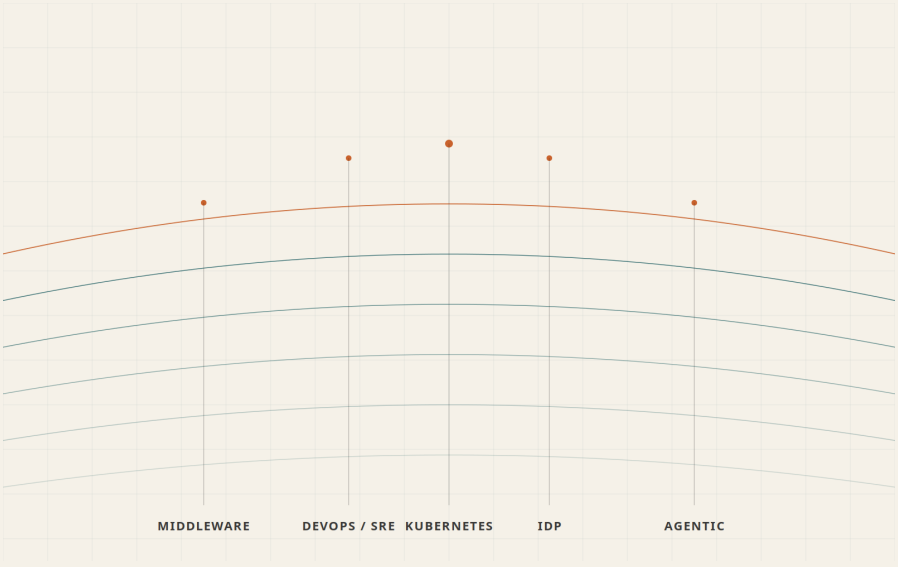
PLATFORM ENGINEERING

What Is Platform Engineering?

Platform engineering follows the application. It always has.



BY ALAN SHIMEL

Founder, CEO & Editor-in-Chief, Techstrong Group
Publisher, PlatformEngineering.com

VOL. 01 · ISSUE 01

TECHSTRONG PUBLISHING
BOCA RATON, FL

About this paper

Every few years the industry settles on a new answer to what platform engineering is, and every few years that answer goes stale. The reason explains the entire discipline: platform engineering follows the application.

This paper is a working thesis, drawn from a decade of covering the discipline at DevOps.com, Cloud Native Now, PlatformEngineering.com, and Techstrong.ai, and informed by ongoing research from The Futurum Group. It is written for CTOs, VPs of engineering, platform team leads, and anyone whose job is to make sense of a category that refuses to hold still.

PUBLISHED
Q3 2026

PUBLISHER
Techstrong Group

READ TIME
18 minutes

INSIDE

The Kubernetes era · Internal developer platforms · Data & observability · AI-native infrastructure · Agentic platforms · Platform Engineering vs DevOps

The short version

Platform engineering is the discipline of building the shared layer that stands between developers and the raw complexity of the technology required to run software. What that layer contains changes constantly. What it does — absorb complexity so the rest of the organization does not have to — does not.

The discipline evolves because the applications it serves evolve. In the last decade alone, it has passed through at least four recognizable forms:

1. The Kubernetes era, when the job was taming container orchestration and its ecosystem.

2. The Internal Developer Platform (IDP) era, when the platform became a product with users, adoption metrics, and a golden path.

3. The data & observability era, when streaming, lakehouses, feature stores, and OpenTelemetry-based telemetry became platform capabilities rather than per-team afterthoughts.

4. The AI & agentic era, when GPU capacity, model serving, vector stores, AI gateways, agent identity, and runtime policy landed on the platform team's desk.

Gartner projects that by 2026, roughly 80% of large software engineering organizations will run dedicated platform teams[1], and Futurum Research finds adoption gaining significant traction across the enterprise[2]. The category is not slowing.

The right way to read the discipline is not to fix it in place. It is to watch it move.

Contents

01 The Quick Answer

Most technical disciplines settle into a stable definition within a decade of acquiring a name. Platform engineering refuses to behave this way. Ask a dozen practitioners what the term means and you will get answers that are all correct, and all incompatible — not because anyone is confused, but because each is describing a different moment in a discipline that has not stopped moving long enough to be photographed.

The platform engineer who held that title in 2021 spent her days taming Kubernetes. She wrote Helm charts, untangled networking policies, and tried to keep a fleet of clusters from drifting into chaos. The platform engineer who holds the title today may spend her time provisioning GPU capacity, standing up model serving, and writing the governance rules that decide which AI agents are allowed to touch production. Same title. Different job. Only a handful of years apart.

That instability is not a flaw in the discipline. It is the discipline. Platform engineering evolves unusually quickly because the thing it serves, the application, evolves unusually quickly. What stays constant through all of it is the mission. The platform exists to stand between the people building software and the raw, unforgiving complexity of the technology required to run it. The platform changes because the applications change. The mission does not.

A search engine would happily quote the technical answer: platform engineering is the discipline of building and operating internal platforms that give developers a self-service, standardized, opinionated path from code to production[3]. It is accurate. It is also incomplete in a way that matters, because it describes platform engineering as though it were a fixed thing. To understand what it actually is, you have to watch it move.

“The platform changes because the applications change. The mission does not.”

That is the frame this paper will use throughout. The rest of the pages ahead trace how the discipline moved — from the middleware era, through Kubernetes, through internal developer platforms, through data and observability, and now into AI and agents — and how each shift redefined what a platform team must build without ever changing what it exists to do.

02 Before Platform Engineering Had a Name

Organizations were building platforms long before anyone thought to call the work platform engineering. The instinct is old. Whenever a group of software teams found themselves solving the same infrastructure problems over and over, someone eventually built a shared layer so the wheel did not have to be reinvented on every project. The layer had different names in different decades — a mainframe abstraction, a middleware bus, a virtualization tier, a private cloud — but the underlying move was always the same.

In the era of enterprise middleware, that layer was the application server and the integration bus, maintained by a central team the rest of the company depended on without quite knowing what they did. Operations teams kept the servers running. Cloud engineering teams, once the cloud arrived, abstracted away the data center. DevOps blurred the line between writing software and running it. Site reliability engineering, born at Google and exported to the industry, applied software discipline to operational problems and treated reliability as a feature to be engineered rather than a hope to be prayed for[4]. Infrastructure as code turned the configuration of entire environments into version-controlled software.

> *"The work existed before the word did. What changed was not the existence of shared infrastructure — it was that the complexity grew large enough to demand a dedicated discipline."*

None of these movements called itself platform engineering, yet each contributed a piece of what the title would eventually describe. Middleware taught the industry that a shared runtime could absorb accidental complexity on behalf of every application above it. Cloud engineering taught it that infrastructure could be treated as a product with an API. DevOps taught it that ownership had to be shared across the wall between writing and running. Site reliability engineering taught it that operations was itself a software problem. Each of these lessons is present in the modern discipline, folded into a single team's remit.

What changed was not the existence of shared infrastructure. What changed was that the complexity of that infrastructure grew large enough to demand a dedicated discipline, and a particular technology arrived that made the demand impossible to ignore.

03 The Kubernetes Era

That technology was Kubernetes. When container orchestration became the default way to run software at scale, it brought a level of power the industry had never had and a level of complexity that few teams were prepared for. Kubernetes did not so much run applications as offer a vast, composable system for describing how applications should run, and it left an enormous amount of decision-making to whoever stood it up. A development team that simply wanted to ship a service suddenly confronted networking policies, ingress controllers, persistent volumes, resource limits, and a configuration language that punished small mistakes with cryptic failures.

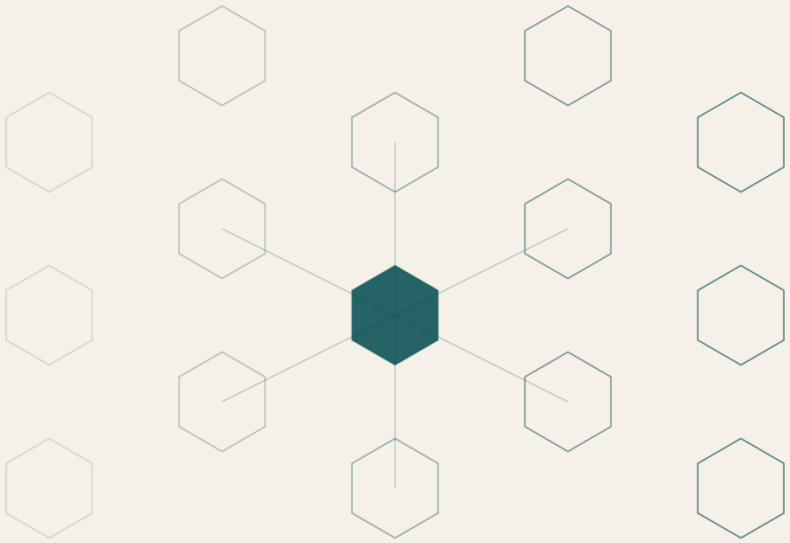


FIG. 01 · A Kubernetes cluster is a composable mesh — power and complexity in equal measure.

Around this complexity, an entire ecosystem assembled. GitOps made the desired state of a system something you declared in a repository and reconciled automatically. Infrastructure as code extended from servers to clusters to the cloud accounts beneath them. Service meshes managed the traffic between services and the security of that traffic. Continuous integration and delivery pipelines carried code from commit to cluster. Secrets management kept credentials out of source control. Policy as code encoded the rules that governed what could be deployed and how. This was the architecture of the moment, and assembling it correctly was a full-time profession[5].

It would have been easy, and many organizations made this mistake, to define the emerging platform team as the group that runs Kubernetes. That framing missed the point entirely. The value of platform engineering in this era was never in operating Kubernetes. It was in abstracting Kubernetes, in building a layer above all that complexity so a developer could deploy a service without learning the internals of a system that took specialists years to master.

The platform team absorbed the complexity so the rest of the engineering organization did not have to. That principle, more than any specific tool, is what the Kubernetes era taught the discipline, and it is the principle that everything afterward would build on. Techstrong's own coverage — from Cloud Native Now to DevOps.com — tracked this shift in real time, chronicling both the tooling frenzy and the slow realization that the tooling was not the point.

"The value of platform engineering in this era was never in operating Kubernetes. It was in abstracting Kubernetes."

That reframing is what let the discipline outgrow the tool that first defined it. The teams that internalized it built abstractions that survived the next wave of change. The teams that did not spent years re-learning the same lesson each time the underlying technology shifted — because operating a technology and abstracting a technology are two different jobs, and only one of them travels well.

By the time the industry began asking what should sit above Kubernetes, the answer had already started to take shape in the teams doing this work. They were building portals, catalogs, and self-service tooling that treated the developer as a customer of the platform. What emerged next was less an invention than a naming of what these teams had quietly been building all along.

The lesson generalizes. Every powerful technology that arrives in engineering does so on the same terms: it grants new capability and demands new complexity in exchange, and the value it creates is unlocked only when someone builds the abstractions that let the rest of the organization use it without becoming specialists in it. Kubernetes was the largest and most visible instance of that pattern in the modern era, but it is a pattern the industry keeps encountering, in different clothes, every few years.

This is why the teams that lived through the Kubernetes era, and did the harder work of building abstractions rather than merely operating clusters, were the ones best prepared for everything that came next. They had internalized the discipline's central move. Everyone else was still learning it.

The evidence is now visible in the shape of the industry itself. The organizations that treat their platform team as a product organization — with roadmaps, users, and a service level to hit — consistently ship faster and adopt new technologies more smoothly than the ones that treat it as an operational function. The distinction is not one of investment. It is one of framing. A platform run as operations is optimized for uptime and cost. A platform run as a product is optimized for the productivity of the developers who depend on it, and productivity, in the long run, is what determines whether the next transition is survivable or not.

04 The Internal Developer Platform Era

The natural consequence of that principle was a shift in how platform teams understood their own work. If the job was to abstract complexity on behalf of developers, then the developers were users, and a thing built for users is a product. This realization moved platform engineering out of the operations mindset and into the product mindset, and it produced the artifact that now sits at the center of the discipline: the internal developer platform[6].

The internal developer platform, usually shortened to IDP, is the curated, self-service layer through which developers interact with everything beneath them. Its purpose is to reduce cognitive load, a phrase that became central to platform engineering because it named the real problem. The modern software stack asks developers to hold an impossible amount of context in their heads, from infrastructure to security to deployment to observability, and most of that context has nothing to do with the business logic they were hired to write.

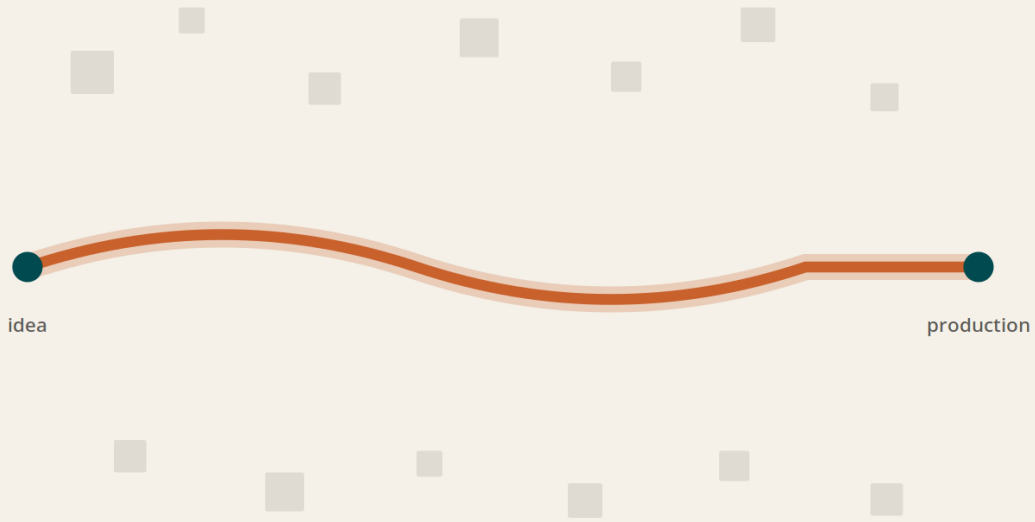


FIG. 02 · The golden path — a well-marked, well-supported route through the complexity.

A good internal platform narrows what a developer must know. It offers golden paths, opinionated and well-supported routes from idea to production that handle the common cases correctly by default[7]. The paved road is a design choice: not the only way to get somewhere, but the way that is fastest, safest, and best supported.

Around and alongside Backstage, commercial platforms emerged: Humanitec built tooling for the orchestration layer beneath the portal[9]. Port offered a portal and software catalog as a product. OpsLevel approached the problem through service maturity and ownership. Mia-Platform packaged the platform as an integrated offering for enterprises. Each took a different route to the same destination.

What unified them was the recognition that the platform had become an internal product with a lifecycle of its own. The platform engineer, almost without noticing, had become a product manager as much as an infrastructure engineer. Success was no longer measured only in uptime. It was measured in adoption — in whether developers chose the paved path because it was genuinely the easiest way to get their work done.

"A platform that engineers route around is a failed platform, no matter how elegant the architecture beneath it."

That single test — do people use it because it is the easiest option — is what separates platforms from mandates. Mandated infrastructure gets endured. Chosen infrastructure gets built on. The teams that treat their internal platform as a product to be earned rather than imposed are the ones whose platforms compound in value over time, and whose developers describe them not as a hurdle but as a head start.

This is where the discipline outgrew its operations origins. Building infrastructure was engineering. Building a product that engineers actually want to use is engineering, plus design, plus research, plus the humility to admit when the paved road is potholed.

Product thinking reshaped how these teams operated day to day. They ran user interviews with the developers who worked next to them. They shipped changes on release cycles rather than as one-time projects. They measured DORA metrics and developer satisfaction alongside SLOs, because a platform nobody enjoys using is a platform whose value quietly leaks out the sides. The platform was no longer a piece of infrastructure to be commissioned. It was a product to be improved, indefinitely.

This was also the moment when platform engineering became a distinct career track. Where earlier practitioners had drifted in from operations or infrastructure, a new generation began entering the discipline on purpose — drawn by the challenge of building tools for other engineers and the leverage that comes from work that multiplies across an entire company.

05 Data Becomes Part of the Platform

While the developer platform was maturing, a parallel evolution was happening one layer over, in the world of data, and for a long time the two were treated as separate concerns. They are not. As applications came to depend on data in motion and data at rest far more than they once did, the infrastructure that moved, stored, and governed that data became another shared platform that someone had to build and maintain.

Streaming systems carried events through organizations in real time. Data lakes stored vast quantities of raw information, and lakehouses emerged to bring the structure of the warehouse to the scale of the lake. Machine learning pipelines turned that data into models, and feature stores gave those pipelines reliable, reusable inputs.

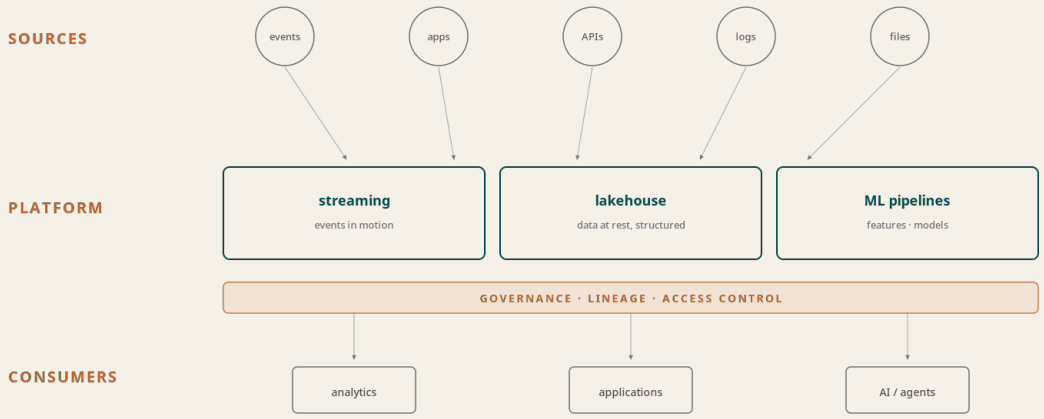


FIG. 03 · The data platform — sources become products, governed end to end.

Data governance grew from a compliance afterthought into a foundational requirement, because shared data infrastructure without governance is a liability waiting to be discovered. The same organizations that had spent years standardizing developer workflows discovered they needed the same discipline for data access, lineage, and quality — and that the tooling had matured enough to make it possible without slowing analytics to a crawl.

Many organizations, looking closely, realized they had been running data platform teams for years without recognizing them as platform engineering. The pattern was identical. A central team built shared infrastructure so the rest of the company could work with data without each group solving the same problems independently. The discipline had simply arrived in the data world under a different name.

06 Observability Matures

Observability followed a similar path, from peripheral tooling to platform capability. For most of its history, monitoring was something operations teams bolted on after the fact, a set of dashboards and alerts that told you whether the lights were still on. As systems became distributed and the number of moving parts exploded, that approach stopped working. You cannot reason about a system of hundreds of services with the tools designed for a handful of servers.

Modern observability rests on three kinds of signal — metrics, logs, and traces — and on the ability to connect them into a coherent picture of how a request actually moved through a system. OpenTelemetry, which graduated to CNCF top-level status in 2026[10], standardized how that telemetry is produced and collected, turning observability from a collection of proprietary silos into a shared capability that could be built into the platform rather than purchased separately for each team.

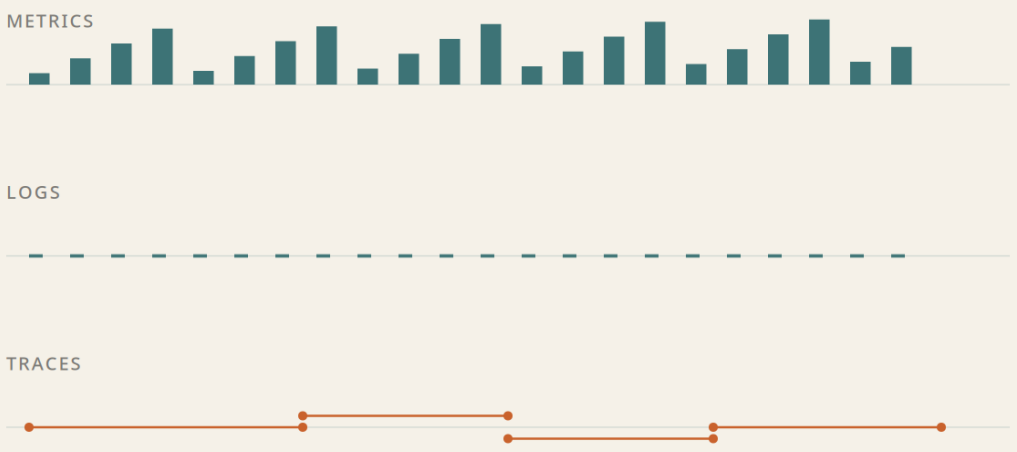


FIG. 04 · Metrics, logs, and traces — three signals, one coherent picture.

The consequence was that observability stopped being a separate purchase decision each team made and started being an expectation the platform delivered. A service deployed through the golden path emits telemetry by default. A dashboard exists before anyone asks. An alert fires against a baseline the team did not have to define alone. Observability, in other words, became infrastructure, in the same way that logging once did.

07 AI Changes Everything the Platform Supports

Then came artificial intelligence, and with it a temptation to declare that AI changes platform engineering. That framing is backward. AI is changing the applications. Platform engineering, as it always has, is changing in response.

This distinction is not pedantic. It is the whole thesis of the discipline restated in the present tense. When applications became AI-native, they developed a new set of needs, and those needs landed on the platform team's desk because that is where infrastructure needs always land. Applications now require GPU infrastructure, which behaves nothing like the commodity compute the platform was built to provision. They require model serving, the specialized work of putting a trained model behind an endpoint and keeping it responsive under load. They require vector databases to store and search the embeddings that power retrieval. They require AI gateways and prompt routing to manage how requests flow to models, and inference infrastructure tuned for a workload profile the previous generation of platforms never anticipated[11].

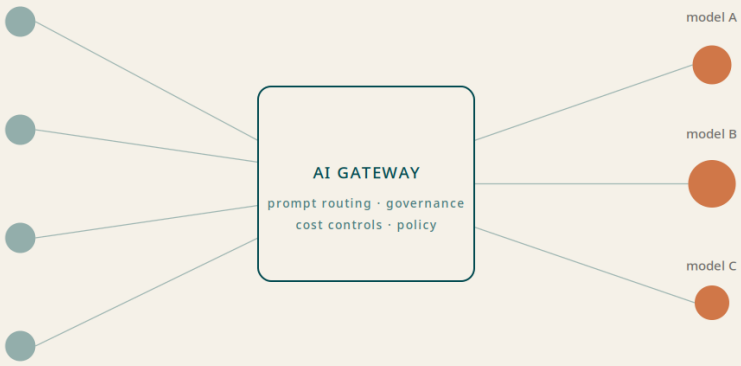


FIG. 05 · The AI gateway — governance, cost controls, and prompt routing become platform capabilities.

AI applications also require a new layer of control. They introduce security concerns that traditional applications do not, because the boundary between data and instruction blurs in ways the industry is still working to understand. They demand governance, so an organization can answer which models are in use, on what data, under what policy. They demand cost controls, because inference at scale can produce a bill that arrives like a weather event[12].

Supporting AI-native applications became the next responsibility of the platform for exactly the same reason that supporting containerized applications became its responsibility a decade earlier. The applications moved, and the platform followed. The specific technologies were new. The pattern was not.

“None of this replaces what platform engineering was doing. It extends it.”

The temptation, in the middle of a technology shift this large, is to imagine that everything before it becomes obsolete. It does not. The Kubernetes-era platform did not throw away what came before it — it added a new layer on top. The IDP did not replace Kubernetes — it abstracted it. The AI-native platform does the same to the layers beneath it.

Each era’s platform absorbs the previous one and adds what the applications now demand. The GPU cluster runs on infrastructure that runs on Kubernetes that runs on the cloud. The AI gateway sits in front of models that were deployed through the same CI/CD pipeline the platform team built five years ago. The discipline compounds. It does not restart.

Which is why the teams that struggle in the AI transition are usually the ones that skipped the earlier work. Without a platform underneath, adding AI-native capabilities is not an extension — it is a rebuild, one whose bill comes due at the worst possible moment.

Conversely, the teams that arrive at this moment with a mature platform underneath treat AI as one more capability to integrate rather than a crisis to survive. They already know how to provision compute, manage secrets, enforce policy, and observe distributed systems. Adding GPU pools, model catalogs, retrieval pipelines, and inference gateways to that foundation is engineering work. It is not existential.

This gap between the prepared and the unprepared is widening. It is why the phrase "AI-ready platform" has begun to appear in analyst reports and board memos: because whether a company can adopt AI at any meaningful scale increasingly depends on infrastructure decisions made years before AI arrived.

08 Agentic Platforms

The newest stage of this evolution is still taking shape, and it may prove the most consequential yet. Applications are no longer content to respond to requests. They increasingly act, reason through multi-step problems, call tools, and pursue goals with a degree of autonomy the industry has started to call agentic[13]. When software begins to behave this way, the platform's responsibilities expand again, and in directions that look less like infrastructure and more like the management of a workforce.

An agent needs an identity, so the platform can know who or what is acting and hold it accountable. It needs memory, managed and bounded, so it can carry context without becoming unpredictable. It needs guardrails that constrain what it is permitted to do, and controlled tool access that determines which systems it can reach. It needs governance and auditability, so every action an agent takes can be traced and explained after the fact — which becomes essential the moment agents touch anything that matters[14].

FIG. 06 · Governing an agent — identity, memory, guardrails, and audit as first-class platform concerns.

Agents also need orchestration to coordinate multiple agents working together, and runtime policy that can intervene while an agent is operating rather than only before it starts. Platform engineering is becoming the discipline that deploys, secures, observes, and governs these agents, applying the same instinct it has always had: to build the shared layer that makes a powerful and complex technology safe to use at scale.

There is a second, recursive turn to this stage. Platform engineering is not only learning to support AI agents. It is beginning to employ them. The operational work platform teams have always done — diagnosis, remediation, the routine toil of keeping systems healthy — is increasingly being handed to agents the platform team supervises, automating its own operations with the same technology it is learning to govern[15].

"The discipline that builds the platform for everyone else is quietly building one for itself."

This inversion is worth pausing on. For most of its history, platform engineering was the team that carried the pager so other teams could sleep. As agents take on more of that toil, the question shifts from "who handles the incident?" to "who governs the systems that handle the incident?" The answer, still, is the platform team — but the job has moved up a layer of abstraction, from operator to steward.

It is a shift with real implications for how these teams are staffed and measured. The successful platform team of the next decade will look less like a group of infrastructure specialists and more like a small product organization overseeing a fleet of intelligent systems. The work is not less technical. It is differently technical.

None of this arrives all at once, and the transition will be uneven. But the direction is clear enough. Every platform team is now, in some sense, building two platforms at once: the one that serves developers, and the one that manages the agents which serve them both.

What makes this stage so consequential is not the technology itself. It is the change in what the platform is expected to guarantee. For most of its history, the platform team's promise was availability: the system will be up, the pipeline will run, the deploy will succeed. In the agentic era, the promise is behavior. The system will do what it is permitted to do, no more and no less — and it will do so with a trail of evidence a regulator or an executive can inspect. That is a different kind of engineering, and it changes what "reliability" means from the ground up.

09 Platform Engineering vs DevOps

No discussion of platform engineering is complete without addressing its relationship to DevOps, a relationship too often framed as rivalry. They are not competitors. They are different kinds of answers to different kinds of questions, and they reinforce each other[16].

DevOps was, at its heart, a cultural movement. It set out to dismantle the wall between the people who wrote software and the people who ran it, to replace handoffs and blame with shared ownership and shared incentives. But culture alone does not scale cleanly. When every team is expected to own its own operations, every team ends up rebuilding the same pipelines, making the same mistakes, and carrying a cognitive load that grows heavier as the underlying technology grows more complex.

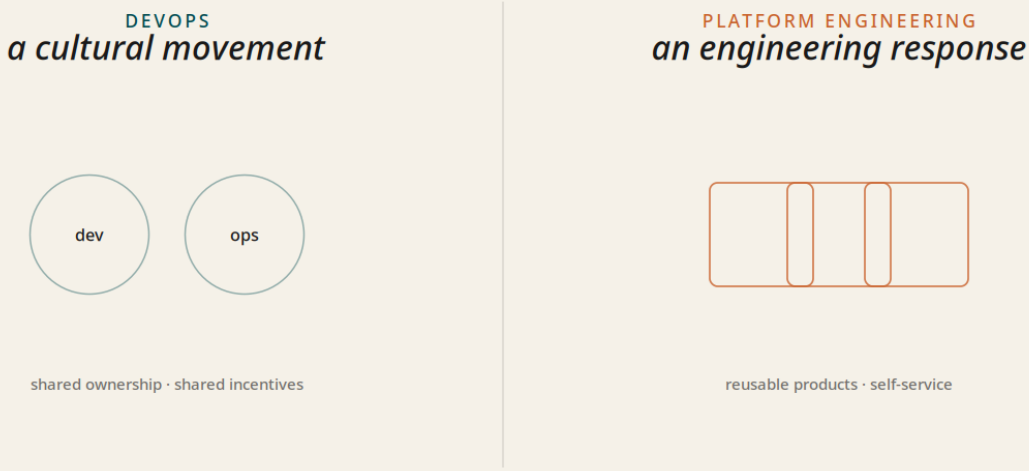


FIG. 07 · DevOps changed how organizations build software. Platform engineering builds the products that make that change scale.

Platform engineering is the engineering response to that strain. It takes the DevOps principle of shared ownership and makes it sustainable by building the reusable products that let teams own their software without each of them having to become infrastructure specialists. One is primarily cultural. One is primarily engineering. The healthiest organizations run both, and would struggle to say where one ends and the other begins.

The rivalry framing does damage in both directions. Organizations that treat platform engineering as a replacement for DevOps end up with well-built platforms that no team feels ownership of. Organizations that treat DevOps as a rejection of platform engineering end up with dozens of teams each rebuilding the same pipelines in slightly different ways. Neither state is healthy. The pattern that works is DevOps culture plus platform engineering practice, running together.

10 The Future

Predicting the future of platform engineering is a fool's errand if you try to predict the technology, and a fairly safe bet if you predict only the pattern. The specific shape of the next platform will be determined by the next generation of applications, and no one can say with confidence what those applications will look like.

What can be said with confidence is that they will look different from today's, that their difference will create new infrastructure needs, and that those needs will land on the platform team. The discipline will evolve because it has always evolved, and it will evolve in whatever direction the applications go[17].

"Anyone who tells you precisely how is guessing. Anyone who tells you that it will keep changing is simply reading the history."

The teams that thrive across these shifts share one habit: they treat their platform as a living product, not a finished project. They budget for the next abstraction the way other engineering organizations budget for the next feature. They read the industry not for tools to adopt but for signals about where the applications are heading, because that is where their work is heading, one step behind.

If there is a durable playbook, it is this. Build for the applications you have, but architect for the applications you can already see coming. Absorb complexity so that developers do not have to. Measure success in adoption, not uptime. Treat the platform as a product with users, not a service with tickets. Assume that whatever is stable today will be table stakes tomorrow — and that whatever feels experimental today may well be the platform in five years.

The specifics will keep changing. That is not a failure of the discipline. That is the discipline.

It is worth naming what this means for the people doing the work. Careers in platform engineering are, by their nature, careers of continuous re-learning. The abstractions that made someone an expert five years ago are ceilings today, and the tools they will use five years from now are being invented right now. The practitioners who thrive in this discipline learn to hold their expertise loosely — deeply respected, seldom re-used verbatim, always in service of what the applications are becoming.

11 So, What Is Platform Engineering?

Which returns us to the question we started with. The technical answer, the one a search engine would happily quote, is true and useful and incomplete. The fuller answer is that platform engineering was never really about Kubernetes, though for a few years it looked like it was. It was not about internal developer platforms, though they remain its defining artifact. It is not about AI, however much AI dominates the present moment. Those are all moments in time, snapshots of a discipline caught in one of its many forms.

Platform engineering exists to remove the complexity that stands between the people building software and the technology required to deliver it. That is the whole of it. As software evolves, the complexity moves, and the platform moves to meet it. As the platform evolves, the people who build it evolve with it, learning new tools and taking on new responsibilities while doing, in essence, the same job they have always done.

This is why the answer to what platform engineering is can never be permanent, and why every attempt to fix it in place will be outdated within a few years. The discipline is defined by its relationship to something that will not hold still. Every previous attempt to draw a hard boundary around what the platform team does has held only until the next generation of applications arrived and moved the boundary somewhere else.

The teams doing this work today are, in an important sense, doing the same work that operations teams did in the 1990s and cloud engineers did in the 2010s. The names change. The technologies change. What does not change is the essential act of building the shared layer that makes powerful, complex technology usable by the rest of the organization — and then rebuilding it, again, when the technology moves.

Platform engineering follows the application. It always has. It almost certainly always will.

ABOUT THE AUTHOR

Alan Shimel is Founder, CEO & Editor-in-Chief of Techstrong Group and publisher of PlatformEngineering.com, DevOps.com, Cloud Native Now, Security Boulevard, and Techstrong.ai. Co-founder of the DevOps Institute.

Sources & Further Reading

01 Gartner: 80% of large software engineering organizations expected to have dedicated platform engineering teams by 2026.
Gartner Top 10 Strategic Technology Trends 2024 (PDF)

02 Futurum Research: Platform engineering adoption gains significant traction across the enterprise.
PlatformEngineering.com — Futurum Research survey

03 CNCF Platforms Working Group white paper: Platforms for Cloud-Native Computing.
CNCF TAG App Delivery — Platforms WG

04 Site Reliability Engineering: How Google Runs Production Systems (Google/O'Reilly).
Google SRE Book

05 Techstrong / DevOps.com — Kubernetes-era platform coverage and analysis.
DevOps.com — Blogs

06 CNCF Platform Engineering Maturity Model.
CNCF — Platform Engineering Maturity Model

07 Team Topologies & the Internal Developer Platform: reducing cognitive load through golden paths.
Team Topologies — Key Concepts

08 Backstage: an open framework for building developer portals (Spotify, donated to CNCF).
backstage.io

09 Humanitec — State of Platform Engineering Report, Volume 3.
Humanitec — State of Platform Engineering

10 CNCF announces OpenTelemetry graduation as the de facto observability standard.
CNCF — OpenTelemetry graduation

11 Techstrong.ai — Google Cloud's new AI agents for data scientists and engineers.
Techstrong.ai — Google Cloud AI agents

12 Decoding the Gartner Hype Cycle for Platform Engineering 2026 — FinOps for Agentic AI and IDPs as governance layer.
TrueFoundry — Gartner Hype Cycle 2026

13 Techstrong.ai — AI Agents are Becoming Distributed Systems.
Techstrong.ai — Agents as distributed systems

14 Techstrong.ai — Stop Treating Agents Like Chatbots: The Platform Blueprint for Production.
Techstrong.ai — Platform blueprint for agents

15 Techstrong TV — Platform Engineering Pay Day: Why It's Headed Toward a $40B Future (with Futurum analyst Guy Currier).
Techstrong TV — $40B future for platform engineering

16 Techstrong TV — The Platform Engineering Show, Ep. 1: Platform Engineering vs DevOps.
Techstrong TV — The Platform Engineering Show

17 PlatformEngineering.com launch — a new hub for the cloud-native era.
Cloud Native Now — PlatformEngineering.com launch

Platform engineering follows the application.
It always has.

TECHSTRONG GROUP

DevOps.com · Cloud Native Now · Security Boulevard
PlatformEngineering.com · Techstrong.ai · Techstrong TV

THE FUTURUM GROUP

Independent research and advisory on software, cloud, AI, and infrastructure.

Published in Boca Raton, Florida.

TECHSTRONG PUBLISHING
VOL. 01 · ISSUE 01