devmio

jaxmagazine

The digital magazine for enterprise developers

Killing the Vibe?

ARCHITECTURE IN THE AGE OF AI

Image created with ChatGPT (DALL-E)

Circular Architecture
Key Container Considerations

TABLE OF CONTENT

How modern AI tools are reshaping how we interact with software

Killing the Vibe? Architecture in the Age of AI

Stefan Toth

14. April 2026

Discussions about artificial intelligence (AI) often revolve around future possibilities and exaggerated promises of productivity. More relevant, however, is what is happening right now: What are today's available AI tools actually capable of, and what impact does this have on the way we develop software systems?

Every day we read marketing messages about "disruption through AI," about "10x productivity," and about "AGI" (artificial general intelligence), which is supposedly just around the corner. Again and again, there is speculation about which roles and jobs will no longer be needed in the future. On the technical side, the number of copilots, MCP servers, and agentic systems is exploding. It's exhausting—and exciting—at the same time.

Even if AI is hard to stomach at the marketing level, the technical side is interesting and inspiring. The more one engages with the possibilities and tools, the clearer it becomes that AI-assisted software development is still "just" engineering—at the same time, however, the way we use tools changes, tasks shift, and some long-held truths of software development are being questioned. In fact, after several years of experience, one can state that software development, system design, and architecture are changing.

Software architecture has traditionally created order in complexity—it structures, abstracts, and deliberately invests in what is "hard to change." As a planning discipline, architectural work requires a certain distance from concrete implementation and aims to avoid costly missteps. With language models, coding assistants, and context engineering, many things are now cheaper: prototypes are built in hours instead of weeks, variants can be implemented faster, and solutions can be ported more quickly. The boundary between what is "hard" and "easy to change" is being redrawn. At the same time, new challenges are emerging that we must address.

Vibe Coding: The Gateway Drug

For many, the first contact with AI tools is the chat interface of a large language model (LLM). When asked questions, these models often provide concrete answers, and when technical questions arise, interface specifications, partial implementations, or executable scripts suddenly emerge. With CLI tools (command-line integration) or LLM integration in IDEs, it is tempting to use these solutions directly. With a bit of routine, one can slip into a "flow state" in which AI tools are used rather carefree, pragmatic concerns are set aside, and rapid progress is celebrated instead. Vibe coding is born.

In experimental, low-risk environments, this approach can indeed be convincing, and many of us have at some point "vibe-coded" a small tool for personal use. With the right problem, the progress can be

impressive, and the productivity promises of the AI bubble start to seem more realistic. This initial rush simultaneously creates followers of the AI movement and people who reject such superficial development practices. But is this really the core of what LLMs can contribute to software development?

AI-Assisted Software Development

Vibe Coding is a mental model or an attitude, orthogonal to the specific choice of tools. We can also implement software solutions quickly and with little design effort outside of the AI context. In the past, we called it Quick Hack or spoke on a higher level of 'accidental architecture.' If we separate this approach from the technical possibilities, an entire field can be unfolded in which we professionally develop AI-supported software.

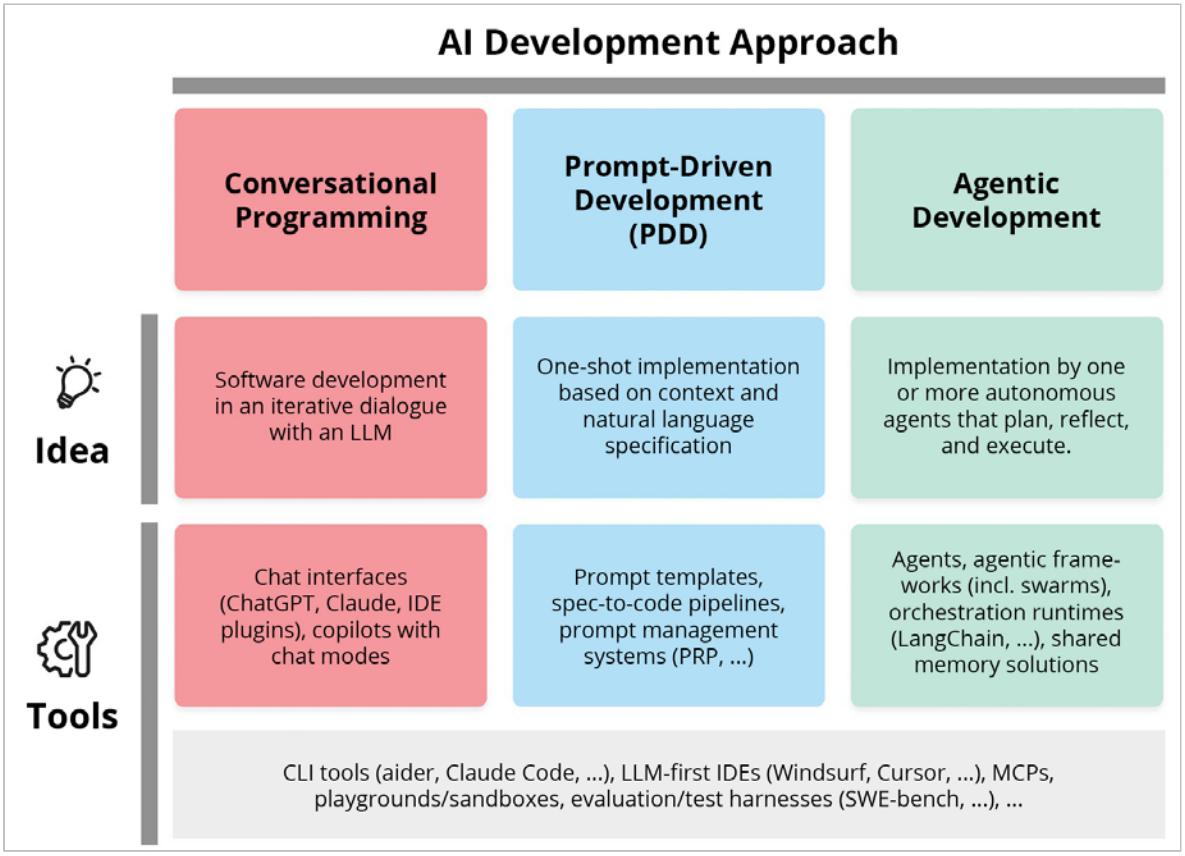


Figure 1: Approaches to AI-Assisted Software Development

Figure 1 shows different development paradigms that arise with LLM support:

- **Conversational Programming:** Software is developed iteratively through dialogue with an LLM (e.g., ChatGPT, IDE plug-ins, copilots).
- **Prompt-Driven Development (PDD):** Low-level coding is replaced by natural language specifications. Key elements are prompt templates, specification-to-code pipelines, and prompt management systems.
- **Agentic Development:** Development is delegated to autonomous agents that plan, execute, and orchestrate tasks (agent frameworks, swarm approaches, orchestration runtimes, shared memory).

More complex setups, such as Product Requirement Prompts (PRPs [1]) or Agent Swarms (e.g., Claude Flow [2] or Archon [3]), attempt to address core problems of simpler conversational programming but bring new challenges. For example, using a "concert" of agents can help counter simple hallucinations, but it also introduces more autonomy in development. This can lead to empty dummy implementations, skipped tests, or overly simplistic solution alternatives. When applied naively, one is often confronted with success messages that have little to do with actual implementation success.

Each approach has its own sweet spot, but all struggle to consistently adhere to a development culture, use tokens efficiently, protect working solutions, or secure data out of the box.

The effort required for implementation also varies more due to the use of AI tools. While in the past there was some variance to estimate development effort, today it is difficult to say whether a solution can be implemented almost for free or whether, after several failed AI attempts, one ultimately ends up with manual implementation as the only viable solution.

All of this requires not only experience with the approaches and tools mentioned but also some new strategies for handling AI-assisted

development.

Strategies for Better AI-Driven Development

Some best practices for working with AI tools emerge fairly quickly with experience. In our implementation projects, we break tasks into small pieces (including those for agents) and sometimes use broader, preliminary prompts to understand potential solutions and structure the problem. The results of such early prompts are usually discarded.

We generally work in sandboxes [4], taking small, incremental steps that are intensive in scope. We try to limit the action range of LLMs through global rules and, in doing so, define aspects of the development culture. All of this has an architectural component, but we can also truly understand and use AI tools as architectural instruments.

From this perspective, some shifts become noticeable—three of which I will highlight below. These are illustrated in **Figure 2**.

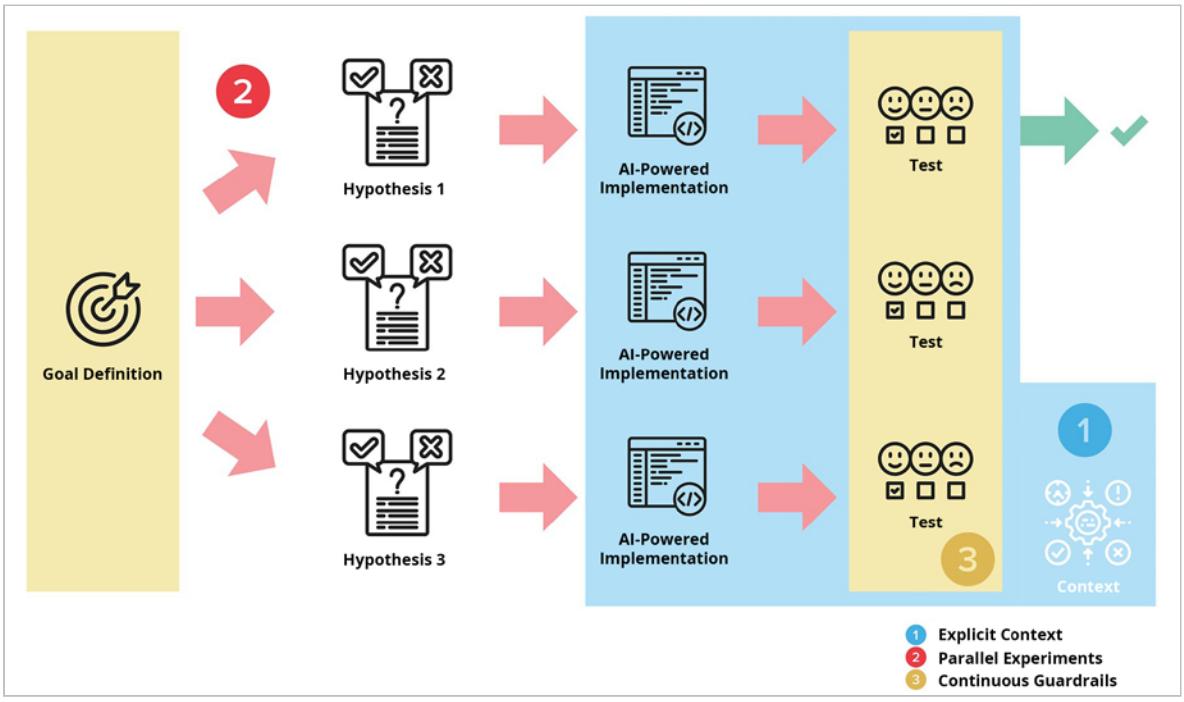


Figure 2: The New Focus of Architectural Work

From Implicit to Explicit Context

We cannot assume that AI tools possess common sense or have been around long enough to already know how a problem in our environment should be correctly solved. While AI tools can quickly produce generic code, a major lever for obtaining architecturally usable solutions lies in context engineering [5]. What used to exist implicitly in team cultures and developers' minds must now become machine-readable: we therefore embed architectural principles, communication patterns, and architecture-relevant quality criteria into the context of LLMs. This can range from simple visibility rules or the intended use of technologies to codified compliance requirements.

Beyond simple entries in an agents.md file, there are more advanced approaches. For example, we can automatically check every AI-assisted function against regulatory requirements stored in a knowledge graph. This makes it clear whether personal data is properly encrypted or whether audit logs are complete. This explicitness affects all layers of architecture. Service graphs can document allowed communication paths. Architecture Decision Records (ADRs) become structured data that LLMs can understand and consider. Commit messages follow schemas that make intentions machine-readable.

From Sequential to Parallel Solution Finding

The classical workflow – analysis design implementation – becomes impractical when solutions can be implemented very quickly, even prototypically. We therefore work more intensively with hypothesis-driven development and parallel experiments.

For example, suppose we observe increasingly high load peaks in our application and need a strategy to handle them. Whereas in the past much was resolved at the discussion and analysis level, today we can define, implement, and test variants: Does horizontal scaling with

Kubernetes help? Should we introduce a Redis cache layer to reduce database load? Can serverless functions support certain key endpoints?

In several projects, we have been able to address such questions practically by implementing hypotheses with AI support rather than discussing them theoretically for a long time. Load tests then provide reliable data, and decisions are based on facts rather than assumptions.

This approach brings benefits on many levels: in order to make reliable comparisons, we engage more intensively with objectives and their quantification. Architectural roles that were more distant from implementation return to the code. Compatibility issues or other hidden difficulties become visible during the selection process.

From Preparation to Continuous Guidance

Architectural work involves considering technical options and risks, generalizing insights from implementation work, and achieving quality goals. When working hypothesis-driven and potentially pursuing multiple options in parallel, it becomes more important to establish continuous feedback in the form of guardrails and fitness functions.

Instead of restricting solutions and manually checking them point by point, continuous guardrails emerge. These allow comparisons of technical alternatives, neutral quality assessment, and decentralized evaluation. For instance, we use review agents that check architectural guidelines for every pull request (layering, API communication, patterns and technology usage, etc.).

These guardrails can be complemented by fitness functions, such as performance benchmarks and security scans, to continuously verify whether architectural properties are maintained, without overly constraining the architecture itself. Every generated code undergoes immediate unit and policy checks. Agents that generate variants

automatically stop if the abort criteria are violated. Together with explicit context, this creates a framework around AI-assisted solutions that enables safe handling of AI tools without sacrificing too much development speed.

New Approaches for Classic Architecture Tasks

The three shifts—explicit context, parallel experiments, and continuous guardrails—are transforming architectural work. It becomes more granular, experimental, and automated. At the same time, information such as architectural patterns, quality goals, or principles from "softer" documents like wikis is being shifted into a systematic context for LLMs. Thus, architectural concepts have a secure impact on the solution, but they also need to be described more explicitly and precisely than before.

Beyond this transformation, we can also use AI tools to support more traditional architecture tasks. **Figure 3** illustrates some ideas in this regard.

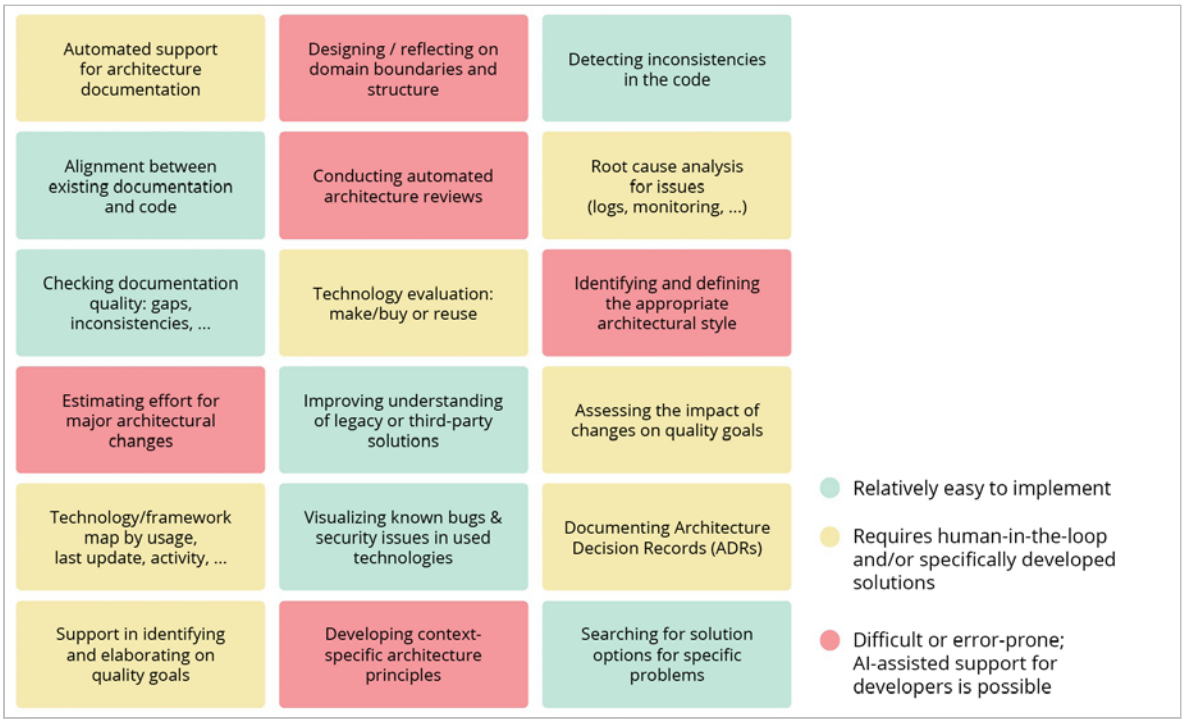


Figure 3: Opportunities for AI Support in Software Architecture

AI agents are particularly well-suited to supporting the typical "meta-tasks" of architecture – tasks that require a broad perspective and contextual understanding. One current weakness is categorization tasks, such as grouping problems by similarity. Even here, however, interaction with LLMs and specifically oriented agents can help support one's own thought processes.

When it comes to structuring the solution space or identifying technical options, the support provided by AI improves. Analytical agents that use code, documentation, configuration, or monitoring data as a foundation work especially well. As a result, gap analyses, simpler documentation tasks (that do not abstract too much), and reviews are often the focus of AI-driven architecture initiatives.

However, caution is warranted. Not every architectural task should be replaced one-to-one with AI agents in the future, even if that were possible. Some tasks are likely to fundamentally change. Take, for example, architectural documentation. In the past, the focus was on preparing knowledge about a solution in a way that could help with future problems. At the time of documentation, we do not know these future problems in detail, so documentation tends to be general. Alternatively, very detailed documentation may be created, but due to its volume, it often becomes unmaintained and outdated.

Either way, documentation is built for a problem we do not yet have. Is it for a new team member who needs a general overview? For a large refactoring that will dissect our data model? For performance issues that challenge our use of events? We do not know.

With LLMs and AI agents, we are suddenly able to assemble the right information at the moment the actual problem arises. We have a tool that "knows" a lot of information: code, tickets, monitoring data, past development prompts, and more. Instead of taking a detour via established documentation templates, it is more effective here to engage in context engineering for the LLM and create a support tool for

problem-solving. Manually created architectural documentation is reduced to simplified overviews, which are easier to maintain due to their conciseness and, in any case, not in the sweet spot of current LLMs.

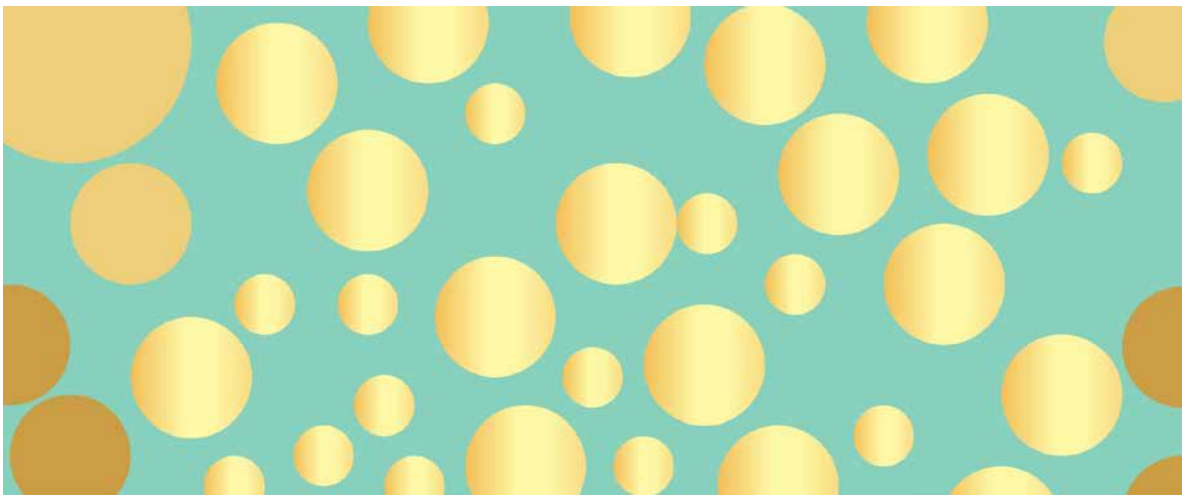
Conclusion: Flow and Vibe Broken?

In this article, we have traced the arc from vibe coding and AI-assisted development to the discipline of architecture. Even today, it is observable that classical roles, such as developers and architects, behave differently, or can behave differently, when a context of AI-supporting tools is present. But what exactly characterizes the change brought by AI? Is it raw development speed? Tenfold productivity? In practice, software development remains a deep engineering field, even with AI support.

Software architecture is experiencing a shift toward continuous, more explicit architectural work that supports parallel experiments. When tools and processes are applied correctly, this hardly slows down development but does provide a greater overview and direction.

Links & References

- [1] https://github.com/Wirasm/PRPs-agentic-eng
- [2] https://github.com/ruvnet/claude-flow
- [3] https://github.com/coleam00/Archon
- [4] https://docs.anthropic.com/en/docs/claude-code/devcontainer
- [5] https://www.philschmid.de/context-engineering



Translating the theoretical concepts to code

Practical Advice For Circular Architecture

Daniel Buza

14. April 2026

This article gives a short summary about circular architecture's generic background motivation, an example code layout and observations about what advantages the structure brings. Keeping these in mind, it can help you to easily apply the same structure to other stacks too.

Do we even need architecture? Well, as soon as you code something, you have a structure that then naturally supports your code to grow in some directions and holds it back from growing towards other directions.

Circular architectures (such as Clean Architecture, Hexagon Architecture, Port and Adapters) are not new, still in many cases they are not utilized to their full potential. For sure, there are cases, when they are not utilized, because they do not fit the problem. But in many

cases, they would fit the problem, it is just that developers are either not familiar with the concept or just lacking practical experience to actually translate the theoretical concepts to code.

Main guiding idea

Circular architectures are a well discussed topic, with multiple aspects, and pros and cons. I do not intend to re-iterate all of the publications on the topic here, so here is my subjective take on how to summarize up the most important guiding idea of them: **Take care who can force change in a given part of the code: let only more important parts force change in less important parts and never the other way around.**

Reading this feels like I told you that the sky is blue. But if you think about it for a bit, in many cases this rule is not respected. Let's take a closer look at the building blocks of this definition.

Firstly, what is a "part"? Is it a class? A package? A method? Well, I choose to write "part" because this rule can be applied at any scale.

Then, how do we define what is "more important"? This is commonly misunderstood as "all the parts that can be sorted by importance". That is not true. In many cases, you can not really tell if utilizing a given database or properly configuring connections towards a Kafka broker is more important. In other cases it is easier to tell.

Think about the question of if modelling your entities according to their business rule or the Java version your system runs on is more important? All in all, if you have two parts where you can not really tell, which is more important then you should try to avoid enabling any of them to force the other to change.

Last, but not least, what does it mean to force changes? Well, in the most trivial way, it means if you change one part, the other will simply not compile. This alone can cause a lot of pain, most of us already have

the experience of applying a small change somewhere in our code causing dozens of methods to change, for example to pass a value through the entire stack of method calls. This, you can usually define as direct source code level dependency: in most cases, imports of a class are fair indicators of which classes can (and trust me, sooner or later, wi ll) force change in a given class.

The bad news is if this (code does not even compile) happens, you can still be happy. At least, the compiler tells you that there is something to do. In some cases, the code is still going to compile. You can deploy it, but when you actually want to use the feature, it will break. A side-note here: there is a lot to tell about the importance of meaningful integration and end-to-end tests (which even in this case would prevent you from believing everything is fine), but I will not drive this article deeper in that direction.

You may notice that "some class can force my class to change" strongly relates to knowledge about that other class. Knowledge can be explicit (my class imports that class, invokes its methods, etc.) or implicit: I as developer know, that I should place @EnableScheduling to a Spring configuration class because some other class has a @Scheduled method which will not be invoked otherwise. Also, the so called "God-Classes" are also a direct product of allowing some class to have a lot of knowledge about all the other classes, utilised technologies, security and performance aspects, etc.

More knowledge means more possibilities to be forced to change. With this in mind,you can re-phrase the summary as: **Apply the need-to-know principle to your code: allow each part of your code to only have the minimally needed amount of knowledge on other parts.**

A practical example

Let's see an example of this. I recently had to plan a web-application which has both its frontend and its backend written in TypeScript.

(Alone the fact that both sides are written in the same language, opens up the possibility to consider moving code between frontend and backend, we will get back to that later.)

So, as we all know, we want to limit what the others know about a specific class. For that, we can utilise interfaces, so we can control exactly what is "seen" from a class from the outside.

Who defines the interface?

This question is really important while planning and implementing circular architecture. In "classic" layered architectures the interfaces are mainly abstracting away the capabilities of a given implementation. Thus, the implementation influences, defines what is present in the interface, meaning what the caller/user of that interface is allowed/enabled to do.

In clean architecture, it is turned around. The one who will call/use the interface, can specify its requirements. It is not like "I can provide you this" (from the perspective of the implementation), it is more like "if you want to be the implementation I use, you have to provide me this" (from the perspective of the caller).

This enables us to keep the dependencies pointing to the right direction.

The three main parts

For a simple case, you can identify three main parts of the application:

- Classes which should know about the UI and should know about the actual presentation framework which is being used.
- Classes which are representing the business core logic.
- Classes which are interacting with downstreams (databases, third party webservices, etc.).

Take note of two hints:

- It is already clear that the middle one is the one which is more important than the other two but it would be hard to define if the first or the third one is more important than the other.
- On the other hand, if you keep writing your business code in a way that it is not aware of it running on the frontend (so it should not know about Angular, for example) or the backend (so it should not know about AWS Lambda runtime, for example) then you will not only be able to move the core logic freely between the two sides but the core will remain focused. This leaves more room to express business logic, business abstractions, and business flows.

You can have many reasons to move services from one side to the other: you might need to monitor something more closely, you might have to protect the code more, you might need to speed up the flow by utilising local resources, etc.

Calling backend

So, UI components are going to remain in the frontend. Calling downstreams will remain in the backend. (Yes, I know, technically you h ave other options too, but for many reasons, usually UI is a frontend-only, talking to the DB is a backend-only task.)

This leads to the question about how the bridge between code on frontend and backend is to be built. Let's think about it. Is there any part of the code, which should know about the details? Obviously (maybe besides some security aspects), no.

Implying the following:

- The caller should not know whether the service it calls, resides on frontend or on backend.
- The service that is being called should not know whether the caller resides on the frontend or on the backend.

These points can be easily achieved by adding two classes to the structure. They should be the classes that know about networking, HTTP and REST details. They are also responsible for serializing and deserializing the business objects.

The following diagram in **Figure 1** summarises up the roles:

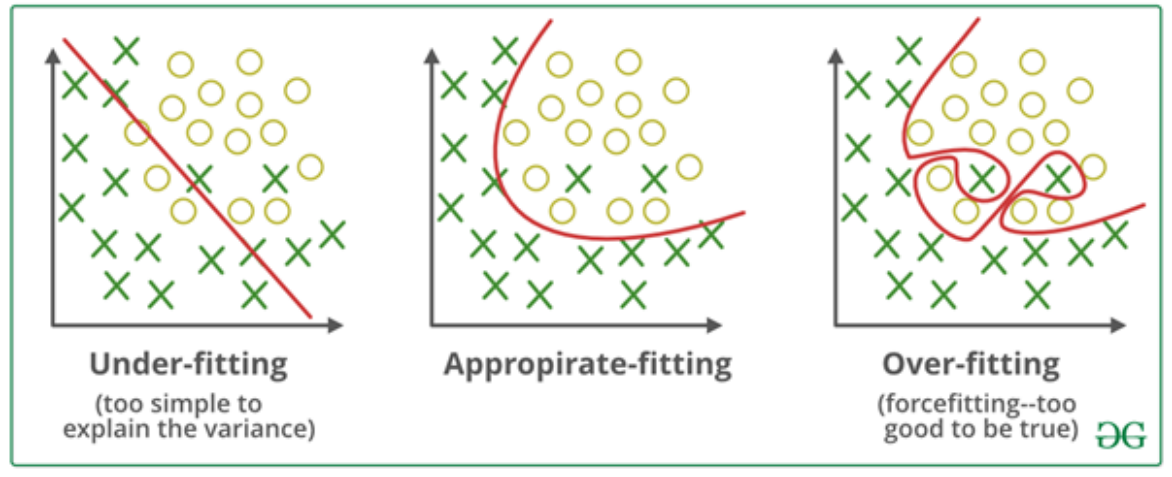


Fig. 1: Role summary

This structure keeps responsibilities clean and prevents the classes from suddenly "knowing too much".

Injecting actual implementations

An additional benefit of this layout is that you can easily make all the classes run locally without any HTTP traffic taking place. This is really important if you want to have encapsulated End2End tests.

If you do not have this option, then you either have to actually start some service which acts like your backend endpoints or you have to mock all the HTTP calls (but that also weakens the End2End tests a lot, as you effectively have turned them to integration tests for your frontend).

Faster development

Another benefit is, this structure allows faster development while the system is in its initial/prototyping mode. You simply do not have to deal with infrastructure as you can run the whole system in a local browser. Still, you can already define the needs of your business classes.

Imagine that you will need some persistent storage. You can simply define that your business service needs an instance of a technical service. This can save an entity and load an entity for an id. The following diagram in **Figure 2** showcases the three levels of maturity for implementing the technical service:

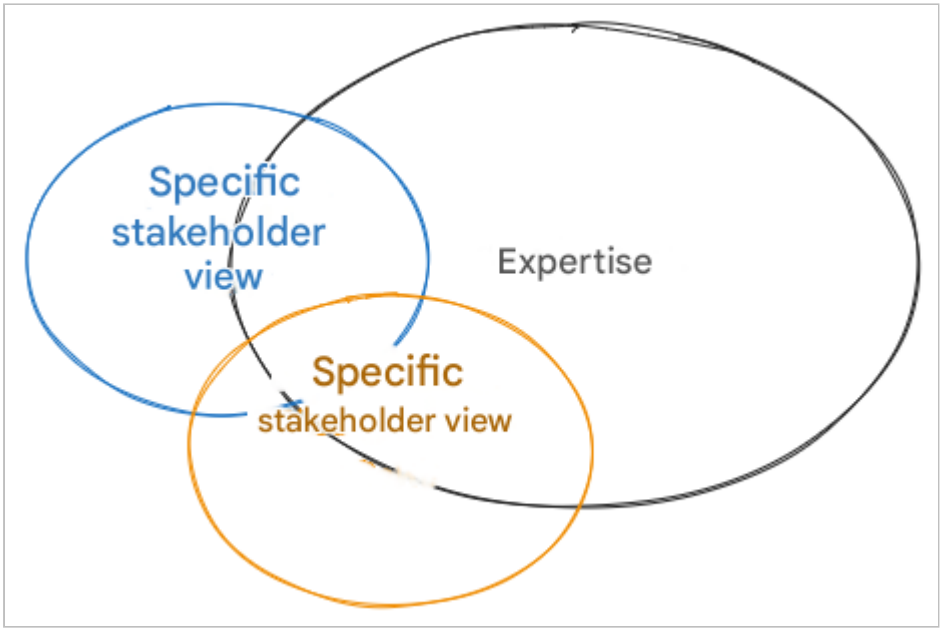


Fig. 2: Three maturity levels

Conclusion

As a conclusion, we can state that applying circular architecture brings many advantages and can be utilised in various situations. Still, keep in mind that clean architecture is not the ultimate answer to all of our questions. It helps in many cases and does not help in others. But if you choose to use it, it is good to be aware of many of its potential benefits.

A guide for aligning engineering work with business outcomes

How Software Teams Can Use OKR Frameworks to Ship Better, Faster, and Smarter

Gaurav Belani

14. April 2026

Software teams often stay busy without making meaningful progress. This article shows how OKR frameworks help engineering teams align with business goals, focus on outcomes over output, and ship software that delivers real impact faster and smarter.

Too often, software teams fall into a trap. Sprint after sprint, they crank out tickets, squash bugs, and push releases. Yet somehow, progress always feels scattered. Business goals shift. Priorities blur. And not everyone is on the same page and quite sure about what success actually looks like.

That's where OKRs come in.

OKRs (short for Objectives and Key Results) give software teams a simple framework to focus their energy with synergy. They create clarity. They connect the dots between code and company goals. And most importantly, they help teams deliver work that matters, not just work that gets done.

In this article, we'll break down how engineering teams can use OKRs to ship better, faster, and smarter. Not in theory, but in real, practical, "let's-do-this" ways.

Aligning Engineering Goals with Business Outcomes

Your team could build the cleanest codebase on the planet, but if it's not moving the business forward, it doesn't matter.

One of the biggest wins with OKRs is alignment. They force engineering teams to zoom out and ask: what are we actually trying to achieve? And more importantly, why does it matter?

Let's say the company's goal is to grow revenue by 30%. You could tie that to an engineering objective such as:

Objective: Improve user onboarding to increase activation rates

Key Results:

- Reduce average onboarding time from 15 minutes to under 5
- Increase week-one retention from 35% to 50%
- Cut critical onboarding bugs by 80%

See the shift? This way, your team is now not just building features but delivering outcomes.

That's the mindset OKRs promote. They nudge engineering teams to think like product owners, to connect commits with customer impact, and to measure progress in results, not activity.

Setting Engineering-Specific OKRs

When it comes to engineering teams, the best OKRs are grounded in real problems and tied to things you can actually measure.

So how do you choose the right ones? Start with a solid framework and understanding (something you can dig up from resources like the OKRs Tool blog). Look at your team's current pain points and performance metrics. These three lenses usually help:

- DORA metrics (deployment frequency, lead time, change failure rate, mean time to recovery)
- Code quality (bugs, test coverage, maintainability)
- Scalability + reliability (latency, uptime, incident counts)

Don't pick metrics just because they look good. Pick ones that highlight what's slowing your team down or putting users at risk.

Here are a few examples of engineering OKRs:

Delivery Velocity

Objective: Ship features faster without breaking things

Key Results:

- Increase deployment frequency from weekly to daily
- Reduce lead time from code commit to production from 3 days to 1
- Automate 90% of regression testing

Technical Debt Reduction

Objective: Tame the legacy beast

Key Results:

- Refactor 5 high-risk modules
- Cut related production incidents to zero
- Improve code readability scores (or developer feedback) by 30%

Code Quality & Test Coverage

Objective: Catch more bugs before they hit prod

Key Results:

- Raise test coverage from 65% to 85%
- Reduce post-release bug reports by 50%
- Implement automated code reviews on all repos

DevOps & Deployment Automation

Objective: Make deployments secure and fast

Key Results:

- Cut CI/CD time by 40%
- Achieve zero manual steps in the release pipeline
- Migrate all services to blue-green or canary deployments

It's tempting to shoot for the stars. But moonshot key results can wreck morale fast if they're not realistic.

Here's how to keep your OKRs grounded:

- Don't confuse OKRs with dreams. They should stretch the team, not break them.

- Avoid binary KRs like “eliminate all bugs.” That’s not helpful.
- Use data from past sprints or incidents to set a realistic baseline.
- Check in early and often. If a KR is way off track, adjust it. Don't let it linger and demotivate everyone.

Bottom line? Good engineering OKRs are a mix of ambition and practicality. They push your team to grow without setting them up to fail.

Cross-Functional Collaboration Through Shared OKRs

Software doesn’t get built in a vacuum. Engineers work with product, design, QA, support—the whole crew. And that’s exactly why shared OKRs matter.

They help teams move in the same direction, not just fast.

Let’s say the product team wants to improve activation. The design team is simplifying UX. The support team’s flooded with onboarding questions. A shared OKR might look like this:

Objective: Make the first 5 minutes in the app delightful and frictionless

Key Results:

- Cut new user drop-off rate by 50%
- Resolve top 3 onboarding support issues
- Increase user task completion rate in first session by 30%

Everyone contributes. Everyone wins.

Put simply, shared OKRs build alignment, reduce handoff chaos, and encourage more fruitful conversations. They remind people they’re not building in isolation. They’re building something together, and that mindset shift alone can level up your team’s output.

Tracking Progress and Measuring What Matters

Don't treat OKRs like a separate thing you look at once a quarter. Make them part of your day-to-day.

Most teams already use tools like Jira, Linear, or ClickUp. This means you can link key results directly to epics or tickets:

- Tag tickets that contribute to a specific KR
- Use labels like OKR-Q3-2024 to group related work
- Show KR status in sprint planning or stand-ups

That way, every sprint has a purpose, and your team sees how their work connects to bigger goals.

In fact, a clear, visible dashboard makes a huge difference. It keeps OKRs top-of-mind, not buried in a doc somewhere.

Some tools that make this easy:

- Gtmhub, Perdoo, and Koan as OKR-first platforms
- Notion and Confluence for custom dashboards
- Google Sheets, if you want to keep it super simple

Use color codes (green/yellow/red), auto-progress bars, and regular status notes. The simpler and more visual, the better.

Also, don't wait until the end of the quarter. Check in regularly. A quick OKR review every week (or every other week) helps:

- Spot blockers early
- Keep KRs from slipping into "we'll get to it later" land
- Create space to adjust if priorities shift

You don't need a full meeting. Even a 10-minute async update can work.

Besides, not every key result will hit the mark. That's okay. What matters is what you do about it. Here's what to ask:

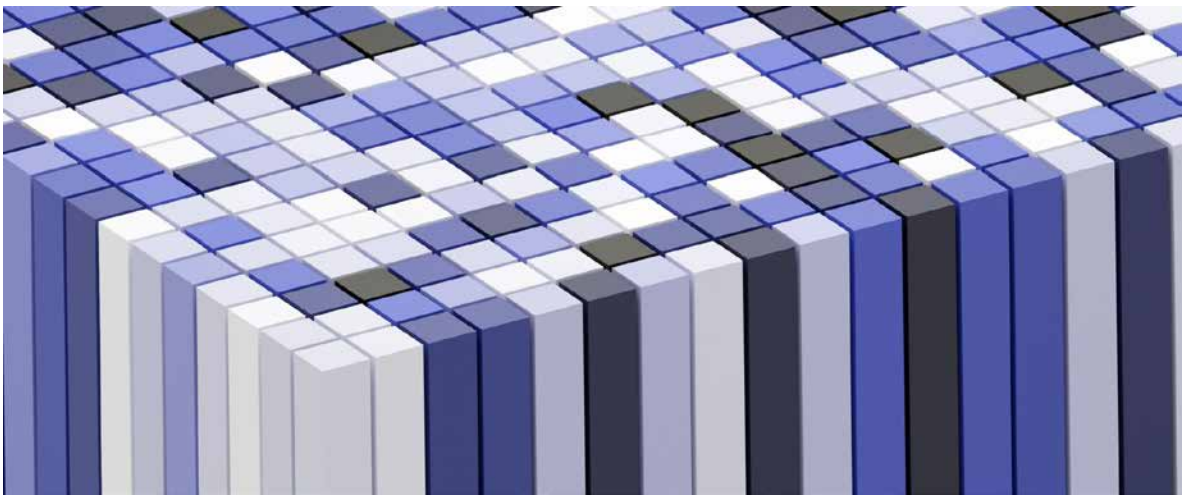
- Is the KR still relevant? Has something changed?
- Are we under-resourced or over-scoped?
- Can we break it down or reset the target?

If it's still worth pursuing, update the plan. If it's no longer a priority, drop it with intention. Better to adjust than pretend everything's fine. Ultimately, the goal isn't a perfect OKR scorecard. It's to stay focused, make better decisions, and course-correct in real time. That's how progress compounds.

Conclusion

OKRs aren't about doing more. They're about doing what matters. For software teams, that means connecting day-to-day work with real outcomes. It means shipping with purpose, not just speed. And it means growing as a team, one cycle at a time.

Start small. Set one objective that truly matters this quarter. Involve your team. Track it visibly. Reflect often. You'll be surprised how fast this newfound clarity leads to insane momentum and results.



APIs: Fundamentals on the Web - Part 1

Why APIs Should Be Designed Independently

Uwe Neukam, Noah Neukam

14. April 2026

APIs are long-term business and technical contracts, not just technical interfaces. Therefore, they must be designed independently from backend systems to ensure stability, scalability, and sustainable evolution in a networked and microservice-driven world. This article will help point out requirements for APIs and explain differences between common API types.

Even before the term API (Application Programming Interface) became well-known, developers exchanged information in the form of data. This worked in a few different ways, like using defined interfaces in order to use a direct electronic path. Changes weren't welcome, and many commonly said that "interfaces are carved in stone".

This article series will prove that API management has become significantly more important in a network-oriented world. We'll help point out ways to meet this requirement.

Is There a World Without Interfaces?

Of course, the question is purely rhetorical, because wherever we look, we find interfaces in many different forms. The best-known interface definition is the one for electrical devices. They must meet a certain specification so that they can consume electricity in order to function at all. They must also meet the specification for plugs so that they can be connected to a standardized socket. This combination has become normal in our daily lives and we use it without thinking about it much. And yet, this type of communication is based on a previously made agreement about a specification. This alone isn't enough for success or gaining broad acceptance. What matters above all is trust and reliability. These two qualities have a high economic value. Device manufacturers trust that electricity providers reliably supply power through sockets and via the specified plug type. Electricity providers trust that device manufacturers won't draw more voltage and/or current than directed in the specification for the defined socket type. And that works without problems, doesn't it?

Back to IT

Here, things are exactly the same. We live from collecting data, analyzing it, modifying it, calculating it, and storing it. It's also important to be able to exchange it with others— ideally both for the benefit of our business goals and the goals of our connected consumers.

In this environment, the same quality principles of trust and reliability apply as assumed for device manufacturers and electricity providers.

What is an API?

What distinguishes an Application Programming Interface (API) from a simple interface? If we consult the internet to answer this question, there are many definitions. Here are a few excerpts:

- "APIs (Application Programming Interfaces) consist of multiple definitions and protocols for developing and integrating application software. An API is an interface that allows independent applications to communicate and exchange data."
- "APIs are mechanisms that enable two software components to communicate with each other via a set of definitions and protocols ..."
- "An Application Programming Interface (API) is a set of defined rules for communication between different applications. It acts as an intermediary layer and, as such, processes data transfers between systems. In this way, companies can make their application data and functionality accessible to external developers, business partners, and also internal departments within their respective organizations."

These sources are largely in agreement: it's about a connection between two systems in order to communicate with each other. Yes, that's almost it.

APIs describe the manner (see: socket design, current, voltage) of communication so that other systems (applications) can use their functionalities or data. For this to work, the definition must be converted into program code and a functioning interface (plug) must be created from it.

Interestingly, this can be done independently by each participant for their own context (programming language, framework). Thus, it's not important for developers to know how an API is implemented. They simply use the defined interface in their environment.

What Are We Talking About?

There are many kinds of different APIs. There are specifications that allow access to databases or, more generally, to hard disks. Some APIs

make it possible to create components for graphical user interfaces (for example, Microsoft's Windows Application Programming Interface) In this series, we'll primarily consider "HTTP APIs," also often referred to as web services.

Now some readers may ask: "Why? It's not that hard to formulate APIs. Plenty of tools support this. Extensions and changes aren't a problem either because we know all of the users."

We can easily imagine these statements because APIs are usually created for internal use. But since Google Maps, we also know that this can change very quickly. Suddenly, the API is available on the web and it becomes more difficult to know one's users.

This scenario is becoming increasingly important as companies have realized that APIs can be monetized. This isn't the only reason that good design is necessary. Interfaces have gained importance within companies. With the trend toward microservices, more teams have their own service and changes to interfaces involve more effort for consumers as well as provider teams than they did in the past.

From a business perspective, they've become more relevant as cost and effort drivers. Changes to APIs are a magnifying glass for the relationships between teams and the business responsibilities they represent. The business, social, and technical level is often relevant in order to know the correct response when two consumers have different requirements for an existing API.

Platform APIs vs. Service APIs

In most cases, APIs are used to exchange information (data). There are two models that ensure a user can consume information, but in different forms and with different motivations.

Platform APIs

Also called data APIs, these are closer to data organization and management, and are modeled accordingly. Platform APIs are characterized by the fact that they represent generic interfaces for reusing multiple services with similar data sets. This model is often seen as the base layer for an API ecosystem.

If we adopt this perspective, one could also say that they represent the first representation. Transformations are usually required for use.

Service APIs

Service APIs (or business APIs) are used to present specific data. For this purpose, they can use aggregated or composite representations of data from platform APIs. They have additional data processing to cover a specific requirement (use cases).

Core Question: Do I Actually Know My Stakeholders?

Both models are very different and have different motivations—keyword: "stakeholder management." Regardless of whether customers can be asked directly, if assumptions must be made, or user behavior analyzed, knowing your stakeholders is worthwhile.

In design, there's a reason that people say "form follows function". The following guiding questions help us understand customers:

- Why is the API needed?
- Who should call it or program against it?
- What information is interesting for consumers?
- How frequently or extensively can the API be used?

How exactly an API is derived and what pitfalls should be considered will be addressed in the second part of this series.

And Then There's The Reality

No matter how much caution, dedication, and passion went into modeling an API, consumers can have a completely different view. Some may say, "the customer is always right in matters of taste", but that isn't always true.

An API's success doesn't solely depend on its technical design, but on many other factors too. Anyone who knows the concept of switching costs from business administration knows that people adapt quickly. So what are the other factors?

Context

If consumers operate in the same or a similar context, the probability is high—perhaps even very high—that they'll have a similar view of things and accept an API design determined by the context.

Environment

This behaves similarly to context if the environment is equated with industry-typical solutions. But it already gets harder when the consumer comes from a different environment (industry) and needs the API for their business success.

Market Power and/or Dependency

This is the extended version of the environment, along the lines of "consumers will follow me; they have no alternative."

Free or Paid

This aspect can shift the prerequisites about if a habitation effect occurs or not. Those who pay for an API are more likely to want a say in its design and further development than if it was free to use. And yet, we providers have a strong interest in designing our API cleanly. These two quotes support our point:

- “APIs are forever” (Werner Vogels, Amazon)
- “An API should not be strictly coupled to the architecture of the backend” (Daniel Kocot, codecentric)

Both statements, from different perspectives, have something to do with customer perspective and customer retention. Consumers usually operate in their environment, with their own problems and requirements. They aren’t very interested in an API’s design weakness or the necessary changes behind APIs on offer. There’s a willingness to accept a somewhat clunky API. But a willingness to accept constant changes, even without a visible customer benefit? There’s significantly less of that. In the worst case, you offer a clunky API that cannot be changed because it has paying users. But you also cannot create a second, better API if it’s tied to the backend’s technology. It’s questionable whether new users can be won over with a clunky API.

The way out of this situation is costly, involves painful architectural or business decisions, and is also risky. We already see that APIs can be a real minefield, either internally or externally. They’re more than just a simple interface that can be programmed against.

They depend on many things, which is why API design can appear quite complex. And sometimes it is. In the following articles we will address a few aspects directly and show how to deal with them and potentially defuse them.

Achieve optimal performance, security, and cost efficiency

7 Key Considerations for Choosing Container Base Images for Java Apps

Dmitry Chuyko

14. April 2026

This article provides a comprehensive guide to selecting container base images for Java applications, explaining how base image choice affects security, performance, cost efficiency, and operational usability in containerized environments. Learn important considerations for optimizing your Java applications by thinking in a holistic, integrated fashion.

At first glance, choosing the best container base image for a Java application may seem simple enough. Teams have a tendency to approach the issue by optimizing layer-by-layer: They choose the smallest base image for efficiency, pick a reputable JDK distribution,

apply Java-level optimizations for performance and harden the image for security.

On paper, this strategy might seem like the best way to create a base image that maximizes security, performance and usability. But in reality, it often results in images that are less than the sum of their parts.

Why? Because optimal performance, security, and cost efficiency don't come from assembling the best individual pieces. They occur when all container components work together seamlessly as an integrated system – and that type of synergy is nearly impossible to achieve when your OS, runtime and tooling come from different sources, and focus on distinct and siloed goals.

Hence the importance of thinking in a holistic, integrated fashion about container base image sourcing. Read on for a deep and practical guide into the many considerations to weigh, along with actionable tips on selecting the ideal container base image for your Java app.

The what and why of selecting container base images

Let's start by defining what a container base image is and which role it plays in application development and deployment.

Defining and using base images

A container base image is the core code that developers use to create a containerized application. Base images for Java apps typically include operating system components, dependencies for the JDK, various utilities and the JDK itself. You can build on top of these to create the custom container environment your app needs to run.

To select a base image, you specify its name when creating a Dockerfile. For example, to start with a base image built using Alpine Linux and the Liberica JDK, you'd include a line like this in the Dockerfile:

```
FROM docker pull bellsoft/liberica-openjdk-alpine:25
```

Why base image selection matters

Virtually all base images do the same basic thing: They provide a foundation for creating a containerized app.

That said, the code and utilities included in base images can vary widely. Some base images provide full-fledged operating systems with almost as many programs and utilities as you'd find in a desktop version of Linux. This is generally a good thing if your application actually needs all that code. It saves you from having to add it yourself.

Other base images are extremely minimalist and don't even include a shell or package manager. This can be advantageous for scenarios where application performance, resource utilization optimization and security are key priorities.

And then, of course, there are base images that fall somewhere in between – ones that include, for example, a Java runtime along with a shell, but not a package manager.

The type of base image that is best for a given project varies widely depending on which type of application you're deploying and what it needs to do. We'll walk through the key considerations to weigh when selecting a base image in the next section. But for now, the takeaway is that base images are not interchangeable, and it's an extremely poor idea to default to using whichever base image is most popular, most familiar or the most readily available. If you do, you're likely to end up with an application that falls short in the realms of performance, security and beyond.

Beyond base images: Other considerations for optimizing Java apps

Of course, the base image you select isn't the sole factor in shaping the performance, security, and usability of a Java app. Also important are:

- How you create the application: Steps like optimizing your runtime (through tools like jlink), building GraalVM-native apps and optimizing your build system (whether it's Gradle, Maven or something else) play critical roles in shaping overall application performance.
- How you launch your application: The contents of your Dockerfile or buildpack have important implications for both security and performance. So do considerations like which orchestrator you use (if any) and how it's configured.

Thus, it's crucial to think not just about how inherently optimized your base image is, but also how well it supports optimizations at later stages of the development process. For instance, if the image lacks the tooling to enable an optimized JRE, you're likely to achieve lower overall application performance, even if the base image itself is "performance-optimized" in the sense that it includes no unnecessary components.

What to consider when selecting a container base image

Now that we've walked through the role that base images play in Java application optimization, let's dive into key considerations for choosing a base image.

To get a sense of what development teams should think about when choosing container base images, my company surveyed nearly 500 actual developers. Following are what they said they prioritize when comparing base image options, along with context on why each consideration matters and how to optimize it for your Java project.

1. Security

29 percent of developers said their top priority when choosing base images is security. Specifically, they look for images that have the fewest number of known Common Vulnerabilities and Exposures (CVEs), which document known security flaws in software. This makes sense because no one wants their app to be hacked, and the more CVEs that affect a container, the more chances threat actors have to breach it.

That said, minimizing CVE count also often means choosing a minimalist base image (since the less code you include in your container, the fewer components you'll have that can be impacted by CVEs). And as we'll explain later in this article, minimalism may come at the expense of operational convenience.

2. Performance

The next most-popular factor that developers cited when asked how they choose base images is performance. They want images that will help ensure their applications start up quickly and are as responsive as possible once they are running. 21 percent of developers named performance as their top consideration.

The default strategy here is usually to focus on a base image that is as minimalist as possible. But that's not actually always the best approach. For example, while a stripped-down image might have a small disk and memory footprint, this could come at the expense of a slower startup process. (Indeed, part of the reason why my team created Alpaquita Linux was to solve startup delays in Alpine Linux that stemmed from this very issue.)

It's also important to select base images that include code that is optimized for your particular use case. For example, when deploying a Java app, you're likely to achieve better performance if you use a base image that includes a Java runtime that has been tuned to maximize

performance in a containerized environment, as opposed to using the Java that ships with more generic base images.

3. Image size

Raw image size – the main consideration for 17 percent of developers we surveyed – is also an understandable factor to weigh. Larger images take up more storage space and generate more network traffic, which can in turn lead to higher cloud bills (if you store images in the cloud).

Large images also take longer to download, which can delay the application deployment and provisioning process if you're deploying an app using Docker or Kubernetes and you need to pull its image from a remote repository first.

An important nuance to bear in mind, however, is that large images are not always a challenge. Teams that run their own image repositories using local storage may not be as worried about storage space because they don't have to pay a monthly bill for it. Plus, if you're pulling an image over the local network, it generally will be much faster than pulling one via the Internet.

But local repositories tend to be the exception, not the norm – so it makes sense why many developers would look for base images that are small in size.

4. Built-in Java support

17 percent of developers also reported that their main consideration is how well a base image supports Java. If you're a Java developer, you can probably understand why. Java apps can be particularly complex to develop and deploy due to the diversity of Java runtimes and versions. It's a real hassle – not to mention a potential security risk – to have to create or modify a container image's Java environment due to limited or non-existent Java support in the base image.

To address this challenge, it helps to choose a base image that supports optimization tooling like CRaC and GraalVM, which can help reduce startup time to almost zero. This is another example of why it's important to think about not only which type of Java your image supports, but also what the performance implications of its Java capabilities are.

5. Ease of use and operational efficiency

11 percent of developers told us they prioritize ease of use and operational efficiency when selecting base images. Specifically, they consider which tools and utilities base images include. Minimalist images may help with performance and security, as we mentioned, but their drawbacks can outweigh their benefits if the images are so bare bones that they lack key utilities such as a shell and package manager. In that case, it becomes very difficult to run commands or add software.

Hence the importance of choosing a base image that balances ease of use on the one hand with performance and security on the other. Selecting images that include utilities you don't need is a bad idea, but so is being so minimalist that you make your life as a developer harder than it needs to be. You already have a lot on your plate and figuring out how to do things like add a shell to a "blank" container image need not be one of them.

6. License compliance

An easily overlooked aspect of base image selection is licensing compliance, meaning whether the software included in an image conforms with the various open source and/or commercial licenses that govern it.

This is important because using software in ways that violate licenses can lead to compliance risks and lawsuits. Plus, researching licensing details can be a time-consuming and frustrating endeavor for

developers. This is especially true in the Java ecosystem, where Oracle's complex licensing terms have a tendency to change unpredictably.

To avoid licensing compliance issues, it's a best practice to choose base images with clear-cut licensing terms. We're talking here not just about the license for the operating system that the image uses, but also for the various individual runtimes, tools and so on within it.

Base images created using generic Linux distributions don't always have clear licensing terms because they include so many components that the projects that develop them rarely take the time to research or guarantee licensing compliance. But base images do exist that include clear and specific software licensing details.

Only 4 percent of developers we surveyed said that licensing compliance is their top consideration when evaluating base image options. We can hope, however, that for many of those who didn't say that licensing is their number-one concern, it's still something they pay attention to.

7. Support and roadmap

For most Java development teams, the work doesn't start when an app has been deployed. It continues indefinitely as developers update and redeploy the app. For this reason, it's important to consider the support and roadmap outlook for your base image. Ideally, you'll select an image that is maintained by an established vendor, which – unlike community projects – is not likely to abandon an image that you depend on to run your apps. Also important to weigh is whether the image includes proprietary tooling, which could create lock-in risks.

A "best of both worlds" solution is one that combines vendor support with open source software, giving teams the assurance of reliable support without locking them into a commercial technology ecosystem.

Best practices for choosing a "best-of-all-worlds" image

In many cases, the various considerations we just described may seem to be in conflict. A base image that is secure because it is minimalist may come at the expense of a positive developer experience, for example.

But as we said in the introduction, the key to resolving these tensions is to choose a base image generated through a holistic process that considers security, performance, and usability in equal measure. When you source images from projects or vendors that specialize in all of these areas, you avoid the pitfalls that would arise if you chose disparate components, each optimized in isolation for a different priority.

To be more specific, a "best-of-all-worlds" experience comes when your base image strategy is guided by the following principles:

- **Assess image components, not size:** Smaller is not always better from a performance, security and licensing compliance perspective. Rather than choosing an image simply because it's tiny, pay attention to the details of what it actually includes, and what their implications are for performance and security.
- **Evaluate upstream security practices:** In addition to considering how many CVEs impact an image, assess how active the image's developers are in discovering and mitigating relevant vulnerabilities. Just because an image has few open CVEs currently doesn't necessarily mean the image's vendor will excel in patching new CVEs when they emerge. Sourcing images from projects with a solid track record of CVE mitigation will go far in minimizing security risks, regardless of overall image size or complexity.
- **Consider support availability:** On a similar note, assess how active an image's developers are in providing support. If something goes wrong – for example, if you run into a Java compatibility issue or are affected by zero-day vulnerability – will they help? If so, you're likely to enjoy a faster, smoother resolution process than you would if you have to sort out issues like these on your own.

- **Choose optimized software components:** As we mentioned, software optimization can be at least as important a factor in shaping application performance as overall image size. For that reason, look for images whose runtimes and libraries are configured out-of-the-box to optimize performance.
- **Don't overlook licensing:** Software licensing may seem like a boring topic, but it's a critical one from the perspective of the business you work for. Avoid compliance headaches – and having to spend hours sorting out licensing details manually – by using images with clear licensing terms.

The bottom line: When deploying your next Java app, don't choose base images based on simple metrics like total size or total CVE count. Instead, assess the full picture by considering how an image's security, performance, usability and support capabilities add up to make your app the best it can be.

APIs: Fundamentals on the Web - Part 2

Approaches to API Design

Uwe Neukam, Noah Neukam

14. April 2026

This article addresses the three important things that must be clearly understood to understand the API consumer's technical expertise: quality, information, and behavior. Answering these central questions will help keep APIs stable, understand what stakeholders need, and improve the usability of APIs for developers.

The first part of this series looked at why APIs (application programming interfaces) are so attractive. They simplify the way in which at least two parties (provider and consumer) communicate and exchange data. But this raises the question: How can this be reliably achieved? After all, requirements that involved parties often place on an interface couldn't be more different. We want to address those requirements and show that it isn't impossible to meet them.

It may come as a surprise to some, but the same mechanisms are used to design both APIs and applications. Although the perspective is

slightly different, the approach is largely the same. There are several different methods and support models for approaching the problem.

Necessary questions

In the first part of the series, we briefly touched on the questions that we should ask. This time, we want to explore them in more depth, since they can be crucial for creating an appropriate design.

- **Why is the API necessary?** This is a key question in our world of technically motivated applications. In other words, drivers for development increasingly come from the technical (functional) environment. So it's important to find out what motivation lies behind a requirement and what added value it has. This aspect should not be underestimated.
- **Who is the API aimed at?** It comes as no surprise that an API is aimed at developers. They are supposed to use it and program against it. However, the question isn't about the user's role, but their environment and context as a consumer.
- **What information is important to the consumer(s)?** This can quickly become complex. The basic questions are: Do I know the consumer? Is there only this one consumer? If there are multiple consumers, do they all operate in the same environment? In this context, it's important to find out how consumers want to consume the information.
- **How often or to what extent can the API be used?** Beyond the content, it is important to know how high the frequency or capacity must be in order to meet consumer requirements.

Why should you ask yourself all these questions—and a whole host of others? Because it's all about trust and reliability. The central question is always how stable the interface is. Interestingly, this applies to both sides of the API.

The consumer expects it because they have their own problems to deal with in their context. They trust that they can reliably retrieve or submit information for their needs. On the other hand, the provider wants to

be able to adapt or change their implementation (e.g., connect other data storage systems) independently of consumers.

“Yes, but that's nothing new, and developers are well aware of it!” It wasn't so long ago that statements like this were made. Nevertheless, we continue to see APIs being designed with a focus on specific, mostly technical aspects. We can use this to take a closer look at API design. Three things need to be internalized in order to understand the API consumer’s technical expertise: quality, information, and behavior.

Quality

This point may seem surprising at first, because quality is usually understood as a product feature and has only limited relevance to technical considerations. Nevertheless, we must address the technically motivated aspects of quality. For example, we’d like to cite something that’s well known in the database field: consistency guarantee. Microsoft's Terry Douglas once elaborated on this excellently in an article: “If we want to report on a baseball game, the medium is crucial. Depending on the reporting stakeholder, a different guarantee of data consistency is required,” he writes. This also applies to APIs and as we can see from the example of the “reporter,” this is purely technically motivated!

It’s often particularly important to monitor the timeliness of data and consistency requirements: the technical transactions. Is there information that must be consistent, like shopping cart contents and the corresponding total amount, or are deviations and delays permitted? This could be the case with a shopping cart and delivery. Just because a customer places an order using a shopping cart doesn’t necessarily mean that a delivery order must be generated for everything at the same time.

If we can obtain such information, then we can draw conclusions about our API design. Can we offer multiple APIs for this information, or does that not make sense due to technical constraints? Is only one

stakeholder interested in this information, or will there be a chain of queries? Does the information need to have the same level of availability and be linked, or do we have the freedom to separate it? The goal is not to find a suitable architecture pattern or a good service interface (which, mind you, should also be done with this information!). The goal is to find connections in the requirements that influence our design. We must understand which things will change together and which can be separated.

A deep understanding of quality requirements helps keep the API stable in the long term, as changes can be limited.

Information

It goes without saying that it has to be clear what information an API needs to provide. This is only mentioned here for the sake of formality. However, we'd like to raise awareness of this issue. An API that spits out all the information relevant to stakeholders with a single request sounds great at first glance, doesn't it? In fact, this approach would be completely unsuitable for most APIs and would result in constant changes. We actually have a different challenge. Namely, understanding the consumer(s) information interests without creating coupling that drags the provider along with every turn.

An example: As a consumer, it's often the case that information is either filtered at an endpoint using the API or a query's result is filtered again in the client. Depending on this filtering, follow-up queries are made on sub-resources, for example. A good API chooses a valid abstraction here. This is a filtering or representation that meets the consumer's interests but doesn't necessarily have to change if the consumer adjusts their behavior or decides to filter differently. We know an excellent description of this situation from the artificial intelligence field. We're referring to overfitting, as shown in **Figure 1**.

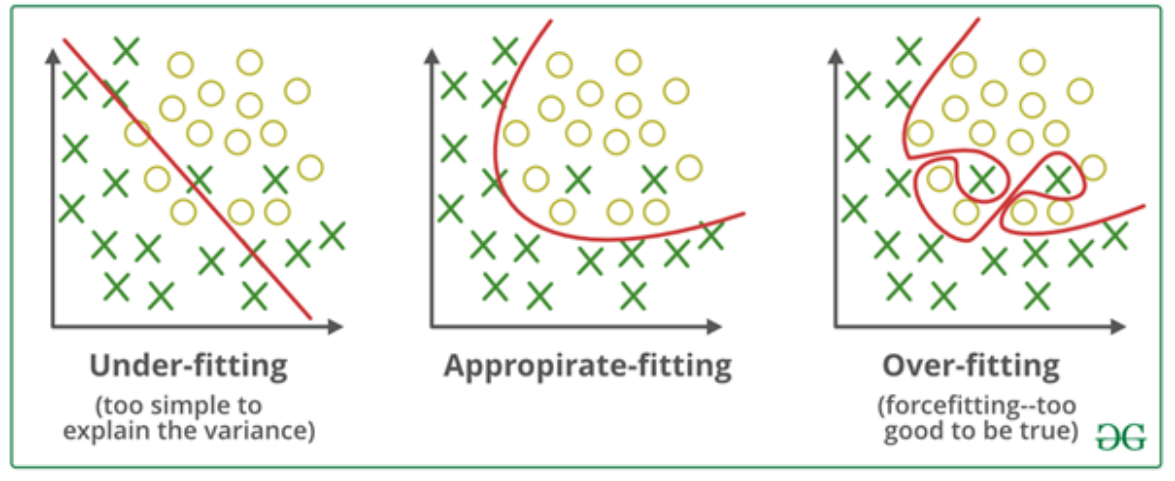


Fig. 1: Different forms of adaptation (Image source)

There's great joy when using an overfitted API for the first time. Everything is perfect, just as we'd like. But as soon as the customer's expertise and their information interests change, we realize that the API isn't elastic! A certain degree of elasticity increases usability and creates resilience to another form of change we've already encountered: change driven by consumer interest.

Elasticity arises when we understand what information is of interest to our consumers, why it is of interest, and when and in what order they will request it. However, we also need to understand how these interests will develop. Anything likely to change should be abstracted or added as an extra. Don't worry, you'll often be wrong. The statement that software architecture involves "limited clairvoyance" doesn't come from nowhere.

Behavior

In the world of API design, the term "behavior" has two meanings. On one hand, it describes the design of technical behavior, which deals with questions such as: "If two filters are set, is their relationship additive or alternative?" On the other hand, it involves the semantics of technical language: "If a domain object or resource is in a certain state, what can happen? What impact does this have on others?" Ideally,

these two factors are related or, more precisely, the technical behavior should follow the behavior of the technical field.

It is important to note that the technical aspects of API design have to be considered keeping the specific context in mind. A stakeholder or stakeholder group's view of the technical aspects describing behaviors or actions may not necessarily occur in other views (**Fig. 2**). They may also be authorized to perform additional role-specific actions.

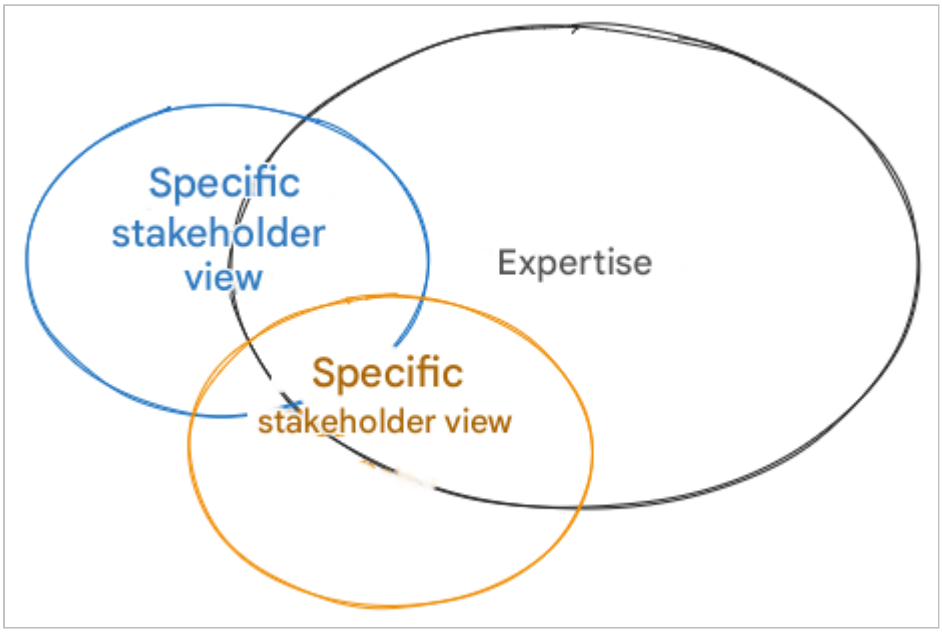


Fig. 2: Different perspectives on professionalism

We must therefore determine what understanding our stakeholders have of professionalism and whether this is accompanied by specific behavior. We should also understand whether this perspective interacts with other stakeholders or can be considered in isolation.

The best way to understand behavior is to imagine it as a language. Languages have certain rules and sequences that make sense because they follow the language's semantics. For example, if something relates to the verb "open," we expect it to have a state of "closed." This can be metaphorical, like a supermarket's in the opening hours, or physical, like a door. If something cannot be "closed," it makes no sense to deal with its "opening." If we consider a technical field as a language, we

come to another frustrating aspect of language learning: hidden rules. Let's take a simple English expression as an example: “the round big ball.”

The round big ball

The expression is easy to translate. “The round big ball” sounds natural to many people – but it causes native English speakers to jump and say that it’s wrong. However, native speakers often cannot explain why. It’s a bit of a special case here, because when several adjectives are used, there’s a fixed, mostly unconscious order: “opinion, size, material quality, shape, age, color, origin, material, type, purpose.” So the correct phrase would be “the big round ball.”

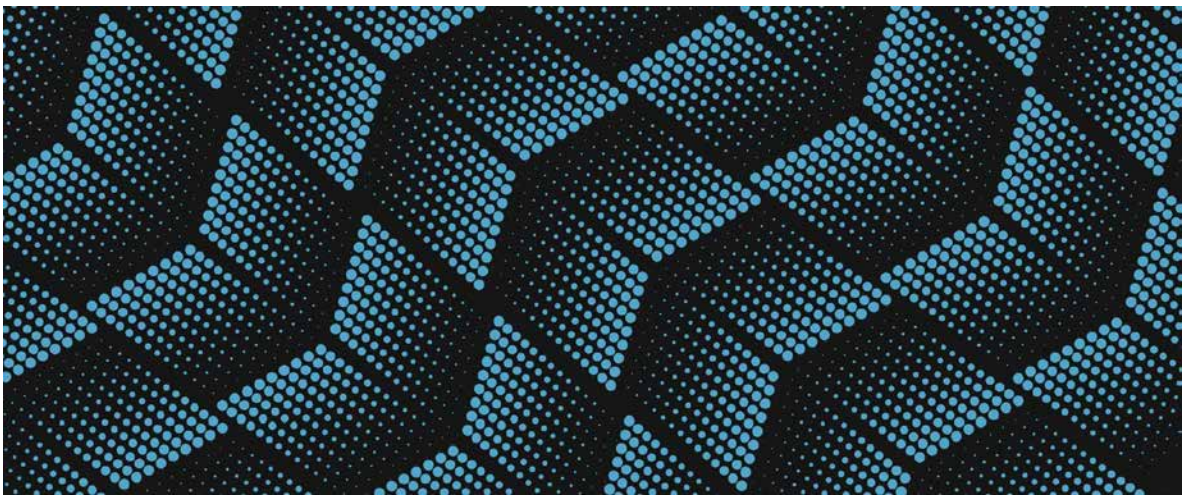
What can we learn from this? First, it’s essential to verify our worldview with stakeholders and subject matter experts when we learn a language. This is the only way we can recognize nuances and hidden rules. Second, we must engage with the technical language. If we constantly and immediately translate things back to our own native language, whether that be “technical design” or “English,” we might lose important insights.

Once we’ve gained an understanding of behavior, we can define the most important basic strategies. At the top of the list is the question: “How many APIs for which stakeholders”. As a result, we can make decisions at lower altitudes, like our previously mentioned question about filters.

Conclusion

Regardless of methodology, good API design is based on understanding coupled with technical skills. This works best with the description of business processes (use cases). To find these, you need experts who know the processes and can explain them—especially to developers. There are a wide variety of collaborative modeling methodologies

(event storming and domain storytelling) from domain-driven design that helps us understand the facts. After all, the task in an API environment is to make this technical construct interesting for developers, since they're the ones who will use the API – and it's best if they enjoy it.



APIs: Fundamentals on the Web - Part 3

API Anti-Patterns

Uwe Neukam, Noah Neukam

14. April 2026

We've learned why APIs are so important and discovered that designing APIs requires at least as much effort as good application design. In this article, we'll focus on established development patterns that don't necessarily hinder API design, but can make further development difficult or lead to the interface breaking. Learn what anti-patterns exist, what the observed behavior is, and how to counter it.

"Interfaces are set in stone" – as if! This saying is as old as the violations against it. The interesting thing is that most developers get upset when they're affected by changes. However, they usually have no problem changing the interface themselves. With today's tools, adapting to change can't be that difficult. Yes, you'd think so, but let's stop right here. No, it isn't pleasant to constantly have to adapt to new circumstances. And yes, I understand when the interface changes due to technical reasons.

But technical expertise doesn't constantly change and most of the time, the world isn't completely different. And when this does happen, the old rules usually still apply to some extent (keyword: grandfathering). These insights actually already apply in "normal" application development with integrating interfaces from used libraries (a very limited area of influence), and even more so when it comes to HTTP APIs (a very broad area of influence), since there we most likely don't know all of our users. As we briefly already touched upon Part 2, it's all about reliability and trust.

But what causes APIs to be less stable than needed?

Most cases have something to do with experience and approaches. Another point lies in the organizational solution spaces where developers operate. Just as users have their own problem space, developers who create the solutions move in their problem spaces and work through their requirements one after the other. Usually, this is done with the certainty that they're only operating in their solution space. And so it comes to pass that a solution that was so simple only a moment ago must suddenly be universal. No one is prepared for that.

We'd like to highlight a few patterns, examine their causes, and present ideas on how to avoid them in the future. First, it must be noted that all of these patterns are characterized by a high frequency of change and are likely to frequently break existing interfaces.

APIs and interfaces

What are we talking about?

APIs thrive on users being able to easily understand and use them. So it seems advisable to focus on an interface, especially if the protagonists involved are familiar with the interfaces.

Observed behavior

You would think “stable”, but this approach tends to lead to unreliability because it’s based on many assumptions that are usually incorrect. Frequent changes occur, if only because the user interface changes. Documentation cannot keep up with this. It might describe a completely different situation and functional completeness is often not given because it’s based on output at the user interface, which is subject to the vagaries of front-end requirements.

What are the causes?

We’ve already made a few suggestions. Perhaps the most serious cause is that the API isn’t considered independent and follows the front end (user interface). It is therefore used to correct problems in the front end. This reduces the API to a mouthpiece between the front end and the back end. The display-related states of the front end migrate to the back end, even though they aren’t necessary for business processes. Once there, they bring along the front end’s “change-happy” nature.

This approach is similar to the typical way we learned to build systems and so it’s widely used. We usually focus on the greatest benefits for the user. Human-machine communication occurs via user interfaces and process and data structuring, as these offer the greatest benefit. However, this leads to backends either existing solely as data storage for frontends and offering little functionality of their own, or attempting to keep pace with the frequency of changes by means of conversions (mappings).

How can I counter this?

Let’s take a different perspective and simply imagine the front end away. In most cases, APIs are designed to support a technical process. Approach your product owners and software architects and ask them to explain how the process works without user intervention (event storming or domain storytelling can help) and what information is needed

to run it correctly. Ask relevant questions about the process, if only to find out whether the API is even suitable for your purpose.

Of course, there are exceptions to the rule. Surface orientation can make sense and be used correctly because the API is purely a data carrier. But in most cases, technical problems are solved (e.g., different display media) and the technical aspects remain the same. Here, the pattern functions as a proxy and uses the actual technical API independently to communicate with the backend.

APIs and databases

What are we talking about?

The purpose of communication between applications is to exchange information (data). We organize this information in data storage systems (databases), which is why we invest a great deal of resources in data organization. The structures are usually clear and comprehensible, making them interesting for consumers. It makes sense to align the API with the data storage structures.

Observed behavior

Focusing on data storage structures makes interfaces difficult to use because the API either cannot meet the requirements or provides too much information. Even when it comes to behavior, like searching or filtering, these are geared towards data storage capabilities, rather than the requirements.

What are the causes?

Technology imposes limitations. Organizing and storing data has already been done and people are familiar with the technology. An API for this seems simple. But API design and database design usually have different motivations. Schemas for databases follow different rules

(e.g., normal form) and are often considered the only representation of subject matter expertise.

It's similar to alignment on the surface. As already mentioned, data design is a strenuous and time-consuming activity. Even minor shortcomings can have a disproportionate effect on performance, and every database access takes what feels like an eternity. But here, too, only the internal view is considered. Any changes made can have a major impact on users. Often, the data must be provided in other forms and formats. How do developers respond to these situations? Right, with conversions (mappings), just like on the surface. However, these only make the application more complex and larger, and at some point, the applications consist only of mappings.

How can I counter this?

A perspective change helps here too. Isolating the database level is advisable where it makes sense to do so. This is usually the case when specialized processes are followed. Although these processes usually work with the same data, they’re structured differently to suit consumer needs. In other words, it’s the API that needs to be optimized, not the database.

Don't overdo it with mappers. Tools like MapStruct only solve problems that we shouldn't really have. As stated, mappers make applications confusing and slow in the long run.

Enterprise architecture should set the technological framework. Diversity is particularly important here, because technologies influence how we think about our APIs.

HTTP (or REST) APIs aren’t always the right choice. This is especially true when it comes to generic interfaces. These are rather difficult to implement with HTTP APIs because both the technical and architectural specifications don’t allow intuitive work with the given artifacts (e.g., GET does not actually recognize a body). When it comes to generic

interfaces, it's important to consider whether the right technology is being used.

APIs as a byproduct and/or without standards

What are we talking about?

It's a question of approach. A lot of course code is created in the transformation towards architectures that promise greater flexibility. Data is actively exchanged and corresponding interface specifications are needed, but these are created based on the programming language used or offered in libraries. Given these circumstances and prerequisites, it isn't surprising that there may be a requirement to make these interfaces available to a wider audience. With tooling support, the effort needed for the API is quite manageable and can be quickly completed.

Observed behavior

Many frameworks offer direct or indirect support (with external libraries) for this approach of generating API specifications from existing code. These are intended to help simplify routine tasks, and let's be honest, API design is often seen as an extra routine. But this view has significant disadvantages:

- **High rate of change:** The API is also affected by changes or modifications to the code and is adjusted accordingly.
- **Enormous scope:** The API's size and depth is based on the scope of the underlying code, meaning that it represents the technical interface and not necessarily the consumer's functional (technical) requirements.
- **Difficult to use:** Although the interface covers the business logic comprehensively, it's still difficult to understand and use due to unfortunate cuts in the API.

What are the causes?

Beyond these behaviors, there are other aspects to consider in tool-supported approaches. For example, which standard a tool supports. Even minor versions can make a difference. Let's take the OpenAPI 3 standard as an example. Version 3.1 recognizes null values, whereas version 3.0 does not. If a code generator doesn't recognize this small difference, or only supports one of the versions, then the generator may have been incorrectly created for the runtime environment, leading to errors in use.

Furthermore, when using the design by code approach, very little consideration is given to the consumer(s). Relevant questions (e.g., what does the customer want?) that typically form the basis of customer-oriented offerings aren't asked; instead, the design is geared more toward technical thought patterns for our problem.

How can I counter this?

We're back to perspective. A change in perspective is primarily about a change in roles. It's about putting yourself in the shoes of the potential consumer (stakeholder) to understand their needs. This helps especially when deciding if a separate API should be offered for each consumer (type) or if one API will suffice. Why is this so important? APIs are intended to facilitate work through their technical offerings, but in many cases they also offer perspectives on technical aspects in order to access the (and ONLY the) necessary information. We can also assess if consumer interests diverge or may do so in the future.

So, approach your customers or stakeholders, get an idea of who they are, and explore their professional interests. The more information you have about their needs, the less information (or expertise) you need to expose, increasing resiliency to change.

API and the wrong places

What are we talking about?

We understand that APIs should be durable, consumer-oriented, and that it's advantageous to specify the interface definition independently of the source code. And yet, breaking changes occur that no one can really explain since we're already doing everything right. Breaking changes are especially unpleasant for consumers since they have to adapt to the changed circumstances.

Observed behavior

Consumers complain that interfaces are constantly changing, even though the provider hasn't actually changed anything. The interface is stable in terms of access, content has not changed either, and yet there's enormous dissatisfaction with the interface.

What happened?

- The calls remain the same, but the results are suddenly different, e.g., restructured.
- Information moved, and is either no longer available or is suddenly located somewhere else.
- I need more APIs than usual to obtain the same information.

What are the causes?

We've been struck by Conway's Law (organization) and/or Alberto Brandolini (technical misjudgments). These laws can lead to hidden breaking changes.

Why is that?

The reason for this is that organizational constraints (reorganizations make everything new) and misunderstandings about technical expertise and responsibilities can lead to unreliable results.

This in turn leads to a situation where either no abstractions or the wrong abstractions are used for an API interface. Or, as already indicated, the design is based on technical or organizational considerations. Assumptions are made and the API has to gradually adapt to reality.

The nasty thing about this is that these situations appear quite harmless at first. Let's take an example. We like to use enumerations in development. They are meaningful and easy to use. This also applies to APIs that comply with the OpenAPI standard.

So everything's fine?

Not really, because what happens if the enumeration is expanded (e.g., due to a lack of technical completeness)?

Yes, it's hard to believe, but for a consumer, this is a breaking change. They cannot understand the new entry in their environment and will encounter an error.

How can I counter this?

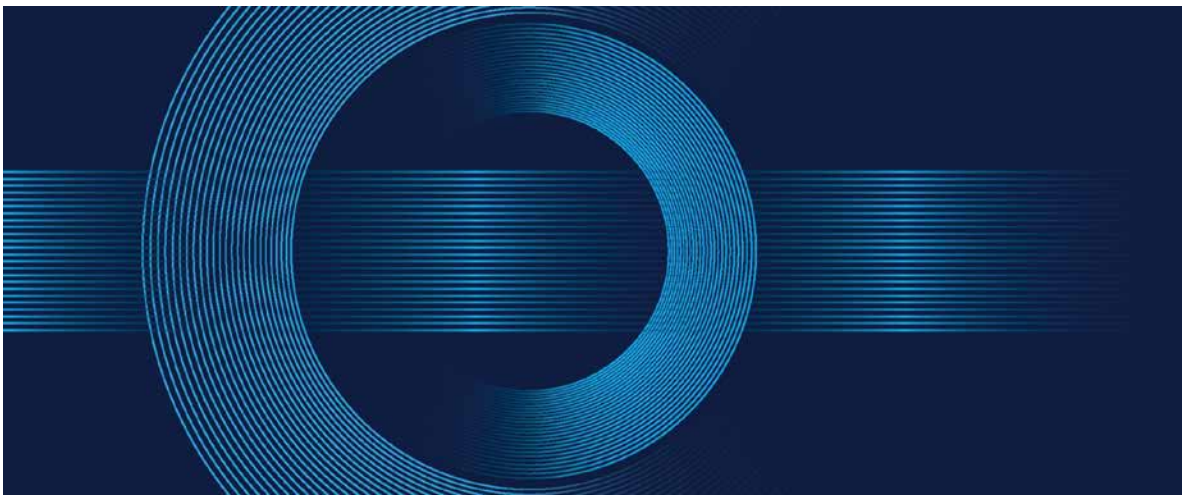
Just like with development, API design requires looking beyond the actual creation because changes will happen. Once again, we have to ask ourselves questions:

- "Am I on the same coherent level?" – No mixing technical and business statements together.
- "When will this API change?" – Preview it, because sometimes you will be forced to make a change (legal changes).
- "Is the motivation behind my abstractions correct?" – Why exactly is this cut?

In this context, it's also helpful to investigate if there are implicit behaviors that affect consumers when changes are made (see abstraction).

There are certainly more anti-patterns for good API design, but these are the ones we encountered most frequently. It's also important to recognize that breaking changes need multiple parties to be involved. Once again, it depends on how many consumers I have and if I know them all. The smaller the scope, the better I know my users, and the more opportunities there are to organize changes. As a general rule, changes should be made sparingly because they're unpleasant for all consumers and usually happen at the wrong time.

As we've seen, a change of perspective helps in most cases, especially when you put yourself in the other person's shoes. In the fourth part of this series, we'll look at the advantages of using standards.



APIs: Fundamentals on the Web - Part 4

How Standards Help With API Design

Uwe Neukam, Noah Neukam

14. April 2026

In this article, we want to focus on the topic of standards. Sometimes these can be seen as obstructive and annoying, but above all, they ensure independence. This doesn't just mean support in tooling; this type of agnosticism ensures interchangeability and central services, like formal validations and/or validities, or authentication and authorization.

People often ask why API enthusiasts keep recommending design by contract. After all, specifying the interface design with programming is much easier. The code includes everything that's needed anyway, including content checks for correctness. Well, that isn't easy to dismiss. The statement isn't necessarily wrong. Generally, code is the single point of truth and is therefore responsible for correctly processing functional data.

However, as in real life, this is only one side of the coin. Design isn't the only aspect that matters; there are other points to consider.

- **Infrastructure:** Before a resource can be accessed, security-related hurdles must be overcome in modern infrastructures, and rightly so. But these also increase a request's latency since the path to the actual processing service is significantly lengthened.
- **Zero trust:** Security once again, but from the perspective that every component in a chain is responsible for security. Authorizations at the API method level also play a role.
- **Vendor lock-in:** This is exactly what must be avoided in order to replace the tooling if necessary.
- **Validation:** "Am I doing the right thing?" How can I tell if this is the case?

These are just a few aspects that need to be examined. They affect almost all levels of technical communication via HTTP APIs.

This is where standards come into play

We can already see this is a complex environment that can't be understood in its entirety. To cope with this complexity, there are comprehensive standards designed to help us design APIs. We'll primarily consider two types of standards:

OpenAPI – Standard for synchronous API communication

The OpenAPI Specification is a standard for describing HTTP application programming interfaces (APIs). It can also be used to define REST-compliant interfaces. The specification is promoted by the OpenAPI Initiative. The initiative's vision is to provide an open and vendor-neutral description format for API services in the spirit of a connected world. The project is supported by the Linux Foundation." (Wikipedia)

OAuth2/OpenID Connect – Standards for authentication and authorization

OAuth 2.0, which stands for “Open Authorization”, is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user. It replaced OAuth 1.0 in 2012 and is now the de facto industry standard for online authorization. OAuth 2.0 provides consented access and restricts actions of what the client app can perform on resources on behalf of the user, without ever sharing the user's credentials. - (Auth0 by Okta).

OpenID Connect (OIDC) is an authentication layer based on the OAuth 2.0 authorization framework. The standard is a layer above the OAuth framework that allows clients to verify a user's identity using an authentication server and obtain basic profile information in an interoperable manner. The implementation of OpenID Connect is based on the HTTP programming interface with REST mechanisms and the JSON data format. The standard is monitored by the OpenID Foundation." - (Wikipedia)

There is actually a third standard for asynchronous communication called AsyncAPI. However, we will not discuss it here.

Someone may say, “That’s all well and good but standards can sometimes be annoying, cumbersome, and restrictive.” That can’t be denied, yes. Standards are usually the result of the process that aims to reconcile as many requirements as possible. But in return, they offer a whole host of advantages.

Independence from manufacturers, programming languages, etc.

This is at the top of the list. Recognized standards let you act agnostically from external influences.

For example, OAuth allows you to work with different identity providers (who's ever logged into another application with their Google account?). This ensures a basic level of security, which the standard has already defined.

HTTP APIs are already highly independent per se. But for development, it's important that OpenAPI sets certain syntactic (type safety) and semantic (correct date specifications) requirements that help enable language-independent communication. We're supported by many tools (just one example: API Design Editor from Swagger) to formulate, validate, and ultimately bring APIs into production.

Structured solution options

Standards also help with development work. They allow patterns to be recognized and corresponding solution spaces to be built up. These are then filled with life in the actual development. This also applies to API design.

We've learned that there is no such thing as the API. But it can be very useful to respond to consumer concerns and, under certain circumstances, offer appropriate communication options. At first, this can sound daunting, especially in terms of development and maintenance.

But there are already design patterns in development to deal with these scenarios. These can also be applied in API design. This makes it possible to specify different APIs, but send them to the same backend with different data structures. This can be taken so far that it's possible to formulate the access behavior for the consumer at runtime (HATEOAS).

Durability and interchangeability

These two issues are especially important for companies because time and time again, developments are viewed as long-term investments. Adam Bien once said, “Use standards that have been around for a long time”. And in a way, he's right: there are always one or two nice implementations or solution ideas, but these approaches come and go. Standards survive, and that's a good thing. Even if the infrastructure I'm using no longer exists, I can switch to another provider without needing to change the actual design. It stays transparent for consumers.

The situation is similar for producers. Although behaviors (e.g., deployments) and sometimes sequences change, the API itself stays the same and can be processed by the new provider—provided the provider supports the standard.

Focus can be placed on technical expertise

This is a brief recap, the question is: “How can I ensure that exchanging information doesn’t backfire on me as a provider?” It’s true that request content should be validated on the backend. But missing type specifications (such as length specifications) in the specification can place an excessive strain on resources in the network. This can go so far that systems fail. This isn’t only because technical details like the type, scope, and/or information format haven’t been described, but merely assigned character strings.

The exciting thing is that tools take these definitions into account. During conversation, generators also take them into account(e.g., into source code). It doesn’t matter whether this involves client or server generation.

API Management

To establish communication, we need infrastructure in the form of API management. What does API management offer? In short, it offers three components:

- **API Manager:** This is the development and management environment for APIs. The environment is designed for developers and supports them in API design, provision, and lifecycle management. In most cases, the environments can support multiple types of APIs (http, GraphQL, etc.). Actions can be defined in this environment to control the throughput of a request. Key words here are validations (is the API content correct), rate limiting (are the available capacities being overloaded?), and/or security (OAuth2: authenticity and identification).
- **API Gateway:** This artifact is the gateway to a provider's processing environment and offers corresponding security measures. These vary in nature and are generally specified as policies (e.g., authentication). Therefore, there will be default policies from context to context that represent basic security. However, these can be extended to include policies that have been specified for specific situations (content validation, rate limits, etc.). Individual validations will be performed via the API specification.
- **Developer Portal:** This is the offer portal for consumers. It displays the entire range of services offered by a provider for machine communication. Here you will find the API's scope, type, documentation, and usability (public or subscription-based). Many developer portals also offer corresponding playgrounds where you can get to know APIs and test them.

This is only a small excerpt of services API Managements offer. They also offer monitoring services, which we won't discuss further here.

What are the benefits of these infrastructure environments? Above all, they save time, enable fast responses (fail fast), and provide security (see Policies). The configurations for each API are stored in the management system and are analyzed and validated according to the set criteria at a very early stage—immediately upon receipt of the request. This means requests can be blocked there because specified

standards haven't been met or because the validation was completed with a negative result.

Standards help us throughout the entire life cycle

Even though standards can sometimes seem annoying, they offer significantly more advantages than disadvantages. We can rely on extensive support, regardless of where we are in the chain. For API providers, the range of possibilities has increased significantly. Standards make it possible for tasks to be performed reliably at other locations in order to conserve their own runtime resources. As a consumer, I can trust the API has been described in detail in terms of scope and behavior and will reliably and correctly deliver the information I need.

Conclusion

Interface design isn't really a new topic. IT simply doesn't work without interfaces. But the introduction of APIs changed a lot. Ultimately, the reach of offering via the Internet has increased dramatically. Companies are also discovering this type of universal communication as a business model and want to earn money by offering a functional API. This requires more attention than interfaces may have been given in the past.

In this article series, we showed that APIs should actually be a separate component of application development. It makes sense to specify them independently of technology, existing software, and data storage. Developers should focus more on potential customers' technical needs and be aware of their use cases. This is a major challenge and applies to both the type of APIs discussed here and all other API types.

We'd like to conclude with a quote from Uwe Friedrichsen (codecentric) that sums up this necessity very well. "Creating a good API requires a lot of effort. "