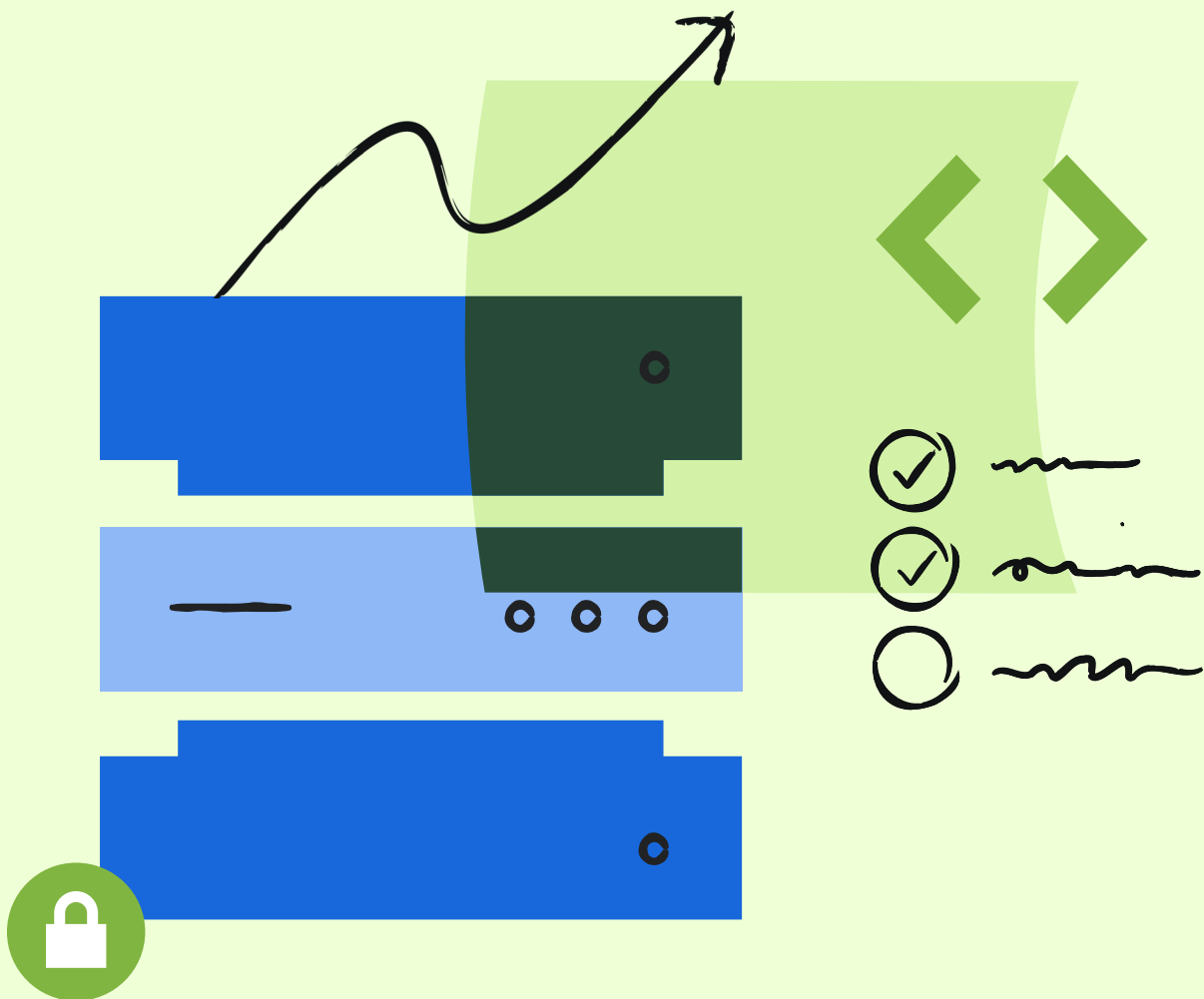


Best practices for DC upgrades



Best practices for DC upgrades

Introduction 1

Deployment best practices 2

- Determine your upgrade cycle
- Use the pre-upgrade planning tool
- Apply configuration as code
- Plan your backup and restore approach

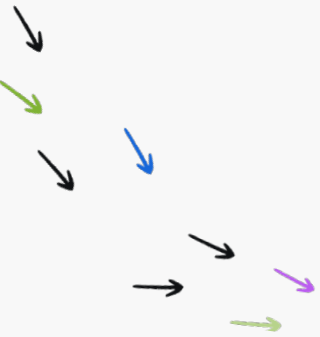
Containerization in secure environments 4

- Virtual Machines vs Containers
- Three options for container registries
- Experience ease of configuration
- Security benefits of containerization
- Compliance benefits of containerization

Kubernetes upgrades 6

- Flexibility and GitOps advantages for deployment
- Configuration and maintenance
- Logs and monitoring
- Cluster scaling
- Get started with Kubernetes
- Recommendations for zero downtime upgrades

Closing thoughts 9



Introduction

In the constantly evolving world of technology, mastering bug fixing and upgrading software is key for organizations looking to ensure the efficiency and security of their environments. If you want to shine a light on the benefits of zero downtime upgrades (ZDUs) and decisions between long-term support and feature releases, you've come to the right place.

This article contains Atlassian's best practices for Data Center (DC) upgrade cycles that customers can apply to their specific setups. This includes:

- establishing regular upgrade cycles
- pre-upgrade planning
- benefitting from configuration as code
- using containers for dynamic data allocation and security
- automating deployments with Kubernetes

These best practices will mostly benefit customers who have multi-node clustered environments. However, even if you're running single-node setups, you can also benefit from these recommendations. We strongly encourage you to explore [clustered environments](#) because they'll enable you to use ZDUs (sometimes also known as rolling upgrades) and increase environment resiliency.

Being proactive and running regular upgrades helps to take advantage of the latest innovations and bug fixes, enhancing system performance. Come along as we explore the essential tools and methodologies for taking your DC upgrade processes to the next level. We'll show you how to keep your systems strong and dependable.

Deployment best practices

Effective deployment practices are essential for smooth upgrades and system reliability. A structured approach can enhance operational efficiency, minimize downtime, and keep systems secure and up to date. This section outlines key strategies to improve the deployment process for your multi-node clustered environments in DC.

Determine your upgrade cycle

Atlassian DC products release feature versions approximately every two months and bug fix versions of our supported feature versions on a fixed monthly schedule, ensuring that updates are rolled out quickly and securely. Your organization can consume those releases in the following ways:

- Feature releases that offer the latest and greatest innovations and bug fixes.
- **Long term support (LTS)** releases that are great for their stability, security patches, and ZDUs. LTS releases come out once a year and are designed for larger, more complex instances that require significant planning and upgrade efforts. A feature release that is designated as an LTS release will receive bug fixes for two years after its release date.

We recommend upgrading regularly, ideally after each feature release, to keep your security in top shape. While upgrading from LTS to LTS once a year provides robust security, opting for regular upgrades ensures an even more secure and efficient environment by leveraging the latest innovations and bug fixes.

It's also crucial to align bug fixing and upgrade cycles with compliance and security policies, which often require collaboration with security teams. For more details, check out our [Data Center security checklist and best practices](#).

Use the pre-upgrade planning tool

Another Atlassian best practice for successful upgrade is using the [pre-upgrade planning tool](#). This tool gives you the following useful information:

- platform support and compatibility to prevent using deprecated databases
- upgrade compatibility checker to identify potential issues such as Marketplace apps that aren't compatible with the target version
- troubleshooting and support checks to reveal issues like duplicated accounts
- data integrity and index checks to help keep your Confluence and Jira systems in optimal condition

We highly recommend using the pre-upgrade planning tool, especially for major upgrades (for example, going from Jira version 9 to 10). The tool will prevent issues arising from changes in supported platforms such as databases.

To get the most out of the pre-upgrade planning tool, it's important to have the latest version of the Atlassian Troubleshooting and Support Tool (ATST), which is bundled with the products.

For environments that don't have access to the internet, you can download the latest version from the [Marketplace](#). You can then upload the app via Universal Plugin Manager directly in the UI and get the latest information about planned releases and other useful support details. For example, you might see a message saying "We've encountered this problem before; check this knowledge base." The documentation isn't included in ATST itself.

Apply configuration as code

By moving towards infrastructure and configuration as code, you can set up a deployment pipeline that is automated and avoids manual errors. Other benefits of configuration as code include:

- reproducibility of the environment and determinist outcomes
- clear change management with versioned artifacts
- possibility to set up multiple similar environments - for example, staging, prod, and dev, allowing to test changes in low-risk environments first
- linters and syntax checking to catch issues such as an extra semicolon and help enhance the code reliability
- standardization and peer review, which make the process similar to a software development lifecycle

Additionally, you can run test simulations. Depending on your pipeline technology, you can even spin up the container, do acceptance testing through the API, and then terminate the container.

We recommend using configuration as code in combination with the containers and Helm charts described below.

Plan your backup and restore approach

Finally, we highly recommend having a solid backup and restore strategy. To avoid data loss, test your backup and restore process even before it's actually required so you can verify its effectiveness and catch errors.

You can use two strategies: XML backup and restore or snapshots. The in-product XML backup is a good option for simpler setups. For complicated and large environments, we recommend snapshots of the underlying shared file system. Check out [Backup and Restoration for Atlassian Data Center](#) for details on your specific product.

This is where containerization can help. In a containerized environment, services are encapsulated and loosely coupled, allowing for independent backup strategies. Your persistent storage, whether Network File System (NFS), Elastic File System (EFS), or others, can be effectively managed, including snapshots. We advocate for ease of recovery as a crucial part of your operational strategy.

Containerization in secure environments

Creating a consistent environment is incredibly important. We maintain official [container images](#) that you can deploy in virtually identical environments across architectures – allowing you to replicate your nodes across any of your available hardware.

Virtual Machines vs Containers

Our internal teams are migrating their acceptance testing from virtual machines or EC2 instances in AWS to containers due to performance and costs.

For example, when allocating resources to a virtual machine, you might choose to allocate 8 CPUs and 32 GB of RAM. To improve storage utilization, you might opt for thin provisioning to allocate disk space on an as-needed basis. While thin provisioning is an option, it does come with the potential pitfall of over-allocating resources. This can be particularly tricky in clusters, where too much soft resource allocation can lead to operational hiccups. Additionally, virtual machines come with overhead because of the hypervisor, which includes both the hypervisor’s operating system and the operating systems of the virtual machines themselves.

On the flip side, containers are more streamlined as they use the host kernel, which provides security benefits and lower operational overhead. For example, when running security scans for containers, you might often spot vulnerabilities tied to the Red Hat kernel. While the container shares the kernel, it doesn’t access it directly; it communicates through the hypervisor, typically using Docker or the Kubernetes controller. So, while the container itself isn’t directly at risk from kernel-related issues, the underlying hardware could be. This situation can sometimes lead to what we call a false positive during container scanning.

Containers also offer dynamic resource allocation, allowing you to scale your CPU and RAM flexibly to match your workloads. You can run multiple workloads on a single host, which really enhances resource utilization. For example, while virtual machines may present performance challenges, a container can effectively perform when deployed on virtual resources or even on bare metal. Overall, you’ll find that containers tend to outperform traditional virtual machines when running on a physical hypervisor.

Three options for container registries

- **Docker Hub**

Atlassian containers are stored in several container registries, with a primary focus on Docker Hub. This public registry is directly managed by Atlassian teams and is integrated with the release pipeline, allowing for real-time releases when new product versions are released.

- **USAF Iron Bank**

The United States Air Force provides a public registry known as Iron Bank, which features hardened containers following strict security guidelines.

- **Private registry**

Organizations may opt for a private registry, such as [Harbor](#), [Docker Registry](#), or [GitLab](#), which can be operated behind a firewall.

Some organizations we've collaborated with take images from Iron Bank or the public Docker Hub registry, bundle them, and transfer them through authorized channels as determined by their security teams. These images are then integrated into their private network and subsequently injected into a private registry. Thus, all of these images can indeed be ported into your private registry and used in a network airtight environment. It's also possible to apply your set of security guidelines and restrictions on the containers further.

Experience ease of configuration

Containers are lightweight and self-contained. You can tailor a Docker Compose YAML file for an individual container or expand it to encompass all resources within a cluster.

In Atlassian, we've implemented what is referred to as "Atlassian Lab in a Box," which uses a single Docker Compose YAML file to provision various components, including a database, a two-node Jira setup, a two-node Confluence setup, a two-node Bitbucket, Fisheye, as well as Bamboo and Crowd. This can all be executed on a local laptop, which is one of the significant advantages of this approach.

The entire process is managed through code configuration, which enhances orchestration's flexibility. You can deploy it to a Docker Swarm cluster, to your laptop, or to Kubernetes. This method offers much more flexibility compared to traditional virtual machines.

Security benefits of containerization

- Image signing and verification make it much easier to verify the origin of the container and avoid a supply chain attack by using signed containers and an included Bill of Materials (BOM).
- Implementation of role-based access control. You can effectively encapsulate role-based access, for example, by separating access for your infrastructure team and your app team and tailoring their roles and responsibilities. This results in a more consistent templating approach.
- Secrets management is very important to remove the need to store the passwords on the system and in the configuration files.

Compliance benefits of containerization

- Scan the included app dependencies before running the apps to detect vulnerabilities.
- Offload audit logging and monitoring to Splunk or an enterprise-grade monitoring system to receive comprehensive audit trails and real-time insights. Containers and Helm charts specifically will make it simpler to configure out of the box.
- Enforce security and compliance policies across environments, reducing the risk of non-compliance due to human error.
- Control data retention and access to minimize unauthorized access risks.
- Enforce image life cycle and tag retention policies. You can track product versions, apply arbitrary tags if needed, test the versions, and then work with the specific tags.

Kubernetes upgrades

Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized apps.

With Kubernetes, you can drive greater agility amongst your teams and enjoy a simplified administrative experience at scale without compromising your organization's regulatory requirements. You can migrate your existing Data Center product deployment to a Kubernetes cluster using the Data Center Helm charts.

[🔗 Check out our migration documentation](#)

Why Kubernetes?

Kubernetes is a mature deployment option, especially when it comes to adapting our support strategies. Over the past few years, we've phased out our support for AWS-specific CloudFormation templates that used to be part of our toolkit. Instead, we're now all about Kubernetes and **Helm charts**, which are the go-to options for deploying our software efficiently at scale.

While Kubernetes offers a host of impressive benefits, keep in mind that it can also come with a learning curve and knowledge requirements and isn't the only option for effective containerization and DevOps practices. There are plenty of alternative solutions out there that don't require diving into Kubernetes. However, for organizations that have the right expertise or are willing to build it, we've seen some significant internal perks from making the switch to Kubernetes.

Flexibility and GitOps advantages for deployment

Kubernetes stands out by eliminating vendor lock-in, allowing organizations to embrace DevOps practices. Kubernetes enables configuration as code and uses Helm charts from Atlassian.

For deployment, options like **Argo**, **Flux**, or custom pipelines enhance operational capabilities. Our **open-source Helm charts** are all about collaboration, and we warmly welcome contributions and feedback. We also have official **Atlassian DC Helm charts** documentation that dives deeper into how to manage and operate the Kubernetes stack.

Kubernetes can be deployed across any cluster, which is a key advantage for us at Atlassian. We're transitioning from Ansible to Kubernetes for internal platform testing, which not only keeps us current but also improves cost efficiency. Running systems on Kubernetes is more budget-friendly, aligning with our mission to optimize infrastructure costs.

We have official support for **Red Hat OpenShift**, which is a significant enhancement for our deployment options. This support is particularly advantageous when considering security aspects, as OpenShift is recognized for its robust security features.

Configuration and maintenance

When it comes to configuration files like `server.xml`, these handy files are created from templates that include your development, staging, and production values. All these values can be stored in Git in a `values.yaml` file. For example, this is a [template for Jira configuration parameters](#). This shows how much flexibility is possible with the Helm charts; of course, you can override only the specific values that differ from the defaults. You can think of this process as being similar to what you'd do with Docker, though the formatting might differ. Essentially, it's like a different dialect of Helm automation for those everyday administrative tasks – like adding certificates to the trust store or setting up built-in health checks.

Speaking of health checks, Kubernetes regularly checks in on the status endpoint to see how the node is doing. If a node goes into maintenance mode or runs into issues, the system can spin up another node. Because adding new nodes to the cluster is much faster than spinning up new VMs, the cluster operation is much smoother.

Logs and monitoring

Some of the other goodness that comes with Kubernetes is the logging and monitoring. Right after installation, you can dive into the default [Grafana dashboards](#), which can be deployed together with the application stack. Plus, with [Prometheus](#) integrated for logging (metrics endpoint exposed), the overall functionality gets a boost in visibility into the health and operation of the application internals. This setup is all about being plug-and-play, keeping in mind that many enterprises have their own tools and processes in place. So, you'll find a lot of flexibility in configuring Kubernetes' product metrics to meet your specific reporting needs.

Cluster scaling

To achieve scheduled scaling with Kubernetes, you can use [Keda](#). When it comes to scaling up, there are some optimization opportunities available while managing a fleet. By considering factors like volume and affinity, you can improve performance efficiency.

One of the things you probably have noticed is the recent [introduction of OpenSearch in Confluence](#). It's a fantastic way forward and a great way of running your indexes. OpenSearch adds more complexity to the overall deployment, but we are taking care of it in the Helm charts, helping you scale up and scale down the OpenSearch clusters.

It's common in Kubernetes to scale applications horizontally without limitation. This isn't possible with Atlassian products due to existing limitations with the sticky sessions. When a user logs into a particular node, they're stuck with it until that node goes down, at which point they are redirected. This explains why auto-scaling is currently not seamless; it doesn't function on a round-robin basis across the nodes.

Get started with Kubernetes

If you're looking for inspiration to configure Kubernetes in your environment, check the [Deployment Automation for Atlassian DC](#) and the corresponding [GitHub repository](#). This proof of concept contains Terraform scripts to deploy from zero to running DC products with datasets (optionally), Kubernetes cluster, and network components.

This Kubernetes project is designed for Atlassian partners to run the [DC App Performance Toolkit](#) and isn't officially supported. We don't recommend using the scripts to deploy production instances but you can use them for inspiration and to deploy testing stacks.

Recommendations for zero downtime upgrades

- Use your stateful set to manage ZDUs. When it comes to pod deployment, the health check plays a crucial role and works this way:
 - if the health check is positive, then that's great; your endpoint is created and healthy, and your actual pod is now accepting the requests
 - if there are any issues – such as ongoing indexing – then the pod is not up yet and does not receive requests
- Test your upgrade in non-critical environments. We absolutely recommend testing in staging or UAT environments for all your upgrades. This practice helps reduce errors, especially misconfiguration such as forgetting to turn off mails or switching configs for staging and production. Having and leveraging Kubernetes or containerization is an easy way to do this testing and make it deterministic.
- Establish a maintenance window, even though there is a ZDU. This is especially true if you need to scale down a node rather than add a node. Depending on the traffic that you're servicing, it's going to get slower, and you might want to give your users a heads-up that that's happening.
- While ZDUs are triggered through REST endpoints, it's a good practice to log into the system and check out the ZDU screens for any lengthy tasks running on a node and the count of active users. This insight helps in making smart decisions about whether to postpone operations or give users a heads-up if any hiccups occur on the nodes.

[🔗 Check out product upgrades for detailed steps](#)

Closing thoughts

Navigating the ins and outs of software upgrades and containerization is key for organizations looking to keep their environments strong and secure. By implementing best practices like regular upgrade cycles, using pre-upgrade planning tools, and adopting configuration as code, you can enhance your operational efficiency and minimize downtime in your clustered environments. Moving from virtual machines to containers, along with the smart use of Kubernetes, brings fantastic benefits in managing resources and scaling up. As your organizations continue to grow in their tech journeys, embracing these strategies will not only simplify processes but also enhance security and compliance, leading to a more agile and resilient infrastructure.