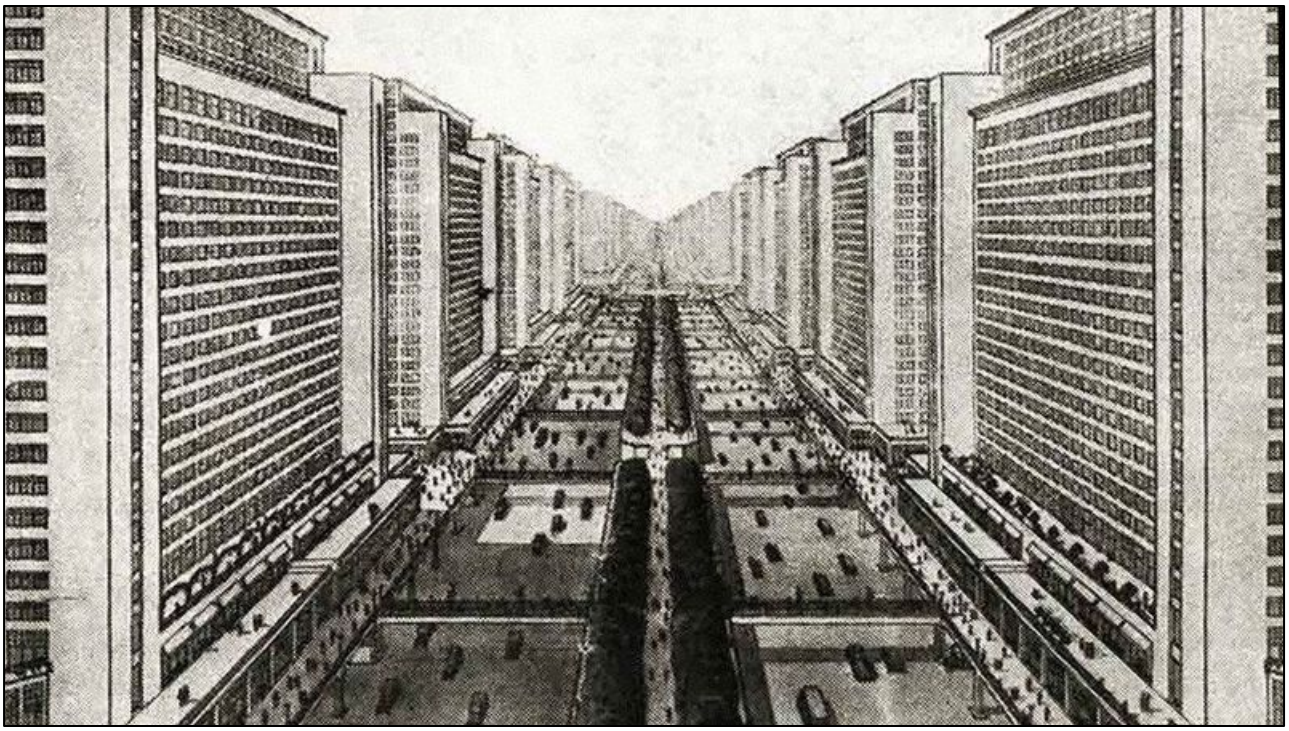
This talk is based on more than a decade of experience on Chrome's Web Platform team, as well as platform experience for Augmented Reality and Android.

This talk is aimed at people who are tasked with growing or looking after a platform--deciding what to add, modify, or remove--and it will take on that perspective. Here, 1P means "the platform owner", and 3P means "other entity (largely other developers) who build things on top of the platform."

If you have comments, thoughts, or just wanted to say you liked (or hated) something, feel free to reach out at alex@komoroske.com!

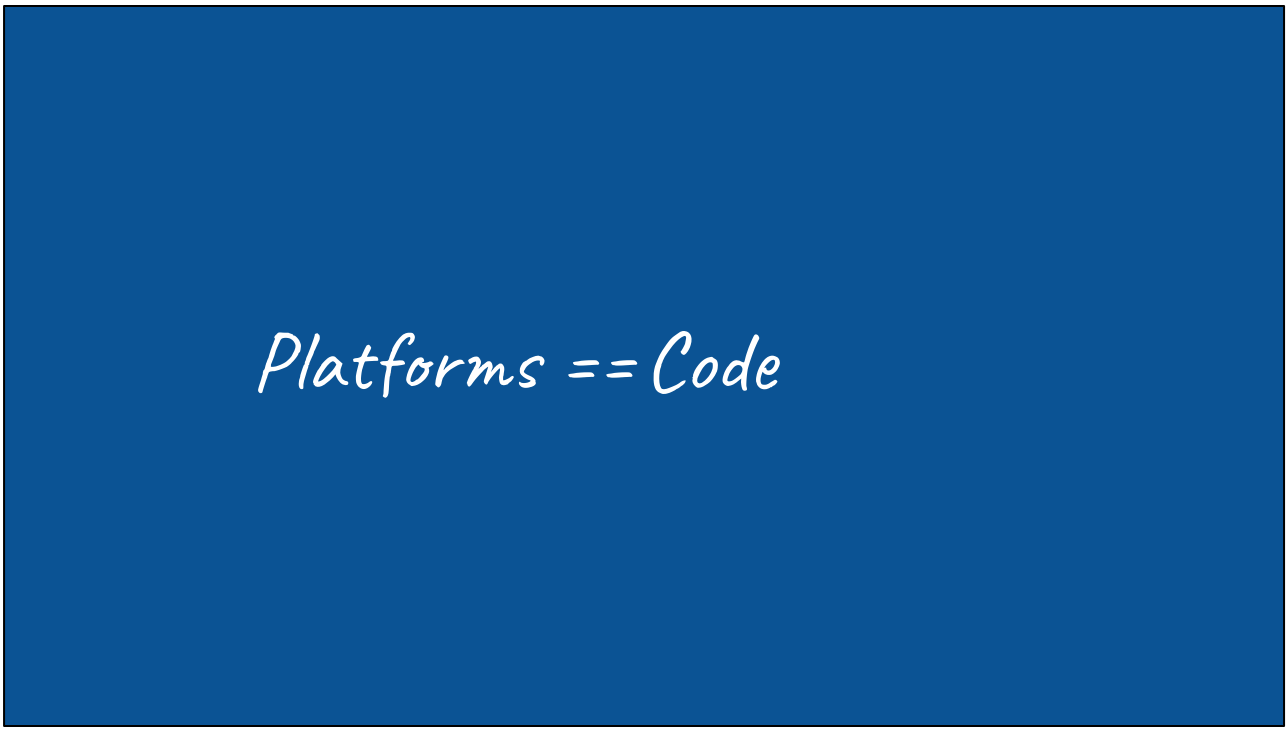


A lot of people try to treat platforms like sleek, modern skyscrapers--and then get frustrated when nothing goes to plan.

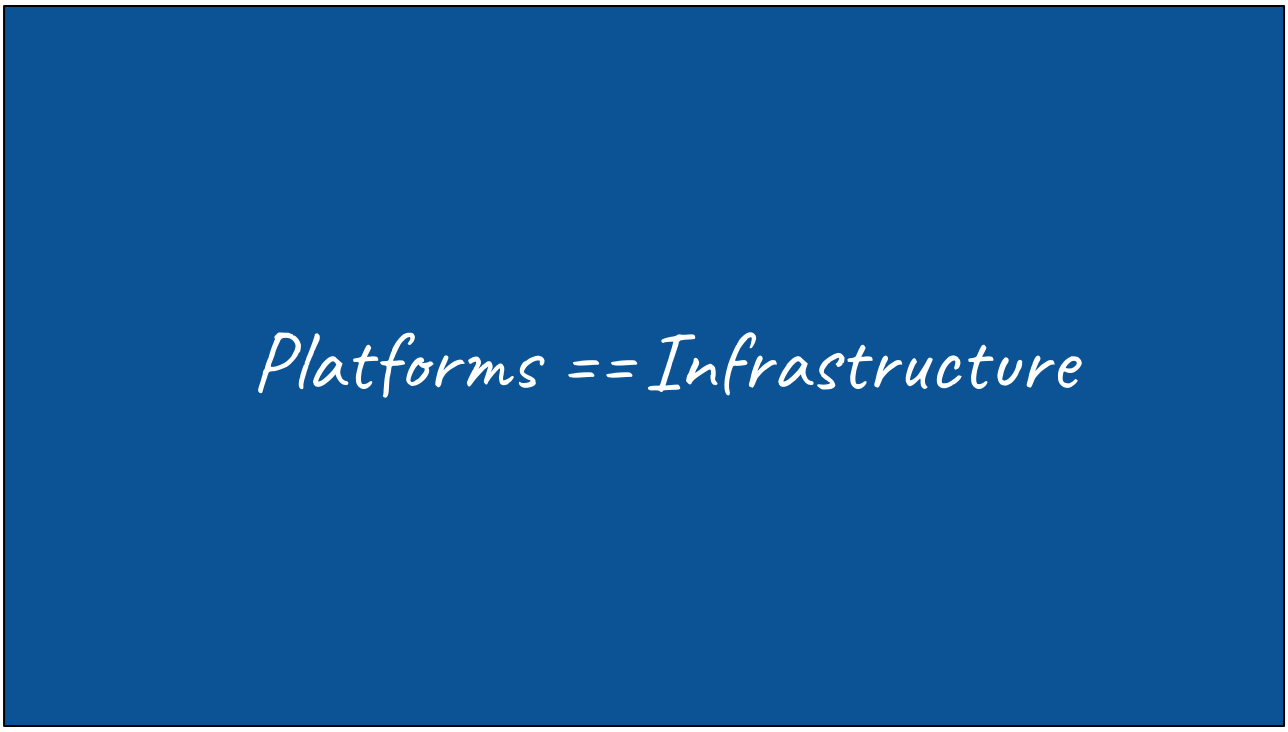
Image: https://www.smithsonianmag.com/smart-news/how-controversial-european-architect-shaped-new-york-180965073/

By seeing them as more like gardens, everything becomes easier.

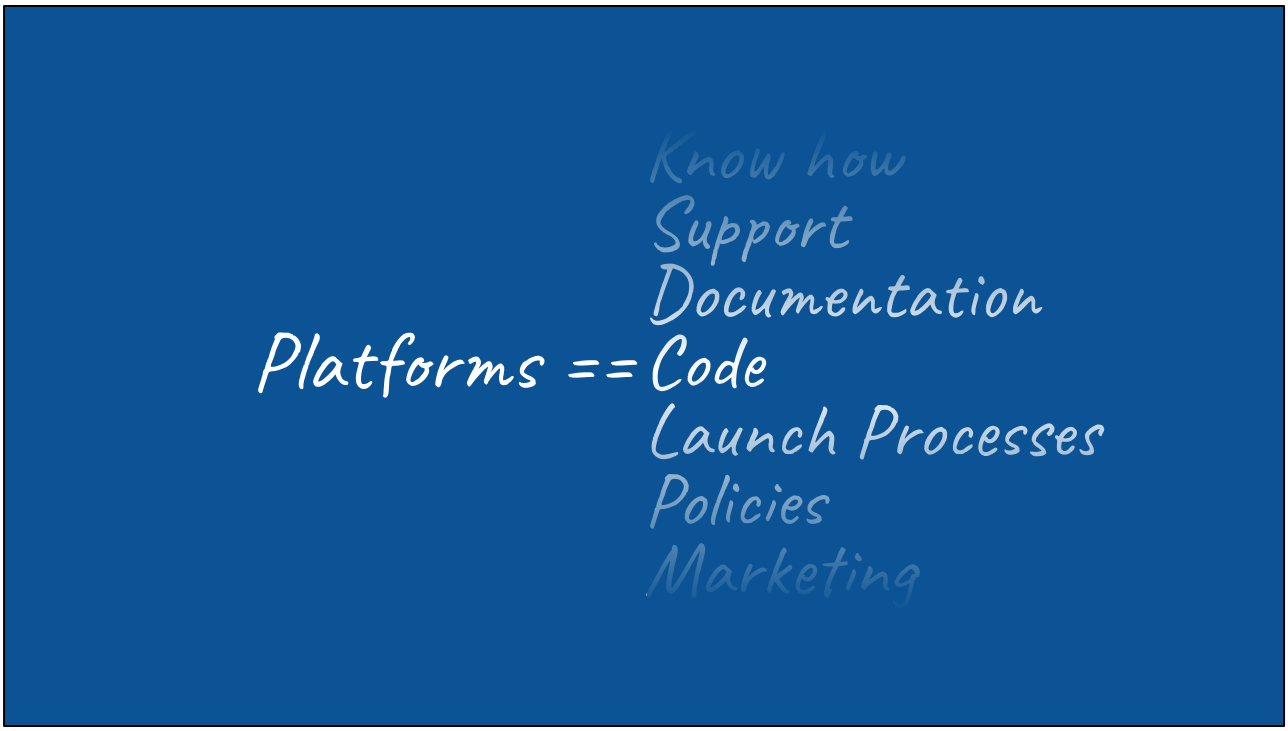
What do I mean by platforms?



People often think of platforms as just the code.



But platforms really mean any kind of shared infrastructure. Infrastructure means things that are pre-existing that any one user can take for granted and build upon.



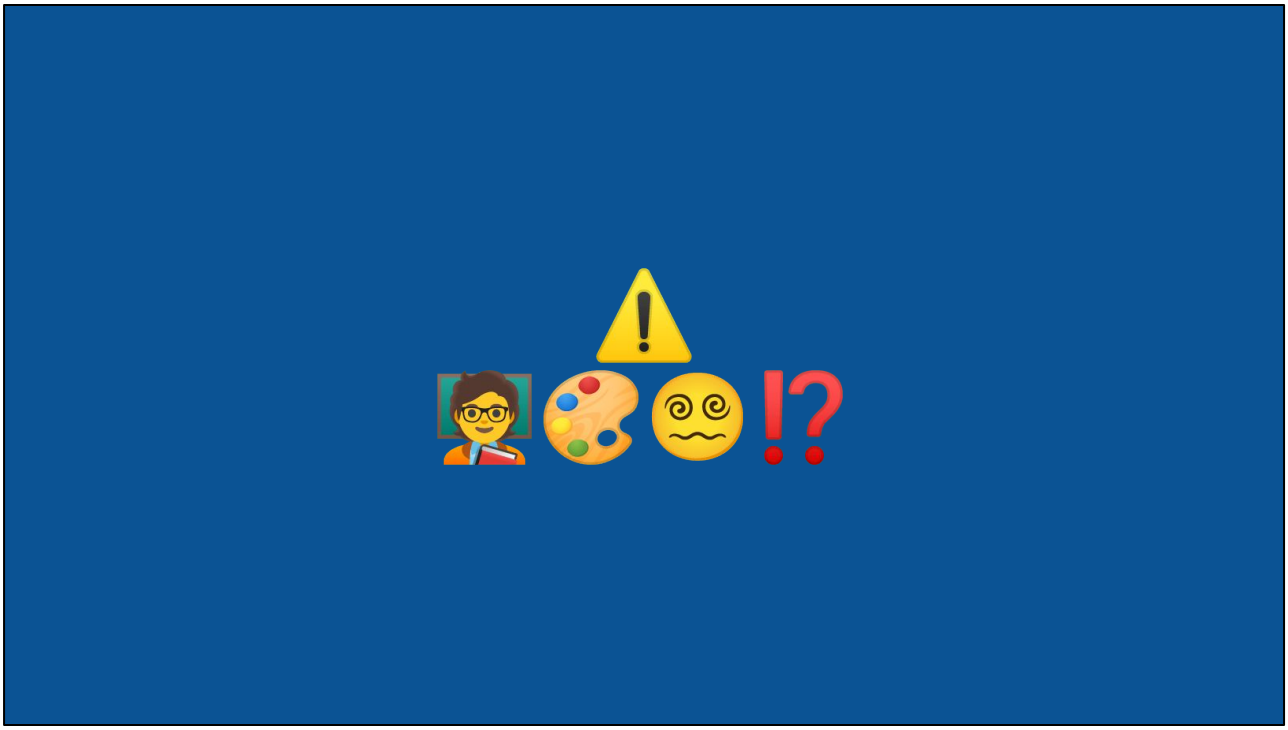
Infrastructure includes all kinds of related things. Some of it is code, but it's also all of the processes, documentation, know-how, marketing, etc. Infrastructure is anything that a given project can take for granted because it already exists. Infrastructure is by definition relied on by multiple things.

This means that platform doesn't necessarily mean "3P developer platform". It can also mean internal infrastructure, or even things without any code.

In fact, using platforms as a lens can be useful even applied to many different problems that people don't typically consider to be platforms.

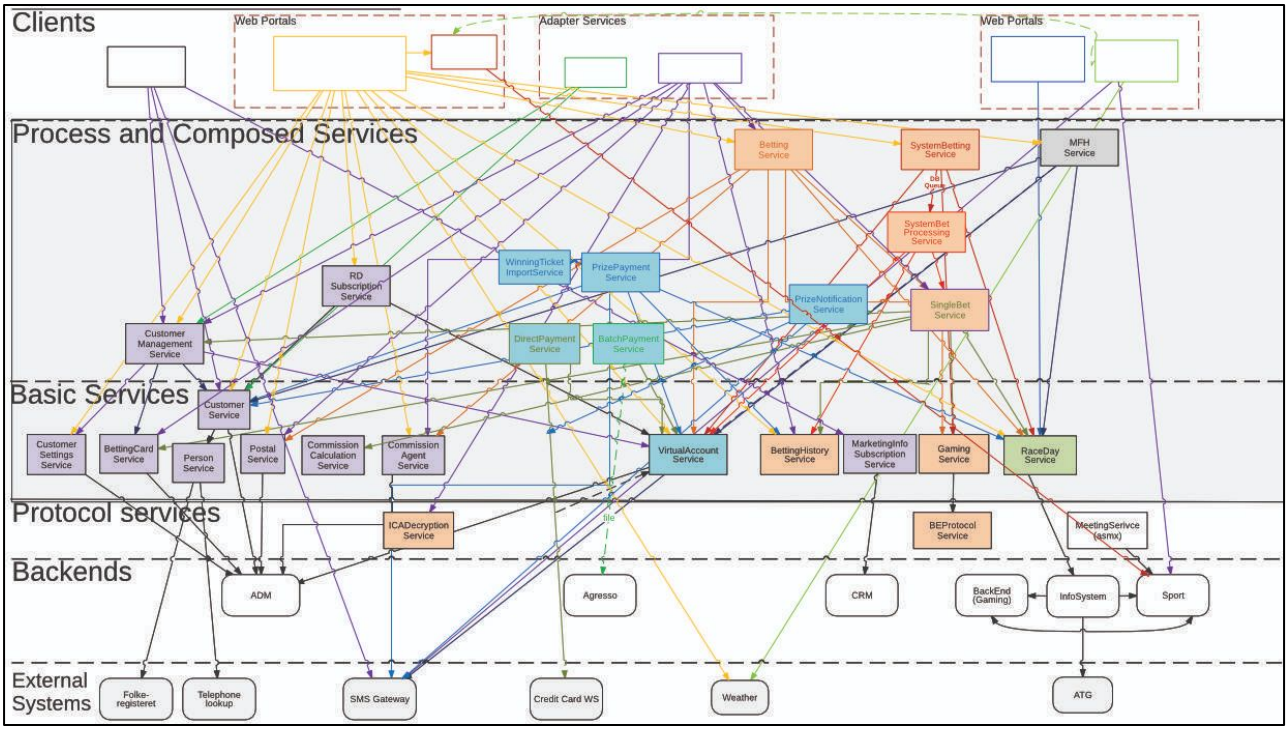
For example, you can apply this lens to: open, standards-based platforms, proprietary platforms, aggregators, internal technical infrastructure, internal process infrastructure, your professional network, you own personal knowledge garden, your team's competencies, etc. Another example is if you're trying to build a two-sided market with supply and demand.

So if you think this deck isn't for you because you don't work on developer platforms, give it a shot!

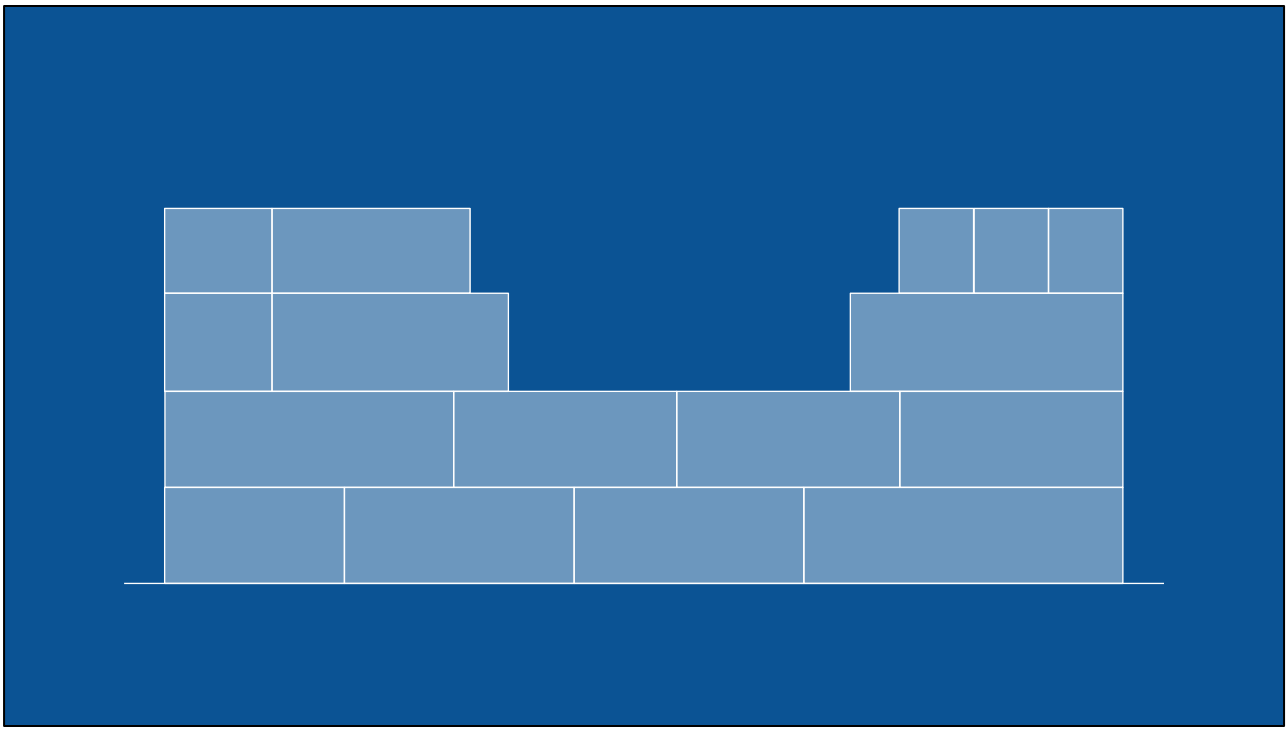


Warning: this deck is going to be very, very abstract!

Complex topics are really hard to understand via examples. One trick is to build up a visual intuition that allows you to grok the deeper concepts. This deck will use a “flipbook” style to help develop that intuition… but that means it will seem overly abstract and maybe a bit flippant.



We could dive into a very specific example in depth and be very concrete. But then we'd miss the big picture.



So we'll focus on idealized abstractions to see the big picture dynamics better.

Rest assured, all of this is based on more than a decade of experience designing and maintaining various real-world platforms.

Abstraction works best when connected with your own practical experience. Instead of free-floating concepts, you can recognize concrete examples from your own experience that fit it.

If this feels over your head or doesn't click, maybe you don't have the practical experience for it yet. That's ok, maybe it will click later! Or maybe you can look at your experience with a fresh set of eyes, consciously trying to apply a platform lens to your past experiences, to unlock the meaning.

Now that we've got that out of the way, here's the agenda.

We'll first dive into fundamental dynamics of platonic ideal platforms--how and why they work.

We'll then turn to how to handle a real-world platform, evolving it closer and closer to the ideal.

And finally we'll talk about how to grow platforms from scratch, including a general playbook for platform problems.

1. Ideal Platforms
2. Evolving Platforms
3. Growing Platforms

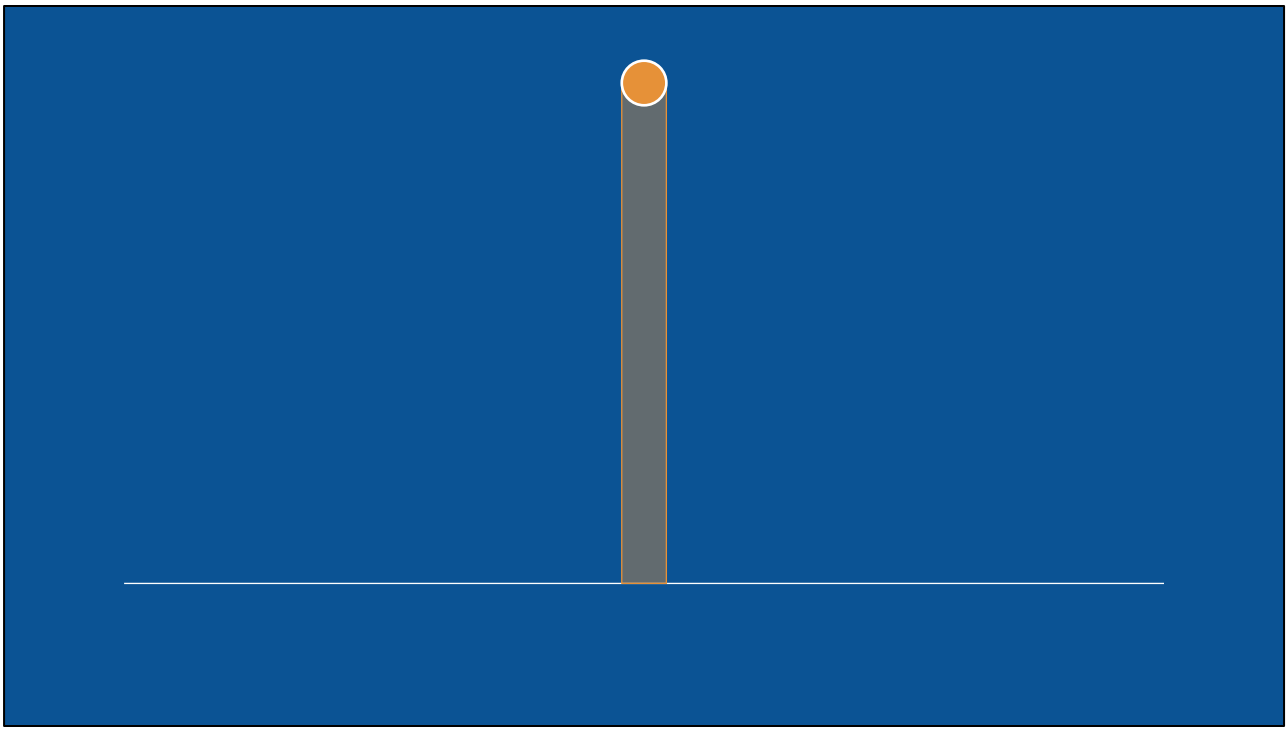
Let’s dive into the properties of the platonic ideal of platforms.

Let’s start at the beginning. Imagine you have your bedrock.

And imagine there's some end-user value out in the world that you could be harnessed--some opportunity that if seized would create value--not just for the end-user, but for the entity who created that value for the user.

This is where the rubber hits the road. Maybe it's "allow an end user to buy a book", or "allow an end user to read local news," or "allow an end user to chat with a loved one."

The point is this is, as far as we're concerned, the **end**. The entire goal. But to achieve that end, you need some *means*.

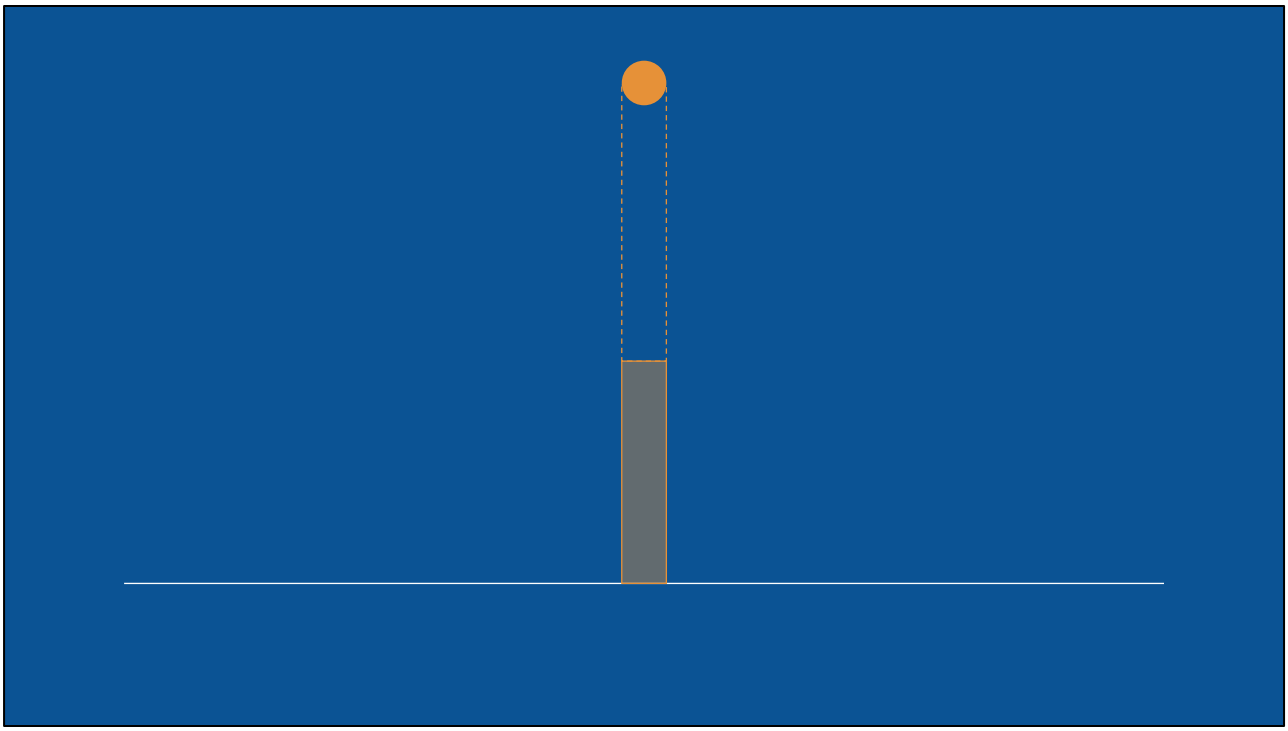


Those means take the form of stuff that has to be built.

Code. Apps. Processes. Support.

Someone does this work. They build this stuff that allows the value to be unlocked. It’s hard, back-breaking work.

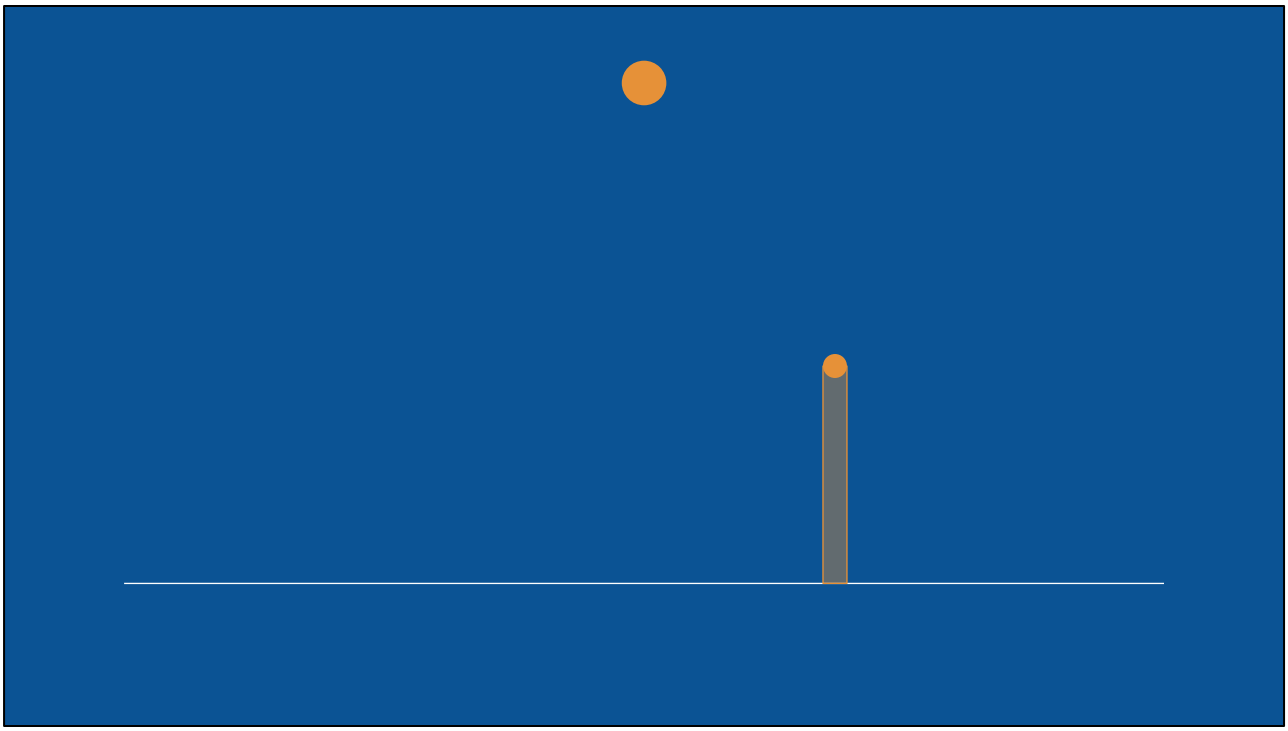
When you build the right things, you can unlock the value. The end-user is happy, the entity who helped the end-user gets money, or loyalty, or engagement. Everyone wins!



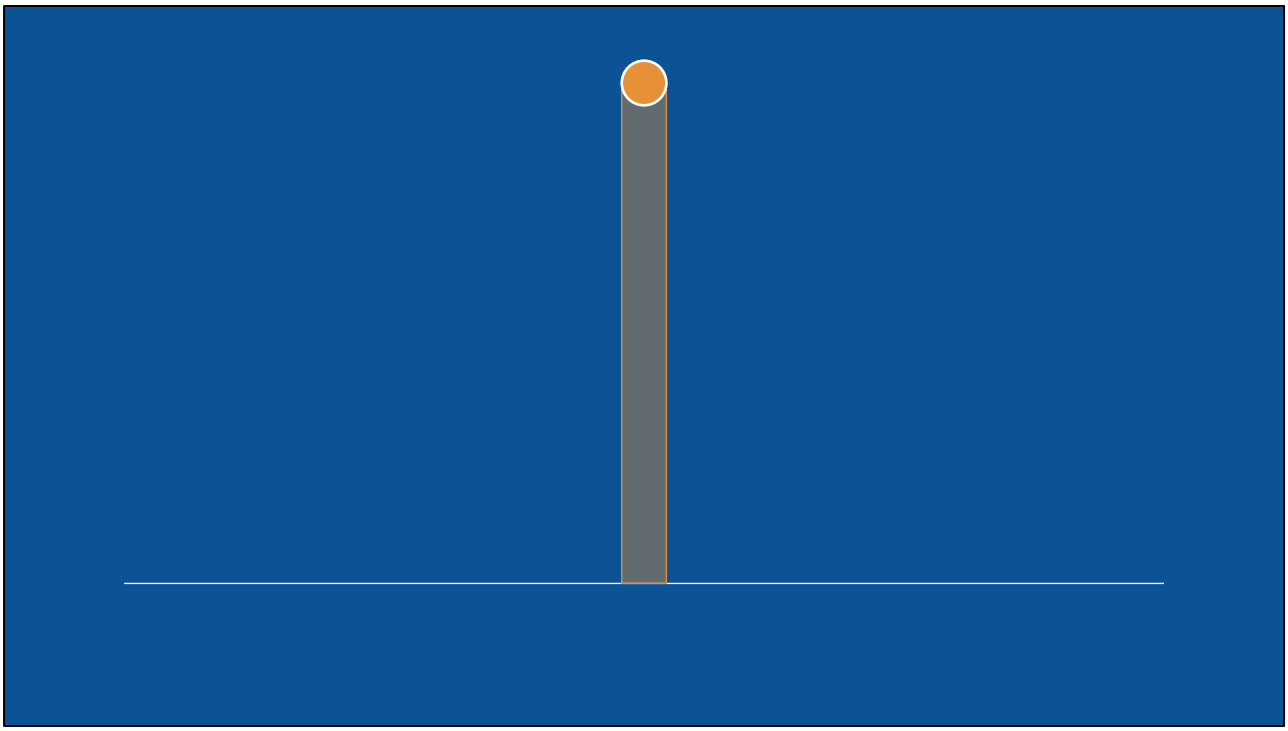
… Or maybe not. Maybe there’s only enough value to justify building part of the way.

… in which case, no one bothers building it. The value sits there, possible, but not actually activated.

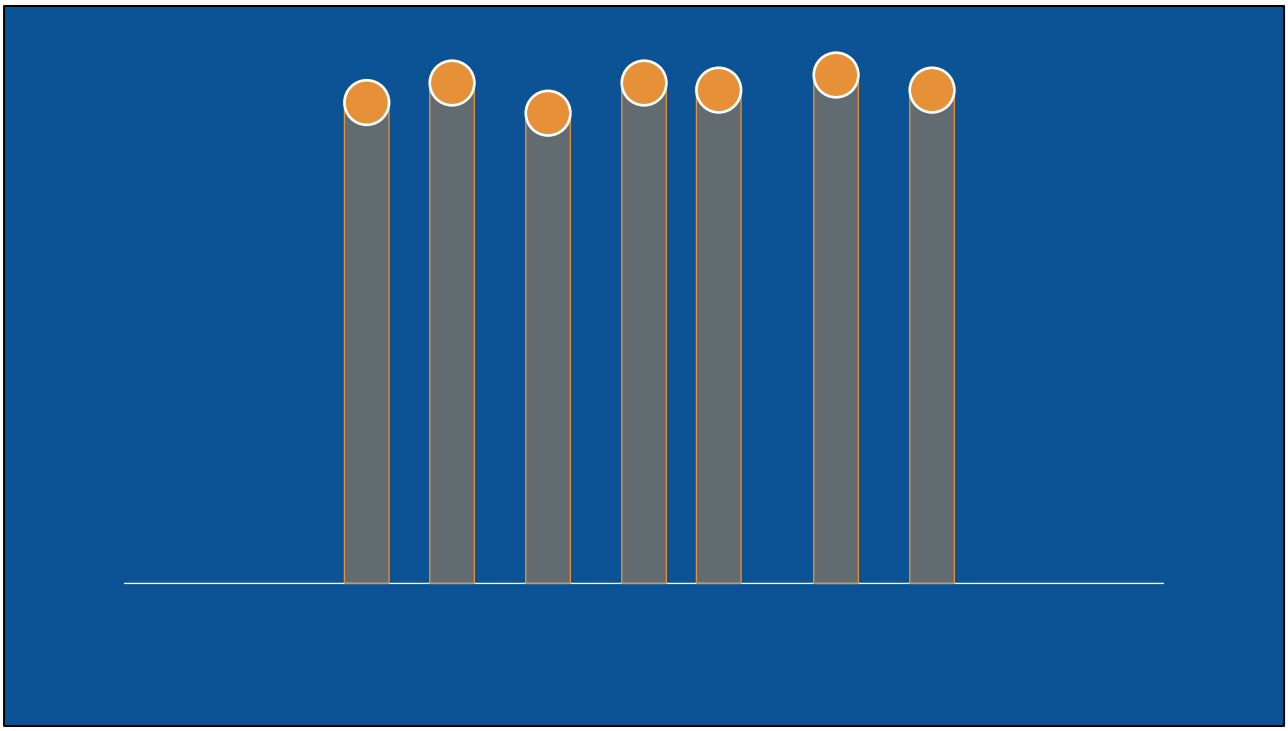
Everyone is worse off.



So maybe they go for a smaller target. A crappier one that’s easier to reach.



But even if it is worth building, remember, this is for one developer, one opportunity to capture.

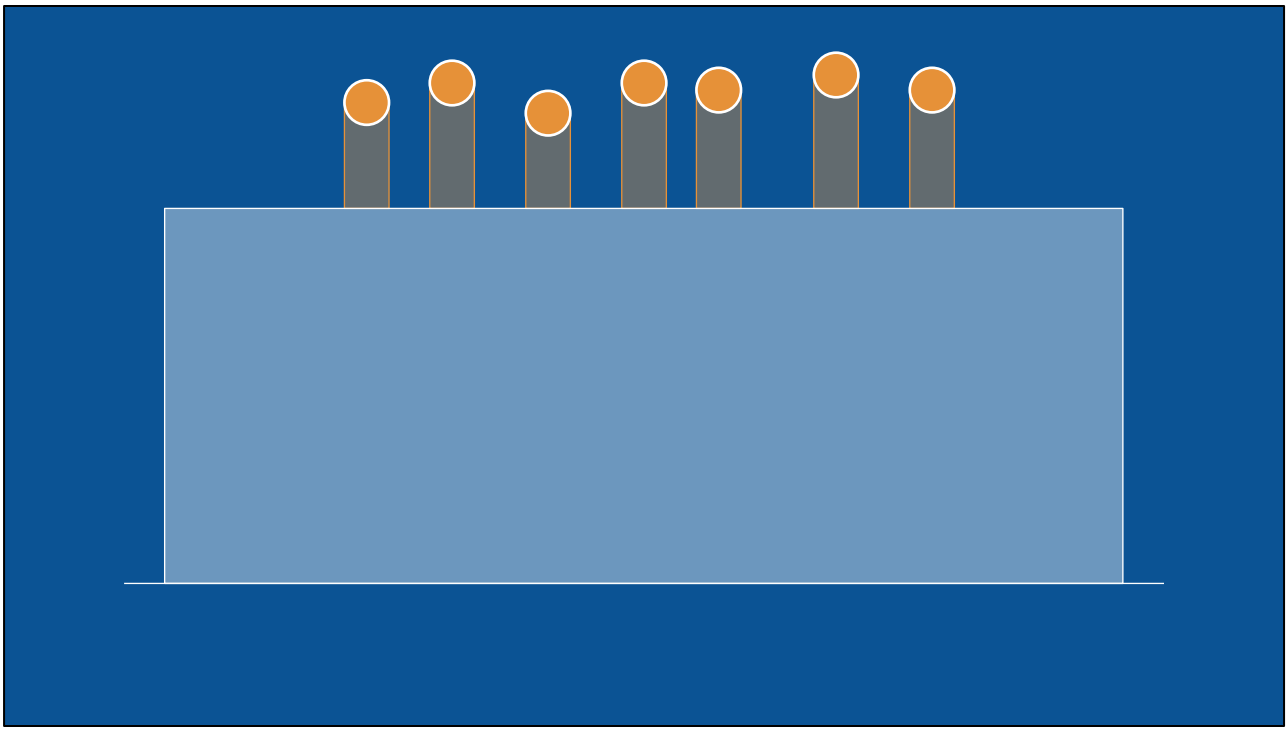


In reality lots of individual entities are having to build tons and tons of bespoke stuff.

We can do better.

Each 3P is doing tons and tons of bespoke work to reach high. What if we helped lift them up, so they could reach higher?

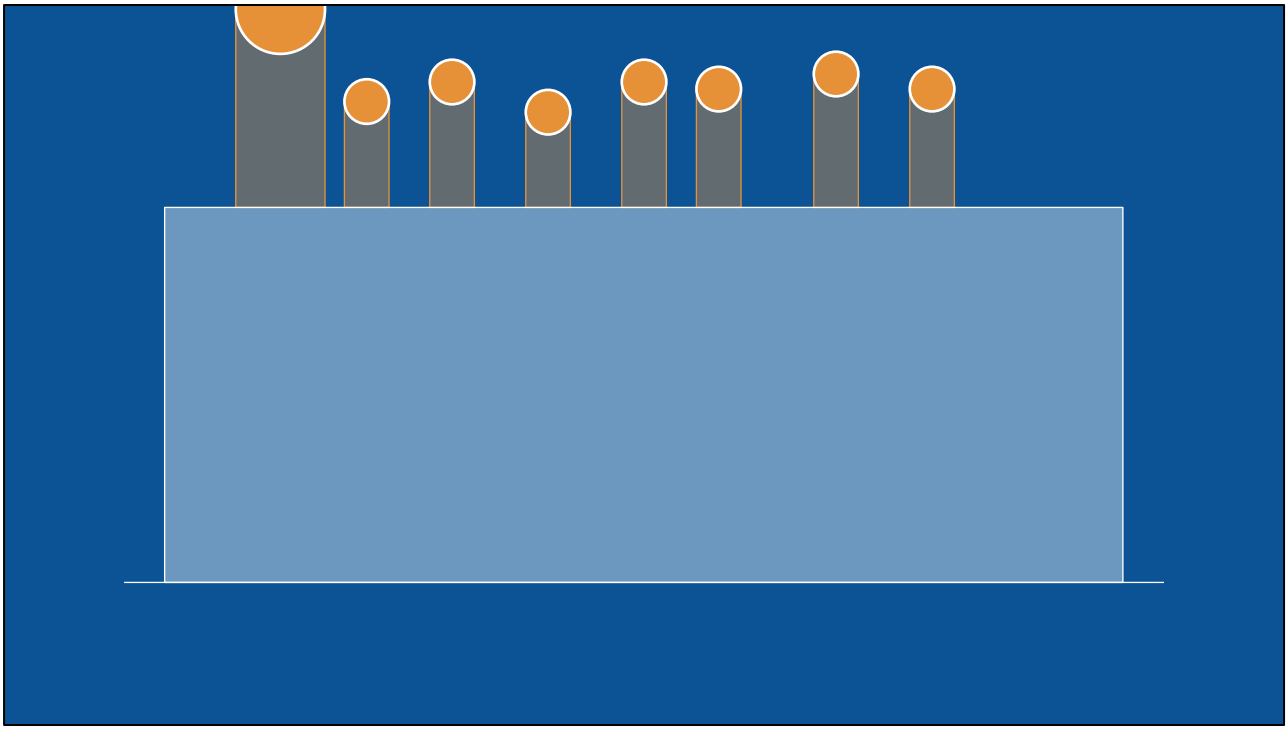
What if we built… some kind of platform?



We could make it so other 3Ps have a higher starting point to build off of. That allows them to reach value much more cheaply.

(Yes, I know that this looks like a birthday cake. As we add complications it will look less like a birthday cake and more obviously like a platform.)

We've effectively paid the cost once and shared it across many things. This is a very valuable service. The 3Ps might even be willing to pay the platform provider.



By the way, the fact the 3Ps can start up higher means they can now reach further and reach higher value.

The magic of platforms!

Before we go further, it's useful to differentiate two different flavors of platforms.

Sometimes these platforms take the form of **tools** and sometimes they take the form of **services**.

A tool is something that the 3P can use however they see fit. Even if the 1P went greedy, incompetent, or evil, the 3P could just go on using the tool on their own computer. The ultimate tool is open source--no one can take it away from you! Worst case, the 3P can fork.

A service is a platform that requires a continuous, ongoing interaction between the 1P and 3P. If the API involves calling an endpoint on the 1P's servers, that's a service. The 1P has way more flexibility--they can have data that the 3Ps can't see, for example--but if the 1P goes greedy, incompetent, or evil, the 3P is out of luck.

What Ben Thompson calls an aggregator is a special on-steroids class of service where the service that is being offered is distribution to the service's own end-users. That distinction is extremely relevant in some situations, but for how to look after and build the platform it doesn't matter too much, so we'll just lump them in as an type of service that's on steroids.

This, like many two-sided things, is not a binary but a smooth continuum.

Why would someone build a platform?

OK, but why would someone build a platform for 3Ps?

An internal platform is easy--you want the whole company, many different teams, to be able to pool effort.

But for 3P platforms it’s a bit more complex. There’s a few reasons.

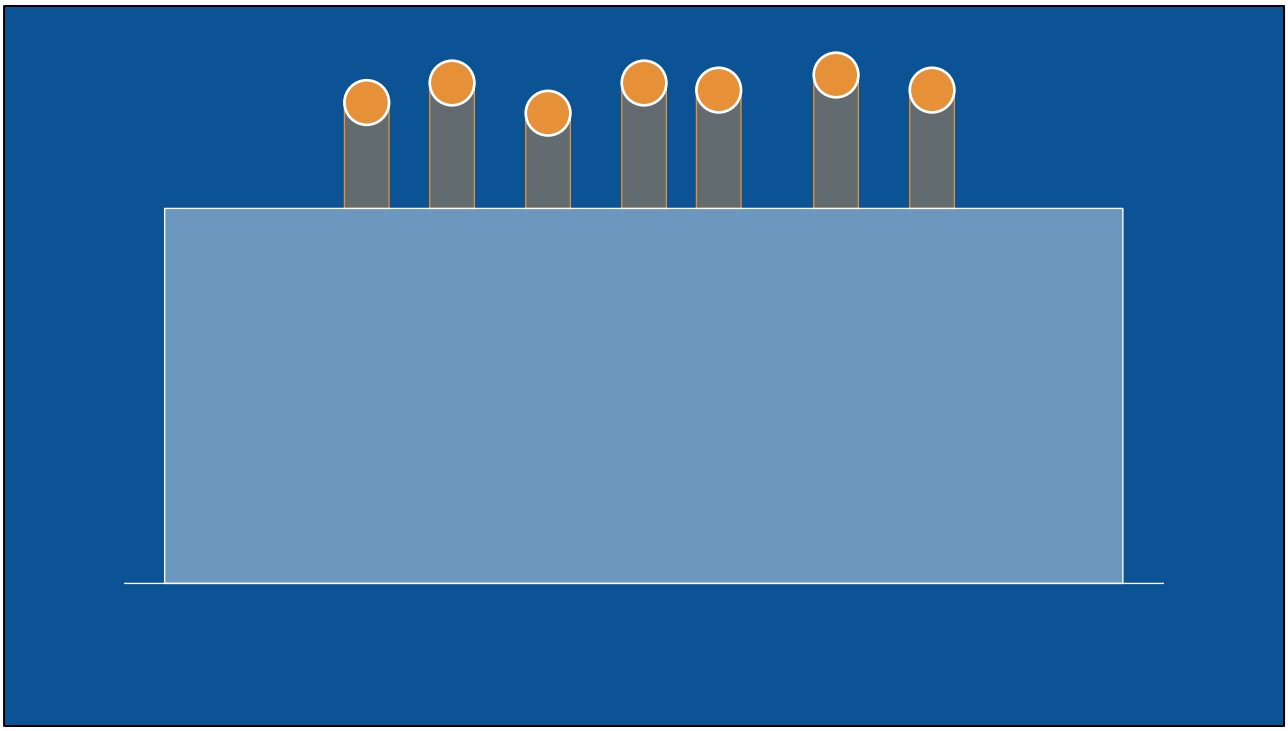
Sometimes it's just out of the kindness of your hearts. Open source. People just wanting to share useful things with others.

Sometimes the 1P has some *complement* to the platform. Every time a 3P realizes value in the ecosystem, the 1P gets a cut of it. So the 1P is incentivized to have more activity in the ecosystems. This is where that classic "commoditize your complement" business strategy comes in.

Sometimes the 1P gets revenue directly as 3Ps use the platform.

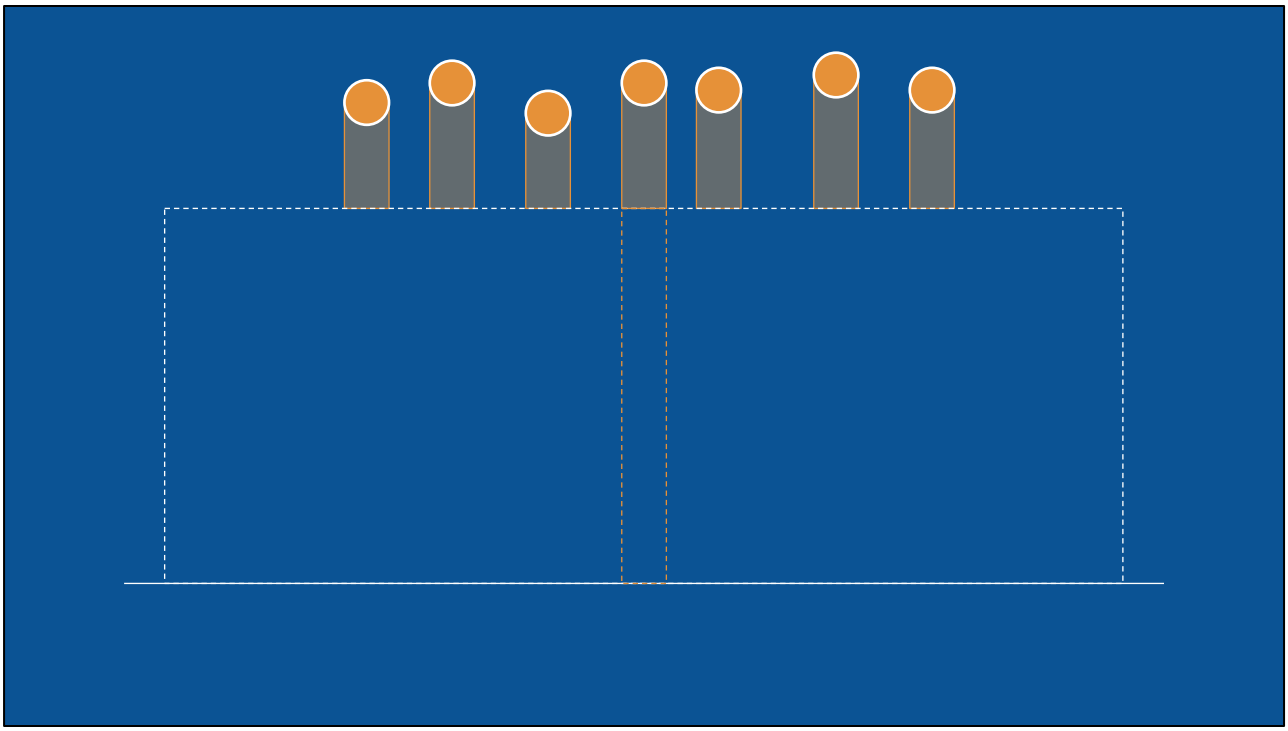
It's possible to have a mix of things. For example, Stripe has a service a the very bottom layer, and a lot of layers on top are free services that are complements to the underlying layer.

By the way, the motivations of 1Ps tend to differ from tool to service.



OK let’s go back to our proto-platform.

Let’s imagine that the 1P is providing some kind of service. They aren’t charging that much for it, so it’s worth it for the 3Ps.



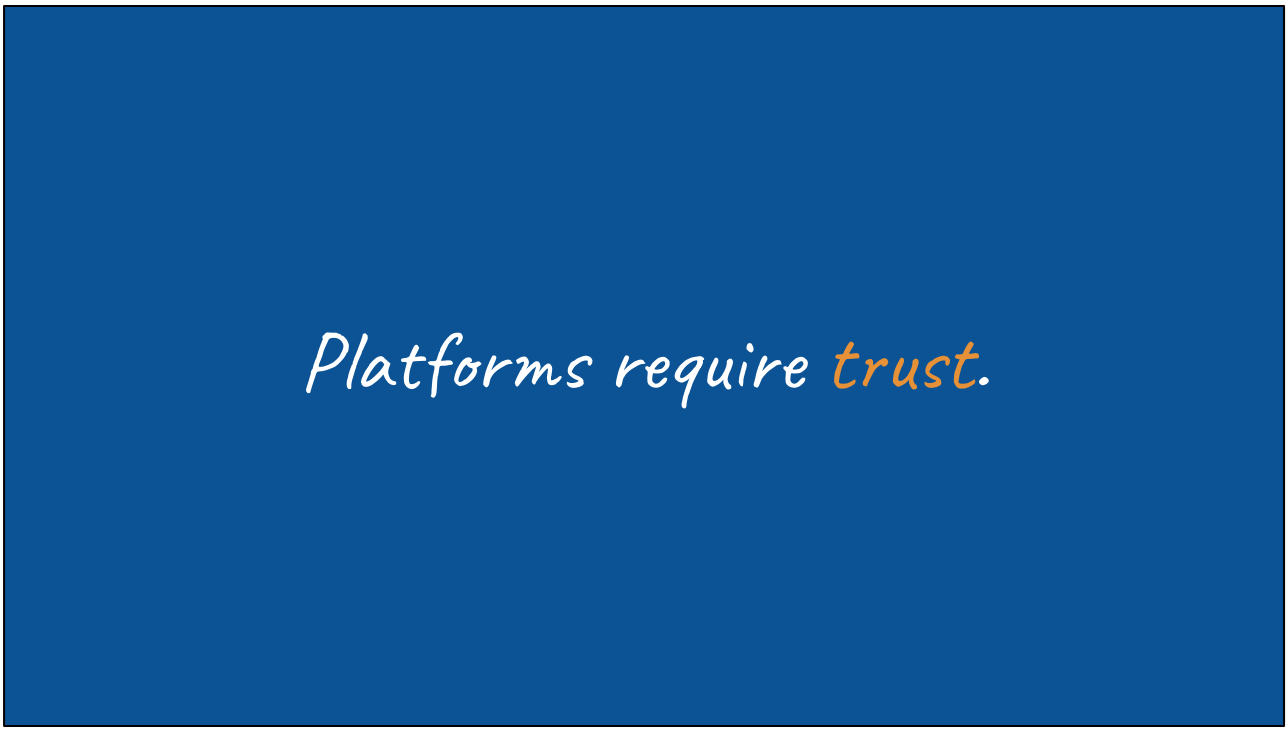
But let's imagine the 1P gets greedy. They double the price they're charging.

Suddenly, it doesn't make business sense for the 3P to use the platform anymore.

… But now the 3P has to build a bunch more code on their own. That's a lot of stuff! They can't build it fast enough, or cheaply enough. Their business is now no longer viable. They're toast!

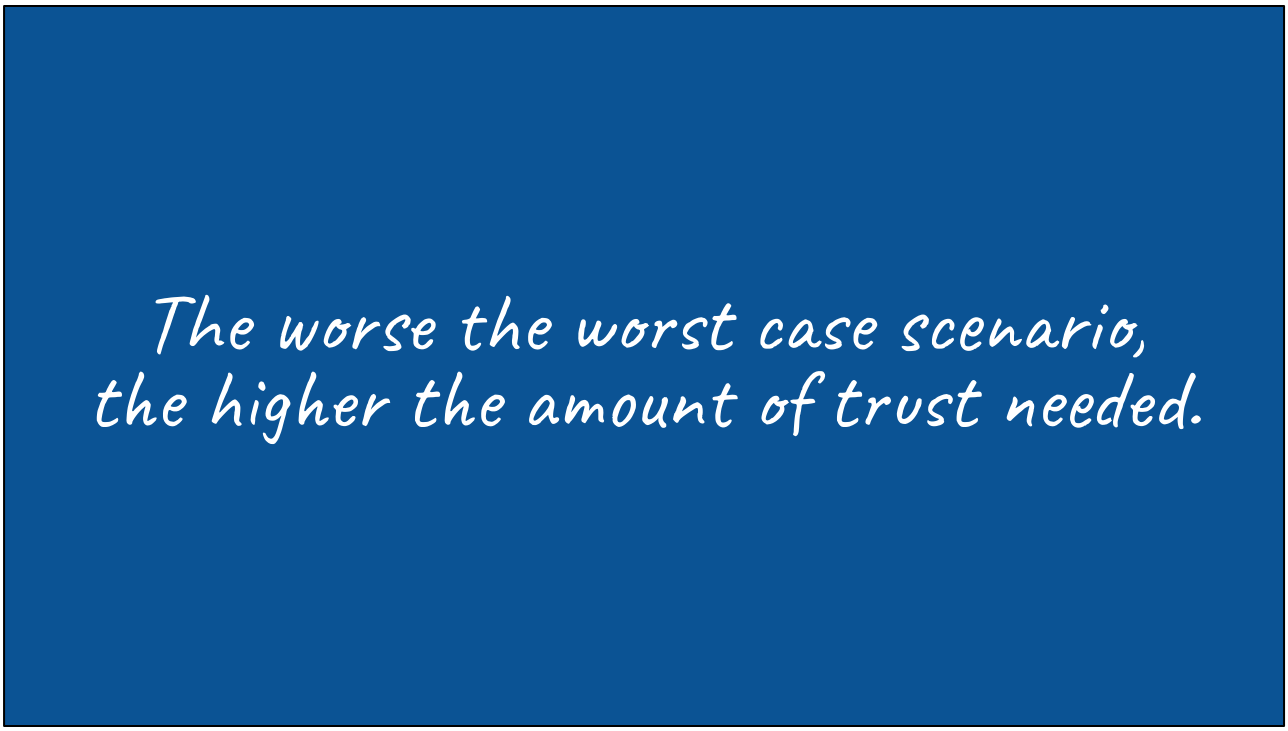
This doesn't just have to come from the platform getting more greedy, by the way. It could come from the platform having a breaking change, or a deprecation, or the 1P going out of business.

The main point is: the 1P can potentially screw over the 3Ps. They have **power** over them.



And where an entity has power over another, it requires authentic trust between the two.

As a platform provider, you have to understand how scary you can be to the 3Ps and end users who rely on you, how much power you wield. That means that if you do something that could be *interpreted* as *maybe* implying you'll change something that will hurt a given 3P, they'll freak out, and lash out violently, as anyone does when threatened by a more powerful entity.



Of course, not every platform requires the same amount of trust.

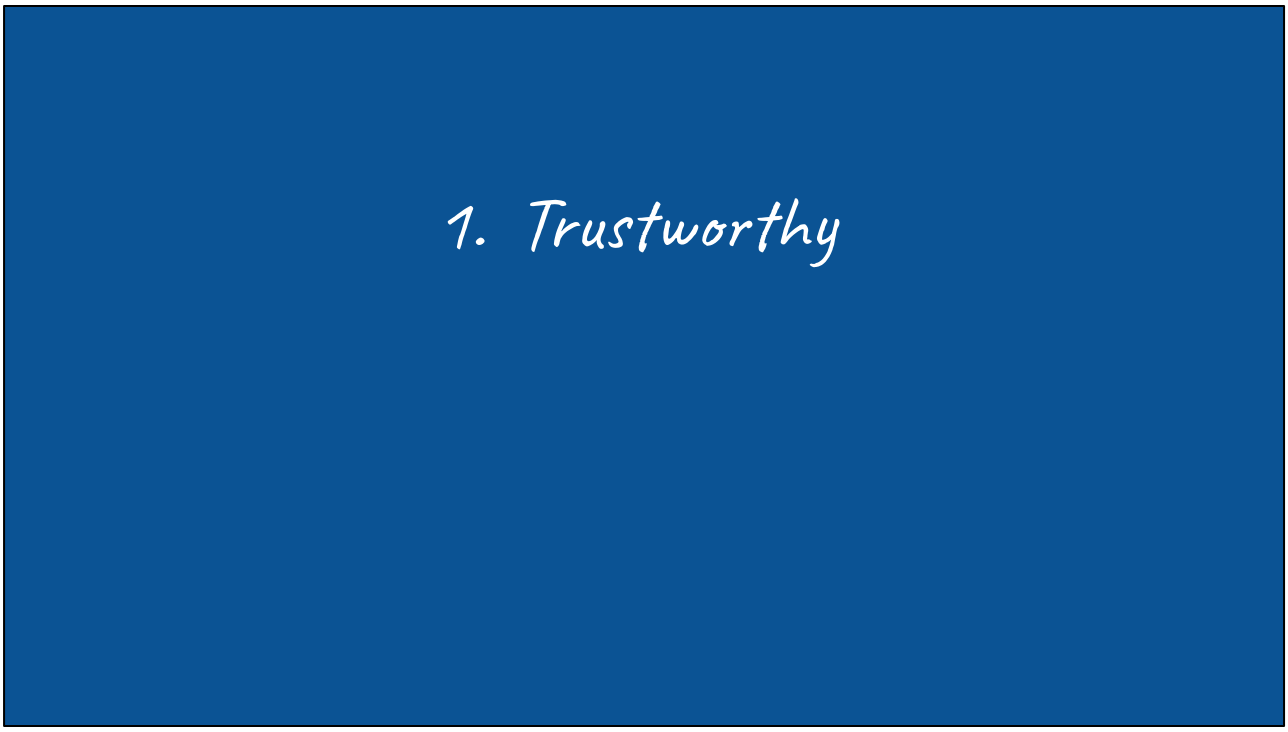
In general, the worse the worst case scenario for a 3P if the 1P went greedy, incompetent, lazy, evil, the more trust is required.

A thriving, fully open source tool that could be forked if necessary? Very low risk, very low trust required.

A difficult-to-recreate platform with access to proprietary data with no viable competitors and high switch cost? The maximum trust required.

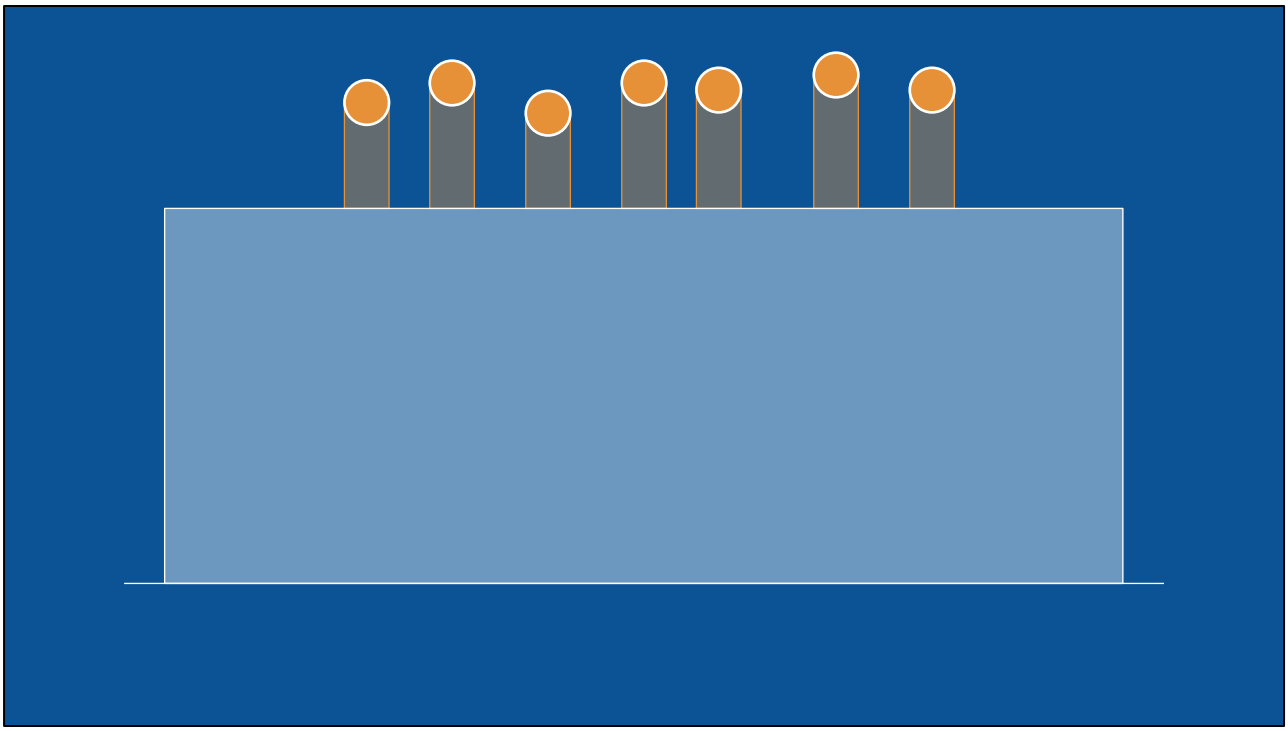
Track record matters. A 1P who has a spotless track record will be easier to trust than one that has been greedy in the past.

Extra Credit: The Meaning of Open digs into these issues of power in platforms.

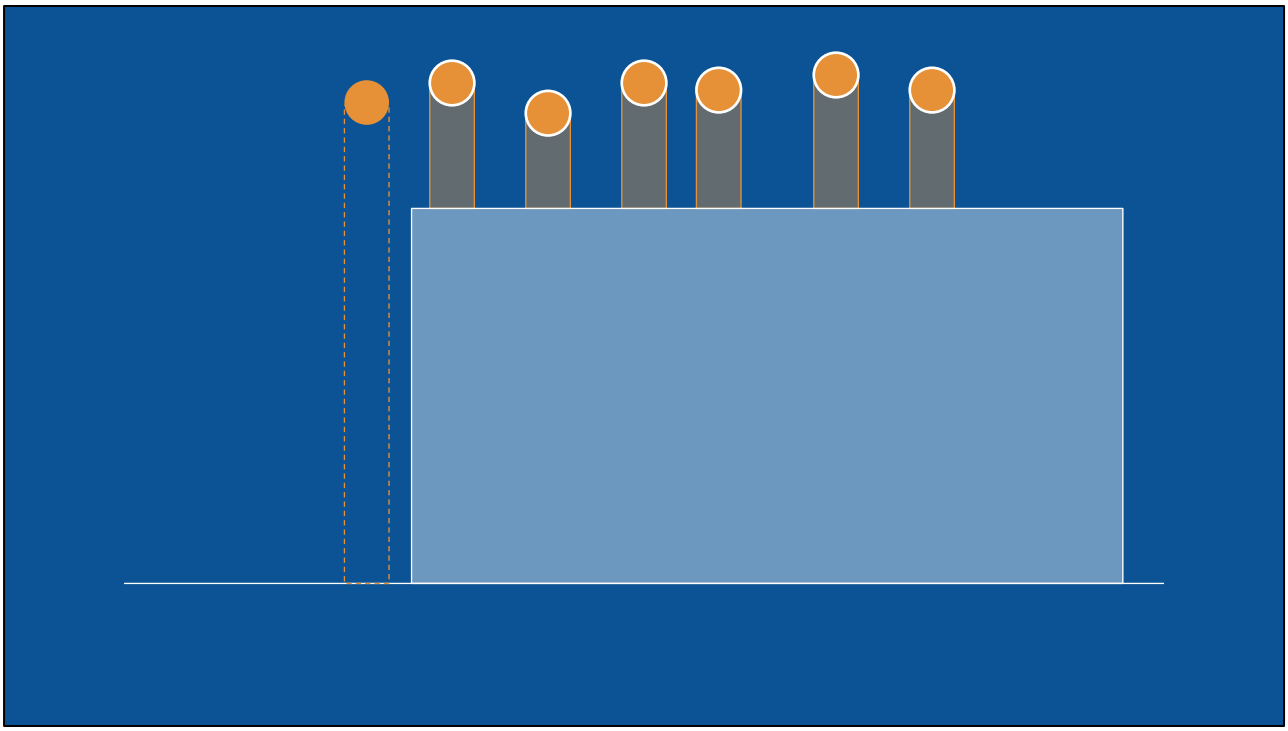


That’s the first property of rational platforms. They have to be trustworthy to be used.

Let’s talk about the next one.



Here’s another important thing: it’s only possible to build things on *top* of a horizontal surface in the platform.

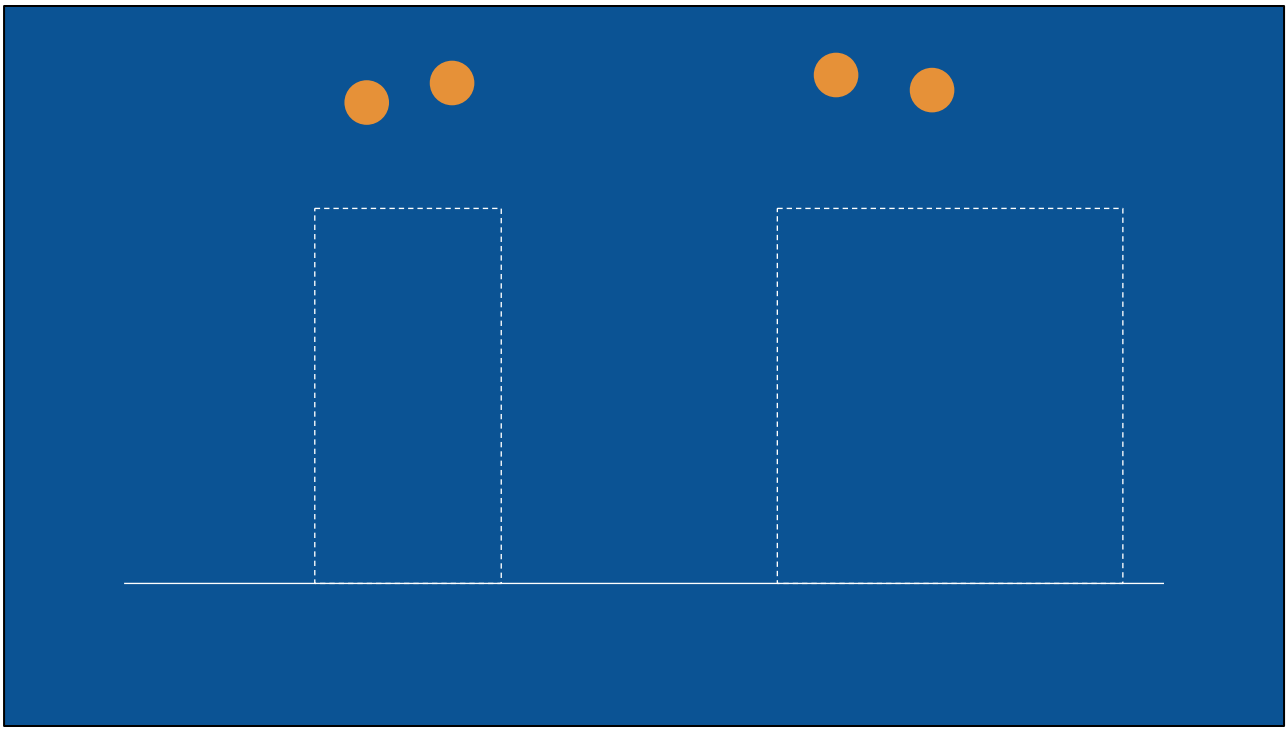


If the platform doesn’t reach underneath the use case, the 3P can’t use it.

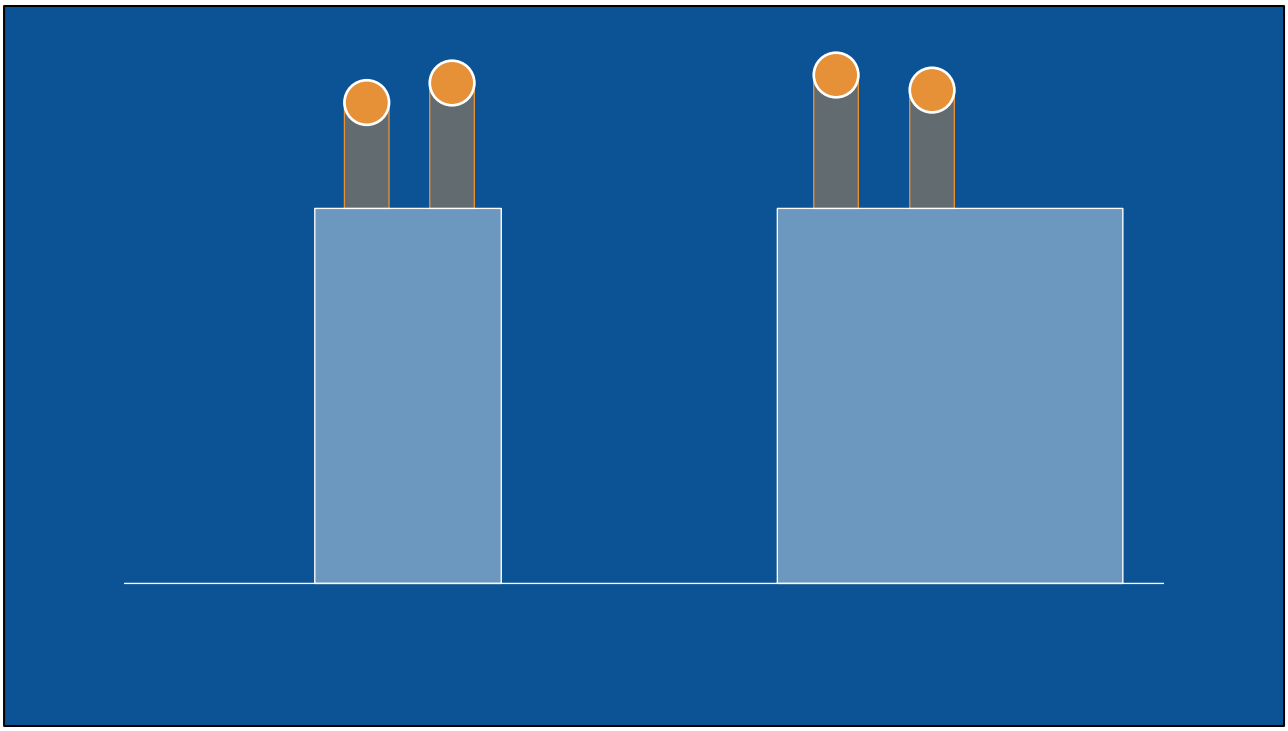
Platforms only enable things on top of their exposed horizontal area.

That means platforms tend to be **wide**.

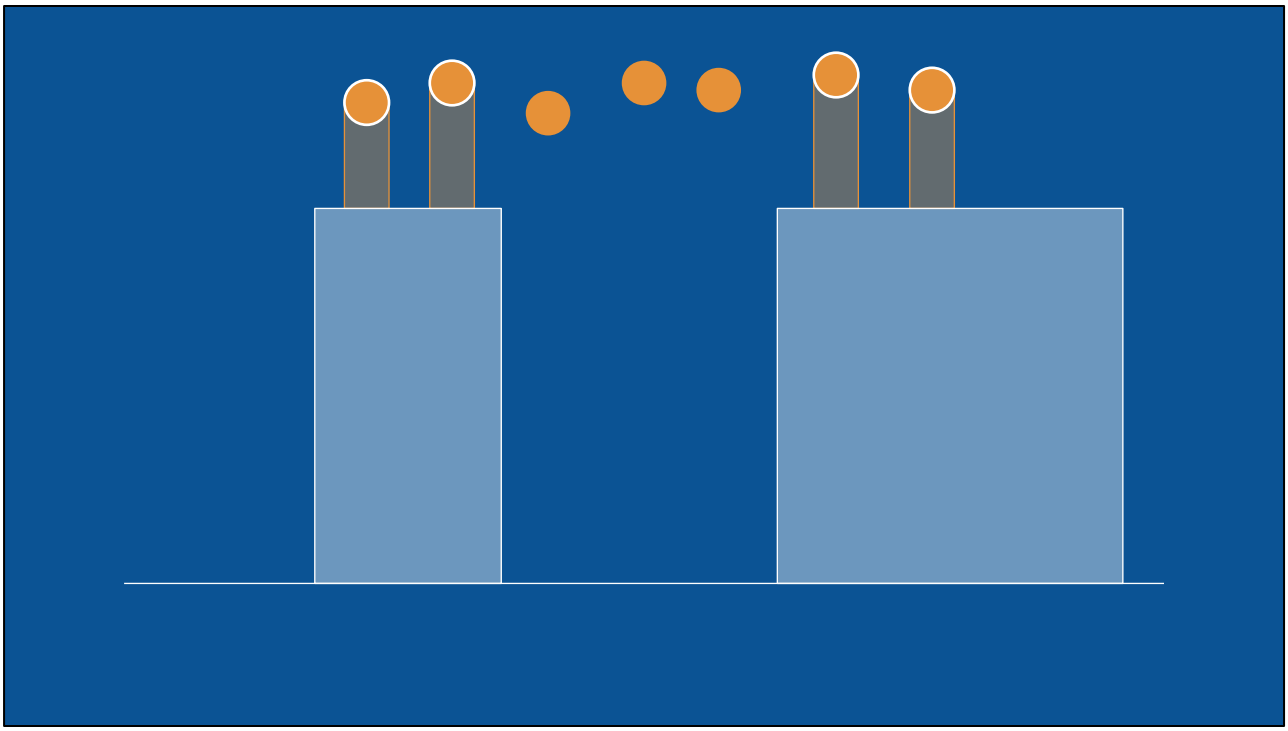
Let’s imagine you’re scoping out whether to build a platform. You do some market research and realize there are a few use cases you want to support.



You want to build a platform to support them. You aren’t aware of any use cases in the middle, and it will cost a bit more time and effort to build it, so you just leave it out.

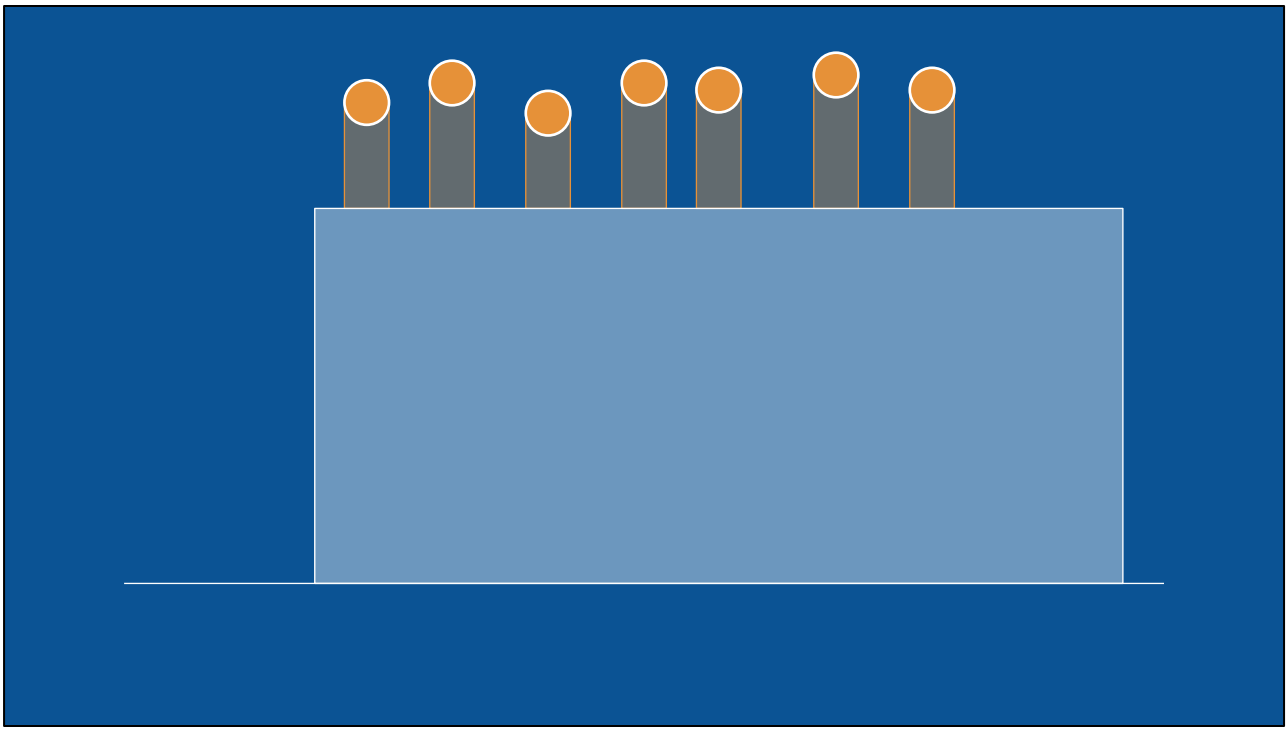


You unlock the value, the 3Ps use it. Everybody wins!



… But your market research had a blindspot. Maybe there was a whole region of use cases you didn’t know about--maybe an industry or country your market research missed.

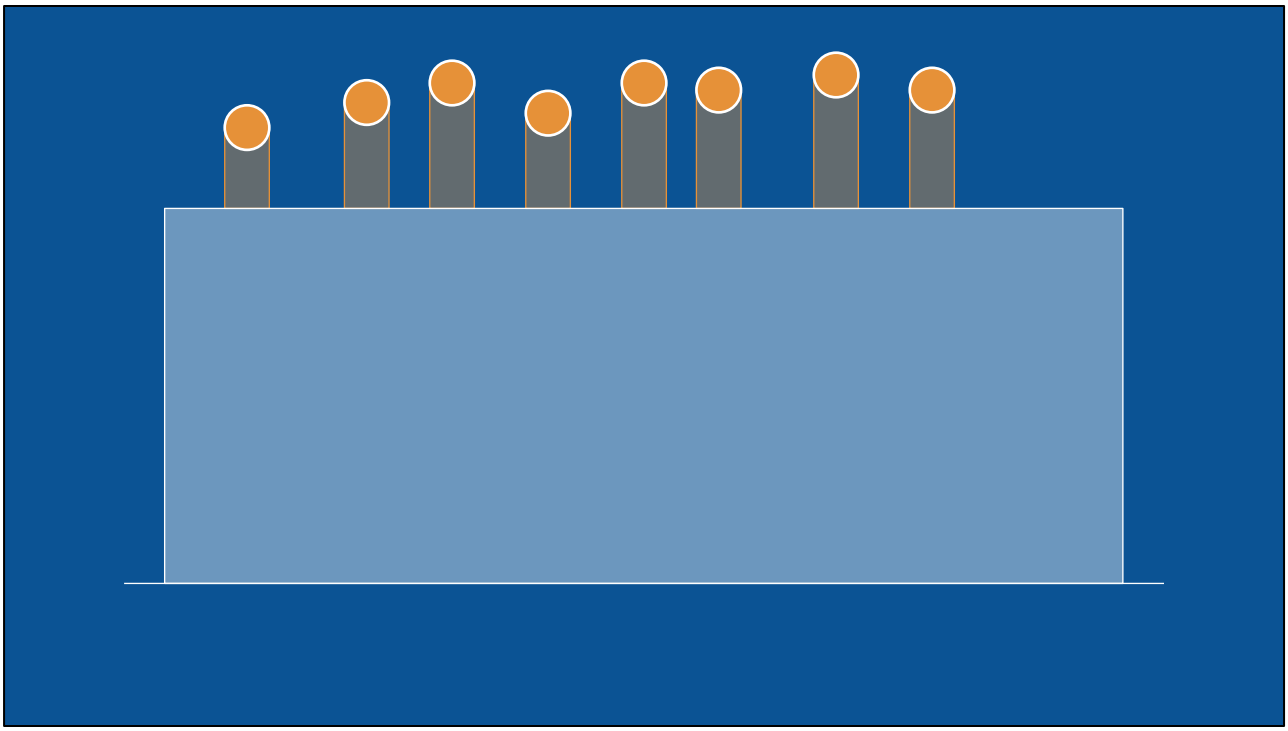
Even though you didn’t know about them, they’re there.



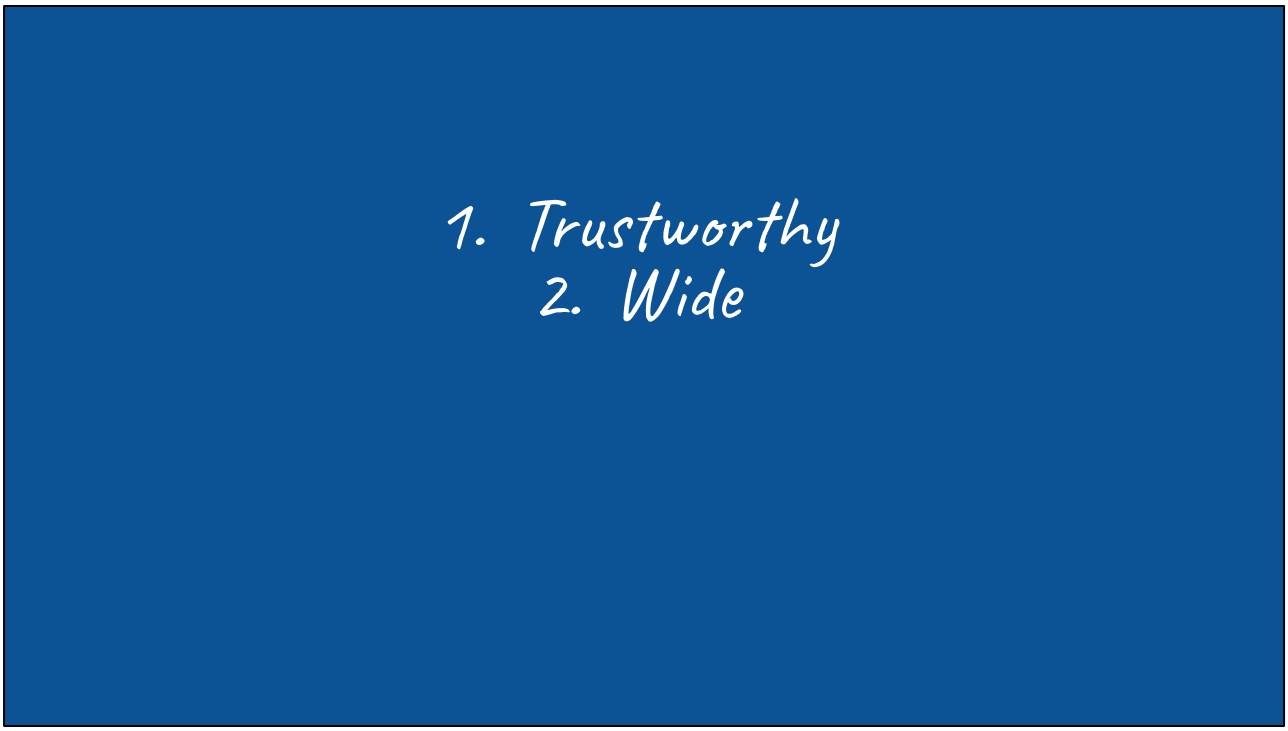
If you would have built wider, you would have been able to capture even value you didn't know about, for free!

Now, building a platform is expensive (we'll cover this in depth later). So you don't want to go crazy with this approach.

But as a rule of thumb: if you know of two concrete use cases with some space in **between** them, and it's not too much more expensive to support the in-between use cases too, just do it!

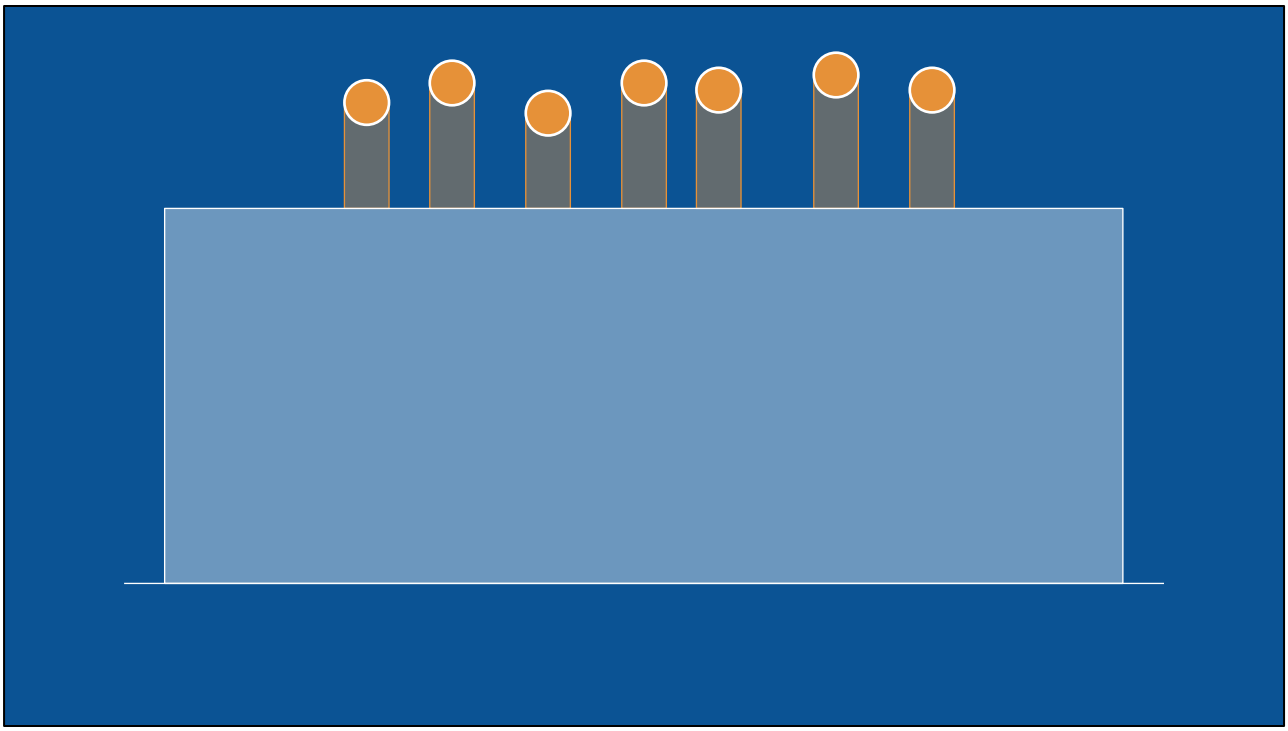


Building *outside* of known use cases can be worth it too, if it's not too expensive. You might get more use cases. … Or you might not.



So that’s the next property of rational platforms: they’re wide.

On to the next one!



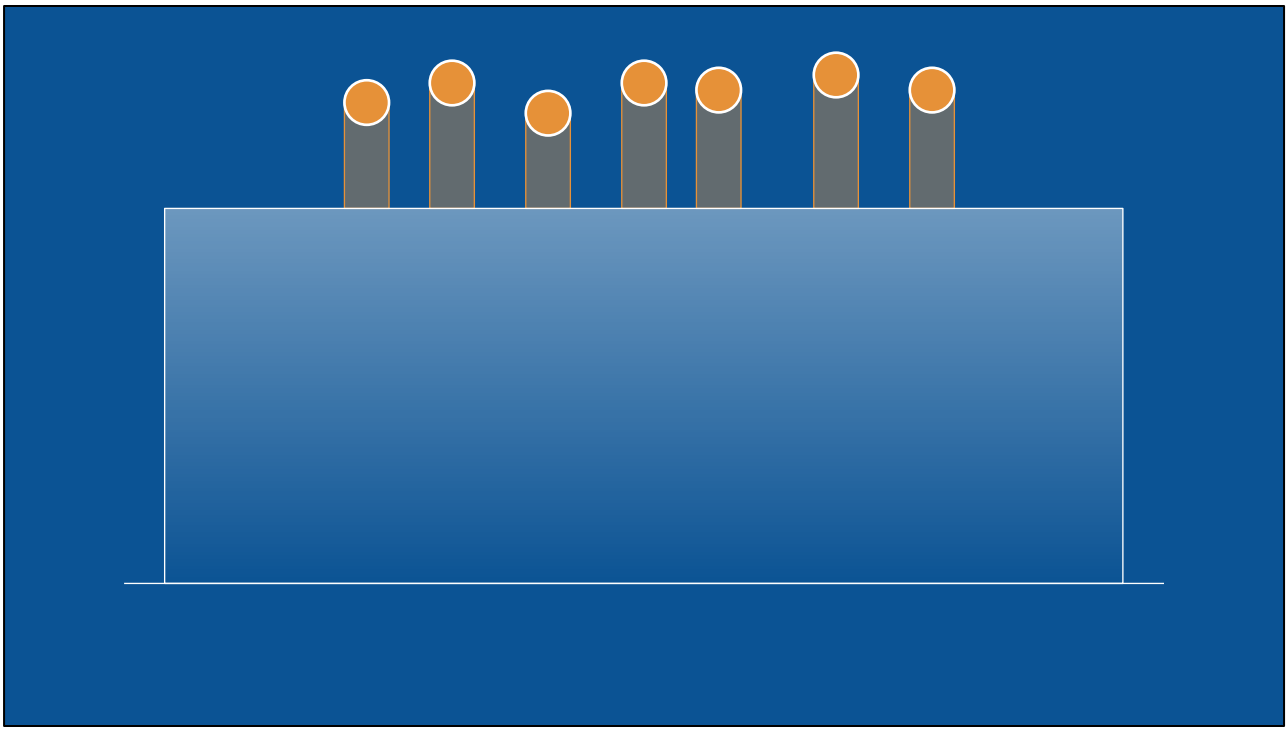
So far we've been looking at our platform like a monolith.

But of course, inside it's whole hunks of code, layered on top of one another.

Exposed semantics vs Internals

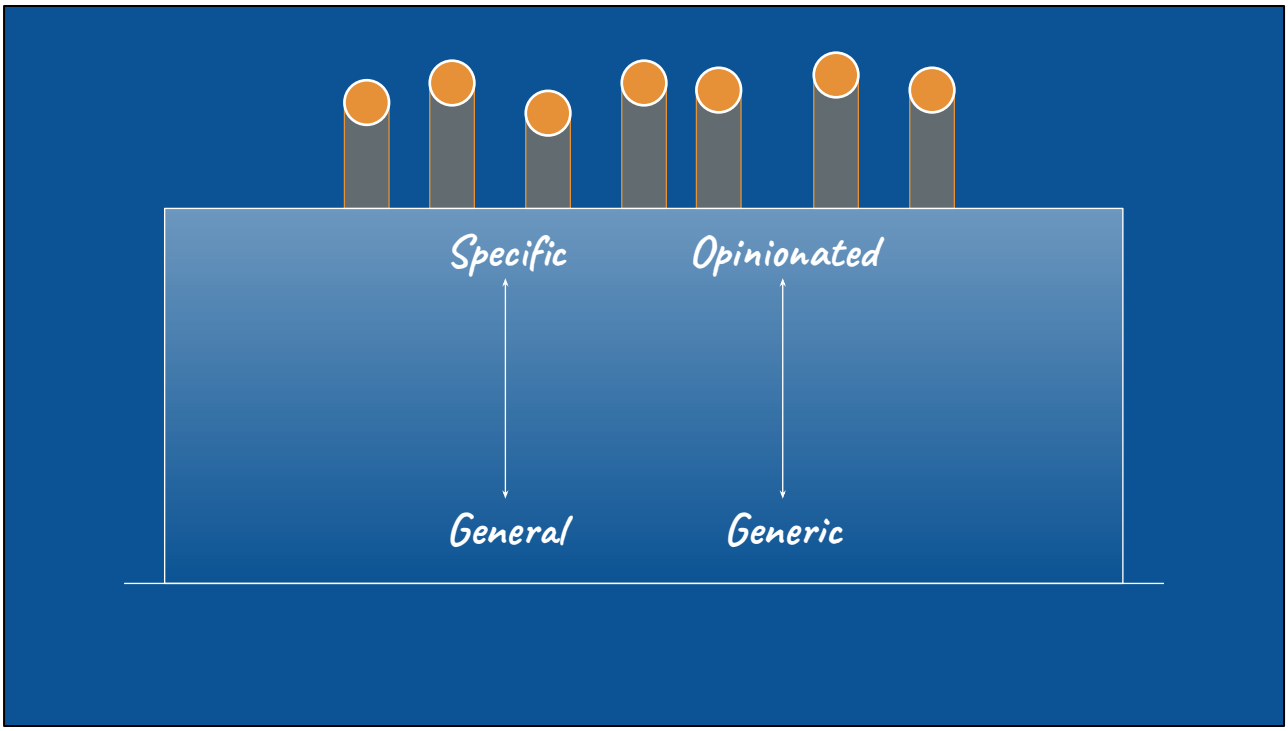
There's this difference between the exposed semantics--what 3Ps can see and rely on--and the internal implementation.

Exposed semantics roughly means your API--but as we'll see later, they aren't precisely the same.

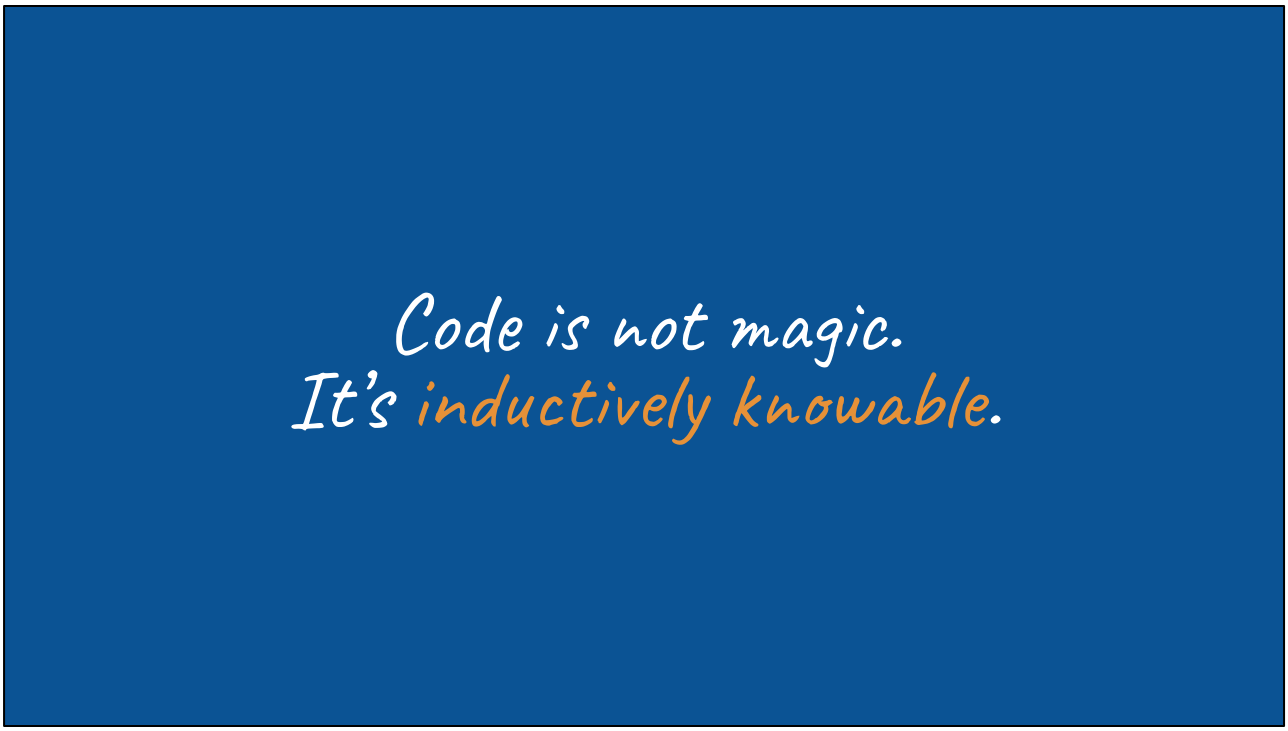


Inside the monolith, there's kind of this gradient thing going on within the implementation.

In a rational codebase, each layer builds on top of lower layers, with no mislayerings.



At the bottom you have more generic things, and more specific things on top of more general.

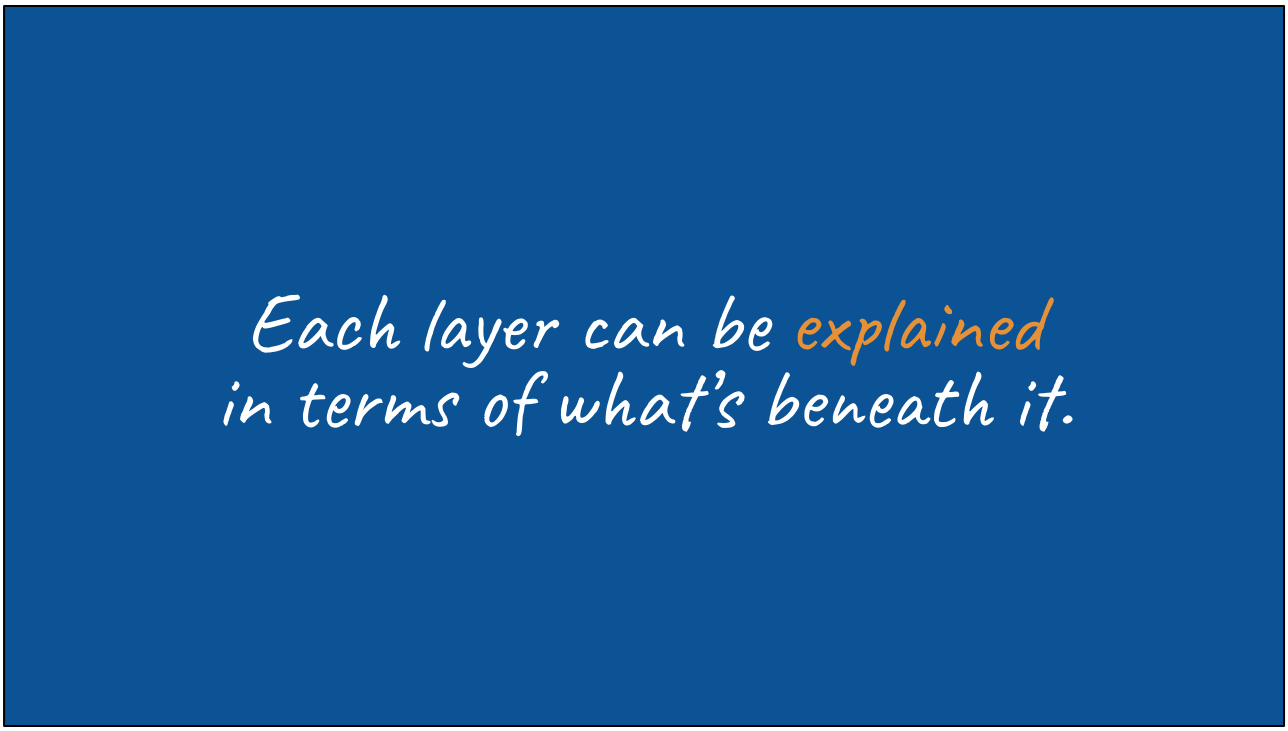


This gives one of the cool properties of code: it's not magic.

Even if you don't understand how the whole thing works, you know that at any layer, you could peel back one more layer of the onion, and that layer would also be possible to comprehend.

The whole system is **inductively knowable.** This is part of the magic of abstraction.

But it works best when things are well layered.



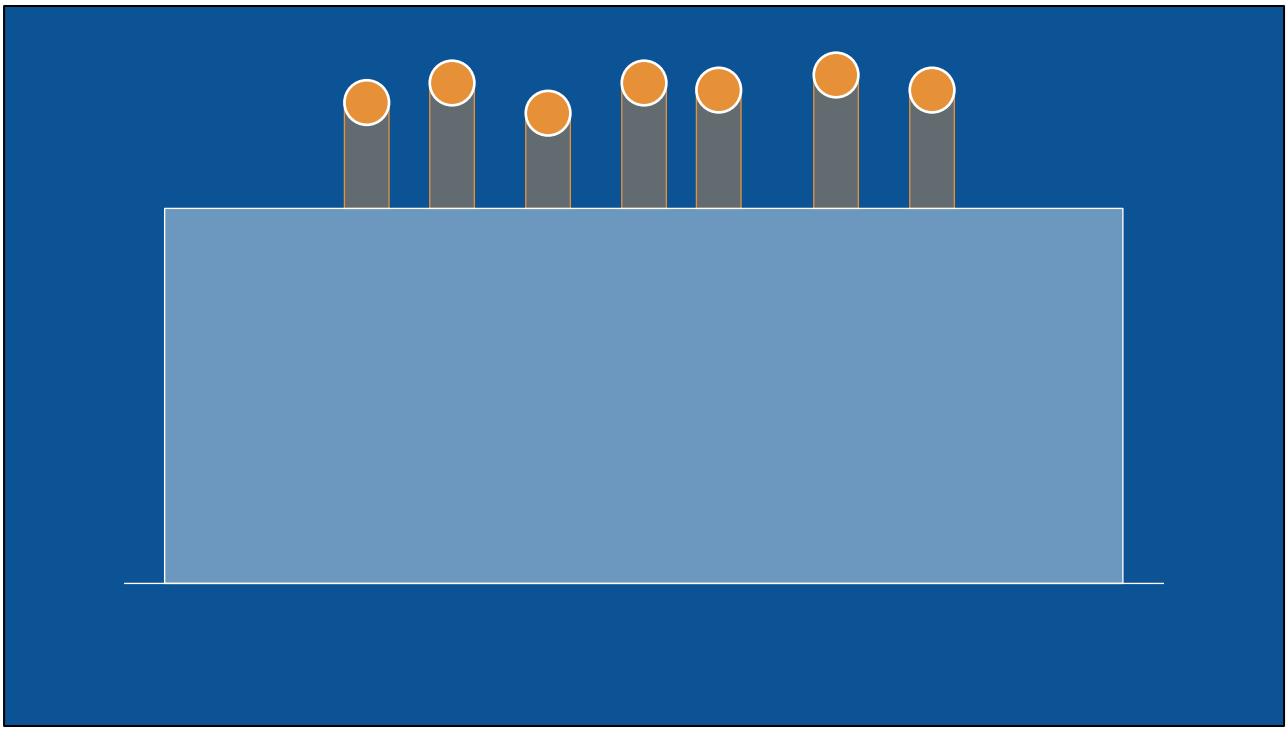
Well-layered platforms are ones where each layer can be explained in terms of layers beneath it, plus a bit more logic layered on top.

You can think of the conceptual complexity of your platform as the word count of hypothetical fully-comprehensive developer documentation. If a platform is cleanly layered and well-factored to be explainable like this, the word count can be quite small. A messy, inconsistent platform would need far more documentation to fully explain to 3Ps all of the edge cases. Every time a 3P experiences a surprise using your platform, that's a sign that your platform isn't fully rational.

Put another way: minimize the amount of magic necessary to explain how your system works.

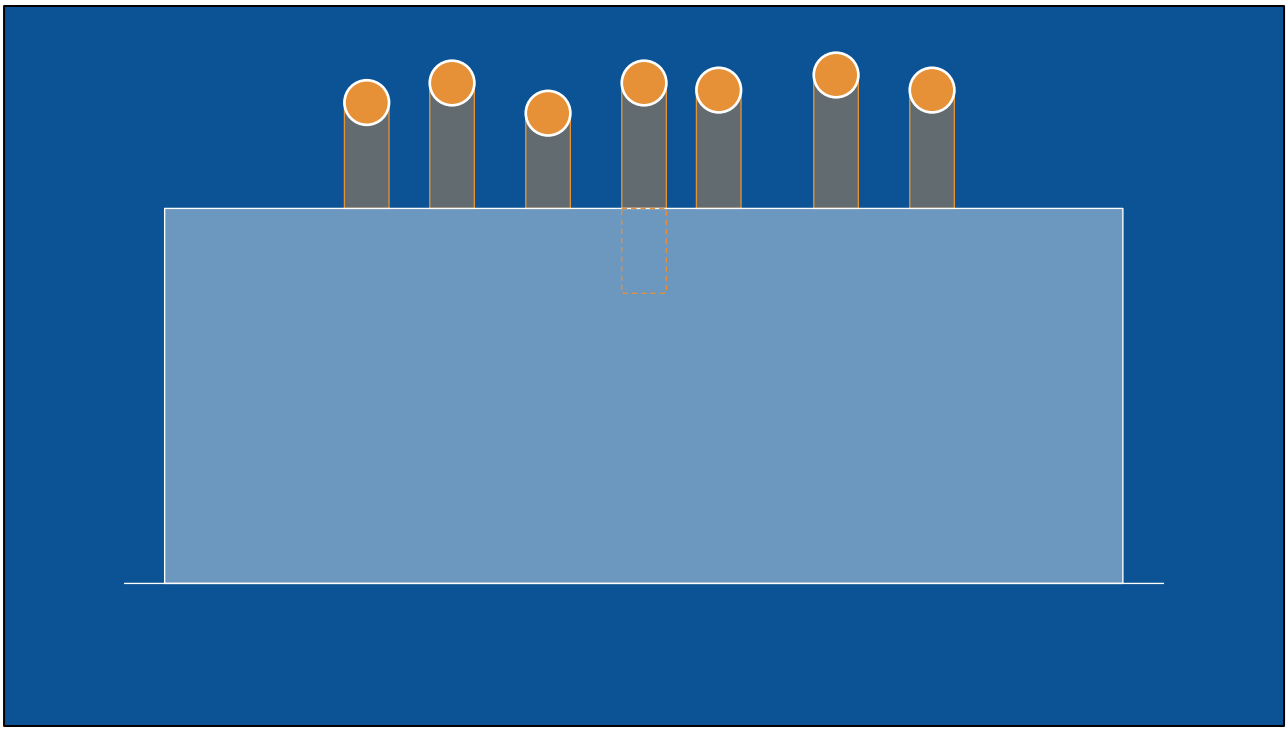
So the next property is that platforms are explainable.

Let’s get to the next one.



We've talked about how within this big black box, there are layers of code. But from the 3P's perspective, all they can see are the exposed semantics. Everything else is a black box.

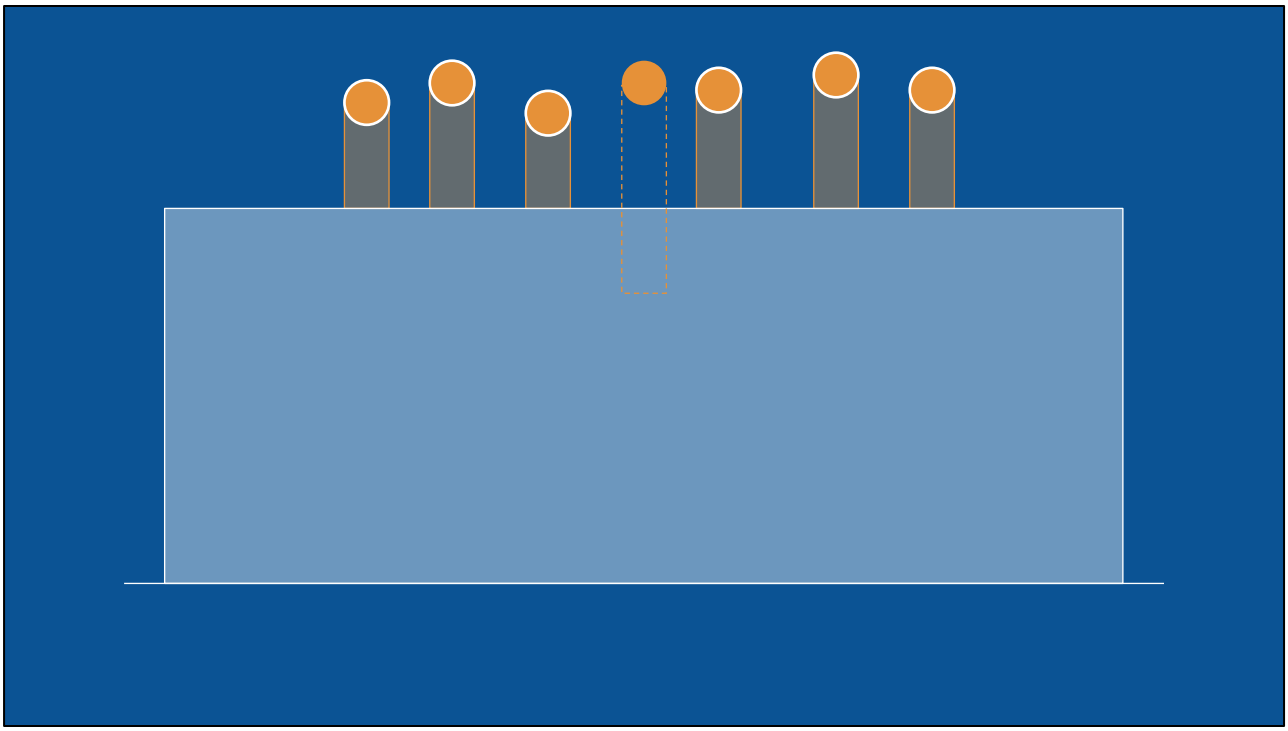
That's good--it's less for 3Ps to worry about. And it allows you more wiggle room to refactor the internals without breaking them.



But what if your exposed semantics aren’t exactly what the 3P needs?

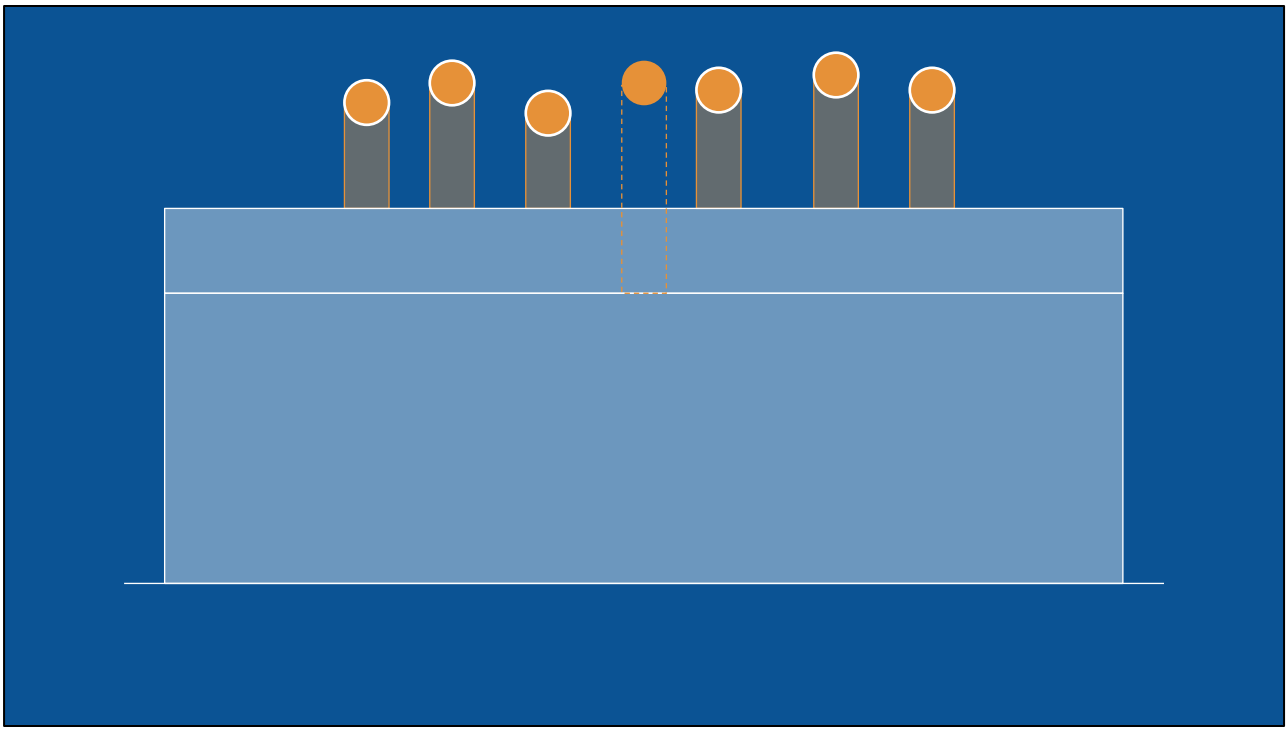
They need to configure some behavior *within* that black box.

But they can’t, because they can’t reach inside!

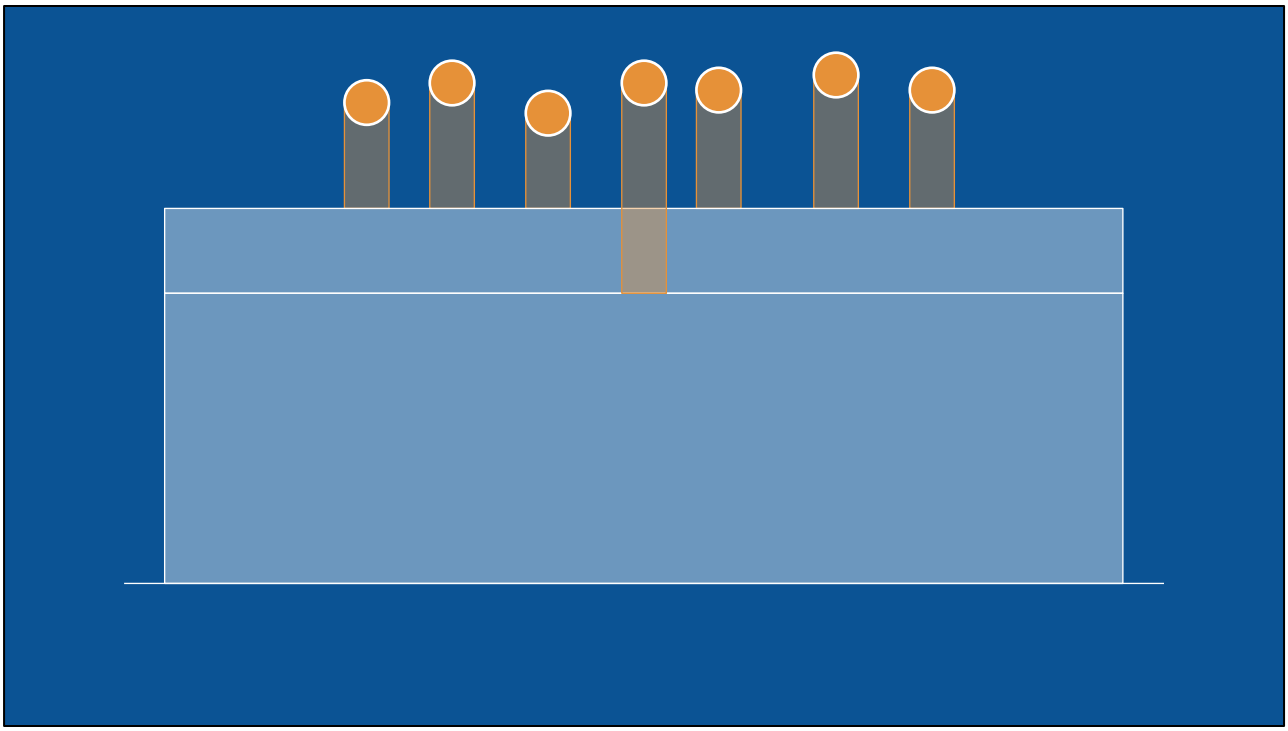


So they just have to give up. They lose, you lose, everyone loses.

But what if you had exposed some of your internal semantics, as layers?

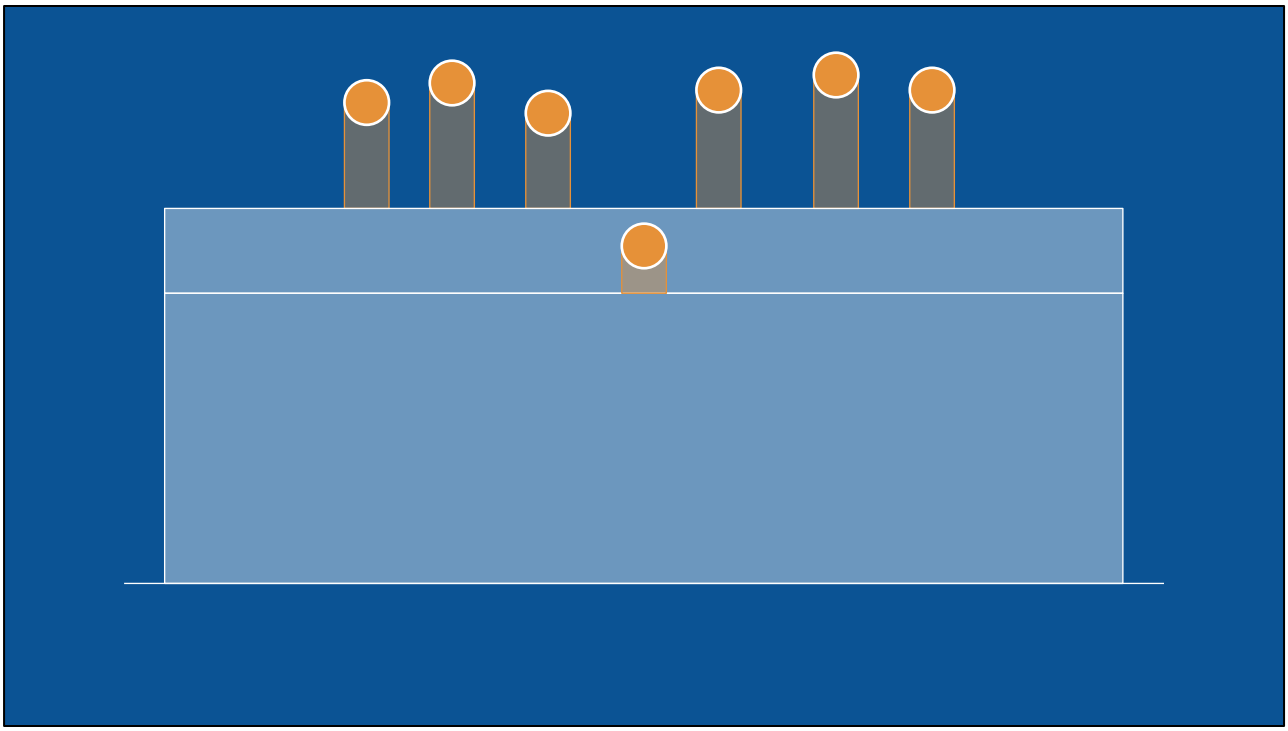


Now it’s possible for that developer to peel back a bit and depend on a slightly deeper layer.

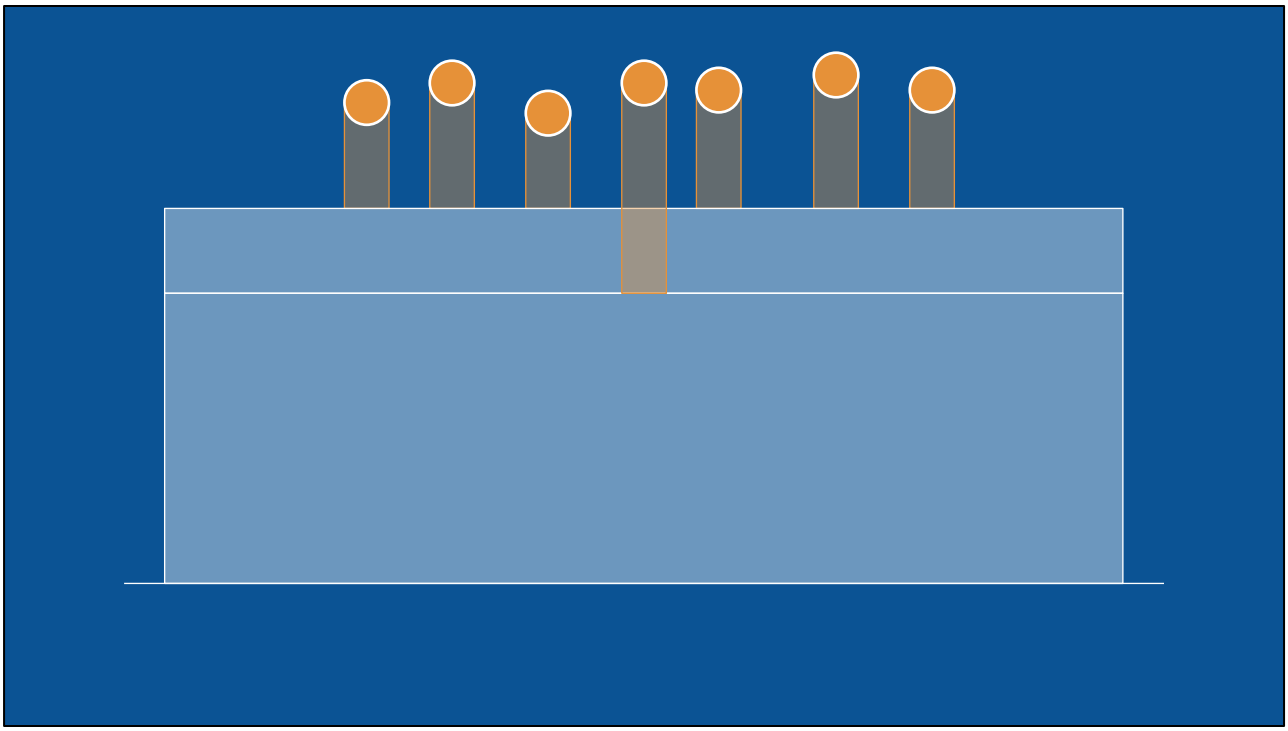


That developer can now build to the lower layer (opting out of the higher layer), while the other customers still use the higher layer. Everybody wins!

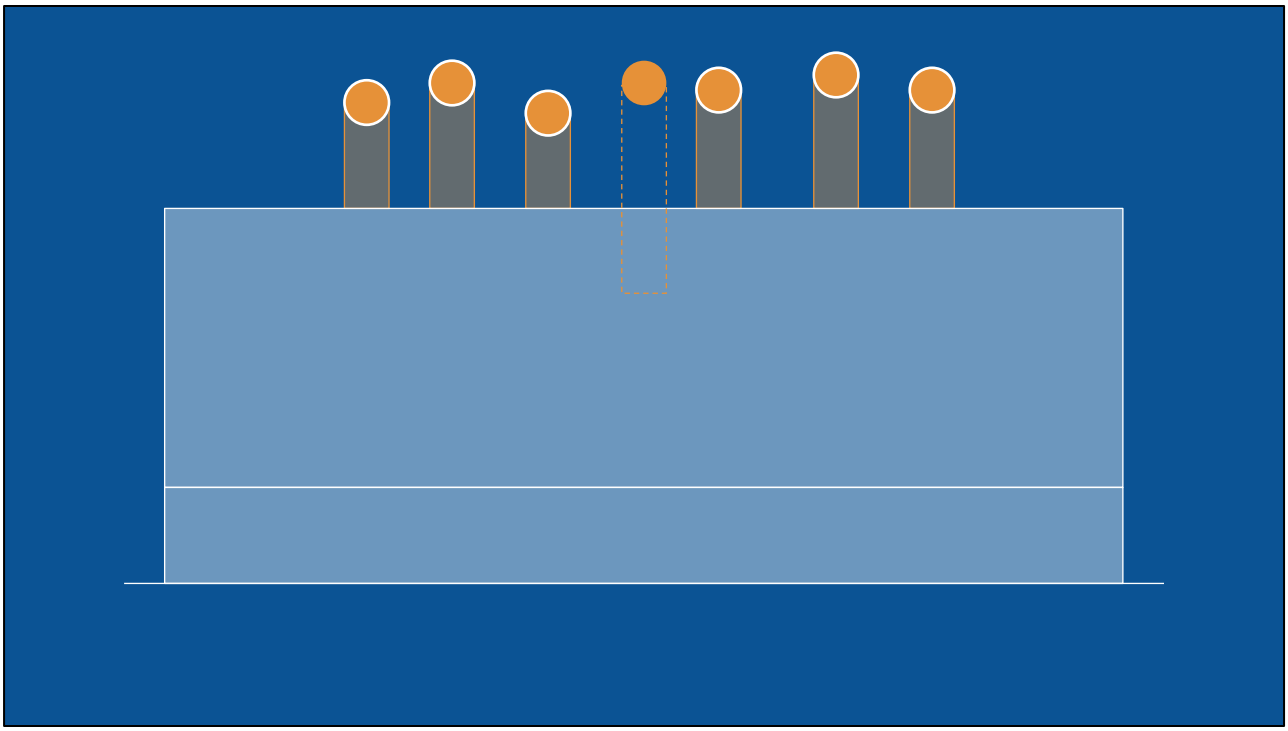
In general, when a monolith is broken up into smaller pieces, all kinds of new use cases become viable that weren't before.



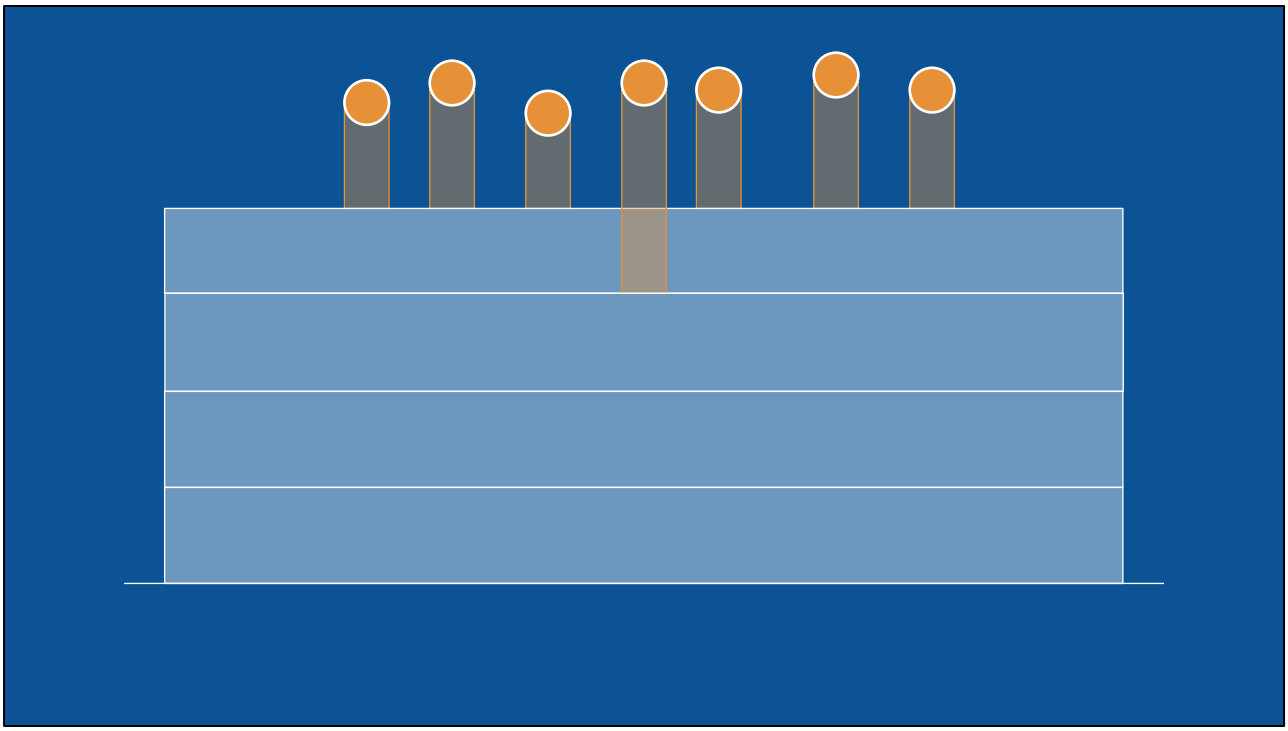
Interior layers might also support value that is within the scope of your platform but that you don’t want to build yourself.



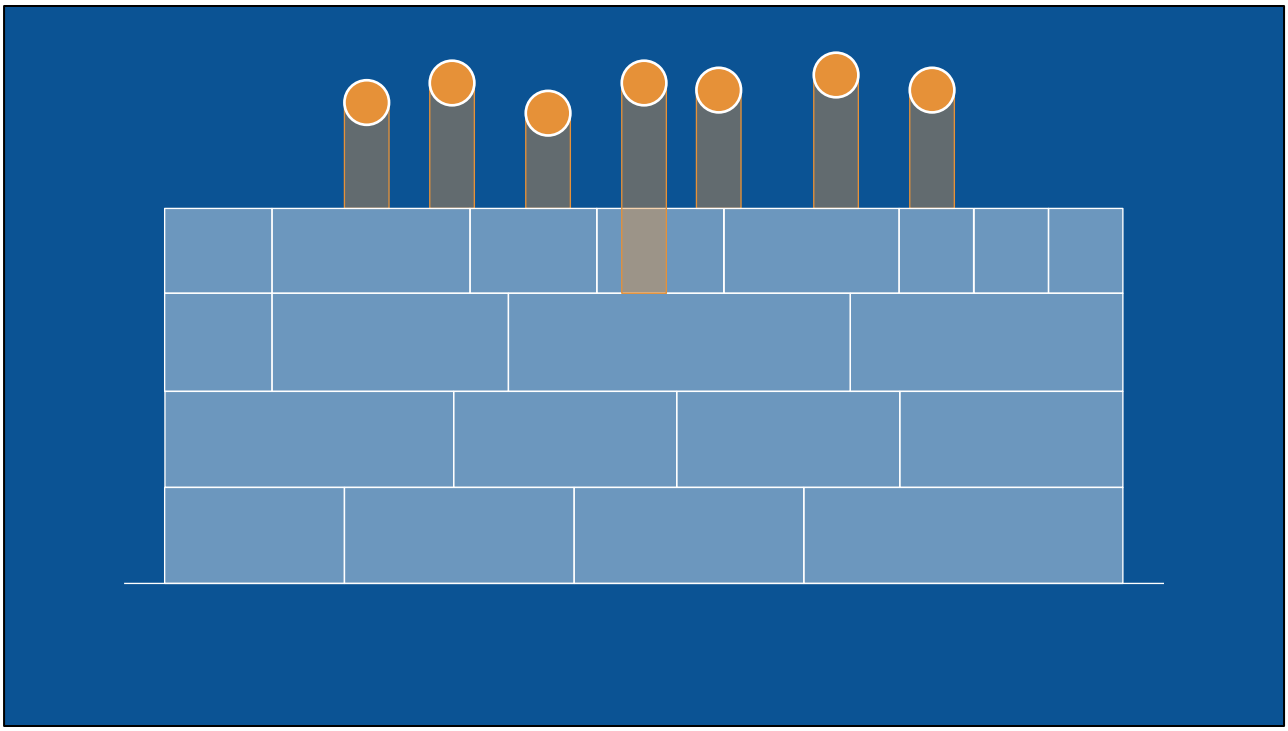
This layer works for this developer, because it’s not too far down.



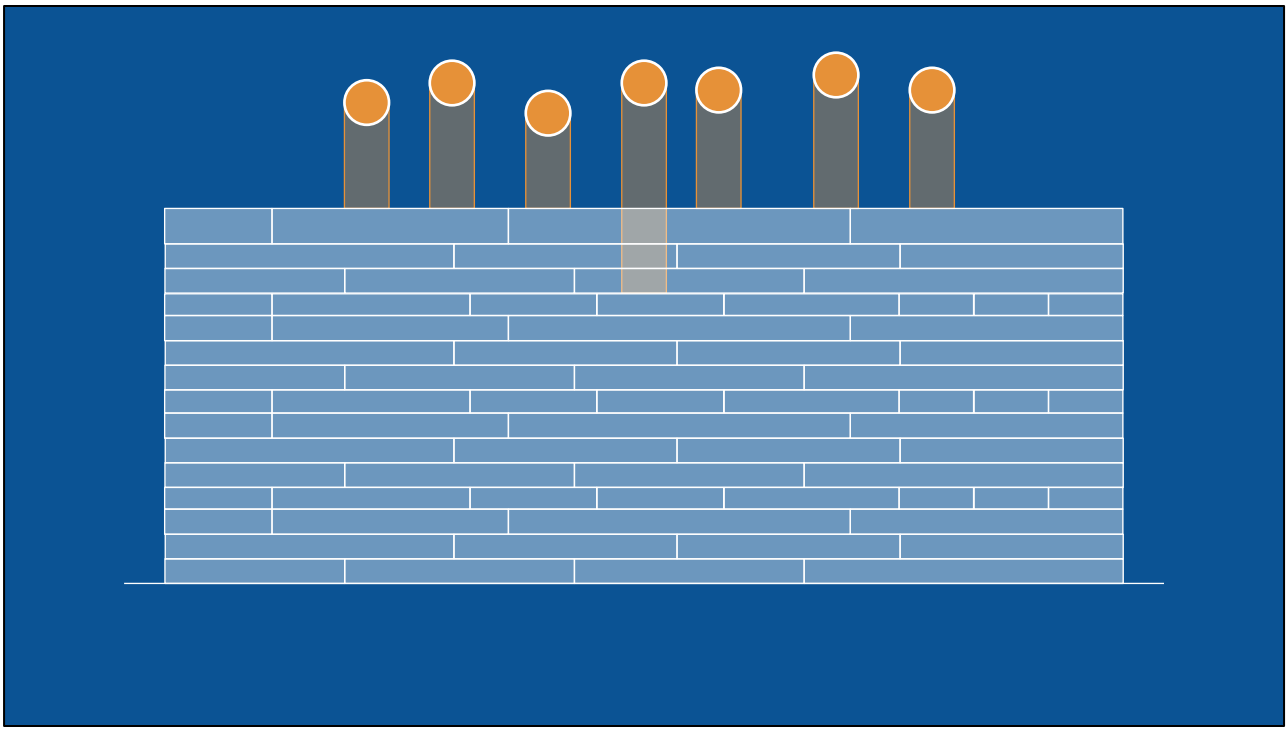
Of course, if the developer had to drop down too far to get to the next lower layer, it might not be viable--it might be a bridge too far.



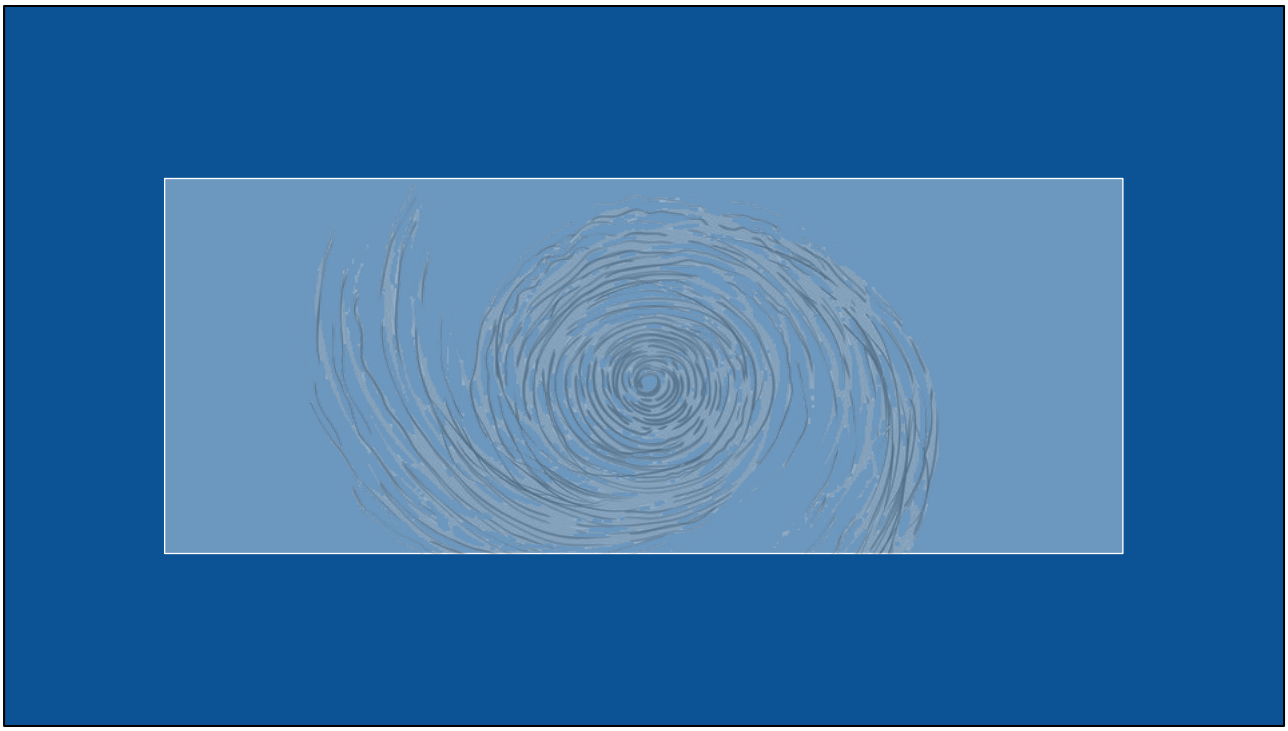
So in practice you want your layers too not be too thick.



You also want to make your layers not too wide, but broken up into chunks, so if someone wants to skip just one part of a layer, they don’t have to throw out all of it.

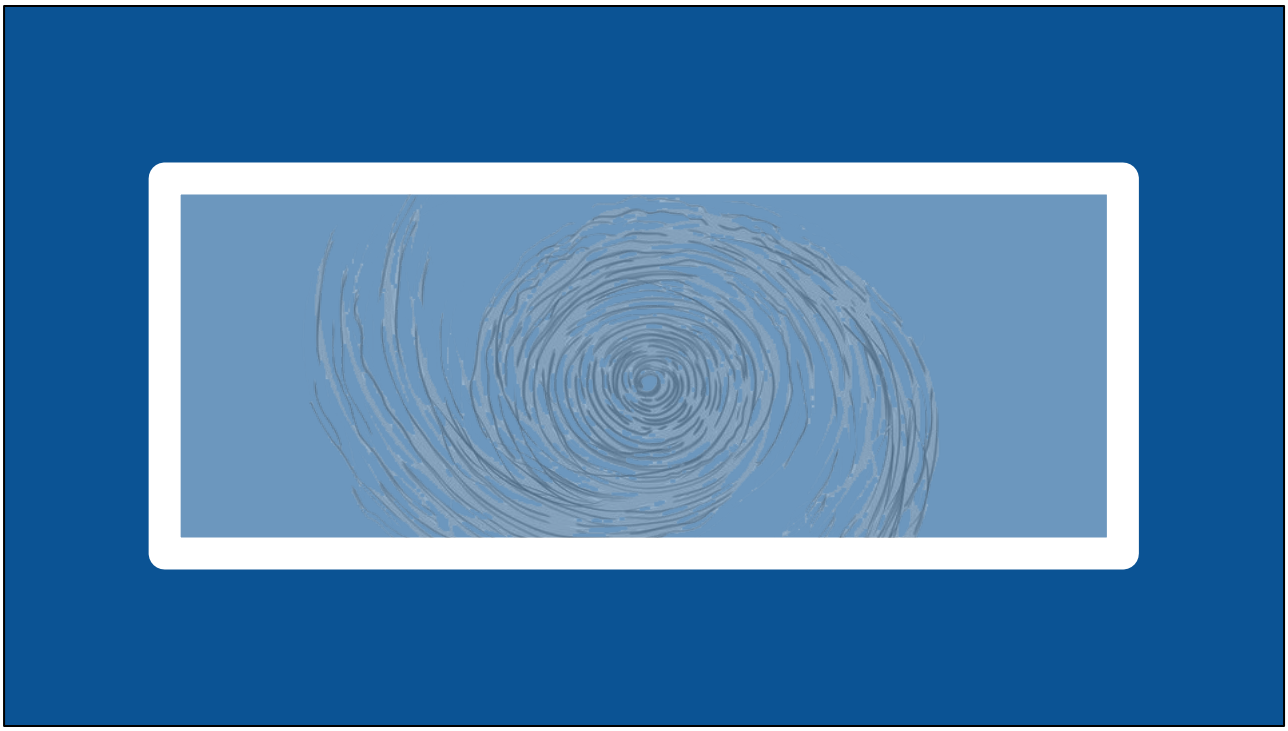


Of course, it is possible to take this too far.



Think about each block.

Inside the block, the internal semantics, are way, way, *way* easier to change. Just coordinate internally, and make sure no exposed semantics change. This gives you flexibility. It’s very fluid, like water.

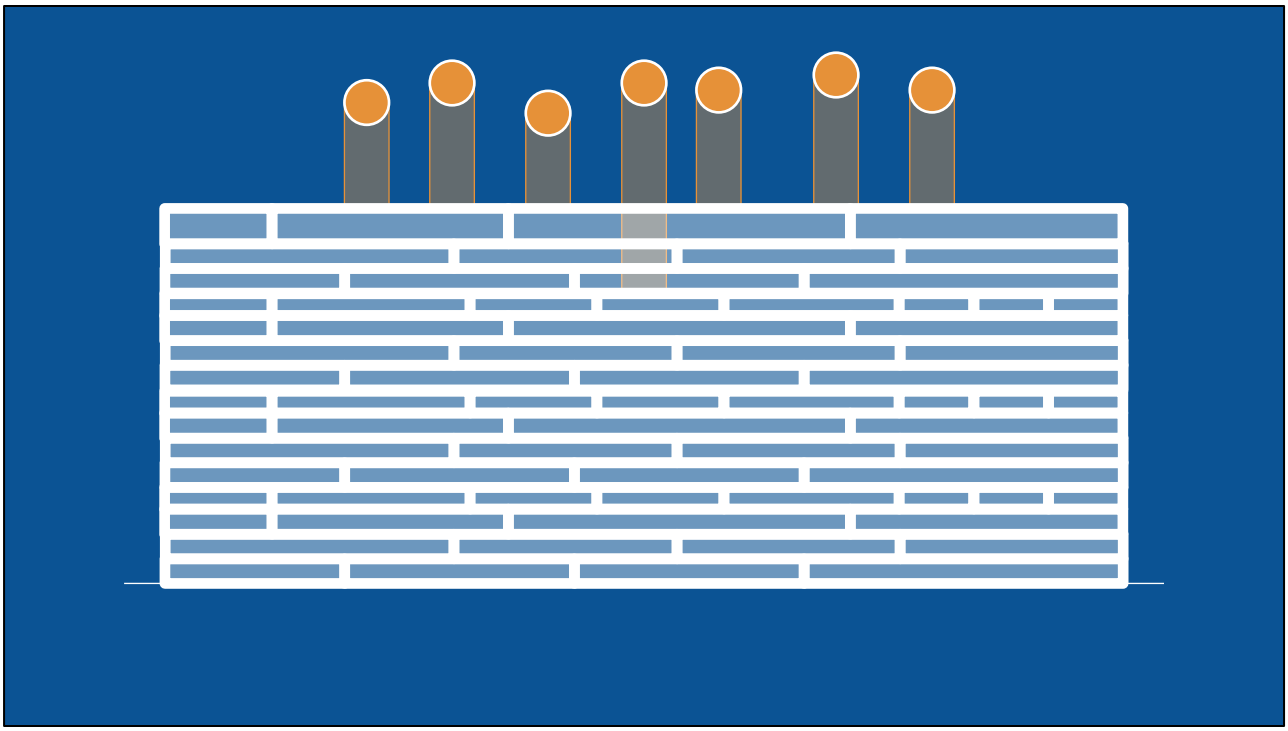


Think about the exposed semantics, the surface area of each block, as being **frozen** in place. To change it, you have to tell all of the developers, and get them to change… something they'll likely drag their feet on.

The more developers you have that rely on an exposed semantic, and the less leverage you have over them, the harder it will be to compel them to change.

The exposed semantics are much more solid, like ice.

In addition, every bit of exposed semantics you have is more surface area to actively maintain (documentation, bug fixes, support requests, etc), and it can get quite costly.

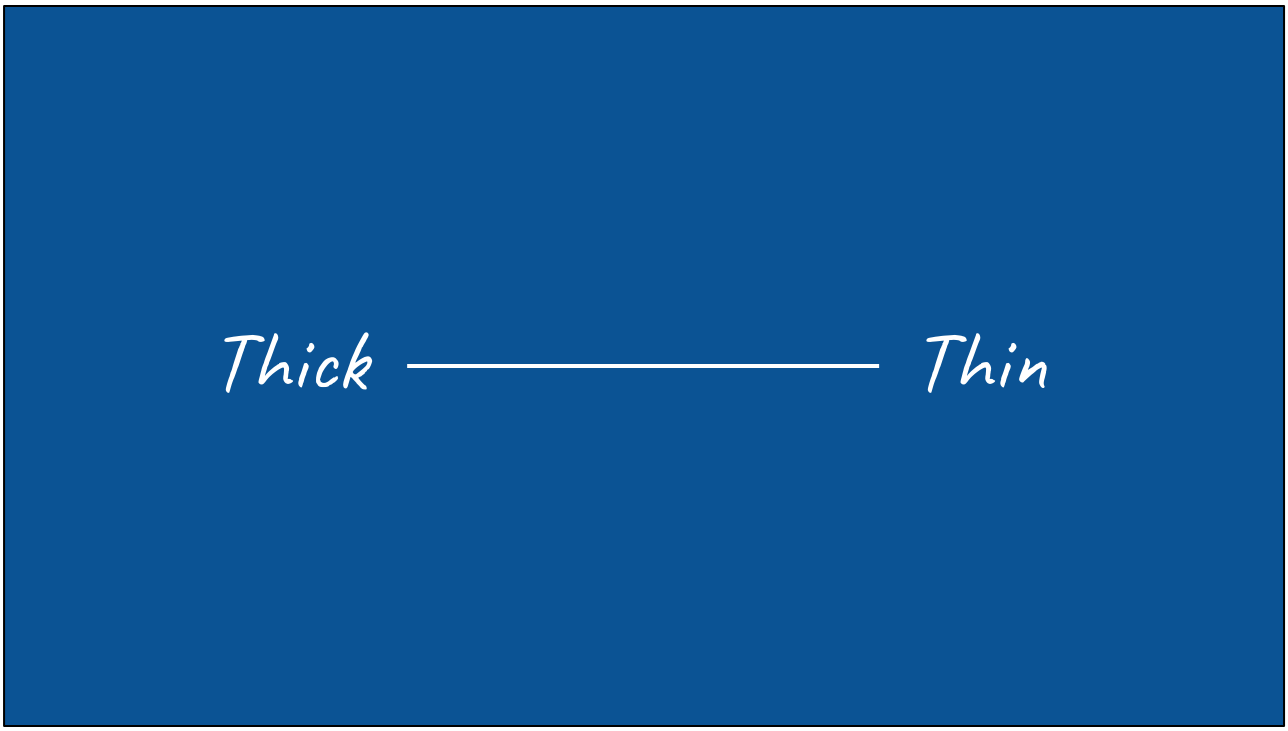


So if you freeze everything, you have no wiggle room. You can’t innovate, or change.

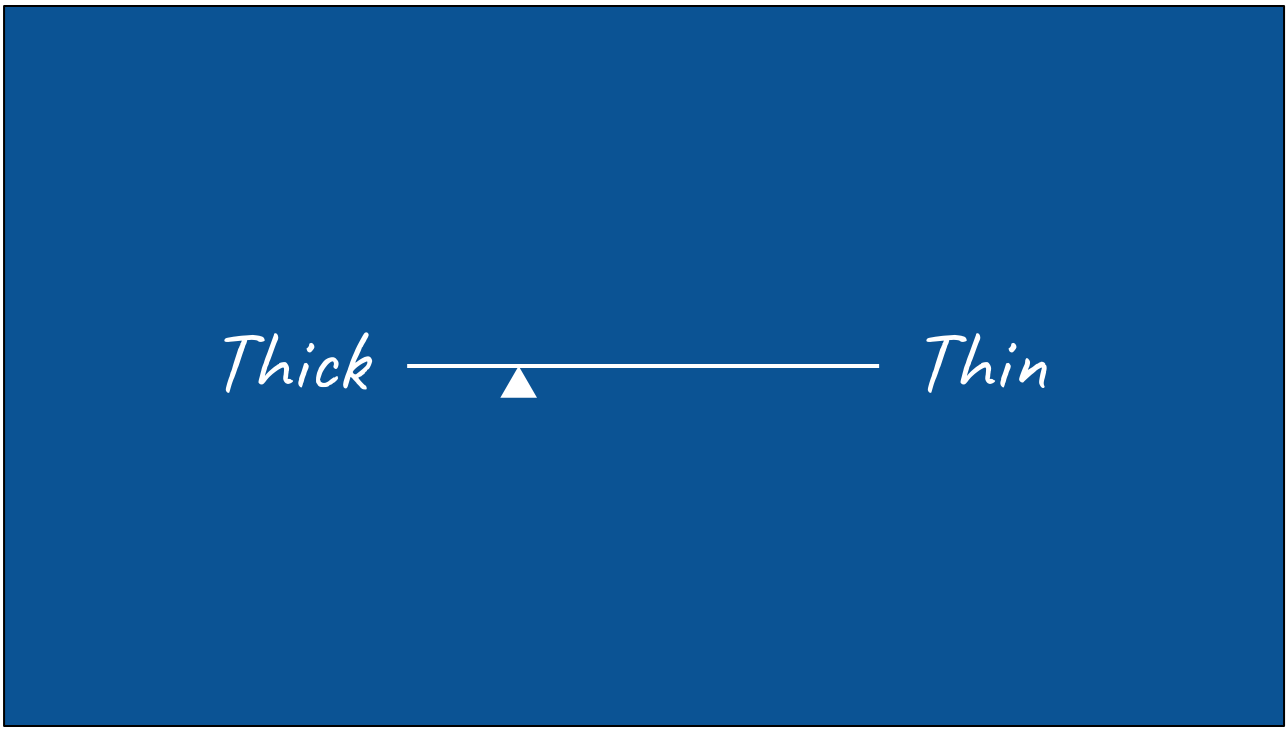
If the environment changes quickly--which happens more often than you’d think--you’ll be frozen in place, unable to adapt quickly enough.

You’re brittle!

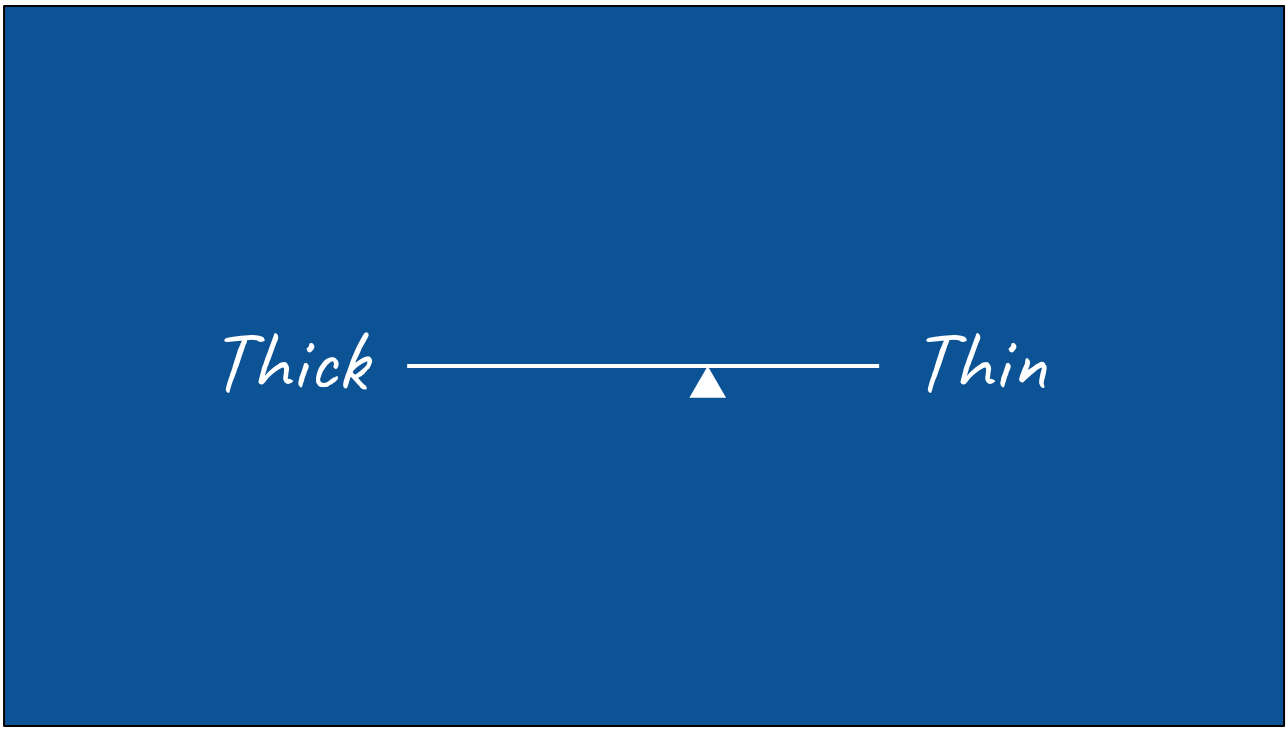
That structure of exposed semantics is an opportunity… but also a liability.



It’s a tradeoff.

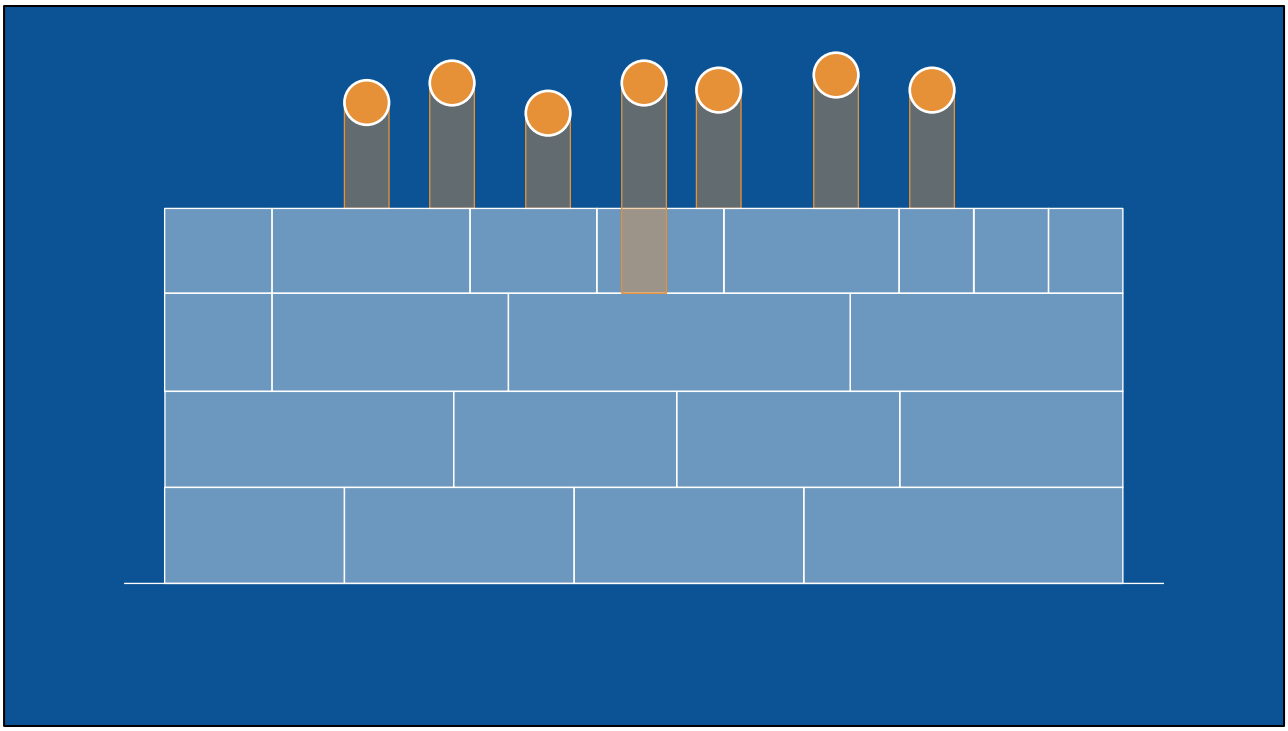


And like all tradeoffs, the balance point is contextual.



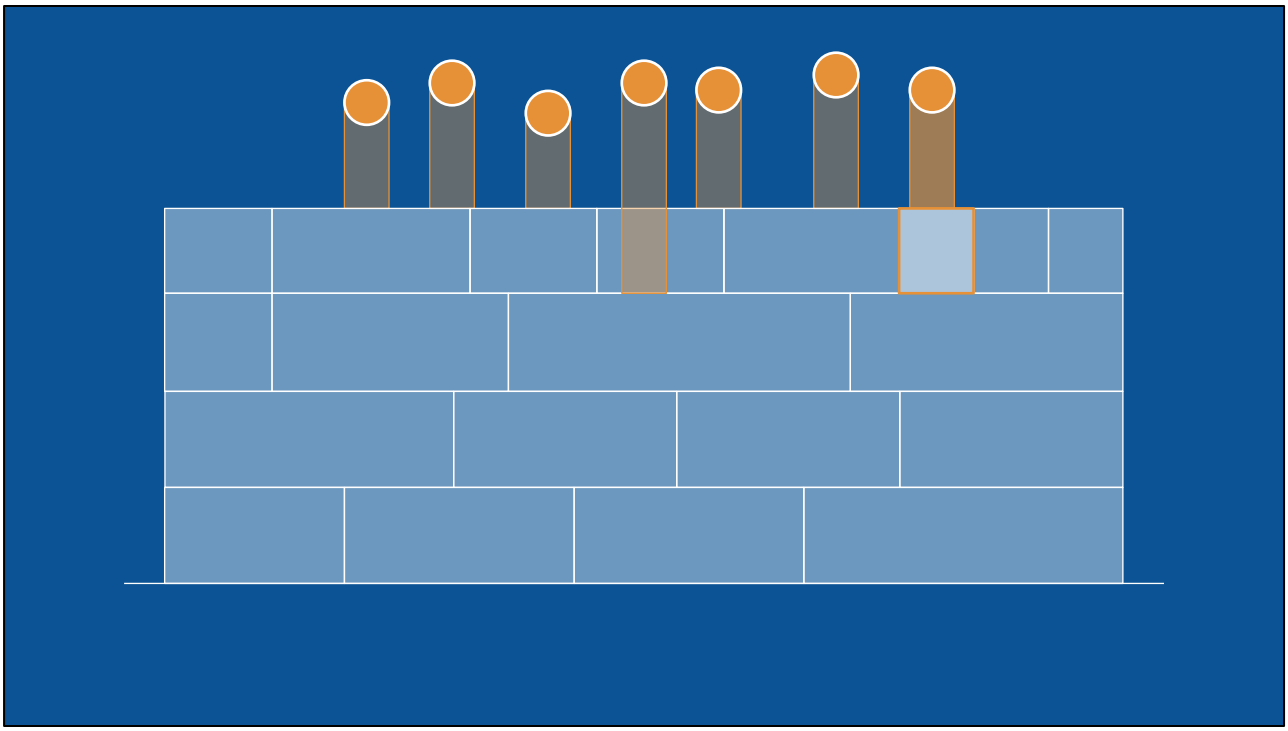
It varies in different situations, and can vary over time.

OK, so we can add another property: well-layered. No monoliths.

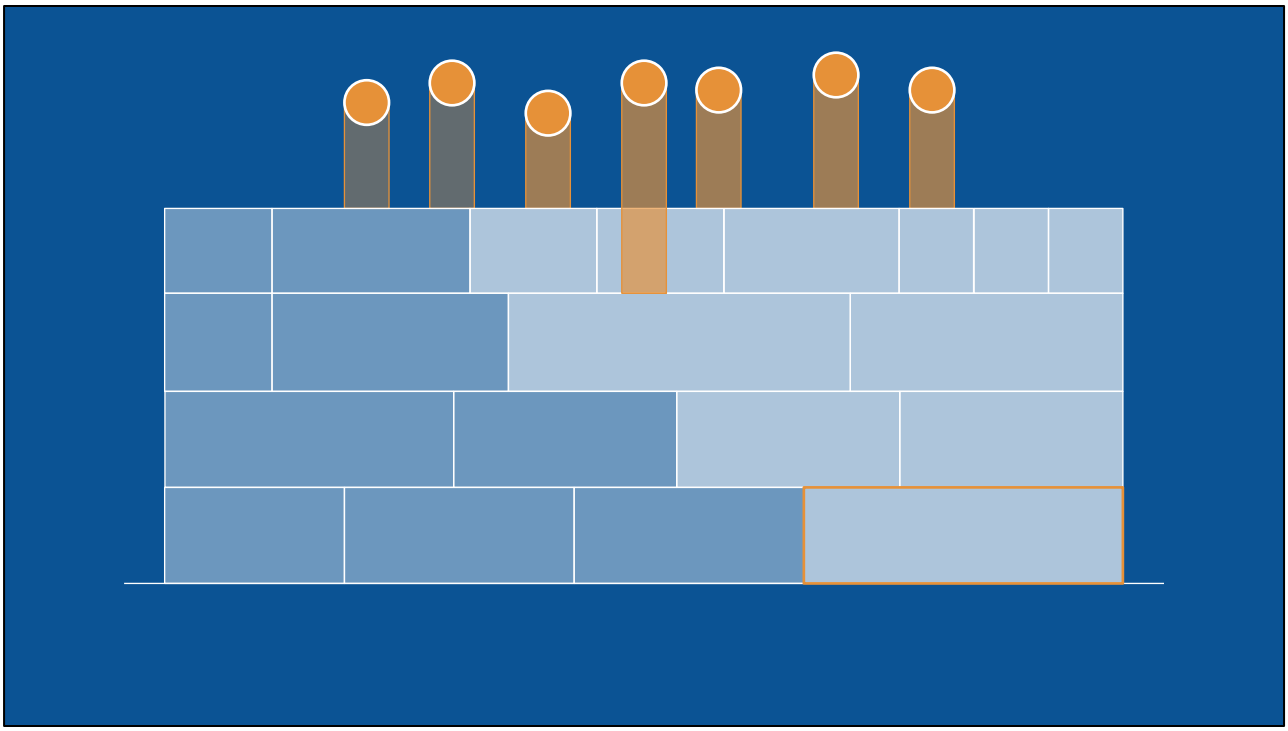


Let’s look back at our layer cake.

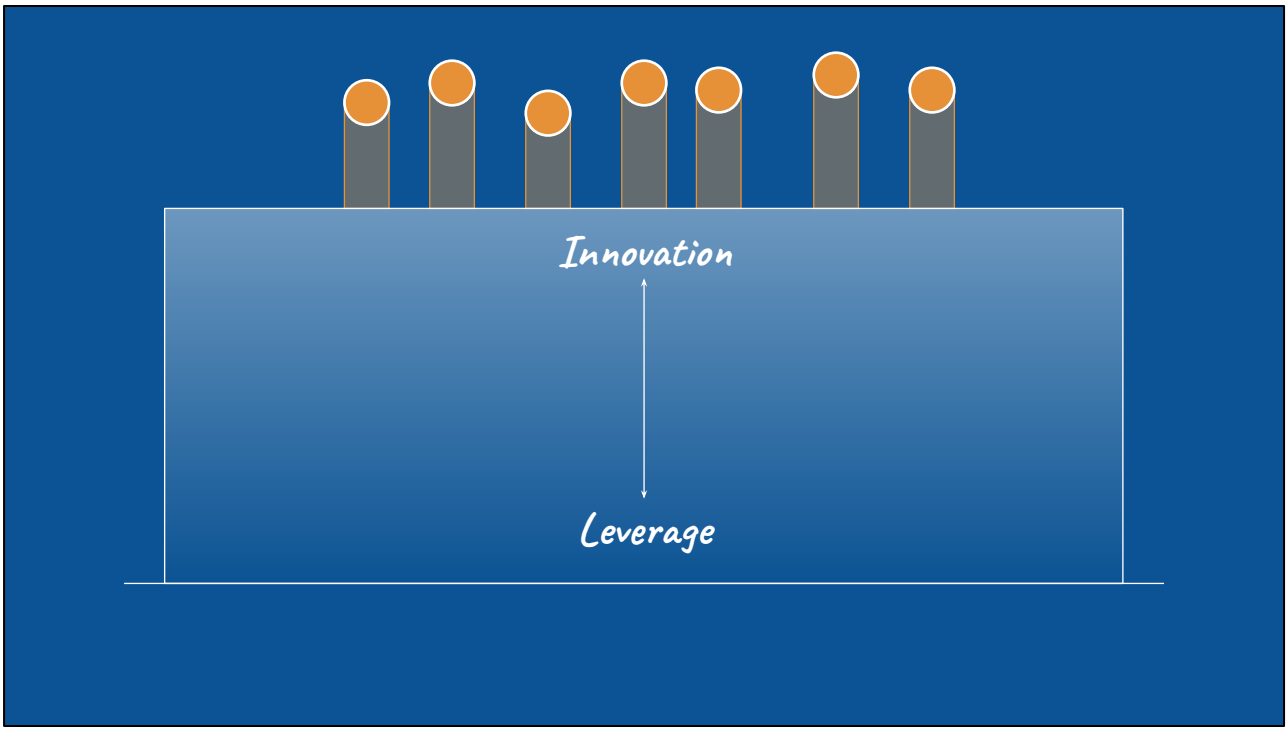
How do the blocks at the bottom differ from the ones at the top?



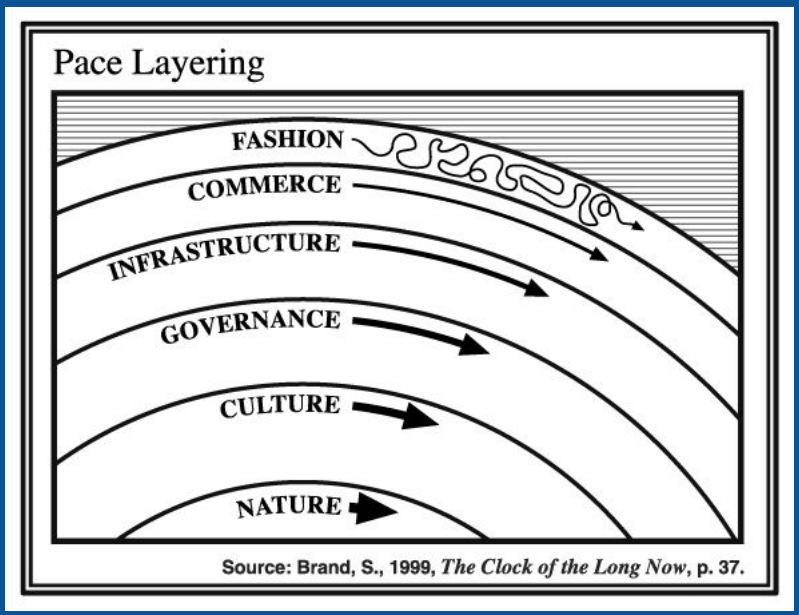
The ones at the top have very little resting on them. They can move quickly. If you need to change them, you only have to alert the people who use them. If there's a bug in one, it only affects the people above it.



Whereas lower ones have a lot more resting on them. A change in one of them--or a bug--could potentially affect *tons* of stuff above them.



Higher layers can move more quickly--they can innovate and change faster.
But lower layers have more leverage--an improvement at a lower layer can have *huge* implications for the rest of the platform.



This kind of layering of paces happens in all systems, by the way. Stewart Brand called it **pace layers**.

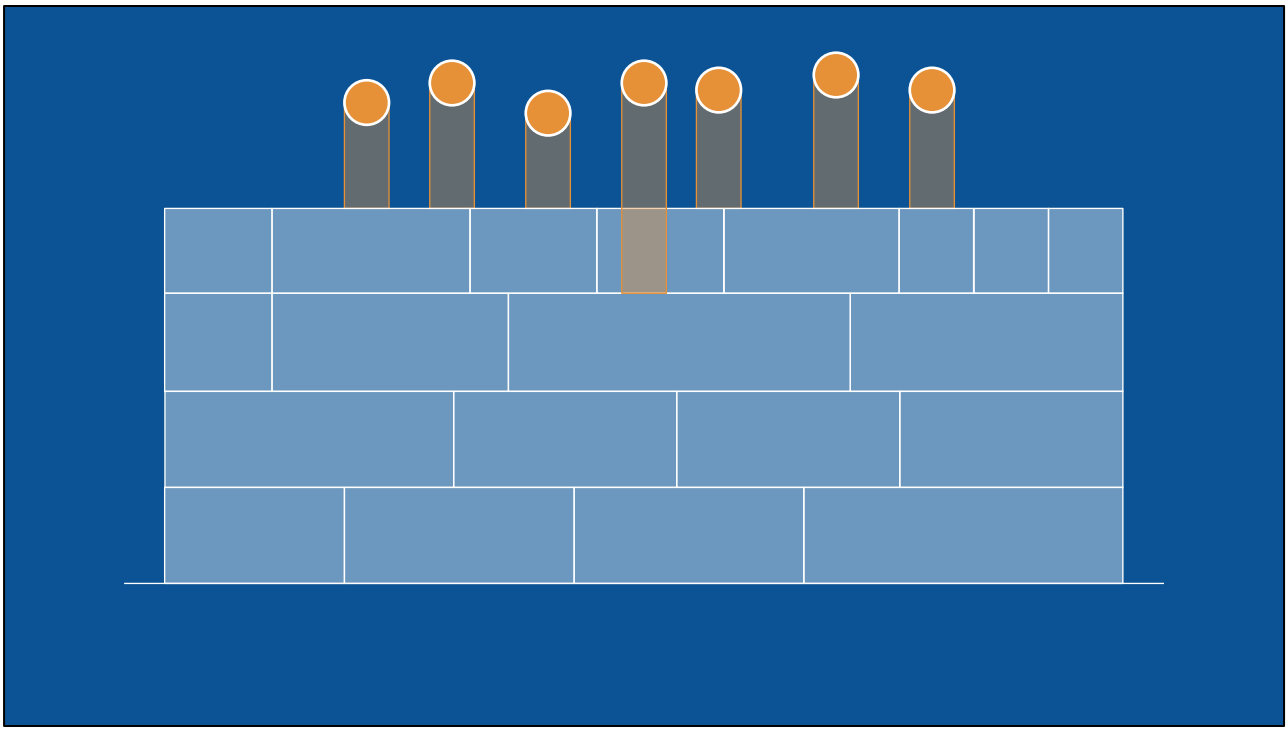
At higher layers, things can move faster. No--they *must* move faster. Because other players at that layer can move at a certain speed, so if you don't keep up they'll out-innovate you and you'll be left behind.

It's really important to know where in the pace layer you are. If you think you're at the top but you're at the bottom--say you think you're adding a very minor API for a very specific partner, but you're actually adding a fundamental security primitive exposed to the whole platform--then you're Going To Have a Bad Time.

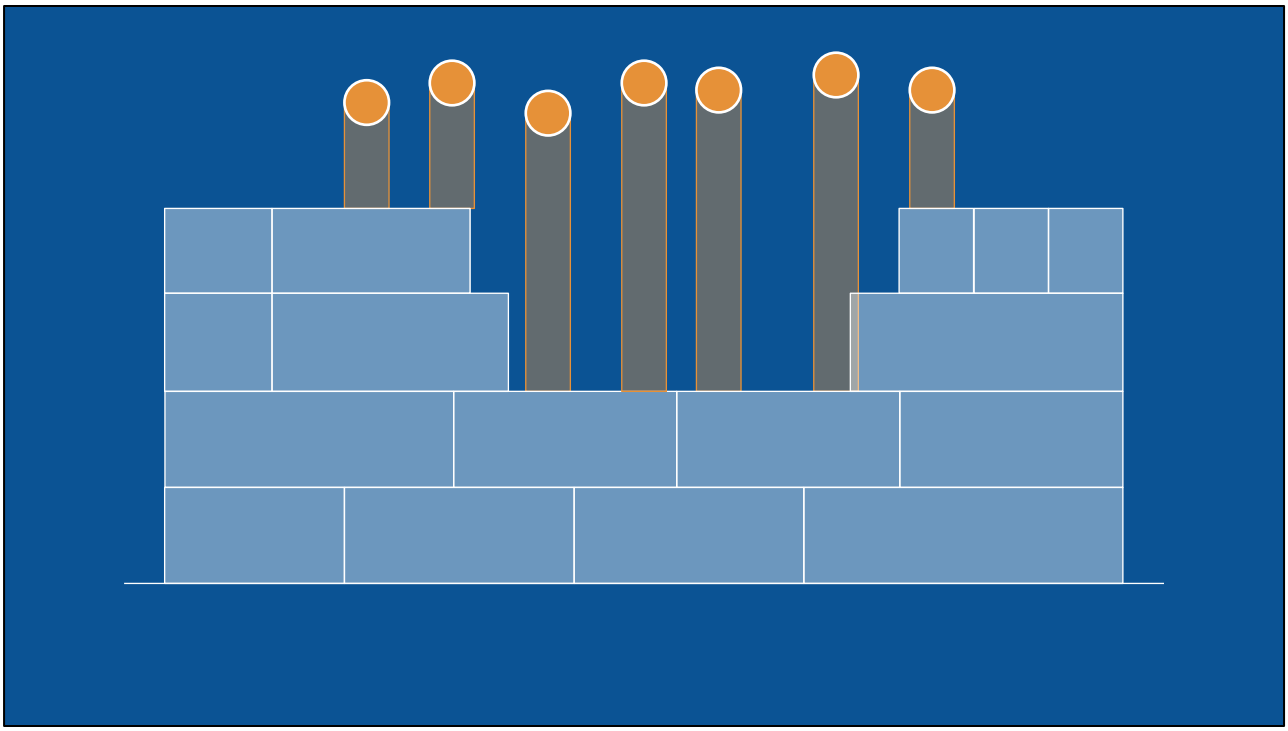
Different software teams thrive at different layers. Teams at different layers will be mutually incomprehensible to one another. "You're going so slow that we'll get left behind!" vs "You're being reckless! You're creating a huge mess to clean up later!"

So that gives us the last major property of ideal platforms: pace layers.

Before we move on, we need to hit one more thing. We have to talk about 3P layers of the platform.

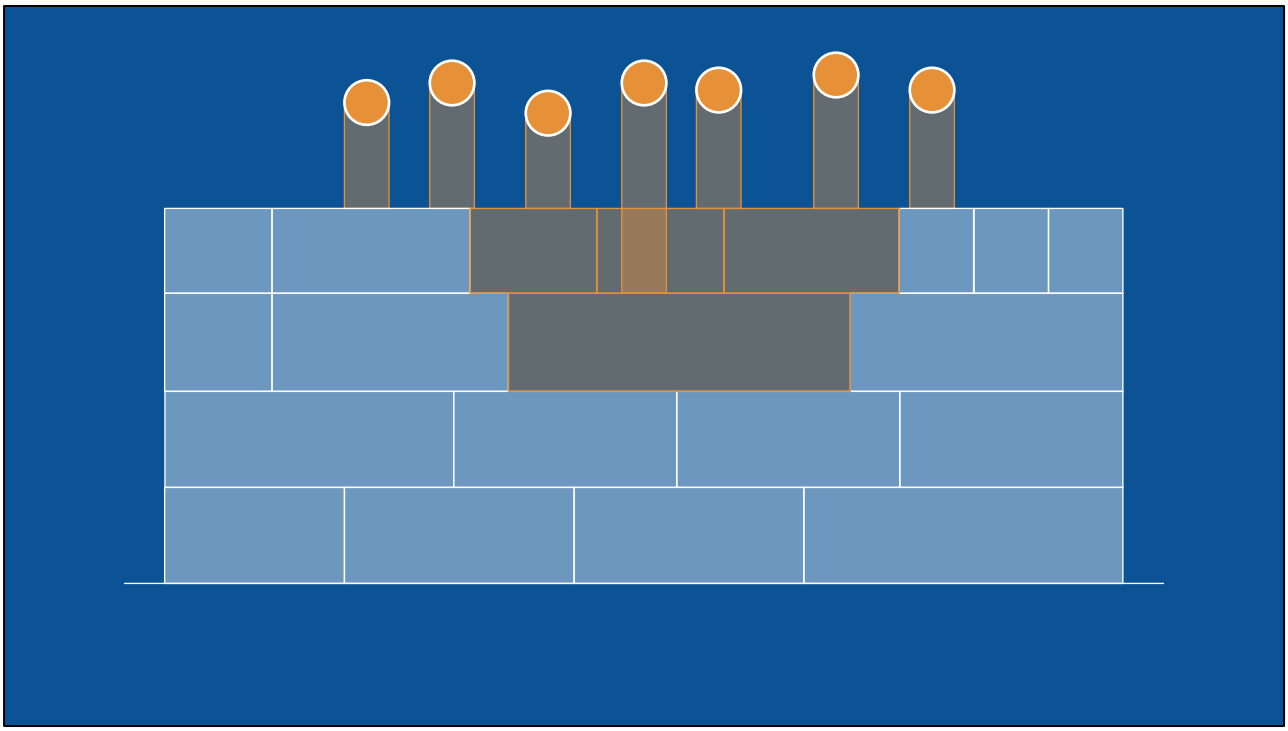


So far we've talked about a platform entirely built by us, the 1P.



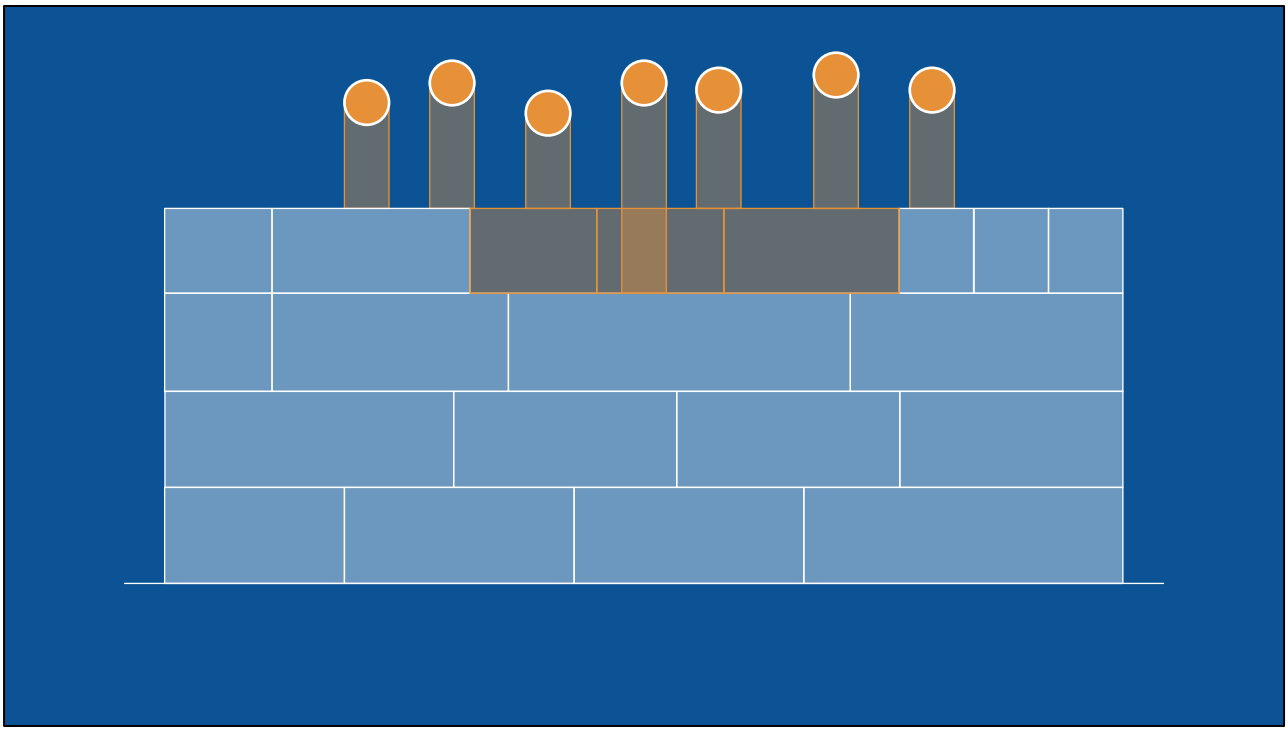
But we can’t build everything. Sometimes it takes time. Sometimes it’s not something we care to do and maintain.

That leaves a gap.

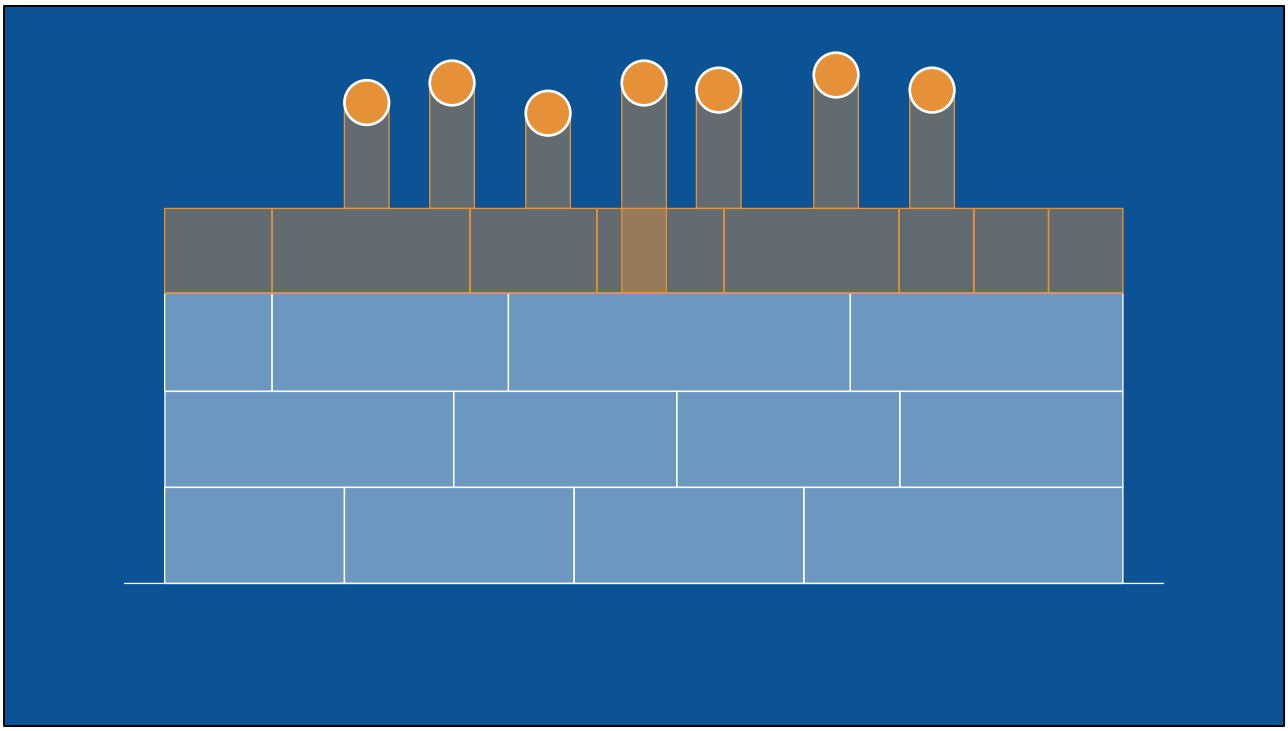


But it’s also totally possible that *someone else* builds layers on top of your platform, filling those gaps.

This can be great, and the sign of a thriving ecosystem, adding value for everyone before you have a chance to get there. Yay!



And often there’s a benefit to using the coherent set of building blocks, so once you do get there, developers will tend to use the “official way”.

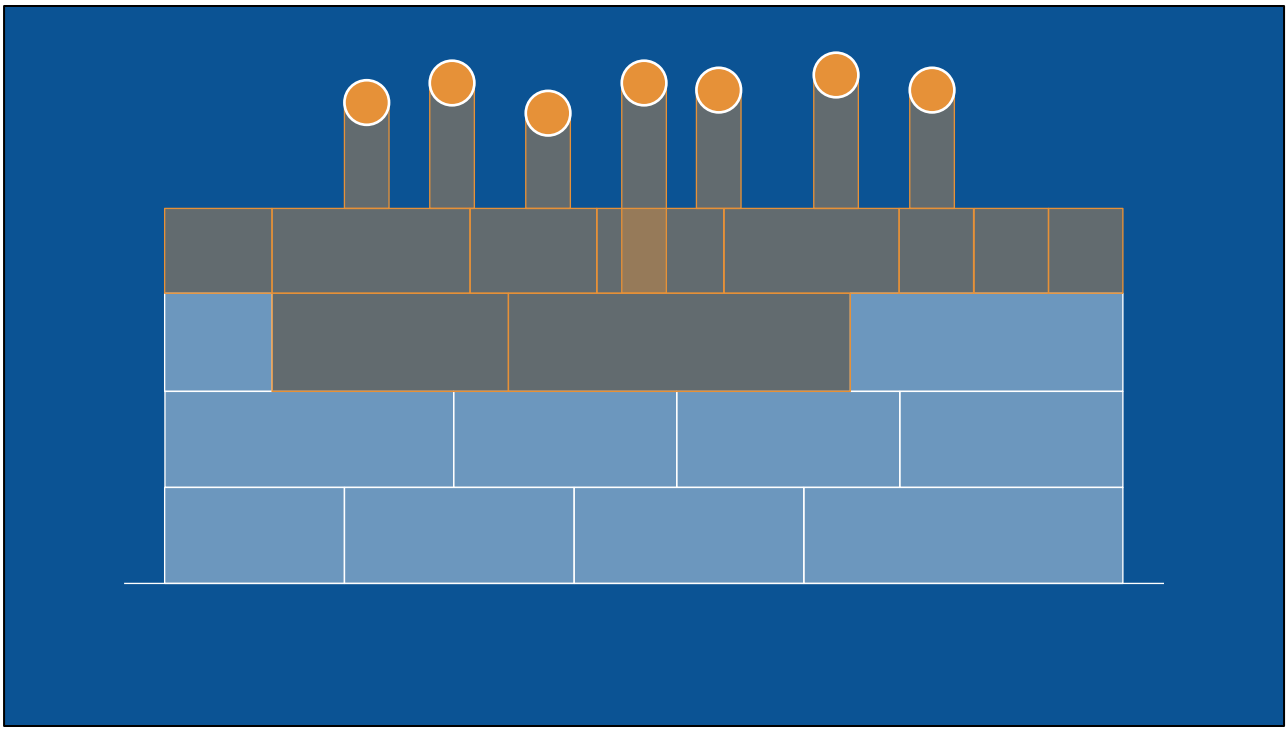


But this can evolve into a dangerous position for you as the platform provider.

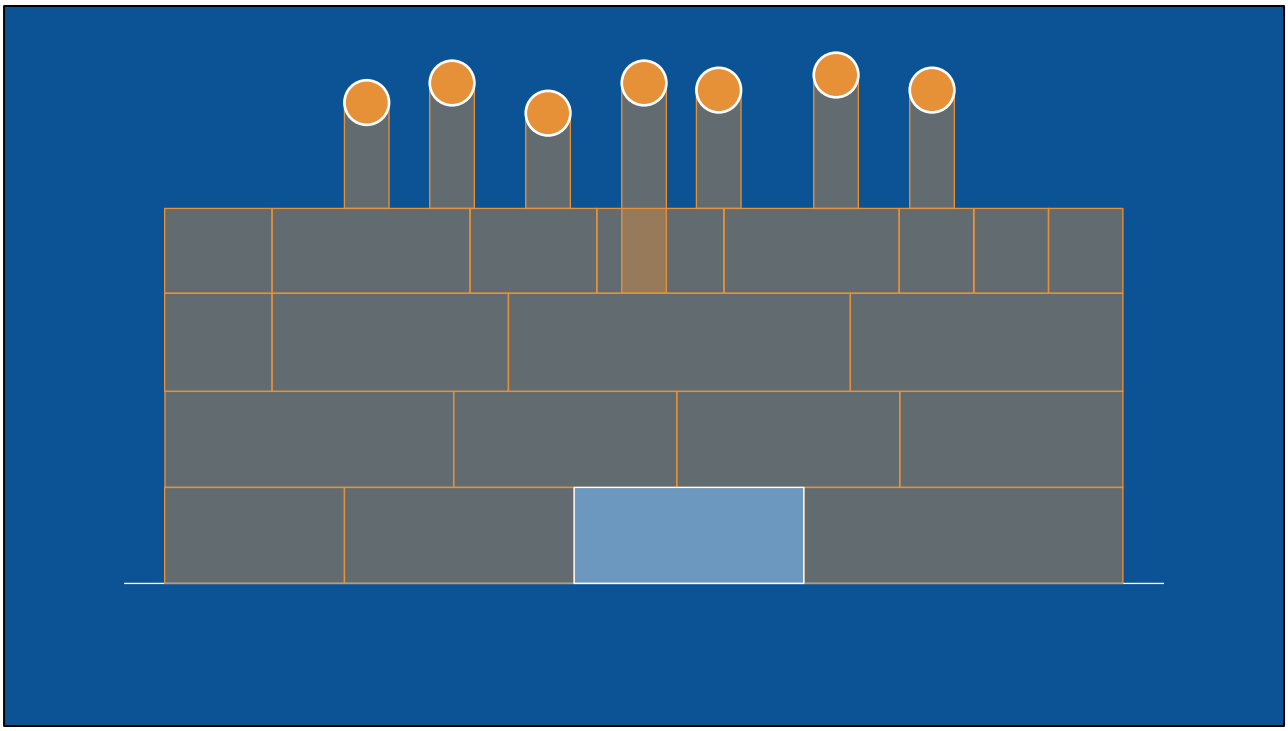
If another 3P builds layers on top of yours that cover the entirety of a layer, effectively putting a dome over your platform, they gain a large amount of leverage. They control all of your paths to the user-facing value and attention, and can squeeze you hard.

Those customers that used to be your customers are now *their* customers. And the other entity might not want to leave much profit on the table for you. If you change your platform and improve it, and the dome on top doesn't want to change, it doesn't matter.

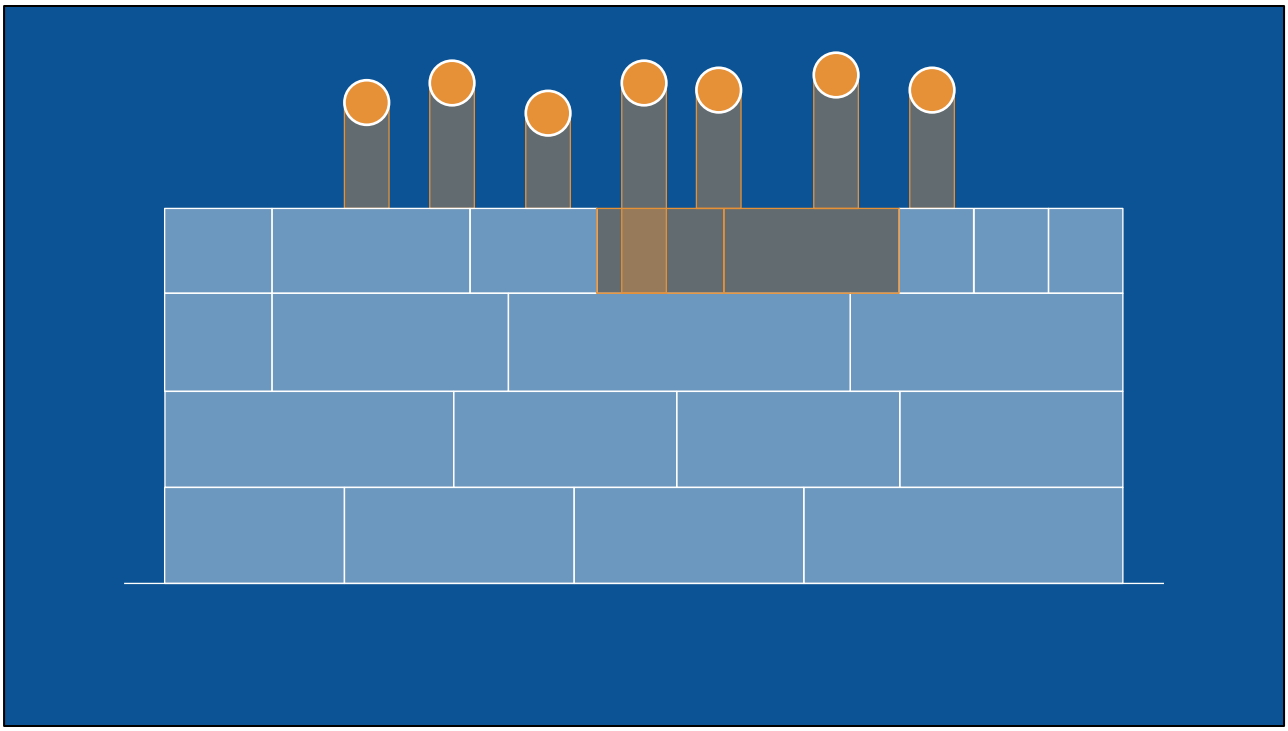
This situation is especially dire if the layer on top is an aggregator.



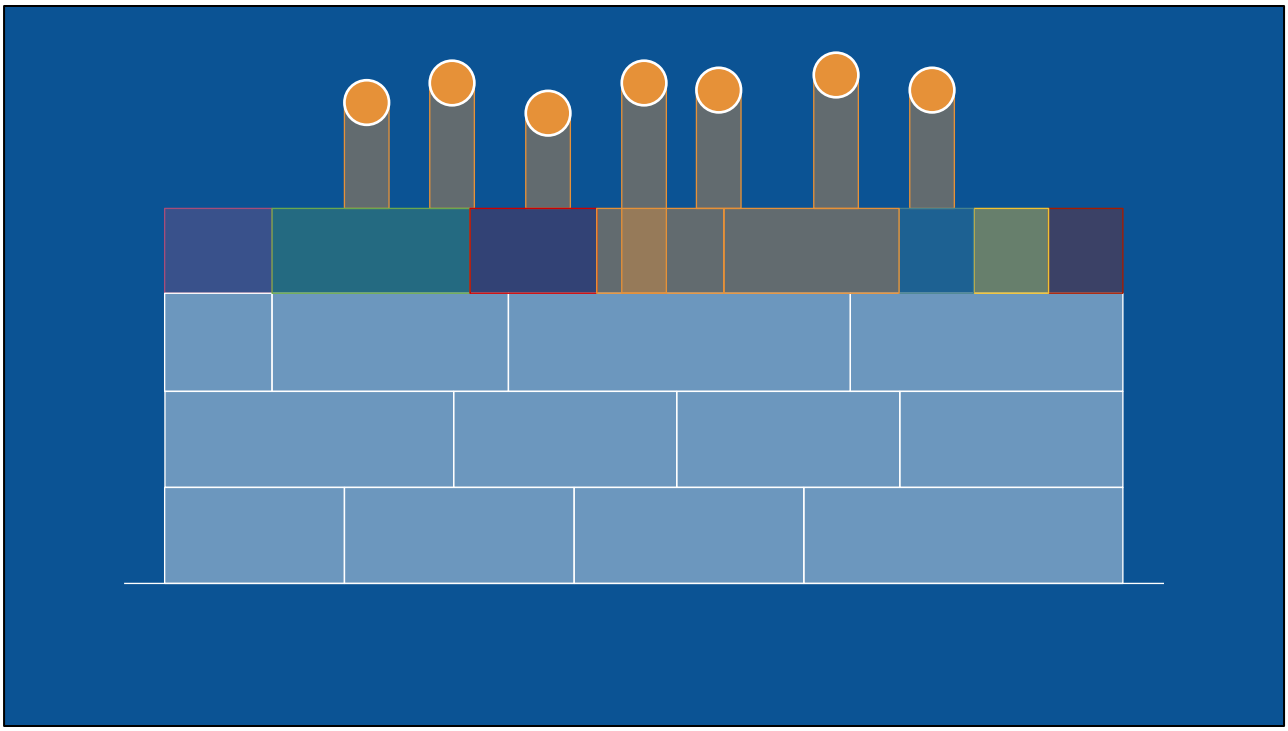
And from there, they can keep burrowing down, replacing parts of your platform...



Over time this situation can get extremely dire, and even threaten you with extinction.

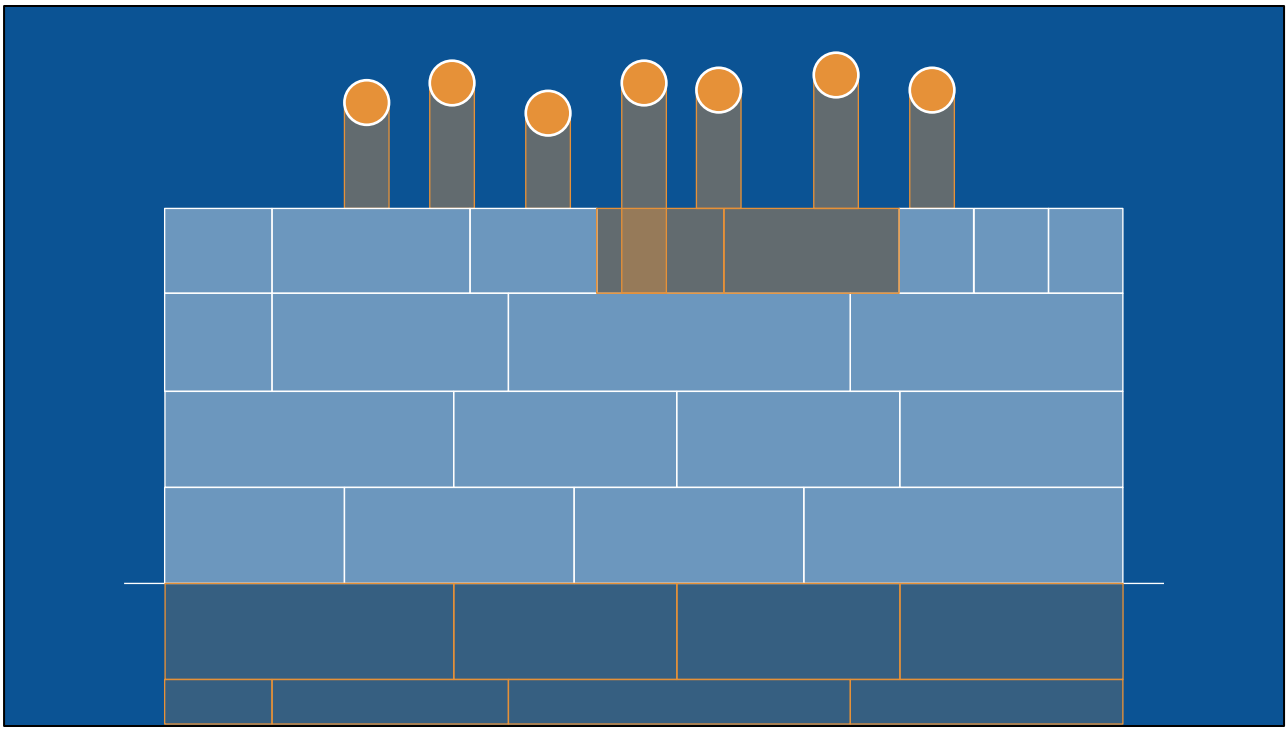


As a rule of thumb, as long as no one entity controls more than 15% or so of a layer, you can rest easy. But if it starts getting out of balance, you'll need to start taking evasive action quickly to prevent it from becoming a much bigger problem down the road.



Note that even if you're completely covered at upper layers, it isn't necessarily the end of the world.

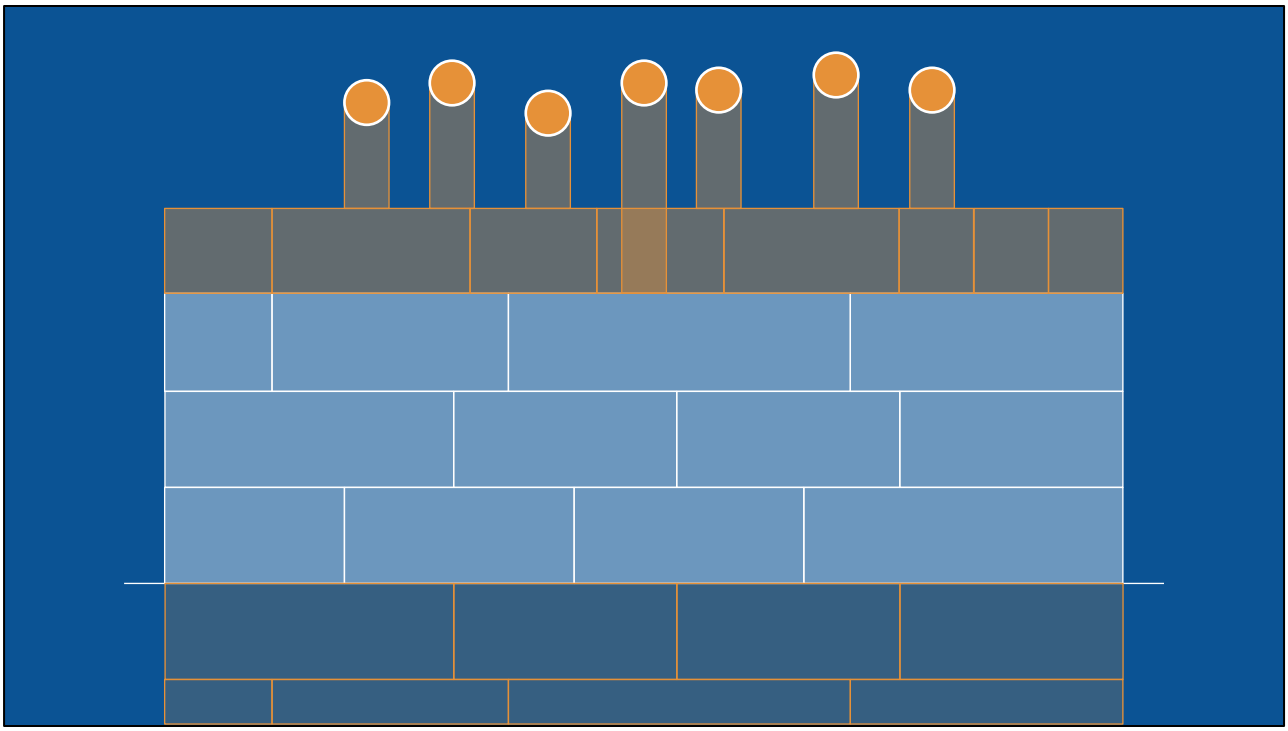
It's only bad if all of them can strongly *coordinate* their actions--which typically requires them to be owned by the same entity. If it's a lot of different players in a stable equilibrium, then no one entity can get too much leverage over you, and you can play the different ones off of each other if you have to. If you're in this situation, make sure to keep an eye on it--if one player starts getting compounding momentum, eating up all of the other players, it can rapidly get into a situation where you're fully covered by a single dome.



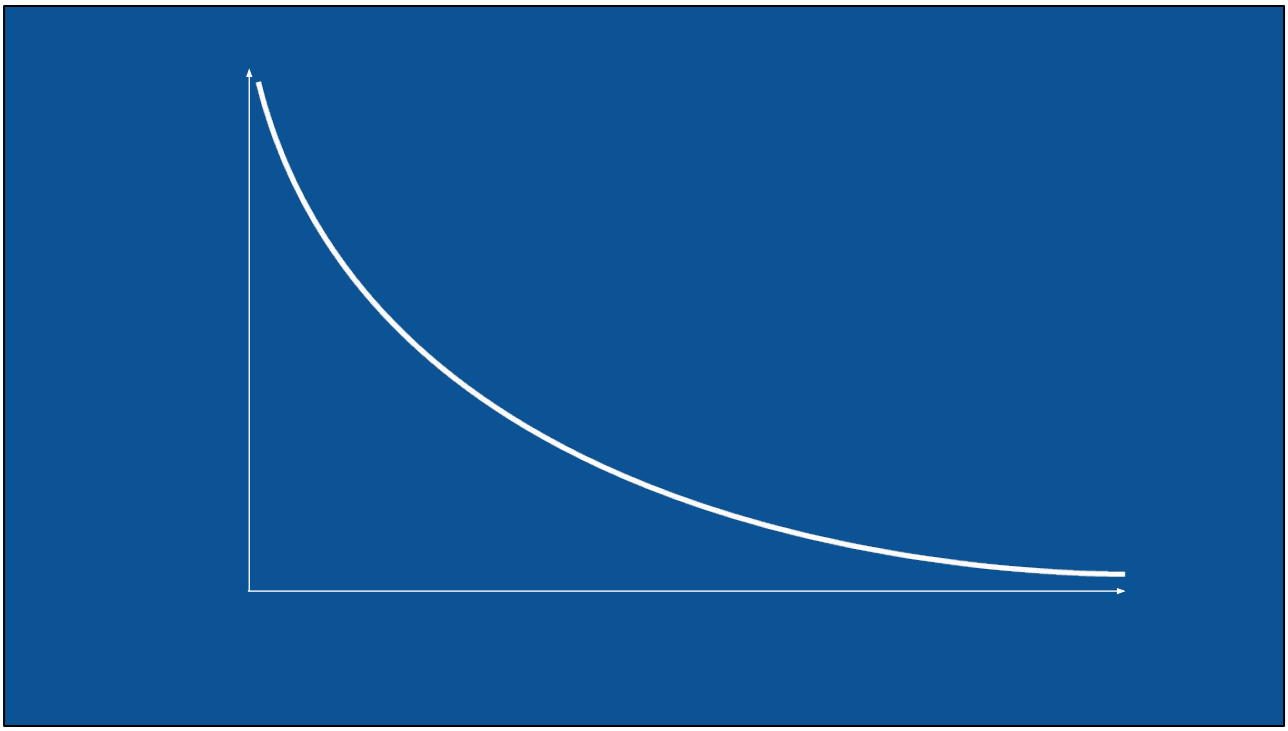
One last thing: it's platforms all the way down. What you think is your own bedrock is almost certainly built on top of someone else's platform. As a platform provider, you are both the provider of your own platform, but also almost certainly the *user* of many others' platforms.

Maybe it's the OS you rely on. Maybe it's the app store that you are distributed via. Maybe it's the literal silicon… or all of those things. Platforms above might try to constrain you--but the platforms *below* you might also try to cut you off, limit what you can do, extract more from you, go incomptent or evil, etc.

It makes sense to always have a contingency plan for what you would do if any single entity beneath you in the stack were to fall into this bad state. The more open the layers beneath you, the less you have to worry about this.



This means it’s possible to be squeezed from above (having the path to end-users be closed off) and squeezed from below as the layers below eat up into your territory. As a general rule, you want a diversity of players immediately above or immediately below you, otherwise you could get squeezed into oblivion over time.



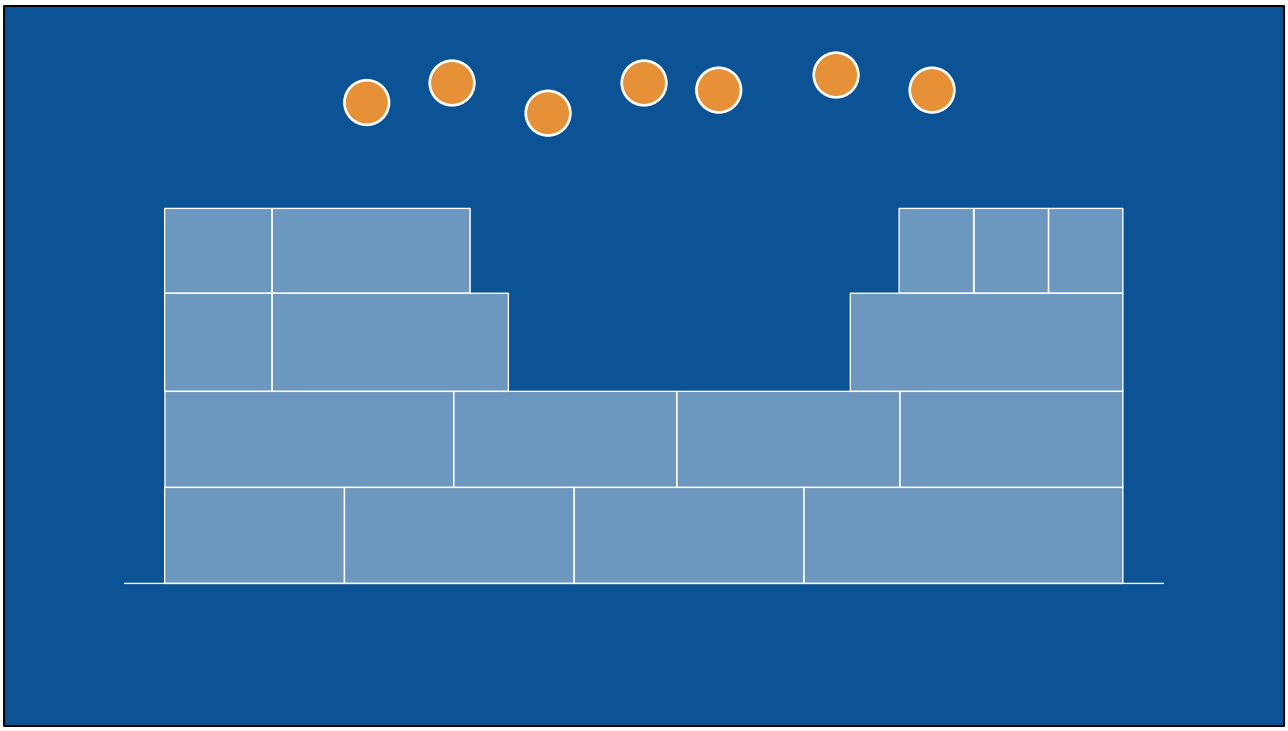
One last thing: within an ecosystem, the distribution of engagement of agents will often follow a **power law**. That is, if you sort all of your ecosystem members by how engaged they are, it will look roughly like this.

There will be a small number of hyper engaged members (maybe 1 percent), a slightly larger moderately engaged members (maybe 9 percent) and the rest only lightly engaged (maybe 90 percent). The precise distribution will have to do with the openness of the system and strength of boundaries between actors. The power law shows up thanks to inherent preferential attachment effects. We won't go deeper now on why that is, but suffice it to say this distribution arises in *every* ecosystem for fundamental reasons.

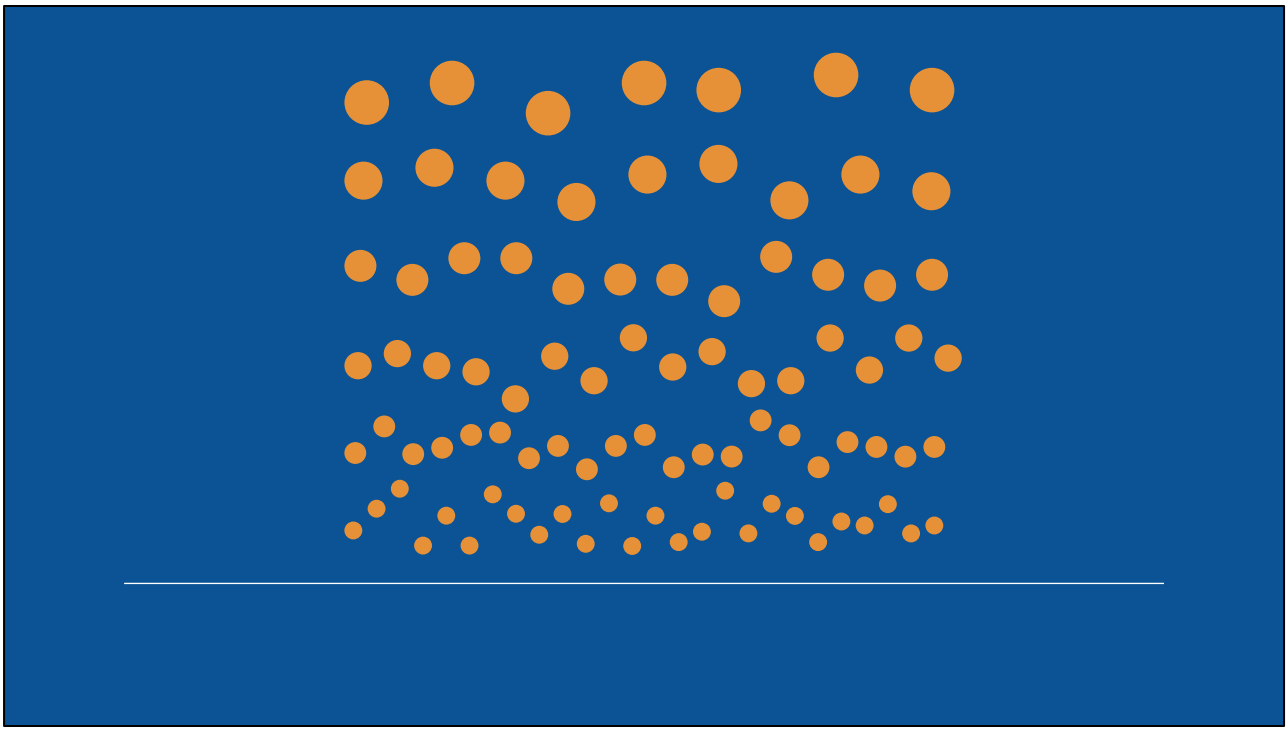
1. Ideal Platforms
2. Evolving Platforms
3. Growing Platforms

OK, that was the general dynamics of ideal platforms.

Now let’s talk about platforms as they actually exist in reality.

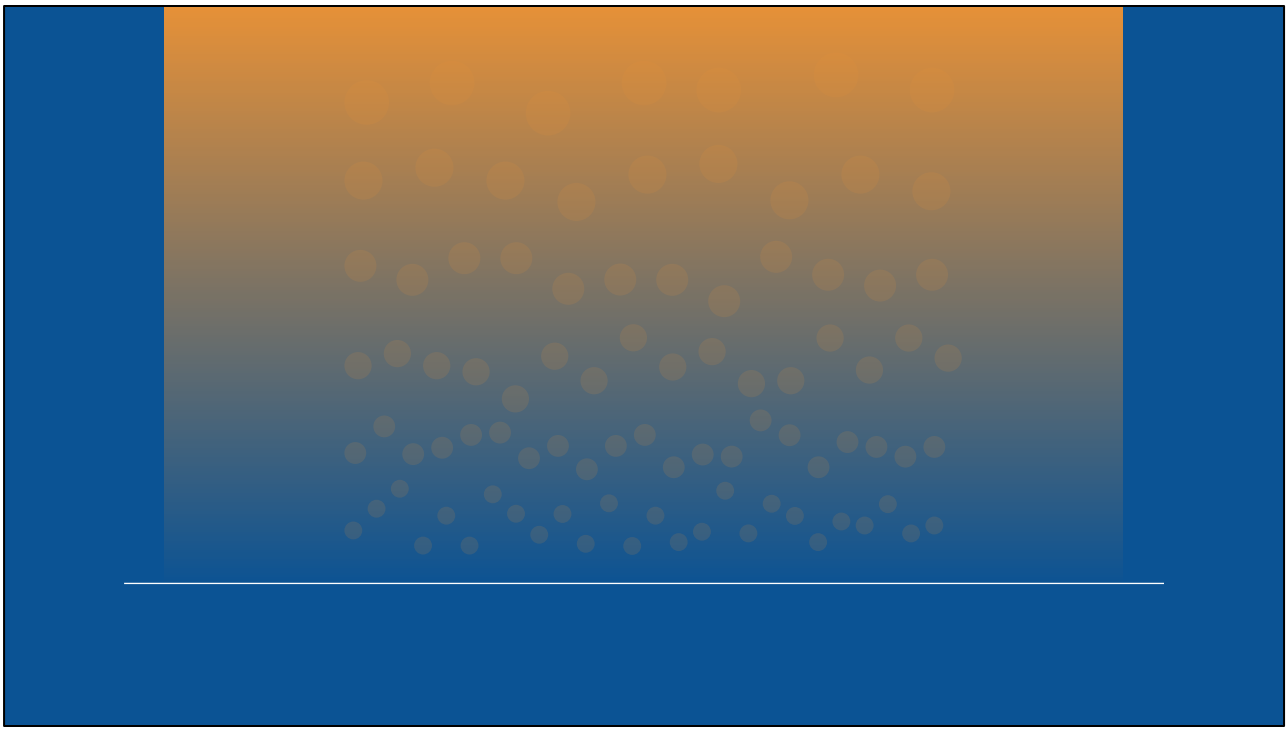


We’ve looked at our diagrams like this. With the opportunities as fixed, obvious points in space.



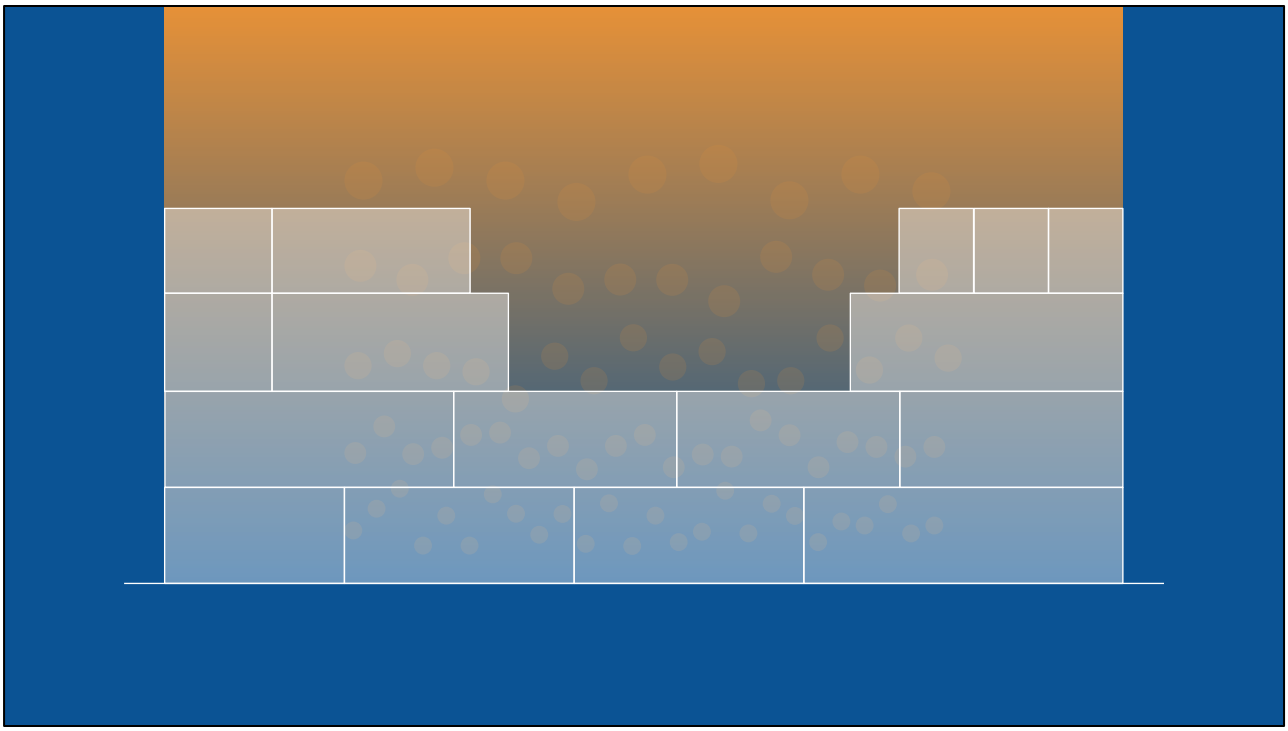
But that implies that the real world is extremely clear and precise about those opportunities.

In reality, the opportunities are a large field of possibilities. They move around constantly, blipping in and out of existence and zipping around as the ecosystem changes. You can think of them like fireflies.



And of course it’s impossible to know precisely where one actually is at any time, because the world is a chaotic, uncertain place. Imagine them being covered in a fog.

You can’t even really see the individual fireflies. All you can see is the cloud of possibility, with bigger possibility generally farther from the ground.



This cloud of possibility is the **generative potential** that goes along with your platform.

It is the **ecosystem.**

A platform without an ecosystem is like a tree falling in the woods with no one to hear it.

An ecosystem without a platform is impossible.

The two are tied, symbiotically. As one changes, the other must also, in a harmonious dance.

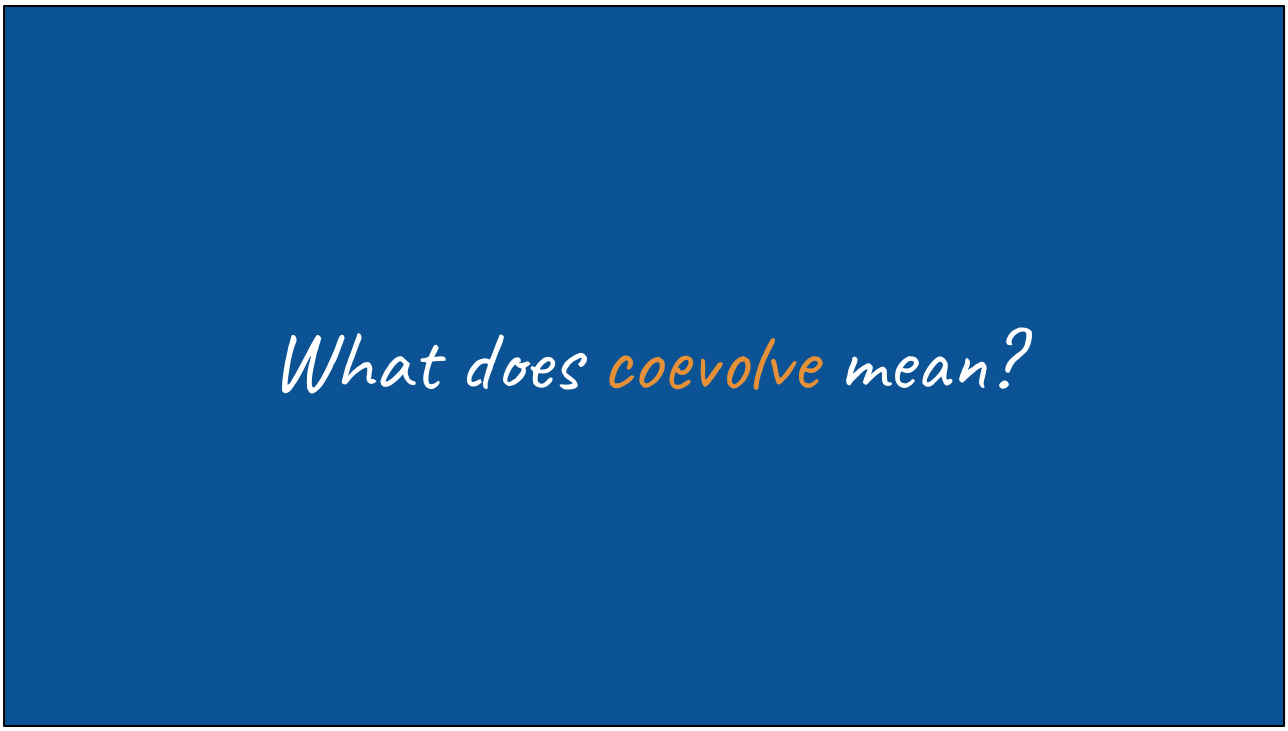
The platform and its ecosystem coevolve, responding to one another. The actions you take as a platform provider affect the ecosystem, and vice versa

Ecosystems are the sum total of all of the agents in them--the 3P developers, the end users--all making decisions based on what's good for them, reacting to their environment (and the actions of others, including the platform).

They actively search out good ideas and opportunities, under their own power.

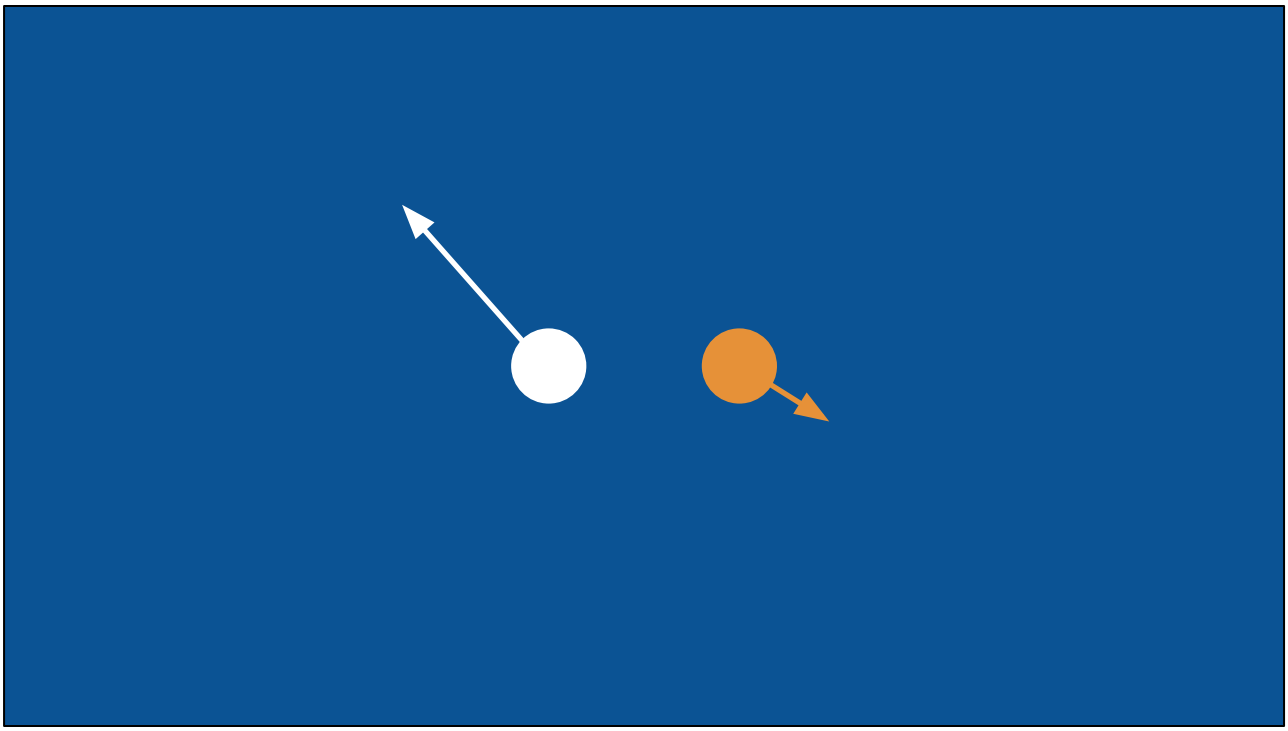
Ecosystems are alive because they are made of of lots of individual pieces that are alive.

Ecosystems are alive and ecosystems and platforms coevolve, and thus platforms must be alive, too.

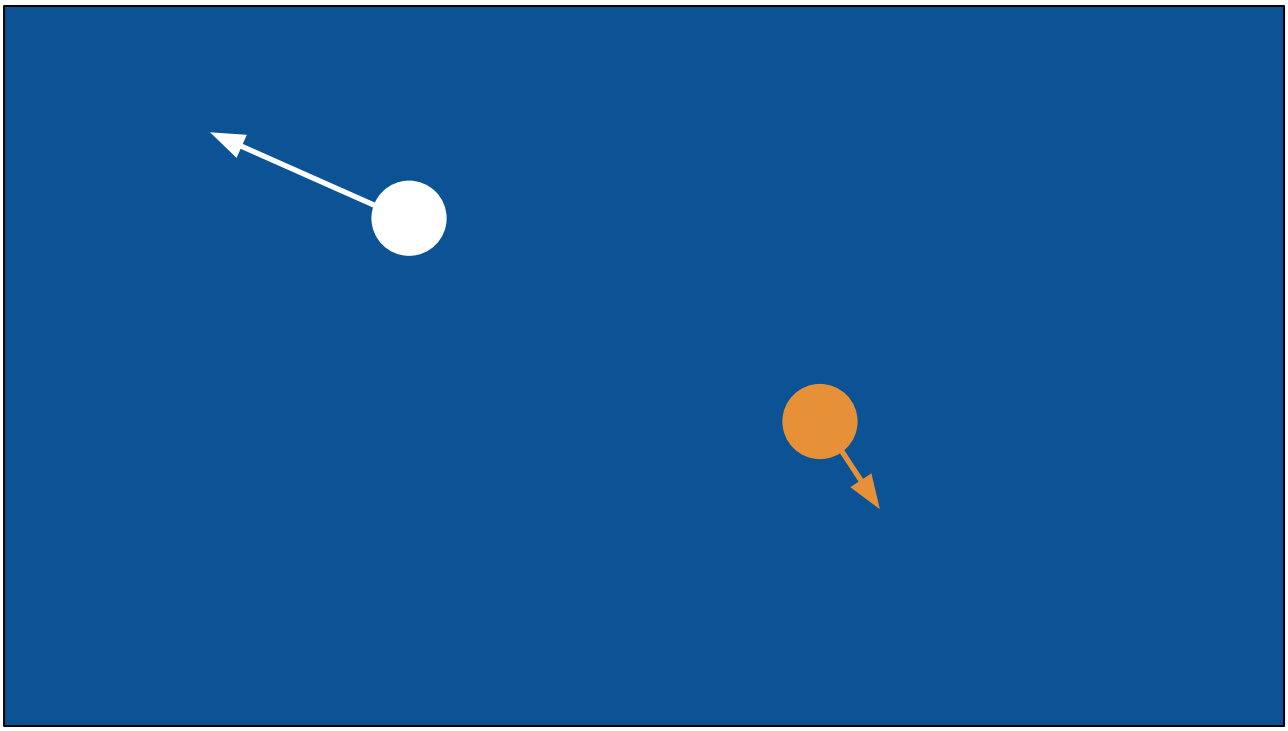


But what, precisely, does coevolve mean here?

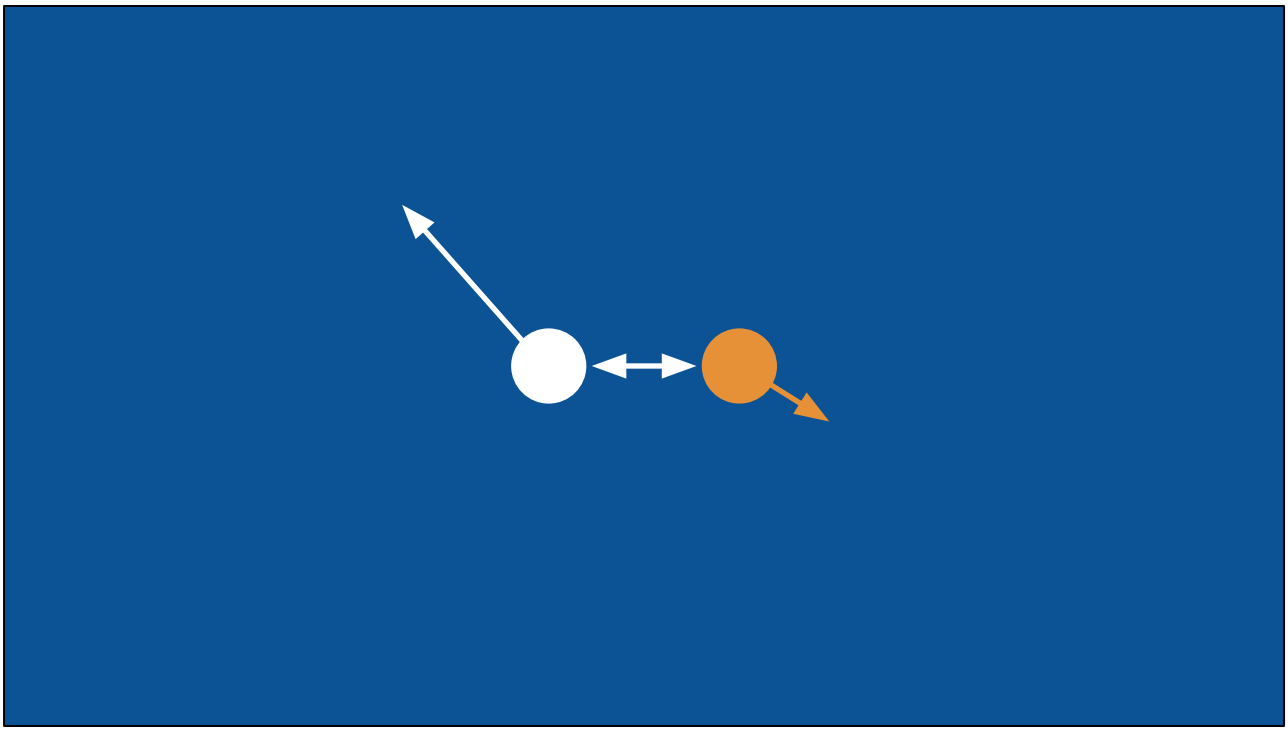
For the sake of illustration, let’s imagine the platform and the ecosystem as dots, the sum total of all of their surface area.



Each one is moving with some vector of force--the sum total of all of the different internal forces emanating from all of the individual decisions happening within them.



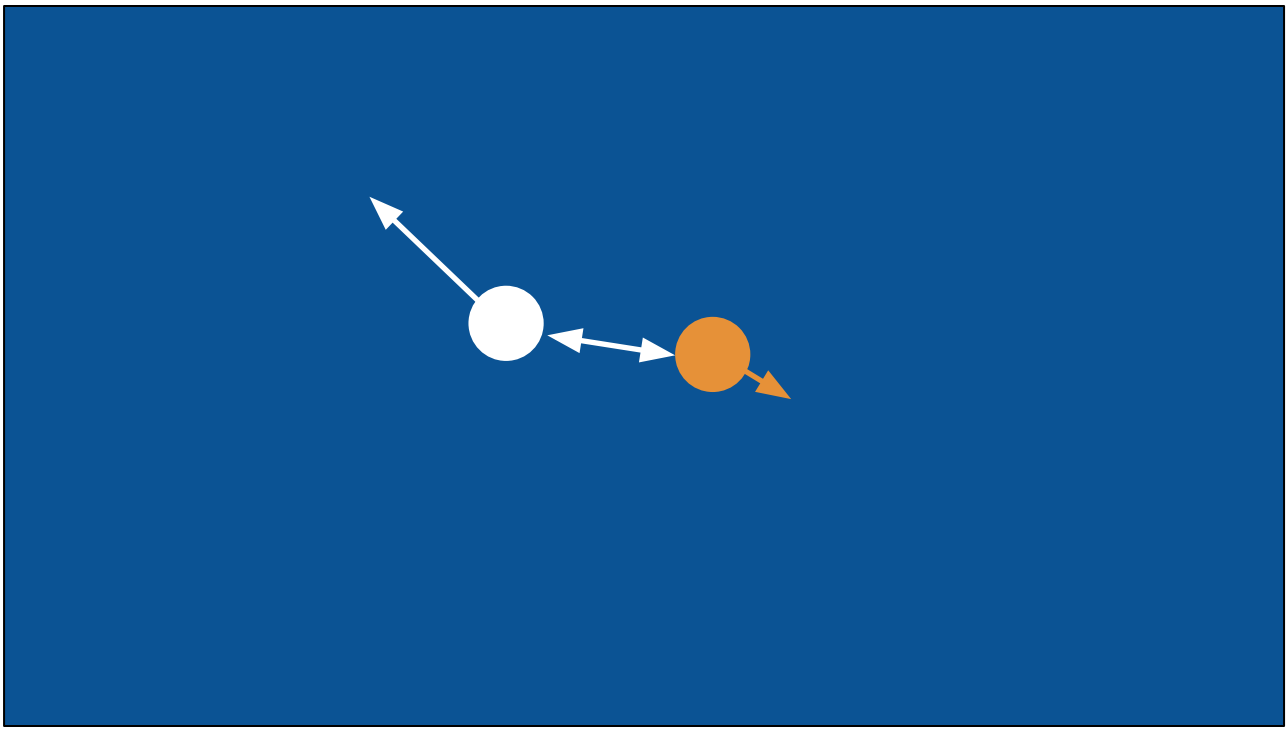
In isolation they’d just act on their own forces.



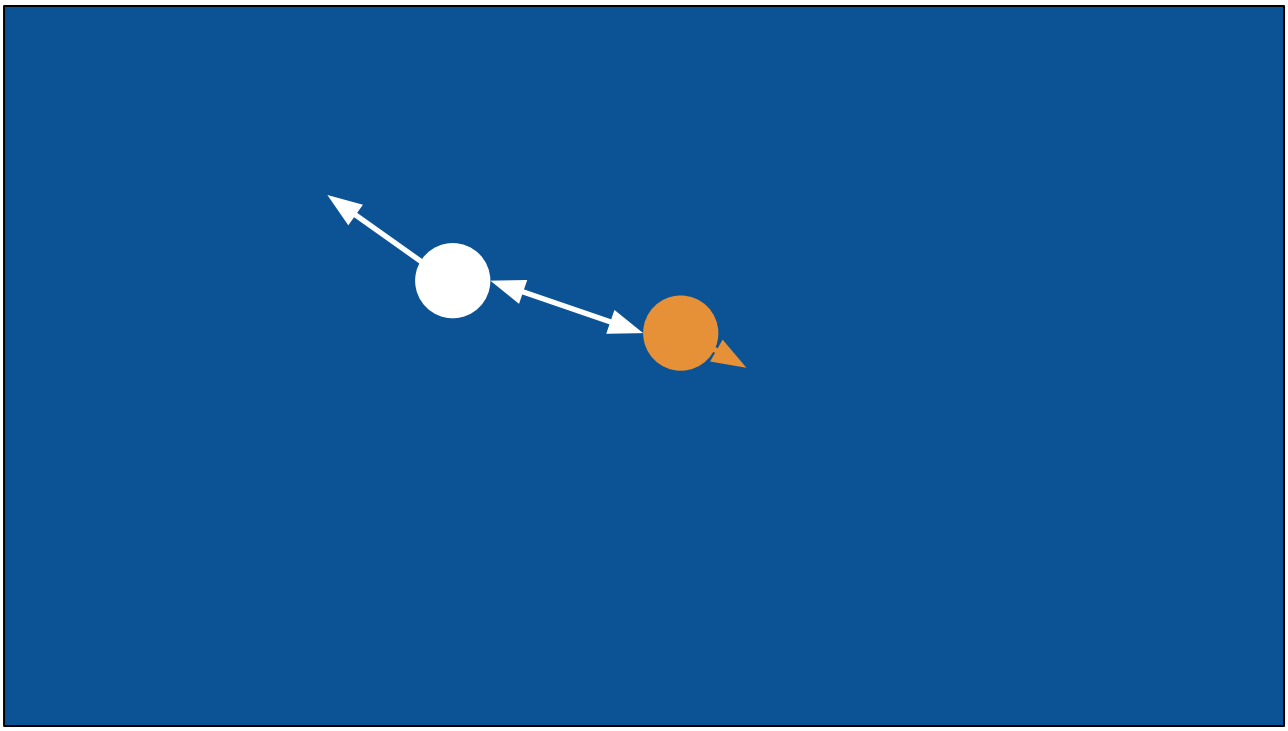
But they aren’t independent; they’re relate symbiotically.

Actions happening in one influence the other, and vice versa. Think of it as feedback loops connecting them in both directions.

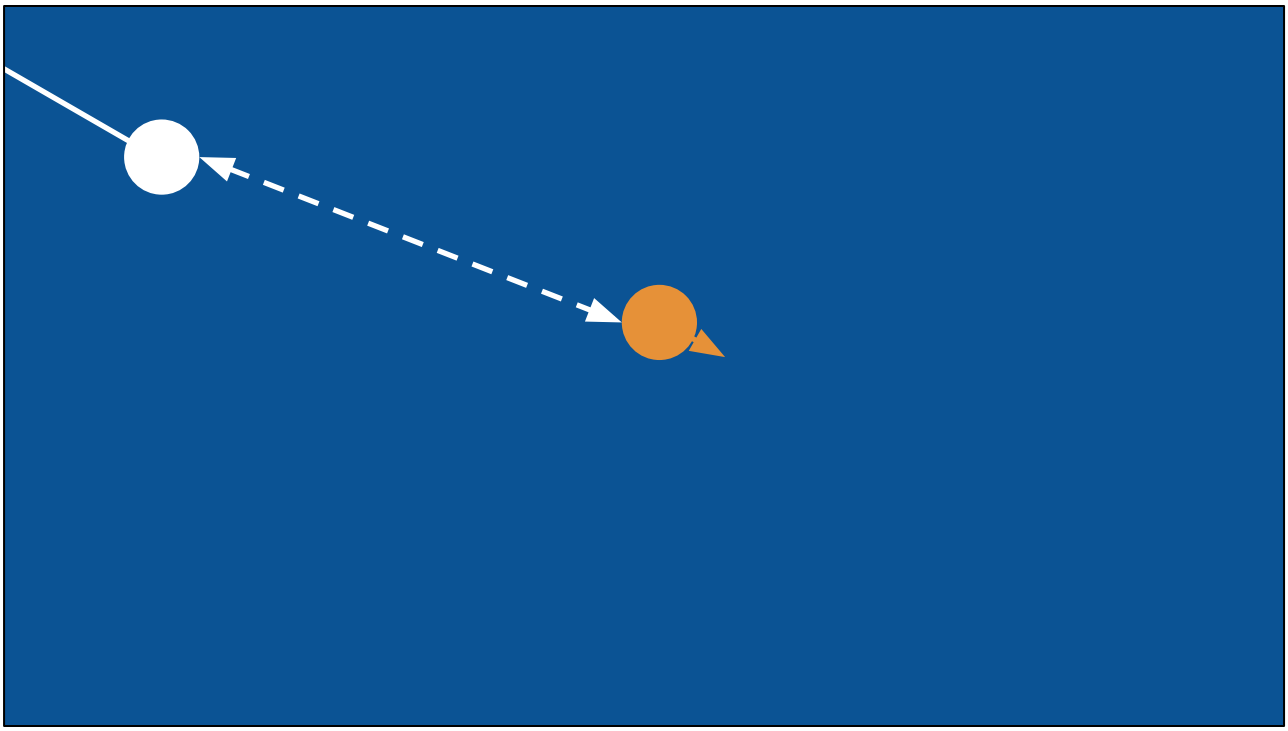
They aren’t rigidly tied together, it’s more like a rubber band linking them. It’s a **gravitational pull**.



They exert influence on each other, pulling each other along.

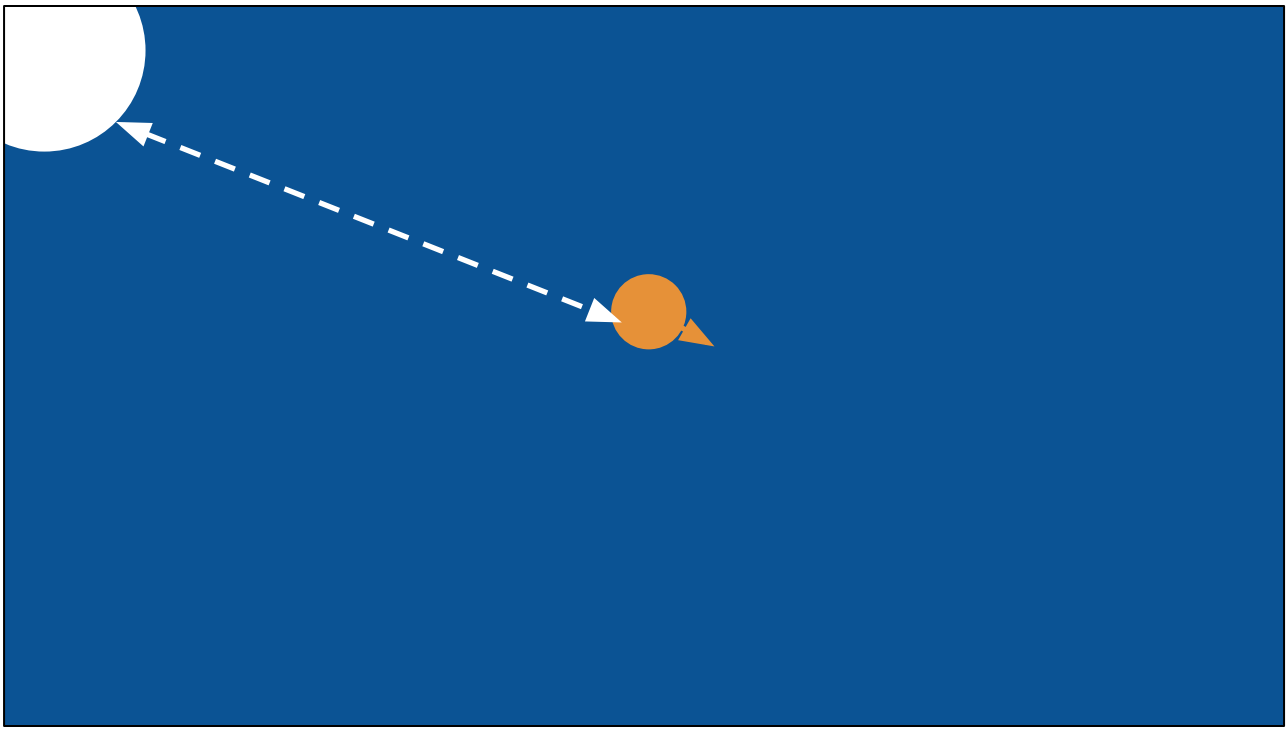


They can balance each other, and dance.

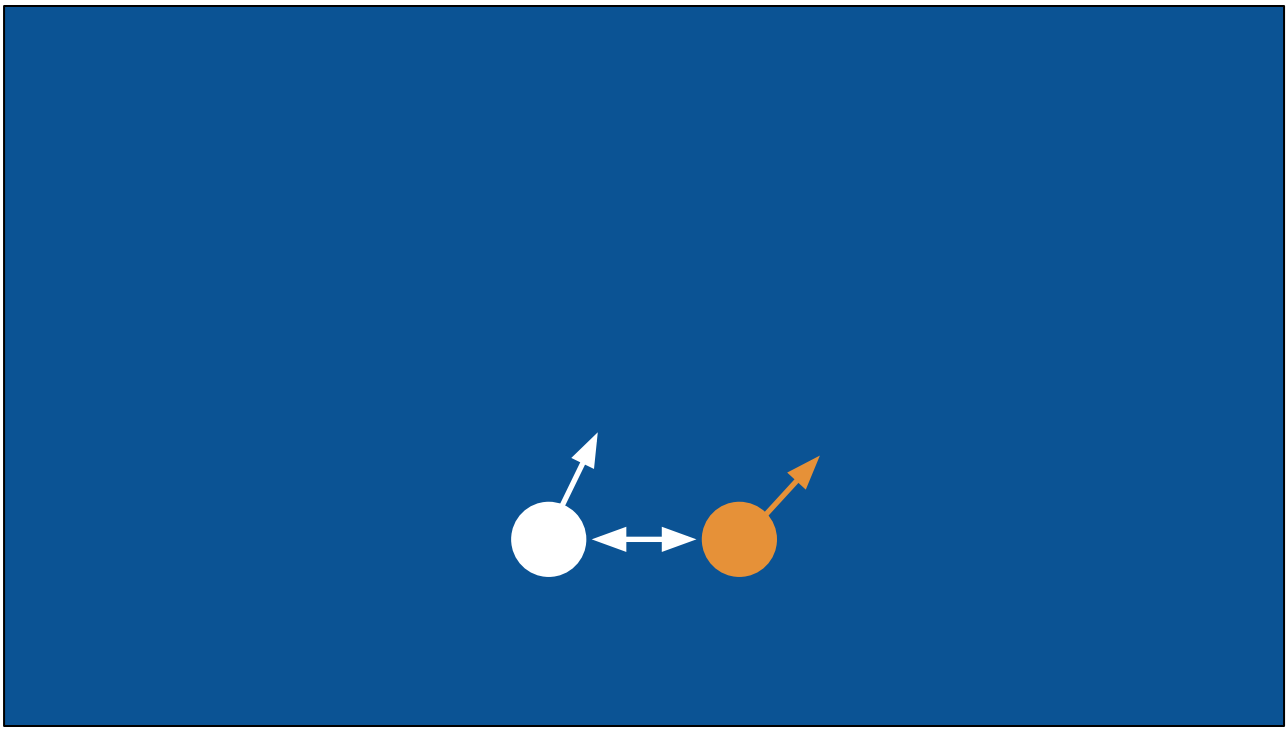


The farther away they get, the less strong the influence.

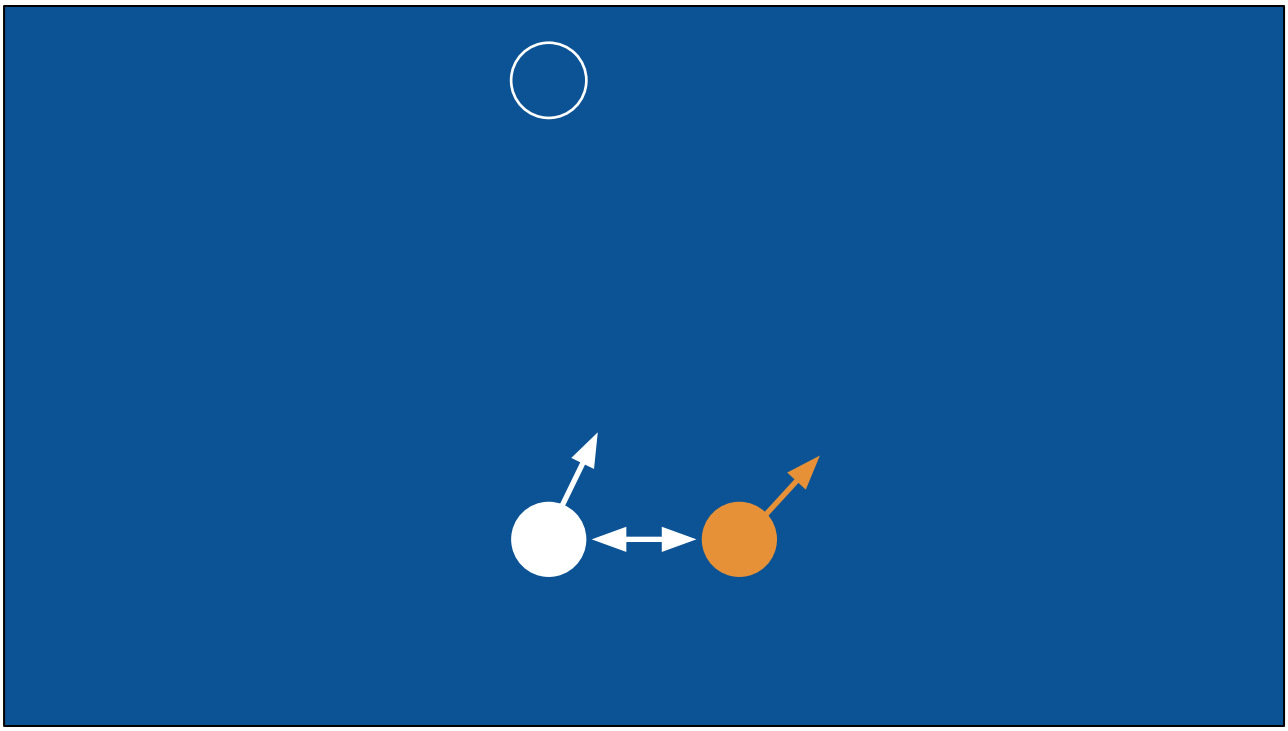
If they get too far away, the bond breaks, and they separate. The ecosystem doesn’t come along.



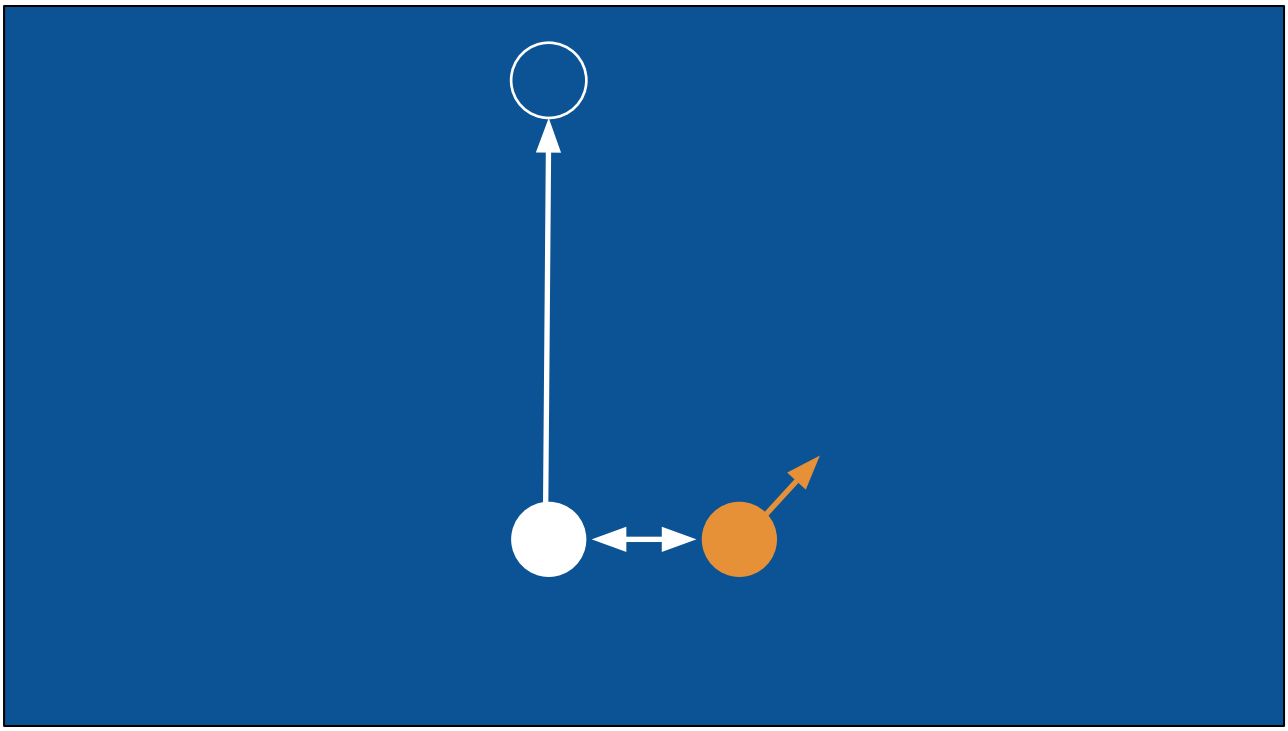
The larger the mass of the platform or the ecosystem, the farther the gravitational field will reach.



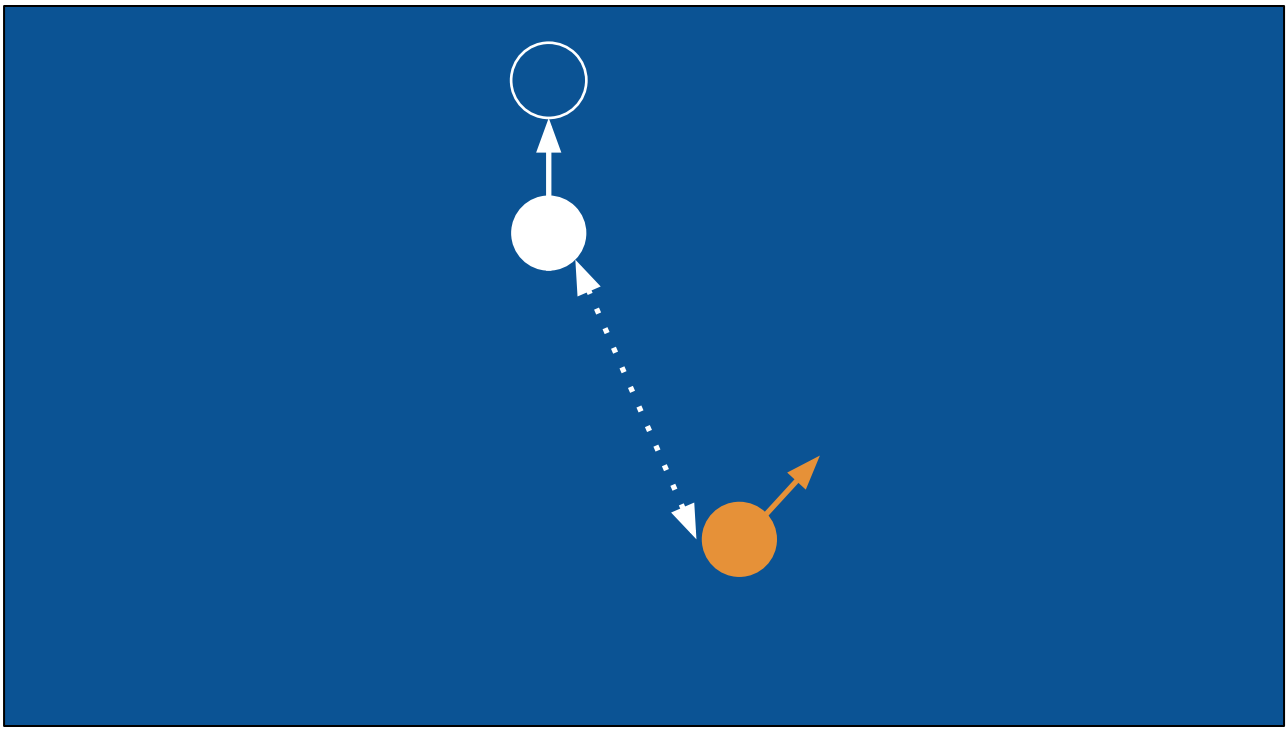
And that’s why it’s important for the two forces to nudge each other carefully, in a balanced way. It’s best when they’re working in harmony.



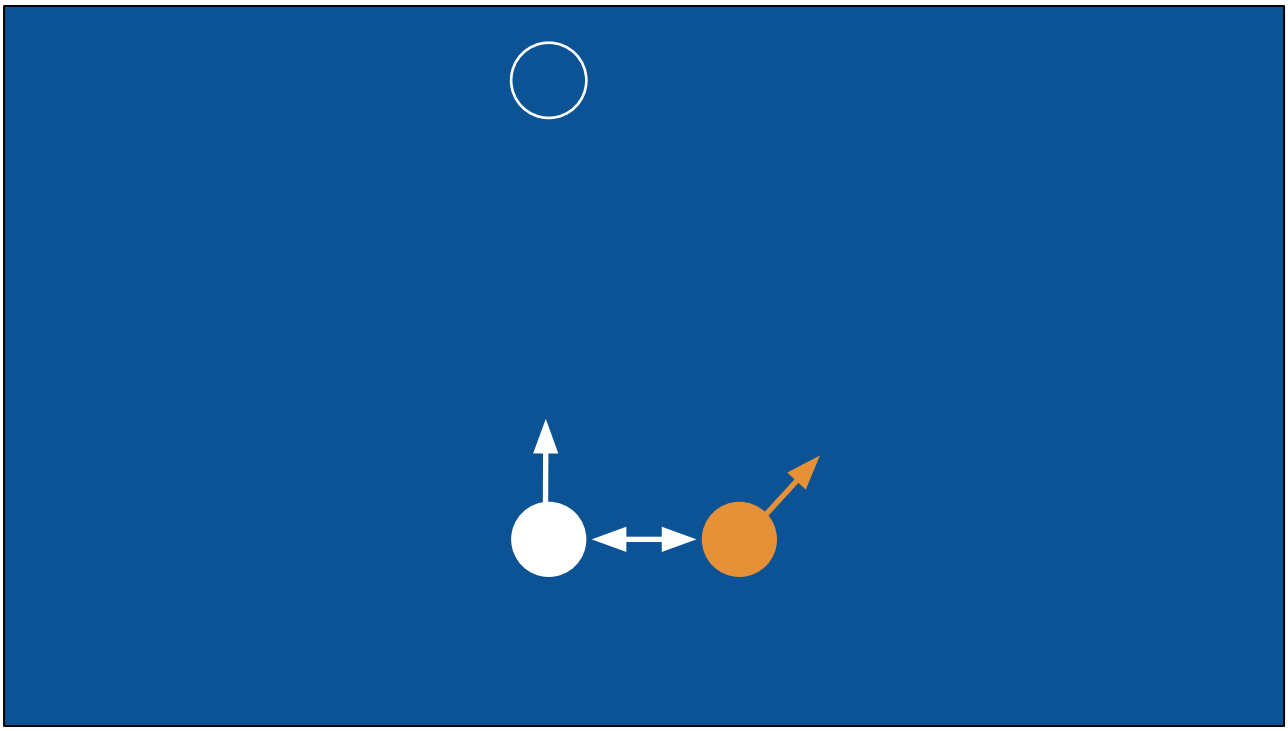
Imagine you have a place you want your platform to get to that’s far from where you currently are.



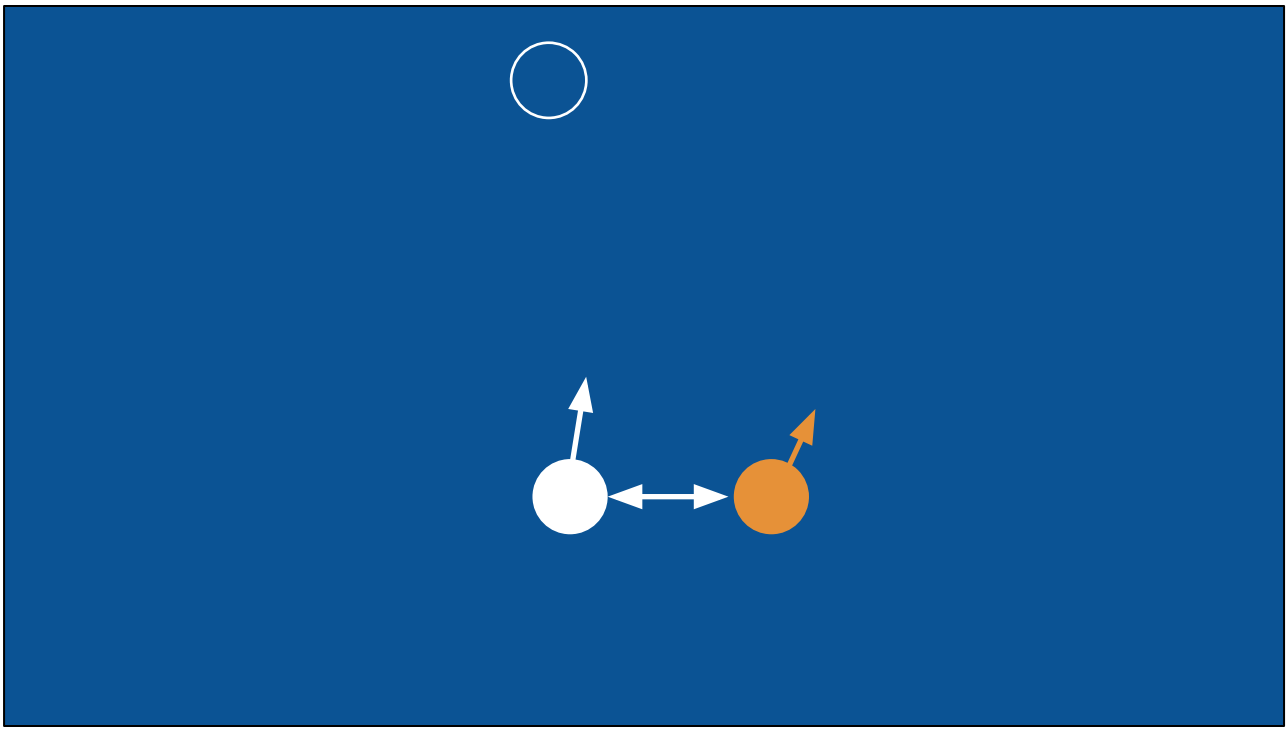
If you push too hard to get there,

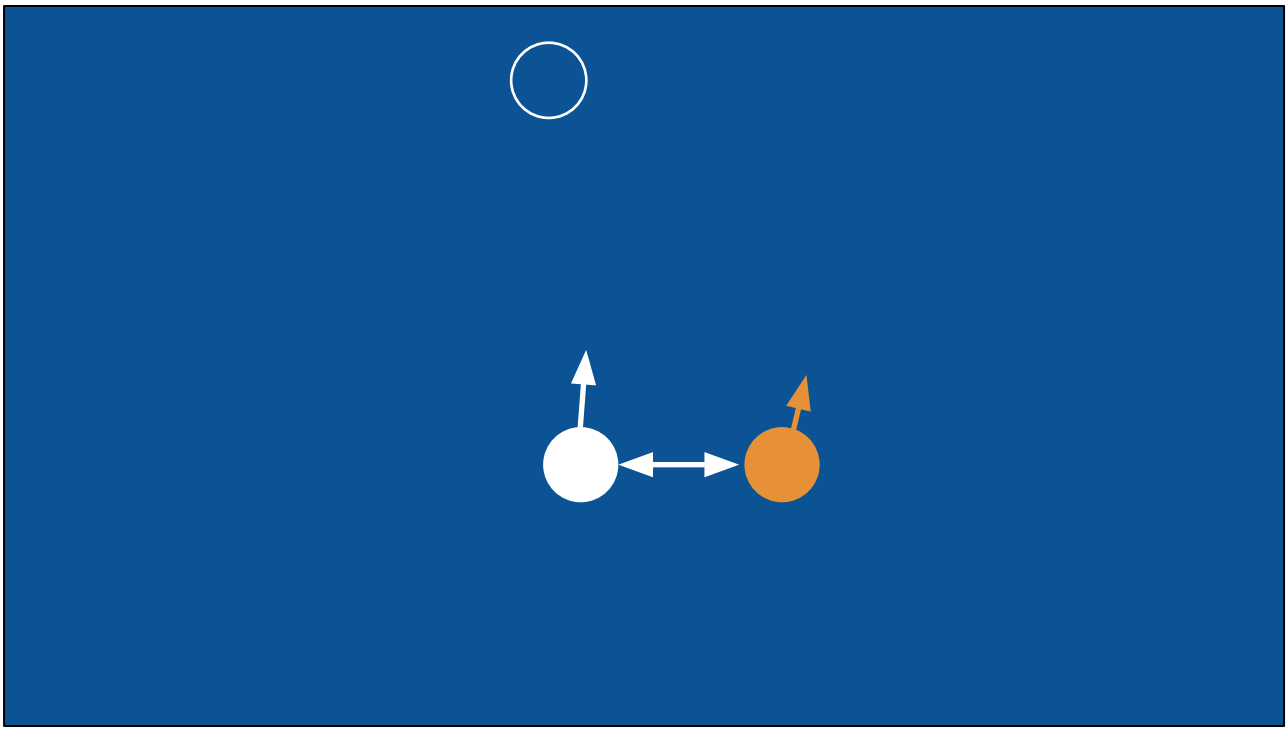


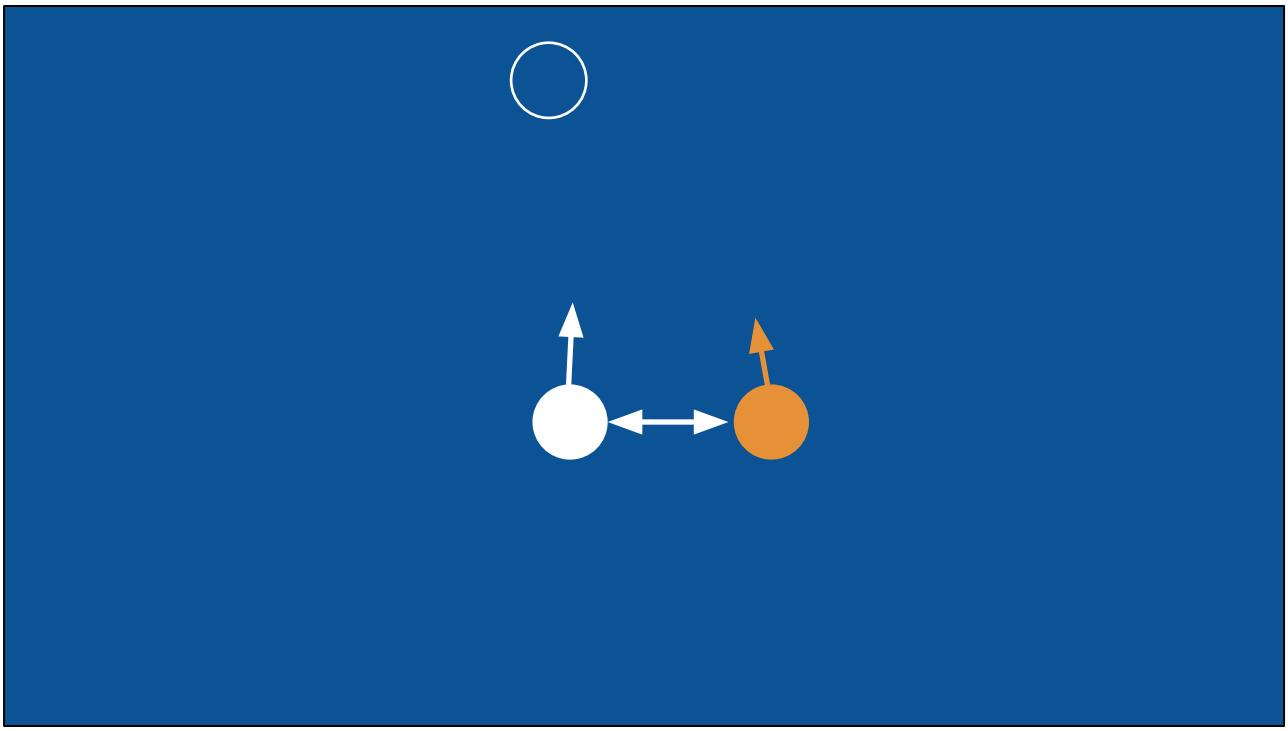
You might break the bond.



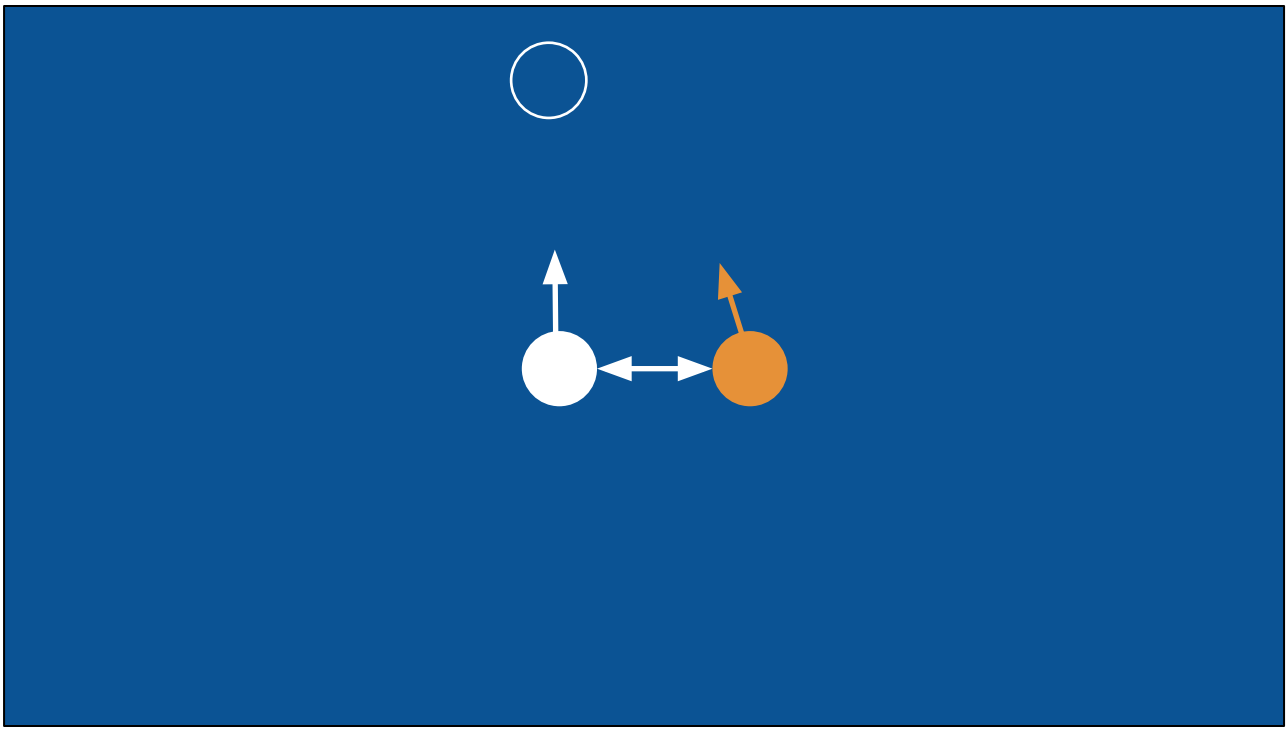
Instead, it’s best to nudge gently.







It will take a bit longer...

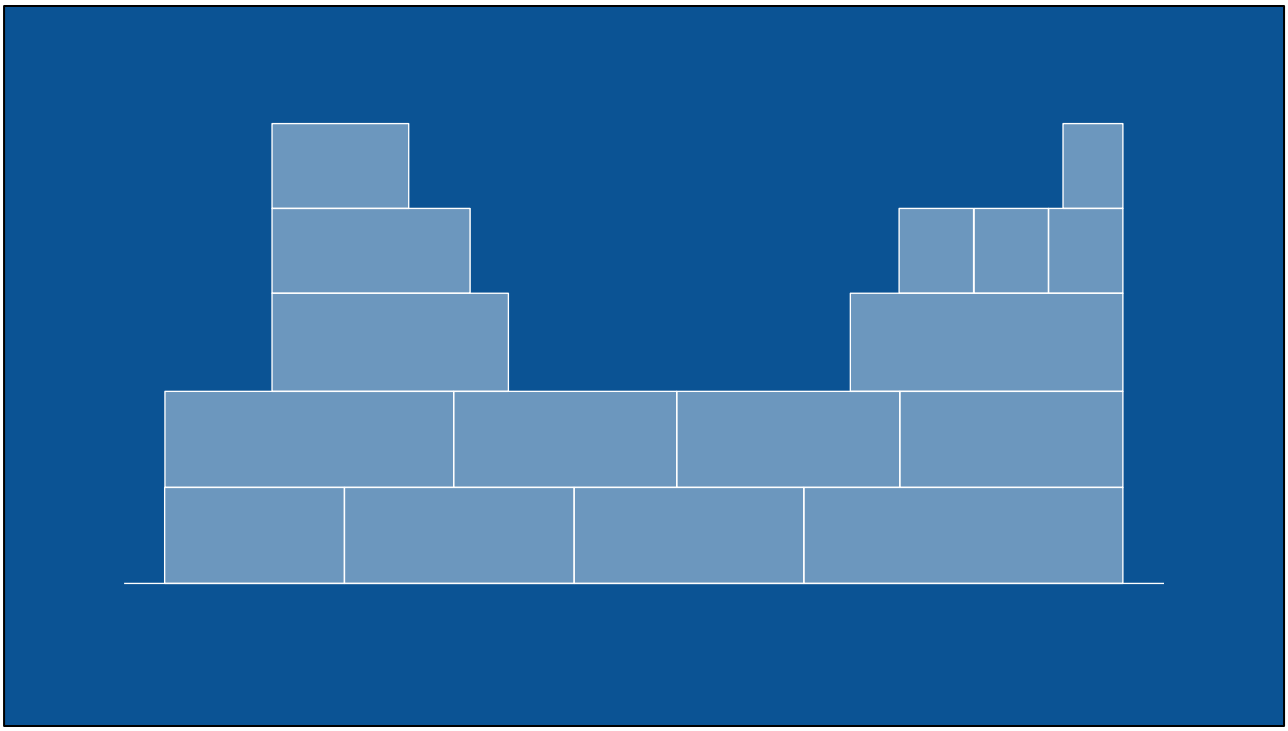


But you don’t risk breaking the bond, and over time you can generally arc things to the right place, with enough patience.

The generative potential of ecosystems--of all of those agents searching for good ideas--is extraordinary. If you have a complementary business to an ecosystem, it produces compounding results even with only linear investment from you.

It's an amazingly powerful force.

So what's the downside? Well, living things are messy. Inherently so. All of those things making independent decisions? That's the source of the generative potential--but also the source of churn and uncertainty. This cannot be "fixed"; it is a fundamental property, and precisely what gives rise to the upside in ecosystems.



We’ve looked at our platforms like this perfect ideal.

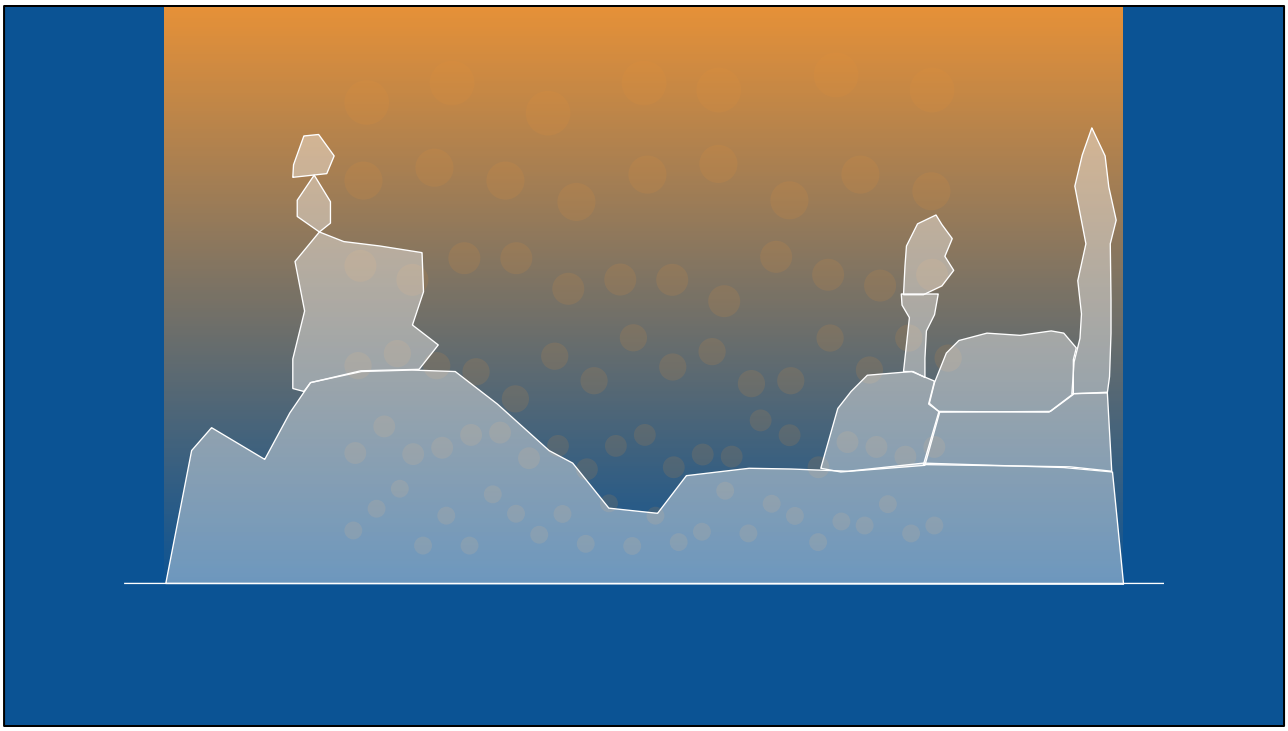
Well layered. Clean. Ideal.

No platform--well, no *widely-used* platform, anyway--in the history of the world has ever looked like this.

What does a more typical platform look like?

… like this.

Platforms and their ecosystems are living things! That means they’re messy, organic, and, in a certain way, ugly.

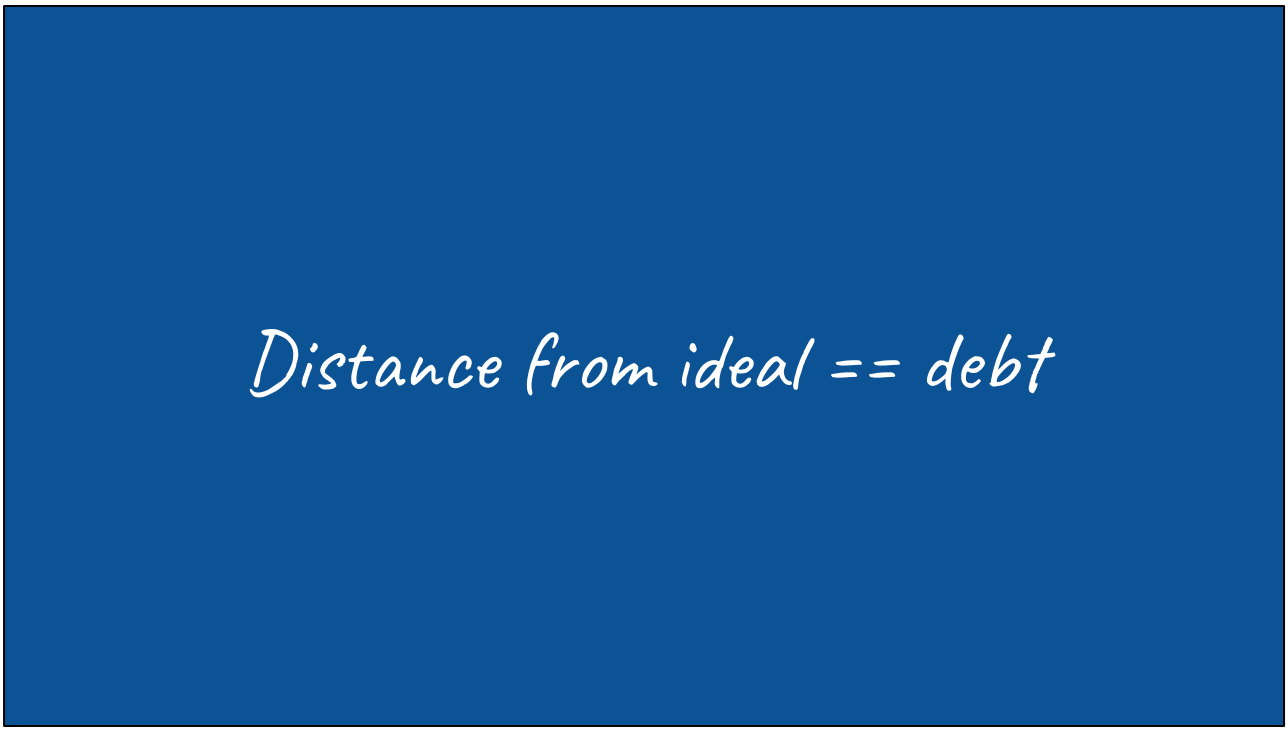


And their ecosystems are messy, so the platforms must be messy too. Constantly adapting, evolving to meet what's happening in the ecosystem. New fires. New opportunities. Swirling and churning.

The platform has to adapt to survive.

The platform and the ecosystem are in harmony.

It's beautiful… and overwhelming.

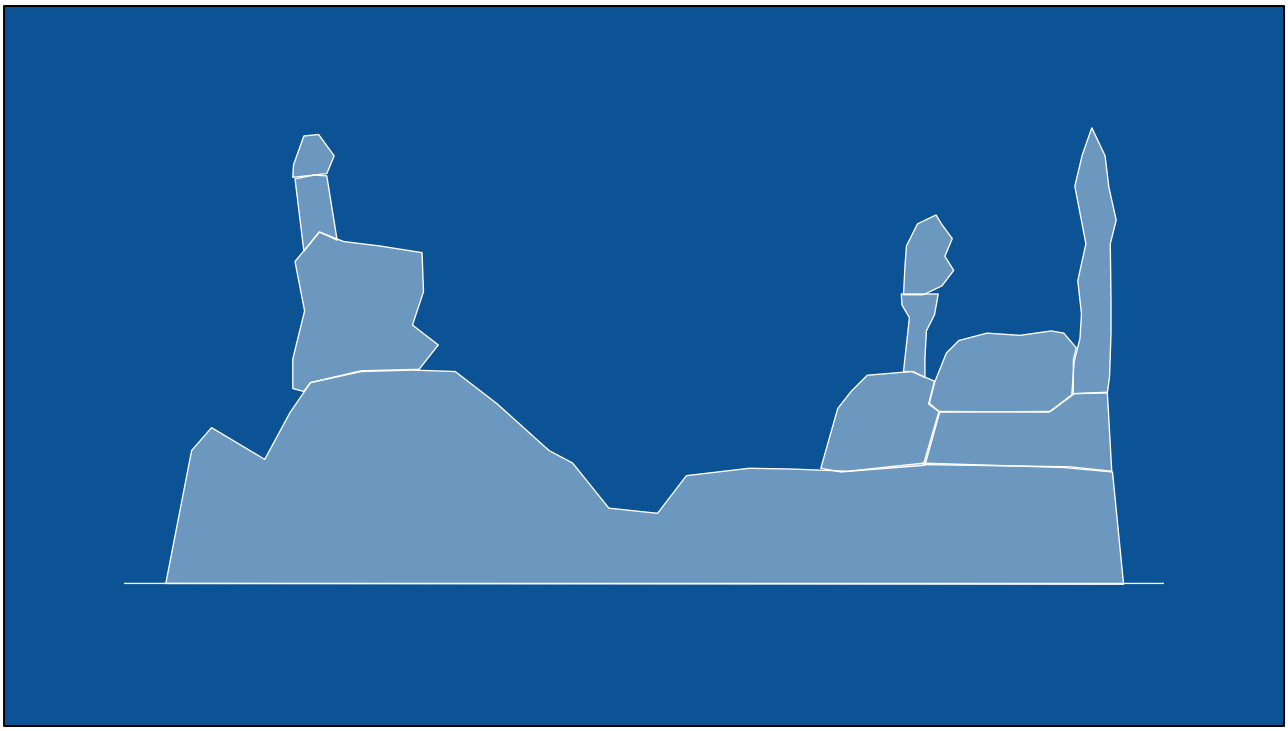


Of course, that distance from the ideal is not free. It's a form of debt that slows down everything and makes it harder to work with.

Every time someone tries to do something and finds it's more complicated than they think. The documentation is out of date, or different concepts don't fit together, or they have to navigate around a bug.

OK, so what do you do if you have a platform like this?

Some things are obvious.



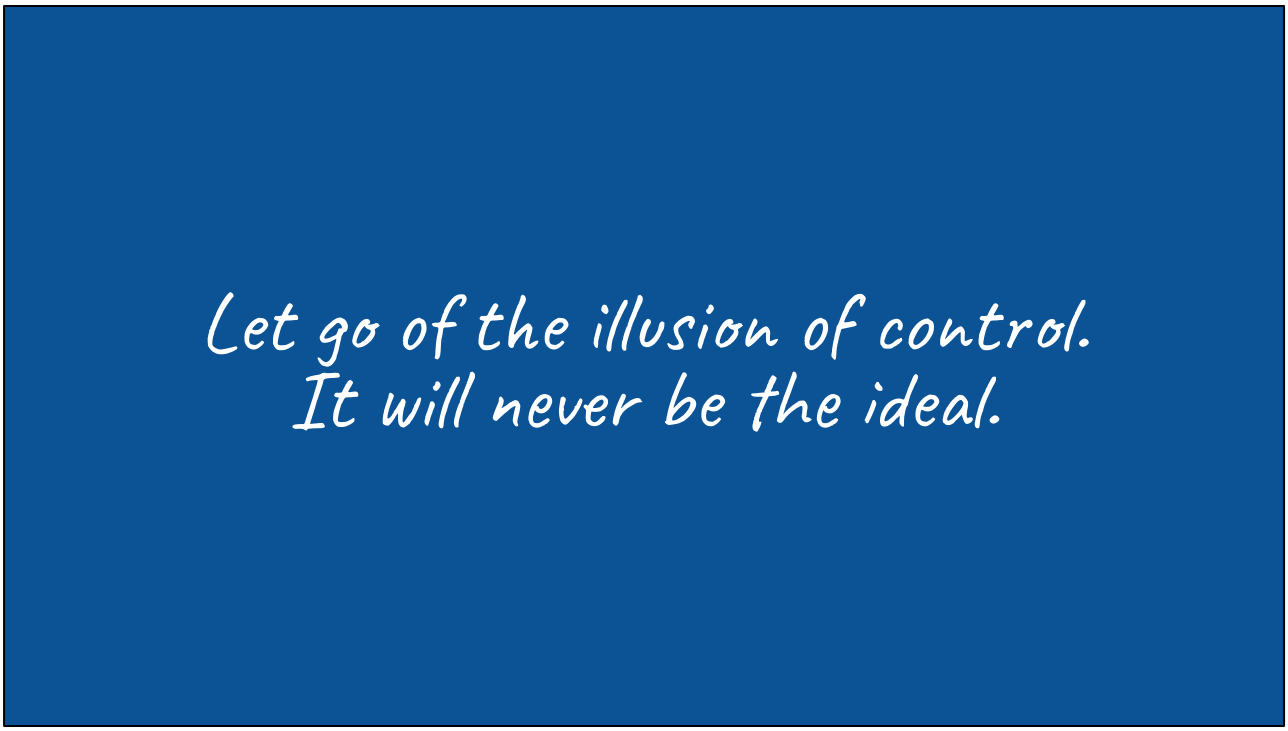
Like, maybe we should shore up that one thing that's going to fall any moment.

But what should we do now?

Remember: changing a platform's exposed semantics is really, *really* hard. You have to get everyone in the ecosystem to change, and many of them would prefer for everything to stay the way it is right now, thank you very much. And others in the ecosystem might actively hate you!

And of course, your exposed semantics are larger than your formal API. *Any* exposed semantic, intentionally or unintentionally, might be relied on by someone in your ecosystem. (This is known as Hyrum's law)

As a general rule of thumb, removing or breaking an API is at *least* 10x harder than adding an API.



To do this, you have to learn to let go of the idea that it will ever be perfect.

Control is always an illusion to some degree, but when you're responsible for a platform it's impossible to ignore.

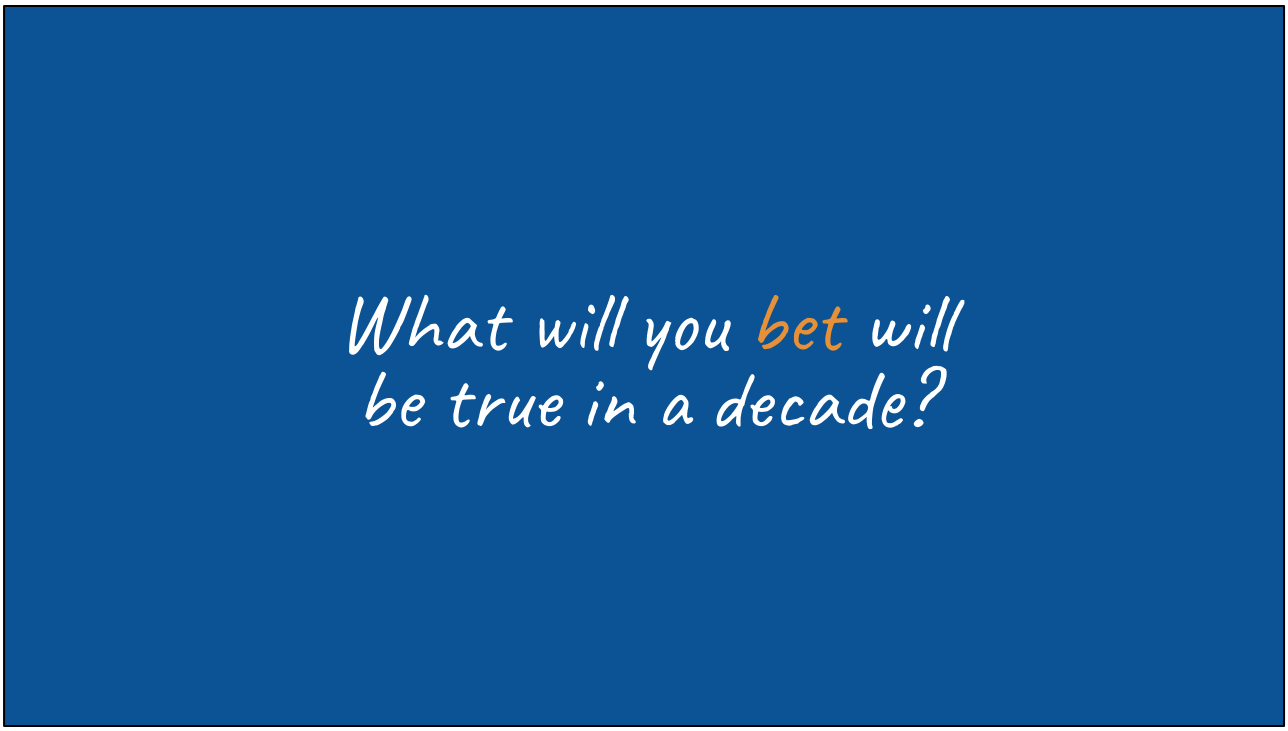
Let go of the illusion of control, or the idea that it will ever be perfect. It's beautiful *because* it's messy.

But that doesn't mean you should just throw your hands up. You can still continuously improve it.

Stay grounded in today, but with an eye towards eventual convergence.

We want a smooth evolution over time. Slowly getting better, in an sustainable way.

But eventual convergence… to what? This is where the long-term thinking comes into play.

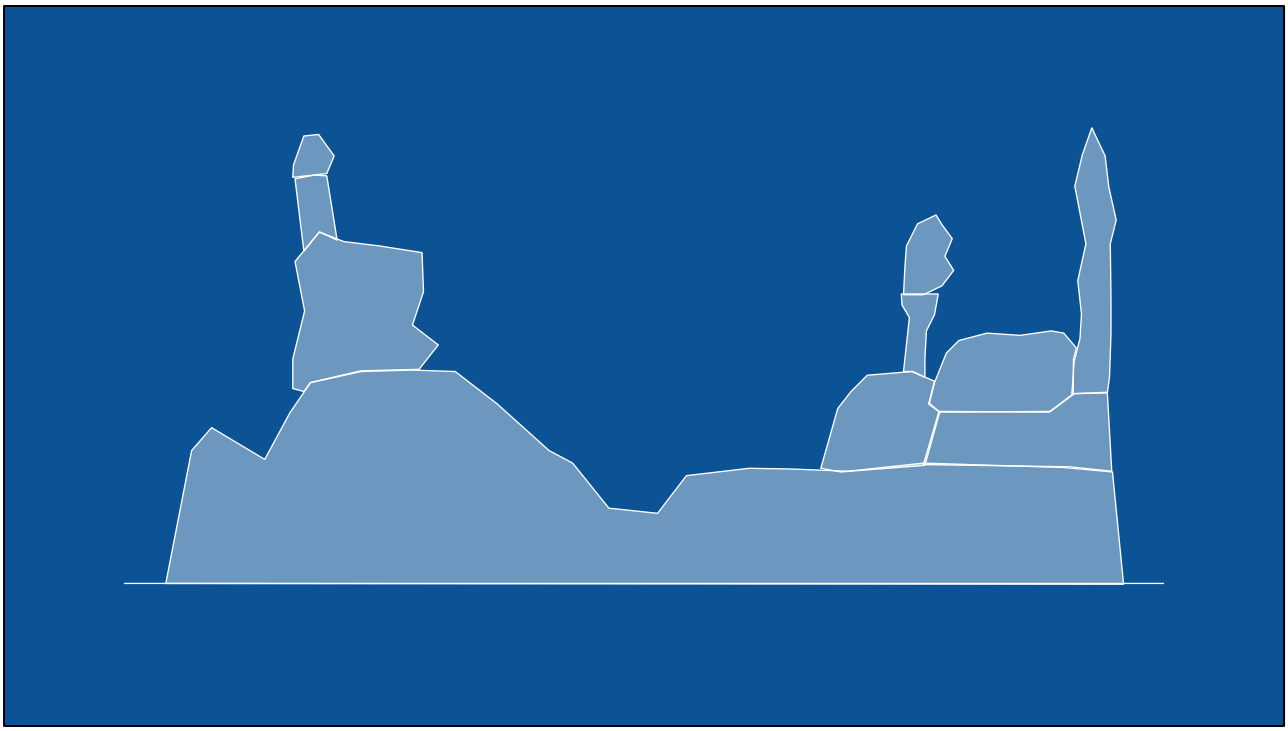


As an exercise, think at a very high level about things you expect to be true in 10 years or more. What will change? What will stay the same? What's an arbitrary implementation detail, and what's a force of gravity?

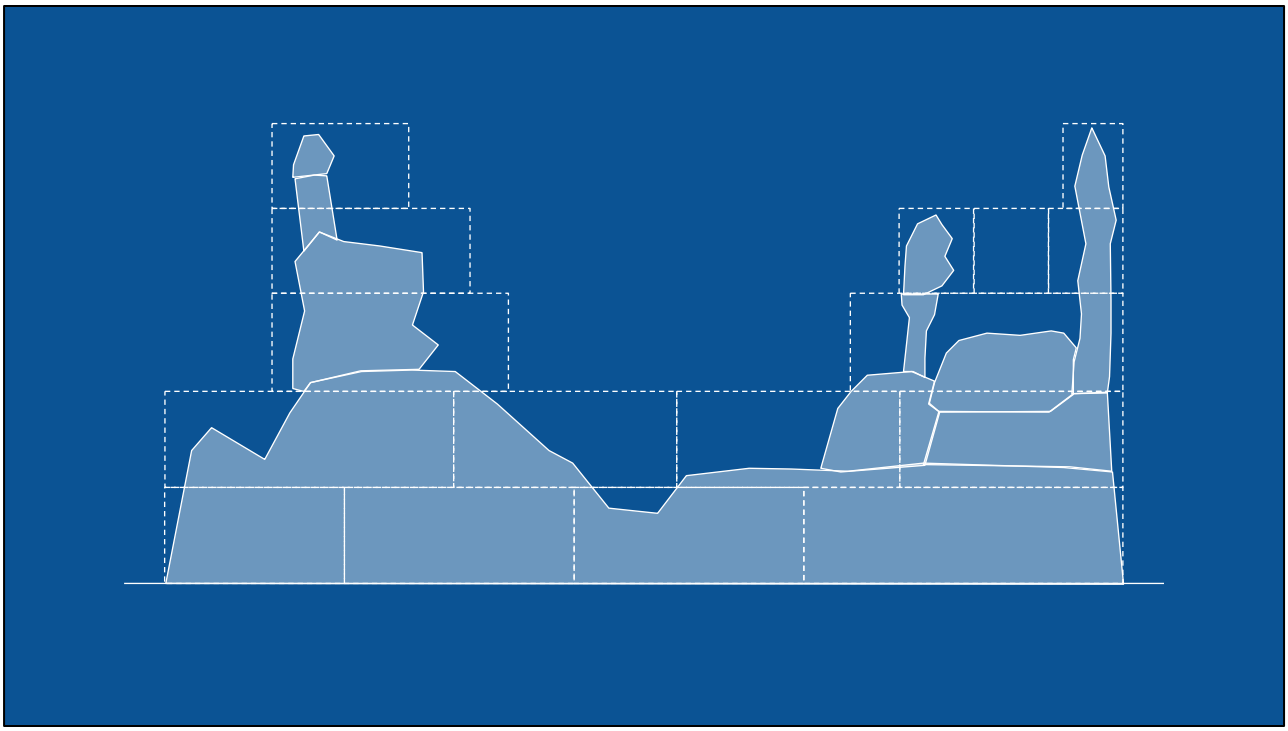
The precise time horizon will depend on how long you expect the platform to be around. Is your startup about to run out of runway? Don't bother thinking further than a few months. Are you a large, stable company where the platform is a core part of your business? Think about the 5-10 year time horizon.

If you are OK with an imprecise view, this is actually pretty easy. What precise uses, and precisely how widespread, will Ethereum have in 10 years? Impossible to say. But will crypto in general be more or less prevalent in a decade? That's *much* easier.

You can use your intuition for this, by the way! As long as you have accountability for the platform over the long-haul, you have skin in the game, so you can trust your calibrated intuition and not have to think about it too much.



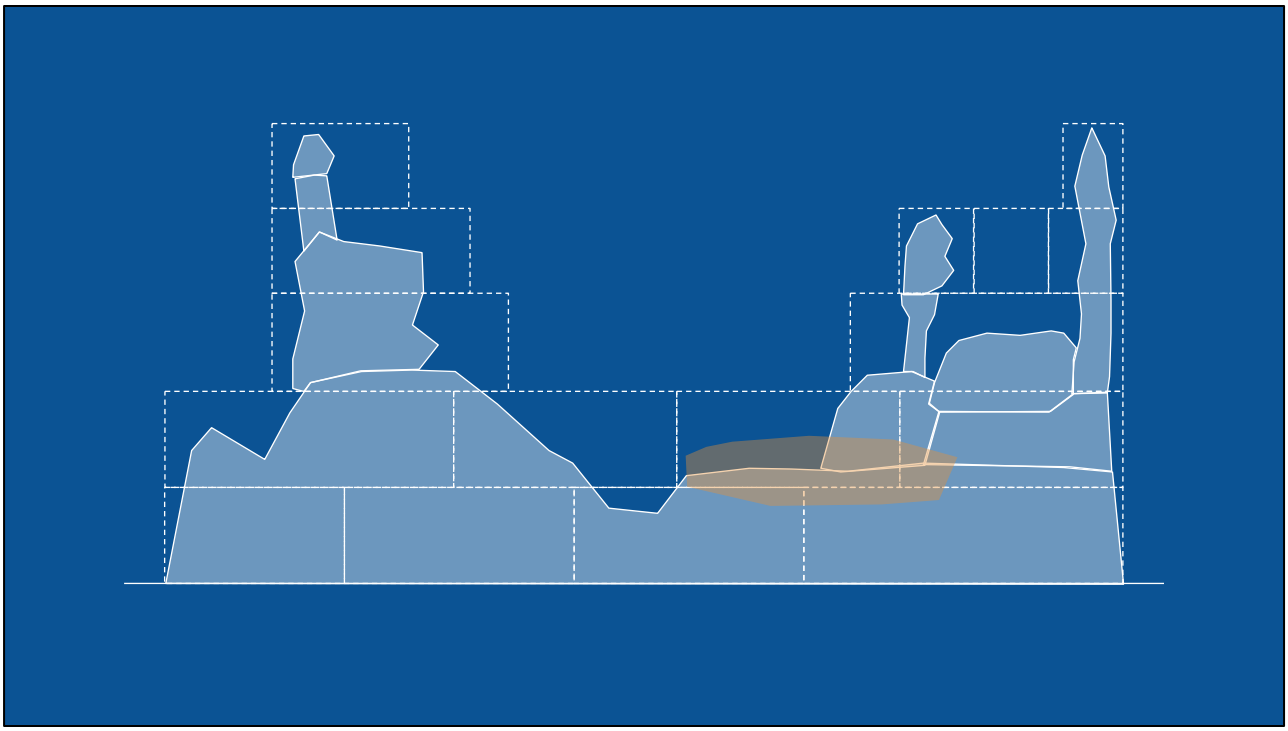
So this is what the platform looks like now



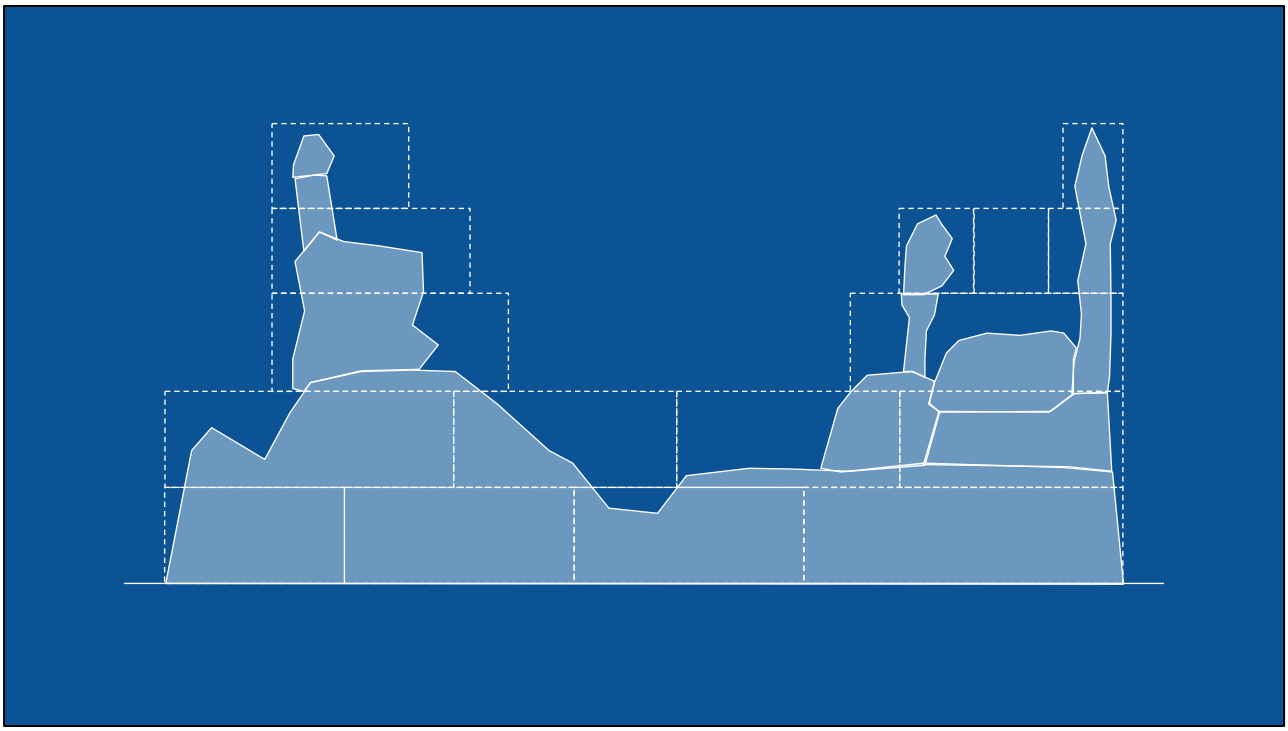
But maybe this is roughly where you bet platform will be in a decade. The platonic ideal of the platform.

Don’t waste too much time getting specific about this, by the way. Everything is always changing. As you and the ecosystem learn more and evolve, the ideal will change. And generalizing things in a platform prematurely is very expensive and likely to be wasted effort.

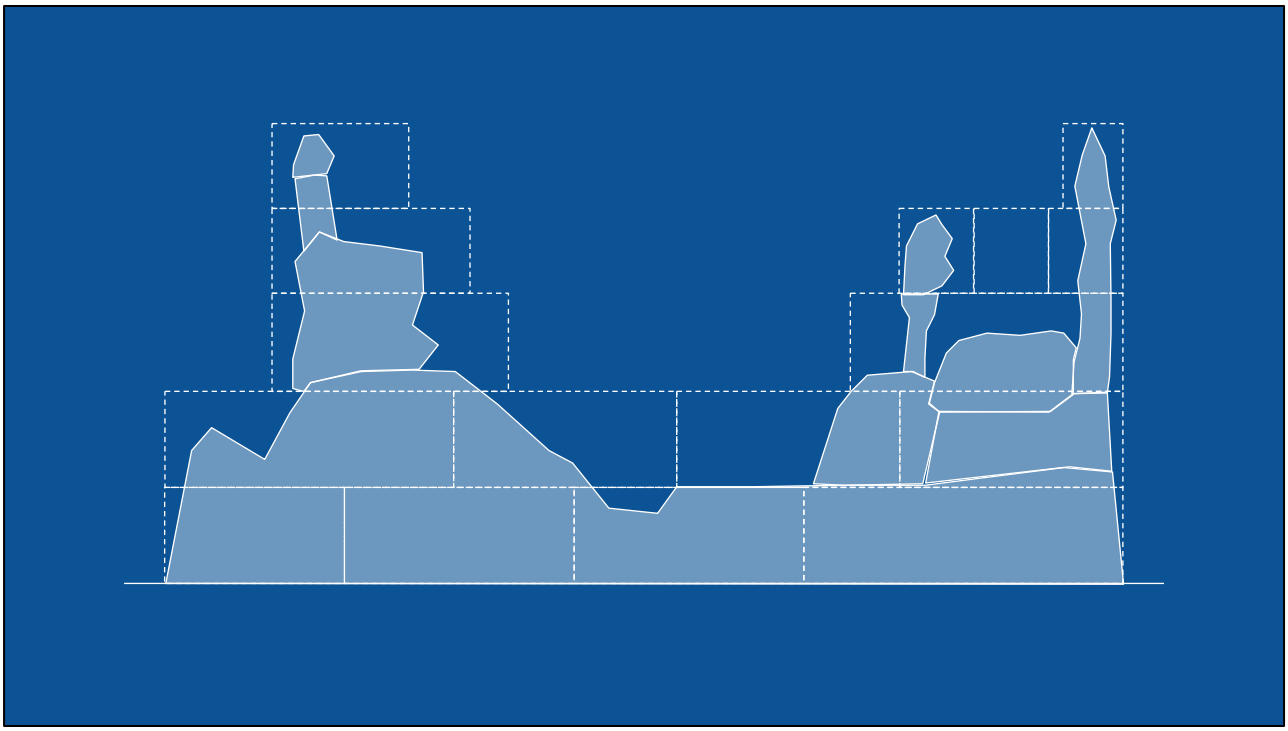
Instead, just keep this *rough* shape in the back of your mind as you work on the platform. You only need a handful of architects that trust each other to have this kind of picture in the back of their heads. One useful lens is to imagine that all of the 1P features of the platform had been implemented into terms of APIs that were also exposed to 3P. What would they have looked like?



So let’s say you’re doing some basic refactoring work in this part of the platform.



You can nudge it a bit closer to the ideal while you're there. Taking maybe 10% more time to clean it up *just a little*. So what looked like this...



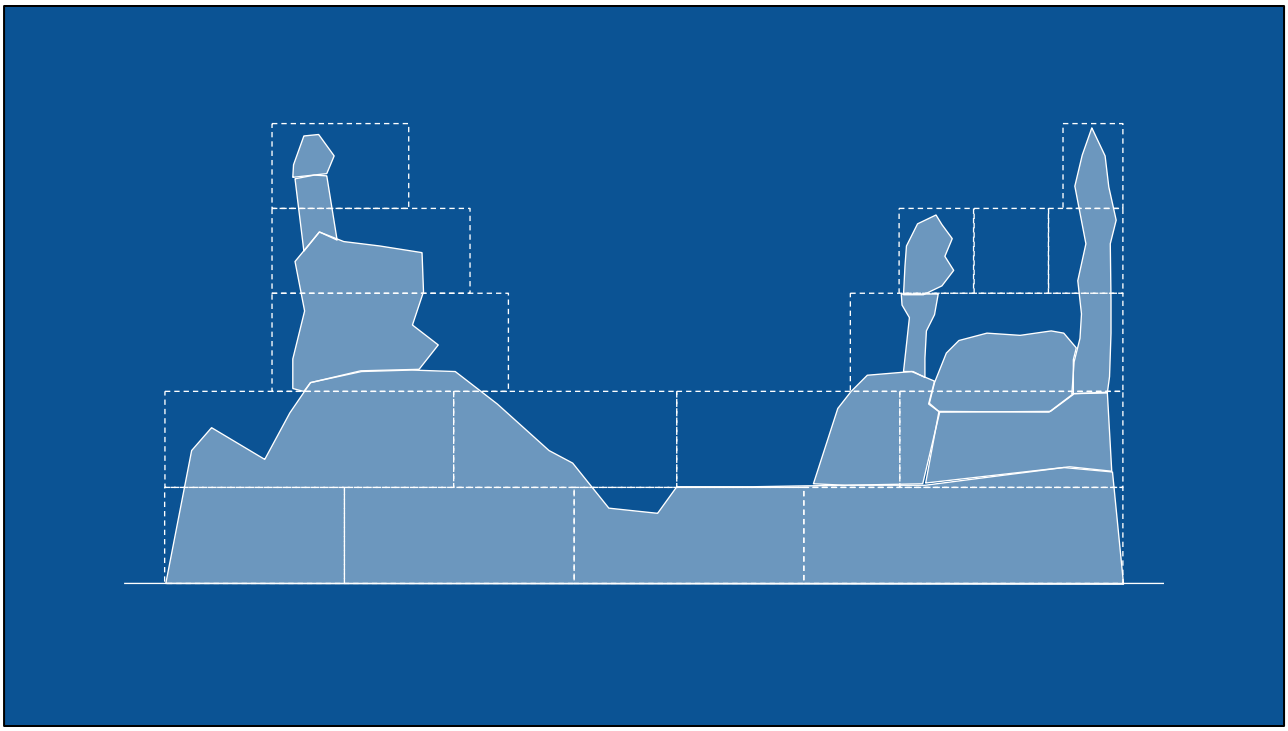
Now looks like this. Just a teensy bit better!

It’s small… but that’s the point! You didn’t have to stick your neck out too far to do it. Instead of a change-the-engines-in-mid-air style change, you just spent a little bit more time and effort to clean it up.

These small, deliberate changes can accumulate.

When you're making a change, don't just think about the direct value and cost. Think about, "what future value does this change unlock?" What extra refactorings, or new APIs, or use cases, or bug fixes, are now easier than they were before?

Remember that you're unlocking value for yourself, but also for all of the developers and users in the ecosystem and what *they* might now be able to do.

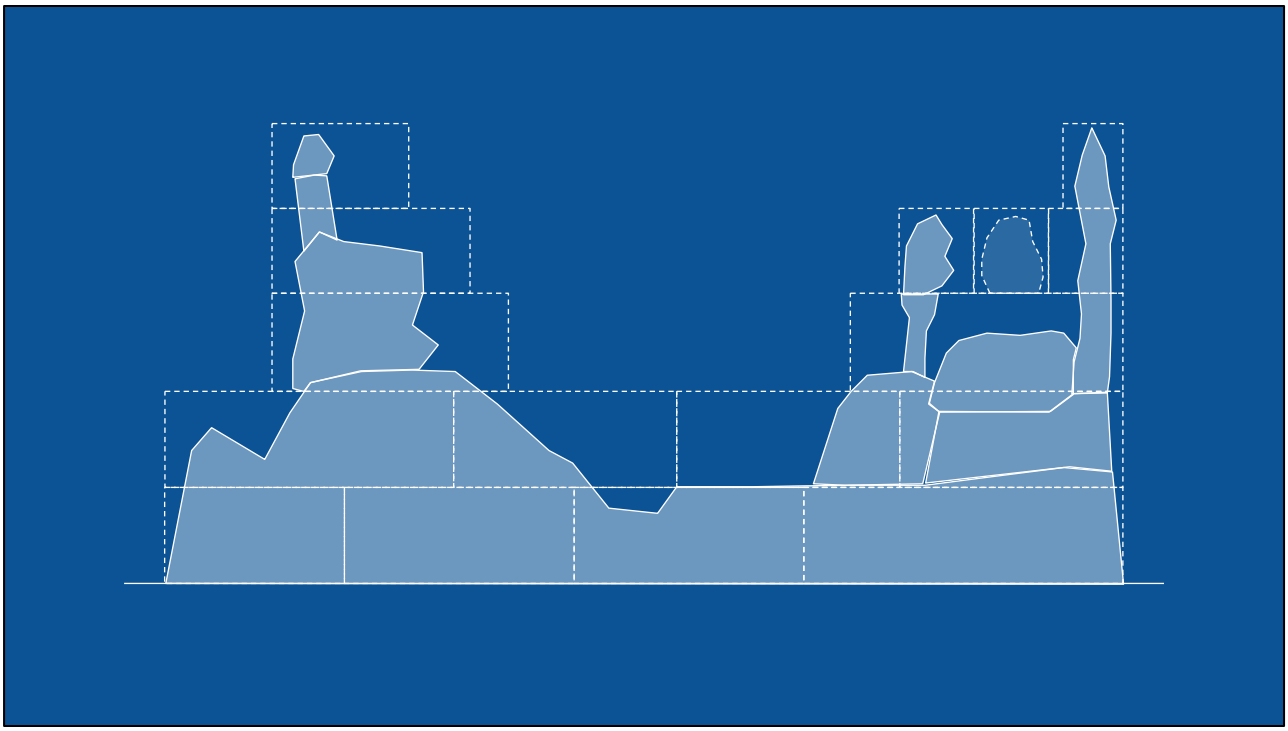


Let’s say we want to unlock some totally new value.

One way to *not* do it is to try to make it perfect and generalized from the start. Generalizing things is extremely expensive, especially when you’re doing it speculatively. Maybe this new API won’t have any product market fit!

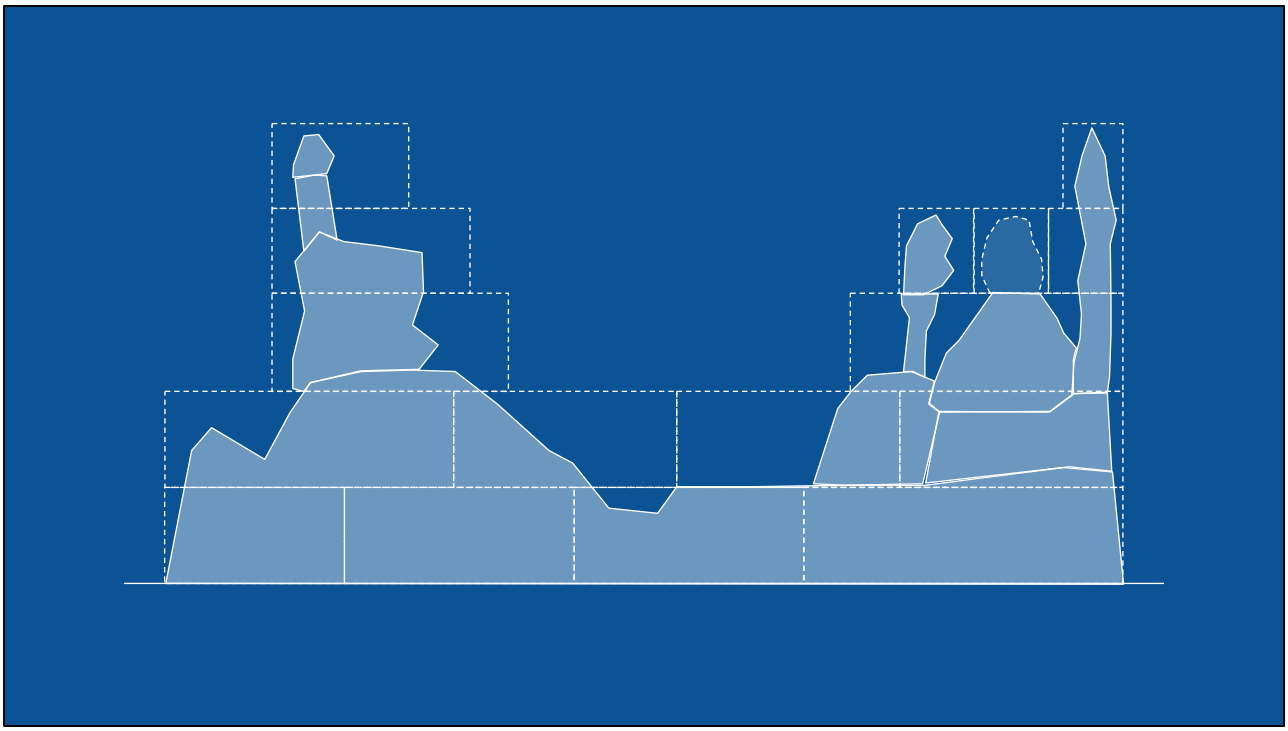
Instead, try to gradually get there.

So let’s say you want value...

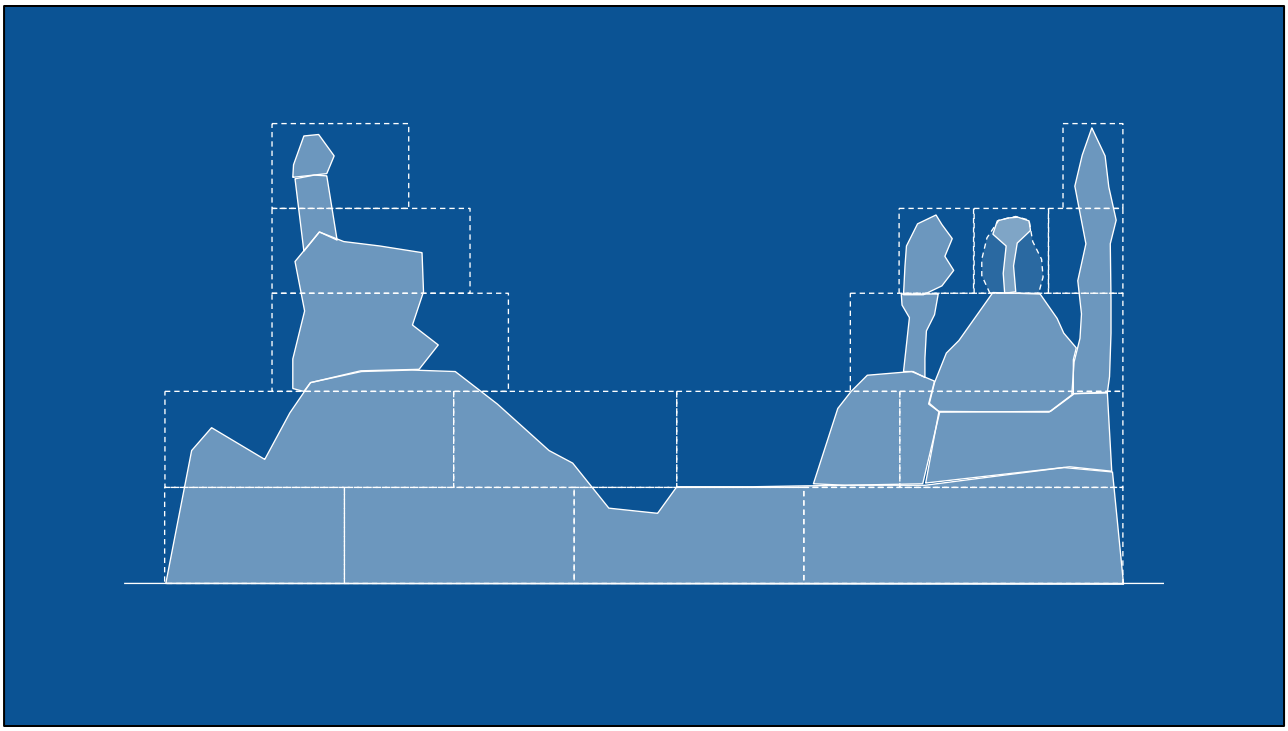


… here.

How do you do it? Well, there's a thing down there that's *almost* the right size and shape beneath it. Maybe you can extend and generalize that concept.



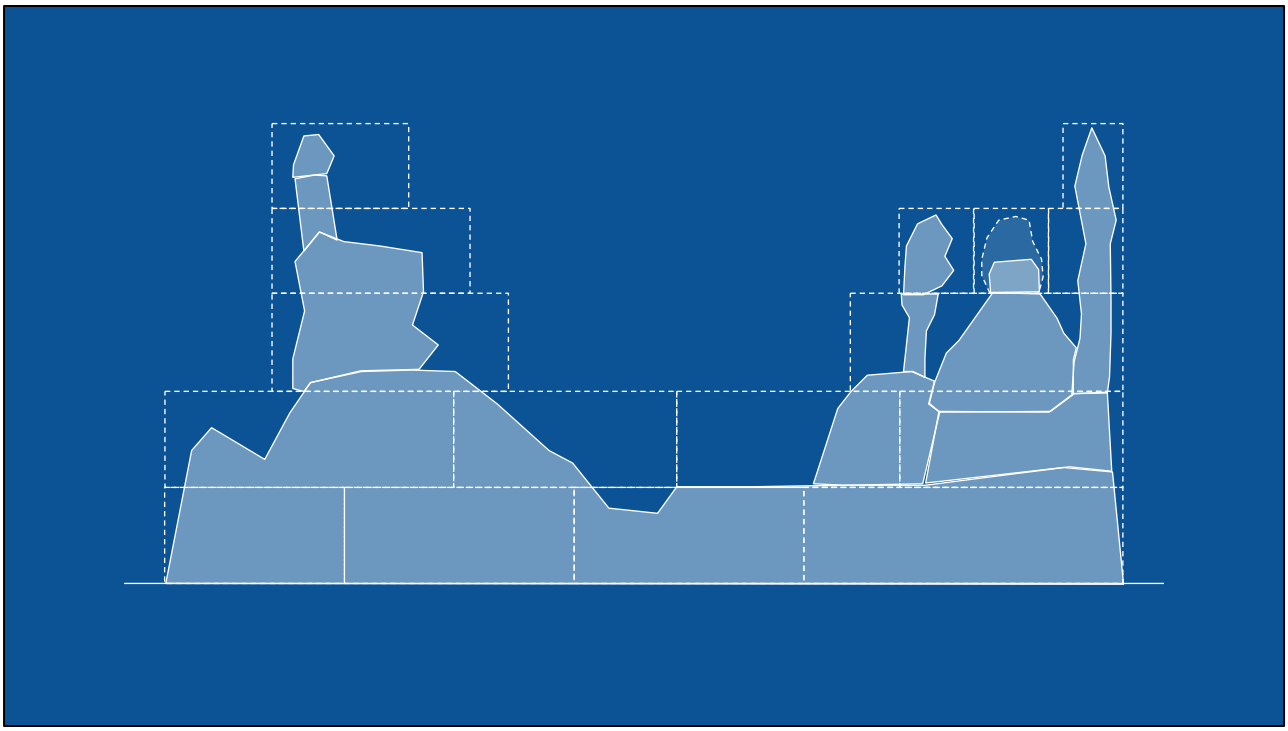
You haven’t started working on the new API yet… but you have cleaned up the platform, gotten it closer to the ideal, and likely unlocked some new use cases in the ecosystem that are now viable. Even if you end up not doing the new feature, you’ve still added value!



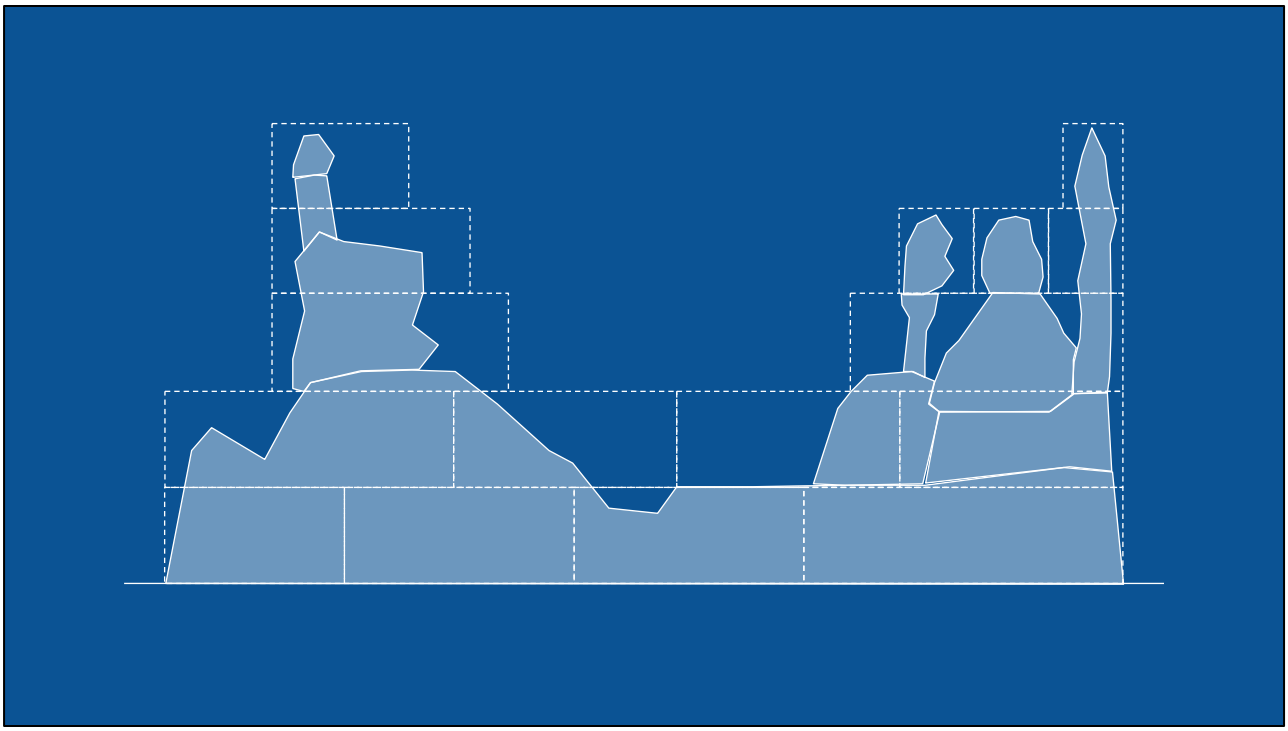
Now you have a good foundation.

You could build up to the top of the use case, as that’s where the most value is.

But that would also be extremely tippy and brittle.

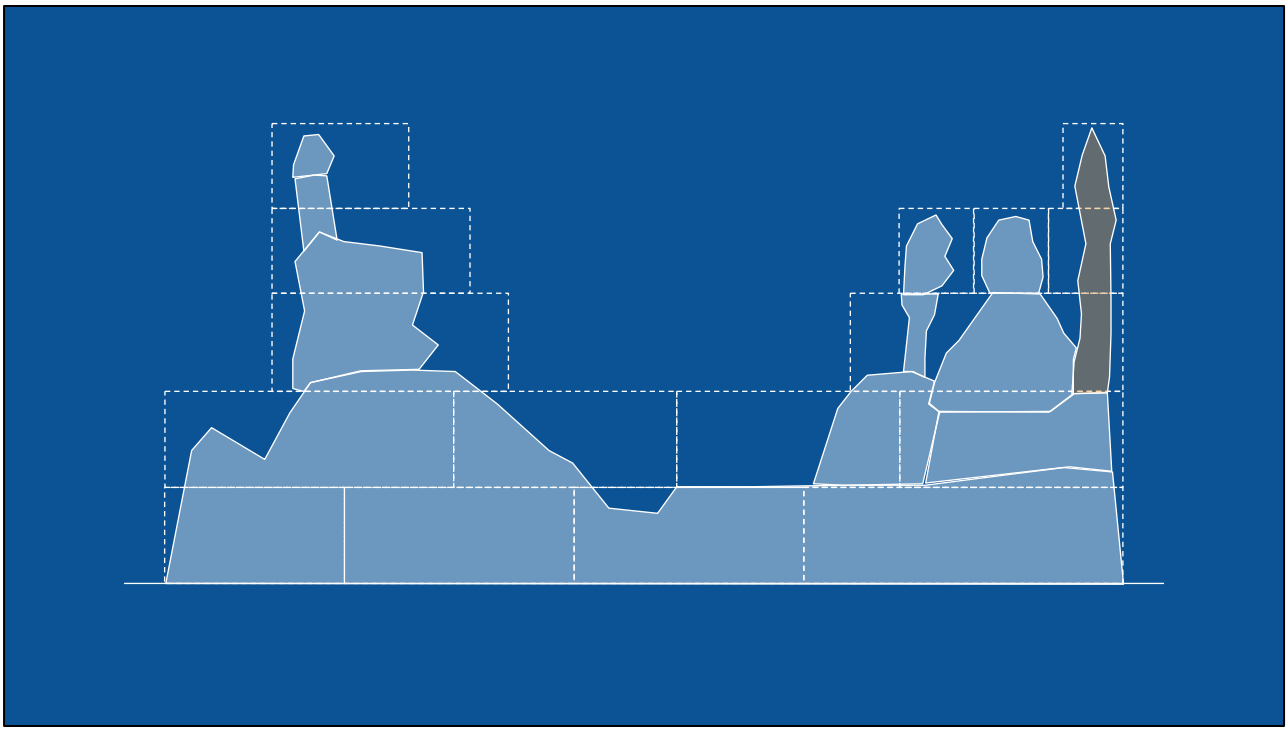


Another approach is to build *upward* from the foundation. It might feel like you're starting with the lower priority use cases, but you'll be leaving a stronger foundation.



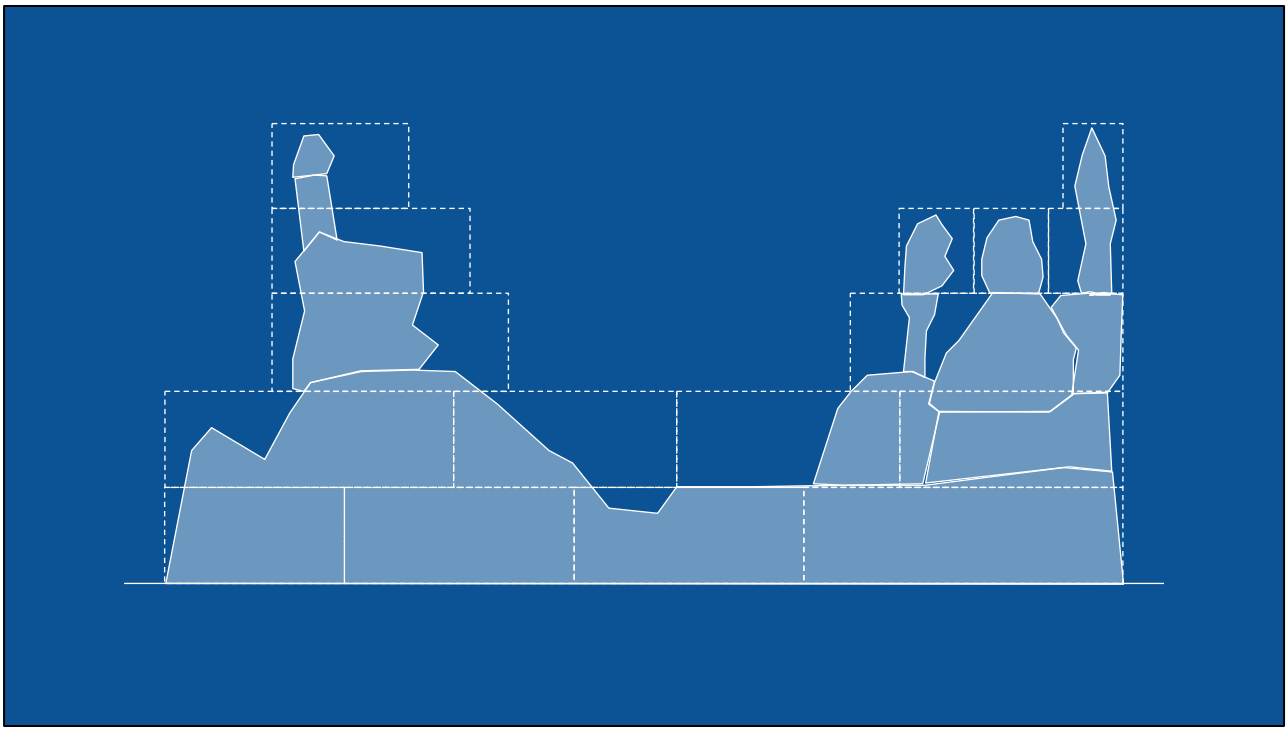
And from there you keep on building upwards.

There’s another way to add value.



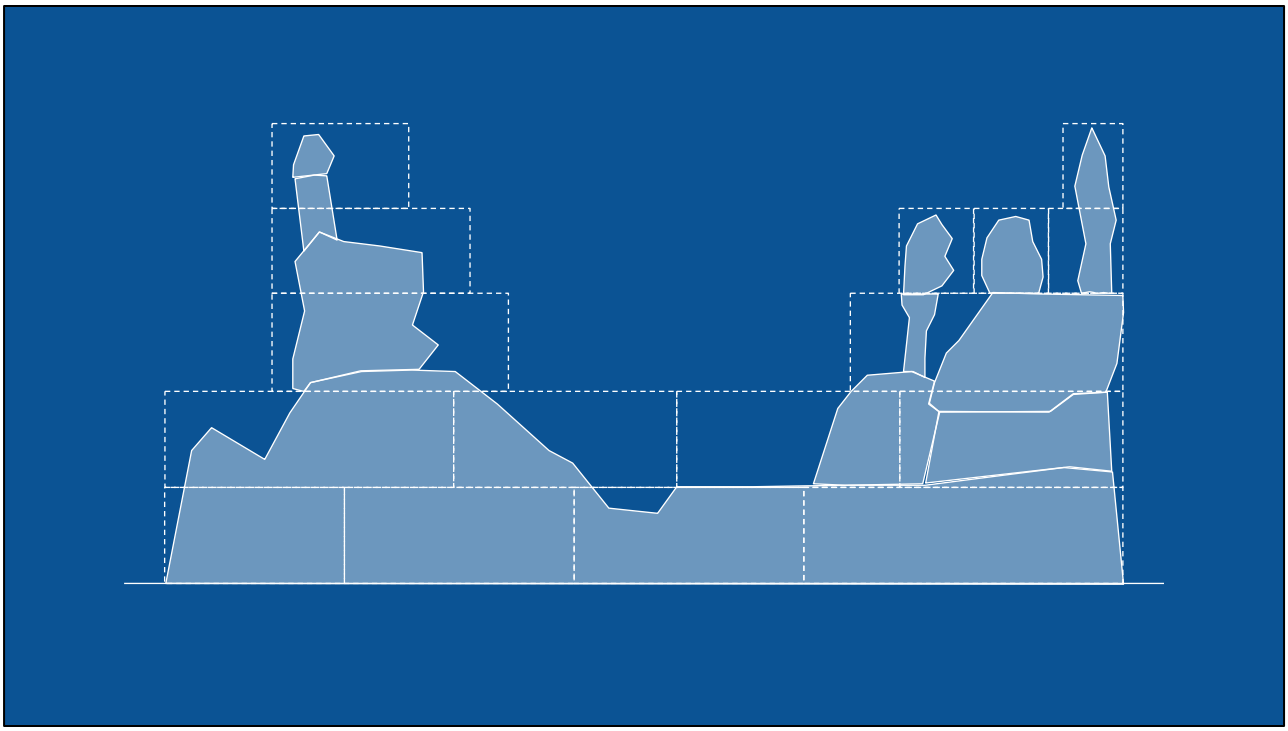
See this monolith here? It crosses multiple API boundaries in the ideal structure.

In fact, some of that new code we just built could rely on parts of the code in the monolith… but it can’t because it’s strongly coupled within the monolith.

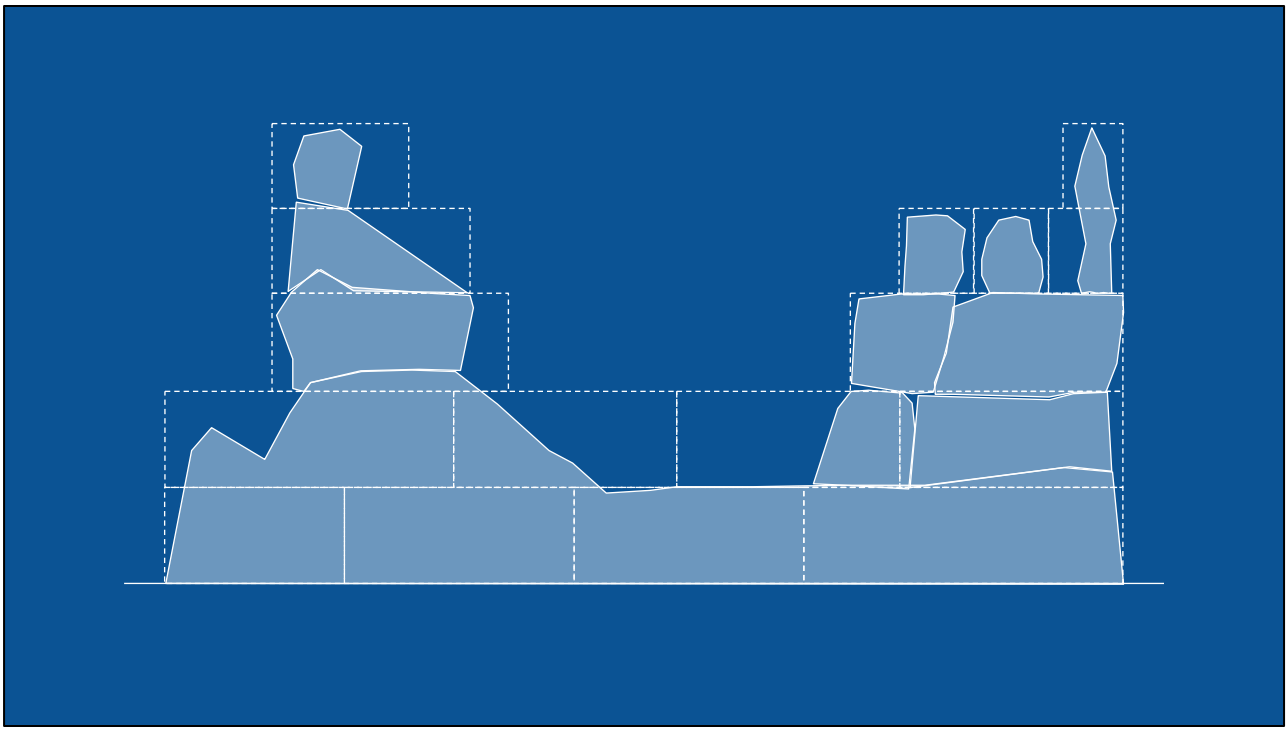


Look for an existing internal API boundary within the monolith that's roughly in the right place, and harden it, reshaping it to be closer to the ideal line, and expose it externally.

This is called "primitive archeology" --looking for semantics that are *implied* in the system and unearthing them.



Now that those two pieces align nicely, you might be able to merge them into one piece, instead of having two subsystems.



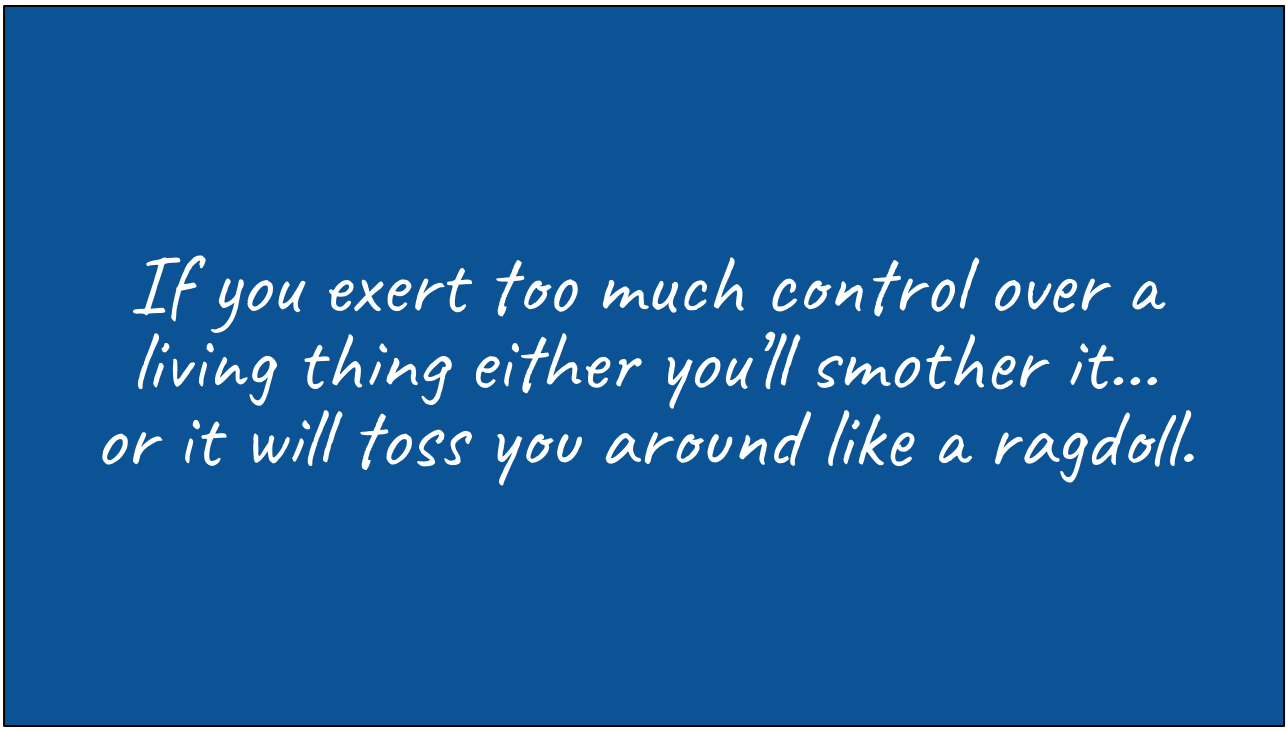
If you keep going, you get something like this.

It… still doesn’t look great.

But if you compare it to where we started, it looks *way* better.

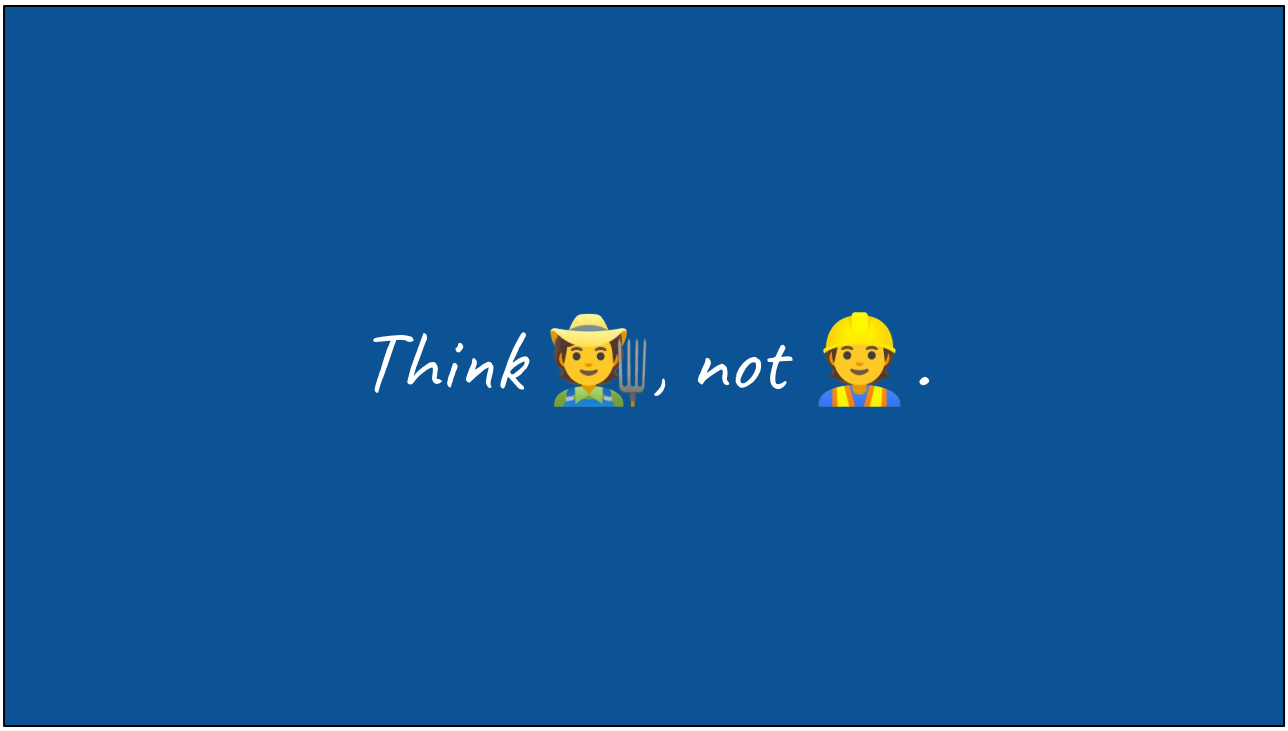
This process might be frustrating… but only if you hold on to an idea of a platform as a designed object.

Platforms are *alive*.



If you treat a living thing like a manufactured object, if you try to control it, you'll smother it.

Or, if the platform is more powerful than you, it will toss you around like a ragdoll and rapidly disabuse you of the notion that you're in control.

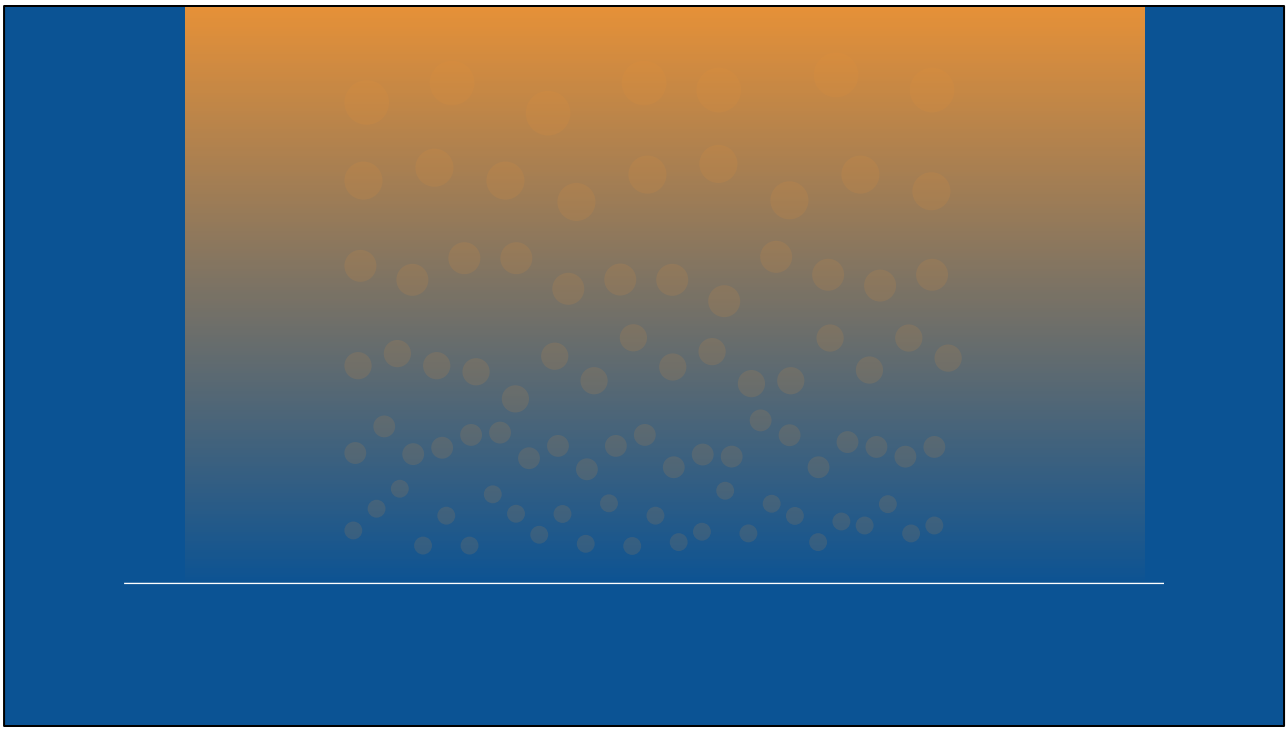


Think gardener, not builder.

1. Ideal Platforms
2. Evolving Platforms
3. Growing Platforms

So now that we know we should be thinking about gardening platforms… let's figure out how to grow them!

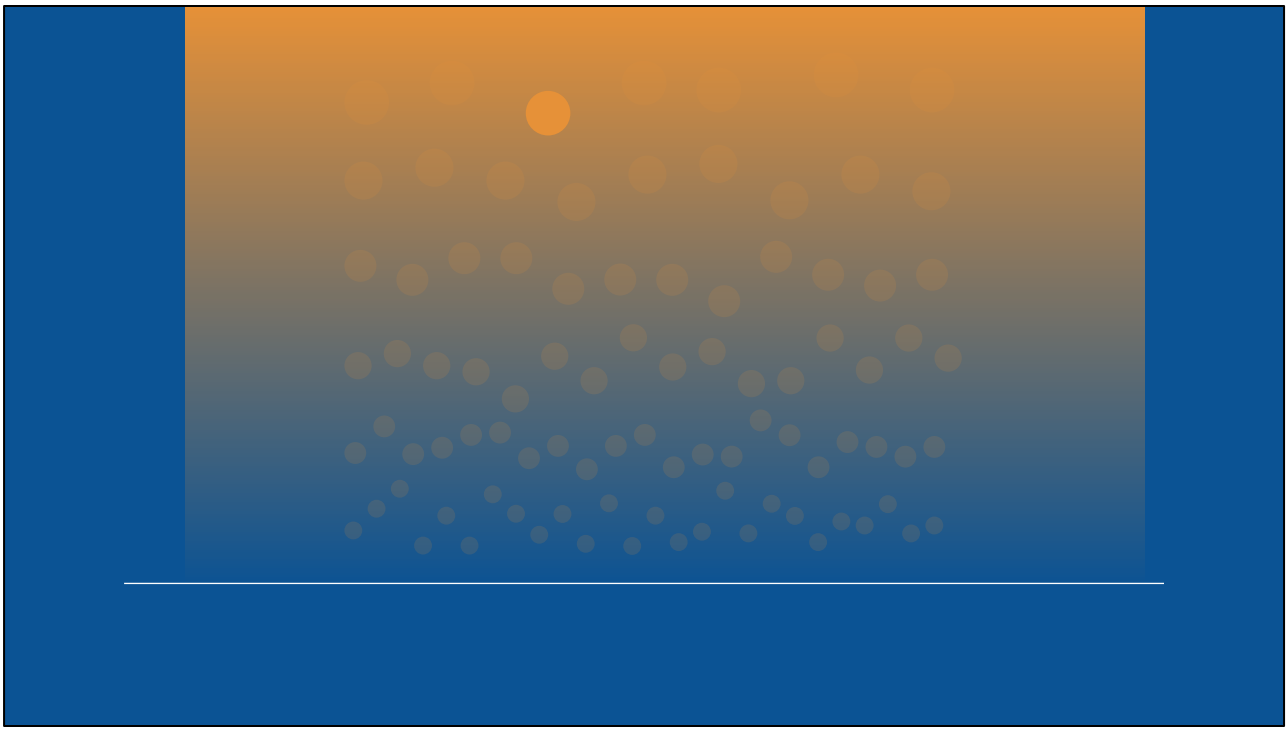
We'll start by building our visual intuition and then get to a playbook.



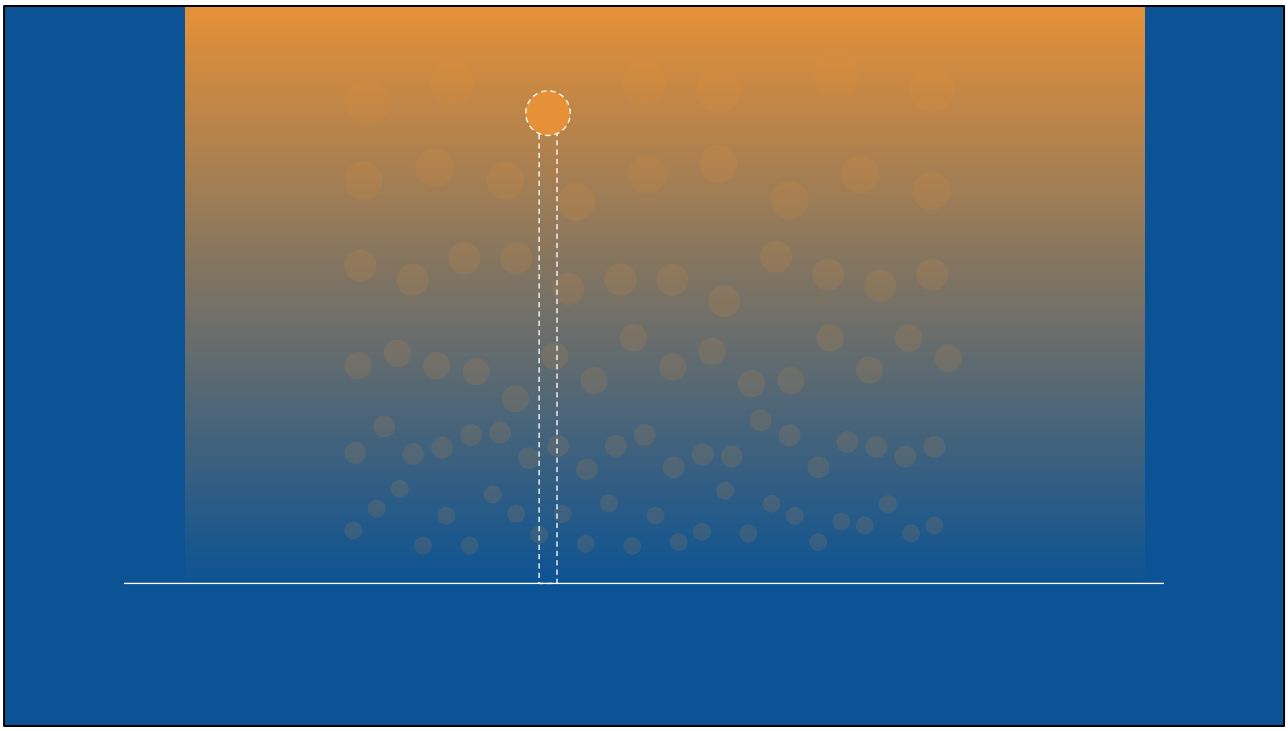
OK, so let’s say you don’t have a platform yet and you want to build one. Maybe this is a totally new company or opportunity. Maybe it’s a new piece of a larger platform.

The same approach works for any of these cases.

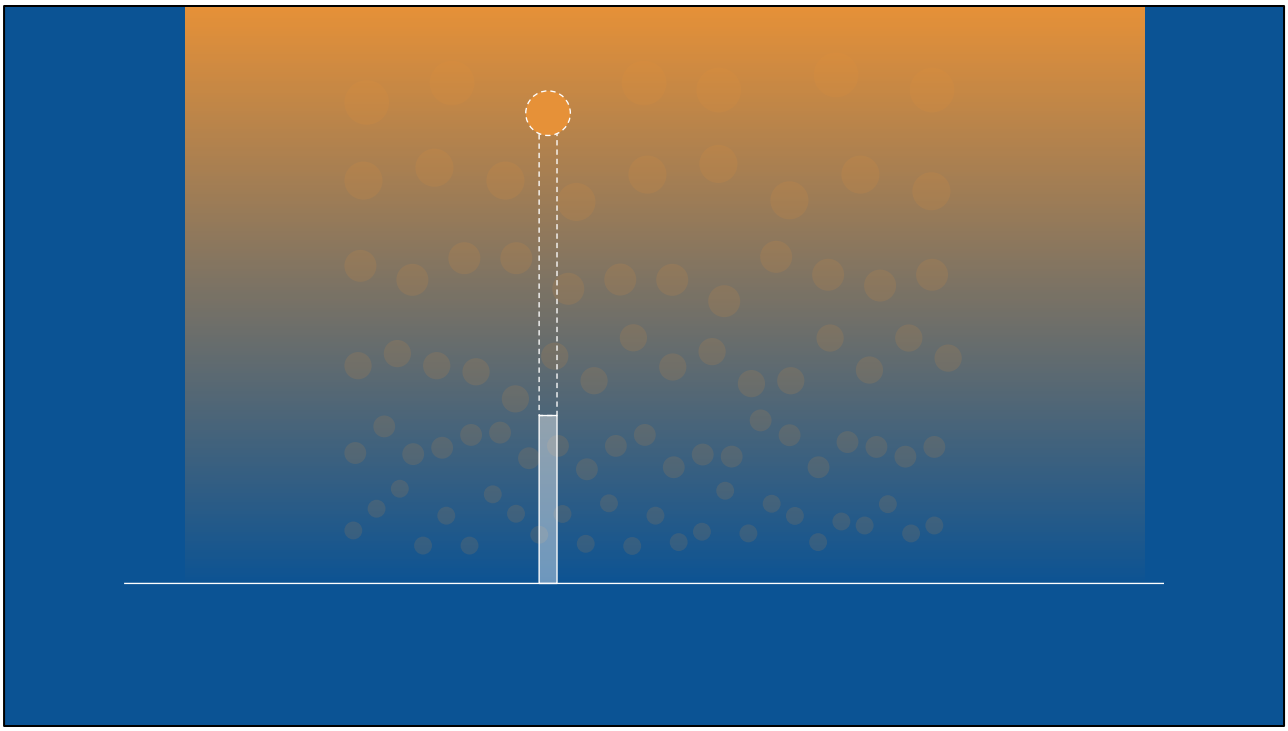
Remember that the bigger opportunities tend to be farther above. But the farther you get from territory where you currently “sit” (to start, the “bedrock”), the harder it is to see the opportunities clearly.



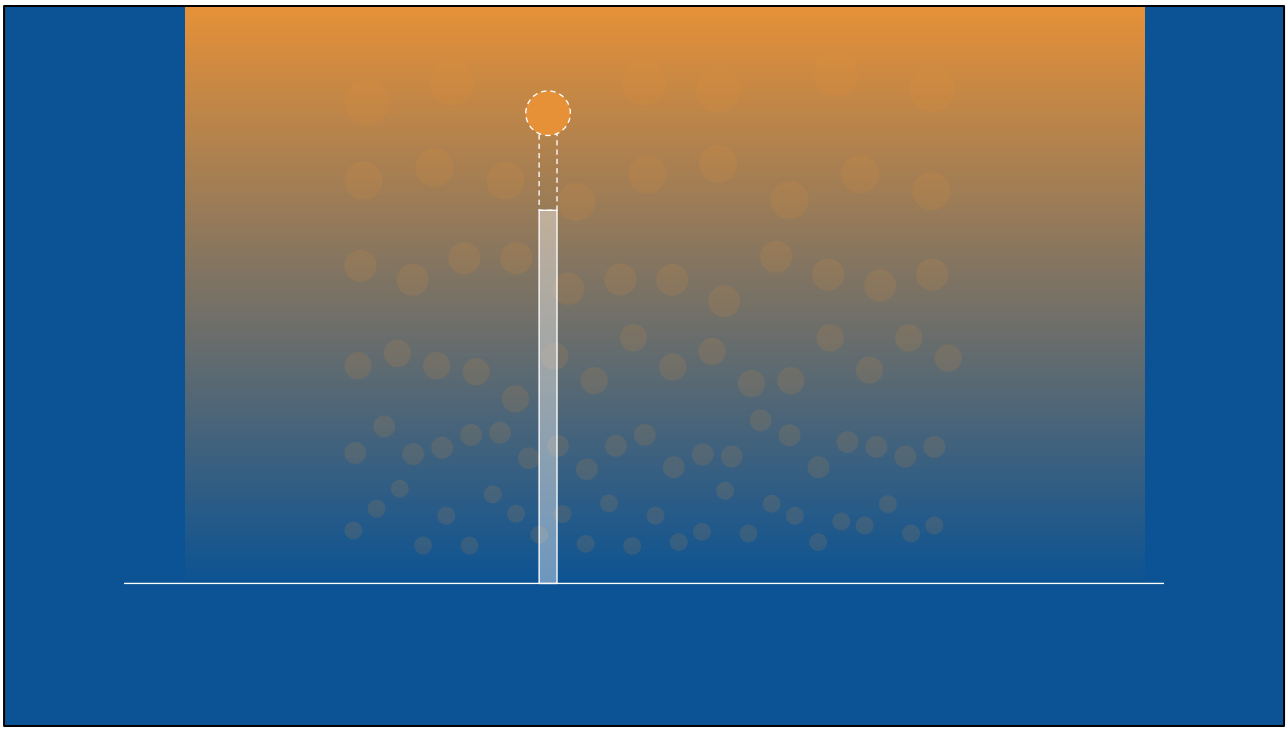
Let’s say you see some value far away that is concrete and big and want to build towards it. Maybe you find it via UXR or other market research.



You have to reach pretty far, so it’s going to have to not be very generalized--more of a tall spindle.

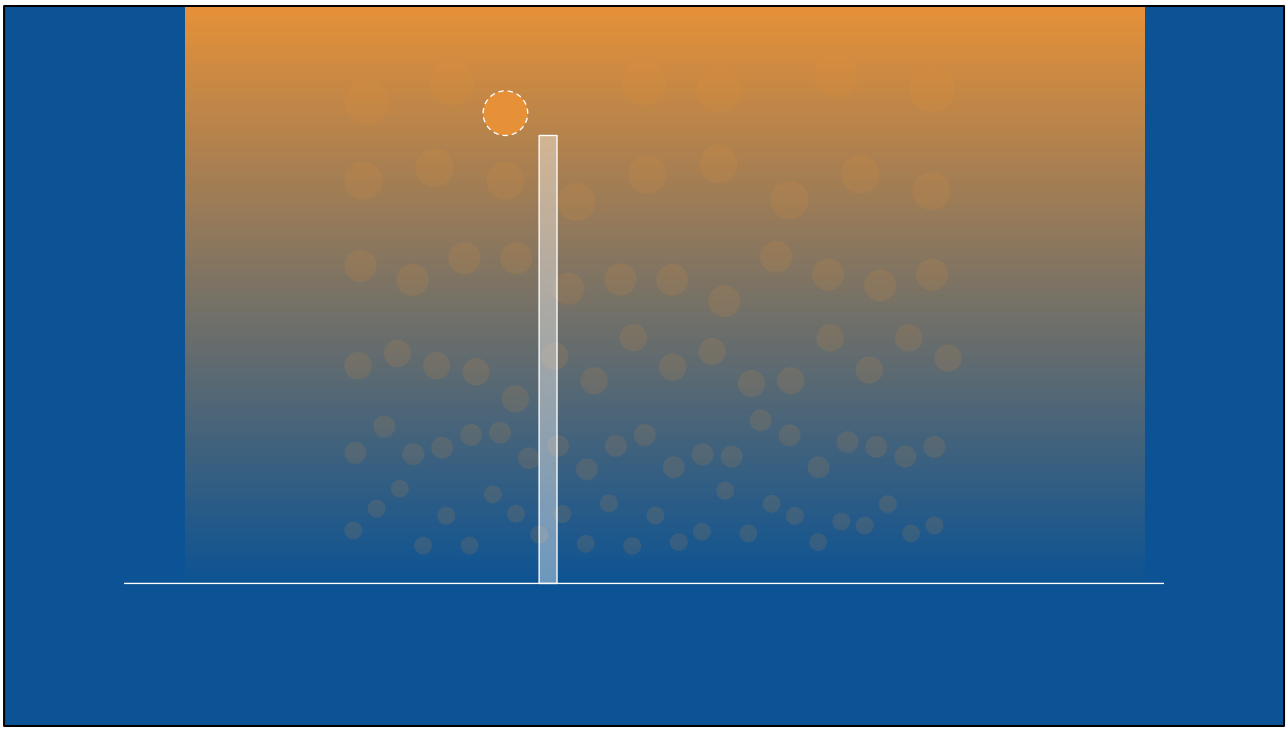


And it’s going to take you awhile to build there.

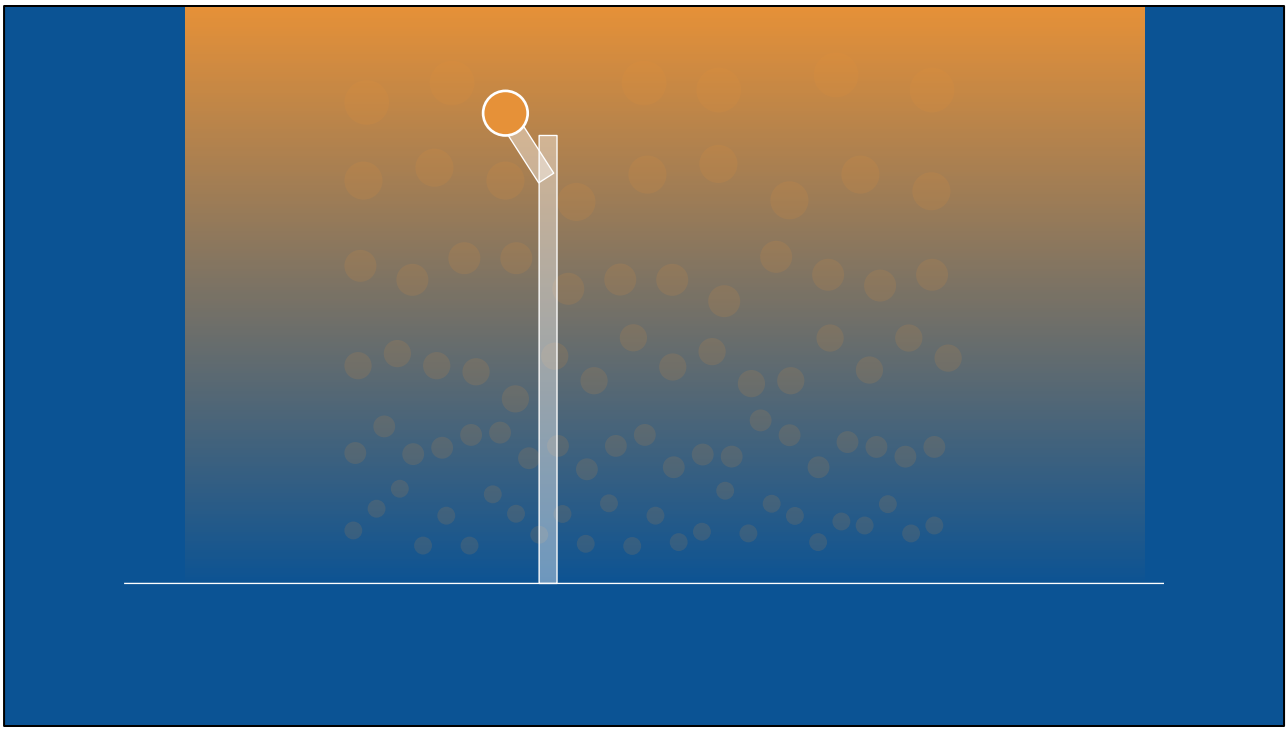


Hopefully nothing bad happens while you're building there. If your team runs out of political capital, or the conditions change, or you run out of funding, you might end up with an expensive partial tower to nowhere.

While you're building, but before you get to your destination, you're exposed and vulnerable.

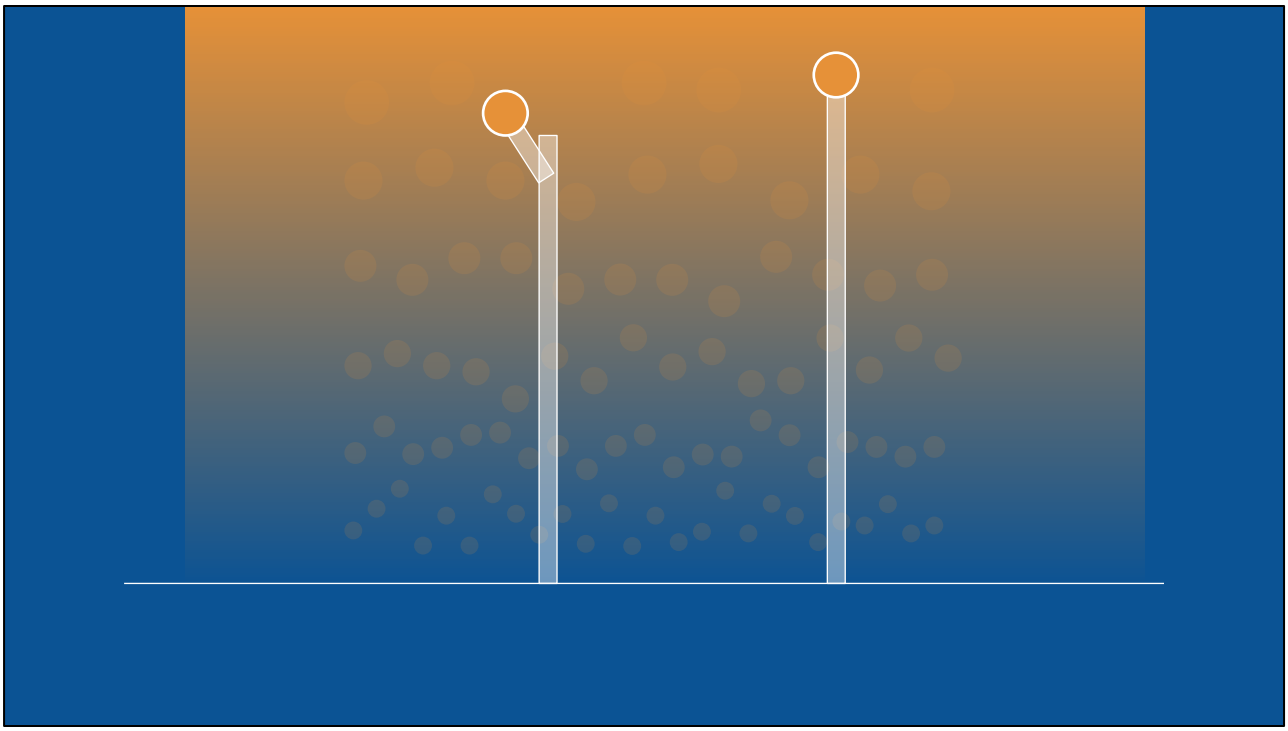


But let’s say you build up to the opportunity and then realize… crap! We sighted off it wrong. Or maybe it moved while we were building. In any case, we’ve built a bridge to nowhere.

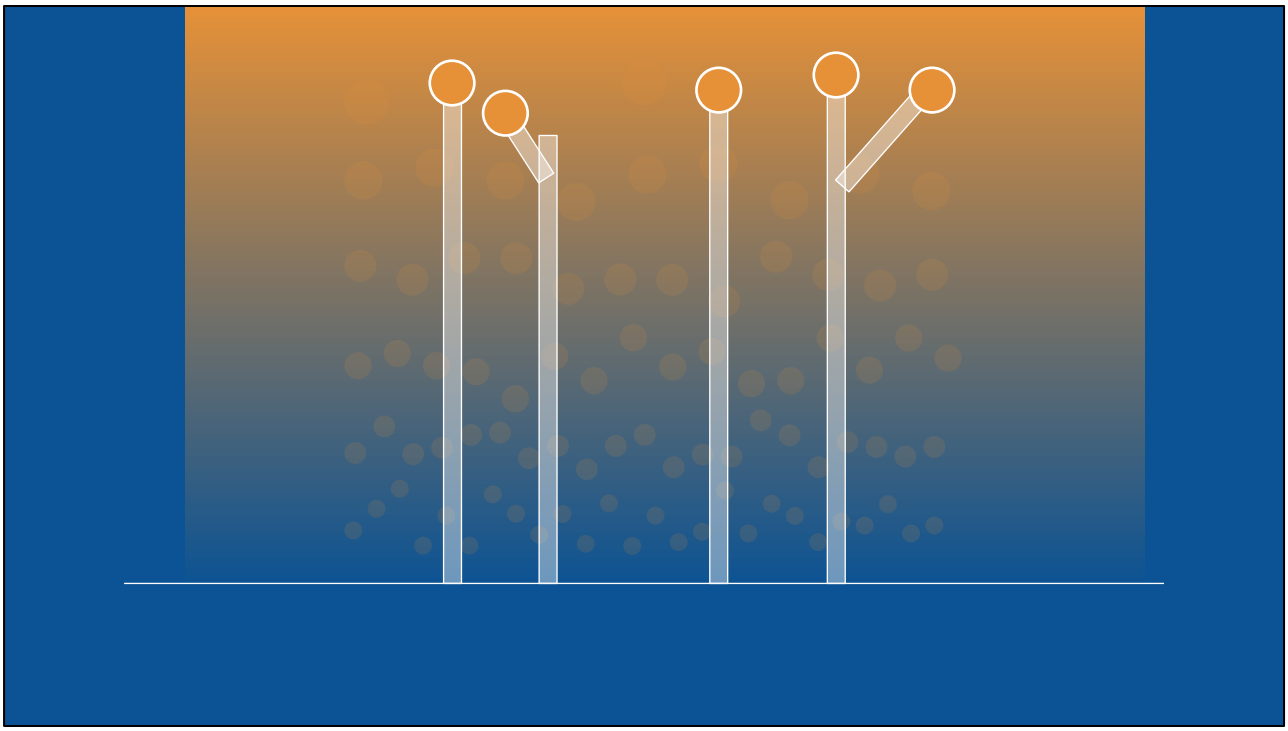


That would be a lot of wasted effort, so we build a hack, duct tape something to the side to bridge just a bit to the left.

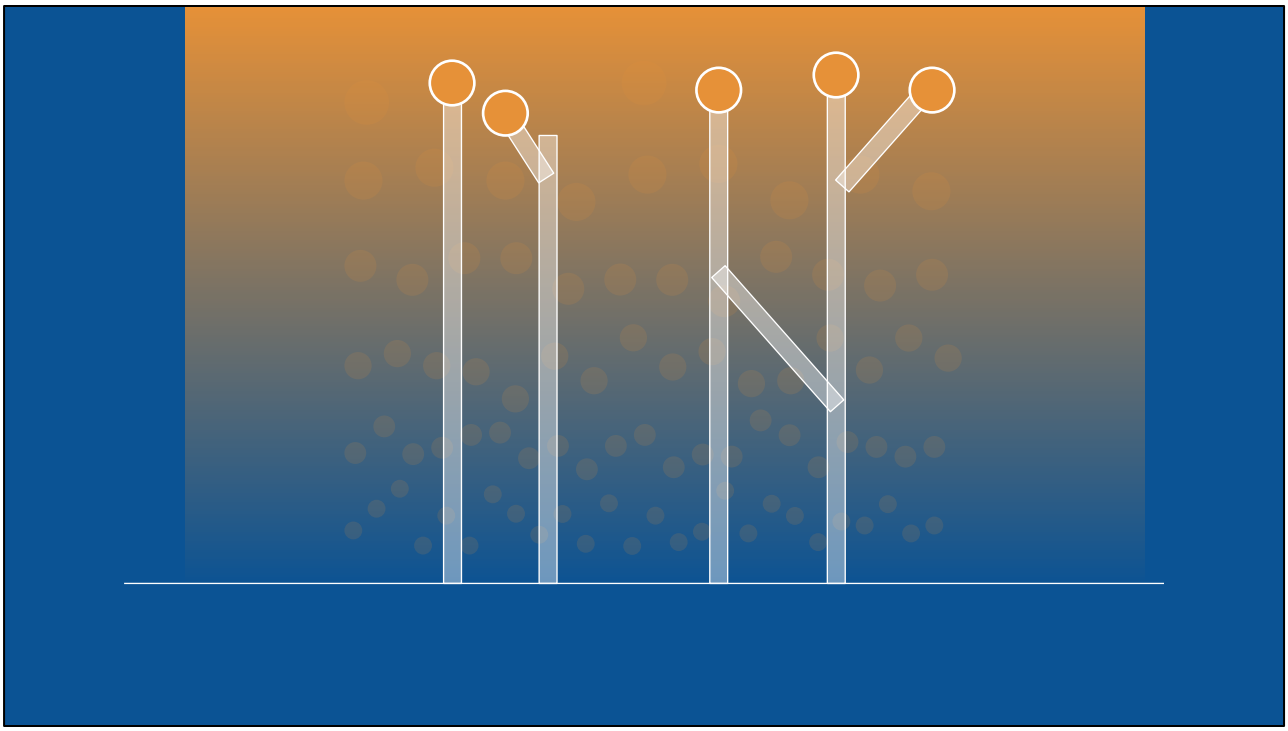
It’s not pretty, but it works...



Now we spot another opportunity, and go for that one.

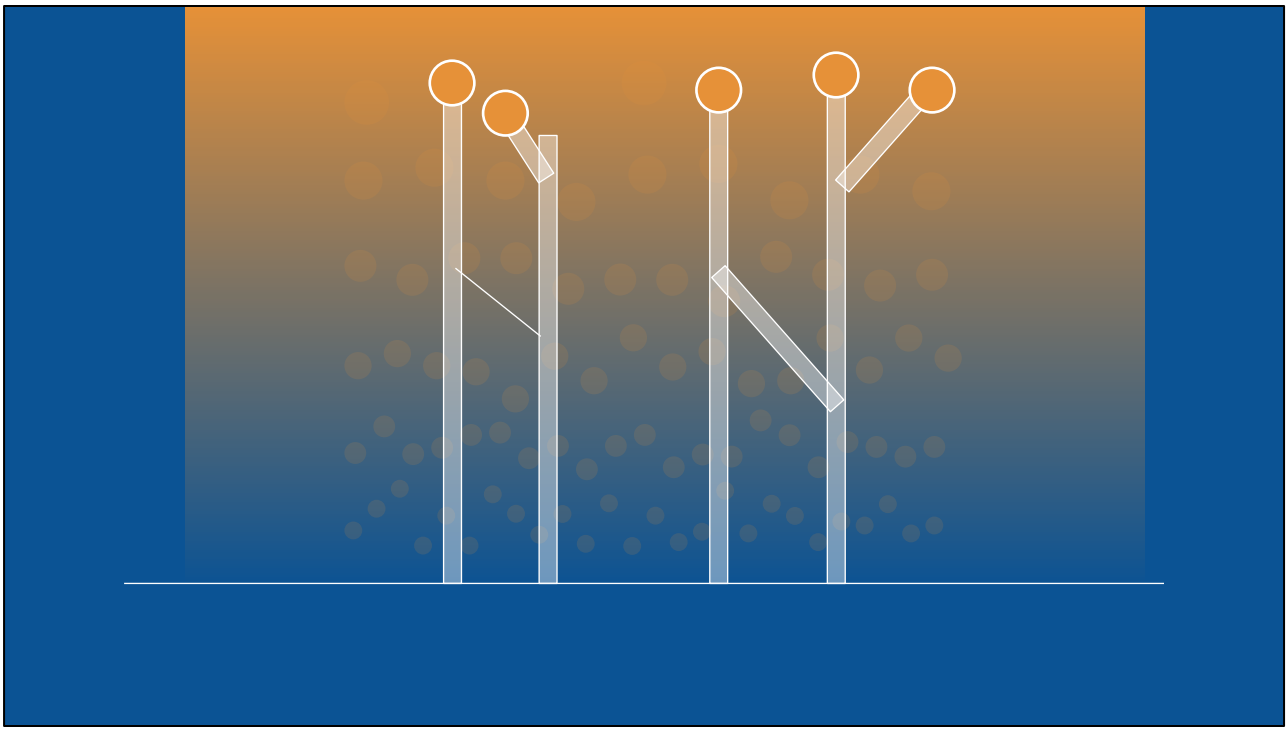


You can keep doing this.



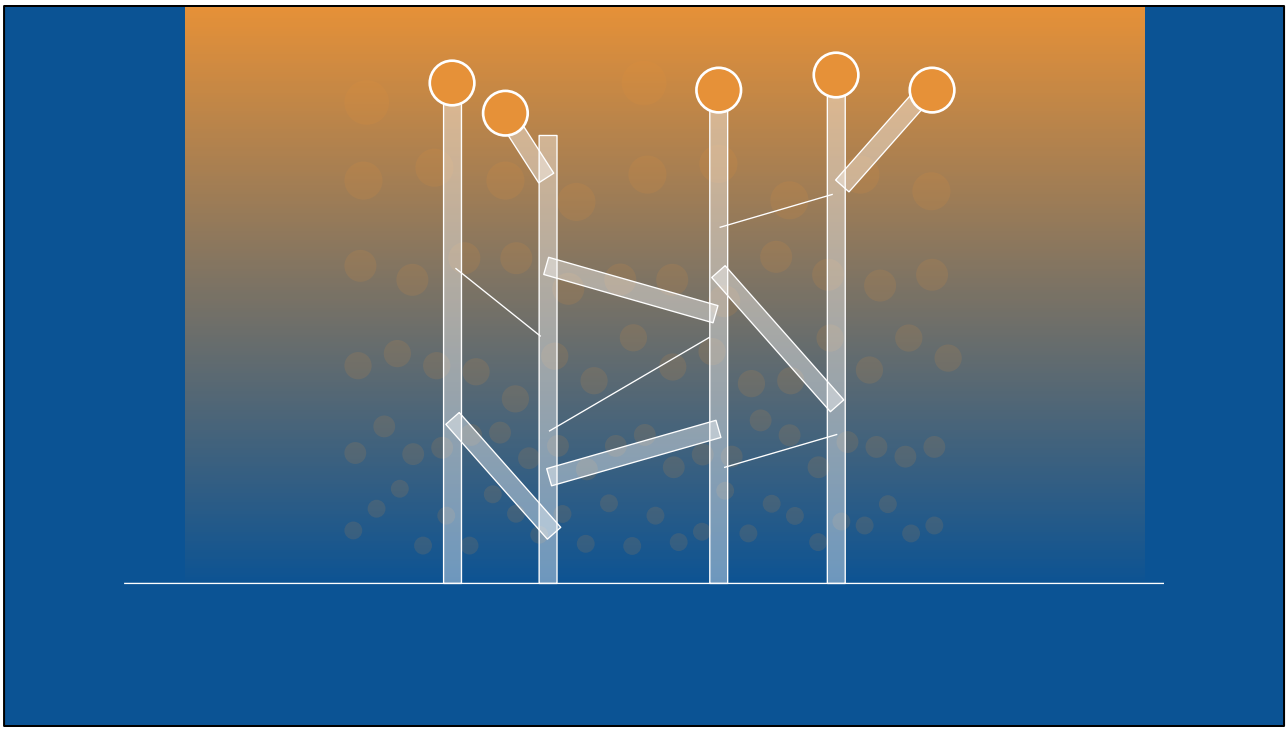
At some point you'll realize you want an integration between different things you've already built.

This one's not that bad.



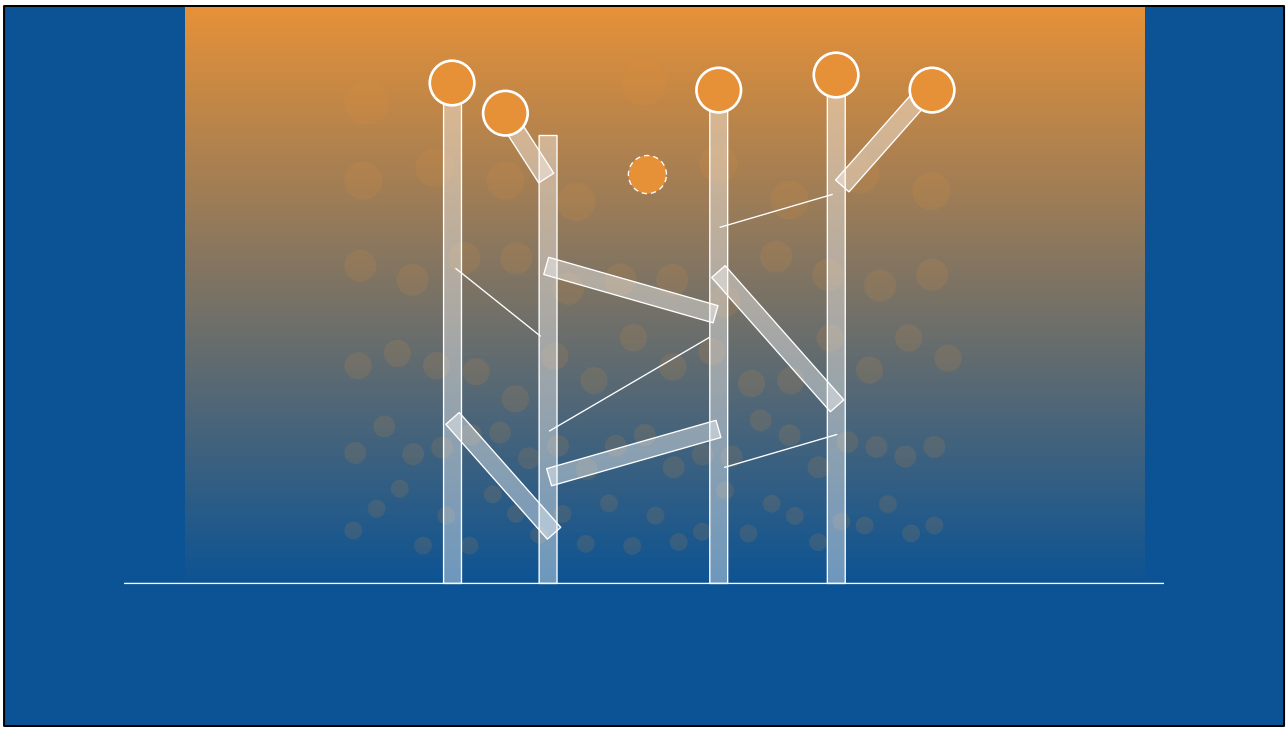
But at some point you'll have a fire drill and need to add an integration RIGHT NOW to land some big opportunity.

So you'll do a bit of flimsy duct tape.



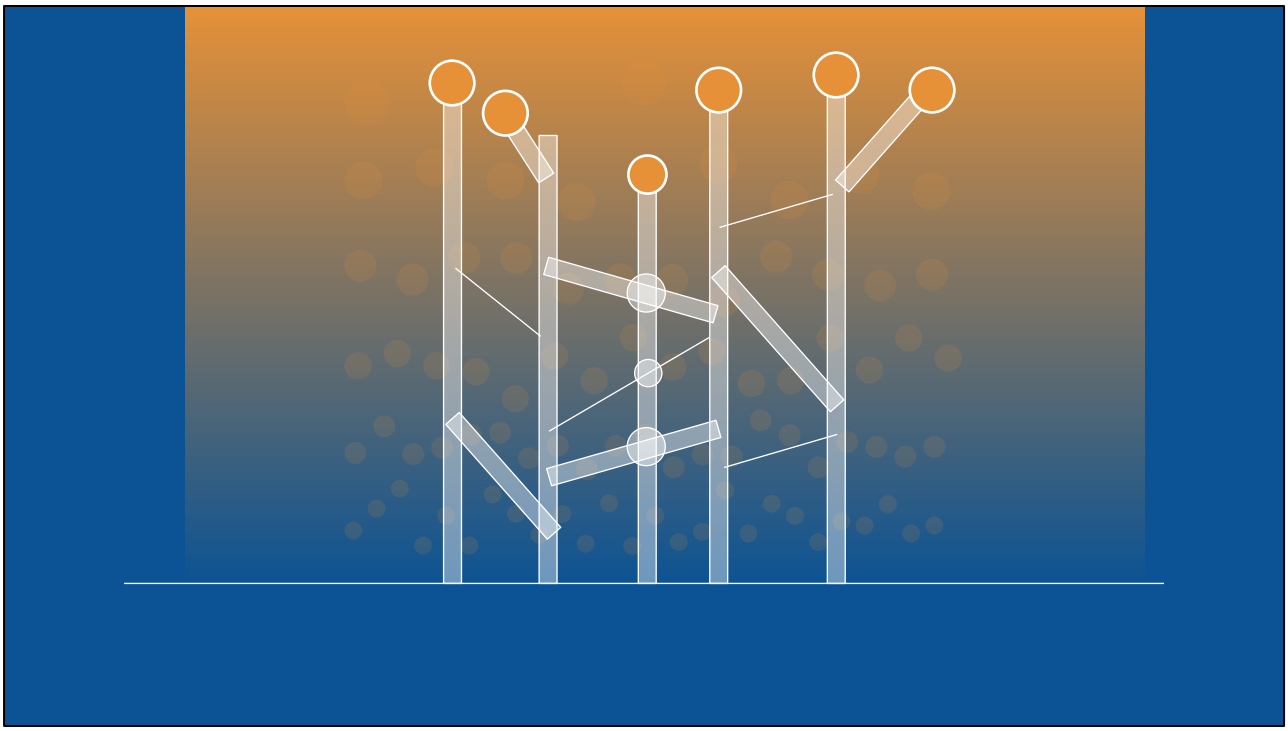
… And then you'll do a bunch more.

This isn't great, but it feels kind of workable.



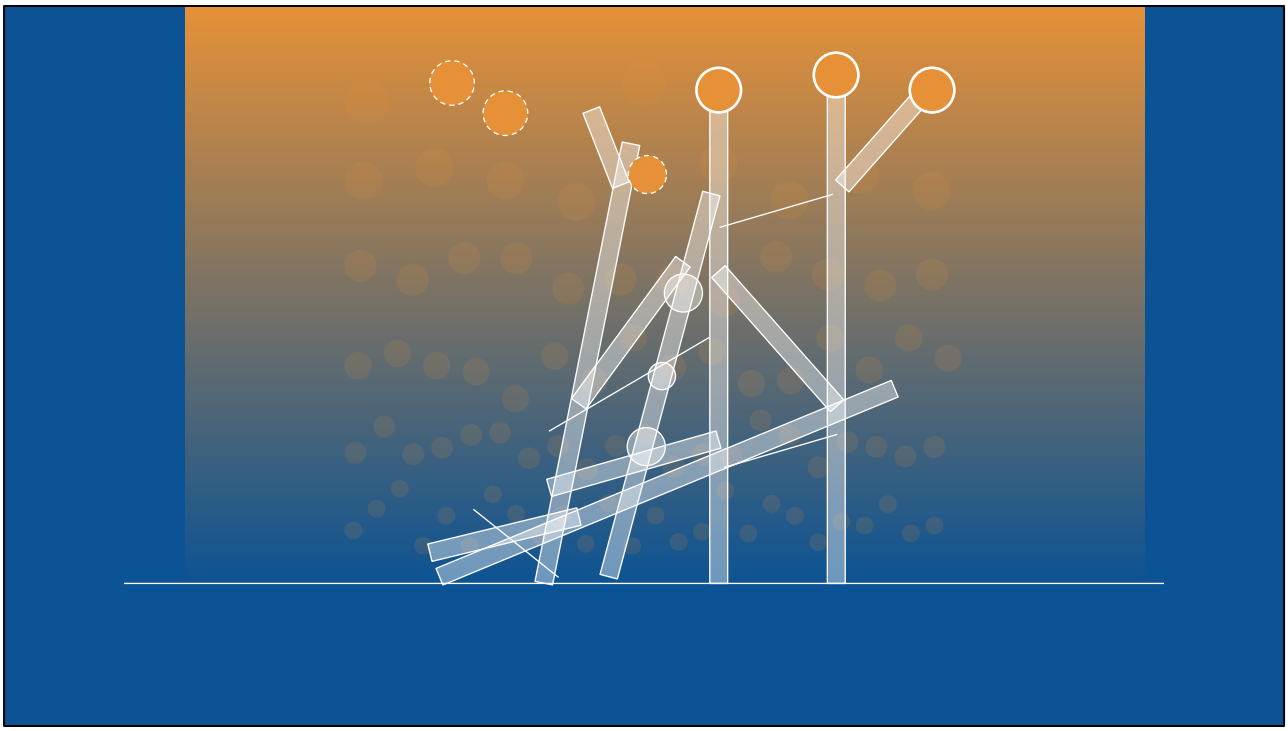
Let’s say you now want to go after this bit of value.

All of those integrations and cross cutting things have created a bit of a thicket. Planning your path will be harder. At each intersection, you have to ask yourself, how does this piece interact with the other existing pieces?



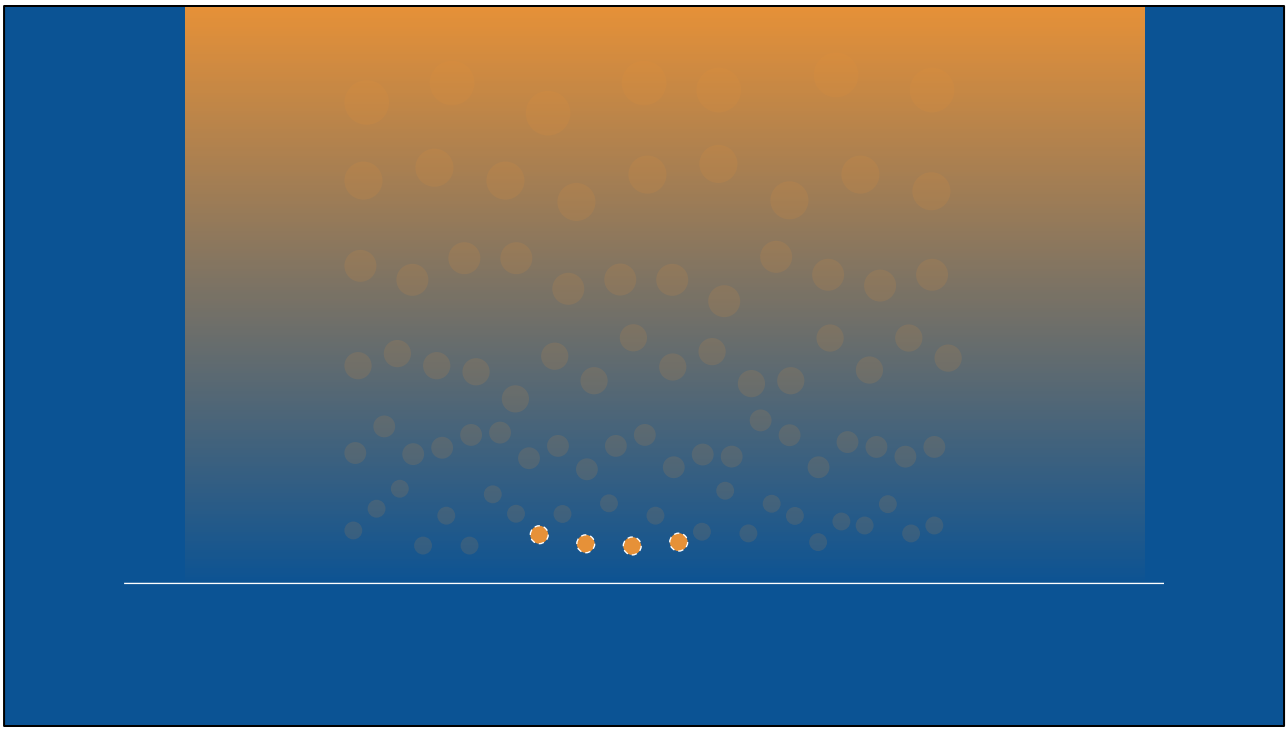
It will take a little longer to build and it will be uglier.

This structure… isn’t great. It’s extremely brittle.



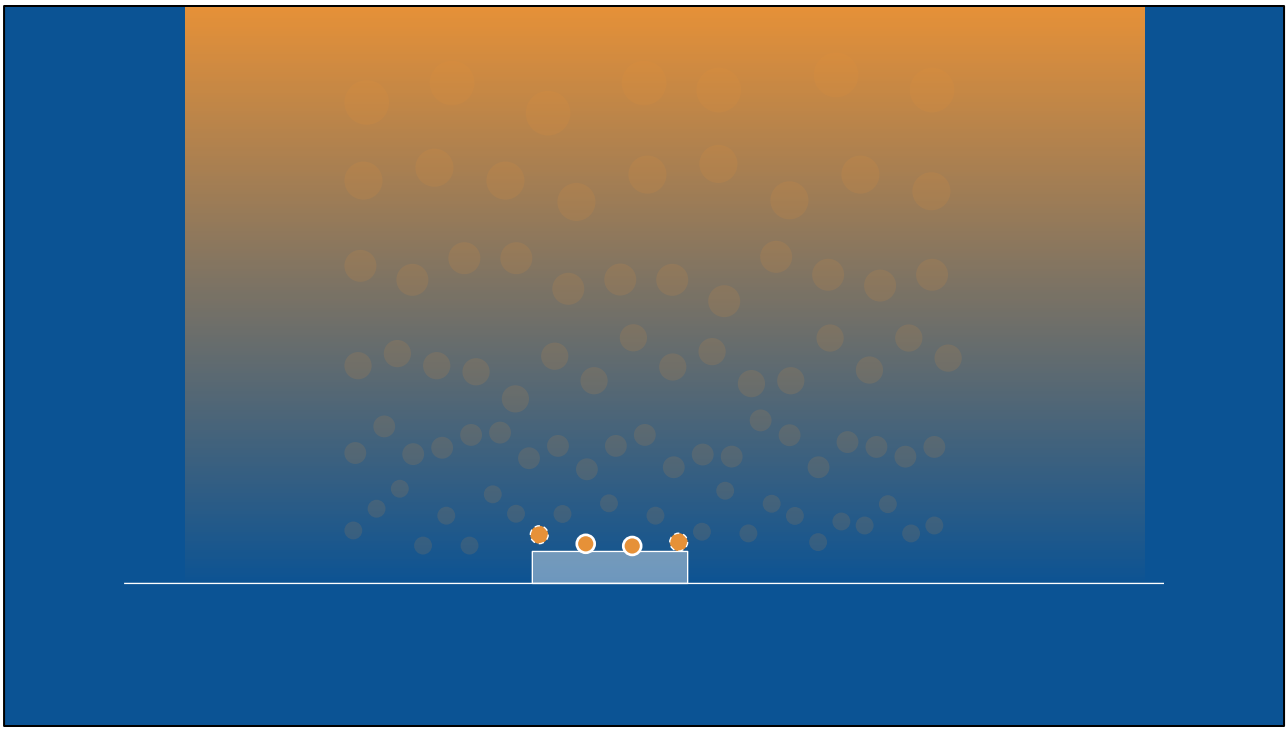
A strong breeze could come over and knock over one of your towers… and do a lot of damage.

This kind of development might be called “vertical”. There’s another way: “horizontal.”



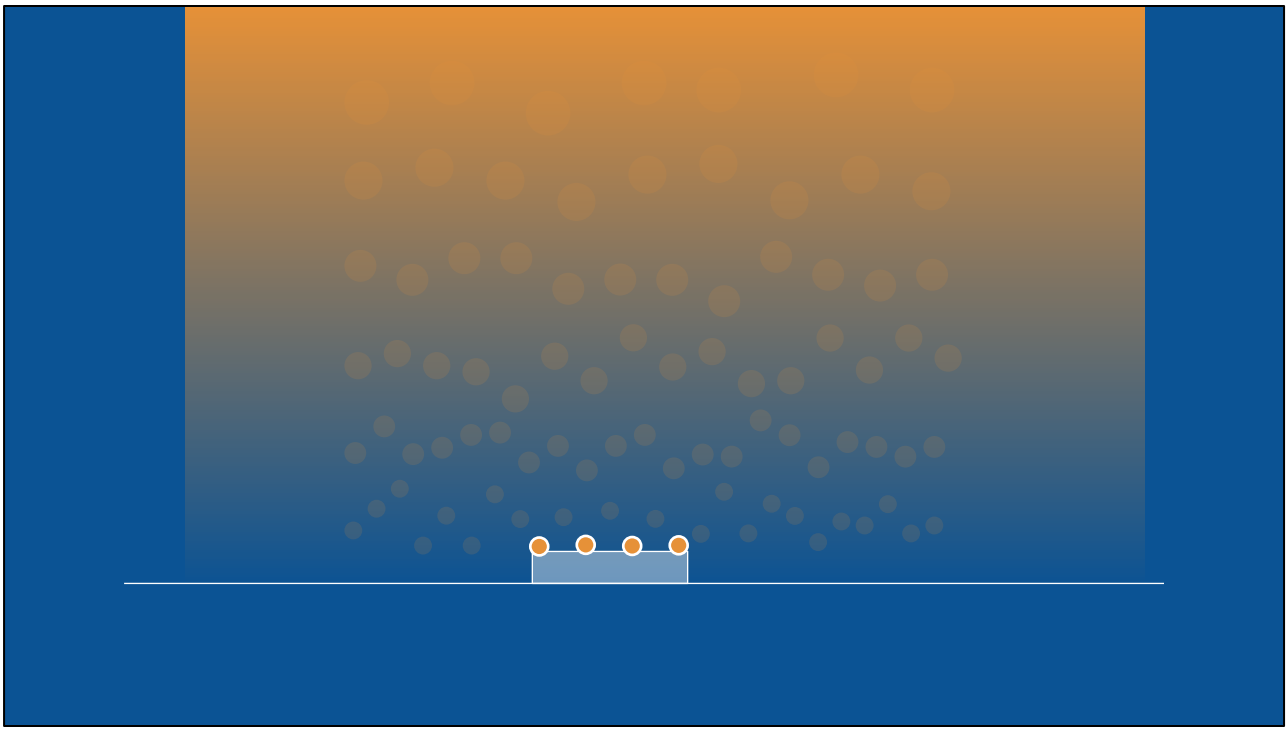
Before, we sighted off the opportunities at the top. They're bigger--but also harder to see clearly.

Instead, sight off the opportunities at the bottom. The ones you can see clearly and are within arm's reach. This is your "**adjacent possible**".



You build something thin and a bit wider than it is tall.

You capture a couple opportunities that you saw. But then something funny happens.



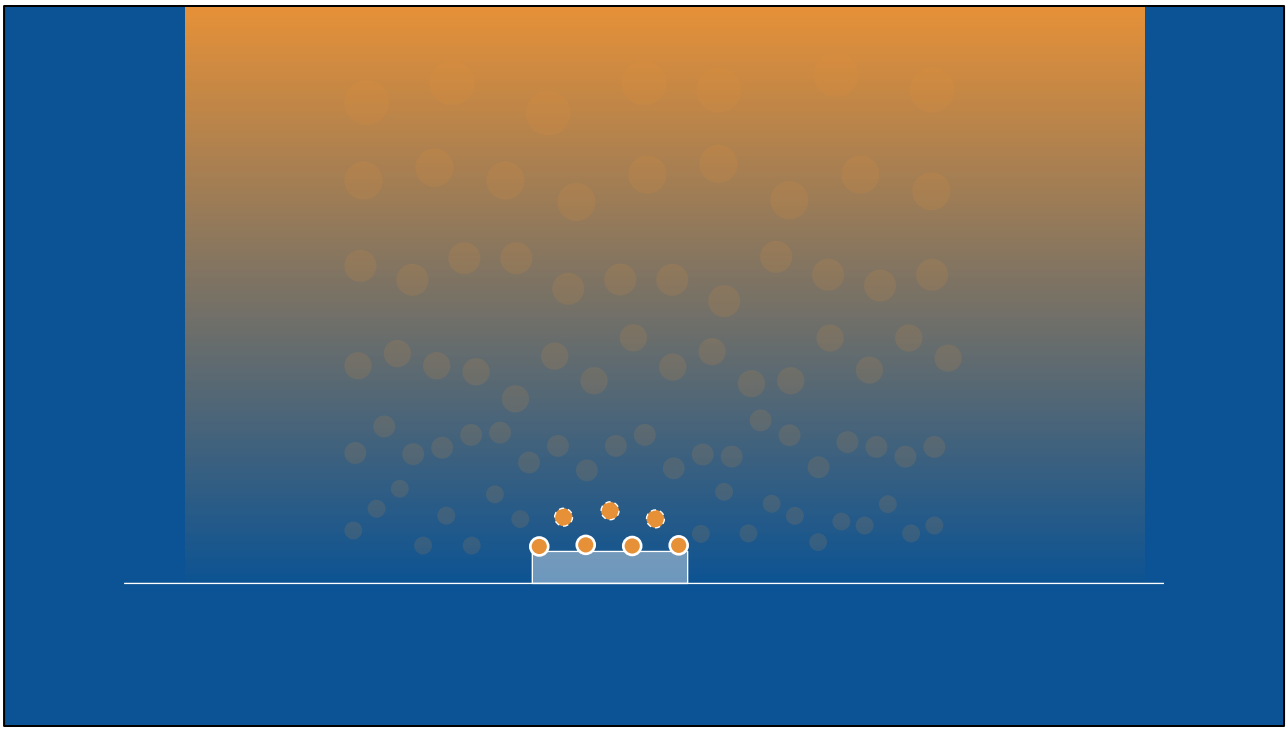
The opportunities shift a little, towards the platform you built. Ones that weren't exactly lined up, line *themselves* up.

Remember that **gravitational pull** between the platform and ecosystem?

The opportunities are not static things or things flitting about randomly. They are people, developers, companies, who have *goals*, and who can make decisions based on what happens around them. The actions you take as a platform provider affect the ecosystem, and vice versa. That's that coevolution thing.

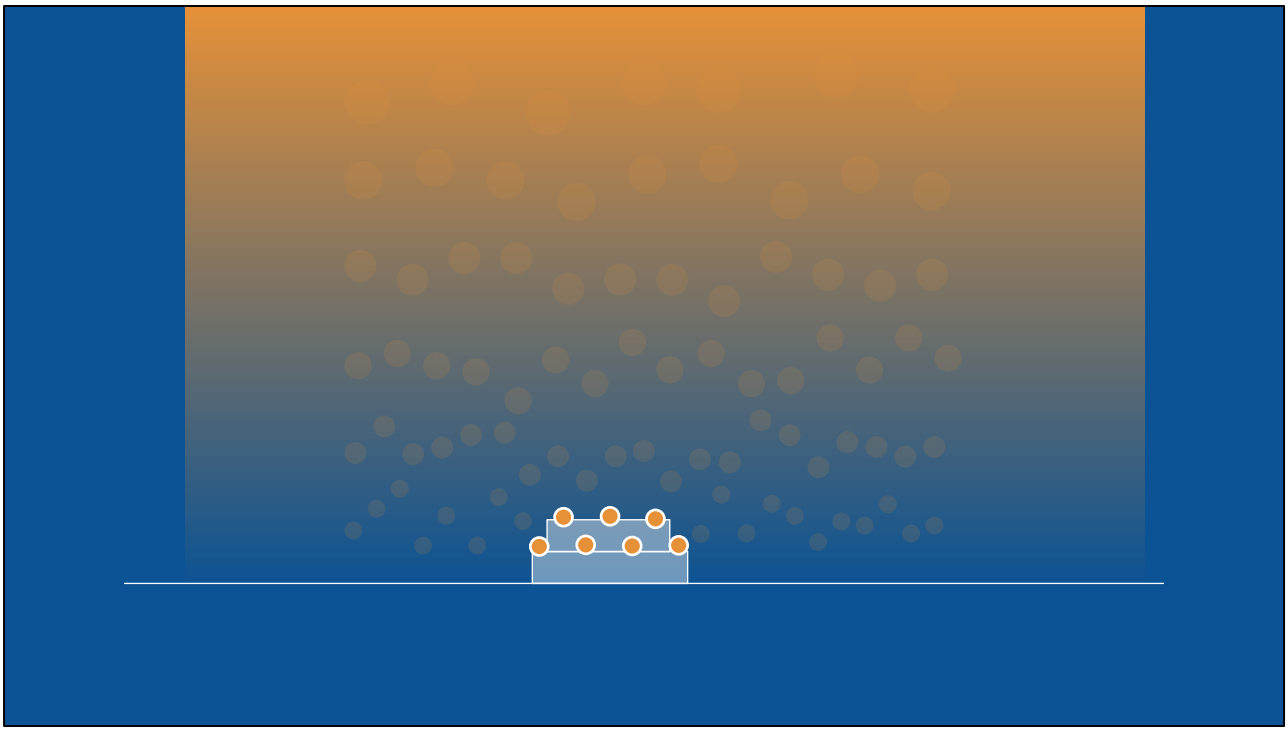
A platform that has more users is strictly better. Your questions are more likely to be answered on Stack Overflow. There's more likely to be useful tutorials. Off-the-shelf libraries are more likely to work with it automatically. The platform is less likely to be deprecated or go out of business. There are less likely to be bugs because the code is battle-tested. End-users will be attracted, making for more attractive opportunities for 3Ps. There's strength in numbers.

The smaller the opportunity (the size of the 3P), the faster they can shift. And the closer it is to a part of your platform, the stronger the gravitational pull.



So now what? Well, you have a solid foundation that's *already* starting to pay the bills. You're on very strong footing.

And now you can see a bit further, too.

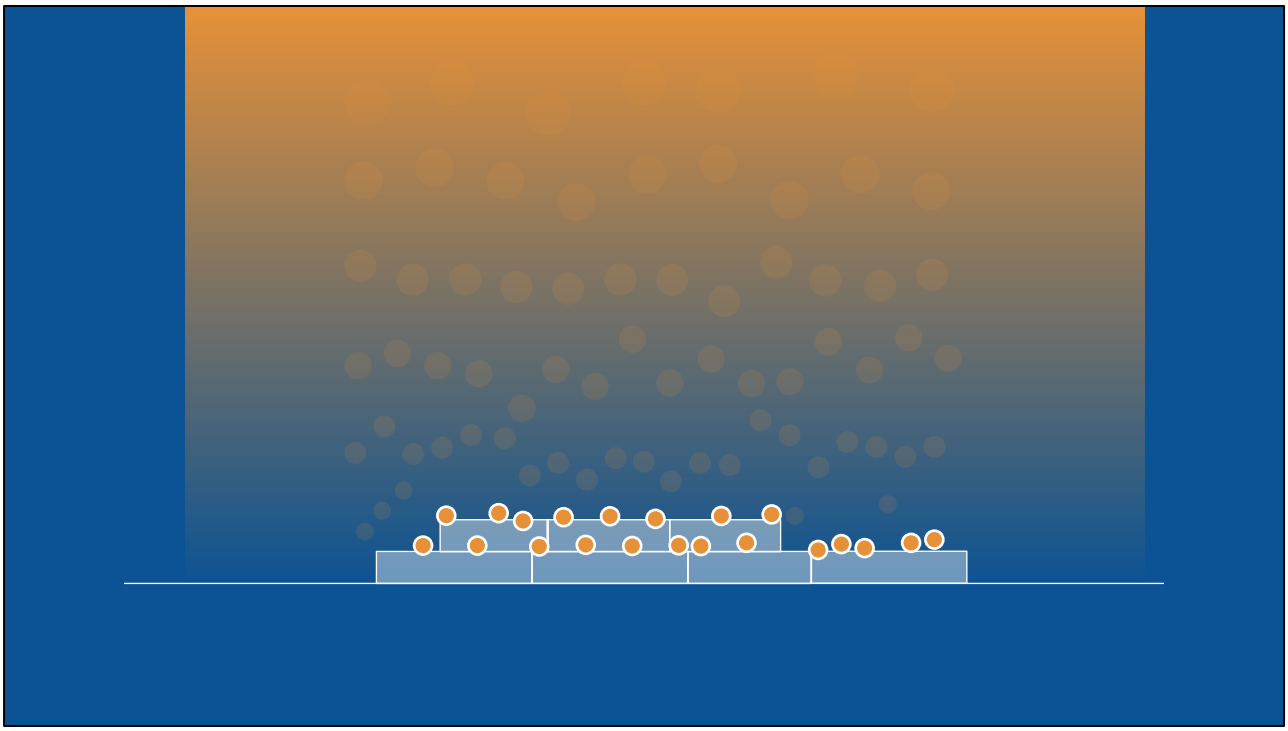


So, you do it again!

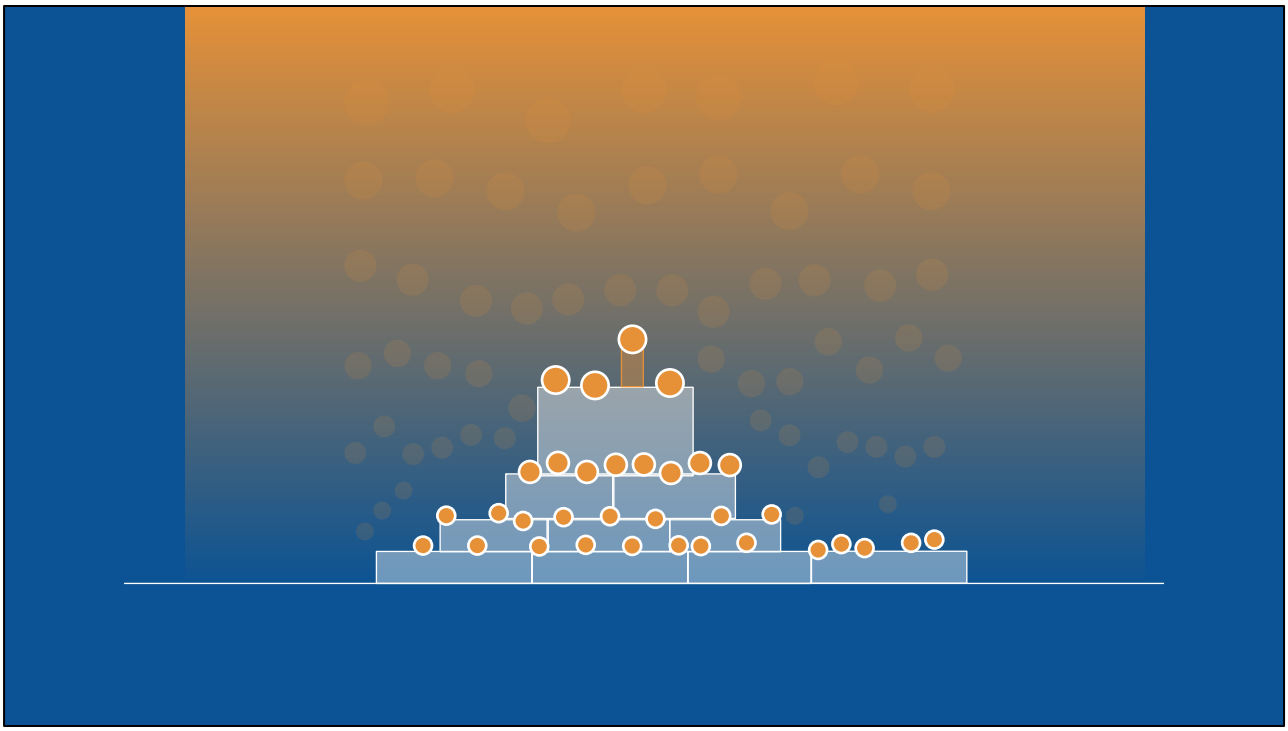
You have a strong stable footing, so as you reach, you aren't sticking your neck out. It's like having a stable tripod stance in rock climbing, taking minimal risk as you reach one of your limbs out to reach farther.

And as you build, that gravitational field gets stronger and stronger as your overall mass grows.

You're still in a really stable position. So you just… do it again! You look for the brightest opportunities within arm's reach and just do it, without overthinking!



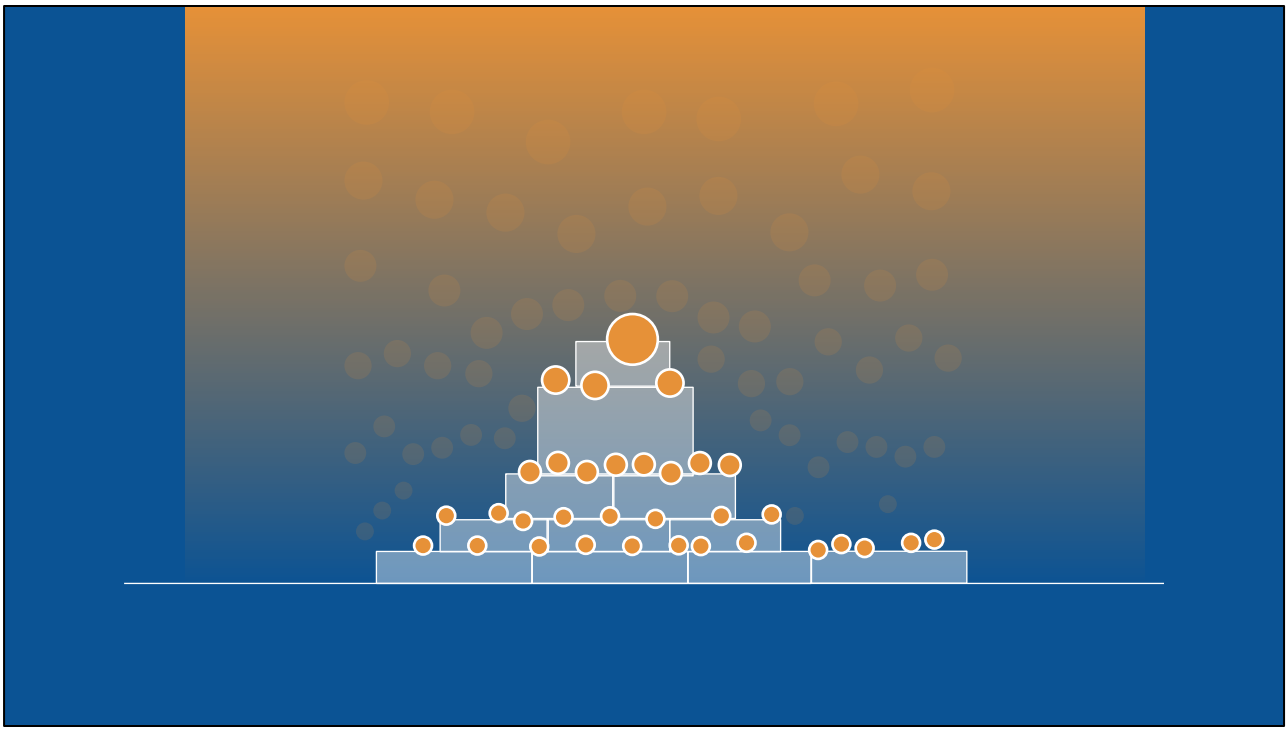
And again!



And again!

You’re getting a ton of momentum now, exerting a large gravitational pull. People are hearing about your platform, and being drawn to a thriving community.

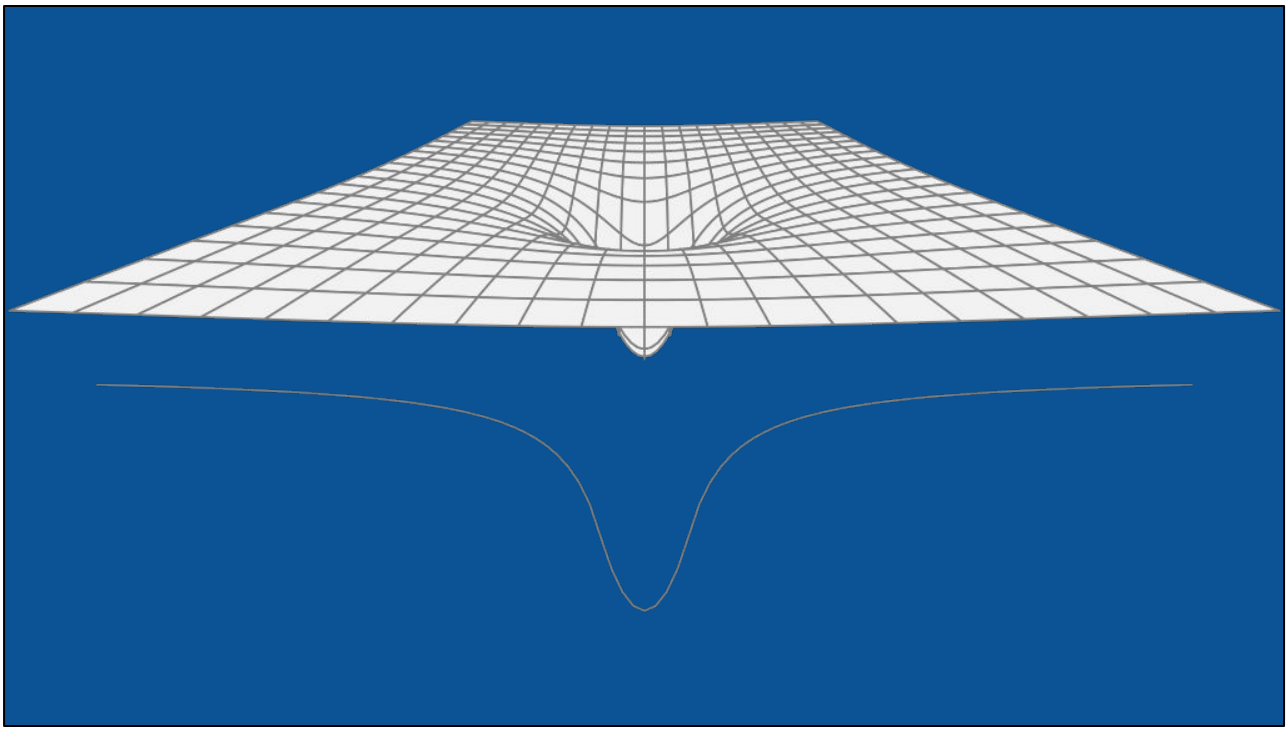
You notice something funny--someone is using your platform despite it not being a good fit. They’re crawling through broken glass to you use it. That means something interesting is going on, so you investigate.



After investigating you realize that there's a very powerful incentive for them, which is why they're crawling through broken glass. You build up your platform to it and the opportunity is even bigger than you thought.

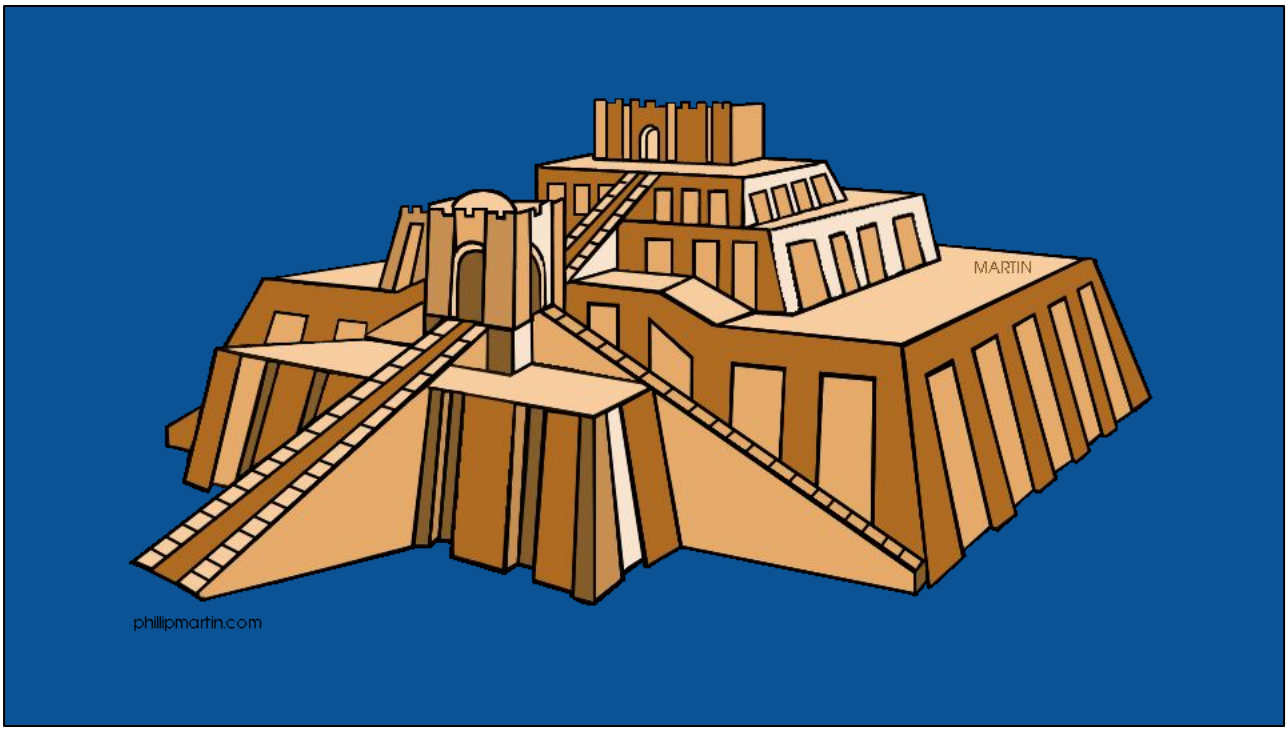
Your gravitational pull is getting larger and larger at every step, and at no time are you taking too much risk. You're continuously viable.

If you keep doing this, you can work yourself up to even the largest customers smoothly, with a stronger and stronger pull of gravity as you go.



You can create a **gravity well**.

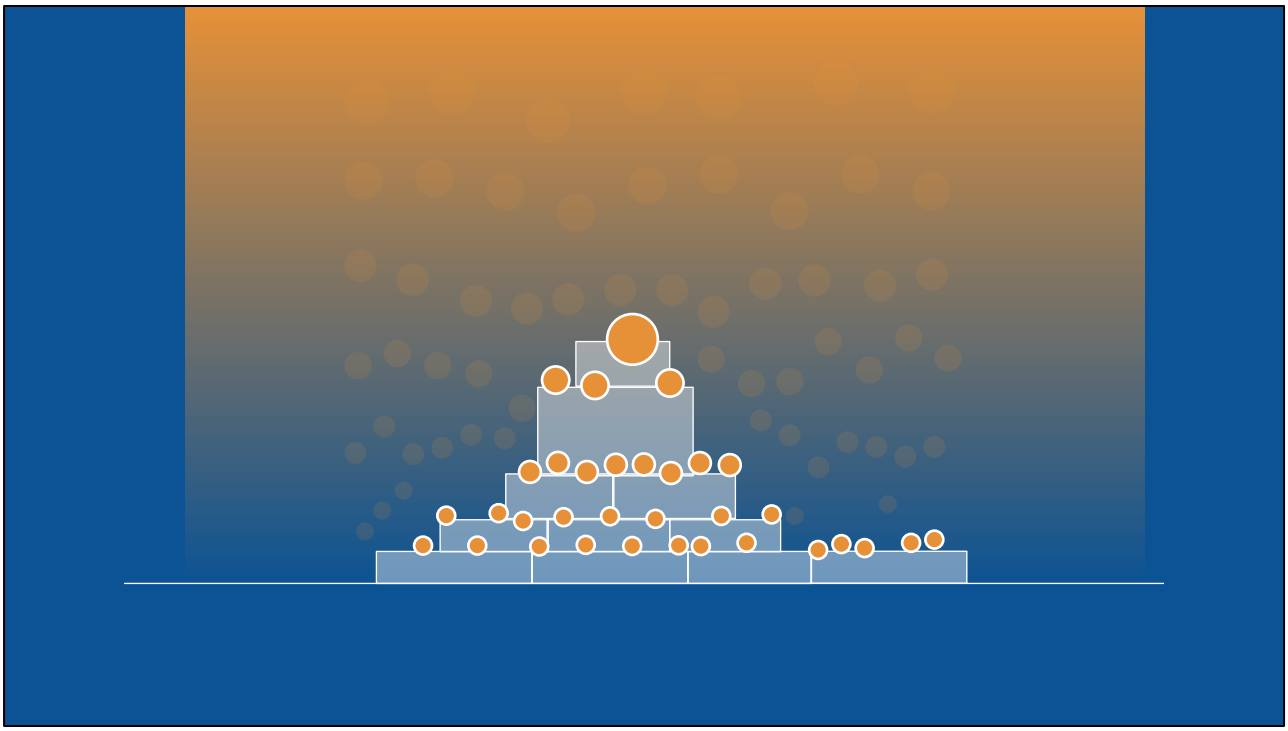
Image source: https://commons.wikimedia.org/wiki/File:Gravity_well_plot.svg



We've built an incredibly strong a stable structure--a **ziggurat**.

> This shape is not just very stable. This slow accretion of layers also leads to a strong foundation, in the same way that trees that grow more slowly have stronger wood.

Image source: https://mesopotamia.mrdonn.org/ziggurats.html



Here’s another thing about platforms that’s magical. As long as each part of the platform is “pulling its weight” and contributing to the overall success of the whole platform, then each part is paying for itself.

As you add more surface area, you expand your adjacent possible, and you strictly accumulate it. You increase your **luck surface area**--your positive exposure to being “in the right place at the right time”.

That means that every time step you have the same or more revenue-generating surface area. Integrating that forward through time, you get a compounding loop of returns.

Compounding loops are the magical things we search out at all costs because they are so valuable. Arguably all of strategy is finding these compounding loops.

And by growing a platform, you've gotten one, with very little risk… just a little bit of patience.

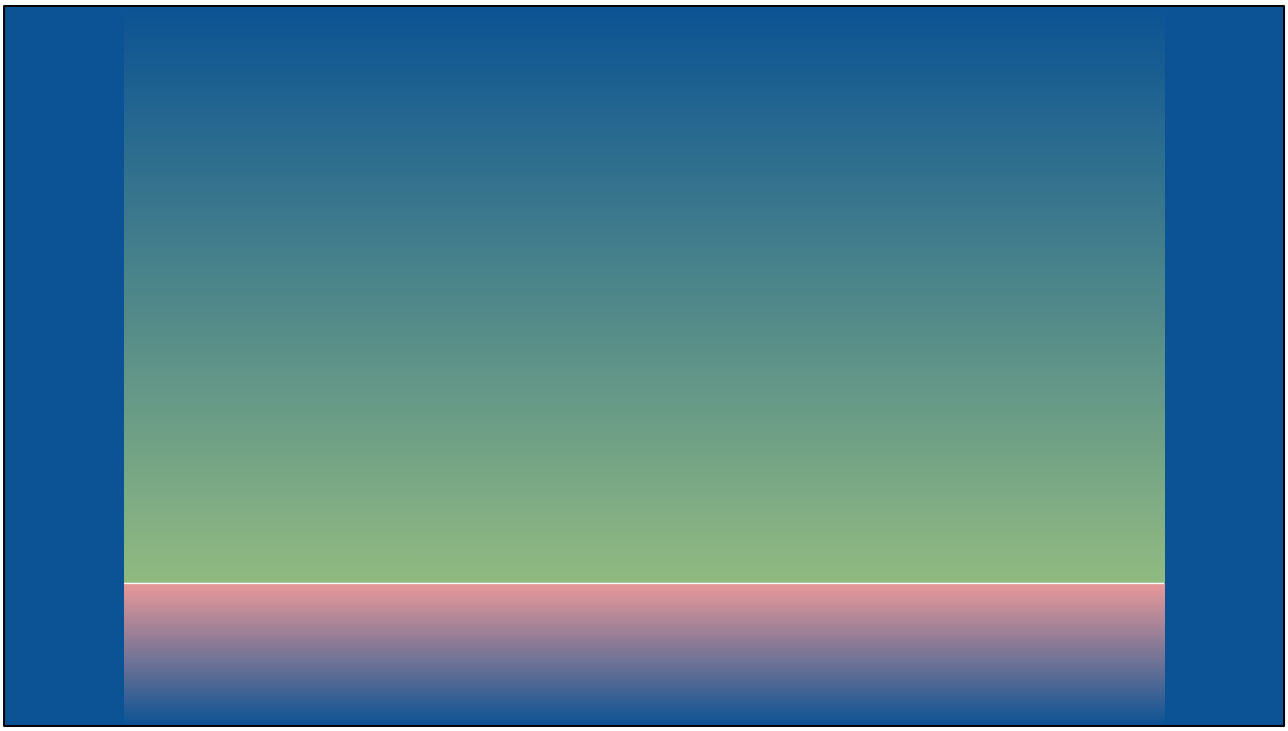
Now this approach as outlined isn't perfect. The layers are too thin, and we're only accumulating, we're never rationalizing.

But this is the intuition of why the platform approach is so powerful.

Let's talk about a series of rules of thumb when growing platforms.

First, everything else in this playbook assumes that you have capped your downside.

You must be playing in a “safe” region where **game-over** scenarios are rare. You also will want to put out any existential-threat short-term fires that are burning.



You can think of mapping out go/no-go regions. Here, the areas below the “bedrock” are off limits, but everything else is OK.

The red regions are dangerous regions. They might be things like “cannibalizes a more important business” or “violates the security model, making abuse more likely, which would be an existential risk to the platform.” For example, in the web platform security model, the bedrock is the same origin security model. These are things that are often very hard to change and require mapping out early.

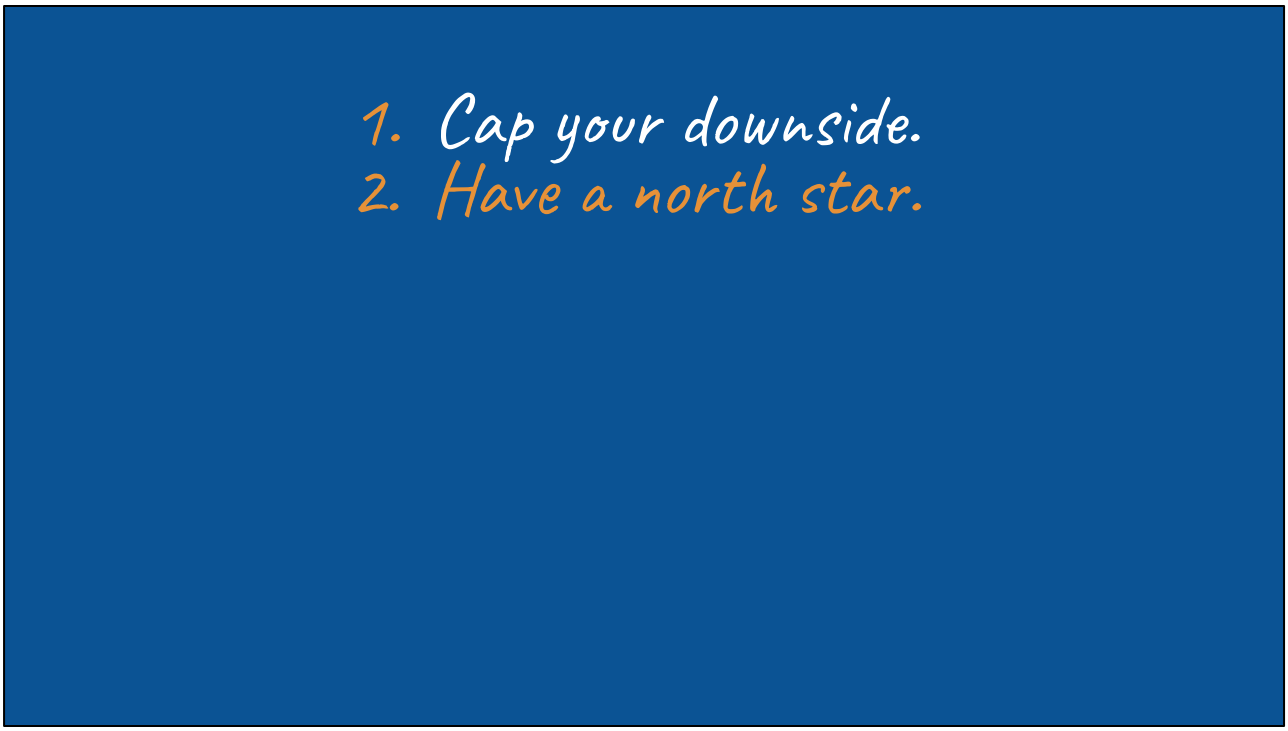
The green region is everything else. When in doubt, consider it to be a green region.

Sometimes it’s simple like this; sometimes there might be a more complex map or red and green, more like a valley of safe areas.

When I say, “cap your downside” I don’t mean “make there be zero downside”--that would lock down way too much and close off many good ideas. Instead, just try to minimize the chance that there’s a harmful surprise to users. Harmful == something so bad that that developer or user will not use that thing again, and will tell their friends to not use it, either. Surprise == something they weren’t expecting. There are ways to make the feature available but still cap downside. For example, if the feature is hard to find or use, then only the most motivated users will find it. That means it will be a self-selecting population of savvy users that is more resilient to mistakes or rough edges, meaning they are less likely to be surprised when something bad happens, but you still have the upside of seeing if that feature has product-market fit.

> And remember that this balance will shift as the ecosystem evolves. For example, at the beginning there might not be much incentive for attackers, but the more people who use the ecosystem, the more incentive there is for attackers, requiring stronger mitigations.

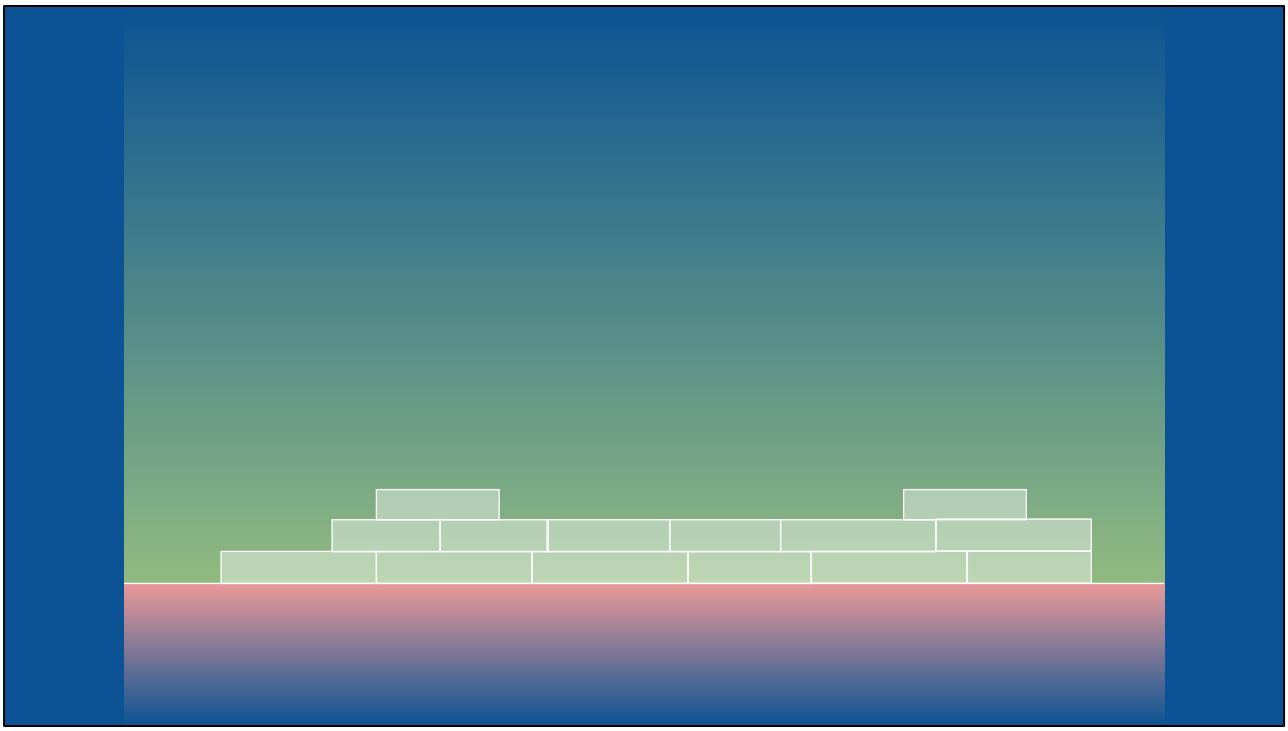
> By the way, another way your downside might manifest is if the platforms you yourself rely on (e.g. the OS vendor, the app store you rely on for distribution, the physical chips, etc) start trying to limit what you can do or extract more value.



Next, you should have *some* kind of long-term north star: a vision that is big and sustainable and makes enough money to at least support itself, if not much more.

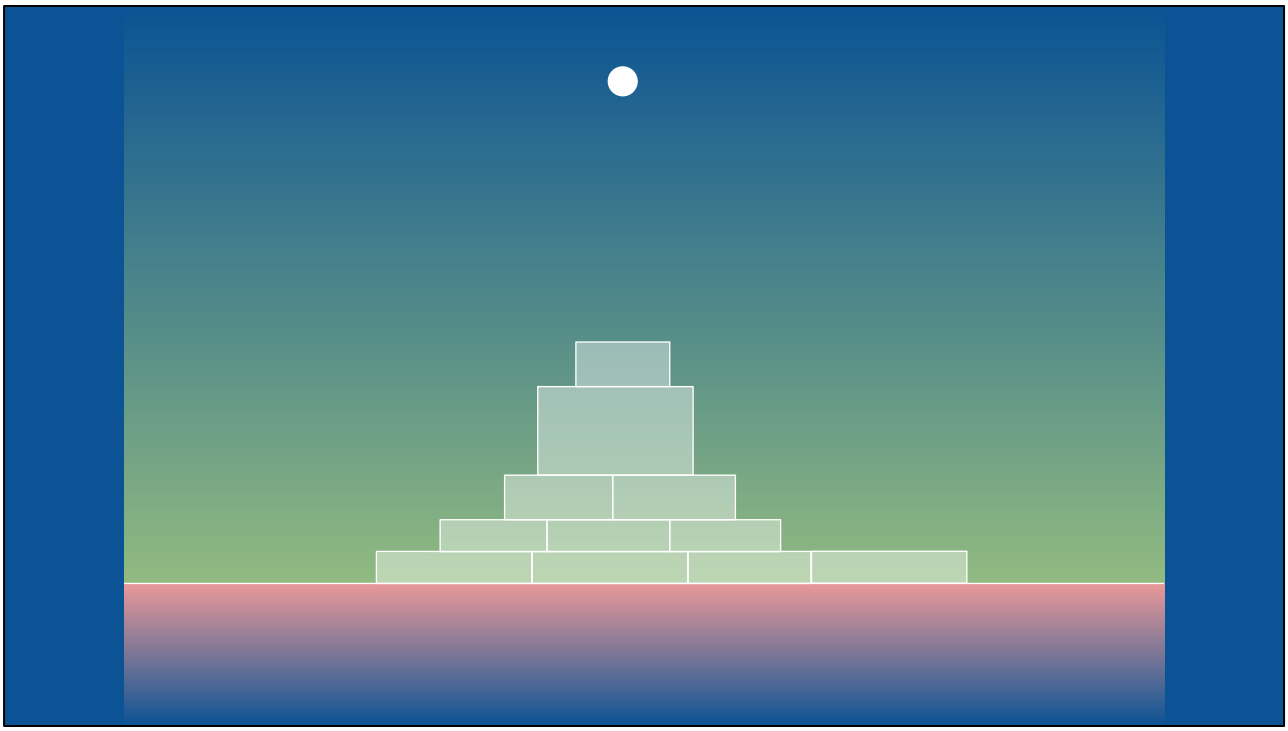
It doesn't have to be great.

Just something to break ties towards so things converge.



If you don’t have a north star and just chase whatever short-term opportunity is best, you might end up building something that doesn’t add up to more than the sum of its parts, or accumulate to reach anywhere.

Remember, higher regions are more valuable.

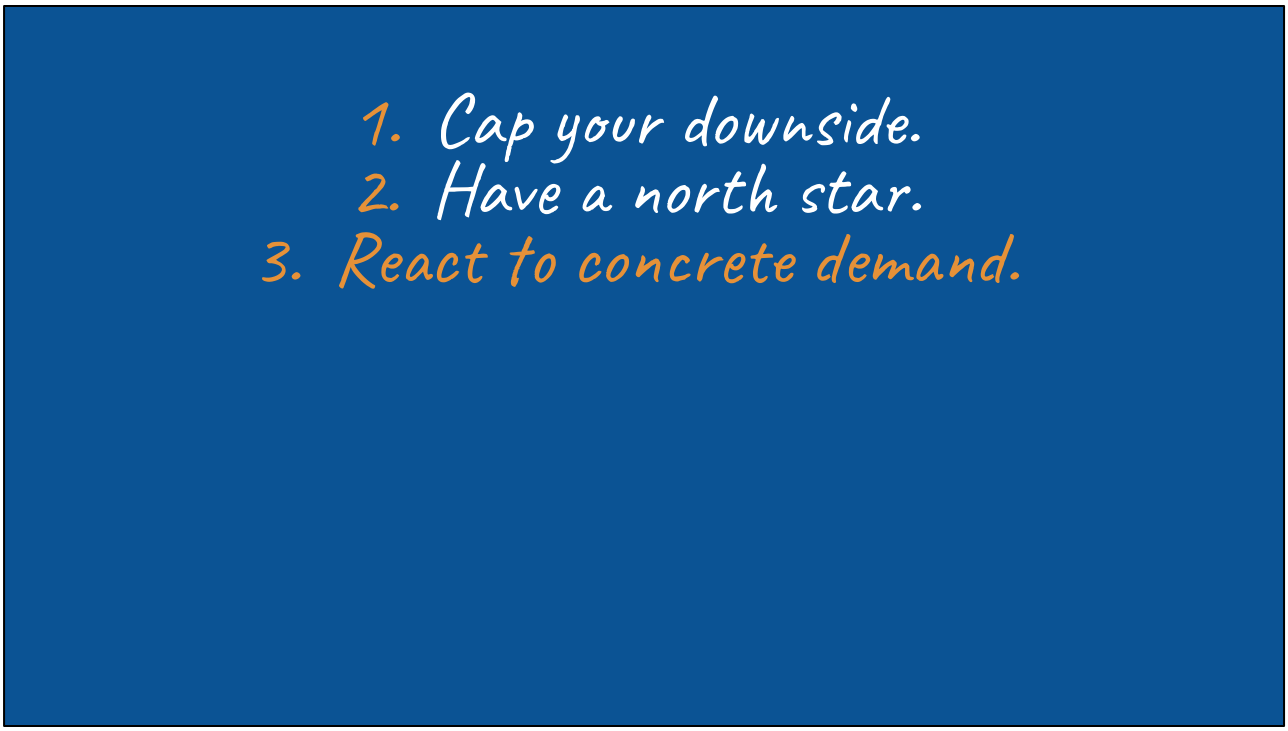


A north-star gives everyone something to sight off of, to break ties. When there's a choice of zigging or zagging, everyone is more likely to go in the same direction, accumulating to something bigger than the sum of its parts.

The north star has to be far off in the future--something that will remain true for 5 or 10 years at least.

> For example, it's generally harder to go from "head" to "tail" customers than vice versa. If you know you'll want to eventually support the long-tail, you'll have to either start there, or make sure at every step that your security model will ultimately allow you to get there.

Having some kind of conceptual north star gives you a point of reference to ask, of any given change, "what's the likelihood I regret doing this 5 years from now?"



The next rule of thumb is to respond to concrete, not speculative, demand.

Don't try to guess where you think there's useful demand. Just listen to your most vocal customers. If there's demand, there will be someone banging down the door asking for it.

If you pitch something to a developer, and they say "hmmm let us know when it's out of beta, maybe we'll try it then?", that doesn't count as strong demand.

You should be able to list concrete names of customers who will use this feature as soon as it ships. You want at least three examples of concrete demand for a feature to convince yourself it's not an overly-specialized thing for one customer.

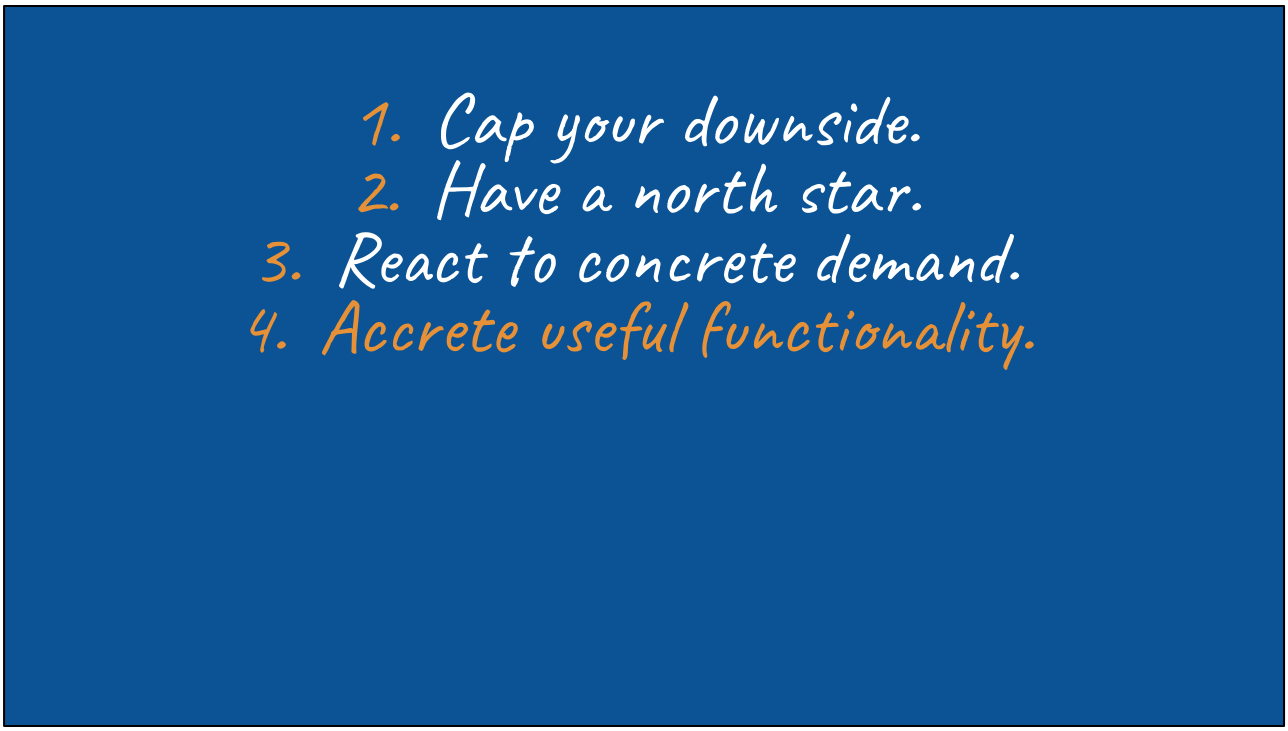
Of course, don't respond to concrete demand that takes you in a direction away from your north star or takes you into dangerous territory.

You can think of this to some degree like paving cowpaths.

> There's often demand even where you think there won't be. That is, the minimum viable product for a platform feature is often way simpler than you think it is. Sometimes you can throw together something very lightweight, even a cardboard cutout of a platform feature, and then put a doorbell on it so that if a developer happens to be interested, they'll let you know. (In practice this will be something like an interest sign-up form.) Standing up the cardboard cutout is very cheap, and you

don’t have to waste energy going and finding them--if they exist, they’ll find you and let you know. It’s a great way to retain basically free option value.

Don’t overthink it. You don’t have to do a ton of UXR to find these opportunities, just ask your developers directly.

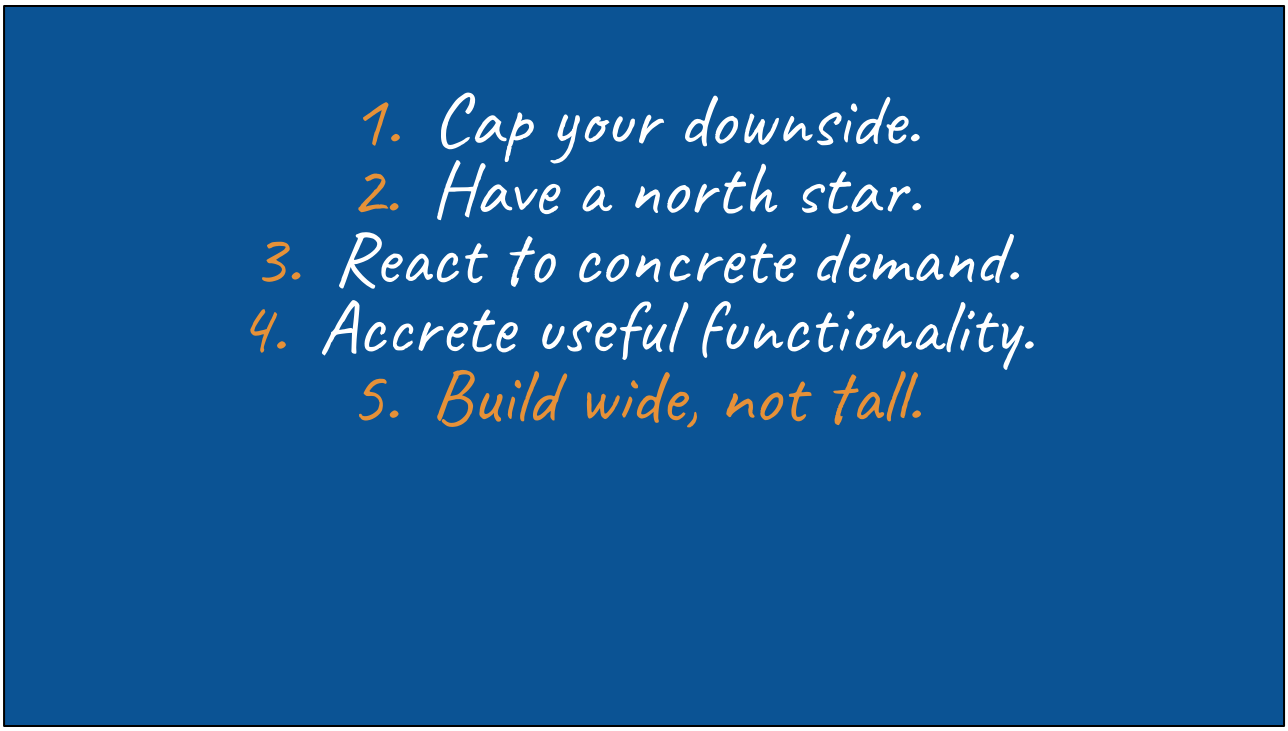


When you find demand that is possible for you to do, just do it. Write the infrastructure and code to support it. Think of accreting layers of known-to-be-useful functionality.

> You'll know it's useful if you're responding to concrete demand. Unused infrastructure is a liability--expensive to build, just sitting there rotting.

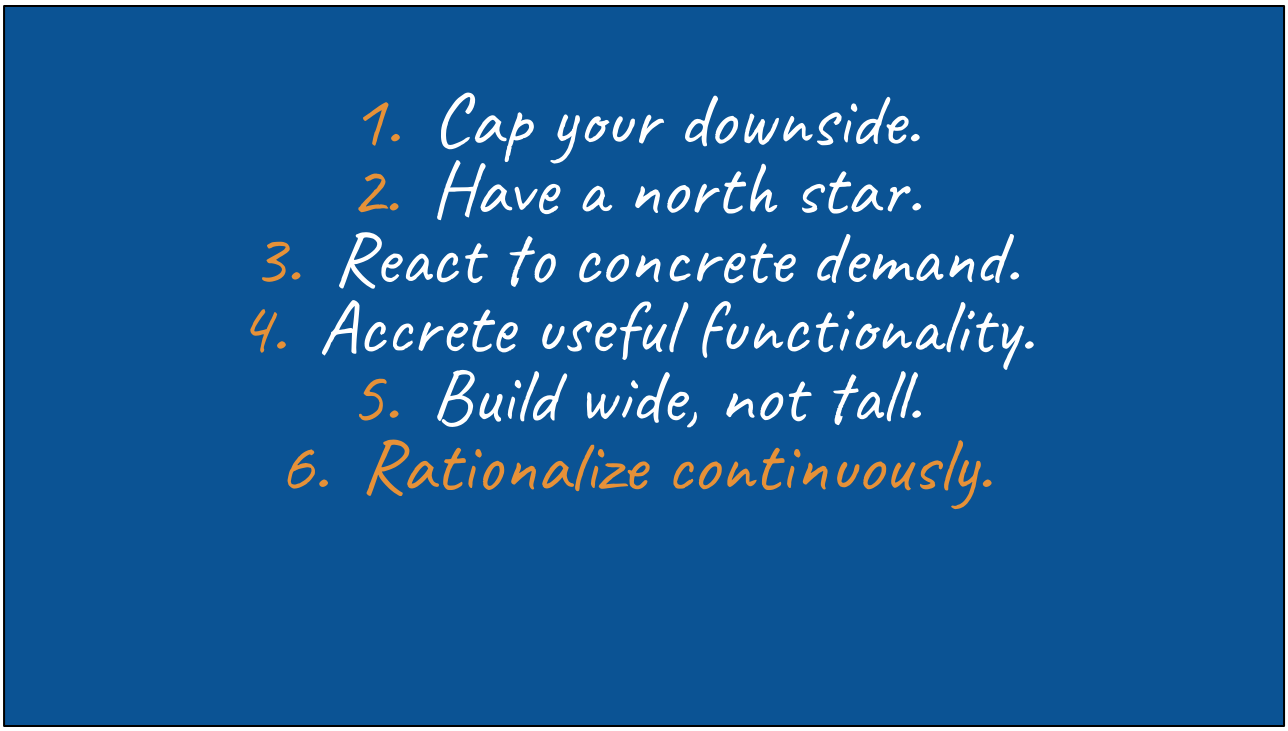
> Sometimes the conditions will change in your ecosystem and something that wasn't useful now becomes useful (perhaps for a purpose you hadn't anticipated). It's hard to figure out what things will be truly useful in a given context too far in the future--that's why it's best to respond to concrete demand.

If you won't actively regret having made the decision to add it in the future, then don't overthink it, just add it! The capped downside plus non-zero upside is where the power of this approach comes from.



When laying down useful functionality, prefer things to be wider than they are tall. That will allow you to support neighboring use cases that you didn't even anticipate ahead of time. Tall things support a narrower use case and are more likely to "miss" their intended use.

Of course, it's very easy to over-generalize too early, creating expensive infrastructure that just goes unused and then rots. This rule of thumb is more about things generally being *slightly* wider than they are tall.



Don't focus on making everything you accrete be perfect at the moment you create it.

Generalizing things and making them durable is really, really expensive. If you aren't 100% sure that you'll want it 5 years from now, then it might be wasted effort, and all of that time at the whiteboard might cause you to miss the opportunity. At the frontier of your platform it's OK to be a bit messier, a bit more speculative, and faster. It's impossible to build a feature perfectly in a vacuum--it's only possible to coevolve it with feedback from real-world usage.

… But then you need to rationalize it afterwards so you don't accumulate a bunch of hard-to-maintain stuff. If code has been there for a couple of years, or supports more than a handful of customers, it's time to spend effort to rationalize it and make it more generalized and sustainable. If it's not used within a couple of years, deprecate it and prune it away.

When adding new functionality, you want to avoid adding one-offs or special cases… but it's unavoidable in certain cases. So it's not "no one offs", it's "no *long-lived* one-offs"--every one-off should either be removed later if it's not taking off, or generalized/rationalized into the whole to be made more sustainable later.

> Remember that every bit of a platform has to be maintained. If a bit of a platform isn't adding value to the whole edifice, it's a burden on the rest of the platform and should be pruned away. (Note that each piece doesn't have to *directly* create its own value; it can help fortify value creation elsewhere in the platform.)

You should do this clean-up work continuously, not in big-bang changes. Think of it as a continuous tax of 10-40% (depending on how quickly moving that part of the platform is, how established it is, and how long your ‘runway’ is for the platform to start creating value).

This tax will always look like a corner to cut for specific feature teams. It’s easy to convince yourself the debt doesn’t matter: “what specific, concrete user value will this debt reduction unlock?”. But that’s the wrong way to frame the question. Debt reduction makes future work in that space--and adjacent spaces--cheaper, making ideas that would have been too expensive now fit in the adjacent possible and be viable. You don’t necessarily know what those features are, but you do know that if your platform becomes an irrational quagmire that whatever features you do want will be much harder to build.

When a platform is thriving, the platform owner has a lot of power from all of the momentum. It can get tempting to "over-extract" value.

But be careful--over-extracting will change the incentives of the ecosystem. People will start looking for alternatives, or ways to lessen their exposure to the worst-case if you continue to get more greedy.

Generally, the less you extract from an ecosystem, the faster it will naturally grow--and that growth compounds.

It's far, far better to take a 5% cut on a compounding-growth ecosystem than to take a 30% cut on a linear or sub-linear growth ecosystem.

> Even if you aren't being greedy now, the ecosystem might fear that you'll be greedy in the future. You can do things to limit yourself in credible ways so they don't have to fear it as much. For example, if you develop the entire platform entirely in the open on GitHub, the worst case scenario is way better--the community can just fork it. Other approaches include making it extremely easy for customers to export their data to competitors, and making difficult-to-retract public promises about that functionality.

> Another thing to do is make it easy for people to exit, no strings attached… but attach a little "bell" to the exit door so you'll notice if there's a larger-than-usual outflow, or some absolute number is reached, and can investigate early to figure out what's going on before it becomes a serious problem.

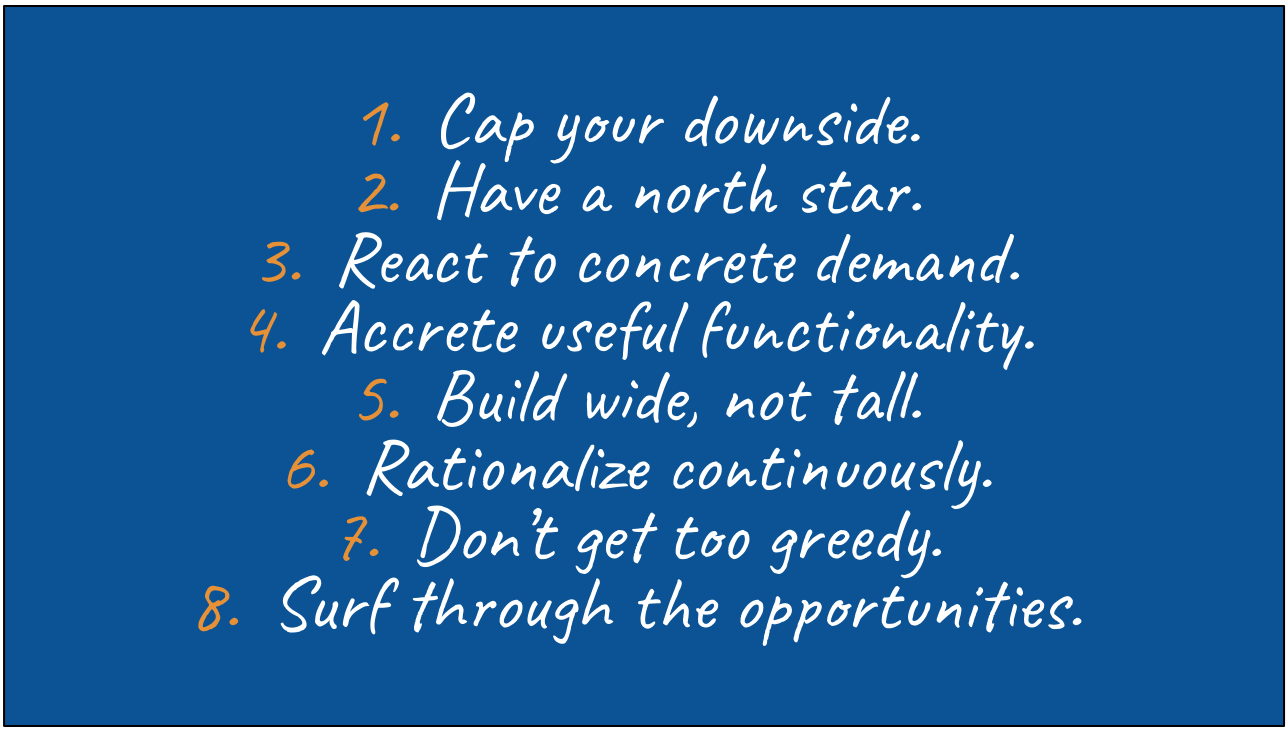
Remember that platforms are living things. If you try to exert too much control, you'll smother it.

Let go of the need to control everything and go with the flow. Let go of the idea that you won't waste effort--you absolutely will! But it's impossible to tell what is wasted effort until after the fact. And it's better to stay flexible.

> Instead of fighting changing conditions, surf them, responding opportunistically and being in the moment.

If you fight the wave, it will capsize you. But if you surf it, you can be propelled by its power.

As long as you've capped your downside, have a north star, and are following the other rules of thumb, it's hard to mess it up too badly.



If you follow these rules of thumb, you can create extremely powerful platforms that generate compounding of value for the platform owner *and* the ecosystem.

This is where we've been.

Growing a platform is one of the most powerful forces in business, and it’s not even hard. It just takes patience and working in a slightly different way.

The results are like magic.

By mastering it you'll become like a wizard. People working in consumer or used to working in the "normal" way will look at you as a bit kooky, but also in awe of your incredible luck and how things just always seem to work out for you.

alex@komoroske.com
https://komoroske.com/writings
@CardsCompendium

More writing available at https://komoroske.com/writings