Cloud Infrastructure in the Federal Government

Modern Practices for Effective Risk Management

NAVA

DORA
DEVOPS RESEARCH & ASSESSMENT

Author

Jez Humble

navapbc.com
devops-research.com

Cloud Infrastructure in the Federal Government

Modern Practices for Effective Risk Management

Abstract

The use of cloud infrastructure combined with modern DevOps practices can deliver significant benefits to federal agencies. Agencies report substantially higher service availability, reduced costs, and faster delivery when implementing recommended principles and practices, and are able to do so while meeting the requirements of FISMA and FITARA. These benefits cannot be achieved, however, by simply applying traditional data center practices to cloud infrastructure.

Agencies and vendors must adopt the new toolchains, processes, and architectural approaches described in this paper, using platform-as-a-service (PaaS) for new systems and infrastructure-as-a-service (IaaS) for systems that lack a cloud-native architecture or for edge cases where a PaaS is not feasible. These innovations allow agencies to deliver high-quality services faster and less expensively while simultaneously improving their ability to manage risk. In this paper, we present principles and practices for cloud infrastructure management that draw from the DevOps movement to enable agile development and that have been successfully implemented in federal government agencies.

Modern Cloud Deployments Require Modern Practices

The use of public cloud services has enjoyed rapid uptake in the private sector, even in regulated domains such as healthcare and finance. Done right, public and community cloud services can substantially reduce investment in infrastructure, shrink the time to deliver services, reduce operational complexity and maintenance costs, and provide better security and compliance outcomes. These benefits are attractive to public-sector customers as well. In fact, as of May 2017 more than 50 US federal government agencies were using FedRAMP Authorized cloud infrastructure services such as Amazon Web Services (AWS), Google Cloud Platform, Salesforce, and Microsoft Azure[1].

A well-designed, centrally managed cloud platform helps agencies meet the requirements of FISMA and FITARA, while also giving teams the flexibility to use the technologies and toolchains they determine are most suitable, and to self-service the operations they need to deploy and operate their systems. This allows modern agile and DevOps principles and practices to be employed when building and operating information systems, which substantially reduces costs and time-to-market while increasing reliability and availability. These modern paradigms are essential if agencies are to avoid business-as-usual: multi-year contracts with inflated price tags and high failure rates in delivering value to agencies, taxpayers, and the public.

1. https://marketplace.fedramp.gov/index.html#products?status=Compliant&sort=productName&serviceModels=IaaS&deploymentModels=Public%20Cloud;Government%20Community%20Cloud

It is important to note, however, that the benefits of a well-designed cloud platform cannot be achieved with poor implementations that simply move traditional data center operations to the cloud, with no other changes in practices. This approach will provide little benefit over traditional data centers, and will not support modern practices in delivering and running software services.

A common but egregious example of failing to implement cloud services correctly relates to on-demand self-service. This is the first of five essential characteristics of the cloud defined in NIST SP 800-145, The NIST Definition of Cloud Computing, which states:

> ***"On-demand self-service. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider."***

Many teams that are building and operating systems hosted on government cloud services must still raise tickets to perform routine operations such as creating a new testing or production environment, making changes to the configuration of their environment, or deploying an application. These tickets are then processed manually by vendors or contractors who make the changes requested through the cloud's console. The result is long lead times and the possibility of errors or misunderstandings (which must be remedied by creating yet another ticket). It can also lead to inconsistent, hard-to-reproduce "works of art." These consist of servers and infrastructure configurations that have evolved through manual changes and that are insufficiently documented for their configuration state to be reliably and deterministically recreated for testing or disaster recovery purposes.

If traditional data center change- and configuration-management processes are used to manage cloud infrastructure, it's impossible to achieve the higher service delivery throughput, operational stability, and availability that is possible with a well-designed cloud platform. Indeed, the result cannot even be called a cloud according to NIST's definition. And while it has been argued that on-demand self-service is impossible to achieve within the context of FISMA, FITARA, and federal procurement practices, compliant implementations that meet these requirements already exist. In recent years, many federal agencies have begun adoption of modern, agile approaches to software delivery, with the goal of building higher-quality services faster and less expensively.

Our central recommendation is that agencies create a multi-tenant cloud platform that implements the majority of the controls required and allows teams that are building and operating information systems to self-service resources. This approach balances the need for effective risk management and governance, which FITARA and FISMA demand, while providing teams the flexibility they require. In this way, agencies can meet their risk-management goals while also enabling the faster delivery, higher reliability and availability, and improved quality that the agile and DevOps movements enable.

This paper describes the critical elements required for compliant, multi-tenant cloud platforms in a federal government context, presents principles and practices for implementing them, and includes a reference implementation that draws from a FedRAMP-authorized service.

Leverage Cloud-Native Architecture to Meet Architectural and Security Goals, Increase Utilization, and Reduce Costs

We have seen multiple instances of agencies that have moved data center operations to the cloud without adopting the architectural design principles that are required for success. The concept of a cloud-native architecture is therefore critical to understanding the recommendations that follow. This approach leverages the unique capabilities of the cloud to meet architectural and security goals such as confidentiality, integrity, and availability, while also increasing utilization and therefore reducing costs. Three characteristics of cloud infrastructure are of particular importance, and are described in more detail below.

1. Disposability of resources

By definition, cloud resources can be provisioned on-demand, which means they can easily be disposed of and recreated as needed. This has substantial implications for cloud-native architecture. Rather than making changes to existing systems, we can provision new versions of a service through a fast and fully automated process and then delete the old version. That reduces the complexities of deployment and disaster recovery, prevents configuration drift, and streamlines the patching process. In addition, if we implement continuous deployment to update services frequently, it makes it significantly harder for attackers to gain a foothold in our production infrastructure[2].

2. Distribution

Systems built to operate on a cloud will necessarily be distributed systems. This follows from the inevitability of network partitions in the context of a cloud infrastructure. The architectural approaches behind using agile methods to build highly available

2. Chad Fowler introduced the concepts of immutable infrastructure and disposable components here: http://chadfowler.com/2013/06/23/immutable-deployments.html

distributed systems find perhaps their best one-sentence expression in an article from 2007 by Jesse Robbins, previously Head of Availability for Amazon:

> ***"Operations at web scale is the ability to consistently create and deploy reliable software to an unreliable platform that scales horizontally."***[3]

In the context of distributed systems, we must assume that our infrastructure may fail at any time. For this reason, we create multiple instances of every service that run in parallel in multiple data centers, allowing for seamless hot-failover in the event of a network partition or failure. Amazon services such as the Relational Database Service (RDS), Lambda, Auto Scaling Groups, and Elastic Load Balancers (ELB)—and their equivalents on other providers—make it straightforward to design systems in this way.

This type of architecture, in which there is no single point of failure, combined with the ability to create new resources on-demand helps prevent outages. It also allows us to scale up our infrastructure horizontally (in other words, by creating more instances of each component) to meet spikes in usage, and scale it down to reduce costs when usage goes down. Amazon's auto-scaling groups allow these activities to be performed automatically in response to algorithms or monitoring telemetry.

3. Cloud primitives

Cloud infrastructure provides primitives—infrastructure objects that can be self-serviced and configured through an API—such as networking components (virtual networks, gateways, routing tables, firewalls), compute (virtual machines), storage (including object and persistent block storage), messaging, and

3. http://radar.oreilly.com/2007/10/operations-is-a-competitive-ad.html

databases. These primitives are designed to be robust, inexpensive, and easy to configure and manage.

Cloud-native architectures should use cloud primitives wherever possible rather than re-implementing them. For example, use native firewall primitives, virtual private gateways, and virtual private cloud (VPC) peering rather than custom appliances, and use cloud-native messaging and database services rather than customized, user-maintained middleware wherever possible.

This approach has multiple benefits:

- Provisioning and configuration is taken care of by the cloud provider, and can be performed self-service using the provider's API.
- Maintenance burden is significantly reduced, since updates, upgrades, and routine maintenance are handled transparently by the vendor.
- Monitoring and alerting for these primitives is typically integrated into the cloud's built-in distributed monitoring and alerting services.
- Where services have received FedRAMP authorization, this can be leveraged in order to reduce the cost and time required to achieve an Authority to Operate (ATO).
- Availability of these services up to a certain service level objective (SLO) is often guaranteed by the vendor, with high-availability configurations provided or documented for many services.

There are very few legitimate situations where the services provided by commercial clouds are not sufficient for agency use. Customization or re-implementation should be avoided since this leads to substantial initial development and ongoing maintenance costs.

Employ DevOps Practices to Increase Availability, Reduce Cycle Times, and Improve Auditability

Infrastructure-as-code (IaC) is a key DevOps practice that helps increase both availability and integrity. In the IaC paradigm, the complete infrastructure specification is kept in machine-readable form in version control, and all changes to our infrastructure are made using programs that understand and can apply these specifications.

The use of IaC enables a fully automated and auditable approach to change management. All changes to the infrastructure configuration should be made through a self-service automated deployment system that will manage the lifecycle and configuration of all infrastructure objects in the cloud infrastructure, including networking and storage. Such a system can, for example, be implemented using GitHub or AWS CodeCommit for source control, a continuous integration tool, and a cloud configuration management tool such as Terraform or AWS CloudFormation to apply changes. This system is referred to as the infrastructure deployment pipeline[4] (IDP), which is shown in Figure 1.

4. For more on deployment pipelines, see https://continuousdelivery.com/implementing/patterns/#the-deployment-pipeline

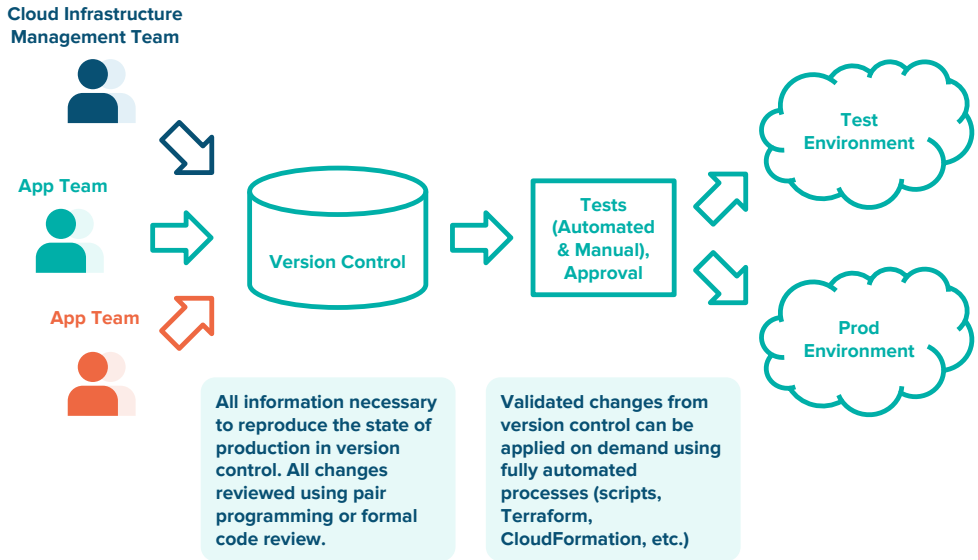

Figure 1: The Infrastructure deployment pipeline (IDP) provides complete version-control history and supports change-control enforcement.

Where IaC is applied comprehensively, it's possible to completely recreate the state of a production infrastructure purely from information and programs in version control. We can also see the complete history of all changes made to the infrastructure by looking through the version-control log for auditing and compliance purposes.

Further, IaC allows infrastructure policy and control implementations to be specified declaratively and then enforced, both as part of an automated change process and dynamically within test and production environments (for example using cloud functions). In this way, we can validate and test for compliance in an automated way continuously throughout the lifecycle of platforms and systems, substantially reducing the burden of continuous monitoring[5].

In the context of cloud infrastructure, IaC allows us to substantially increase both the integrity and availability of our systems by ensuring we can reproduce the state of our production environment exactly (using the cloud's API) in a predictable time. In the case of a disaster recovery scenario, IaC enables us to restore service quickly and predictably, as opposed to a process that involves making changes manually (which is typically both time-consuming and error-prone). IaC also enables us to create production-like test environments on demand.

It's important also to extend the use of IaC to the machine images (templates from which new virtual machine instances are created) used in cloud environments. These images should be reproducible purely from information stored in version control using an automated process, rather than the result of manual configuration. This is implemented in a similar way to the IDP, creating a machine image

5. The OpenControl community has done some important work in this area: http://open-control.org/

deployment pipeline (Packer and Netflix Aminator are two open source tools that can be used to implement this pipeline). In this way, they can easily be audited, upgraded, and updated in the event of new versions of components or libraries becoming available, or patching due to vulnerabilities (CVEs).

Principles and Practices for Creating a Cloud Platform

Applications don't exist in a vacuum—they require an infrastructure platform. In traditional data centers, significant attention was paid to issues such as disaster recovery, storage, and networking. In the context of the cloud, these concerns are taken care of at the platform and application layers. Cloud platforms can implement a number of controls that can be leveraged by applications running on them, which substantially reduces the burden of delivering FISMA-compliant information systems.

Applications don't exist in a vacuum—they require an infrastructure platform.

Building a cloud platform requires an additional platform layer on top of the infrastructure layer offered by cloud service providers. In this section, we'll discuss the motivations behind creating this platform and the principles that should govern its design, particularly in the context of the federal government.

Create a Platform to Manage Service Lifecycles, Dependencies, and Costs

In a cloud platform, deploying and managing the lifecycle of applications, network traffic routing, and logical process and network isolation are all managed by a cloud-scheduler service combined with a container platform. However, building and operating a platform—particularly in a multi-tenant context—is significantly more complex than deploying (for example) Kubernetes.

When designing a cloud platform, there are higher-level services we care about, such as logging, monitoring, and alerting. These services must be provided not just for the platform but also for the applications that run on it. As far as possible, the applications should not need to know anything about how these are implemented. For example, applications should be able to simply log to the default system log and the platform should take care of gathering, aggregating, and storing log files and making them searchable.

Procurement and Contracting Considerations

Procuring cloud services can be a frustrating exercise. Due to the infrastructure-metering model (which depends on the level of usage by end users) precision is impossible in practice. While many agencies attempt to use firm-fixed-price contracts since these are relatively simple to formulate, these contracts are unsuitable for cloud services since, like telecom services, demand for usage is inherently unpredictable. It's a mistake to attempt to enumerate each individual service provided by a cloud, fix the price-per-unit, and estimate the amount of usage.

Further, the potential consequences of a usage estimate that is too low are catastrophic, such as having to switch off public services in order to avoid violating the Antideficiency Act. Unfortunately, that leads to padding estimates significantly, resulting in poor resource allocation and waste. This is compounded by the fact that firm-fixed-price contracts must have their full contract value allocated up front. These contracts also fail to take advantage of the fact that the cost of cloud services inevitably trends downwards over time, and thus represents poor value for agencies and taxpayers.

Cloud services are effectively commodities (another term for cloud is "utility computing"), which allows for a substantially better approach. By procuring the entire catalog of cloud services as a single product using a time-and-materials/labor hours (T&M/LH) contract, and then use a Lowest Price Technically Acceptable (LPTA) source-selection process, buyers can select the cloud provider with the cheapest unit pricing for the services required. Crucially, T&M/LH contracts can be funded incrementally, which means the total contract value need not be allocated up front.

It's still essential to manage and allocate costs in order to avoid violating the Antideficiency Act. This is another key goal of creating a cloud platform. Many platform technologies enable quotas to be established for each information system, allowing resource usage to be constrained and charged back to the team that owns the system. This can be done on a fixed-price basis, provided the price more than covers the cost of the resources used. This leaves the cost of operating the platform itself, which should be relatively straightforward to estimate and manage, as most cloud service providers have industry-standard tooling for managing and monitoring spend.

It's also important to consider the services and resources that applications depend on. If an application requires a database, non-ephemeral storage, or an https endpoint with a valid TLS certificate, the operator of that application should be able to self-service and configure these services. Furthermore, the platform should take care of managing the lifecycle of these services and their configuration so that we don't need to implement a separate process for provisioning them and tracking their configuration state.

If these concerns are not taken care of, we face the following negative consequences[6]:

- Complex applications that have tight coupling to the underlying platform.
- High operational burden to manage services that applications depend on, and longer wait times if teams cannot self-service the provisioning and configuration of the services their applications depend on.
- Cloud sprawl, with large quantities of left-over infrastructure components whose purpose is unknown and which are risky to delete but must be paid for. This can be mitigated through additional configuration-management processes, but this also leads to higher operational burden.

A platform balances the demands of FISMA and FITARA against the need for teams to self-service their own infrastructure.

Centralize the Platform, Decentralize Delivery

A platform balances the demands of FISMA and FITARA against the need for teams to self-service their own infrastructure. With a centrally operated platform, costs can be managed by creating quotas for systems that run on the platform, and much of the compliance architecture can be implemented at the platform layer and leveraged by the systems running on it.

6. For further discussion of how to manage these issues, see https://18f.gsa.gov/2016/08/10/patterns-for-managing-multi-tenant-cloud-environments/

Containerization further enables a clean separation of concerns at the host level. This includes the platform maintainer's responsibility (for example, applying patches to the operating system and maintaining antivirus and intrusion-detection software) and the platform customer's needs (the ability to self-service application deployments and install necessary dependencies).

A common platform that all vendors use to develop and operate their systems also substantially reduces the effort required to create, operate, maintain, understand, and change these services. This, in turn, makes it easier to change vendors if necessary over the lifecycle of the information system.

The boundary for central control, however, should be the API through which individual delivery teams self-service platform operations including environment creation, application deployment, and services such as database instances. Teams should be free to choose which technologies and toolchains they use, within certain constraints. The system must be able to be built, tested, and deployed in a fully automated way, drawing from source code, scripts, and libraries that are stored in public or centrally maintained version-controlled repositories. Teams should also have the flexibility to choose the software stack installed on hosts. In this way, teams are free to make technology choices appropriate to the skills of the team and the system being developed.

Allowing teams to make their own technology choices carries risks, which can be exacerbated by multiple vendors that will typically be involved in designing, developing, and operating information systems over their lifecycle. Sharing a common platform mitigates some of these, but there are two further practices that can help.

1. Version-controlled repositories

Everything required to build, test, and deploy a service should be stored in public or centrally maintained, version-controlled repositories. It should be possible for a new developer to download a copy of the repository from version control and run a single command to build and run tests locally in a sandbox, and run another single command to deploy the service to the cloud platform. Documentation, automated tests, source code, database migration scripts, and everything else required to deploy or upgrade the service in production should be in this repository. In this way, we ensure new vendors can rapidly get up to speed with systems they are working on.

2. Cross-functional teams

Rather than engaging vendors based on role—one vendor for development, another for test, and a third for operations—we should be engaging vendors with all the necessary skills required to build, test, and operate services in cross-functional teams. In a service-oriented architecture, a good heuristic for decomposing a system into services is that it should be possible to develop and operate any given service with a team of about ten people[7]. These people should include all the necessary skills required to design, develop, and operate the service; a model Werner Vogels, CTO of Amazon, calls "you build it, you run it."[8] By reducing hand-offs and coordination costs, we can substantially reduce the time to design and operate services. We also ensure teams can deliver significant value from early in the lifecycle of a service, get feedback, and continue to make regular changes and improvements for the duration of the service's life.

In general, agency information-security teams will

7. The cloud.gov platform described below was created by a team of approximately ten people, and Amazon and Google both employ teams of about this size to create and operate the primitives that comprise their cloud services.
8. http:// queue.acm.org/detail.cfm?id=1142065

be responsible for ensuring that the platform and the applications running on it are authorized through the application of the NIST Risk Management Framework. They should give service providers and customers maximum flexibility in how they achieve these goals, while encouraging them to leverage existing FedRAMP authorized solutions and open source tools (per OMB memorandum M-16-21[9]) wherever possible to enable re-use of existing solutions.

Design for Multi-Tenancy

Multiple different teams and vendors are typically involved in the creation, operation, and ongoing development of any given system over its lifecycle. It is also common for an information system to be comprised of multiple services. For these reasons, any cloud infrastructure in a federal government has to consider multi-tenancy as a primary requirement, even if it's only hosting a single information system.

Any cloud infrastructure in a federal government has to consider multi-tenancy as a primary requirement, even if it's only hosting a single information system.

Multi-tenancy has an important implication for access control: Users working on one service should only be able to access resources allocated to that service. This is difficult in practice, however. Per-primitive access controls provided by IaaS platforms are not designed for this purpose, and using them in this way is complex, error-prone, and imposes an ongoing maintenance burden.

As a result, vendors who manage IaaS services for agencies typically prevent developers and operators from making changes to the configuration of the infrastructure directly. They must instead request changes through the vendor. This prevents the implementation of the on-demand self-service capability that forms part of NIST's definition of a cloud.

9. https://sourcecode.cio.gov/

Access controls that do not allow changes to be scoped by service also present a challenge to implementing infrastructure-as-code, since any change must be validated to ensure it only impacts the intended services.

As a result of these limitations, many vendors simply make changes manually through the console, recording the changes in a ticketing system. This leads to several problems that are exacerbated by the unique nature of cloud infrastructure:

- Slower lead times for making changes, since all changes must go through a request process that involves manual steps. This negatively impacts both delivery speed and time-to-restore in the event of an incident.
- Increased error rates caused by manual changes.
- More complex audit processes to discover the history of a given configuration item and the reason for its existence (if traceability from a given configuration change to a ticket is even possible).
- The creation of "works of art," which refers to production configurations whose state has been created through a series of manual changes. This, in turn, makes it difficult to patch vulnerabilities, create test environments, triage issues, and reproduce the state of production in the event of a disaster recovery scenario.

An effective multi-tenant cloud infrastructure must satisfy the following requirements:

- The cloud platform must provide an API for authorized users to self-service changes on-demand (thus satisfying NIST's requirements for a cloud).
- The cloud platform must prevent changes from affecting services to which the user requesting the change should not have access.

Good Practice: Implement a Service-Oriented Architecture

In the service-oriented architecture (SOA) paradigm, an information system is decomposed into multiple services, each of which handles a cohesive set of features end-to-end (from API to data management). Each service can be tested and deployed independently. Each service also manages its own data; integrating into a common database schema is strictly forbidden since it creates dependencies and coupling at the database level which prevent services from being deployed and tested independently. Services access each other's information over the network via API calls.

- It must be possible to implement infrastructure-as-code such that all changes to the state of the production system are only made using an automated process purely from information stored in version control.

Fundamentally there are two ways to handle both multi-tenancy and the separation of roles and responsibilities of the service provider and its tenants: through architecture, through process, or both. PaaS takes an architectural approach, which has a higher initial implementation cost and reduces flexibility, but cleanly separates these concerns. IaaS requires a combination of both process and architecture. While the process-based approach allows more flexibility, it is also more error-prone and time-consuming in operation, and requires ongoing communication, collaboration, and improvement work from stakeholders to ensure processes continue to meet their needs over time. Since the design trade-offs are well-understood in the case of PaaS, architectural approaches are preferred due to their substantially lower ongoing maintenance costs.

A correctly implemented SOA provides multiple benefits:

- Higher levels of resilience and availability
- Elimination of complex, failure-prone orchestrated deployments
- Easier testing through the provision of a universal standard way to run every service in a debuggable sandbox
- The ability for different services to be implemented in different languages
- Services that are easier to evolve, understand, and maintain

Some of these benefits—particularly the reduced complexity of understanding, testing, and deploying services—gain increased importance in the context of modular contracting, where services may be evolved and maintained by multiple contractors over their lifecycle. The decomposition of information systems to create an SOA is an essential ingredient to developing a cloud-native architecture, particularly when building large systems with multiple vendors.

Reference Architecture: PaaS

In this section, we present a reference architecture for a PaaS that satisfies the principles presented. The reference architecture uses Cloud Foundry, but this could be replaced with OpenShift or another Kubernetes-based platform. This architecture draws heavily from cloud.gov, a PaaS created to address these principles while also aiming to reduce the time and cost required to authorize information systems built on top of it.

The reference PaaS (see Figure 2) includes three virtual private clouds (VPCs): a production VPC, a staging VPC, and a tooling VPC. The tooling VPC includes the components necessary to deploy and administer the platform, including BOSH director, the UAA user authentication and authorization system, and the concourse CI tool that implements the infrastructure deployment pipeline used to make changes to the configuration of the PaaS. The production VPC contains a public subnet with an ELB endpoint and a NAT gateway, and four private subnets: one for Cloud Foundry's core components, one for the production apps deployed into the PaaS, one for RDS instances, and one for services that can be bound to running apps. The tooling VPC is peered to both the production and staging VPC.

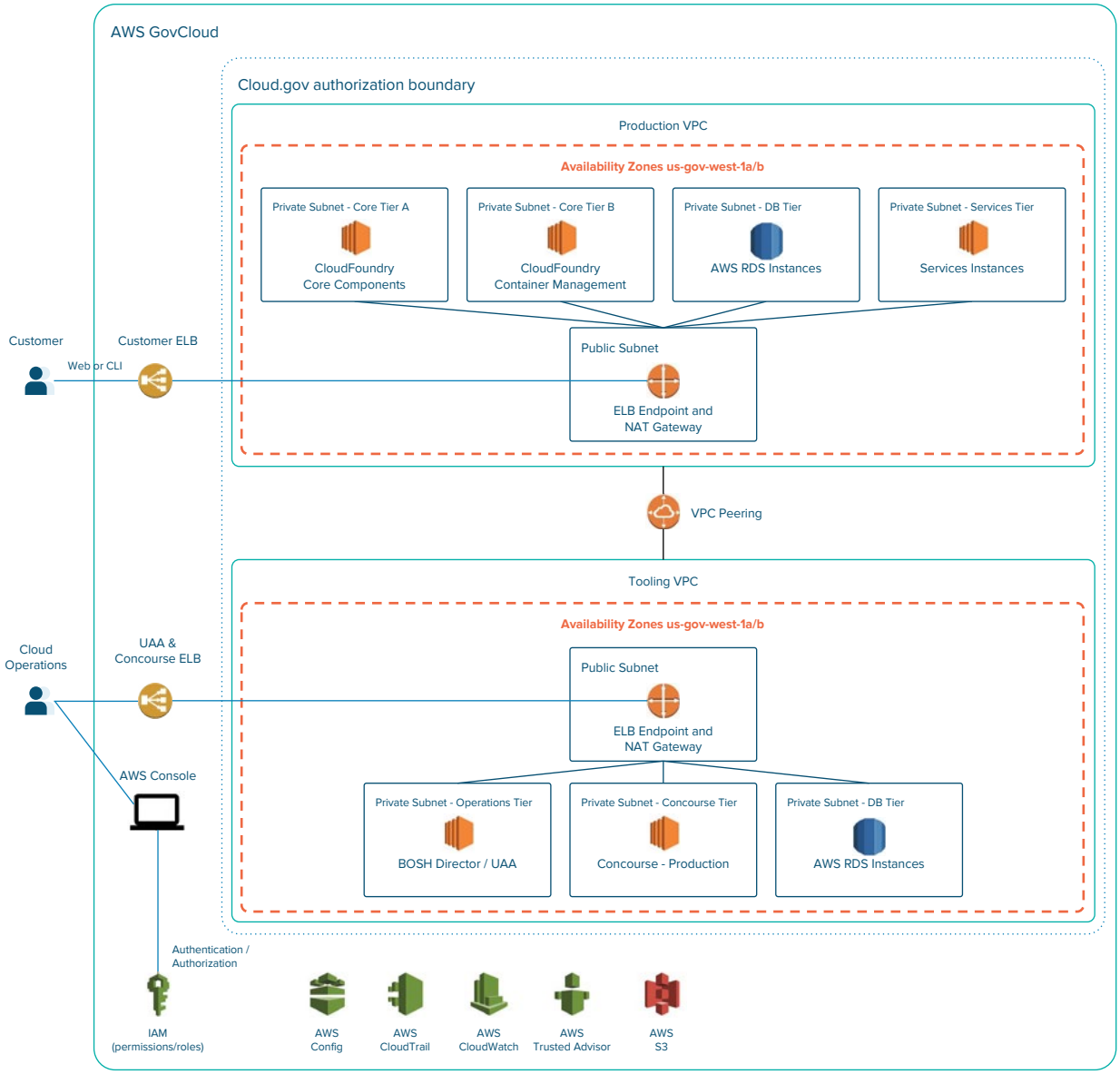

Figure 2: Network diagram for reference PaaS supports multi-tenancy through Cloud Foundry's role-based access-control (RBAC) system.

The reference PaaS supports multi-tenancy through Cloud Foundry's role-based access-control (RBAC) system[10], which logically partitions resources for different customers (see Figure 3). New customers are given access to their own organization, the highest-level logical tier. Organizations contain multiple spaces (for example, prod, staging, and dev). Each space can contain multiple applications. Services such as databases and TLS termination are bound to applications. There are multiple roles associated with

10. https://docs.cloudfoundry.org/concepts/roles.html

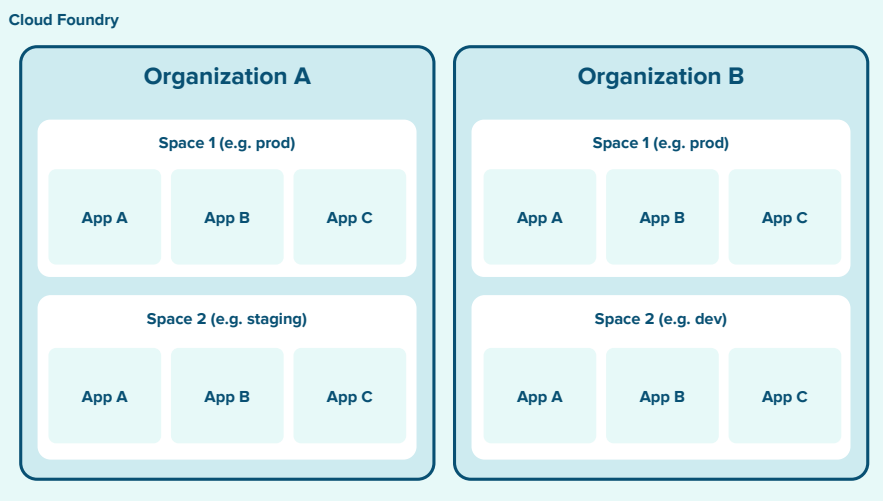

Figure 3: Cloud Foundry's role-based access-control (RBAC) system logically partitions resources for different customers to support multi-tenancy.

each logical tier which grant various privileges to create, modify, or delete objects within Cloud Foundry. As with Heroku, applications can be deployed from developer workstations or through a CI server with a single command.

Cloud Foundry uses its own container scheduler, Diego[11], to deploy and manage user applications. Diego, like Docker, natively supports the OCI container image format, but applications are typically deployed on top of buildpacks—pre-prepared, hardened, technology-specific images supported by the Cloud Foundry team (such as Java, .NET Core, Python, Go). When using a supported buildpack, the PaaS takes care of more controls for your information system.

In the event that the platform needs to be patched—for example, when a new vulnerability is discovered—the hosts can be updated without impacting service. Every application runs on at least two container instances so that if one container instance becomes unavailable, traffic can be routed to the second. This architecture is leveraged for updates and upgrades:

11. https://docs.cloudfoundry.org/concepts/diego/diego-architecture.html

New container instances are created on patched hosts and client traffic is routed to these new instances, following which old instances are shut down. Customers can also upgrade their applications or buildpacks on-demand without service interruption using the same mechanism.

The reference PaaS also provides a number of other services at the platform layer: continuous monitoring, anti-malware, network security, scaling, logging, and alerting. In order to implement these services, a number of open source and commercial components can be employed inside the information system boundary, such as ElasticSearch, Logstash and Kibana (the ELK stack), Nessus, ClamAV, Snort, and Tripwire. The reference PaaS also utilizes AWS' native CloudTrail, CloudWatch, Config, and Trusted Advisor services.

Roles and Responsibilities

The components of the reference PaaS are shown in Figure 4, along with who is responsible for managing each component. Logging should be implemented at the platform layer. The platform should pipe all standard output from applications running on it to a central store (using the ELK stack or NewRelic, for example) with a multi-tenant front-end that lets application owners view or search logs created by their applications.

Monitoring and alerting should be implemented at both the platform and application layers. The platform should have monitoring and alerting for every VM and service it operates. Application operators should install tools for monitoring and alerting such as NewRelic, SteelCentral, and Honeycomb. The platform operator should also have alerting for platform-level incidents along with a portal to inform application operators of the status of the platform.

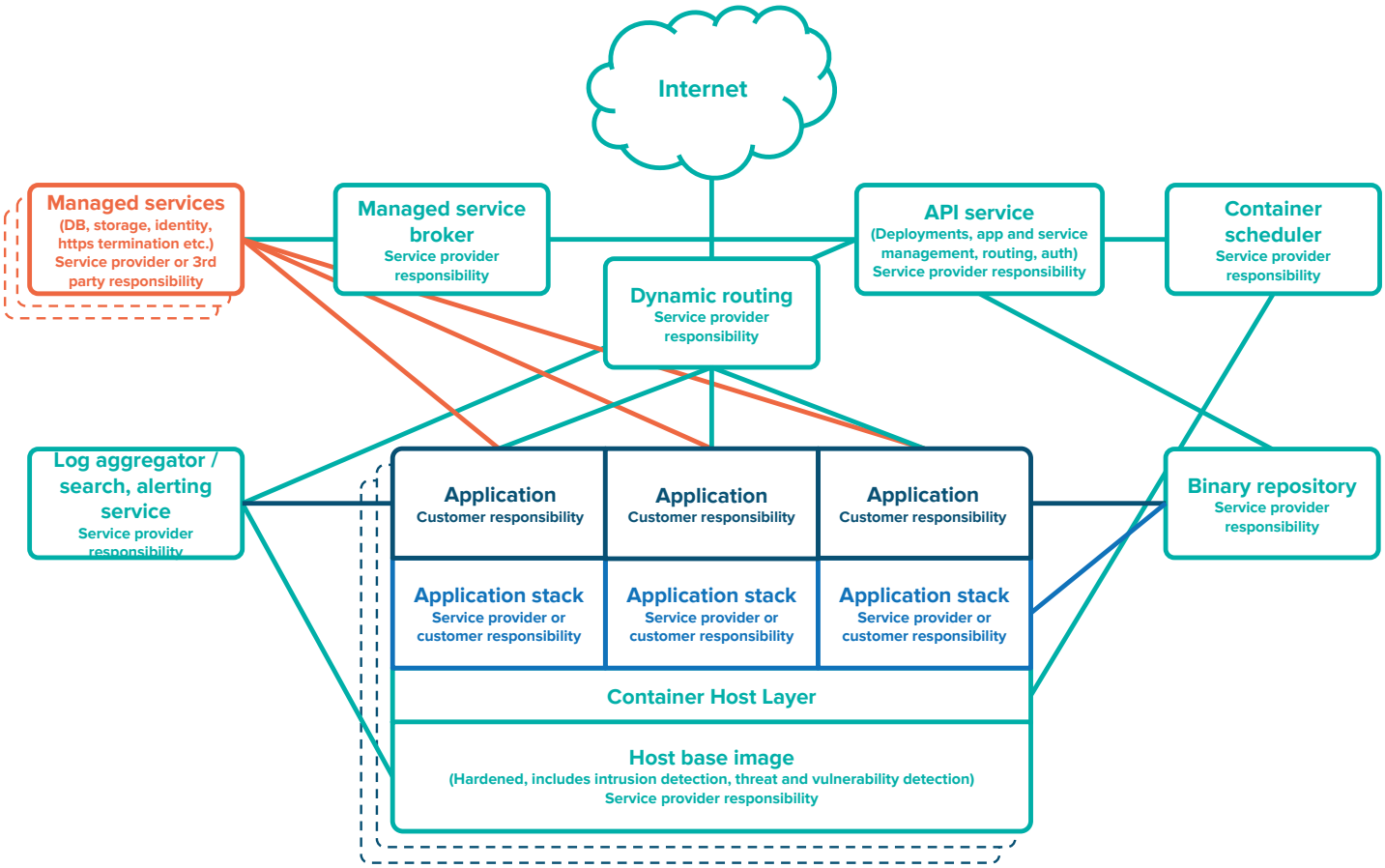

Figure 4: PaaS roles and responsibilities.

The application stack can be either the service provider or customer responsibility. Cloud Foundry provides standard buildpacks for popular technologies, enabling one-click deployment of applications built on supported stacks. Development teams can also self-service custom buildpacks, or even deploy custom Docker images if they require more flexibility. In return, developers are responsible for implementing, documenting, and testing additional controls. The ability to provide this flexibility—while still ensuring that threat and vulnerability scanning and host-intrusion detection are performed at the platform layer—is a key benefit of containerization technology.

The "serverless" paradigm, in which server-side code "is run in stateless compute containers that are event-triggered, ephemeral... and fully managed by a 3rd party" is already in use in several agencies (including defense agencies). In implementation terms, it is similar to the platform described above, but with the cloud service provider taking responsibility for the application stack and with reduced configuration flexibility. In return, more of the compliance architecture can be implemented at the platform level, customers pay substantially less for the service, and they don't need to take care of issues such as patching and upgrading the stack and scaling up and down.

cloud.gov

The goal of the cloud.gov team was to produce a FISMA-compliant platform that provides similar capabilities to Heroku. Work began on cloud.gov in March 2015. The cloud.gov team began preparing for the FedRAMP process in March 2016 and received FedRAMP Ready status in May 2016. In February 2017, cloud.gov received a provisional ATO as a platform for moderate impact system from the FedRAMP Joint Authorization Board consisting of the CIOs of DoD, DHS, and GSA. At the time of writing, multiple agencies have applications running in cloud.gov.

Cloud.gov is a PaaS whose primary components are the open source Cloud Foundry platform running on top of AWS GovCloud. While cloud.gov's System Security Plan (SSP) is not public, a list of source code repositories[1] and network and data-flow diagrams[2] are available.

Of the 325 security controls required for moderate-impact systems, cloud.gov handles 269. An additional 41 controls are a shared responsibility in which cloud.gov provides part of the requirement, and your applications provide the rest. Agencies provide full implementations for the remaining 15 controls, such as ensuring data backups and using reliable DNS (Domain Name System) name servers for websites.

1. cloud.gov/docs/ops/repos
2. diagrams.fr.cloud.gov

Managing Infrastructure-as-a-Service

A modern PaaS provides many advantages such as multi-tenancy support, infrastructure lifecycle management, one-click deployments, traffic routing, automated patching, networking, logging, alerting, and versioning. There are scenarios, however, where it makes sense for application developers to deal directly with the infrastructure layer.

Three common scenarios include:

- Performing a "lift and shift"—taking an application hosted in a data center that doesn't easily fit into the PaaS deployment model described above.
- Developing a new system where a suitable platform doesn't already exist, or the platform doesn't satisfy the NIST criteria for a cloud.
- Developing a new system that can't easily be adapted to fit the PaaS deployment model, such as a batch-processing system.

While new systems should be developed on a PaaS following the guidelines described, IaaS can provide a stepping-stone from a data center to a modern PaaS or serverless architecture. In this model, agencies "lift and shift" existing applications into IaaS to enjoy substantially reduced infrastructure costs before refactoring applications into a PaaS in order to improve reliability and availability and reduce ongoing maintenance costs.

When deploying to IaaS, it may make sense to give application developers direct access to cloud

infrastructure. There are various concerns that must be addressed in this approach.

- **Comprehensive change and configuration management.** The infrastructure-as-code approach to configuration and change management is essential in this model as it substantially reduces the ongoing maintenance cost and risk of managing a multi-tenant IaaS platform. Because all configuration information is kept in version control and an infrastructure deployment pipeline (IDP) is responsible for applying all changes, we can enforce separation of concerns between different teams as well as standards for infrastructure configuration using the change-management toolchain (see Standardization, below).
- **Multi-tenancy.** Multiple teams must be able to use the infrastructure without the possibility of interfering with each other's work. One way to achieve this is to use completely separate logical infrastructure accounts for each information system. This mechanism can be supplemented by creating policies, rules, and templates (see Standardization, below) which can help detect and reject changes that impact other teams' infrastructure.
- **Garbage collection.** We must be able to identify and remove infrastructure that is no longer needed. One approach is to regularly schedule the deletion of all infrastructure that is not tagged with the information system it belongs to. When combined with the use of an automated deployment system and infrastructure-as-code, we should be able to trace every piece of infrastructure back to version-controlled configuration.
- **Standardization.** Standardization of the infrastructure configuration is important for several reasons: it reduces ongoing maintenance costs, expedites the risk-management process, and aids collaboration between teams. The use of standardized infrastructure templates (see Figure 5 for an example) and patterns, combined

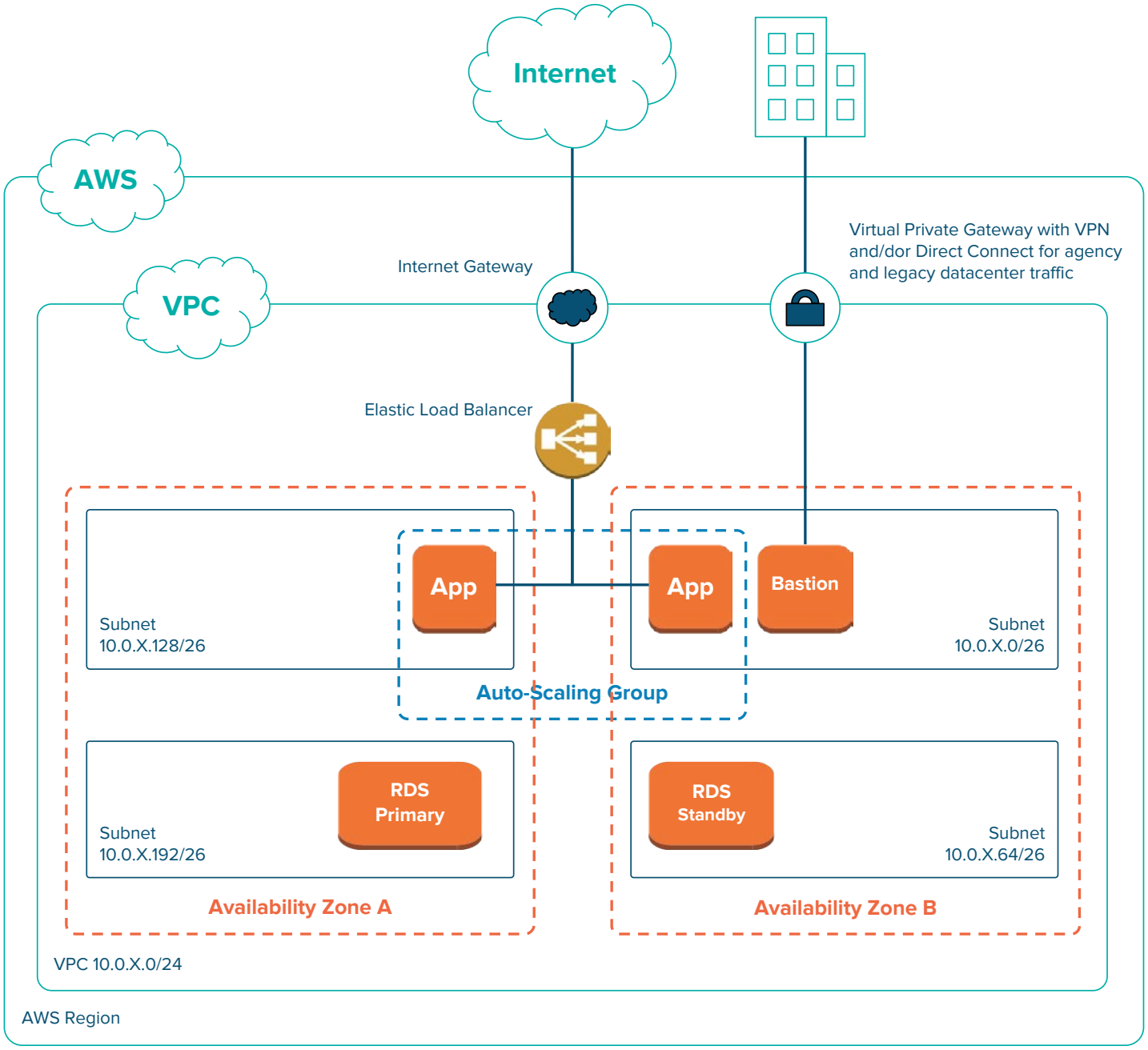

Figure 5: Example of a network diagram for a potential infrastructure template.

with a standard toolchain to create and deploy infrastructure, makes it easier for vendors to work with systems created by other vendors. It also allows templates to be created for the documentation required by the NIST risk-management process, such as the system security plan (SSP) and security assessment report (SAR). Teams should also standardize on processes for managing deployment, rollback, migration, high availability, and scaling—all of which are provided for free in the PaaS model.

While the infrastructure-as-code paradigm provides fewer constraints (and hence more design flexibility) and easier access to developers, there are some trade-offs when compared to the PaaS approach:

- Significantly more controls must be documented and the implementations tested compared to the PaaS approach. Templates can reduce some of this burden, but they cannot eliminate it. This means longer times to achieve Authority to Operate (ATO).
- Maintaining and evolving infrastructure configuration, keeping systems and software patched and up-to-date, preventing configuration drift between environments, and continuous monitoring represent a significant ongoing burden that requires substantial engineering effort. This doesn't only mean more work for the team building the system; Provision must also be made to continue this work when the initial development contract ends. This work requires a certain level of technical skill, experience, and discipline from developers working in IaaS over and above that required from a PaaS-hosted system.

Thus, an IaaS approach is only recommended in the scenarios described at the start of this section, and then only as a transition to a platform-based approach.

A hybrid approach is also possible, in which some information systems are hosted on a PaaS, with others hosted on IaaS. In this situation, systems hosted on IaaS should re-use platform services such as logging, alerting, and hardened OS images wherever possible. Systems hosted on IaaS should also provide a service interface that allows PaaS-hosted systems to access them in a standardized way. For example, Cloud Foundry, Kubernetes, and OpenShift all provide a standard way to expose remote services to applications running on them.

Roles and Responsibilities

An effective multi-tenant IaaS implementation allows delivery and operation of information systems faster, at lower cost, and with higher quality compared to traditional data centers. These goals sometimes come into conflict, however. For example, enabling rapid response times for infrastructure requests (which helps development teams to go faster) might require more operators, which can drive up costs for the service provider.

An effective multi-tenant IaaS implementation allows delivery and operation of information systems faster, at lower cost, and with higher quality compared to traditional data centers.

An important metric for the performance of an infrastructure platform is operators-per-developer. The goal is to keep the number of operators constant as the number of developers the platform serves increases. In order to achieve this, it's essential to make all routine infrastructure operations self-service (for example, creating new environments including routes, and deploying). The only manual step should be the initial grant of platform access to an administrator on each customer team who can then manage all further development accounts.

This in turn requires the use of an infrastructure deployment pipeline (IDP) as described in this document. Ultimately, it's the IDP that makes all changes to the infrastructure, based on changes submitted to version control. There remains the issue of how to separate the responsibilities of the team managing the cloud infrastructure and the teams using that infrastructure. In the PaaS model, this is enforced through architecture. In the IaaS model, this must be done through process. The IDP can help us with this problem by enforcing rules and policies that let teams make changes to the infrastructure for applications they are responsible for, without impacting other teams' infrastructure or shared infrastructure.

Finally, we must consider the machine images. In the PaaS model, an architectural

approach—containerization—is used to separate the responsibility of the cloud infrastructure management team (the base virtual image) from the responsibility of the application teams (the container image). In both cases, the images must be created using a fully automated process from information in version control, using a machine image deployment pipeline. The alternative to using containerization is to have machine images created through collaboration between teams. Since multiple controls from NIST SP 800-53 are implemented through the machine image, this brings teams that contribute to machine image configuration in scope for the continuous monitoring process. The powerful audit and policy-enforcement capabilities provided by the deployment pipeline can ameliorate this burden.

Conclusion and Recommendations

Many federal agencies have begun adoption of modern, agile approaches to software delivery, with the goal of building higher quality services faster and more cheaply. While there are significant barriers to the adoption of this paradigm in the federal government, we offer specific principles and practices that have already achieved success.

Our recommendations for federal agencies balance the need to meet FITARA and FISMA requirements while providing teams the flexibility they require. In this way, agencies can meet their risk-management goals while also enabling the faster delivery, higher reliability and availability, and improved quality that the agile and devops movements enable. Agency teams, vendors, and IT leadership should embrace the power and flexibility these platforms provide.

Key Recommendations

- Create a centrally managed cloud platform-as-a-service (PaaS) that enables teams to self-service the operations they need (such as creating environments, deploying applications to them, and creating database instances) on-demand through an API without requiring a ticketing system.
- Use infrastructure-as-code and containerization to implement as many as possible of the controls required by the NIST Risk Management Framework at the platform layer.
- Use infrastructure-as-a-service (IaaS) combined with the principles and practices described in this document as a stepping-stone approach until a PaaS

can be procured and provisioned, to “lift and shift” existing systems that cannot easily be ported to a PaaS, and for other edge cases where a PaaS is unsuitable.
- Use infrastructure-as-code and deployment pipelines to manage configuration and make changes to your cloud infrastructure.
- Design your platform for multi-tenancy, including built-in role-based access control and the ability to assign and manage quotas to control and charge back costs.
- Eliminate custom configuration, appliances, and middleware and use native cloud primitives and services instead.
- Give teams the flexibility to choose the most suitable technologies and toolchain to create, build, and deploy their systems.
- Ensure you can reproduce both production and testing instances of your platform (including machine images), and every service running on it, from information in version control using a fully automated process. Validate this capability by practicing disaster recovery and failover.
- Don’t procure cloud services on firm fixed price contracts. Instead, treat commercial infrastructure-as-a-service clouds as a commodity and procure with incrementally-funded contracts.

Acknowledgements

Many thanks to Rohan Bhobe, Nicole Forsgren, Mark Hopson, Kenneth Howard, Sha Hwang, Diego Lapiduz, Bret Mogilefsky, Mark Schwartz, Will Slack, and Gabe Smedresman for their feedback on drafts of this white paper, and to Cheryl Coupé for editing.

Author

Jez Humble is co-author of The DevOps Handbook, Lean Enterprise, and the Jolt Award-winning Continuous Delivery. He has spent his career tinkering with code, infrastructure, and product development in companies of varying sizes across three continents, most recently working for the US Federal Government at 18F. He is currently researching how to build high-performing organizations at the startup he co-founded, DevOps Research and Assessment LLC, and teaching at UC Berkeley.

About Nava

Nava is a public benefit corporation working with government agencies to improve their digital services. Founded from a team brought in to help the Centers for Medicare and Medicaid Services (CMS) recover HealthCare.gov in 2013, Nava now works across both federal and state government agencies to modernize critical systems with highly secure, reliable, and fault-tolerant cloud infrastructure.

navapbc.com
hello@navapbc.com

About DORA

DevOps Research and Assessment (DORA), founded by Dr. Nicole Forsgren, Jez Humble, and Gene Kim, conducts research into understanding high performance in the context of software development and the factors that predict it. DORA's research over the last four years and over 23,000 data points serves as the basis for a set of evidence-based tools for evaluating and benchmarking technology organizations.

devops-research.com