WHITE PAPER

# The Upside-Down Economics of Building Your Own Platform

You've Got a Business to Transform—Get on With It

By Jared Ruckle and Bryan Friedman
In Collaboration with Matt Walburn

Pivotal

# Table of Contents

# Background

## The Path to Cloud Native

Over the last twenty years, information technology has crept closer to the center of business. IT now powers nearly every function of the modern enterprise. So where do we go from here?

Large enterprises are coming to grips with the reality of digital disruption. This realization has been shaping IT strategy and driving more and more parts of the business to a cloud computing model. The move to cloud (both public cloud and on-premises) is causing a shift in application architecture. New software patterns, collectively called “cloud-native”, deliver unheard of application resilience and flexibility. This requires a new type of platform, one that supports continuous delivery and horizontal scale.

How can an organization respond to these demands?

One path embraces “full-stack engineering”: developers build and operate the entire application stack. This is **the do-it-yourself (DIY) option**, a philosophy born from IT tradition.

The other approach: deploy a commercial platform. Developers and operators get to focus on custom software, and a vendor provides patches, new features, and ongoing support.

## You’ve Got a Business to Transform

Companies that decide to roll their own platform quickly realize a cold truth: it’s more expensive than they thought. Their investment in platform engineers balloons higher than projected. Multiple product teams are needed. Each team demands multiple engineers and a product manager.

In working with numerous Fortune 500 companies, we’ve found that even a minimal DIY platform effort can take 2 years to build and cost $14M in payroll alone (for 60 engineers). And that’s just to get to a minimum viable product. Can your company wait two years to start your cloud-native transformation in earnest?

Of course, once the platform is live in production, the costs continue to grow.

Adding new features to a custom platform requires a corresponding investment in engineering staff. After a while, your platform team starts to look a lot like a cloud platform software company, with one important difference:

**The platform you have built doesn’t actually generate revenue for your core business.
It generates expense and quickly accrues technical debt.**

This expense can’t be justified, unless your core business is selling cloud platform services. The problem of creating a vibrant, scalable, and secure enterprise platform has already been solved by Pivotal with Pivotal Cloud Foundry (PCF).

Pivotal customers like Boeing, Mastercard, T-Mobile, StubHub, Rabobank, Dick's Sporting Goods, Liberty Mutual, and Comcast are using our cloud-native platform to ship high-quality software faster than ever before, enabling their businesses to innovate and thrive as a result.

In this whitepaper, we'll arm you with the tools and facts you need to effectively discuss and discredit the DIY platform approach.

You will be able to persuade your colleagues that, for most companies, a homebrew platform will result in lost time and wasted budget, and ultimately lost market opportunities. The leaders in your industry embrace an opinionated platform like Pivotal Cloud Foundry.

## What is a Cloud Native Platform Anyway?

A platform delivers predictable deployment and operations of software by wrapping cumbersome, repetitive, and error-prone tasks with abstractions that are easy for developers (and operators) to consume. These abstractions enforce the desired predictability across the entire application lifecycle by establishing explicit promises between adjacent layers of the stack.

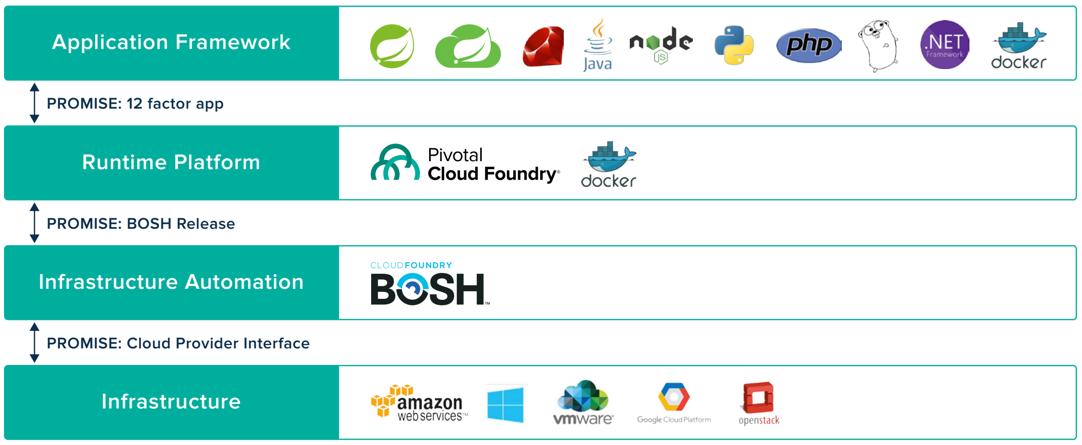

Figure 1. The promises between the layers of cloud-native platforms.

What's the point of these promises? They help drive consistent and repeatable automation among components. With standardization like this, organizations achieve high levels of efficiency and economies of scale.

Put another way, the constraints set you free!

### The Five Domains of a Cloud Native Platform

"Platform" is one of the most misused and misunderstood words in IT. Let's clarify what a modern platform must consider. These five domains in Figure 2 are a useful starting point.

| Infrastructure | Operations | Deployment | Backing Services | Security |
|---|---|---|---|---|
| Container Orchestration | Service Monitoring and Dependency Management | Lifecycle Management Deploy \| Patch \| Upgrade \| Retire | HTTP / Reverse Proxy | Control Plane Audit & Compliance |
| Infrastructure Orchestration | Inventory, Capacity, and Management | Release Packaging, Management & Deployment | Application Runtime | Security Event & Incident Management |
| Service Discovery | Event Management and Routing | CI/CD Orchestration | In-Memory Object Cache | Secrets Management |
| Configuration Management | Persistent Team Chat | TDD Frameworks | Search | Certificate Management |
| Core IaaS | Metrics & Logging Analytics & Visualization | Container Registry/Artifact Repository | Messaging | Identity Management |
| NAT, DNS, SDN, IPAM | Log Aggregation, Indexing & Search | Standard Builds & Configurations | NoSQL Document Store | Threat & Vulnerability Scanning |
| Firewalls, WAN & VPN, Storage, Load Balancers, Compute, Network | Metrics Collection, Storage & Retrieval | Source Control Management | NoSQL Key/Value Store | Network Security |

Figure 2. The five domains of a cloud-native platform.

Pivotal

Figure 2 is one view of what's needed to build and operate a cloud-native platform. It meets common enterprise requirements for managing applications and infrastructure *at scale*. Platform builders often omit one (or more) of these domains. The root cause is usually a shortsighted view, without adequate consideration for the complexities of longer-term scale. Let's dig deeper into each of these areas.

**Any credible cloud-native platform will have deep capabilities in these 5 domains:**

1. **Infrastructure**. In the cloud era, infrastructure is provided "as a service," commonly thought of as "IaaS" whether public or private. The platform requests, and manages the health of this infrastructure via APIs and programmatic automation on behalf of the developers and their applications. In addition, the platform should provide a consistent way to address these APIs, across providers. This ensures the platform and its applications can be run on and even moved across any provider.
2. **Operations**. Metrics and data are the lifeblood of a successful operations team. They provide the insights required to assess the health of the platform and applications running on it. When issues arise, operational systems help troubleshoot the problem. Log files, metrics, alerts, and other types of data guide the day-to-day management of the platform. Ideally, you only need a small team of operators to support hundreds of developers.
3. **Deployment**. Deployment tooling enables continuous building, testing, integration, and packaging of both application and platform source code into approved, versioned releases. It provides a consistent and durable means to store build artifacts from these processes. Lastly, it coordinates releasing new versions of applications and services into production in a way that is automated, non-disruptive, and eliminates downtime for consumers during the process.
4. **Runtime & Data**. The components of the stack that interact with custom code directly. This includes application runtimes and development frameworks, in addition to commercial and open source versions of databases, HTTP proxies, caching, and messaging solutions. Both closed and open source stacks must have highly standardized and automated components. Developers can access these features via self-service, eschewing cumbersome manual ticketing procedures. These services must also consume API-driven infrastructure, operations tooling for ongoing health assessment, and continuous delivery tooling.
5. **Security**. The notions of "enterprise security compliance" and "rapid velocity" have historically been at odds. But that no longer has to be the case. The cloud-native era requires their coexistence! Platform security components ensure frictionless access to systems, according to the user's role in the company. Regulators may require certain security provisions to support specific compliance standards.

## "Why Do I Need a Platform Again?"

The future of your enterprise hinges on your ability to continuously deliver cloud-native applications. The prior section illustrates the enormity of this job. Platform or no platform, these are capabilities you must have.

What happens in the absence of a platform? Developers must slog through ticketing systems and other means of "request for service" across a bewildering maze of teams and intake processes. This is the hallmark of inefficient, functionally-siloed IT organizations. Agonizing wait times and slower time-to-value take their toll on employee morale and your company's market share. There is also the security consideration. How quickly can your operations team patch systems when a security vulnerability is identified? For many companies, it is a highly manual process that takes weeks or months. Sometimes a given patch will never be applied! This is not a risk you can afford to take.

This is why many companies adopt self-service platforms to deliver highly automated software delivery and lifecycle management. A platform provides a consistent, predictable, and secure path for developers to deploy and run their apps, in production, at scale. Platforms deliver crucial features to developers in a unified self-service model. In other words, platforms hide the messy details of a very complex set of IT capabilities.

The question isn't whether your organization needs or will realize value from a platform. The question is: how long are you willing to wait to realize the benefits of a platform-enabled future?

## "But Wait...Can't We Just Use Kubernetes for That?"

You may be looking at some of these capabilities and thinking "yeah, we do that with Kubernetes." There's no doubt that Kubernetes has become the de facto standard when it comes to container orchestration. It powerfully handles the necessary functions around deployment, management, scaling, networking, and availability of containers.

But maintaining Kubernetes as part of a custom-built solution presents some challenges. For one, Kubernetes releases quarterly. That means four major releases per year. If you have a problem updating your systems today, you will have this same problem with Kubernetes. What's more, the Kubernetes community doesn't have the notion of a long-term support model. It's now "table stakes" to be able to upgrade Kubernetes at will. What is your plan for that?

Now ask yourself how many Kubernetes clusters you think you'll need to support your development team. The answer is definitely more than one. Multi-tenancy is difficult to realize with a single large cluster. Many development teams end up creating one cluster per team or many clusters to support different sets of users. This requires a platform that supports self-service, on-demand provisioning of Kubernetes clusters.

Yes, Kubernetes is wonderful tech. And yes, it's tricky to run with a traditional operational mindset. It's also only a piece of the overall platform equation. Think back to Figure 2, the five domains of a cloud-native platform. A container orchestrator is merely one component of a modern platform. By itself, however, Kubernetes is not a full featured cloud-native platform. Even commercial distributions of Kubernetes still include additional tooling to support all the components required in an enterprise-ready platform.

Kubernetes alone isn't sufficient. The question still remains: should you build your own platform, or not? Can you get the business outcomes you desire with a homegrown system?

A do-it-yourself path offers minimal upside (particularly at scale), with the chance for many missteps, as we'll now examine.

# The 5 DIY Pitfalls

We've observed a common story when it comes to DIY platforms. At first, the task seems relatively easy and most of the components are readily available to assemble. However, the engineering team quickly realizes that the scope of the platform is much larger than it seemed at first: all of the functionality in the platform reference architecture and the integration thereof. To meet this need, the team will require more time and people, increasing the overall cost of the platform project. Once a working platform is delivered, that organization must stay in place and continue evolving the platform as new features are requested, bugs are found, and new best practices emerge.

Much of this work is necessary, but not value-added: the platform merely supports the actual applications the business uses. Like an operating system or a database, the platform is required to support applications, but the applications provide the actual value to the business. Organizations, therefore, should plan to spend most of their resources where the value is—on the applications—and delegate the non-value work of platform development and maintenance to other parties.

Let's look through the process that organizations often go through to come to this conclusion. Hopefully you can learn from others' experiences and avoid their DIY platform pitfalls.

## 1 – Oversimplification

### DIY Myth

The DIY engineers in a company often interact with a fraction of the capabilities listed in the five cloud-native platform domains.

### DIY Reality

Most of the time, engineers that experiment with cloud-native technologies are narrowly focused on the "day one" problem of simply getting their applications deployed. This is the obvious first step, and is easily accomplished with commodity container tools and an open source container scheduler like Kubernetes.

Except, Kubernetes is not enough. Containers aren't the hard part.

The hard part arrives after the applications are deployed, on "day two". The highly distributed, dynamic, and interdependent nature of cloud-native workloads requires a platform that ensures that both the platform and its applications can not only be deployed, but operated, scaled and enhanced through high-velocity continuous delivery practices. And this has to be done without compromising high availability and enterprise security.

For teams that embark on the DIY path, this complexity is an unknown dimension that hasn't been considered. Or, it's a "kick the can" exercise, where the "hard part" is deferred to some far off point in the future. In either case, the result is a high degree of risk for your company and its applications on day two.

The oft-overlooked part: **the sustained investment of time and resources to build, operate, maintain, secure, and scale the platform to meet the changing needs of its customers.**

### Operating at Scale

A homebrew platform often gets built in the beginning to fit the needs of one small group of application developers. Many times the developers themselves have pieced together the platform to serve their specific requirements. For small, “app-scale” teams working independently and managing a small number of workloads, an “incomplete” platform may be just fine. These teams may be better off using just a container orchestrator and some simple CI automation—the smaller footprint could be cheaper to manage operationally in this case.

For “enterprise-scale” teams, though, this build-it-yourself model is just not sustainable. Without all the necessary components and standardizations that a true enterprise-scale platform provides, it’s impossible to achieve the same level of efficiency gains and cost savings when running at scale. Be especially careful if you’re just starting small but plan to expand in the future. Even with just a few apps to start, it’s better to use a platform that’s ready to handle it when it’s time to add a lot more.

## 2 – Underestimating Ongoing Investment

### DIY Myth

The unspoken assumptions behind every DIY platform: a sufficient platform can be assembled quickly and easily. It won’t cost anything, because we can just download open source components. We’ll assemble and integrate everything in our “spare time”.

### DIY Reality

It’s not uncommon for a team of 60 engineers to require two years to build a platform comparable to commercially available alternatives. You have invested $14M and two years of time just to get a platform built. Now, you have to start on building the applications on top of it. And you can’t repurpose the platform engineering team to modernize the app layer, without degrading the underlying stability or robustness of the platform.

Building, maintaining, and continually improving a platform for an enterprise is a full time job. Compounding the problem: operations and development teams are often already pushed to the limit. After years of “optimizing” and cutting costs, most IT departments are short on “spare time.”

Business and IT leaders need to challenge the “quick and easy” reasoning. A successful platform that delights developers and customers is a product, and requires investment as such. A platform needs an engineering organization behind it to ensure long-term viability and ROI.

#### Platform as a Product

“What do you mean the platform is a product?”

Cloud platform products are:

- Composed of smaller products, where each service is engineered and delivered as “products” themselves. They have their own teams and backlog of prioritized work.
- Optimized for time-to-value, driving speed, quality, and consistency in the process.
- Consumed and enhanced through self-service APIs and source code repositories.
- Quickly patched—with zero downtime—as new vulnerabilities are uncovered and fixed.

- Delivered just like the apps that run atop a platform. They leverage the same practices and capabilities offered to developers—automation, monitoring, reporting, and continuous testing, integration, and deployment.
- Designed to measure and improve KPIs for all technology products, based on feedback provided by telemetry and customer feedback.
- Consider the product philosophy espoused above. It's an important thing to get right. (In fact, Pivotal just published a whitepaper on this topic.) Now consider the Reference Architecture from earlier. What kind of staffing do you need at scale?

Dozens of engineers, at minimum.

The organization is choosing to swap the cost of commercial platform software for the people cost required to build, customize, and integrate open source and bespoke software into a coherent, consumable platform.

Below is one example of an organizational structure for a DIY platform engineering team.

| Seven Agile (Scrum) Product Teams<br>(To Align With Reference Architecture Capability Domains) | Total of approximately 60 total resources to build an enterprise cloud-native platform team: | Total Payroll Cost of DIY Platform Engineering Team |
|---|---|---|
| 1. Infrastructure<br>2. Operations<br>3 Deployment<br>4. Runtime & Middleware<br>5. Database<br>6. Security<br>7. Coaching & Developer Enablement (to train the developers on the platform) | • Each product team requires an average of 7–9 engineers per team (two-pizza teams)<br>• Scrum master shared 2–3 across teams<br>• Product owners shared 2–3 across teams | • 2 years<br>• 7 teams<br>• x8 people<br>• x $125k/year<br>• ~$7,000,000 per year in payroll |

Figure 3. DIY platform engineering team structure.

## 3 – Thinking the Platform is Done

### DIY Myth

Your company has made the initial investment in building a custom platform. It's tempting to say "Once the platform reaches version 1.0, we'll turn it over to support. That team will keep the lights on, so we can reallocate engineers to other projects."

Do you see the problem here? If your platform is a product, it is never "done", only shipped. Each product team in the platform organization is responsible for their respective services. They are also responsible for ensuring their services are:

- Continuously developed, refactored, and matured to the satisfaction of their developer "customers."
- Continuously tested, integrated, delivered, and improved with tools and processes delivered by the Deployment domain.

Pivotal

- Continuously monitored and reported on with metrics and dashboards provided by the Operations Domain to ensure operational SLAs and outcomes are delivered: platform, application, or business
- Evolving to support the components of the application stack delivered by the Runtime & Middleware, and Data domains.
- Continuously secured by the Security domain
- Aligned with the philosophy of the Coaching and Developer Enablement team

### DIY Reality

Delivering, operating, and maturing a platform in perpetuity is an investment that most businesses cannot justify.

Organizations that make this investment find that, despite a strong initial offering, it becomes impossible to keep pace. The innovation from commercial platform providers quickly surpasses what internal teams are capable of. The velocity gap grows as the DIY team is stuck with day-to-day operations and support. The scope and burden is simply too costly for even the rosiest business case.

#### A Paradigm Shift for Platform Operations

Thankfully, a commercial offering means there's a product team already taking care of all of the responsibilities listed above. Even a commercial platform, though, will need someone to be responsible for its operational management.

Most commercial platforms will certainly require a smaller team to maintain them than would be required for a custom-built solution. However, the best commercial platforms require very small teams, with only 3-4 platform operators running a platform that supports thousands of developers. That's because they follow the principles of *autonomy* as opposed to automation.

Remember, the platform is a product. It should be treated the same way developers treat their applications. This way, the platform itself can be continuously delivered. Instead of a spaghetti set of configuration management scripts, platform operations teams should use a declarative model that can be fed through an automated pipeline. This allows the platform to be completely rebuilt, in place, with no downtime, while applying updated OS images, CVE patches, or new features delivered by the commercial offering.

## 4 – The Lock-In Boogeyman

### DIY Myth

"We must avoid lock-in to maintain leverage over our suppliers!"

"Thanks to open source software, I can save the organization from lock-in, and create opportunities to exert cost pressure over suppliers lower in the stack, particularly infrastructure as a service providers. No supplier is going to hold us hostage!"

This is a common argument.

A DIY platform may indeed prevent supplier lock-in and hold-ups. But it almost certainly results in internal lock-in, and messy conflicts between the organization and its employees.

## DIY Reality

So what's the problem with a homebrew platform? Let's call it the Snowflake Effect. Even if the platform is comprised of open source components, the fully integrated platform only exists within the context of a single organization, becoming a unique structure; a snowflake.

This condition introduces many issues around documentation, training, and support. It all becomes the exclusive burden of the platform team. Formal documentation may be replaced by the idea of "self-documenting code" which may prove difficult to follow for those who did not write the code itself. This leads to an environment where knowledge is tribal, putting the organization at risk of incurring a material impact to platform health, stability, and supportability if key individuals leave. Hiring new talent becomes more difficult (and more expensive) as people must be willing to digest and become proficient with the various platform components and custom integrations.

So what's the alternative? A commercial product that is backed by an open source foundation provides enormous benefits. This option frees your engineers to add value higher in the stack. It reduces risks associated with building and maintaining a snowflake platform. And the list goes on. With a commercial platform (backed by an open source foundation) you gain:

- A consistent and coherent engineering effort across the entire product. The product is designed to work as an integrated solution instead of a solution of custom-integrated pieces and parts.
- The inner workings of the platform are widely understood. Knowledge is institutionalized, and accessible via public documentation and source code repositories.
- Expertise is at the ready. Best practices for implementing, operating, and supporting the product is created (and shared!) across multiple customers, multiple distributions, multiple open source contributors and foundation members, and the commercial product organization itself.
- An ecosystem of "value-add" components thrives. Official mechanisms for premium/enterprise support, training, and consulting are provided as optional parts of the overall solution, reducing dependence on any one particular set of individuals.

Another benefit? If you select a platform that works **across all major public and private clouds**—you can continue to avoid lock-in and maintain cost-pressure for infrastructure as a service providers.

### Assessing Cloud Portability

Folks want to avoid lock-in because they desire portability. Who doesn't want the option to move their apps around across infrastructure targets?

In our experience, portability should give way to the much more practical goal of *choice*. The ability to choose which IaaS vendor to run on or what provider to use for a particular backing service is important to IT leaders. Platforms can make this simpler, but it's complicated to get right. Providing too many options may cripple your team. But you want to offer enough choice to allow for some flexibility.

Think carefully about your goals here. Your chosen platform should offer capabilities that enable the right amount of choice while standardizing operations across infrastructure.

## 5 – “We’ve Already Started, It’s Too Late to Abandon the Project.”

### DIY Myth

So the homebrew platform train has left the station and you’re already on board. Maybe you’re even halfway through your project. Sure, you might run into some obstacles - perhaps it’s already proved harder than you thought. But shouldn’t you see this thing through?

No you shouldn’t! This is a classic case of the sunk cost fallacy. Don’t fall into this trap. It’s better to stop before you get any deeper. Pivotal frequently sees clients coming back a year or two after their DIY effort kicked-off, only to ultimately go the “buy” route. Think of the business outcomes they could have achieved during that time, if they had only chosen the commercial route in the first place!

### DIY Reality

Don’t be afraid to turn back. Shift your direction, and aggressively pursue the commercial platform route.

It’s understandably a painful proposition for some. Navigating a big change in strategy like this can be challenging in large organizations. But it’s worth the difficult discussion. Consider the following topics to get the conversation started:

- Keep the bigger picture in mind. Make decisions that are based on what is best for the future instead of what sounded good in the past.
- Does your custom platform effort look like it may be doomed by one of the first four pitfalls? Don’t forget pitfall #3—your platform is never done. The longer you wait, the more resources you will end up sinking into your custom solution. Take a moment to pause and consider what you could gain by stopping now and adjusting course. As the old saying goes, if you find yourself in a hole, stop digging.
- Get an outside opinion. It can still be someone from within the organization, just not someone from the team building the platform. The team can often get personally attached to what they’re building. It’s hard to let go when you’re in the thick of it, so an objective viewpoint can help clarify. It may be worth a call to an analyst firm like Gartner for their thoughts on your situation.
- Review your “lessons learned” so far. Not finishing what you’ve started may feel like a waste. But you don’t have to throw out what you learned. Once you’ve selected the right platform, you can take advantage of the knowledge you gained, and use it to your advantage.
- Migrating from the homegrown path to a commercial platform may be less painful than you think. If you've been building a platform to run containers and use backing services, they will run great on the platform you choose. Your custom software will still work!

These are the pitfalls. Now, let’s discuss how to prioritize your engineering resources differently—so they drive value for your business, instead of wallowing in the plumbing of undifferentiated activities.

# What Should I Buy? What Should I Build?

**Your applications provide business value, your DIY platform does not.** Don't discourage teams from building. Reframe the discussion to focus on building the "right" things and buy the rest!

## Build The "Right" Software

### Ship Frequently and Improve Time to Value

A platform that just works frees developers from the burden of maintaining the infrastructure and application runtime dependencies that support their software.

They can now focus on developing more software that's of higher quality, and adds more business value. Here's how this will show up:

- More frequent releases through continuous delivery
- Higher quality software via test driven development
- Experimentation and rapid learning
- Exploration of new architectures that reinforce the benefits above

### Full Lifecycle Engineering

Focus on your app. Trust (and verify) that the chosen platform works for your organization.

With these two conditions in place, engineers can manage their software throughout its lifecycle. That means a reduced burden on operations, since developers are doing more of the work.

It also leads to shorter feedback loops between developers, their code, and end users. Opportunities to improve quality and value shine through.

**Full Lifecycle Engineering (for Platform and Applications) includes these undifferentiated activities:**

- Product management (features, enhancements, and priorities)
- Continuous testing, integration, and delivery
- Release management and deployment (continuous or otherwise)
- Operations and support, including on-call
- Capacity, availability, and performance management

Operators and developers alike should ask themselves: Does the task at hand add value to the business?

## Buy the "Right" Platform: Key Criteria

We've explored platform technical capabilities in detail throughout this whitepaper, but there are other key factors you should consider. Let's review the most important buying criteria, based on feedback from hundreds of enterprises. By way of example, we'll review how Pivotal Cloud Foundry meets these criteria:

- **Completeness.** As we've discussed, with an incomplete platform, you're running uphill. It's harder to realize the levels of efficiency, consistency, and quality that a more comprehensive solution delivers. Your team will fill in the gaps, with custom software, introducing wasteful variance and diversity. Developer and operator practices will diverge, creating stability and security issues.

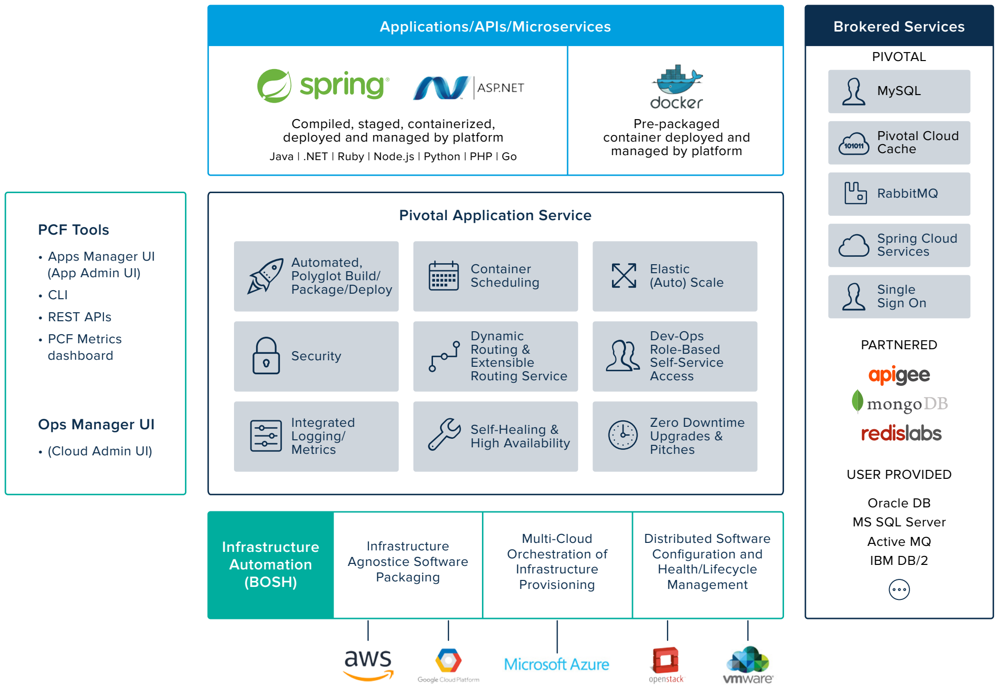
Applications/APIs/Microservices
spring
ASP.NET
Compiled, staged, containerized, deployed and managed by platform
Java I .NET I Ruby I Node.js I Python I PHP I Go
docker
Pre-packaged container deployed and managed by platform
Brokered Services
PIVOTAL
MySQL
Pivotal Cloud Cache
RabbitMQ
Spring Cloud Services
Single Sign On
PARTNERED
apigee
mongoDB
redislabs
USER PROVIDED
Oracle DB
MS SQL Server
Active MQ
IBM DB/2
PCF Tools
Apps Manager UI (App Admin UI)
CLI
REST APIs
PCF Metrics dashboard
Ops Manager UI
(Cloud Admin UI)
Pivotal Application Service
Automated, Polyglot Build/ Package/Deploy
Container Scheduling
Elastic (Auto) Scale
Security
Dynamic Routing & Extensible Routing Service
Dev-Ops Role-Based Self-Service Access
Integrated Logging/ Metrics
Self-Healing & High Availability
Zero Downtime Upgrades & Pitches
Infrastructure Automation (BOSH)
Infrastructure Agnostic Software Packaging
Multi-Cloud Orchestration of Infrastructure Provisioning
Distributed Software Configuration and Health/Lifecycle Management
aws
Google Cloud Platform
Microsoft Azure
openstack
vmware

Figure 4. Pivotal Cloud Foundry: A complete cloud-native platform that meets many enterprise requirements. The product also features clean abstraction from infrastructure and other undifferentiated activities, so developers and operators can be more productive.

- **Maturity.** There are many platform products available in the open source community and commercial software market today. Many of these solutions were created in the last twelve to eighteen months. Even among the most popular, few have a proven track record of running in production for the Fortune 1000. Ensure that the platform you choose has a successful history of enabling companies like yours in production, over a sustained period of time.

Figure 5. Pivotal Cloud Foundry customers in a variety of industries use the product in production for their most critical enterprise apps.

- **Open Source.** A platform that benefits from a vibrant open source community is paramount to your sustained innovation and success. The top open source products get even better with an industry-backed foundation model. A formal foundation grows a simple open source community model with contributions and guidance from corporations across multiple industry verticals.

Platinum

CISCO DELLEMC IBM Pivotal SAP SUSE vmware

Gold

accenture High performance. Delivered. Alibaba Cloud Allianz Allstate You're in good hands American Airlines Cognizant Ford

Google HCL THE HOME DEPOT HUAWEI Microsoft NTT PHILIPS sas swisscom VOLKSWAGEN AKTIENGESELLSCHAFT

Figure 6. Platinum and Gold members of the Cloud Foundry Foundation, an example of a thriving, open-source ecosystem.

- **Multi-cloud.** These days, the fact that companies are using multiple cloud providers is just a given. Even IT departments who claim to have gone “all-in” with a single cloud provider typically have at least a hybrid cloud model with some workloads on-prem and some in the public cloud. Commercial platforms insulate your business as much as possible from vendor lock-in. Avoid the proprietary APIs of cloud providers and instead run your apps on a cloud-agnostic platform like Pivotal Cloud Foundry that runs on the best infrastructure Amazon, Microsoft, Google, and even VMware have to offer. You’ll still be able to use service brokers to tap into the higher-level services found on those cloud providers. Pick which best-of-breed options fit your needs to find the right balance of flexibility and portability.
- **Velocity/Roadmap.** The pace of change in the platform industry is remarkable, but how does an enterprise like yours capitalize? How do you know what’s hype, and what’s real? How do you gain long-term value from this velocity? By trusting a vendor to help you along the way. This goes for new automation, support for new cloud providers, ecosystem extensions, and so on. All of these changes should occur at a regular cadence, with major features released multiple times a year. Enhancements should be integrated, production-ready platform features. To help our customers, Pivotal releases several times a year, with comprehensive documentation for each enhancement.
- **Ecosystem.** No single platform can provide every feature to every customer. Nor should it! Look for platforms that have a robust ecosystem of partners and ISVs that extend capabilities

in relevant areas. Your platform should, ideally, have a marketplace—think App Store—for cloud-native services and extensions. Your platform should expose the marketplace as a collection of automated, self-service “dial-tone” bits that just work, every time. This frees developers to deliver at the speed of their imaginations, without getting bogged down in manual handoffs.

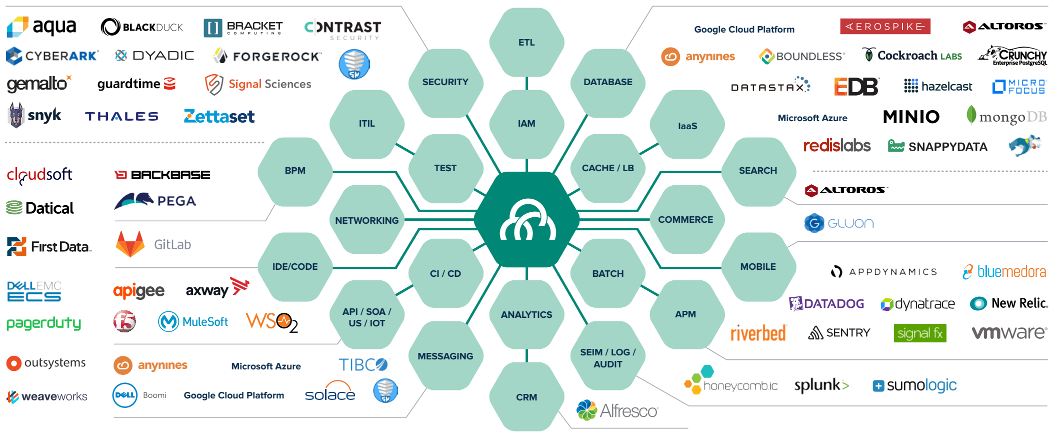

Figure 7. Pivotal’s ecosystem includes dozens of partners with popular offerings that can be quickly integrated.

- **Developer Productivity.** High levels of abstraction allow developers to focus purely on writing value-added application code. No more worrying about packaging code or runtime images. No more writing complex deployment scripts. No more management of underlying infrastructure. Simply write, commit, test, and push your code to the platform. This notion is at the core of Pivotal Cloud Foundry, for example. Developers love their software. Make sure your platform lets them give it their full attention.
- **Operator Efficiency.** Operators love to fix what’s broken. But a good platform fixes itself, and can continuously deploy itself. Instead of manually deploying, patching, upgrading, and scaling infrastructure and related services, your platform should do this in a way that’s automated, stable, and secure. This allows operators to work more closely with their partners in engineering and bring their ideas for value-added business applications into production with a high degree of confidence. With Pivotal Cloud Foundry, operations can extract itself from the day-to-day firefighting, and focus on helping the organization build better software.
- **Comprehensive Security.** Ask platform vendors how often they submit patches and issue new components vetted to protect against the most recent CVEs. Ideally, security patches should be issued within days of an identified threat. At Pivotal, we aim to patch all “high” and “critical” CVEs within 48 hours. Pivotal’s own Cloud Operations team can apply patches within hours, as they did when addressing the Meltdown and Spectre vulnerabilities. We enable our customers to follow a similar process for rapid patching. Refer to https://www.pivotal.io/security for more details.
- **High Availability.** Outages are expensive. System downtime costs enterprises tens of millions of dollars every year. But whether it is a network interruption, misconfiguration, or faulty infrastructure, failure is inevitable. Platforms must be built for resilience. This way, overall uptime is maintained in the face of failing components. Pivotal Cloud Foundry includes four layers of high availability. It goes all the way up the stack from availability zone, to virtual machine, to underlying processes and the application instance itself.

| Infrastructure | |
|---|---|
| **Capability** | **Solution** |
| Multi-Region Deployment | AWS Regions (US-East) |
| Multi-AZ Deployment | AWS AZs |
| Networks Circuits/VPN | AWS DirectConnect |
| Private Networks/SDN | AWS VPC |
| Routing & Segmentation of Private Networks (internal) | AWS VPC |
| Firewall/Segmentation (external) | AWS VPC and Security Groups |
| DNS | AWS Route 53 |
| Outbound Network Traffic | iptables |
| OS Image Build | Packer |
| Configuration Management | Chef |
| Service Discovery | Zookeeper |
| Container Format | Docker |
| Container Scheduler | Amazon EKS (Kubernetes) |

| Visualization & Reporting | |
|---|---|
| **Capability** | **Solution** |
| Platform Logging | AWS CloudWatch |
| Platform Dashboard | AWS CloudWatch |
| Platform Monitoring | AWS CloudWatch |
| Platform Alerting | AWS CloudWatch |
| Application Logging | Fluentd |
| Application Dashboard | Grafana |
| Application Monitoring | Prometheus |
| Application Alerting | PagerDuty |
| Usage Reporting | *Custom* |

| Orchestration and Release Management | |
|---|---|
| **Capability** | **Solution** |
| Continuous Integration | Jenkins |
| Continuous Delivery | AWS CodePipeline |
| Platform Automation | Terraform or AWS CloudFormation |
| Container Registry | Amazon ECR |
| Deployment Policy Enforcement | *Custom* |
| Versioning and Bundling of Deployment Artifacts | *Custom* |
| Application Deployment Abstraction | *Custom* |
| Automated Blue-Green Traffic Switching | *Custom* |
| Service Catalog of Approved Versions of Deployment Artifacts | *Custom* |

| Security | |
|---|---|
| Network Security Monitoring | Bro |
| Identity Management | AWS IAM |
| Secrets Management | Vault |
| Certificate Management | AWS Certificate Manager |
| Platform Security/RBAC | AWS IAM |
| Application Security/RBAC | *Custom* |
| Vulnerability Scanning | Clair |
| Incident Response | *Custom* |

| Backing Services | |
|---|---|
| HTTP Proxy | Nginx |
| Object Cache | Redis |
| Document Store | MongoDB |
| Messaging | NATS or Amazon SQS |
| Geo-Replicated Messaging | Kafka or Kinesis |
| Geo-Replicated NoSQL | Cassandra |
| Data Search and Analytics | Apache Spark |

Figure 8. DIY Cloud-Native Platform—Reference Architecture on AWS

## Comparison: DIY OSS Platform & Pivotal Cloud Foundry

We have examined platforms through the lens of the 5 domains, and their respective sub-components. We've also mentioned a number of vendors and products that compose the world of cloud-native. Let's now combine these perspectives. The first diagram illustrates the components of the DIY approach and a solution or vendor commonly used for each area.

This reference architecture example represents just one way of building a platform. It features a mix of open source projects, commercial products, and AWS-specific solutions. Sure, this is just one configuration. But even if you are selecting from the leaders in something like the CNCF landscape, the challenges you'll face are clear. What's the support and security plan for the open source components? How do you maintain the custom solutions your team has built? Can you integrate all of the pieces so they work together seamlessly? Even selecting AWS-based options doesn't guarantee an answer to this last question. There is a considerable amount of work to be done even when integrating public cloud services together. And this doesn't even take into account the common multi-cloud requirement.

Now, take a look at this same list in a Pivotal Cloud Foundry deployment.

| Infrastructure | | Visualization & Reporting | | Orchestration and Release Management | |
|---|---|---|---|---|---|
| **Capability** | **Solution** | **Capability** | **Solution** | **Capability** | **Solution** |
| **Multi-Region Deployment** | Pivotal Ops Manager | **Platform Logging** | PCF Healthwatch | **Continuous Integration** | Concourse |
| **Multi-AZ Deployment** | Pivotal Ops Manger | **Platform Dashboard** | PCF Healthwatch | **Continuous Delivery** | Spinnaker |
| **Networks Circuits/VPN** | AWS DirectConnect | **Platform Monitoring** | PCF Healthwatch | **Platform Automation** | PCF Platform Automation with Concourse (PCF Pipelines) |
| **Private Networks/SDN** | Pivotal Ops Manager/ NSX-T | **Platform Alerting** | PCF Healthwatch | **Container Registry** | Blobstore [PAS] / Harbor [PKS] |
| **Routing & Segmentation of Private Networks (internal)** | Pivotal Ops Manger | **Application Logging** | Loggregator | **Deployment Policy Enforcement** | Apps Manager and Diego |
| **Firewall/Segmentation (external)** | Application Security Groups/NSX-T | **Application Dashboard** | PCF Metrics | **Versioning and Bundling of Deployment Artifacts** | PAS and BOSH |
| **DNS** | Gorouter | **Application Monitoring** | PCF Metrics | **Application Deployment Abstraction** | PAS (cf push) |
| **Outbound Network Traffic** | Gorouter | **Application Alerting** | PCF Events Alerts | **Automated Blue-Green Traffic Switching** | Gorouter |
| **OS Image Build** | Stemcells and Buildpacks | **Usage Reporting** | Apps Manager | **Service Catalog of Approved Versions of Deployment Artifacts** | Pivotal Services Marketplace |
| **Configuration Management** | Stemcells and Buildpacks | | | | |
| **Service Discovery** | Spring Cloud Services: Eureka | | | | |
| **Container Format** | Garden [PAS] / Docker / OCI | | | | |
| **Container Scheduler** | Diego [PAS] / K8s [PKS] | | | | |

| Security | | Backing Services | |
|---|---|---|---|
| **Network Security Monitoring** | Bro | **HTTP Proxy** | Gorouter |
| **Identity Management** | Cloud Foundry UAA | **Object Cache** | Pivotal Cloud Cache |
| **Secrets Management** | CredHub | **Document Store** | MongoDB for PCF |
| **Certificate Management** | Envoy (mTLS) | **Messaging** | RabbitMQ for PCF |
| **Platform Security/RBAC** | Pivotal Ops Manager | **Geo-Replicated Messaging** | Kafka for PCF |
| **Application Security/RBAC** | Apps Manager | **Geo-Replicated NoSQL** | YugaByte DB |
| **Vulnerability Scanning** | Clair (via Harbor) [PKS] | **Data Search and Analytics** | Elasticsearch |
| **Incident Response** | PCF Events Alerts | | |

Figure 9. Pivotal Cloud Foundry—Reference Architecture on All Major Public & Private Cloud

The components highlighted in green are all covered for you by Pivotal Cloud Foundry. It's easy to see how Pivotal's platform offers the most complete experience you'll find from any cloud-native platform.

## Have You Thought of Everything Your Platform is Going to Need?

Based on your platform's scope, is it ready for production applications? Has your security team signed off on all these requirements? The truth is, with the evolving nature of technology, no platform is ever done. But there is a minimum set of features you need to have in place in order to get any value from it. Once you hit that point, developers and operators will request new features to fill in gaps and to handle their unmet needs. The trick is to choose the platform that provides as much of what you need as possible. When evaluating a platform for purchase, see how many of the boxes in the reference architecture above you can fill in with out-of-the-box or easily integrated ecosystem components.

Alternatively, you can start with this simpler, high-level view:

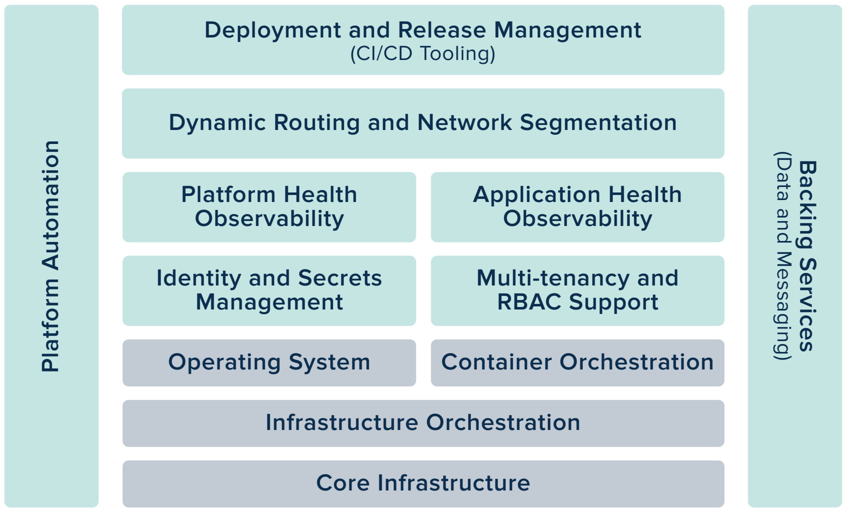
Platform Automation
Deployment and Release Management
(CI/CD Tooling)
Dynamic Routing and Network Segmentation
Platform Health Observability
Application Health Observability
Identity and Secrets Management
Multi-tenancy and RBAC Support
Operating System
Container Orchestration
Infrastructure Orchestration
Core Infrastructure
Backing Services
(Data and Messaging)

Figure 10. Cloud-native Platform—High-Level Reference Architecture

Your initial homegrown platform effort might be running fine with a few apps. But think back to our "Operating at Scale" section. A small sample size of apps is unlikely to reflect the total complexity of your enterprise app portfolio. You can perhaps get away with fewer features and a few manual steps here and there. But how does this scale to 20 apps? 100? 1000? 10,000?

Of course a complete platform like Pivotal Cloud Foundry requires some additional effort up front. But you're planning to use it to run your business! The investment quickly becomes worth it as you scale. Consider Figure 11:

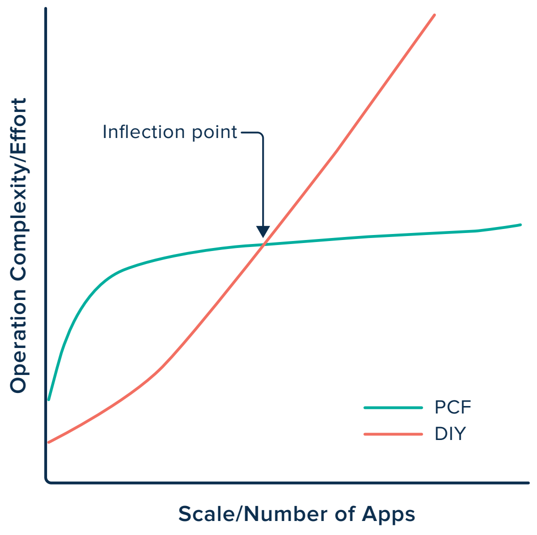

Figure 11. DIY Platforms are Challenging to Run at Scale - PCF is Built for Scale

Where's the inflection point? For some organizations it can be as low as 10 applications. This graph illustrates the trend typically seen for customers who have shifted from using custom-built platforms to running PCF.

**But what about OSS Cloud Foundry?**
If Cloud Foundry handles all these things, why not just use the open source version instead of Pivotal's commercial solution? While some organizations may consider this, the DIY pitfalls still apply.

For one thing, Pivotal provides additional components beyond what you get with open source Cloud Foundry. Many of these are critical pieces of a complete platform including PCF Healthwatch, PCF Metrics, Operations Manager, and an entire ecosystem of partner integrations. Plus, you don't have to add new features or handle integration between components. Pivotal handles "the risk of ongoing investment" for you. In fact, the company spends over $5 million per year on cloud infrastructure to continuously integrate and test more than 20 platform tiles (and the work of 60 teams) before shipping them to the Pivotal Network.

You're also paying Pivotal to handle security and support. Along with providing CVE patches within 48 hours, Pivotal gives you access to industry experts who can help you solve problems quickly.

With Pivotal's industrialized open source approach, the company packages the best of open source tech and makes it consumable for you. Pivotal will also seamlessly incorporate new open source projects as they mature. All of this reduces the risk and complexity of doing it yourself.

Let's remember the big picture. You aren't in the business of building software platforms. Let Pivotal do it for you!

# Conclusion

It might be possible for your organization to build a platform. But it's a completely different proposition to keep up with the market over time, especially this market. The cloud-native market evolves and expands daily.

## Ask yourself these questions:

- Even if you can justify an initial investment in building your own platform, is this the best place to focus my organization's scarce human resources that are already stretched thin?
- What differentiating work is my organization not doing, because we're focused on building an undifferentiated platform instead of business value?
- Do you really want to start your transformation in two years, after version one is finally ready, or do you want to start tomorrow?

This is what "build vs. buy" in the cloud era really means.

Even if you have the money, you don't have enough time and you won't find enough people to justify building a cloud-native platform yourself.

**You've Got a Business to Transform—Get on With It**

# Appendix

## Cloud Native Capability Overview

When building a platform across the five domains described, the following capabilities must be considered and engineered in a way that is:

- Fully automated across its entire lifecycle: deploy, patch, upgrade, and retire
- Integrated with other platform components in a way that lifecycle events don't disrupt consumers
- Made available to consumers in an easy-to-consume, on-demand manner via API and/or graphical user interface

### Infrastructure—Core Infrastructure as a Service

- **Compute**: Configuration and delivery of stateful or ephemeral virtual machines and/or containers, composed of CPU, RAM, block storage, and network interfaces.
- **Network**: Configuration and delivery of layer 2 and layer 3; subnets and routing.
- **Storage**: Configuration and delivery of block, network, or object offerings.
- **Load Balancers**: Configuration and delivery of load balancers (public or private) to distribute traffic across horizontally scaled workloads running on virtual machines and/or containers; also used to route traffic across availability zones and regions.
- **Firewalls**: Management of firewall rules, and dynamic application of rules to compute resources and other services.
- **WAN & VPN**: Network technologies that integrate off-premises cloud services to corporate datacenters.
- **SDN**: Software Defined Networks provide the ability to programmatically create and configure multiple virtual networks on the same physical infrastructure, including networks with overlapping IP address space. SDN is a key mechanism for creating and enforcing multi-tenancy within IaaS providers.
- **IPAM**: IP Address Management prevents IP address conflicts or collisions.
- **NAT**: Network Address Translation allows resources on private networks to communicate with public resources on the Internet.
- **DNS**: Dynamic management of DNS records.

### Infrastructure as Code and Container Orchestration

- **Configuration Management:** Tooling that ensures consistent configuration of environments over time and always brings systems into compliance with a known desired state. Required because when dealing with massively distributed cloud-native architectures and microservices it is impossible to ensure consistency across systems any other way.
- **Service Discovery:** The ability for applications and services to self-discover and self-configure each other through dynamic service registration and discovery frameworks. This is another

Pivotal

capability impossible to manage manually at scale with cloud-native architectures. Manually managing things, such as adding new instances to load-balancing pools and dynamically integrating with other distributed systems in real-time, necessitates service discovery.

- **Infrastructure Orchestration:** Automation that consumes infrastructure components via APIs, and then assembles the resources into end-to-end system architectures. This tooling can manage high availability (auto-scaling groups) and enforce security policies (firewall rules).
- **Container Orchestration:** Automation to dynamically deploy and schedule containers to a cluster of compute hosts. The orchestrator ensures that configuration and availability evolves to meet the desired state as conditions change. May also provide other platform capabilities, such as dynamic routing and load balancing to container instances.

## Operations

Operations teams must properly instrument the platform and the associated applications, so that the health of both can be assessed. The data gathered from this instrumentation helps platform operators and application developers troubleshoot issues when they arise and, hopefully, fix those issues.

This effort serves an organizational purpose as well. Operations teams are bound by service level agreements (SLAs). These SLAs are a baseline expectation of platform availability and performance for customers and employees.

Application teams and product owners also have availability and operational performance SLAs with the business that must be measured and enforced. Platform instrumentation delivers the data required for both. Platform operations and application teams will report on:

- Service Availability Metrics
- Service Delivery/Efficiency Metrics
- Incident Response & Resolution Time Metrics
- System Performance & Response Time Metrics
- Key Business Metrics (Transactions)
- Cost

Real-time alerts and events help operators bring systems back into a successful state quickly. Reports and dashboards provide trends and historical data.

### Common operational capabilities include:

- **Metrics Collection, Storage, and Retrieval:** The collection of structured time-series metrics from both platform and applications; also includes storage of this data and its on-demand retrieval.
- **Log Aggregation, Indexing, and Search:** The collection of all unstructured infrastructure and application logs into a centralized location. Logs must be indexed and easily searchable. Log data must be available for other systems, particularly for analytics and visualization.
- **Metrics and Logging Analytics, including Visualization:** The on-demand generation of reports, dashboards, and notifications against time-series data and logs. These reports, if thoughtfully authored, provide insight from very complex sets of operational data.

- **Persistent Team Chat (aka "ChatOps"):** Tooling for platform and application teams to collaborate in real-time; includes persistent, indexed chat transcripts for later use. Provides APIs for other platform components and systems to integrate with team chats. Integration points may include code commits and package deployments, results from CI tooling, and platform alerts.
- **Event Management and Routing:** The routing of alerts (collected from telemetry, metrics or logs) to the appropriate responders. This is usually done with a ticketing system. These alerts often arrive via text message, email, phone call, push notification, or persistent team chat.
- **Inventory, Capacity, and Financial Management:** The tracking and reporting of inventory—such as cloud infrastructure and software licenses—mapped to the platform, developer, or business unit consumer. Utilization of each asset is also collected to identify waste. Together, inventory and utilization optimize cost, and identify violators of corporate IT spending policies.
- **Service Monitoring and Dependency Management:** The monitoring of not only single applications, but of entire complex business workflows that are composed of many, many applications. This is particularly important in loosely-coupled microservices architectures, where each application may be composed of even smaller microservices. Think of this as an abstraction over traditional monitoring. As long as the workflow is still available and responsive to its consumers, the failures of individual pieces and parts don't matter as much, provided the system eventually brings equivalents back into service.

### Deployment

- **Source Control Management:** A centralized repository of all source code required to build applications, infrastructure, their dependencies, and the platform itself. All changes to the environment are initiated, tracked, and versioned here.
- **Standard Builds and Configurations:** Repositories of approved and tested base artifacts used by development teams. Artifacts could include base configuration management files, golden virtual machine images, golden container images, golden orchestration templates, and container orchestration manifests.
- **Container Registry / Artifact Repository:** A centralized store of all artifacts—infrastructure, application code, or container images—that meet two criteria. They have successfully been built and have subsequently passed all automated tests in the continuous integration process.
- **Test-Driven Development (TDD) Frameworks:** Frameworks for automated, programmatic testing of infrastructure and application code. Includes unit tests, integration tests, browser compatibility tests, mobile compatibility tests, and performance tests. May also include security tests, like code-scanning and automated threat and vulnerability tests.
- **Continuous Integration / Delivery Orchestration:** Configuration, execution, and orchestration of continuous delivery elements, pipelines, and processes. This includes continuous builds, and automated execution and auditing of TDD frameworks and tests.
- **Release Packaging & Management:** Automation that builds source code into deployable packages. Examples include application binaries, configuration management artifacts, orchestration artifacts, virtual machine images, and container images.

- **Deployment Orchestration:** Automation of the artifact deployment process. This initiates once said artifacts have been successfully built, tested, and pushed to the artifact repository. Common scenarios include blue/green deployments, canary deployments, and rolling updates.
- **Lifecycle Management:** The automated detection of outdated artifacts. When an artifact becomes out of compliance with a desired (and newer) baseline, remediation automatically kicks in and updates the artifact to the new standard. This most often occurs when applying patches, performing upgrades, or completely retiring artifacts that are no longer compliant. This process encompasses and integrates with other deployment-related capabilities.

### Runtime, Middleware, & Data

- **HTTP / Reverse Proxy:** The front-end "router" of incoming connections to instances of the application.
- **Application Runtime:** The application container or runtime where the business logic is executed.
- **In-Memory Object Cache:** An in-memory object grid for storing objects and data; used to improve performance by avoiding calls to a database for frequently accessed data.
- **Search:** A general purpose search service for indexing and querying structured or unstructured data.
- **Messaging:** Integration technology required to enable pub-sub functionality from producers to consumers.
- **NoSQL Document Store:** A NoSQL database (i.e. unstructured, not relational) optimized for storing "documents."
- **NoSQL Key/Value Store:** A NoSQL database optimized for key/value storage.

### Security

- **Network Security:** Automation that constantly watches network traffic and dynamically compensates for threats. Includes firewalls, outbound security proxies, as well as intrusion detection systems (both network and host-based).
- **Threat & Vulnerability Management:** Proactive scanning of infrastructure to uncover vulnerabilities. Includes automated scans of a host for unnecessary or insecure services, and scans within the host looking for outdated, insecure, or misconfigured infrastructure or software components. The scanning services will collect and inventory these defects. Platform operators and developers then address these defects with corresponding changes in their CI/CD pipelines.
- **Identity Management:** Automation for controlling access to applications and platform components. Includes user management, user authentication, and user authorization. Generally performed via federation with an existing directory and identity management system, such as Active Directory.
- **Certificate Management:** Automation for SSL certificates: requesting, provisioning, expiring, and rotating.
- **Secrets Management:** Automation for storing, retrieving, updating, and deleting "secrets", such as passwords, API keys, SSL certificates, and SSH keys.

- **Security Event & Incident Management:** Reactive forensic analysis of a compromised system. Most often performed after receiving telemetry from a system and subsequent analysis of this data.
- **Control Plane Audit & Compliance:** The automated process of auditing and remediating the cloud provider's control plane to ensure that it is configured to adhere to corporate security policy.

## DIY Team Sizes and Scope of Ongoing Work

### Infrastructure & Operations

- **Number of Engineers: 6**
- **Platform Services Provided**
  - Infrastructure as a Service: AWS
  - Base Cloud Network Design: VPC
  - WAN & VPN Platform: DirectConnect + Telco
  - Firewall Design: Security Groups
  - Config Management Platform: Chef Server
  - Service Discovery Platform: Consul
  - DNS Platform: BIND & Route 53
  - NAT Platform: iptables + BRO Sensors
  - Object Storage Platform: S3
  - Infrastructure Orchestration Platform: Cloud Formation
  - Container Orchestration Platform: Kubernetes
- **Scope of Work**
  - Operations, Support, and Lifecycle Management
  - Base Network and VPC Automation: Cloud Formation
  - Base Config Management Automation: Chef
  - Base Security Automation: Chef + Cloud Formation
  - Base Resiliency Automation: Chef + Consul
  - Base Monitoring & Metrics Automation: Sensu + Graphite + Grafana
  - Base VM and Container Automation: Packer
  - Base Infrastructure Orchestration Automation: Cloud Formation
  - Base Container Orchestration Automation: Kubernetes

### Metrics & Reporting

- **Number of Engineers: 6**
- **Platform Services Provided**
  - Metrics Collection: Sensu
  - Metrics Storage & Retrieval: Graphite
  - Metrics Analytics & Visualization: Grafana

  - Event Management: Sensu + PagerDuty
  - Log Collection: Logstash
  - Log Index & Search: Elasticsearch
  - Log Analytics & Visualization: Kibana
  - Financial Reporting: Netflix ICE
  - Capacity and Inventory Reporting
  - Cross-Portfolio Operational Health Dashboarding
- **Scope of Work**
  - Operations, Support, and Lifecycle Management
  - Base Config Management Automation: Chef
  - Base Security Automation: Chef + Cloud Formation
  - Base Resiliency Automation: Chef + Consul
  - Base Monitoring & Metrics Automation: Sensu + Graphite + Grafana
  - Base VM and Container Automation: Packer
  - Base Infrastructure Orchestration Automation: Cloud Formation
  - Base Container Orchestration Automation: Kubernetes

**Continuous Delivery & Release Management**

- **Number of Engineers: 12**
- **Continuous Delivery & Release Management Platform Stack**
  - Full-Stack Deployment & Release Management
  - Cloud Artifact Packaging and Versioning
  - Lifecycle Management
  - Blue-Green Deployment & Traffic Shifting
  - CI Orchestration: Jenkins
  - Artifact & Container Repository: Artifactory or ECR
  - Chef Cookbook Repository: Chef Supermarket
  - Full-Stack TDD & Automated Testing
  - Base Operating System: CentOS + Chef
  - VM Image Factory: Packer + Jenkins = Amazon Machine Images
  - Container Factory: Packer + Jenkins = Docker Images
- **Scope of Work**
  - Operations, Support, and Lifecycle Management
  - Base Config Automation: Chef
  - Base Security Automation: Chef
  - Base Resiliency Automation: Chef + Consul
  - Base Monitoring & Metrics Automation: Prometheus + Fluentd + Grafana

- Base Image Automation: Packer
- Base Container Automation: Packer
- Base Orchestration Automation: CloudFormation or Terraform

### Microservices Platform Operations

#### Runtime

- **Number of Engineers: 6**
- **Microservices Platform Stack**
  - HTTP / Reverse Proxy: Nginx
  - Microservice Runtime: Spring Boot
  - In-Memory Object Cache: Redis
  - Asynchronous Messaging: RabbitMQ
  - Search: Solr
- **Scope of Work**
  - Operations, Support, and Lifecycle Management
  - Base Config Automation: Chef
  - Base Security Automation: Chef
  - Base Resiliency Automation: Chef + Consul
  - Base Monitoring & Metrics Automation: Prometheus + Fluentd + Grafana
  - Base Image Automation: Packer
  - Base Container Automation: Packer
  - Base Orchestration Automation: CloudFormation or Terraform
  - Coordination of Lifecycle Management Activities of Microservices Platform with Customers
    - Patching
    - Upgrades
    - Retirement

#### Data

- **Number of Engineers: 6**
- **Data Platform Stack**
  - NoSQL Key/Value Store: DataStax Cassandra
  - NoSQL Document Store: MongoDB
  - Log-Based Messaging/Streaming: Kafka
  - Unstructured Data Index & Search: Elasticsearch
- **Scope of Work**
  - Operations, Support, and Lifecycle Management
  - Base Config Automation: Chef
  - Base Security Automation: Chef
  - Base Resiliency Automation: Chef + Consul

- Base Monitoring & Metrics Automation: Prometheus + Fluentd + Grafana
- Base Image Automation: Packer
- Base Container Automation: Packer
- Base Orchestration Automation: CloudFormation or Terraform
- Coordination of Lifecycle Management Activities of Microservices Platform with Development Teams
    - Patching
    - Upgrades
    - Retirement

**Security**

- **Number of Engineers: 8**
- **Security Platform Stack**
    - Identity and Access Management: IAM & Ping Federate
    - Certificate Management
    - Secrets Management: Vault
    - Security Event & Incident Management: Logstash, Kafka, and Arcsight
    - Incident Response: Google Rapid Response
    - Threat and Vulnerability Scanning: Tenable
    - Network Security: BRO
    - Cloud Provider Management Console Audit: evident.io
- **Scope of Work**
    - Operations, Support, and Lifecycle Management
    - Base Config Automation: Chef
    - Base Security Automation: Chef
    - Base Resiliency Automation: Chef + Consul
    - Base Monitoring & Metrics Automation: Prometheus + Fluentd + Grafana
    - Base Image Automation: Packer
    - Base Container Automation: Packer
    - Base Orchestration Automation: Terraform or CloudFormation
    - Consultation with Development Teams of Secure SDLC
    - Coordination of Lifecycle Management Activities of Security Platform with Development
    - Teams and Cloud Platform Teams
        - Patching
        - Upgrades
        - Retirement

**Dev Practices / Coaching Product**

- **Number of Engineers: 8**
- **Coaching Team Products & Offerings**
  - Cloud Learning & Development (Training & Open Labs)
  - Pairing with Application Teams on CI/CD, Microservices, and Cloud-Native Platform
  - Application Modernization and/or Migration Planning & Execution
  - Technology Lifecycle Management Planning w/ Product Teams

**All Teams**

- **Total Number of Engineers: 52**

Building a platform is hard in and of itself, and if you're lucky enough to get to a credible 1.0, then you've only just begun. The effort and investment required to keep your platform stable, secure, and meeting the needs of your customers can only only be justified if it delivers true business value.