O'REILLY®

Free Chapters

compliments of

NGINX

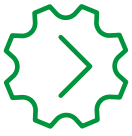
Continuous API Management

MAKING THE RIGHT DECISIONS IN AN EVOLVING LANDSCAPE

Mehdi Medjaoui, Erik Wilde,
Ronnie Mitra & Mike Amundsen
Foreword by Kin Lane

Why Trust Your APIs to Anyone Else?

Traditional API management tools are complex and slow. As the most-trusted API gateway, we knew we could do better. NGINX has modernized full API lifecycle management.



API Definition and Publication

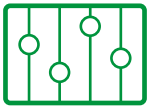
Define APIs using an intuitive interface.

Rate Limiting

Protection against malicious API clients.

Authentication and Authorization

Applying fine-grained access control for better security.



Real-Time Monitoring and Alerting

Get critical insights into application performance.

Dashboards

Monitor and troubleshoot API Gateways quickly.

Learn more at nginx.com/apim

NGINX

Continuous API Management

Making the Right Decisions in an Evolving Landscape

This excerpt contains Chapters 1–3 of the book *Continuous API Management*. The complete book is available on oreilly.com and through other retailers.

Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Continuous API Management
by Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Alicia Young
Production Editor: Justin Billing
Copyeditor: Rachel Head
Proofreader: James Fraleigh
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrators: Rebecca Demarest and Ronnie Mitra

November 2018: First Edition

Revision History for the First Edition
2018-11-14: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492043553* for release details.

This work is part of a collaboration between O'Reilly and NGINX. See our statement of editorial independence.

978-1-492-04355-3

[LSI]

Table of Contents

Foreword

APIs have grown tremendously in recent years. There are now more than 20,000 APIs for building mobile and web applications available in the repository at ProgrammableWeb, a leading source of news and information about APIs, with about 170 new APIs being added every month. And these are just the *external* APIs that enterprises expose to third-party developers; the number of *internal* APIs is poised for huge growth over the next few years, fueled by the adoption of microservices that use APIs for service-to-service communication.

As you expand your organization's API footprint, how do you scale your technology, people, processes, and governance? What kind of guidelines should you provide for designing, implementing, and deploying APIs in your organization? This excerpt of the first three chapters of *Continuous API Management* provides answers to these critical questions. This is a must-have book for technical practitioners as well as professionals responsible for an enterprise's API business strategy. It sheds light on the critical business elements of delivering APIs—from establishing drivers for building them to deploying, operationalizing, refining, and evolving them continually at scale.

Together, authors Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen from the API Academy bring a wealth of experience to guide your development of an API program in a holistic fashion from ideation to realization, taking both technical and business aspects into account. In addition to sharing their own expertise on API management, they provide critical insights into common approaches adopted by dozens of enterprises across multiple industries.

NGINX powers many traditional API management solutions, including Axway, IBM DataPower, Kong, MuleSoft, Red Hat 3Scale, and Torry Harris. According to our 2018 user survey, more than 40% of our users have deployed NGINX as an API gateway. NGINX Controller's API Management Module is a full API lifecycle management solution that is lightweight and delivers high performance. NGINX's unique architecture for API management is well suited to the needs of both monolithic applications and modern distributed applications based on microservices.

Whether you are just starting to build APIs for your enterprise or needing to enhance your current API landscape, we sincerely hope you enjoy this excerpt as you develop a stable and successful API program.

— Karthik Krishnaswamy
Director, Product Marketing
NGINX, Inc.

CHAPTER 1

The Challenge of API Management

> Management is, above all, a practice where art, science, and craft meet
>
> —Henry Mintzberg

A survey from Coleman Parkes released in 2017 shows that almost 9 in 10 global enterprises have some form of API program. This same survey shows that these companies are seeing a wide variety of benefits from their API programs, including an average increase in speed-to-market of around 18%. However, only about 50% of these same companies say they have an advanced API management program. This points to a key gap in many enterprise-level API programs: the distance between the operational APIs that represent key contributions to revenue and the management skills and infrastructure needed to support these revenue-producing APIs. It is this gap that this book hopes to address.

The good news is there are many companies out there successfully managing their API programs. The not-so-good news is that their experience and expertise is not easily shared or commonly available. There are several reasons for this. Most of the time, organizations that are doing well in their API management programs are simply too busy to share their experiences with others. In a few cases, we've talked to companies that are very careful about how much of their API management expertise they share with the outside world; they are convinced API skills are a competitive advantage and are slow to make their findings public. Finally, even when companies share their experience at public conferences and through articles and blog posts, the information they share is usually company-specific and difficult to translate to a wide range of organizations' API programs.

This book is an attempt to tackle that last problem—translating company-specific examples into shared experience all organizations can use. To that end, we have visited with dozens of companies, interviewed many API technologists, and tried to find the common threads between the examples companies have shared with us and with

the public. There are a small handful of themes that run through this book that we'll share here in this introductory chapter.

A key challenge to identify right up front is sorting out just what people mean when they talk about APIs. First, the term "API" can be applied to just the *interface* (e.g., an HTTP request URL and JSON response). It can also refer to the code and deployment elements needed to place an accessible service into production (e.g., the `customerOnBoarding` API). Finally, we sometimes use "API" to refer to a single *instance* of a running API (e.g., the `customerOnBoarding` API running in the AWS cloud vs. the `customerOnBoarding` API running on the Azure cloud).

Another important challenge in managing APIs is the difference between the work of designing, building, and releasing a *single API* and supporting and managing *many APIs*—what we call an *API landscape*. We will spend a good deal of time in this book on both ends of this spectrum. Concepts like *API-as-a-Product* (AaaP) and the skills needed to create and maintain APIs (what we call *API pillars*) are examples of dealing with the challenges of a single API. We will also talk about the role of API maturity models and the work of dealing with change over time as important aspects of managing an API.

The other end of that spectrum is the work of managing the API landscape. Your landscape is the collection of APIs from all business domains, running on all platforms, managed by all the API teams in your company. There are several aspects to this landscape challenges, including how scale and scope change the way APIs are designed and implemented as well as how large ecosystems can introduce added volatility and vulnerability just because of their size.

Finally, we touch on the process of decision making when managing your API ecosystem. In our experience this is the key to creating a successful *governance* plan for your API programs. It turns out the way you make decisions needs to change along with your landscape; holding on to old governance models can limit your API program's success and even introduce more risk into your existing APIs.

Before we dive into the details on how you can learn to deal with both challenges—your individual APIs and your API landscape—let's take a look at two important questions: what is API management, and why is it so hard?

What Is API Management?

As mentioned, API management involves more than just governing the design, implementation, and release of APIs. It also includes the management of an API ecosystem, the distribution of decisions within your organization, and even the process of migrating existing APIs into your growing API landscape. In this section, we'll spend time on each of these concepts—but first, a short explanation of what we mean by "API."

What Is an API?

Sometimes when people use the term "API" they are talking about not only the interface, but also the functionality—the code behind the interface. For example, someone might say, "We need to release the updated Customer API soon so that other teams can start using the new search functionality we implemented." Other times, people may use the term to refer only to the details of the interface itself. For example, someone on your team might say, "What I'd like to do is design a new JSON API for the existing SOAP services that support our customer onboarding workflow." Both are correct, of course—and it seems pretty clear what is meant in both cases—but it can be confusing at times.

To try to clear up the distinction and make it easier for us to talk about both the interface and the functionality, we are going to introduce some additional terms: interface, implementation, and instance.

Interface, implementation, and instance

The acronym API stands for *application programming interface*. We use *interfaces* to gain access to something running "behind" the API. For example, you may have an API that exposes tasks for managing user accounts. This interface might allow developers to:

- Onboard a new account.
- Edit an existing account profile.
- Change the status of (suspend or activate) an account.

This interface is usually expressed using shared protocols such as HTTP, Thrift, TCP/IP, etc. and relies on standardized formats like JSON, XML, or HTML.

But that's just the interface. Something else actually needs to perform the requested tasks. That something else is what we'll be referring to as the *Implementation*. The implementation is the part that provides the actual functionality. Often this implementation is written in a programming language such as Java, C#, Ruby, or Python. Continuing with the example of the user account, a `UserManagement` implementation could contain the ability to create, add, edit, and remove users. This functionality could then be exposed using the interface mentioned previously.

Decoupling the Interface from the Implementation

Note that the functionality of the implementation described is a simple set of actions using the Create, Read, Update, Delete (CRUD) pattern, but the interface we described has three actions (`OnboardAccount`, `EditAccount`, and `ChangeAccountStatus`). This seeming "mismatch" between the implementation and the interface is common and can be powerful; it decouples the exact implementation of each service from the interface used to *access* that service, making it easier to change over time without disruption.

The third term in our list is *instance*. An API instance is a combination of the interface and the implementation. This is a handy way to talk about the actual running API that has been released into production. We manage instances using metrics to make sure they are healthy. We register and document instances in order to make it easy for developers to find and use the API to solve real-world problems. And we secure the instance to make sure that only authorized users are able to execute the actions and read/write the data needed to make those actions possible.

Figure 1-1 clarifies the relationship between the three elements. Often in this book, when we write "API" we're talking about the instance of the API: a fully operational combination of interface and implementation. In cases where we want to highlight *just* the interface or *only* the implementation, we'll call that out in the text.

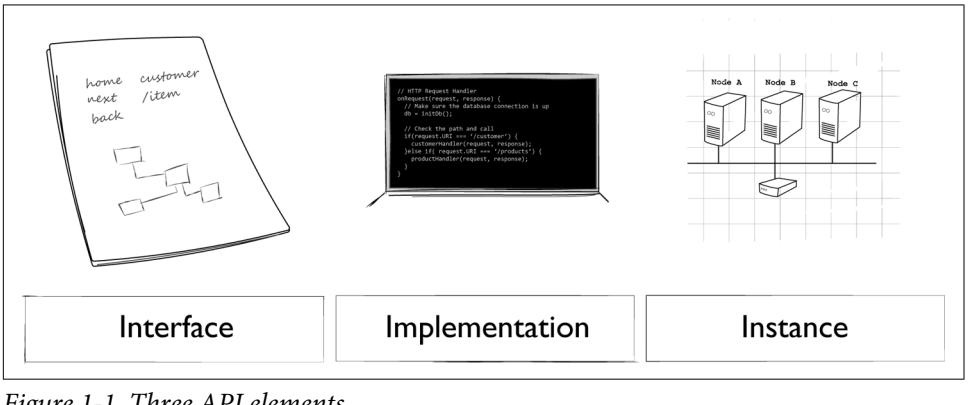


Figure 1-1. Three API elements

More Than Just the API

The API itself—the technical details of interface and implementation—is just part of the story, too. The traditional elements of design-build-deploy are, of course, critical to the life of your APIs. But actually *managing* APIs also means testing them, documenting them, and publishing them to a portal so that the right audience (internal

developers, partners, third-party anonymous app developers, etc.) can find and learn how to use them properly. You also need to secure your APIs, monitor them at runtime, and maintain them (including handling changes) over their lifetime. All these additional elements of an API are what we call *API pillars*: elements that all APIs need and all API program managers need to deal with. We'll dig into pillars when we walk through the list of ten key practices vital to creating and maintaining healthy APIs.

The good news about these practice areas is that they transcend any single API. For example, the skill of documenting APIs well is transferable from one API team to the next. The same goes for learning proper testing skills, security patterns, and so forth. That also means that even when you have separate teams for each API domain (sales team, product team, backoffice team, etc.), you also have "cross-cutting" interests that bind people within teams to other people in other teams.[1] And this is another important aspect of managing APIs—enabling and engineering the teams that build them. We talk more about how this works in different organizations later in this book.

API Maturity Stages

Knowing and understanding the API pillars is not the entire picture, either. Each API in your program goes through its own "lifecycle"—a series of predictable and useful stages. Knowing where you are in the API journey can help you determine how much time and resources to invest in the API at the moment. Understanding how APIs *mature* allows you to recognize the same stages for a wide range of APIs and helps you prepare for and respond to the varying requirements of time and energy at each stage.

On the surface, it makes sense to consider that all of the API pillars need to be dealt with when designing, building, and releasing your APIs. But reality is different. Often, for early-stage APIs it is most important to focus on the design and build aspects and reduce efforts on documentation, for example. At other stages (e.g., once a prototype is in the hands of beta testers), spending more time on monitoring the use of the API and securing it against misuse is more important. Understanding maturity stages will help you determine how to allocate limited resources for maximum effect. We'll walk you through this process later in this book.

More Than a Single API

As many readers may already know, things change when you start managing a lot of APIs. We have customers with thousands of APIs that they need to build, monitor,

1 At music streaming service, Spotify, they call these cross-cutting groups "guilds."

and manage over time. In this situation, you focus less on the details of how a single API is implemented and more on the details of how these APIs coexist in an ever-growing, dynamic ecosystem. As mentioned earlier, we call this ecosystem the API landscape, and we devote several chapters to this concept in the second half of the book.

Much of the challenge here is how to assure some level of consistency without causing bottlenecks and slowdowns due to centralized management and review of all the API details. This is usually accomplished by extending responsibility for those details to the individual API teams and focusing central management/governance efforts on normalizing the way APIs interact with each other, ensuring that there is a core set of shared services or infrastructure (security, monitoring, etc.) in place and available for all API teams, and generally providing guidance and coaching to more autonomous teams. That is, it's often necessary to move away from the usual centralized command-and-control model.

One of the challenges when working toward distributing decision making and autonomy deeper in the organization is that it can be easy for those higher up in the organization to lose visibility into important activities happening at the team level. Whereas in the past a team might have had to ask permission to take an action, companies that extend additional autonomy to the individual teams will encourage them to act without waiting for upper-level review and permission.

Most of the challenges of managing a landscape of APIs have to do with *scale* and *scope*. It turns out that as your API program grows, it doesn't just get bigger; it also changes in *shape*. We'll discuss this in more detail later in this chapter (see "Why Is API Management Difficult?" on page 7).

The Business of APIs

Beyond the details of creating APIs and managing them in a landscape, it is important to keep in mind that all this work is meant to support business goals and objectives. APIs are more than the technical details of JSON or XML, synchronous or asynchronous, etc. They are a way to connect business units together, to expose important functionality and knowledge in a way that helps the company be effective. APIs are often a way to unlock value that is already there in the organization, for example through creating new applications, enabling new revenue streams, and initiating new business.

This kind of thinking focuses more on the needs of API consumers instead of those producing and publishing the APIs. This consumer-centric approach is commonly referred to as "Jobs to Be Done," or JTBD. It was introduced by Harvard Business School's Clayton Christensen, whose books *The Innovator's Dilemma* and *The Innovator's Solution* (Harvard Business Review Press) explore the power of this approach in depth. For the purposes of launching and managing a successful API program, it

serves as a clear reminder that APIs exist to solve business problems. In our experience, companies that are good at applying APIs to business problems treat their APIs as *products* that are meant to "get a job done" in the same sense that Christensen's JTBD framework solves consumer problems.

One way an API program can help the business is by creating a flexible set of "tools" (the APIs) to build new solutions without incurring a high cost. For example, if you have an `OnlineSales` API that allows key partners to manage and track their sales activity and a `MarketingPromotions` API—that allows the marketing team to design and track product promotional campaigns, you have an opportunity to create a new partner solution: the `SalesAndPromotions` tracking application.

Another way APIs can contribute to the business is by making it easy to access important customer or market data that can be correlated to emerging trends or unique behaviors in new customer segments. By making this data safely and easily available (properly anonymized and filtered), APIs may enable your business to discover new opportunities, realize new products/services, or even start new initiatives at a reduced cost and faster time to market.

We cover this important aspect of AaaP in Chapter 3.

Why Is API Management Difficult?

As we mentioned at the beginning of this chapter, while most companies have already launched an API program, only about 50% consider themselves to be doing a good job *managing* their APIs. What's going on here? What are the challenges, and how can you help your company overcome them?

As we visit with companies all over the world, talking about API lifecycle management, a few basic themes emerge:

Scope
: Just what is it that central software architecture teams should be focusing upon when governing APIs over time?

Scale
: Often, what works when companies are just starting out on their API journey doesn't scale as the program grows from a few small teams to a global initiative.

Standards
: What we find is that, as programs mature, management and governance efforts need to move from detailed advice on API design and implementation to more general standardization of the API landscape, freeing teams to make more of their own decisions at a detailed level.

Essentially, it is the continued balance of these three elements—scope, scale, and standards—that powers a healthy, growing API management program. For this reason, it is worth digging into these a bit more.

Scope

One of the big challenges of operating a healthy API management program is achieving the proper level of central control. And, to make it even more challenging, the proper level changes as the program matures.

Early in the program, it makes sense to focus on the details of designing the API directly. In cases where APIs are in their infancy, these design details might come directly from the team creating the API—they look at existing programs "in the wild," adopt tooling and libraries that make sense for the style of API they plan to create, and go ahead and implement that API.

In this "early-stage" API program everything is new; all problems are encountered (and solved) for the first time. These initial experiences often end up being chronicled as the company's "API Best Practices" or company guidelines, etc. And they make sense for a small team working on a few APIs for the very first time. However, those initial guidelines may turn out to be incomplete.

As the number of teams working on APIs at the company grows, so does the variety of styles, experiences, and points of view. It gets more difficult to maintain consistency across all the teams—and not just because some teams are not adhering to the published company guidelines. It may be that a new team is working with a different set of off-the-shelf products that constrain their ability to follow the initial guidelines. Maybe they don't work in an event-streaming environment and are supporting XML-based call-and-response-style APIs. They need guidance, of course, but it needs to fit *their* domain, *their* customers' needs.

There are certainly some guidelines that all teams need to share, but that guidance needs to fit their problem domains as well as their API customers' needs. As your community widens, your diversity increases, and it is essential that you don't make the mistake of trying to eliminate that diversity. This is where your lever of control needs to move from giving *orders* (e.g., "All APIs MUST use the following URL patterns…") to giving *guidance* (e.g., "APIs running over HTTP SHOULD use one of the following URL templates…").

In other words, as your program's scope expands, your collection of guidelines needs to expand appropriately. This is especially important for global enterprises where local culture, language, and history play an important role in the way teams think, create, and solve problems.

And that leads us to the next key element: scale.

Scale

Another big challenge for creating and maintaining a healthy API management program is dealing with changes in scale over time. As we discussed in the previous section, growing the number of teams and the number of APIs created by those teams can be a challenge. The processes needed to monitor and manage the APIs at runtime will also change as the system matures. The tooling needed to keep track of a handful of APIs all built by the same team in a single physical location is very different from the tooling needed to keep track of hundreds or thousands of API entry points scattered across multiple time zones and countries.

In this book we talk about this aspect of API management as the "landscape." As your program scales up, you need to be able to keep an eye on lots of processes by lots of teams in lots of locations. You'll rely more on monitoring runtime behavior to get a sense of how healthy your system is at any one moment. In the second part of this book we'll explore how the notion of managing the API landscape can help you figure out which elements deserve your focus and what tools and processes can help you keep a handle on your growing API platform.

API landscapes pose a new set of challenges. The processes you use to design, implement, and maintain a single API are not always the same when you need to scale your ecosystem. This is basically a game of numbers: the more APIs you have in your system, the more likely it is that they will interact with each other, and that increases the likelihood that some of those interactions will result in unexpected behavior (or "errors"). This is the way large systems work—there are more interactions and more unexpected results. Trying to remove these unexpected results only gets you part of the way. You can't eliminate all the bugs.

And that leads to the third challenge most growing API programs encounter: how can you reduce unexpected changes by applying the appropriate level of standards within your API program?

Standards

One of the key shifts that happen when you begin managing at the landscape level instead of the API level is in the power of standards in providing consistent guidance for teams designing, implementing, and deploying APIs in your organization.

As groups grow larger—including the group of teams responsible for your organization's APIs—there is a coordination cost that is incurred (see "Decisions" on page 16). The growing scale requires a change in scope. And a key way to deal with this challenge is to rely more on general standards instead of specific design constraints.

For example, one of the reasons the World Wide Web has been able to continue to function well since its inception in 1990 is that its designers decided early on to rely on general standards that apply to all types of software platforms and languages

instead of creating tightly focused implementation guidance based on any single language or framework. This allows creative teams to invent new languages, architecture patterns, and even runtime frameworks without breaking any existing implementations.

A common thread that runs through the long-lived standards that have helped the web continue to be successful is the focus on standardizing the *interaction* between components and systems. Instead of standardizing the way components are implemented internally (e.g., use this library, this data model, etc.), web standards aim to make it easy for parties to understand each other over the wire. Similarly, as your API program grows to a more mature level, the guidance you provide to your API community needs to focus more on general interaction standards instead of specific implementation details.

This can be a tough transition to make, but it is essential to moving up the ladder to a healthy API landscape where it is possible for teams to build APIs that can easily interact with both the existing and the future APIs in your system.

Managing the API Landscape

As mentioned at the start of this chapter, there are two key challenges in the API management space: managing the life of a single API and managing the landscape of all the APIs. In our visits to many companies and our research into API management in general, we find many versions of the "managing a single API" story. There are lots of "lifecycles" and "maturity models" out there that provide solid advice on identifying and mitigating the challenges of designing, building, and deploying an API. But we have not found much in the way of guidance when it comes to an ecosystem (we call it a landscape) of APIs.

Landscapes have their own challenges; their own behaviors and tendencies. What you need to take into account when you design a single API is not the same as what you must consider when you have to support tens, hundreds, or even thousands of APIs. There are new challenges *at scale* that happen in an ecosystem—things that don't happen for a single instance or implementation of an API. We dive deep into the API landscape later in the book, but we want to point out three ways in which API landscapes present unique challenges for API management here at the start of the book:

- Scaling technology
- Scaling teams
- Scaling governance

Let's take a moment to review each of these aspects of API management with regard to landscapes.

Technology

When you are first starting your API program, there are a series of technical decisions to make that will affect all your APIs. The fact that "all" your APIs is just a small set at this point is not important. What is important is that you have a consistent set of tools and technologies that you can rely upon as you build out your initial API program. As you'll see when we get into the details of the API lifecycle and API maturity, API programs are not cheap, and you need to carefully monitor your investments of time and energy into activities that will have a high impact on your API's success without risking lots of capital too early in the process. This usually means selecting and supporting a small set of tools and providing a very clear, often detailed set of guidance documents to help your API teams design and build APIs that both solve your business problems and work well together. In other words, you can gain early wins by limiting your technical scope.

This works well at the start, for all the reasons we've mentioned. However, as your program scales up in volume and its scope widens (e.g., more teams building more APIs to serve more business domains in more locations, etc.). the challenges also change. As you grow your API program, relying on a limited set of tools and technologies can become one of the key things that slow you down. While at the beginning, when you had a small set of teams, limiting choices made things move faster, placing limits on a large set of teams is a costly and risky enterprise. This is especially true if you start to add teams in geographically distant locations and/or when you embrace new business units or acquire new companies to add to your API landscape. At this point variety becomes a much more important success driver for your ecosystem.

So, an important part of managing technology for API landscapes is identifying when the landscape has grown large enough to start increasing the variety of technologies instead of restricting them. Some of this has to do with the realities of existing implementations. If your API landscape needs to support your organization's existing SOAP-over-TCP/IP services, you can't require all these services to use the same URL guidance you created for your greenfield CRUD-over-HTTP APIs. The same goes for creating services for new event-driven Angular implementations or the legacy remote procedure call (RPC) implementations.

A wider scope means more technological variety in your landscape.

Teams

Technology is not the only aspect of API management that surfaces a new set of challenges as the program grows. The makeup of the teams themselves needs to adjust as the landscape changes, too. Again, at the start of your API program, you can operate with just a few committed individuals doing—for the most part—everything. This is when you hear names like "full-stack developer," or "MEAN" [MongoDB, Express.js, Angular.js, Node.js] developer or some other variation on the idea of a single devel-

oper that has skills for all aspects of your API program. You also may hear lots of talk about "startup teams" or "self-contained teams." It all boils down to having all the skills you need in one team.

This makes sense when your APIs are few and they all are designed and implemented using the same set of tools (see "Technology" on page 11). But as the scale and scope of your API program grows, the number of skills required to build and maintain your APIs gross, too. You can no longer expect each API team to consist of a set number of people with skills in design, database, backend, frontend, testing, and deployment. You might have a team whose job is to design and build a data-centric dashboard interface used by a wide range of other teams. Their skills may, for example, need to cover all the data formats used and tools needed to collect that data. Or you might have a team whose primary job is to build mobile apps that use a single technology like GraphQL or some other query-centric library. As technological variety grows, your teams may need to become more specialized. We'll have a chance to explore this in detail later in the book.

Another way in which teams will need to change as your API landscape grows is the way in which they participate in day-to-day decision-making processes. When you have a small number of teams and their experience is not very deep, it can make sense to centralize the decision making to a single, guiding group. In large organizations this is often the Enterprise Architecture group or something with a similar name. This works at smaller scales and scopes but becomes a big problem as your ecosystem becomes less homogeneous and more wide-ranging. As tech gets more involved, a single team is unlikely to be able to keep up with the details of each tool and framework. And as you add more and more teams, decision making itself needs to be distributed; a central committee rarely understands the realities of the day-to-day operations in a global enterprise.

The solution is to break down the decision-making process into what we call decision elements (see "The Elements of a Decision" on page 27) and distribute those elements to the proper levels within your company. A growing ecosystem means teams need to become more specialized on a technical level and more responsible at the decision-making level.

Governance

The last area that we want to touch on in regards to the challenge of API landscapes is the general approach to *governance* of your API program. Again, as in other cases mentioned here, it is our observation that the role and levers of governance will change as your ecosystem grows. New challenges appear, and old methods are not as effective as they were in the past. In fact, especially at the enterprise level, sticking to old governance models can slow or even stall the success of your APIs.

Just as in any area of leadership, when the scope and scale are limited, an approach based on providing direct guidance can be the most effective. This is often true not just for small teams, but also for *new* teams. When there is not a lot of operating experience, the quickest way to success is to provide that experience in the form of detailed guidance and/or process documents. For example, we find early API program governance often takes the form of multipage process documents that explain specific tasks: how to design the URLs for an API, or which names are valid for URLs, or where the version number must appear in an HTTP header. Providing clear guidelines with few options makes it hard for developers to stray from the approved way of implementing your APIs.

But again, as your program grows, as you add more teams and support more business domains, the sheer size and scope of the community begin to make it very difficult to maintain a single guidance document that applies to all teams. And while it is possible to "farm out" the job of writing and maintaining detailed process documents for the entire enterprise, it is usually not a good idea anyway—as we mentioned in "Technology" on page 11, technology variety becomes a strength in a large ecosystem, and attempting to rein it in at the enterprise governance level can slow your program's progress.

That's why as your API landscape expands, your governance documents need to change in tone from offering direct process instructions toward providing general principles. For example, instead of writing up details on what constitutes a valid URL for your company, it is better to point developers to the Internet Engineering Task Force's guidelines on URI design and ownership (RFC 7320) and provide general guidance on how to apply this public standard within your organization. Another great example of this kind of *principled guidance* can be found in most UI/UX guidelines, such as the "10 Usability Heuristics for User Interface Design" from the Nielsen Norman Group. These kinds of documents provide lots of options and rationales for using one UI pattern over another. They offer developers and designers guidance on why and when to use something instead of simply setting requirements for them to follow.

Finally, for very large organizations, and especially companies that operate in multiple locations and time zones, governance needs to move from distributing principles to collecting advice. This essentially reverses the typical central governance model. Instead of telling teams what to do, the primary role of the central governance committee becomes to collect experience information from the field, find correlations, and echo back guidance that reflects "best practice" within the wider organization.

So, as your API landscape grows, your API governance model needs to move from providing direct advice to presenting general principles to collecting and sharing practices from experienced teams within your company. As we'll see in Chapter 2,

there are a handful of principles and practices you can leverage in order to create the kind of governance model that works for your company.

Summary

In this opening chapter, we touched on a number of important aspects of API management that appear within this book. We acknowledged that while APIs continue to be a driving force, barely 50% of companies surveyed are confident of their ability to properly manage these APIs. We also clarified the many uses of the term "API" and how these different uses may make it harder to provide a consistent governance model for your program.

And, most importantly, we introduced the notion that managing "an API" is very different from managing your "API landscape." In the first case, you can rely on API-as-a-Product, API lifecycle, and API maturity models. Change management for APIs is also very much focused on this "an API" way of thinking. But this is just part of the story.

Next, we discussed managing your API landscape—the entire API ecosystem within your organization. Managing a growing landscape of APIs takes a different set of skills and metrics; skills in dealing with variety, volume, volatility, vulnerability, and several other aspects. In fact, these landscape aspects all affect the API lifecycle, and we'll review them in detail later in this book.

Finally, we pointed out that even the way you make your decisions about your API program will need to change over time. As your system grows, you need to distribute decision making just as you distribute IT elements like data storage, computational power, security, and other parts of your company's infrastructure.

With this introduction as a background, let's start by focusing on the notion of *governance* and how you can use decision-making and the distribution of decisions as a primary element in your overall API management approach.

CHAPTER 2
API Governance

> Hey, a rule is a rule, and let's face it, without rules there's chaos.
>
> —Cosmo Kramer

Governance isn't the kind of thing people get excited about. It's also a topic that carries a bit of emotional baggage. After all, few people want to be governed and most people have had bad experiences with poorly designed governance policies and nonsensical rules. Bad governance (like bad design) makes life harder. But in our experience, it's difficult to talk about API management without addressing it.

In fact, we'll go as far as saying that it's *impossible* to manage your APIs without governing them.

Sometimes, API governance happens in a company, but the term "governance" is never used. That's perfectly fine. Names matter, and in some organizations governance implies a desire to be highly centralized and authoritative. That can run counter to a culture that embraces decentralization and worker empowerment, so it makes sense that governance is a bad word in those kinds of places. No matter what it's called, even in this type of decentralized work culture, some form of decision governance is taking place—but it will probably look radically different from the governance system at a more traditional, top-down organization.

The question "Should you govern your APIs?" isn't very interesting, because in our opinion, the answer is always yes. Instead, ask yourself: "Which decisions need to be governed?" and "Where should that governance happen?" Deciding on the answers to these types of questions is the work of designing a governance system. Different styles of governance can produce vastly different working cultures, productivity rates, product quality, and strategic value. You'll need to design a system that works for you. Our goal in this chapter is to give you the building blocks to do that.

We'll start by exploring the three foundational elements of good API governance: decisions, governance, and complexity. Armed with this understanding, we'll take a closer look at how decisions can actually be distributed in your company and how that impacts the work you do. That means taking a closer look at centralization, decentralization, and the elements of what makes a decision. Finally, we'll take a look at what it means to build a governance system and take a tour of three governance styles.

Governance is a core part of API management, and the concepts we introduce in this chapter will be built upon throughout the rest of this book. So, it's worthwhile to spend some time understanding what API governance really means and how it can help you build a better API management system.

Understanding API Governance

Technology work is the work of making decisions—lots of decisions, in fact. Some of those decisions are vitally important, while others are trivial. All this decision making is the reason that we can say a technology team's work is *knowledge work*. The key skill for a knowledge worker is to make many high-quality decisions, over and over again. That's a fairly obvious concept, but also one that's easy to forget when you're managing APIs.

No matter which technologies you introduce, how you design your architecture, or which companies you choose to partner with, it's the decision-making abilities of everyone involved that dictate the fate of your business. That's why governance matters. You need to shape all of those decisions in a way that helps you achieve your organizational goals.

That's harder to do than it sounds. To give yourself a better chance of success you'll need a better understanding of the foundational concepts of governance and how they relate to each other. Let's start by taking a quick look at API decisions.

Decisions

Your work and the work that many people in your organization perform is primarily the work of making decisions. That's why governance is so important. If you can make better decisions as a group you can produce better results. But don't forget that those decisions aren't just choices about technology—you'll need to make a broad range of decisions in the API domain. Consider the following list of choices an API team might need to make:

1. Should our API's URI be `/payments` or `/PaymentCollection`?
2. Which cloud provider should we host our API in?
3. We have two customer information APIs—which one do we retire?

4. Who's going to be on the development team?
5. What should I name this Java variable?

From this short list of decisions we can make a few observations. First, API management choices span a wide spectrum of concerns and people—making those choices will require a lot of coordination between people and teams. Second, the individual choices people make have different levels of impact—the choice of a cloud provider is likely to affect your API management strategy much more than the name of a Java variable. Third, small choices can have a big impact at scale—if 10,000 Java variables are named poorly, the maintainability of your API implementations will suffer greatly.

All of these choices, spanning multiple domains, being made in coordination and at scale, need to come together to produce the best result. That's a big and messy job. Later in this chapter we'll pick this problem apart and give you some guidance for shaping your decision system. But first, let's take a closer look at what it means to *govern* these decisions and why governance is so important.

Governing Decisions

If you've ever worked on a small project by yourself, you know that the success or failure of that work relies solely on you. If you make good decisions consistently, you can make something good happen. A single, highly skilled programmer can produce some amazing things. But this way of working doesn't scale very well. When the thing you produce starts getting used, the demand for more changes and more features grows. That means you need to make many more decisions in a shorter space of time—which means you'll need more decision makers. Scaling decision making like this requires care. You can't afford for the quality of your decisions to drop just because there are more people making them.

That's where governance comes in. *Governance is the process of managing decision making and decision implementation.* Notice that we aren't saying that governance is about control or authority. Governance isn't about power. It's about improving the decision-making quality of your people. In the API domain, high-quality governance means producing APIs that help your organization succeed. You may need some level of control and authority to achieve that, but it's not the goal.

You can apply governance to your API work in lots of different ways. For example, you could introduce a policy that all API teams in your company must use the same standardized technology stack. Or you could introduce a policy that all APIs need to pass a set of standardized quality measures before they can be launched. One policy is more heavy-handed than the other, but both policies might achieve similar results. In practice, you'll be managing lots of different types of decisions at the same time and

your governance system is going to be a mix of many different constraints, rewards, policies, and processes.

Keep in mind that governance always has a cost. Constraints need to be communicated, enforced, and maintained. Rewards that shape decision-making behavior need to be kept valuable and attractive to your audience. Standards, policies, and processes need to be documented, taught, and kept up to date. On top of that, constant information gathering is needed to observe the impact of all of this on the system. You may even need to hire more people just to support your governance efforts.

Beyond those general costs of maintaining the machinery of governance, there are also the hidden costs of applying governance to your system. These are the impact costs that come up when you actually start governing the system. For example, if you mandate the technology stack that all developers must use, what is the organizational cost in terms of technological innovation? Also, what will be the cost to employee happiness? Will it become more difficult to attract good talent?

It turns out that these kinds of costs are difficult to predict. That's because in reality you're governing a complex system of people, processes, and technology. To govern an API system, you'll first need to learn what it takes to manage a complex system in general.

Governing Complex Systems

The good news is that you don't need to control every single decision in your organization to get great results from governance. The bad news is that you'll need to figure out which decisions you *will* need to control in order to get those good results. That's not an easy problem to solve, and you won't find a definitive answer in this book. That's because it's impossible to give you an answer that will fit your unique context and goal.

If all you wanted to do was bake a sponge cake, we could give you a pretty definitive recipe for making one. We'd tell you how much flour and how many eggs you'd need and what temperature to set your oven at. We could even tell you exactly how to check if the cake is done. That's because there is very little variability in modern baking. The ingredients are reasonably consistent no matter where you purchase them from. Ovens are designed to cook at specific, standardized temperatures. Most importantly, the goal is the same—a specific kind of cake.

But you aren't making a cake, and this isn't a recipe book. You'll need to deal with an incredible amount of variability. For example, the people in your company will have varying levels of decision-making talent. The regulatory constraints you operate in wiill be unique to your industry and location. You'll also be serving your own dynamically changing consumer market with its own consumer culture. On top of all that, your organizational goals and strategy will be entirely unique to you.

All this variability makes it tough to prescribe a single correct "recipe" for API governance. To make things even harder, there's also the small problem of knock-on effects. Every time you introduce a rule, or create a new standard, or apply any form of governance, you'll have to deal with unintended consequences. That's because all the various parts of your organization are intertwined and connected. For example, to improve the consistency and quality of your API code, you could introduce a standard technology stack. That new stack might result in bigger code packages as programmers start adding more libraries and frameworks. And that could result in a change to the deployment process because the bigger deployment packages can't be supported with the existing system.

With the right information, maybe you could predict and prevent that outcome. But it's impossible to do that for every possible eventuality, especially within a reasonable amount of time. Instead, you'll need to accept the fact that you are working with a complex adaptive system. As it turns out, this is a feature, not a bug. You'll just need to figure out how to use it to your advantage.

Complex adaptive systems

When we say that your organization is a complex adaptive system, we mean:

- It has lots of parts that are interdependent (e.g., people, technologies, process, culture).
- Those parts can change their behavior and adapt to system changes (e.g., changing deployment practices when containerization is introduced).

The universe is full of these kinds of systems, and the study of complexity has become an established scientific discipline. Even you yourself are a complex adaptive system. You might think of yourself as a single unit—a self—but "self" is just an abstraction. In reality, you're a collection of organic cells, albeit a collection of cells that is capable of amazing feats: thinking, moving, sensing and reacting to external events as an emergent whole "being." At the cellular level, your individual cells are specialized; old, dying cells are replaced and groups of cells work together to produce big impacts in your body. The complexity of the biological system that you are composed of makes your body highly resilient and adaptable. You're probably not immortal, but you're equally likely to be able to withstand massive amounts of environmental change and even bodily damage, thanks to your complex biological system.

Usually, when we talk about "systems" in technology we focus on software systems and network-based architecture. Those kinds of systems can definitely grow to be complex. For example, the web is a perfect example of system-level complexity and emergence. A network of individual servers run independently, but through their dependencies and interconnections produce an emergent whole that we call "the web." But most of that software isn't really *adaptive*.

The API software you write today is pretty dumb. That doesn't mean that your code is of poor quality or that it doesn't do the job it was designed for. In fact, it's just the opposite; most of the APIs you implement will do exactly what they're supposed to do. And that's the problem. You can make an API that's smart enough to adapt to a changing traffic pattern or an increasing number of errors, but it's impractical to make one that can add a new feature without human intervention, or correct a complex bug by itself, or update its own documentation to make it easier to learn.

Now, all of that might change in the future. But as it stands today, it's your people that drive the behavior of your software system. The good news is that people are very good at adapting (especially when compared to software). Your API organization is a complex adaptive system. All of the individual people in your organization make many *local* decisions, sometimes collectively and sometimes individually. When all those decisions happen at scale and over time, a system emerges. Just like your body, that system is capable of adapting to a lot of change.

But working with a complex system requires a special kind of approach. It's difficult to predict the impact of changes in a complex system—making a change to one part of your system can lead to unintended consequences in another part. That's because the people in your organization are constantly adapting to the changing environment. For example, introducing a rule that deploying software in "containers" is forbidden would have a wide-reaching impact, affecting software design, hiring, deployment processes, and culture.

All of this means that you can't get the outputs you want from the system by implementing large changes and waiting for results. Instead, you'll need to "nudge" the system by making smaller changes and assessing their impact. It requires an approach of continuous adjustment and improvement, in the same way you might tend to a garden, pruning branches, planting seeds, and watering while continuously observing and adjusting your approach.

Governing Decisions

In the last section we introduced the concept of governing decisions inside a complex system. Hopefully, that's helped you to understand a fundamental rule for API governance: if you want your governance system to be effective, you'll need to get better at managing decisions. We think one of the best ways to do that is to focus on where decisions are happening and who is making them. It turns out that there isn't a single best way to map those decisions out. For example, consider how API design governance could be handled in two different fictional companies:

Company A: Pendant Software
: At Pendant Software, all API teams are provided with access to the "Pendant Guidelines for API Design" e-book. These guidelines are published quarterly by

Pendant's API Center of Excellence and Enablement—a small team of API experts working inside the company. The guidelines contain highly prescriptive and very specific rules for designing APIs. All teams are expected to adhere to the guidelines and APIs are automatically tested for conformance before they can be published.

As a result of these policies, Pendant has been able to publish a set of industry-leading, highly consistent APIs that developers rate very favorably. These APIs have helped Pendant differentiate itself from competitors in the marketplace.

Company B: Vandelay Insurance

At Vandelay, API teams are given the company's business goals and expected results for their API products. These goals and results are defined by the executive teams and are updated regularly. Each API team has the freedom to address an overall business goal in the manner they choose and multiple teams can pursue the same goal. API teams can design and implement APIs however they like, but every product must adhere to Vandelay's enterprise measurement and monitoring standards. The standards are defined by Vandelay's System Commune, a group made up of individuals from each of the API teams who join voluntarily and define the set of standards that everyone needs to follow.

As a result of these policies, Vandelay has been able to build a highly innovative, adaptive API architecture. This API system has enabled Vandelay to outmaneuver its competition with innovative business practices that can be delivered very quickly in its technology platform.

In our fictional case studies, both Pendant and Vandelay were wildly successful in their management of decision making. But the way they governed their work was incredibly different. Pendant found success with a highly centralized, authoritative approach, while Vandelay preferred a results-oriented method. Neither approach is "correct," and both styles of governance have merit.

To govern decisions effectively, you'll need to address three key questions:

1. Which decisions should be managed?
2. Where should those decisions be made (and by whom)?
3. How will the system be impacted by your decision management strategy?

Later in the book we'll dig into the questions of which decisions should be managed and how those decisions will impact your system. For now, we'll focus on the second question of where in the system the most important decisions should be made. To help you address decision distribution, we are going to dig deeper into the subject of governing a decision. We'll tackle the trade-off between centralized and decentralized decision making and we'll take a closer look at what it means to distribute a decision.

Centralization and Decentralization

Earlier in this chapter, we introduced the concept of a complex adaptive system and we used the human body as an example. These kinds of systems abound in nature, and you are surrounded by them. For example, the ecosystem of a small pond can be thought of as a complex adaptive system. It continues to survive thanks to the activities and interdependence of the animals and vegetation that live in it. The ecosystem adapts to changing conditions thanks to the localized decision making of each of these living things.

But the pond doesn't have a manager, and there is no evidence that the frogs, snakes, and fish hold quarterly management meetings. Instead, each agent in the system makes individual decisions and exhibits individual behaviors. Taken together these individual decisions and actions form a collective, emergent whole that can survive even as individual parts of the system change or appear and disappear over time. Like most of the natural world, the pond system succeeds because system-level decisions are *decentralized* and *distributed*.

As we established earlier, your organization is also a complex adaptive system. It's a product of all the collective individual decisions made by your employees. Just like in a human body or a pond ecosystem, if you were to allow individual workers to have complete freedom and autonomy, the organization as a whole would become more resilient and adaptive. You'd have a bossless, decentralized organization that could find its way thanks to the individual decisions of its employees (see Figure 2-1).

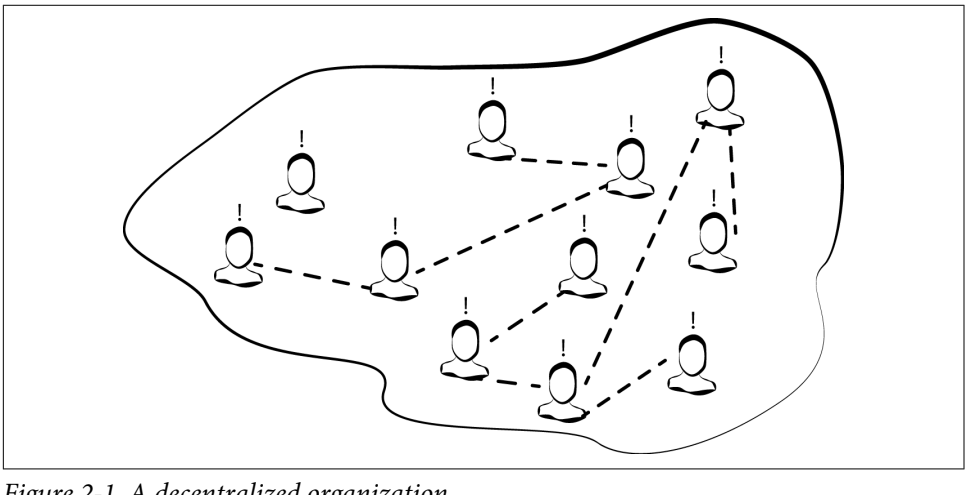


Figure 2-1. A decentralized organization

You could do this, but you might run into some problems, primarily because it's difficult to succeed with a free-market organization in exactly the same way that complex systems succeed in nature. The biosystem of a pond is directed by the hand of

natural selection. Every agent in the system has become optimized for the survival of its species. There's no system-level goal beyond survival. On top of that, in nature it's normal for systems to fail. For example, if an invasive species is introduced the entire pond system might die. In the natural world, that can be OK because something else might take its place—the system as a whole remains resilient.

However, businesses leaders don't respond well to this level of uncertainty and lack of control. Chances are you'll need to steer your system toward specific goals that go beyond survival. Also, it's likely that you aren't willing to risk letting your company die for the sake of a better company taking its place. You'll almost certainly want to reduce the risk that any individual agent can destroy the whole company because of a bad decision. That means you'll need to reduce decision-making freedom for individuals and introduce some accountability. One way of doing that is to introduce decision *centralization* (Figure 2-2).

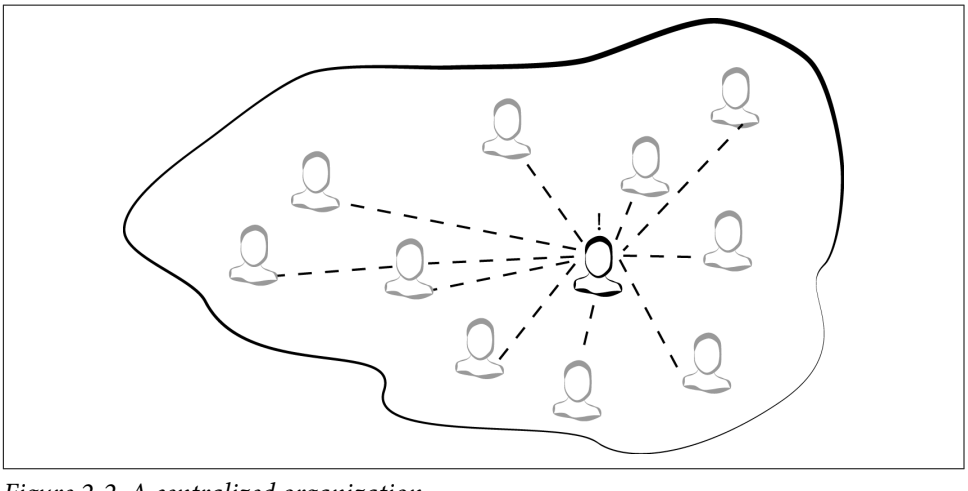


Figure 2-2. A centralized organization

By this, we mean that decision making is constrained to a particular person or team in your organization. That centralized team makes a decision that the rest of the company will need to adhere to. Decentralization is the opposite case: individual teams can make decisions that only they need to adhere to.

The truth is that there is no perfectly centralized or perfectly decentralized organization. Instead, different types of decisions are distributed within the organization in different ways—some are more centralized while others are more decentralized. You'll need to decide how to distribute the decisions that impact your system the most. So, which ones should be more centralized and which ones should be more decentralized?

Remember, a primary goal of governing decisions is to help your organization succeed and survive. What that means is entirely dependent on your business context, but generally speaking it means that decisions need to be timely enough to enable business agility and of sufficient quality to improve the business (or at the very least avoid damaging it). There are three factors that impact the ability to make decisions:

Availability and accuracy of information
: It's really difficult to make a good decision if you base it on information that is incorrect or missing. That could mean being misled about the goal or context of the decision, but it could also mean not knowing what the decision's impact will be on the system. Most of the time, we assume the responsibility for gathering decision-making information rests at the feet of the decision makers. But for the purposes of distributing decisions, we also need to think about how centralizing or decentralizing a decision affects the information that's available.

Decision-making talent
: Generally speaking, decision quality improves if the decision maker is good at making high-quality decisions. Or, in simpler language—highly talented people with lots of experience will make better decisions than less-talented people with no experience. When it comes to distributing decision making, the challenge is to also distribute your talent in a way that helps you the most.

Coordination costs
: Complex decisions can't be made in a timely manner unless the decision making is shared. But whenever you share decision-making work you'll incur a coordination cost. If that coordination cost grows too high, you won't be able to make decisions quickly enough. Centralization and decentralization of decisions can have a big impact on coordination costs.

Thinking about decisions in terms of these factors will help you decide when a decision should be centralized or decentralized. To help you understand how to do that, we'll take a look at it from two perspectives: scope of optimization and scale of operation. Let's start by digging into scope and its relationship with decision-making information.

Scope of optimization

The big difference between a centralized decision and a decentralized decision has to do with their scope. When you make a centralized decision, you are making it for the entire organization. So, your scope for the decision includes the whole system and your goal is to make a decision that improves that system. Another way of saying this is that the decision you are making is meant to *optimize the system scope*. For example, a centralized team might decide on a development methodology for the entire company to follow. The same team might also make decisions about which

APIs in the system should be retired. Both of these decisions would be made with the goal of doing what's best for the entire system.

Conversely, the primary characteristic of a decentralized decision is that it is *optimized for a local scope*. When you are optimizing for the local scope, you are making a decision that will improve your local context—the set of information that pertains only to your local situation. While your decision might have an impact on the wider system, your goal is to improve your local results. For example, an API team can make a local decision to use a waterfall development process because they're sharing the work with an external company that insists on it.

The great thing about decentralized decision making is that it can help you make big gains in efficiency, innovation, and agility for your business overall. That's because decentralized decision makers are able to limit their scope of information to a local context that they understand. This means they can form a decision based on accurate information about their own problem space, which helps them produce better decisions. For any modern business that is trying to succeed with a strategy of agility and innovation, the decentralized decision pattern should be the default approach.

However, making decisions that focus only on optimizing the local scope can cause problems, particularly if those decisions have the potential to impact the system negatively and in irreversible ways. When Amazon CEO Jeff Bezos talks about the impact of decisions, he splits them into two types: "type 1" decisions that can be easily reversed if they are wrong and "type 2" decisions that are near impossible to recover from. For example, a lot of big companies choose to centralize decisions about API security configuration to prevent a local optimization from creating a system vulnerability.

Beyond dangers to the system, there are times when system-level consistency is more valuable than local optimization. For example, an individual API team might choose an API style that makes the most sense for their problem domain. But if every API team chooses a different API style, the job of learning to use each API becomes more difficult due to a lack of consistency, especially when many APIs need to be used to accomplish a single task. In this case, optimizing the API style decision for the system scope might be better.

You'll need to think about the scope of optimization carefully when you plan where a decision should happen. If a decision has the potential to impact your system in an irreversible way, start by centralizing it so that it can be optimized for system scope. If decision quality could benefit from the local context of information, start by decentralizing it. If decentralizing a decision could result in unacceptable inconsistency at the system level, consider centralizing it.

Scale of operation

If you had unlimited resources for making good decisions, you'd only need to think about scope for decision making. But you don't. So, in addition to scope, you'll need to think about the scale of decisions being made. That's because if there is a bigger decision demand, there will be more pressure on your decision-making talent supply and an upward pressure on your coordination costs. If you want your API work to scale as your organization grows, you'll need to plan your decision distribution pattern carefully.

Decentralizing a decision creates a big talent demand when you are operating at scale. When you decentralize a decision you are distributing it to more than one team. If you want all of those decisions to be high quality, you'll need to fill each of those teams with talented decision makers. If you can't afford to do that, you'll end up generating lots of bad decisions. So, it's worthwhile to hire the best decision-makers you can for every decision making position in your company.

Unfortunately, hiring good people isn't an industry secret. There are a limited number of talented and experienced people available and a lot of companies competing to hire them. Some companies are willing to spend whatever it takes to make sure that they get the best talent in the world. If you are lucky enough to be in that situation, you can decentralize more of your decisions because you have the talent to make them. Otherwise, you'll need to be more pragmatic with your distribution decisions.

If your supply of top-level, "grade A" decision-making talent is limited, you may choose to pool that talent together and centralize the most important decisions to that group of people. That way, you have a greater chance of producing better decisions, faster. But an increasing scale of decision demand wreaks havoc on this model too, because as the demand for decision making grows, the centralized team will need to grow along with it. As the team grows, so too will the cost of coordinated decision making. No matter how talented the people are, the cost of coordinating a decision grows as you add more people. Eventually you'll reach a number that makes it impossible to reach decisions affordably.

All of this means that decision distribution will involve a lot of trade-offs. If the decision is highly impactful, like the "type 1" decisions that Jeff Bezos describes, you'll need to centralize it and pay the price of lower decision-making throughput. Conversely, if speed and local optimization are most important, you can decentralize the decision and either pay for better people or accept the net reduction in quality of decisions.

That said, there is a way to manage this trade-off in a more nuanced and flexible way. It involves distributing the *parts* of the decision instead of the entre decision itself, and it's what we are going to focus on in the next section.

The Elements of a Decision

It's difficult to distribute a decision in the way we've described so far because it's a bit of an all-or-nothing affair. Do you let your teams decide which development method they want to use, or do you choose one and make every team use it? Do you let the teams decide when their API should retire, or do you take the choice away from them completely? In reality, governance requires more nuance. In this section, we'll explore a way of distributing decisions with more flexibility by breaking them up into pieces.

Instead of distributing the entire decision, you can distribute parts of the decision. That way you can get the benefits of system-level optimization along with highly contextual local optimization at the same time. Some parts of a decision can be centralized while other parts are decentralized. To help you accomplish distribution with this kind of precision, we've broken down API decisions into the six *decision elements* you'll need to distribute (see Figure 2-3):

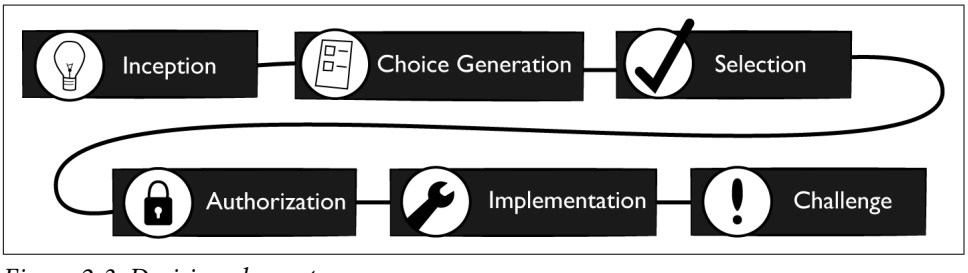


Figure 2-3. Decision elements

This isn't meant to be an authoritative, universal model for decision making. Instead, it's a model that we've developed to distinguish the parts of a decision that have the biggest impact on a system when they are either centralized or decentralized. These parts are based on the various five-, six-, and seven-step models of decision-making that abound in the business management domain. Although the steps we'll describe could be applied to a decision made by a single person, they're most useful when we are talking about decisions made in coordination between a group of people.

Let's start by taking a look at how distributing the inception of a decision impacts your system.

Inception

Every decision happens because someone thinks that decision needed to be made. It means that someone has identified that a problem or opportunity exists with more than one possible solution. Sometimes this is obvious, but in many cases spotting a decision-making opportunity requires talent and expertise. You'll need to think

about which decisions will naturally ignite on their own and which ones will need special handling to make sure that they happen.

Kicking off decisions about API work happens naturally in the course of day-to-day problem solving. For example, choosing which database to use for storing persistent data would be a difficult decision for a typical implementer to ignore. The decision happens because the work can't continue without it. But there will also be situations where you'll need to force inception to happen. This is usually for one of two reasons:

Habitualized decision making
: Over time, if a team makes the same decision over and over, the decision may disappear. That is, the possibilities are no longer considered and instead an assumption is made that work will continue in the same way it always has. For example, if every API implementation is written in the Java programming language, it may not occur to anyone to consider a different choice of language.

Decision blindness
: Sometimes, teams will miss opportunities to make impactful decisions. This can happen because of habit, but also because of limited information, experience, or talent. For example, a team may focus on the choice of which database to use for storage, but fail to identify that the API could be designed in a way that doesn't require persistent storage.

Not every decision needs to happen, and it's perfectly fine for decisions to be missed or for a cultural habit to make them implicit. It's only a problem if not making a decision negatively impacts the results you are getting from your APIs. Arbitrarily demanding that more decisions happen could have a nightmarish impact on productivity. Instead, the role of API governance is to generate more of the decisions that will lead to optimal results and less of the decisions that will provide little value.

Choice generation

It's hard to choose if you don't know your options, and that's what this element is all about. Choice generation is the work of identifying the choices to choose from.

If you're making a decision in a domain you have a lot of experience in, generating choices can be pretty easy. But if there are lots of unknowns, you'll need to spend more time identifying the possibilities. For example, an experienced C programmer already has a good idea of their options when they are deciding on a loop structure, but a beginner will probably need to do some investigation to learn that they can use a `for` loop or a `while` loop and the differences between the two.

Even if you know a domain fairly well, you'll probably spend more time on choice generation if the cost and impact of the decision are very high. For example, you may have intimate knowledge of the different cloud hosting environments, but will still perform your due diligence of research when it comes time to sign a contract with

one of them. Are there new vendors available that you didn't know about? Are the prices and terms still the same as you remember?

From a governance perspective, choice generation is important because it's where the *boundaries* of decision making are set. This is especially useful when the people coming up with the list of choices are not the same as the people making the selection. For example, you could standardize a list of possible API description formats, but let individual teams decide which format they like best. If you take this approach you'll need to be careful about the quality of the "menu" you are providing. If the choices are overly restrictive or of poor quality, you'll run into problems.

Selection

Selection is the act of choosing from the list of possible options. Selection is the heart of decision making and it's the step most people focus on, but the importance of the selection element depends a lot on the scope of choices that have been made available. If that scope is very wide, then the selection process is integral to the quality of the decision. But if that scope has been constrained to safe choices with little differentiating them, the selection step can be quick and less impactful.

Let's walk through an example of this in action. Suppose you're responsible for configuring Transport Layer Security (TLS) for your HTTP API. Part of that work includes a decision on which cipher suites (sets of cryptography algorithms) the server should support. It's an important decision because some cipher suites have become vulnerable with age, so picking the wrong ones can make your API less secure. Also, if you choose cipher suites that your users' client software doesn't understand, nobody will be able to use your API.

In one scenario, you might be given a list of all the known cipher suites and asked to select the ones that the server should support. In this case, selection would need special care. You'd probably do a lot of research and only feel comfortable making a selection once you'd gathered as much information as possible. In fact, if you didn't have a good amount of experience securing servers, you'd probably look for someone who did and ask them to make a selection for you.

But what if instead of being given the set of all possible cipher suites, you were given a curated list of them? The list of options might also include relevant information about how well supported each cipher suite is and what the known vulnerabilities are. Armed with this information you could probably make a faster choice. Equally, you're choice is likely to be safer because your decision scope is limited to choices that have been deemed safe enough to use. In this case, you'd make a decision based on what you know about the clients using the API and the sensitivity and business importance of the API.

Finally, you might be given only one choice: a single cipher suite that you must use. A single-choice decision makes selection a trivial affair—the decision has been made for

you. In this case, the quality of the decision is entirely dependent on the people who generated that choice. Hopefully it's a good fit for the specific requirements you have.

So, the importance of selection depends a lot on the scope of the choices offered. There's a bit of a trade-off at work here. If you push more of the decision-making investment into choice generation you'll spend less time on selection, and vice-versa. That has implications for how you distribute decision elements and who should be responsible for them. Whichever decision element becomes more important will require a suitably talented decision maker to make it.

It also means you can combine system scope and local scope by distributing choice *generation* and choice *selection*. For example, you can centralize the generation of development method choices based on the system context while still allowing individual teams to choose their preferred method using their local context. This happens to be a particularly useful pattern for governing large API landscapes at scale and preserving both safety and speed of change.

Authorization

Just because a choice has been selected doesn't mean the decision is done. The selection needs to be authorized before it can be realized. Authorization is the work of deciding on the validity of the selected choice. Was the right selection made? Is it implementable? Is it safe? Does it make sense in the context of other decisions that have been made?

Authorization can be implicit or explicit. When authorization is explicit it means that someone or some team must expressly authorize the decision before it can go forward. It becomes an approval step in the decision-making process. We're sure you've been involved in many decisions that required some kind of approval. For example, in many companies, workers can select their holiday time from a list of work dates, but it's up to their manager to make the final approval decision on the schedule.

Implicit authorization means that authorization happens automatically when some set of criteria has been met. Examples of this are the role of the person making the selection, the cost of the selection that was made, or adherence to a specific policy. In particular, authorization can become implicit when the person making the selection is also the person authorizing the selection. In effect, they become their own approver.

Explicit authorization is useful because it can further improve the safety of the decision. But if there are lots of decisions being made and all of them are being *centrally* authorized, then there is likely to be a reduction in decision speed. Lots of people will end up waiting for their approvals. Implicit authorization greatly increases the speed of decision making by empowering selection, but comes with greater risk.

How authorization should be distributed will be an important decision for you to make in your governance design. You'll need to consider the quality of decision makers, the business impact of bad decisions, and the amount of risk built into the choices offered. For highly sensitive decisions, you'll probably want more explicit authorization. For time-sensitive, large-scale decisions you'll need to figure out how to introduce an implicit authorization system.

Implementation

The decision process doesn't end when the choice is authorized. A decision isn't realized until someone does the work of executing or implementing the choice that has been made. Implementation is an important part of API management work. If the implementation of decisions is too slow or of poor quality, then all of your decision making is for naught.

Oftentimes a decision isn't implemented by the people who made the selection. In these cases it's important to understand what that means for the availability of accurate information gathering. For example, you might choose to introduce the hypermedia style of APIs into your landscape, but if the implementation of hypermedia APIs turns out to be too difficult for the designers and developers you'll need to reevaluate your decision. A good governance design will have to take these practicalities into account. It's no good managing decisions in a way that makes them only *theoretically* better. When you are determining the quality of decision making you'll need to include the implementability of the decision you are managing.

Challenge

Decisions aren't immutable, and each decision you make for your API management system should be open to being challenged. Oftentimes we don't consider how the decisions we make may need to be revisited, altered, even reversed in the future. Defining a challenge element allows us to plan for continuous change at the decision-making level.

For example, if you've defined a "menu" of choices for API teams to choose from, it's wise to also define a process to go "off-menu." That way you can sustain a decent level of innovation and prevent bad decisions from being made. But if everyone can challenge the decision to constrain these choices, then there aren't really any constraints. So, you'll need to identify who can challenge the decision and in what circumstances.

It's also important to allow decisions to be challenged over time. As business strategies and context change, so too should the decisions of your system. To plan for that kind of adaptability you'll need to build the challenge function into your system. That means you'll need to think about whom in your organization will have the ability to "pull the cord" and challenge an existing decision.

Decision Mapping

We now know that decisions are composed of a number of elements. Understanding that decisions have atomic elements allow us to distribute the pieces of a decision rather than the entire decision process. This turns out to be a powerful feature of organizational design and will allow you to exert greater influence over the balance of efficiency and thoroughness.

For example, a decision about the style a new API should have is an important one. In the clumsy, binary centralization versus decentralization discussion, the API management designer might consider whether the members of the API team should own the API style decision (decentralized) or a central body should maintain control of it (centralized). The advantage of distributing the decision-making power to the API teams is that each team can make the decision within a local context. The advantage of centralizing the decision within a single strategic team is that the variation in API styles is reduced and control over the quality of the style choice is maintained and controlled.

This is a difficult trade-off to make. But, if instead you distribute the *elements* of the decision, it's possible to design an API management system that lives somewhere in between these two binary options. For example, you might decide that for an API style decision, the elements of *research* and *choice generation* should be owned by a centralized, strategic API management team, while the elements of *choice selection*, *authorization*, and *implementation* are owned by the API teams themselves. In this way, you choose to sacrifice some of the innovation that comes from distributing choice generation in order to gain the benefits of a known set of API styles within the company. At the same time, distribution of the API style selection and authorization elements allows the API teams to continue to operate at speed (i.e., they do not need to ask permission in order to choose a suitable style).

To get the most out of decision mapping, you'll need to distribute decisions based on your context and goals. Let's take a look at two fairly common decision scenarios to see how decision mapping can be a useful tool.

Decision mapping example: Choosing a programming language

You've identified that the decision of which programming language to choose for API implementation is highly impactful, and you'd like to govern it. Your organization has adopted a *microservices* style of architecture, and freedom to choose the programming language for implementation has been raised as a requirement. But after running a few experiments, you've noticed that variation in programming languages makes it harder for developers to move between teams and harder for security and operations teams to support applications.

As a result, you've decided to try out the decision distribution in Table 2-1 for deciding on a programming language.

Table 2-1. Programming language decision map

Inception	Choice generation	Choice selection	Authorization	Implementation	Challenge
Centralized	Centralized	Decentralized	Decentralized	Decentralized	Decentralized

This way you constrain the programming languages to a set of choices that are optimized for the system as a whole, but allow the individual teams to optimize for their local contexts within those constraints. You've also allowed API teams to challenge the decision so that you can accommodate new language choices and changing situations.

Decision mapping example: Tool selection

Your CTO is trying to improve the level of agility and innovation of your software platform. As part of this initiative they have decided to allow API teams to choose their own software stacks for implementations, including the use of open source software. However, your procurement and legal teams have raised concerns based on legal risks and risks to supplier relationships. To get started with this cultural transition, you've decided to implement the decision map in Table 2-2 for the software stack decision on a trial basis.

Table 2-2. Tool selection decision map

Inception	Choice generation	Choice selection	Authorization	Implementation	Challenge
Decentralized	Decentralized	Decentralized	Centralized	Decentralized	Centralized

Local optimization is one of the keys to your CTO's strategy, so you chose to completely decentralize inception, choice generation, and selection. However, to reduce the system-level risk of a choice, you've mapped the authorization element to the centralized procurement and legal teams. This should work for now, but you are also aware that over time and at scale this has the potential to be a big bottleneck in your system, so you make a note to keep measuring the process and tune it accordingly.

Designing Your Governance System

We've spent a lot of time going into the details of decision distribution because we think it's a foundational concept for a governance system. But it's not the only thing you'll need to pay attention to if you want to introduce effective API governance. A good API governance system should have the following features:

- Decision distribution based on impact, scope, and scale

- Enforcement of system constraints and validation of implementation (from centralized decisions)
- Incentivization to shape decision making (for decentralized decisions)
- Adaptiveness through impact measurement and continuous improvement

It's difficult to get the advantages of decision centralization if the rest of the organization doesn't conform to the decision. That's why enforcement and validation needs to be a feature of an API governance system. We've purposefully steered away from the authoritative parts of governance so far, but ultimately you'll need to build at least some constraints into your system. Even the most decentralized organizations have rules that need to be followed. Of course, validation and enforcement will require some level of obedience. If the centralized decision-making team has no authority, the decisions will carry no weight.

If you don't have authority, you can use incentivization instead of enforcement. This is especially useful when you've decided to decentralize decisions but still want to shape the selections that are being made. For example, an architecture team could alter a deployment process so that deployment of immutable containers is made much cheaper and easier than any other type of deployment. The goal here would be to incentivize API teams who have authority over their own implementation decisions to choose containerization more often.

In truth, neither the "carrot" of incentivization nor the "stick" of enforcement is enough to steer your system on its own—you'll need to use both. Generally speaking, if a decision's authorization element has been decentralized, you'll have to use incentivization if you want to shape it. If selection and authorization have been centralized and implementation is decentralized, you'll need to make sure you've instituted some level of enforcement or validation. Table 2-3 highlights when you should enforce or incentivize a decision based on your decision mapping design.

Table 2-3. When to enforce and when to incentivize

Enforce or incentivize?	Inception	Choice generation	Choice selection	Authorization	Implementation	Challenge
Enforce		Centralized	Centralized or decentralized	Centralized or decentralized		
Incentivize		Decentralized	Decentralized	Decentralized		

No matter how you distribute your decisions or change decision-making behavior, it's crucial that you measure the impact you are having on the system itself. Ideally, your organization should have some existing process indicators and measurements that you can use to assess the impact of your changes. If there isn't anything like that, instituting organizational measurements should be one of your first priorities. Later, we'll talk about product measurement patterns for APIs. Although we'll be focusing

on API product measurement specifically, you can still use that section as an introductory guide for designing governance measurements for your system.

To help tie all this together, let's take a look at three API governance patterns. These patterns capture different approaches to API governance, but all of them use the core principles of decision distribution, enforcement, incentivization, and measurement. Keep in mind, we aren't offering you a menu—you aren't supposed to choose one of these to be *your* governance system. We are offering you these patterns as a way of illustrating how an API governance system can be implemented at a conceptual level.

For each governance pattern described, we'll identify a few key decisions and how they are mapped, how desired behaviors are enforced and incentivized, how talent is distributed, and the costs, benefits, and measures for the approach.

Governance Pattern #1: Interface Supervision

This pattern emphasizes the importance of the interface model for an API. Interface supervision centralizes all decisions related to the design of the interface in order to ensure that all interfaces are consistent, secure, and highly usable (see Table 2-4).

Table 2-4. Decision map

Decision space	Inception	Choice generation	Choice selection	Authorization	Implementation	Challenge
API design	Centralized	Centralized	Decentralized	Centralized	Decentralized	Decentralized
API implementation	Decentralized	Decentralized	Decentralized	Decentralized	Decentralized	Centralized
API deployment	Decentralized	Decentralized	Decentralized	Decentralized	Decentralized	Centralized

Enforcement and incentivization
: API implementation and deployments are reviewed by the centralized interface design team. Although teams have the freedom to make their own implementation and deployment decisions, the central team can flag and remove an API if it doesn't conform to the interface model.

Talent distribution
: Interface design talent is pooled in the central team, while programming and operations talent can be decentralized.

Costs and benefits
: The segregation of design and implementation teams means that there is a risk of making designs that are difficult or costly to implement. But this separation also benefits from a "pure design" perspective for the interface design team, which can produce more user-centric designs. At scale, there is a very high risk of a bottleneck due to the resource constraints of a centralized interface design team.

This may especially be a problem when small changes to many interfaces are required.

Impact measurements
- API usability measurements
- Product and project schedule metrics
- Implementation and operational issues

Governance Pattern #2: Machine-Driven Governance

Machine-driven governance uses the machinery of standardization and automation to constrain decision making. In this pattern, the centralized team tries to maximize control of the system with machinery, but limit the impact on decision-making throughput. This is done by only centralizing the decision space of API work (i.e., the choice generation element). Teams have the freedom to make decisions as long as they conform to the choices that have been codified into the standards (see Table 2-5).

Table 2-5. Decision map

Decision space	Inception	Choice generation	Choice selection	Authorization	Implementation	Challenge
API design	Decentralized	Centralized	Decentralized	Decentralized	Decentralized	Decentralized
API implementation	Decentralized	Centralized	Decentralized	Decentralized	Decentralized	Decentralized
API deployment	Decentralized	Centralized	Decentralized	Decentralized	Decentralized	Decentralized

Enforcement and incentivization
: Because the choices have been implemented in a standardized way, all aspects of design, implementation, and deployment can be validated automatically with tooling. For example, API teams must document interface designs in a machine-readable language, which is validated using a "lint" tool.

Talent distribution
: The central team needs to be populated with highly experienced designers, implementers, and architects to ensure that the centralized choices are the best ones. If the centralized choices have been made holistically and are of good quality, there is less of a talent requirement for designers and implementers in the decentralized teams.

Costs and benefits
: Machinery is always expensive to design, create, maintain, and tune. There will be a large initial investment to create the best set of standards for this type of system and a consistent challenge in keeping the choices and tools up to date as contexts change. But the payoff comes in the form of a reduced need for

distributed decisions and an improvement in decision-making throughput thanks to automation. One possible system impact of this pattern is unhappiness within API teams due to a loss of freedom—if the choices are too constrained, it may be difficult to attract good people.

Impact measurements

- Product and project schedule metrics
- Choice popularity (tracking when and how standardized choices are used)
- API team metrics

Governance Pattern #3: Collaborative Governance

In the collaborative governance pattern, API decisions are made individually, but a shared understanding of system impacts is developed collaboratively. The goal is to create a "shared brain" in terms of the system-level view, but maintain the speed and local optimization scope of a decentralized system (see Table 2-6).

Table 2-6. Decision map

Decision space	Inception	Choice generation	Choice selection	Authorization	Implementation	Challenge
API design	Centralized	Decentralized	Decentralized	Decentralized	Decentralized	Decentralized
API implementation	Decentralized	Decentralized	Decentralized	Decentralized	Decentralized	Decentralized
API deployment	Decentralized	Decentralized	Decentralized	Decentralized	Decentralized	Centralized
API measurement	Centralized	Centralized	Centralized	Centralized	Decentralized	Decentralized

Enforcement and incentivization

In collaborative governance most of the decisions are completely decentralized, with the exception of an API's inception and its measurement. This creates a "results-oriented" view of APIs in the system. It follows that enforcement is entirely results-oriented—if the API doesn't achieve the expected result it is retired and the team may be disbanded. Although design, implementation, and deployment decisions are decentralized, those decisions are typically influenced through incentivization. For example, if a team's decisions produce favorable results and those decisions are shared with the organization, they can be financially rewarded. The combination of a reward and transparency can influence the decisions of other teams in the organization.

Because most of the work is decentralized, collaboration between teams will need to be encouraged. That means that collaboration should be incentivized (or enforced) at the system level.

Talent distribution
: A collaborative governance pattern is talent-intensive. This level of decentralization requires a suitable level of talent distributed amongst the teams. It doesn't mean that every single worker has to be a star employee, but it does mean that each team needs enough talent to produce safe, high-quality decisions consistently.

Costs and benefits
: Highly skilled decentralized teams can produce innovative APIs of high quality. The main costs to achieving this are in talent and support for collaboration. As the scale of work increases, so too will these costs.

Impact measurements
- API product metrics
- API team metrics
- Usability metrics

Summary

In this chapter we gave you our definition of governance: *managing decision making and decision implementation*. From that definition, we took a closer look at what it means to make a decision and what it means to govern a decision. You learned that API decisions can be small ("What should my next line of code be?") or big ("Which supplier should we partner with?") and can range massively in scope. Most importantly, you learned that the system you are trying to govern is a *complex adaptive system*, which means it's difficult to predict the results of any decision management strategy you apply.

Next, we took a closer look at decision distribution and compared centralization and decentralization. To help you understand the differences, we compared them in terms of the *scope of optimization* and *scale of operation*. Then we discussed how you can break decisions down into their essential elements of inception, choice generation, selection, authorization, implementation, and challenge. By putting all of these concepts together, along with some enforcement and incentivization, you can build an effective API governance system.

Governance is at the heart of API management, so it's not a big surprise that it's a core concept for this book. Our goal in this chapter was to introduce the major concepts and levers of governance. In the rest of the book we'll dive deeper into the domain of API governance by tackling the specific challenges of which decisions matter the most, how to manage the people involved, and what to do as APIs mature and the scale of the APIs grows. In the next chapter, we'll start that journey by investigating how product thinking can help you identify the API work decisions that matter the most.

CHAPTER 3
The API as a Product

> If you build a great experience, customers tell each other about that. Word of mouth is really powerful.
>
> —Jeff Bezos, *founder and CEO of Amazon*

The phrase "API-as-a-Product" (AaaP) is something we hear often when talking to companies who have built and maintained sucessful API programs. It's a play on the *<Something>-as-a-Service* monikers that are often used in technical circles (Software-as-a-Service, Platform-as-a-Service, etc.) and is usually meant to indicate an important point of view when designing, implementing, and releasing APIs: that the API is a *product* fully deserving of proper design thinking, prototyping, customer research, and testing, as well as long-term monitoring and maintenance. "We treat our APIs just like any other product we offer" is the common meaning of the phrase.

In this chapter, we'll explore the AaaP approach and how you can use it to better design, deploy, and manage your APIs. As you may have gathered from Chapter 2, the AaaP approach involves understanding which decisions are critical for the success of your APIs and where within your organization those decisions should be made. It can help you think about what work needs to be centralized and what you can successfully decentralize, where enforcement and incentives are best applied, and how you can measure the impact of these decisions in order to quickly adapt your products (your APIs) when needed.

There are lots of decisions to make when creating new products for your customers. That is true whether you are creating a portable music player, a laptop computer, or a message queuing API. In all three cases, you need to know your audience, understand and solve their most pressing problems, and pay attention to customers when they give you feedback on how you can improve your product. These three things can be encapsulated in three lessons we will focus on in this chapter:

- Design thinking in order to make sure you know your audience and understand their problems
- Customer onboarding as a way to quickly show customers how they can succeed with your product
- Developer experience for managing the post-release lifecycle of your product and to gain insights for future modifications.

Along the way we'll learn from companies like Apple about the power of design thinking and customer onboarding. We will also see how Jeff Bezos helped the Amazon Web Services (AWS) division create an implementation mandate that establishes a clear, predictable developer experience. Most companies we talk to understand the notion of AaaP, but not all of them are able to turn this understanding into tangible action. However, the organizations that have a good track record for designing and releasing successful API products all have figured out how to meet the three big challenges we've just mentioned—the first of which has to do with how your teams *think* about the API products they are creating.

Design Thinking

One of the things that Apple is known for in product design circles is its ability to engage in *design thinking*. For example, when describing the work that went into Apple's Mac OS X, one of the key software architects, Cordell Ratzlaff, said: "We focused on what we thought people would need and want, and how they would interact with their computer." And this focus played out in real and tangible ways. "There were three evaluations required at the inception of a product idea: a marketing requirement document, an engineering requirement document, and a user-experience document," explained onetime Apple vice president (and one of the people credited with founding the field of human–computer interaction design) Donald Norman.

This attention to meeting people's needs definitely resulted in creating viable business for Apple. A continuing string of products over multiple decades contributed to Apple's reputation for defining new trends in technology and helped it capture the greater market share more than once.

Tim Brown, CEO of the California-based design and consulting firm, IDEO, defines the term "design thinking" as:

> A design discipline that uses the designer's sensibility and methods to match people's needs with what is technologically feasible and what a viable business strategy can convert into customer value and market opportunity.

There is a lot to unpack in that definition. For our purposes we'll focus on the ideas of "matching people's needs" and a "viable business strategy."

Matching People's Needs

One of the key reasons to build an API at all is to "match people's needs"—to solve a problem. Discovering problems to solve and deciding which problems have priority is just part of the challenge of the AaaP approach—that is the *what* of APIs. An even more fundamental element is knowing the *who*. Who are the people you are serving with this API? Correctly identifying the audience and their problem can go a long way toward ensuring you build the *right* product: one that works well and is used often by your target audience.

Harvard Business School's Clayton Christensen calls this work of understanding the needs of your audience the theory of *Jobs to Be Done*. He says, "People don't simply buy products or services, they 'hire' them to make progress in specific circumstances." People (your customers) want to make progress (solve problems), and they will use (or hire) whatever products or services they find will help them do that.

Should You Apply AaaP to Both Internal and External APIs?

Yes. Maybe not with the same level of investement of time and resources—we will cover that the next section—but this is one of the lessons Jeff Bezos taught us in "The Bezos Mandate" on page 42 that led Amazon to open the initially internal AWS platform for use as a revenue-generating external API. Because Amazon adopted AaaP from the start, not only was it *possible* (e.g., safe) to start to offer the same internal API to external users, but it was also *profitable*.

In most companies, the IT department is in the business of helping others (customers) solve problems. Most of the time, these customers are fellow employees within the same company (private internal developers). Sometimes the customers are important business partners or even anonymous public developers of third-party applications (external developers). Each of these developer audiences (private, partner, and public) has its own set of problems to solve and its own way of thinking about (and resolving) those problems. Design thinking encourages teams to get to know their audience before starting the process of creating APIs as a solution. We'll explore this topic in "Knowing Your Audience" on page 49.

Viable Business Strategy

Another important part of design thinking is determining a *viable business strategy* for your API product. It doesn't make sense to invest a lot of time and money in an API product that has little to no return value. Even when you do a good job of designing the right product for the right audience, you need to make sure you spend an appropriate amount of time and money and that you have a clear idea of what the payback will be when the API is up and running.

For most companies, there is only a finite amount of time, money, and energy that can be devoted to creating APIs to solve problems. That means deciding *which* problems get solved is of critical importance. Sometimes we encounter companies where the APIs that were built don't solve important business problems. Instead, they solve known problems in the IT department: things like exposing database tables or automating internal department processes. These are usually important problems to solve, but they might not be solutions that have a big impact on the day-to-day business operations or "move the needle" when it comes to meeting the company's annual sales or product goals.

Figuring out which problems matter for the business can be tricky. It might be difficult for leadership to communicate company goals in ways that the IT department can easily understand. And even when the IT team has a grasp of what problems could make a difference to the company, the department may not have good measures and metrics to confirm their assumptions and track their progress. For these reasons, it is important to have a standardized way to communicate key business objectives and relevant performance indicators. We'll talk more about this aspect of assessing your API's success later in the book.

The Bezos Mandate

No matter how old or new your company is, launching a successful API program—one that will transform your company—is not a simple task. One of the most well-respected companies who worked through this process (and which continues to transform itself more than a decade later) is Amazon, with its AWS platform. First created in the early 2000s, the platform is widely regarded as a brilliant master-stroke executed cleanly by a savvy team of IT and business executives. Although the AWS platform has become a huge success, it was born out of an internal need: a deep frustration with the amount of time needed for Amazon's IT programs to act upon and deliver the business team's requests. The AWS team was too slow to act, and what they eventually created was less than adequate at both the technical (scaling) and business (product quality) level.

As current AWS CEO Andy Jassy tells it, the AWS team (along with Amazon CEO Jeff Bezos and others) spent time identifying just what it was they were good at and what it would take to design and build out a core set of shared services on an interoperative platform. Their plan took more than three years to develop, but in the end formed the basis for Amazon's ability to offer its now famous Infrastructure-as-a-Service (IaaS) platform. This now $17bn business only happened because of careful attention to detail and relentless iterations to improve upon the original idea. Much the same way as Apple has transformed the way consumers thought of handheld devices, AWS has tranformed the way that businesses think of servers and other infrastructure.

One of the important ways in which AWS was able to change the point of view *internally* was through what is now known as the *Bezos Mandate*. Steve Yegge, former senior manager of software development at Amazon, describes the mandate in his "Google Platforms Rant" from 2005. One of the key points in the blog post is that Bezos issued a mandate that all teams must expose their functionality through APIs and that the only way to consume other teams' functionality must be through APIs. In other words, APIs are the only way to get things done. He also required that all APIs be designed and built *as if they would be exposed outside the company boundaries*. This idea that "APIs must be externalizable" was another key constraint that affected the way the APIs were designed, built, and managed.

So, design thinking is about matching the needs of your audience and committing to supporting viable business strategies when deciding which APIs are worthy of your limited resources and attention. What does that look like in real terms? How can you apply these product lessons to your API management efforts in order to express the API-as-a-Product approach?

Applying Design Thinking to APIs

You can elevate your APIs from utilities to products by applying the principles of design thinking to your design and creation process. Several companies we've talked to in the last few years are doing just that. They have made the decision that their APIs, even the APIs that are just used within the organization, deserve the same level of care, study, and design sense as any product or service that company already provides. For many companies, this means teaching their API developers and others in the IT department the principles of design thinking directly. For others, it means creating a "bridge" between the product design teams and the API teams within the same organization. In a few organizations we've worked with, we've seen both activities at the same time: teaching design thinking to the developers and strengthening the bridge between the product teams and the developer teams.

The actual content of a design-thinking curriculum is out of scope for this book. However, most design-thinking courses provide a mix of topics like the ones we've already mentioned in this chapter, such as:

- Design thinking skills
- Understanding the customer
- Service/workflow design
- Prototyping and testing
- Business considerations
- Measurement and assessment

If your company already has staff dedicated to product design, they can be a great resource for teaching your developer teams how to start thinking and acting like product designers. Even if your company doesn't have dedicated design staff, you can usually find product design classes on offer at a local college or university. Many of these institutions will offer to customize a course for delivery on site. Finally, even if you're a small company or just a single individual interested in the topic, you'll be able to find online courses in design thinking.

One company we talked to (a large consumer bank) decided to create its own internal design thinking course, with the product design staff delivering the sessions to API teams at various company locations. These trainers then became important resources that the API teams could call upon when they needed advice on how to improve their API designs. The goal was not to turn all their developers and software architects into skilled designers. What they were aiming to do was simply improve the API teams' understanding of the design process and teach them how to apply these skills to their own work.

It is important to remember that the results of design thinking are more than just improved usability or aesthetic appeal of your APIs. It can result in better understanding of the target audience (customers), a focus on creating APIs that meet viable business strategies, and a more reliable process for measuring the relative success of the APIs your team releases into the ecosystem.

As important as design is in the overall AaaP approach, it is just the start. It is also important to pay attention to the initial customer experience once the API is released and available for use. And that's what we'll cover in the next section.

Customer Onboarding

Anyone who's purchased anything from Apple in recent years knows how unboxing their products can be a memorable experience. And that is not by coincidence. For years, Apple has had a dedicated team whose only job is to focus on delivering the best "unpacking experience."

According to Adam Lashinsky, author of the book *Inside Apple* (Business Plus), "For months, a packaging designer was holed up in this room performing the most mundane of tasks—opening boxes." He continues, "Apple always wants to use the box that elicits the perfect emotional response on opening…One after another, the designer created and tested an endless series of arrows, colors, and tapes for a tiny tab designed to show the consumer where to pull back the invisible, full-bleed sticker adhered to the top of the clear iPod box. Getting it just right was this particular designer's obsession."

And this attention to detail went well beyond just opening the box and taking out the device. Apple made sure the battery was fully charged, that customers could be "up

and running" within seconds, and that the overall experience was pleasant and seamless. Apple's product teams wanted customers to *love* their product from the very start: as Stefan Thomke and Barbara Feinberg wrote in "Design Thinking and Innovation at Apple," "Helping people 'love' their equipment and the experience of using it animated—and continues to motivate—how Apple products were and are designed today."

When the API Is Your Only Product

Stripe is a successful payment service delivered via a great API that developers really love. The startup's recent valuation was about $10 bn with less than 700 employees. The founders' entire business strategy was to deliver their payment services via APIs. For this reason, they decided to invest in design thinking and the API-as-a-Product approach from the very beginning. For Stripe, the API was their *only* product. Treating their API as a product helped them meet both their technical and business goals.

This same attention to the initial experience of product customers applies to APIs. Making it possible for developers to *love* them may seem a far-fetched notion, but it has long-reaching implications. If your API is difficult to understand in the beginning, developers will struggle with it, and if it takes "too long" to get started, they will just walk away in frustration. In the API world, the time it takes to "get things working" is often referred to as *TTFHW*, or "Time to first Hello, World." In the online application space this is sometimes called "Time to Wow!" (TTW).

Time to Wow!

In his article "Growth Hacking: Creating a Wow Moment," David Skok, part of the equity investment firm Matrix Partners, describes the importance of a customer's "Wow!" moment as a key hurdle to cross in any customer relationship: "Wow! is the moment…where your buyer suddenly sees the benefit they get from using your product, and says to themselves 'Wow! This is great!'" And while Skok is talking directly to people designing and selling apps and online services to consumers, the same principles apply to people designing and deploying APIs.

A key element to the TTW approach is understanding not just the problem to solve (see "Design Thinking" on page 40) but also the time and work required to get to Wow! The effort it takes to reach a point where the API consumer understands how to use the API and learns that it will solve their important problems is the hurdle each and every API must cross in order to win over the consumer. Skok's approach is to map out the steps needed to experience the Wow! moment and work to reduce friction and effort along the way.

For example, consider the process of using an API that returns a list of hot leads for your company's key product, `WidgetA`. A typical process flow might look like this:

1. Send a `login` request to get an `access_token`.
2. Retrieve the `access_token` and store it.
3. Compose and send a request for the `product_list` using the `access_token`.
4. From the returned list, find the item where `name="WidgetA"` and get that record's `sales_lead_url`.
5. Use that `sales_lead_url` to send a request for all the sales leads where `status="hot"` (using the `access_token`).
6. You now have a list of hot sales leads for the `WidgetA` product.

That's a lot of steps, but we've seen workflows with many more than this. And each step along the way is an opportunity for the API consumer to make a mistake (e.g., send a malformed request) and for the API provider to return an error (e.g., a timeout for a data request). There are three possible request/response failures here (`login`, `product_list`, and `sales_leads`). The TTW will be limited to how long it takes a brand new developer to figure the API out and get it working. The longer it takes, the less likely they are to ever get their "Wow!" moment, and to keep using the API.

There are a number of ways to improve the TTW for this example. First, we could adjust the design by offering a direct call to get the list of hot leads (e.g., `GET /hot-leads-by-product-name?name=WidgetA`). We might also spend time writing "scenario" documentation that shows new users exactly how to solve this particular problem. We could even offer a sandbox environment for testing examples like this one that allows users to skip the authentication work while they learn the API.

API Pillars

Design, documentation, and testing are what we call *API pillars*. Those and others are covered in detail later in this book.

Anything you can do to reduce the time it takes to get to "Wow!" will improve the API consumer's opinion of your API and increase the chances that the API will be used by more developers both inside and outside your organization.

Onboarding for Your APIs

Just as Apple spends time on its "unboxing" experience, companies that are good at adopting the AaaP approach spend time making sure the "onboarding" experience for their APIs is as smooth and rewarding as possible. And just as Apple makes sure

the battery is already charged up when you open your new mobile phone, media player, tablet, etc., APIs can be "fully charged" at first use, making it easy for developers to get started and make an impact within minutes of trying out a new API.

Early in our work on APIs and API management we used to tell our customers they needed to get a brand new user from the initial view of their API's landing page to a live working example in about 30 minutes. Anything more than that risked losing a potential user and wasting all the time and money put into designing and deploying the API. However, after one of us completed a presentation on API onboarding, a representative of Twilio, the SMS API company, came up to us and told us they aim for an initial onboarding experience of 15 minutes or less.

Twilio's field (SMS APIs) is notoriously fiddly and confusing. Imagine trying to design a single API that works with dozens of different SMS gateways and companies *and* is easy to use and understand. Not an easy task. One of the keys to achieving their 15-minute onboarding goal is the copious use of measurements and metrics in their tutorials to identify bottlenecks—points where API users "drop out"—and determine just how long it takes for them to complete the tasks. "We are obsessed about metrics, we constantly monitor growth at different [stages] of the customer adoption funnel and collect NPS associated to the different activities we run," said Elisa Bellagamba when she was in charge of adoption for Twilio's voice products.

Twilio's Neo Moment

In 2011, Twilio's API evangelist Rob Spectre wrote a blog post relating his experiences teaching others how to use Twilio's SMS API. He tells the story of helping a developer to use the API for the first time:

> *In fifteen minutes we worked through a Twilio quickstart guide for outgoing calls and after navigating a few speedbumps, his Nokia feature handset lit up as his code executed. He looked up at me, looked back at his screen, answered his phone and heard his code say, "Hello world."*
>
> *"Whoa dude," he said, stunned. "I just did that."*
>
> *And that is pure magic.*

Spectre calls this the "Neo Moment" (referring to the character Neo from the *Matrix* movies) and says it can be a "powerful inspiration" for developers.

Twilio has worked diligently to engineer its API and onboarding experience to maximize these inspirational moments.

So, a great onboarding experience is more than just the result of a good design process. It includes well-crafted "getting started" and other initial tutorials, and diligent tracking of API consumers' *use* of these tutorials. Gathering data helps provide you with the information you need to improve the experience. Just as you design the API,

you need to design the onboarding experience, too. And improving the onboarding experience means acting on the feedback (both personal and automated) you get from API users.

But the AaaP approach doesn't stop with onboarding. Hopefully, you've gained a community of avid API consumers that will stick with you well past the initial introduction. And that means you need to focus on the overall *developer experience* for your APIs.

Developer Experience

Customer interactions with a product typically last well beyond the initial unboxing. Even though it is important to make sure the product "works right out of the box," it is also important to keep in mind that the customer will (hopefully) continue to use the product for quite a while. And, over time, customers' expectations change. They want to try new things. They get bored with some things they loved at the beginning. They start to explore options and even come up with unique ways to use the product and its features to solve new problems not initially covered by the product release. This continuing relationship between consumer and product is typically called the *user experience* (UX).

Apple pays attention to this ongoing relationship, too. Tai Tran, CEO and founder of social app Blue and former Apple employee, put it this way: "Whenever there's a question about whether we should do something or not we always come back to the question of, 'How would this impact the customer experience?'" Like any good product company, Apple tells its employees that the customer is king and to pay close attention to the way they interact with Apple products. And they're not worried about making lots of changes if that means making meaningful improvements along the way. For instance, between 1992 and 1997, Apple created more than 70 models of its Performa desktop computer (some of which were never even released to the public), each an attempt to take advantage of what it had learned from customer experience feedback on previous releases.

But probably the best example of managing the UX of its products is Apple's approach to customer service: the Genius Bar. As Van Baker of Gartner Research says, "The Genius Bar is a real differentiator for the stores and the fact that it is free really sets the stores apart from the other offerings in the industry." By offering customers a place to go with all their questions and problems, Apple illustrates the importance of the continuing relationship between customer and product.

All these UX elements—acknowledging an ongoing relationship, dedication to making small improvements, and offering easy access to support—are key to creating successful API products and experiences.

Knowing Your Audience

A big part of creating a successful API as a product is to make sure you target the right audience. That means knowing *who* is using your API and *what* problems they are trying to solve. We covered this in "Design Thinking" on page 40, and it is also an important part of the ongoing developer relationship. By focusing on the *who* and *what* of your API, you not only gain insight into what is important, but you can also think more creatively about the *how* of your API: what it is your API has to do in order to help your audience solve their problems.

We talked earlier in this chapter about the concept of matching people's needs when working through the design process. This same work needs to continue after your API is released. Gathering feedback, confirming your user stories, and generally paying close attention to how the APIs are used (or not used) is all part of the ongoing developer experience. Three important elements are:

- API discovery
- Error reporting
- API usage tracking

These three (and others) will be covered in depth later in this book, so we'll just highlight some aspects of them that are important to the overall developer experience (DX) of your AaaP strategy.

API discovery

API discovery is the work of "being where your developers are"—of making your API "findable" at the right time. One of the challenges of API programs at large organizations is that, even when an appropriate API is available, developers end up creating their own APIs—sometimes many times over throughout the organization. While sometimes seen as a kind of rebellion inside the company ("They won't use the APIs we give them!"), this explosion of duplicate functionality is more often evidence that developers cannot find the API they need when they need it.

Having a central registry for your APIs can help solve this problem. Establishing an API search hub or a portal where documentation, examples, and other important information can be accessed is another good way to improve the discoverability of your existing APIs.

Searching for APIs

As of the release of this book, there is no single, commonly used public search engine for APIs. One reason for this is that it is hard to index services on the web since most of them don't expose crawlable links and they rarely include links to other dependent

services. Another problem is that most of the APIs in use today are behind private firewalls and gateways, which makes them "invisible" to any publicly operated API search crawlers.

There are some open source projects and formats working to make API crawlers possible, including {API}Search, the API description format, and the Application-Level Profile Semantics (ALPS) service description format. These and others offer the possibility of a future API search engine available to all. In the meantime, invidivual organizations can use these standards internally to start the process of creating a searchable API landscape.

At least one company we talked to made publishing to a central discovery registry a required step in the build pipeline. That meant the developers building an API could not actually release it into production until they'd added it to the company's API registry and ensured all important APIs within the organization would be findable in one location—a big step toward improving the discovery quotient of their API program.

Error reporting

Errors happen all the time. They're part of the "landscape" of APIs. While you can use good design to try to reduce user errors and testing to try to eliminate development bugs in your own code, you will never get rid of all the errors. Instead of trying to do the impossible (eliminate all errors), a better tactic is to monitor your APIs closely so you can record and report the errors that do occur. This act of recording and reporting will give you important insight into the way your target audience is using your APIs—and that can lead to improving the developer experience.

One of the challenges encountered when creating and releasing a physical product (e.g., clothing, furniture, office supplies, etc.) is that it can be difficult to see errors when they occur during use. Unless you are standing right next to the person while they use your product, you're likely to miss details and lose out on valuable feedback. For this reason, most product companies engage in extensive prototyping and in-person monitored testing. The good news is, in the age of electronics and virtual products (e.g., mobile applications), you can build in error reporting and collect important feedback even after the product has left your control and is in the hands of users.

You can implement error reporting at a number of key touchpoints along the way for your APIs. For example:

End user error reporting
: You can add an error-reporting feature to your application. This prompts the user for permission to send detailed information if and when an error occurs. In

this way you can capture unexpected conditions on the user's end of the transaction.

Gateway error reporting
: You can add error reporting at the API router or gateway. This allows you to collect the state of the request when it first arrives "on your doorstep" and can help you discover malformed API requests or other network-related problems.

Service error reporting
: You can add error reporting within the service being called by your API. This helps you discover errors in coding the service and some component-level problems, such as issues with dependencies or internal issues due to changes within your organization's ecosystem.

Error reporting is a great way to get important feedback on how your API is being used and where problems occur. But it is only half of the tracking story. It is also important to track *successful* API usage.

API usage tracking

API usage tracking covers more than errors. It means tracking all requests and, eventually, analyzing the tracking information to find helpful patterns. As we mentioned in "Viable Business Strategy" on page 41, a big reason for creating and deploying APIs is to support your business strategies. As the well-known API evangelist Kin Lane puts it: "Understanding [how] APIs will (or won't) assist [the] organization to better reach their audience is what the API(s) are all about."

The data needed to determine whether your API is helping your organization to better reach your target audience is usually expressed as OKRs (objectives and key results) and KPIs (key performance indicators). We'll dig deeper into these later, but for now it is important to recognize that in order to meet your goals, you need to know just how your APIs are doing along these lines. That means tracking not just the errors that occur, as described in the previous section, but also the successes.

For example, you'll want to collect data on which applications are making which API calls and whether those applications are effectively meeting the needs of their users. Tracking has the added benefit of helping you to see patterns over a wide range of users — patterns that individual users may not be able to notice. For example, you might discover that applications continue to make the same series of API calls over and over again, such as:

```
GET http://api.mycompany.org/customers/?last-order=90days
GET http://api.mycompany.org/promotions/?flyer=90daypromo
POST http://api.mycompany.org/mailing/
customerid=1&content="It has been more than 90 days since...."
POST http://api.mycompany.org/mailing/
customerid=2&content="It has been more than 90 days since...."
```

```
...
POST http://api.mycompany.org/mailing/
customerid=99&content="It has been more than 90 days since...."
```

This pattern might indicate the need for a new, more efficient way for your target audience to send out mailings to key customer groups—a single call from the application that will combine the target customer group with the selected promotional content. For example:

```
POST http://api.mycompany.org/bulk-mailing
customer-filter=last-order-90days&content-flyer=90daypromo
```

This call creates less client/server traffic, reduces the number of possible network failures, and is easier to use for API consumers. And it was "suggested" not by a customer, but by paying attention to the API usage tracking information.

Drink Your Own Champagne

Three years ago, a European national railway company decided to organize some hackathons for its developer communities; one for external developers and another for internal developers.

The external event was coordinated by the communications and product management leadership. They arranged to have the IT department produce some static data available for external use and helped the IT teams design a set of simple, task-focused APIs for accessing things like station locations, departure schedules, etc. These were implemented quickly and viewed by the IT department as "less powerful" than its down "full-featured" internal APIs. The event went quite well.

Six months later, the IT department arranged its own hackathon using the "official" internal APIs. After a while the hackathon organizers realized the internal developer teams had switched from using the "full-featured" internal APIs to the easier, more task-focused external APIs. And the teams were more effective and productive, too.

There are a few lessons to be learned from this experience. First, the task-focused APIs were preferred by all developers. Second, creating these "simpler" APIs did not take much time or resources. And third, it is always best for IT departments to pay attention to which APIs are popular and used more often. This last lesson leads to the common phrase "Drinking your own champagne." With APIs as with any other product, it is often best for internal teams to be using the same product external teams are using.

This leads us to one more important area of developer experience (DX): making it safe and easy for developers to "do the right thing" with your API.

Making It Safe and Easy

Along with supporting easy API discovery and accurate tracking of both errors and general API usage, it is important to provide easy access to ongoing support and training to your API developers. In fact, it is the experience that occurs *after* you've successfully onboarded your developers that will ensure a long-term positive relationship. We saw an example of this kind of attention to the ongoing relationship earlier in this section, with Apple's use of the Genius Bar as a source of support for existing customers. Your APIs need their own Genius Bar, too.

Another important aspect of support for developers is making your product *safe for use*. In other words, it should be somewhat difficult to misuse the product in ways that result in some sort of harm. For example, it might be hard to delete important data, remove the only admin account, and so forth. Paying attention to how your API consumers (e.g., developers) *use* the product can help you identify areas where some added safety efforts can pay off.

It takes a mix of both these elements—ease and safety—to create a powerful and ongoing connection with your API developers.

Making APIs safe to use

There are a number of elements of an API that can represent *risk* from the developer's point of view. Sometimes certain API calls can do dangerous things, like delete all customer records or change all service prices to zero. Sometimes even *connecting* to an API server can represent some risk. For example, setting up a connection string to a data API might make it too easy to expose usernames and passwords in URLs or unencrypted message bodies. We've seen lots of these types of safety issues in our reviews of APIs.

Often risks can be *designed out* of the API. That is, you can make changes in the design that make encountering a particular risk less likely. For example, you can design an API that deletes critical data to also support an "undo" API call. That way, if someone mistakenly deletes important data, they can also invoke the undo call to reverse it. Or you can require elevated access rights to execute certain operations, such as requiring an extra data field (e.g., a passcode) to be sent with calls that update critical information.

However, sometimes it can be difficult to mitigate the risk through API design changes. There may be some cases where executing an API call is simply inherently risky. Any API call that deletes data is risky, no matter how many design changes you make to it. Some API calls might always take a long time to execute, possibly consuming lots of server-side resources. Other APIs might execute quickly and result in quite a lot of data in return. For example, a filter query might potentially return hundreds of thousands of records.

In cases where API calls represent unavoidable risk, you can reduce negative impacts by adding warnings to the API documentation itself. In this way, you can make it easier for API consumers to recognize potential dangers ahead of time and possibly avoid making critical mistakes. There are lots of ways you can format documentation to help point out possible dangers. Highlighted text telling the user of the problem ("Warning: This API call may return over a million records, depending on your filter settings") is one way to do it. Another way to warn API users is to adopt a kind of labeling method using symbols. This way, there is no need to add lots of text to your documentation: readers can just recognize the warning label instead.

Physical products use information and warning symbols quite often (see Figure 3-1).

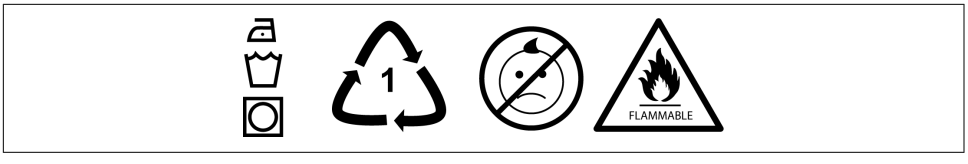


Figure 3-1. Examples of household product labels

You can adopt a similar approach for your APIs, too (see Figure 3-2).

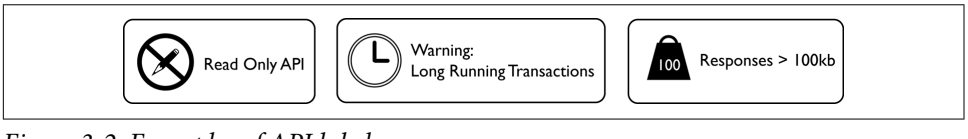


Figure 3-2. Examples of API labels

Easy-to-read warning symbols combined with design changes to make it more unlikely for API users to make regrettable mistakes are good practices for increasing the safety of your API product.

Making APIs easy to use

It is also important to make your API relatively easy to use for your API consumers. If it takes too many steps to accomplish a task, if the names and numbers of arguments API developers need to pass are confusing or complicated, or if the names of the API calls themselves don't make much sense to consumers, your API can run into problems. Not only will developers be unhappy using your API, but they might make more errors, too.

You can *design in* ease of use by adopting naming patterns that fit your developers' jobs-to-be-done vocabulary. This goes back to understanding your audience ("Matching People's Needs" on page 41) and solving their problems ("Viable Business Strategy" on page 41). But even when you do that, if your API is large (e.g., lots of URLs or actions) or just plain complicated (lots of options to deal with), you can't always rely on design to solve your problem. Instead, you may need to make it easier

for API consumers to ask the right questions and find appropriate answers. Your API needs a kind of "Genius Bar" for developers.

Probably the easiest way to provide your developers an API Genius Bar is through the documentation. By adding more than simple reference documentation (e.g., API name, methods, arguments, and return values), you can elevate your API docs to "genius" level. For example, you can add a Frequently Asked Questions (FAQ) section where you provide answers (or pointers) to the most common consumer questions. You can expand your FAQ support by adding a "How Do I…?" section that gives step-by-step short examples on how to accomplish common tasks. You can even provide fully functional examples that developers can use as starter material for their own projects.

The next level up from enhanced documentation is an active online support form or chat channel. Support forums provide an ongoing conversation space where developers can ask questions to a larger group and share solutions. In the case of large API communities, these forums can even become a source of important bug fixes and feature requests. Forums can also become a valuable repository of knowledge accumulated over time, especially when you have a robust search mechanism.

Chat channels offer an even more immediate means of providing Genius Bar support for your API consumers. Chats often happen in real time and can add an additional level of personalization to your developer experience. This is also another great place to leverage and grow community knowledge about your API product.

Finally, for large API communities and/or large organizations, it can make sense to provide in-person support for your product in the form of API evangelists, trainers, or troubleshooters. Your company can arrange meetups or hack events where API users come together to work on projects or test new features. This works whether your primary API community is internal (e.g., company employees) or external (e.g., partners or public API users). The more personal you can make your connection to your developers, the more likely you are to be able to learn from them and improve the ease of use of your API.

Taking the time to make your APIs safer to use and easy to work with can go a long way toward establishing a positive relationship with your API consumers and, in turn, improving your overall developer experience.

Summary

In this chapter, we introduced the API-as-a-Product approach and how you can use it to better design, deploy, and manage your APIs. Adopting this approach means knowing your audience, understanding and solving their problems, and acting on API users' feedback.

The three key concepts we explored in the AaaP space were:

- Using design thinking to make sure you know your audience and understand their problems
- Focusing on customer onboarding as a way to quickly show customers how they can succeed with your product
- Investing in providing a developer experience for managing the post-release lifecycle of your product and gaining insights for future modifications

Along the way we learned how dedication to AaaP principles helped companies like Apple, Amazon, Twilio, and others build not just successful products, but also loyal customers. And, regardless of whether your API program is targeting only internal users or both internal and external developers, a loyal user community is critical to its long-term health and success.

Now that you have a grasp of the foundational principles of AaaP, we can turn to the common set of skills that we find all successful API programs nurture and grow. We call these the "API pillars," and that's what we'll cover in the next chapter.

About the Authors

Mehdi Medjaoui is an entrepreneur in the API industry, cofounder of OAuth.io, and creator of APIDays Conferences, the main worldwide series of API conferences held every year in seven countries. As lead API economist at the API Academy, Mehdi advises API decision makers about the impact of API adoption in their digital transformation strategies at the micro and macro level. He designed the API Industry Landscape, has been a coauthor of the "Banking APIs: State of the Market" industry report since 2015, and serves as a European Commission expert on the APIs for Digital Government (APIs4DGov) project. He also lectures on entrepreneurship in the digital age at HEC Paris MBA and is a board advisor at several API tooling startups.

An expert in protocol design and structured data, **Erik Wilde** consults with organizations to help them get the most out of APIs and microservices. Erik has been involved in the development of innovative technologies since the advent of the web and is active in the IETF and W3C communities. He obtained his PhD from ETH Zurich and taught at ETH Zurich and UC Berkeley before working at EMC, Siemens, and, most recently, CA Technologies

Ronnie Mitra helps companies around the world, both large and small, improve their organizational designs and system architectures. In his role as the lead designer at the API Academy, he combines a focus on UX principles and system complexity to tackle the challenges of building effective API programs and establishing practical strategies for transformation.

An internationally known author and speaker, **Mike Amundsen** travels the world consulting and talking about network architecture, web development, and the intersection of technology and society. As lead API architect for the API Academy, he works with companies to provide insight on how best to capitalize on the opportunities APIs present to both consumers and the enterprise.

Colophon

The animal on the cover of *Continuous API Management* is the Welsh Shepherd, (Welsh: *Ci Defaid Cymreig*), a breed of collie-type domestic herding dog native to Wales. Appearing in black-and-white, red-and-white, and tri-color varieties, with a high incidence of merle markings, they have longer limbs and a broader chest and muzzle than the Border Collie.

Welsh Shepherds are extremely strong-willed and energetic dogs, and function mostly independently of human direction once trained in herding duties. However, they lack the low posture and strong eye contact of the border collie (lupine predation traits that allow a dog to manage a herd with less effort), making them less popular candidates for modern livestock supervision.

Due to a combination of breeding for behavioral characteristics rather than features, and dilution due to cross-breeding with the Border Collie, the Welsh Sheepdog is not recognized as a standardized breed by any major kennel organization. In recent years, efforts have been made to preserve the breed, mostly for domestic purposes.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from J.G. Wood's *Animate Creation*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.