex your API is or any of the problems than using it yourself. On the other side, by consuming your own APIs alongside other stellar APIs, you learn so much about best practices that go into API building and design.

5. Early access and ample feedback.

Great API design involves giving early access to developers, so you can iterate on their feedback, and then design delightful APIs that developers will actually use.

Brandon has a great piece on how to incorporate feedback loops into your API design. And remember, it's better to get feedback early and often, than to get loads of support calls and folks leaving your API behind.

6. And some API design basics you can't miss.

A lot of what we talked about above is simply good design and software development, but there are some ingredients you cannot forget to add:

Logical endpoint naming conventions that are consistent throughout, so much that a developer a bit used to your API could probably guess the right next command.

Clear error codes - If you are designing a REST API, there are standard error codes. Others you will need to define in your documentation before releasing.

Uptime - of course you need your API to work and you meet your SLA. Design tests early on to make sure your API will work and within the your overall organizational architecture and integrations.

It should be really fast to get someone to Hello World. (30 seconds when possible.)

Access control - who can access what data within your API — this and API security must be designed from the start.

API is to expose data, so how you're planning to do that securely needs to be drawn out well before you start building your API.

7. Pretend it's external from the start.

Want to design an API with all these best practices in mind? Even if it's intended to be eternally internal, design it with externalizing in mind. There will be fewer corners cut and more best practices put into practice when you not only design API first, but open API first.

Design-first API product development

The traditional build-first approach

Let's kick this discussion off with the more "traditional" code-first design. Many companies build application-first and then go ahead and build an application programming interface (or a lot of them) later when they want to extend functionalities, features, or partnerships. With this "build-first" approach API is usually built in isolation by different developers and even different departments than the Web or mobile app it's being built to serve. Lately, this is being called an "artificial API" because it wasn't built as part of the design phase and wasn't properly tested. This device-centric product design results in duplicated effort and wasted engineering work. Most often, it's built as a quick fix to a problem instead of the integral backbone of the solution.

Developer evangelist at Grab ride-hailing and payment platform Yos Riady presents a great example of the flaws that come with the build-first approach:

"You're building an API...

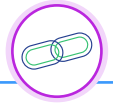
You develop a backend service with a few endpoints and deploy it to production. You publish several official language-specific API clients as well as an API documentation. The day ends on a happy note.

The following day, a new feature is going to be added the API. To do that, you have to:

Update the server implementation to support the new feature.

Update all client libraries (one SDK for each supported platform and language).

Update the documentation.



All the above must be **consistent with each other.**

Also, **the front-end team is blocked** until your backend API is complete.

...You let out a heavy sigh."

Lately more and more companies are realizing that it's building the API first and then your apps on top of it is a much better, device-agnostic, language-agnostic approach. This "clean-room" approach forces the API to be designed for usage, guarantees a more developer-friendly REST API, avoids integration failures, supports rapid prototyping, and prevents future spaghetti code, as the API acts as a sturdier foundation.

The innovative design-first approach

Design-first API product development is a narrative style of communicating in a machine-readable way that provides a simple developer experience. It's usually built with as a REST API.

Principal architect at Adobe Lars Trieloff sums up API-first product development in three principles:

1. Your API is the first user interface of your application.

2. Your API comes first, then the implementation.

3. Your API is described (and maybe even self-descriptive.)

In other words, your API is ground zero before you build on top of it and you describe it even before you venture out to build. Trieloff argues that your API is the strong and stable part of your build, while the applications you build on top of it are constantly changing.

"Here's another way to think about this approach: If your API is the surface area of your product; the functionality is its volume. Doubling the functionality will only grow your API surface by 25 percent," Trieloff explains.

As the name implies, API-first design puts the API first from scratch, outside of your current architectural constraints and limitations. It follows trends like microservices and containers, where developers build delightful code that meets more precisely the need at hand and doesn't need to fit into the overall complicated architectural monolith.

Eventually your API will have to connect with the rest of your system, either helping refactor a legacy system or simply connecting to part of it.

API-first design is very consumer focused, making sure to define clearly the interface between the infrastructure at the bottom and the API-consuming clients on top, so these two partners can go ahead and start building with an assurance that all three pieces will fit together.

This trend really kicked off with the introduction of API description formats, and an emphasis on mocking, scoping, and virtualizing your APIs before even building it. It all fits into the agile development trend of getting something — even an incomplete product — out to your internal or external to try it on for size before investing too much time into building something nobody wants.

Most notably in the world of API description formats are the OpenAPI Specification (OAS, formally known as “Swagger”), API Blueprint, and RAML. All three of these popular formats support RESTful API modeling languages. API specifications can be written in YAML or JSON, with formats that are supposed to be easy to learn and readable to both humans and machines — with nothing missing yet nothing superfluous for either audience.

Stoplight is a product built on top of Swagger/OAS. Like all good API-first designing, you start from scratch by plugging into the API Designer your simple details like the name and description of your API, the open protocols and hosting you’ve built on top of, and your base path that connects your endpoints. When you map it out ahead, you are able to leverage a modular API tool like Stoplight to fill in the blank and describe endpoints. In return, this makes for much cleaner code that’s easy to build and is both machine- and human-readable.

Using a JSON schema editor, you can then create quick, yet complex models that reference back.

The idea is that you can mock up an entire API or just specific endpoints, making design considerations and testing them out early on. With Stoplight, you can even mock up a complete APIand define dynamic endpoints with sample data, so you can test it all out more quickly.

Tisserand cited as one of his favorite features in Stoplight “the Scenarios module that helps us to test the APIs without the UI.”

Just don’t forget to get feedback from your key users in every step of your trials. And don’t forget to collaborate! While the IT world is moving toward more individual developer decision-making, it’s important we collaborate, share and test each other’s work, particularly in the API space where hopefully you will be building interfaces that have multiple use cases and can reuse code.

For the team at Credit Mutuel Arkea, collaboration means architects helping other less-tech team members like Product Owners use tools.

Tisserand said, “My team is only composed of architects. Our role is to provide support to other teams of the company. We help them through individual assistance on projects and with training sessions in which we teach how to use Stoplight and our API gateway. We also speak about REST, governance and strategy. These training sessions are dedicated to developers AND product owners and managers.”

This is also why they like Stoplight because it’s much simpler than teaching the rather arduous and clunky Swagger.

“If I had to quote just one advantage, it would be the WYSIWYG-like design capabilities of Stoplight. For us, it’s difficult to drive Product Owners or business people to design an API. It’s an important change of their day-to-day job. A visual editor like Stoplight’s is the first step to access to the API world,” Tisserand continued.

Stoplight enables cross-functional collaboration over API documentation.

“By using Stoplight, we are enforcing Product Owners to be clear on what they want. They must take part to the design process and this is done through Stoplight. Moreover, we have now a real Swagger with relevant comments. Not generated ones,” he said.

Nine basics of API-first design

While APIs are like snowflakes, each successful API-first design follows the same rough outline:

1. **Plan Ahead:** Focus on domain-driven design and your business goals up front. Sketch out your needs ahead.
2. **Design and Validate:** Plan out how your design is going to look and how you can test those pieces. Mock it up and test out your API design against use cases.
3. **API Specification:** Figure out what the first rendition of your API is going to look like (it will of course evolve based on user needs) and generate the API documentation from it. Use a tool like Stoplight to mock up your use cases.
4. **Test, Test, Test!:** This now should be a step in the rest of your API development through depreciation. Once you have your spec, you can automatically generate tests from it. Just don’t forget to also eat your own dogfood and test out your user experience too.

5. **Implement:** Allow app developers and others to build on top of your API (truly checking your developer experience). And see how your servers and infrastructure is interacting with your API.
6. **Monitor:** Make sure everything is working with automatic API testing and continuous integration testing, easily generated via beautiful, machine-readable API documentation.
7. **Operate:** Put your API out there into production use. Be eager to support your users.
8. **Engage:** Use a developer portal to communicate with and get feedback from your users, so you can iterate and improve on all.
9. **Team:** Use your retrospectives to meet as a team to discuss what worked and what doesn’t to uncover and acknowledge **API design best practices.**

The Learn > Reuse > Recycle theme you see above is important because — sadly, since it's the number one thing API consumers are looking for — most teams have very little budget for API documentation. Tools like OpenAPI allow you to describe your API and create a spec around it even before it's developed, while making sure it's all going to be built on top of your company's and your industry's standards.

The benefits of API-first design

Good API-first design smooths the rest of the API lifecycle, too. Another benefit of API-first design is how it helps the API lifecycle down the line. When you design APIs to solve real challenges for your end users, and when you put developer experience in the forefront of your API development, you will find that tasks like testing, documenting, and securing your APIs become far less time-consuming and much more automated, saving you time, which of course saves you money. Everything grows out of that spec you just built. Plus, it creates a sense of transparency both across your team and with your users, which means everything is clearer from the start.

Worried that you're already tethered to a monstrous monolith? Ideally, API-first design sees you building a base of application programming interfaces before you build your Web, mobile, or one-page application, and only then do you define the channels you'll be making that API's resources available to — but that's not always the case. You can adopt as a team or as a company to be API-first from now on and follow all this advice in your next strategic developer decisions.

In addition, when you are looking at software to help enhance your products through integrations, keep your eyes out for open source, since they are more likely to be built API-first. Partnering with API-first can help keep you on your newfound API-first path.

Leverage Your API Documentation to Streamline Your API Design

API-first product development more and more often sees self-describing APIs including links that enable the discovery of other resources, closing the API lifecycle loop, by enabling API documentation and testing.

Banking API architect and API-first proponent Arnaud Lauret (aka the "API Handyman") offers us ways to use your API documentation and the standards behind it to make sure you are encouraging and enforcing reliable API design.

Error codes	Clarify key terms	Examples
Your documentation should define all of them, particularly if they aren't typical to REST APIs in general, but perhaps are industry specific, like Lauret sees in the payments and banking space.	All dates should be written a certain way. All name titles this way. All monetary amounts (commas and periods/full stops are inverted in much of Europe) must be like this.	Loads of them! This is another great way to assure your API design is going to be simplistic by filling your documentation with key code samples that you've derived from your specs. By building the examples ahead of the API, you help make sure it's easy to use.

Know how your users act and build docs and APIs as close to that behavior as possible. Then use your API documentation and specification ahead of time to enforce it. And then add manual key pieces to the docs that clarifies anything that verges off that course.

How to make the cultural change to API design first

People before processes and always putting the customers first — these are some of the major points made in the Agile Manifesto originally authored in 2001. And yet when we hear the word "documentation" we automatically cringe about Waterfall. Like all technical changes and transformations, it's mostly about the culture backing it all.

As Tisserand put it: "I think that it's important to point out that the people are the key. This is one of our main problems — convince the people and some hierarchical managers, make them change of mindset, and so on. In a company of almost 10,000 employees like mine, it's difficult to achieve.

"APIs are our future products. It's the way we'll commercialize our banking services, even for our own needs. It's our digital transformation."

If you are an old-school company that's into Waterfall project management, then probably you aren't ready to design the API first with its focus on feedback loops and iterative improvements. While documentation leads API-first design, it's about mapping out the API needs, not going into excessive detail. Mainly it's about those tight feedback loops that lead to iterative and customer-centric rapid learning process which makes it much more agile.

Indeed, there are some traps of traditional code-first API design that seems haunted by overly pedantic Waterfalls. These are some anti-patterns that arise when you build code before spec or docs:

You begin bottom-up from the database or code, without considering how consumers will use your API

You break the API design after a general release, without proper notification and change management, and without considering the impact to your consumers

You completely miss the purpose of an API by designing and implementing it in black-box isolation

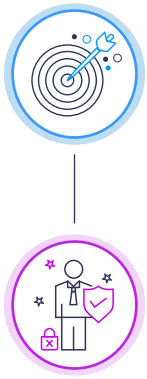
On the other hand, API-first design puts your customers and the stability of your code at the forefront, all while making sure everyone on your team has insight into your API and a vision for how it'll fit into your overall business architecture.

Plus, following another line of the Agile Manifesto — "Seek the simple by maximizing the amount of work not done." — API-first is also very agile because it leverages API specification to automatically create the backbone of your code, which in turn will create your API documentation. Then, if you leverage a tool like Stoplight, you don't even have to map out the whole specification. Just send HTTP traffic through Prism, and Stoplight will map out your API specification.

API-first is all about automating the boring work so you can focus on improving your software in a way that better responds to client needs. API-first is a driving force of business agility.

Step-by-step of how API-first design meets business agility

It is often best if we don't immediately implement our API, but rather take a more agile approach to our API design process:



Agile API Step #1:

Discover the capabilities that need to be delivered. Focus on behavior and desired outcomes before you zero in on the data.

Agile API Step #2:

Model and design your API to meet your users' needs. Don't just assume them, make sure you understand your users' needs (and that they may evolve along with your product.)

Agile API Step #3:

Document your first-pass API design enough to provide understanding and provide some code examples to demonstrate common use cases. If it's complicated, so will your code be.

Agile API Step #4:

Seek feedback from your stakeholders to make sure it meets your users needs. As you adapt, repeat your feedback requests.

Agile API Step #5:

Incorporate that feedback, repeating steps one through four as often as needed.

Agile API Step #6:

Mock your API from your OpenAPI specification and let your key stakeholders — internal users, partners, or some key external customers — review it all.

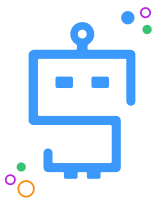
Agile API Step #7:

Release your implemented API as a preview release. One final chance (or multiple depending on your iterations) to get more feedback and make the necessary changes to suit your customers' needs.

Agile API Step #8:

Promote the release to production. When making developer-delighting software, you will repeat the first seven steps as much as needed before you feel sure enough to produce a well-designed API.

In summary, good API design — and good design in general — continuously considers its consumers and leverages their feedback to make products they actually want to use.



Email: sales@stoplight.io

https://www.stoplight.io