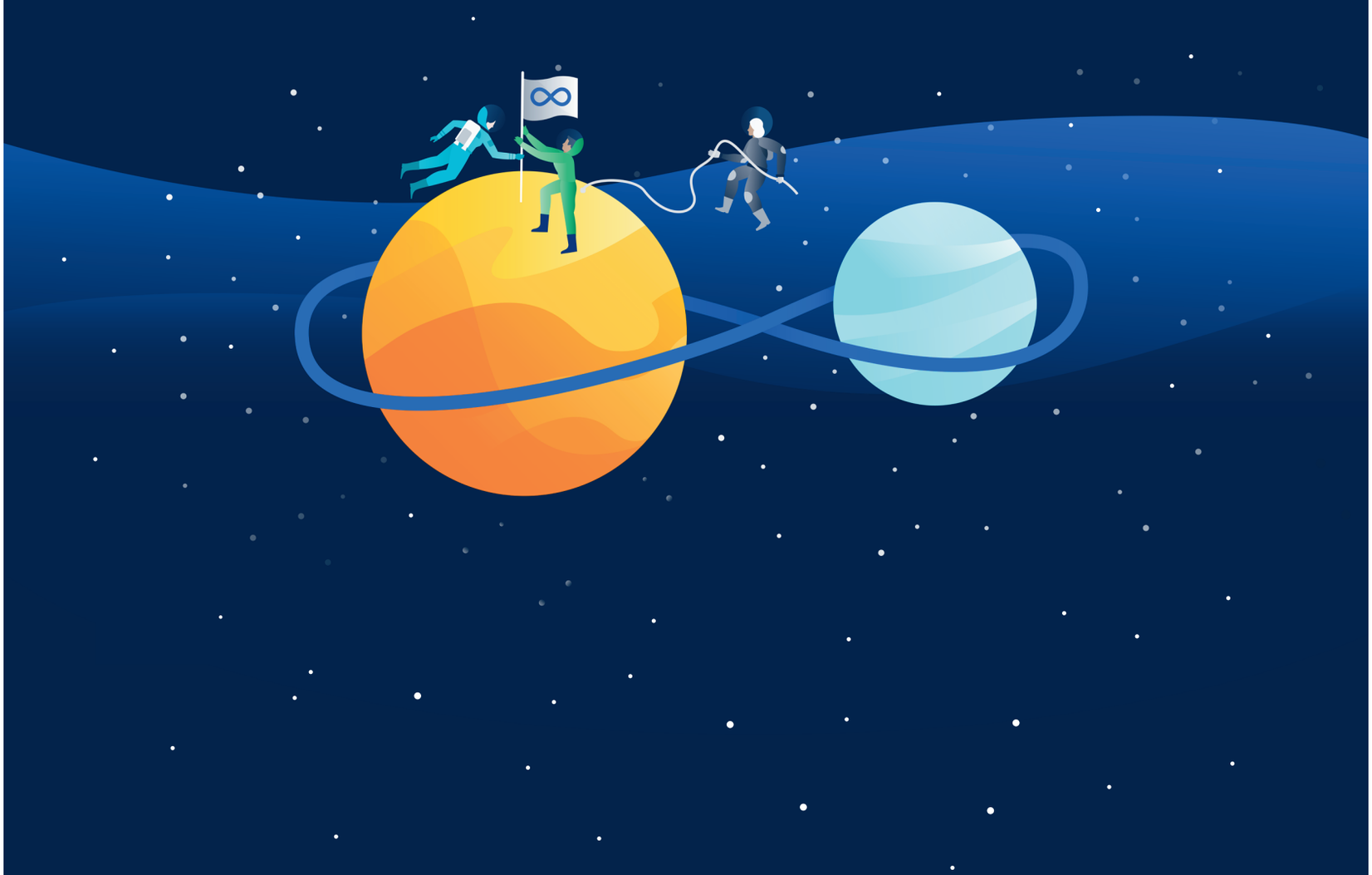
ATLASSIAN



DevOps

with Atlassian

Contents

01

What is DevOps?

Five years ago, Marc Andreesen proclaimed that software is eating the world. After all, what company isn't a software company? Case in point:

MODERN CARS CONTAIN HUNDREDS OF MILLIONS OF LINES OF CODE

Far more than all of Facebook, from Zuckerberg's dorm years to today.

PIZZA DELIVERY HAS GONE HIGH TECH

With advanced mobile applications for placing orders and tracking deliveries, Dominos Pizza has increased its IT workforce by 240%.

NIKE IS TURNING FOOTWEAR INTO A FULLY CONNECTED PLATFORM

Nike is turning footwear into a fully connected platform by integrating shoes with lifestyle and fitness applications.

8,000+ new developers hired in the past 2 years

Nike is turning the shoe into a connected platform

UPS plans routes with real-time traffic data

240% increase in IT workforce for Domino's Pizza

Player statistics and analytics platform added on NHL.com

Companies that practice DevOps enjoy:

- 200x more frequent deployments
- 2555x faster lead times
- 24x faster recovery
- 3x lower change failure rates

Puppet Labs 2016 State of DevOps Report

Old-school development models just don't hold up to such high-demand, high-growth environments. Traditionally, Development and Operations teams work separately in silos, hindering the ability to move fast. The response to this contentious relationship was a movement called DevOps. It's a fancy phrase for a simple idea: your dev and ops teams work better together. It advocates for better communication and collaboration so that developing, testing, releasing, and running software can happen more rapidly and reliably. Instead of delivering big, infrequent releases (once every 3 to 9 months) like traditional development teams at major enterprises, DevOps takes a "continuous delivery" approach. This means releasing small, incremental improvements regularly—often even several times per day. The results are enormous, and go far beyond the operational.

These results aren't limited to major enterprises with billion-dollar dev teams, either. You can achieve them yourself, no matter how small your team is. The #1 success factor is teamwork. At Atlassian, the key to faster, higher quality releases is a strong relationship between our dev and ops teams, and the right tools and processes in place to support them. So what does that look like at Atlassian, and how did we get started?

> **"High-performing organizations are decisively outperforming their lower-performing peers in terms of throughput and improving quality is everyone's job.**
>
> Puppet Labs 2016 State of DevOps Report

PRO TIP

The #1 success factor is teamwork.

At Atlassian, the key to faster, higher quality releases is a strong relationship between our dev and ops teams, and the right tools and processes in place to support them.

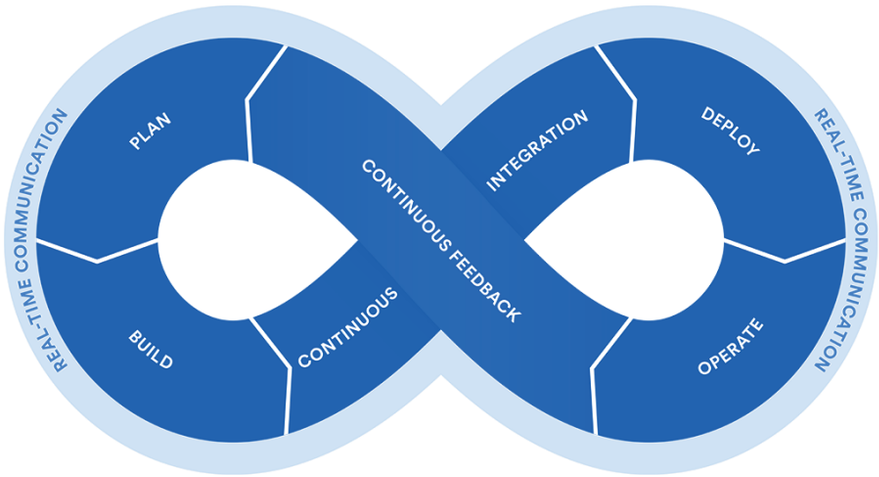
02

DevOps and Atlassian

At some major big box retailers, really heavy items have “team lift” stickers on them to indicate when several employees need to help move the items from shelf to shopping cart. “Team lift” is actually a perfect analogy for the entire DevOps methodology, since DevOps isn’t any single person’s job—it’s everyone’s job.

At Atlassian, we use our own products to understand our various use cases, and provide additional testing before we release them to our customers. In short, we dogfood our own products.

In this ebook, we’ll cover each step in detail, and exactly how we use each Atlassian solution. For now, let’s start with our process, which looks a bit like a hot tasty pretzel.



- First, we plan the features we will deliver to our customers. We use Confluence and Jira Software to organize customer feedback and list requirements. We create issues in Jira Software to start tracking the stories and epics we define for each software project.

- Then, we build the software—writing code and running tests until we get it right. Bitbucket lets us create branches for each new feature we need to create, and it also allows us to code more collaboratively, since we can use pull requests to facilitate faster reviews, and comment inline and hold conversations between our developers right within the code.

- We continuously integrate new features back into a master branch for deployment. Bamboo makes this easier, helping us automate builds, tests, and releases along the way. It really speeds up deploying to AWS, too—we use Docker and Bamboo together for even faster, more efficient deployment.

- Jira Software's release hub also gives us full visibility across all our branches, builds, pull requests, and deployment warnings, so we can release with confidence.

- Once we've deployed a new feature into production, it's time to run and operate it. At Atlassian, our developers are fully responsible for the features they build, so using Jira Service Desk helps them track and resolve incidents faster. Statuspage increases transparency by enabling us to communicate incidents to affected users and to keep them informed throughout the resolution process. We integrate Jira Service Desk with Statuspage to display real-time status information right at the top of the service desk portal to deflect duplicate tickets being submitted. We use Confluence to manage run books, knowledge base articles, and related documentation at every step.

- We deliver continuous feedback (via reports, tickets, etc.) to our development teams, so they can plan new releases, fix bugs, and deliver faster, more reliable software to our customers. With Jira Service Desk, we can even request customer feedback from both internal and external users.

- We learn and improve from every process – especially incidents – and make sure we communicate to our users what we are doing to make sure the same problem doesn't happen again. We are again very candid with this information through post-incident reviews (PIRs) that we are able to draft as a team in Confluence and then deliver to our customers via Statuspage.

Throughout the entire lifecycle, Hipchat is the secret salty coating to our pretzel. It adds an additional layer of collaboration on top of our already collaborative processes and technology by letting our teams swarm on incidents, wherever they are, via desktop, mobile apps, and even wearables.

That's just the basics, though, and you came here for details. So let's dive in.

03
Building Products, DevOps Style

Let's say your engineering team has gone Agile. They work in sprints, collaborate, and are building a lot of great features. But there's just one catch: you still have to wait for the release train to leave the station, and customers aren't getting value fast enough.

We'll show you our best practices for building products, DevOps style. Let's start with feedback; because no matter the product, your success is solely based on your users.

HOW TO GATHER FEEDBACK—AND USE IT TO SHAPE AND BUILD FEATURES

We've learned over the years that the easiest way to make our product better is to listen to the people that use it. Thousands of companies use Hipchat, and thousands of Atlassian use it internally, too. You can collect feedback from just about every source imaginable.

- Ask for in-product feedback
- Collect user feedback from Jira Service Desk
- Monitor social media channels like Twitter and Facebook
- Use Apdex scores to monitor whether our users are satisfied with Hipchat's response times
- Gather monitoring data from third party solutions like Datadog and New Relic

What do we do with all that feedback? Here's what we do with it. Keep in mind: this may or may not work for your team, but is nonetheless a useful starting framework that you can tweak.

We send all feedback to Hipchat as notifications. For example, we get a ton of tweets:

> hey @Hipchat, any news about deeper Jira integrations? issue links!
>
> Eric Wood @ejwood79

We route them, along with all our other social media mentions, bug reports, etc. into dedicated Hipchat rooms where the whole team can discuss each notification and help shape our backlog.

Important feedback, like bugs, is then converted into a Jira Software ticket—which we then prioritize into the backlog. If there's a new feature, we'll typically create a Confluence page to spec out goals and requirements.

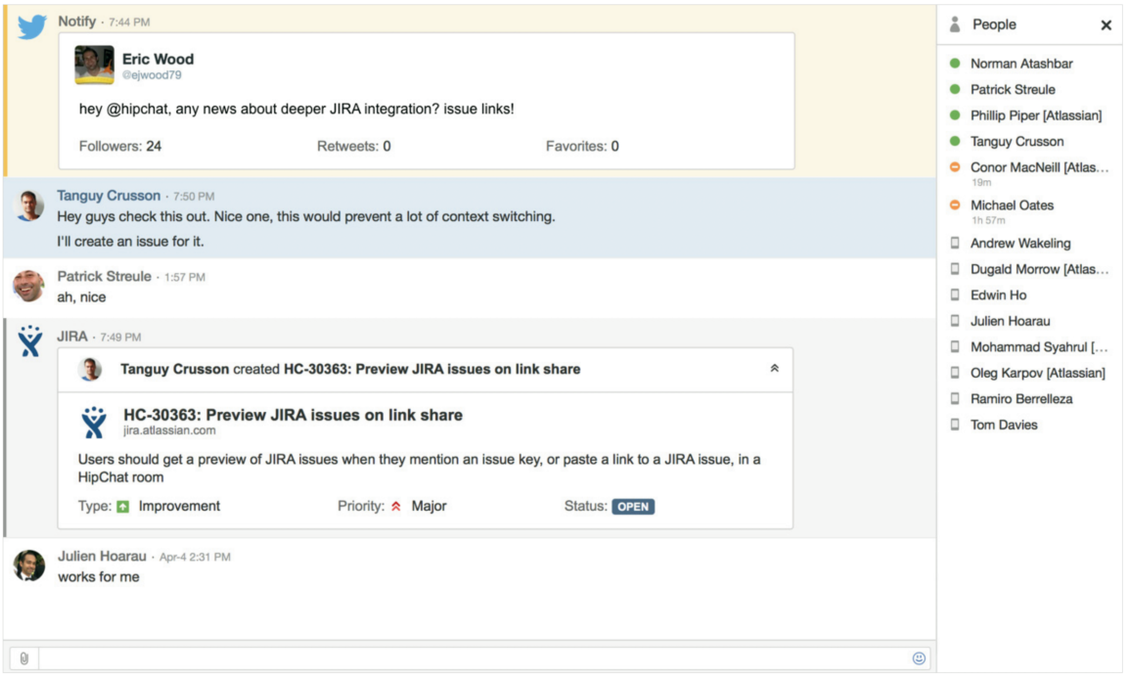
In either case, we make sure to always listen to our customer feedback, wherever they are, and take action when possible.

PLAN TOGETHER IN SPRINTS

So, how exactly do we plan what we're going to build? Our small development teams regroup and meet for an hour every week. We use the hour to:

- Demo everything that was built in the previous week to keep the team informed and connected.
- Review the objectives and sprint goals we established the previous week and agree on whether we achieved them.
- Define our objectives for our next sprints. At Atlassian, a sprint objective isn't the same thing as a ticket. A sprint objective is a unit of work that you have to be able to demo to the team, or ship to production at the end of the sprint.

After the meeting, we break out. With our new objectives in hand, our developers can go through all the issues in our backlog and pick out the ones that will help us achieve the sprint objectives we took on during the meeting.



The end result is complete buy-in from the team. Everyone is fully involved in defining our goals, how we are going to achieve them, and how we are dividing the work.

SPIKE EARLY AND OFTEN

You're probably familiar with the term "spike" in agile development. A spike is a short effort to gather information, validate ideas, identify early obstacles, and guesstimate the size of initiatives. Instead of building a shippable product, we focus on end-to-end prototyping, to arm us with the knowledge we need to get the job done right.

At the end of each spike, we have a better idea of the size and technical obstacles we will encounter for each initiative, and we categorize them: Extra Small, Small, Medium, Large, Extra Large, or Godzilla.

We regularly rotate between normal sprints and spikes, and hold regular "innovation weeks" that result in really amazing prototypes and insights around project scope and approach. Most teams at Atlassian hold innovation weeks, too, and they love to write about them.

KEEP EVEN THE BIGGEST CHANGES SMALL

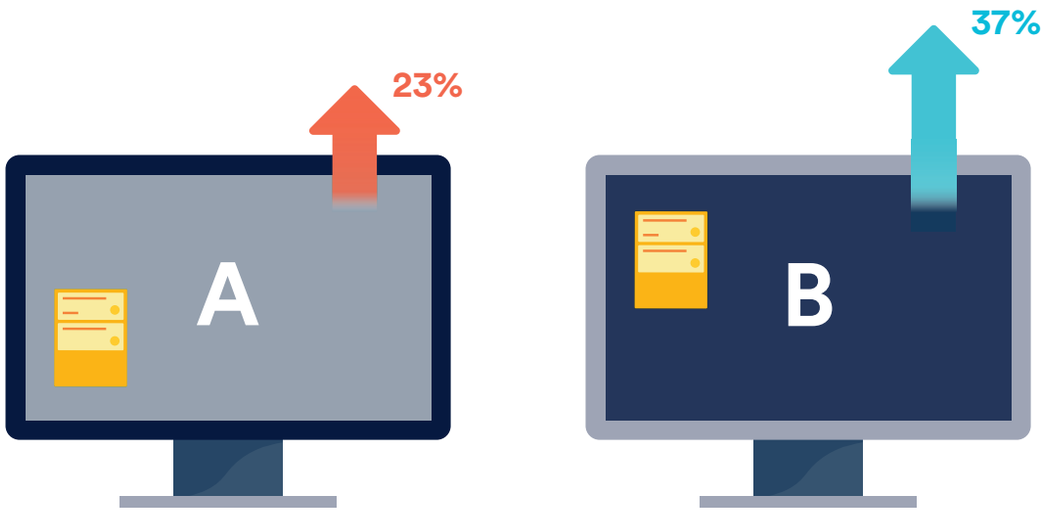
Instead of shipping big things infrequently, ship small changes very often. It makes it very easy to roll back a particular change if we need to, or even better: fix and roll forward, and it helps us iterate fast.

For really big changes—like highly anticipated new features, for example—we still take a "start small" approach, setting "step by step" goals and running frequent A/B tests and experiments to see what our users like best.

> **Instead of shipping big things infrequently, ship small changes very often.**

To test, we divide our users into cohorts. For example, cohort A might see one version of a product feature, and cohort B might see a slightly different version. We look at the usage data to see which version of the feature is performing best against the goals we defined during planning—and we keep iterating and testing until we get to the best version of that feature.

A tool we use during these testing phases is Launch Darkly, which lets us release new features to small segments of users, gather feedback, and then gradually increase the audience size until we've fully deployed. We often start with just 5% of users running the new feature—and then slowly increase by 10 or 15 percent increments after each feedback and revision cycle.

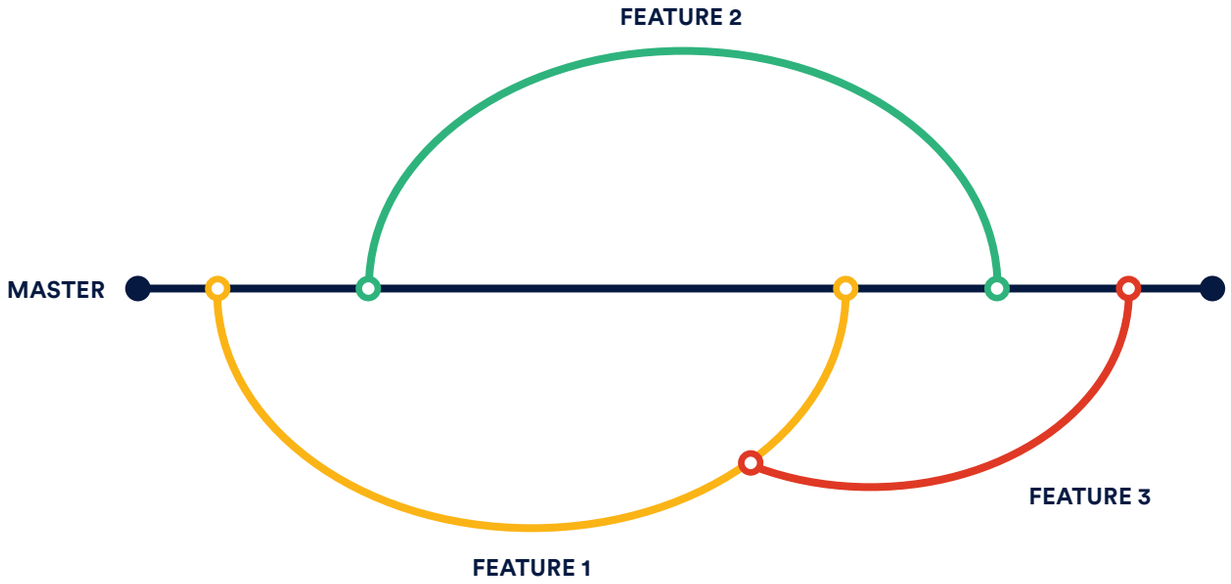


GIT + BITBUCKET + BAMBOO = AWESOME AUTOMATION

We're heavy users of Git and Bitbucket, using feature branches to make continuous integration far more effective. Any feature however small, translates into a feature branch, which is automatically tested via our Bamboo builds.

After we test a feature branch, we create a pull request to merge it back to the master branch, and we select a minimum of two reviewers from our team to review and verify the code. Once you get a green build and 2 approvals, you're good to go.

Since our master branch is what gets shipped to production, we require that the master be "green"—no known bugs, issues, or errors—at all times. If a build goes "red," that means all hands on deck, and the entire team has to drop everything to fix the build.



ENCOURAGE ACCOUNTABILITY

A big difference between our team and many other DevOps teams is our ownership model. We're big on "you build it, you ship it, you run it", meaning the team that is responsible for writing a feature also becomes the team responsible for deploying it and providing ongoing maintenance once it's live.

But isn't that going to introduce a lot of issues in production? In fact it's quite the contrary: It encourages every developer to build the very best version of something, and gives each of us a vested interest in its ongoing success.

What this leads to is 100+ developers being able to ship to production at any point in time. This is made possible with the right process and especially the right tools. We use Chef and Puppet for automation, and developed a number of Chat Apps (Hipchat add-ons) to help us coordinate this process.

Finally, accountability for us also means keeping our users informed of what's going on. Occasionally, bad stuff happens, and glitches have the potential to impact all of our users. We rely on Statuspage to keep everyone up to date on the status of our services through honest and frequent incident updates. We also integrate Statuspage with Hipchat which pipes real-time status updates into dedicated rooms so our DevOps teams are able to stay on the same page, no matter where they are located.

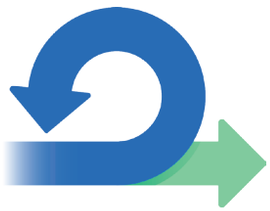
PRO TIP

DevOps isn't any single person's job.

It's everyone's job.

04
Continuous Delivery for Infrastructure

It's not just development teams that can use DevOps practices. You can apply the same practices to your hardware and configuration work too. At Atlassian, we've built a team of a dozen employees (called Build Engineers) that are dedicated to helping our developers code faster, by giving them the best hardware and infrastructure services possible. We oversee our continuous integration service (Bamboo), our artifact storage and retrieval service (Sonotype Nexus), and all the hardware, server configurations, applications, and services that glue them together and provide a smooth experience to our dev teams.



STABLE AND FAST CI

ARTIFACT STORAGE AND RETRIEVAL

ASSOCIATED TOOLING

Let's take a deeper dive into the technology and processes we depend on, and a few top tips for running a Build Engineering team more efficiently and effectively.

GATHER FEEDBACK FROM DEVELOPERS

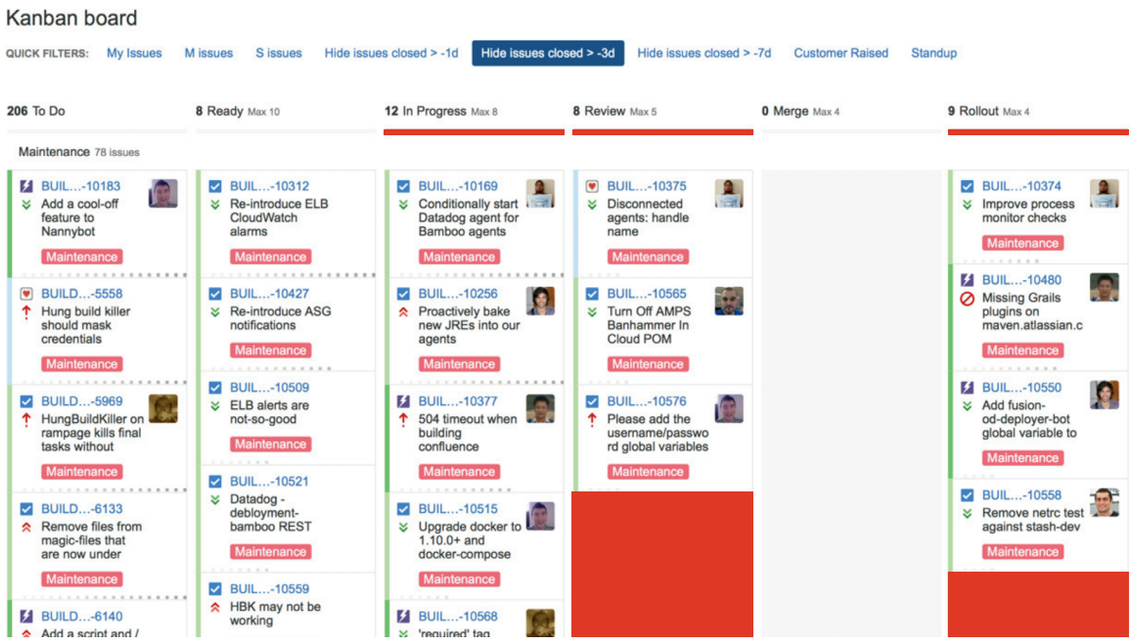
Our customers are Atlassian's developers. We use Jira Service Desk to create our own engineering service desk, and that's how the developers submit requests and feedback.

"WALK THE BOARD" DURING STANDUPS

Each morning, we have standups just like most software dev teams, where we go through all the issues in flight using our Kanban board in Jira Software. Each issue is categorized as:

TO DO	REVIEW
READY	MERGE
IN PROGRESS	ROLLOUT

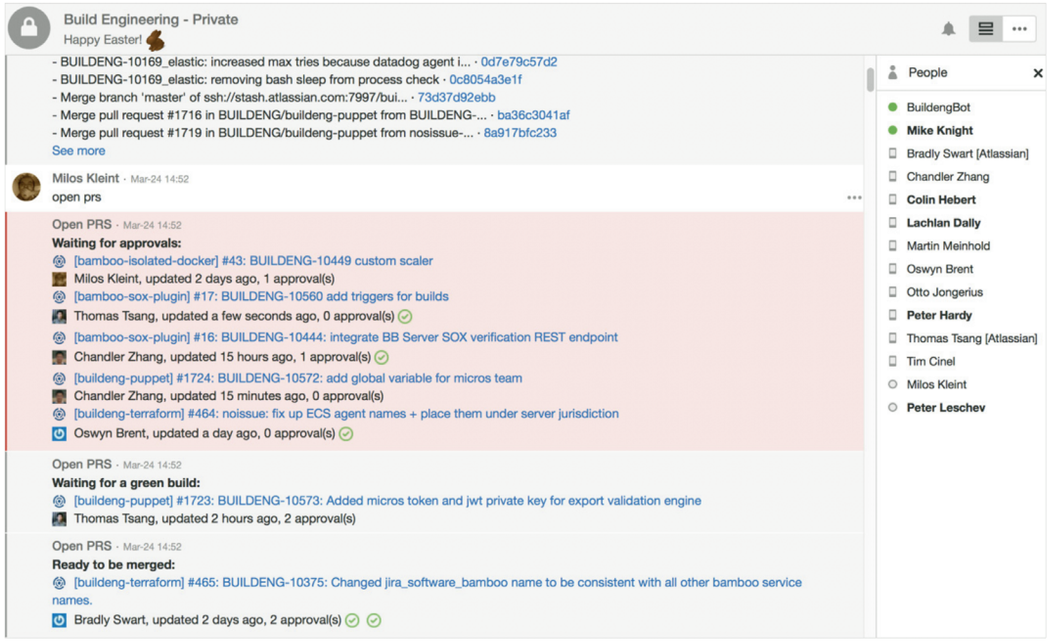
We set a maximum threshold for the number of issues that can be in each status column. Below, you'll see a few columns that have "gone red" because we've exceeded our defined thresholds. This helps us determine in our standup that we need to finish the work in that column before we pick up anything new.



PULL REQUESTS: SWARMS, APPROVALS AND KEEPING THINGS GREEN

We create branches for any hardware or configuration change, no matter how small, exactly the same way that our developers do. Every single pull request is linked to a Jira issue, and we manage the pull requests in Bitbucket, requiring two approvals from our colleagues (plus a green feature branch build) to move forward.

Our team also has a Hipchat room where we wrote a bot to keep track of all our pull requests. It shows all open pull requests, and how close they are to being merged. We leave it up to the team to swarm over the pull requests and jump in and provide feedback for the ones they feel most qualified to review. Everyone pitches in and works really well to move us through the pipeline faster and knock out our in-process work. So Jira Software, Jira Service Desk, Bitbucket, and Hipchat are a big part of our day-to-day operations.



FAVORITE PIPELINE TOOLS

You might be wondering what tools to use for handling software, configuration, and hardware deployments. Here are a few of our favorites:

SOFTWARE PIPELINE

Just like our software development team, we use Bamboo on the infrastructure side, to manage and run our build plans and deployments. We use Bamboo to manage Puppet, where we write new modules to install and configure components on our servers, like a model to install the SSH keys from everyone on our team.

Vagrant lets us spin up test servers easily, which we apply Puppet configurations to for testing purposes. Puppet and Vagrant integrate really well, and the combination makes it really easy to test new AWS server configurations automatically.

Cucumber is great for testing, too. We use it to confirm that our agents are installed properly, and that the changes we have made haven't broken anything.

Once we're finished testing a configuration or change, we deploy our new Puppet tree out to production, and Hipchat will automatically post a notification to the issue assignee to verify that the change is working in production, and to also close the issue in Jira.

As always, Bamboo shows the status of the build, and the details of each release, like which environments it's been deployed to, and which Jira issues are addressed in each build and release.

HARDWARE PIPELINE

Bamboo manages everything in our hardware pipeline as well, from start to finish. Since we make quite extensive use of Amazon Web Services (AWS), we use Terraform to manage our hardware infrastructure. We love it because it allows us to use software best practices and workflows to make changes to our hardware.

For example: Changes we request to our hardware infrastructure through Terraform have to be verified through pull requests, and deployed through a continuous delivery pipeline—the same process our software developers have to follow for their work. This keeps us consistent about how we manage quality across the board.

Here's a quick example of what Terraform code looks like, just in case you're curious:

```
resource "aws_instance" "nat" {
    ami = "${var.aws_nat_ami}"
    availability_zone = "us-east-1b"
    instance_type = "m1.small"
    key_name = "${var.aws_key_name}"
    security_groups = ["${aws_security_group.nat.id}"]
    subnet_id = "${aws_subnet.us-east-1b-public.id}"
    associate_public_ip_address = true
    source_dest_check = false
}

resource "aws_subnet" "us-east-1b-public" {
    vpc_id = "${aws_vpc.nat-vpc.id}"
    cidr_block = "10.0.0.0/24"
    availability_zone = "us-east-1b"
}
```

Here, we're basically setting up a new NAT server on AWS. We use code to set all the parameters, like subnet, etc. We can feed an entire hardware configuration into Terraform, and it will figure out all the API calls it needs to make to AWS to change our server topography from its current state to what is specified by the code. Then, we can ask Terraform to execute the plan and make those changes. It's magical.

We track all of these releases with Bamboo, just like we do our software. Bamboo deploys each Terraform release into our staging environment first, and then our production environment once we're ready. Bamboo is also used to see which releases have been deployed across what environments.

THREE CORE CONCEPTS TO REMEMBER

Nothing changed the game more for us than the idea of "infrastructure as code." It's allowed us to adopt software development's best practices, but apply them to hardware and configuration management, and it's greatly improved the stability of our platform. Doubling the number of servers dedicated to running Bamboo at Atlassian was pretty much the same amount of work as just adding one would have been in a less efficient model.

Our team follows three basic principles that pretty much any engineering team can adopt:

Automate everything

It's critical that our builds work. If we don't test them thoroughly, we can't be confident they will work. Automated testing helps prevent regressions, gives us confidence in our changes, and makes continuous delivery possible for us.

We automate notifications, too, and just about anything we can to reduce human error and make sure we don't miss important tasks.

Finally, with more automation, we can keep our team smaller. That means less communications overhead, and more speed—which is exactly our team's charter.

Stay focused on continuous delivery

Stable hardware and reliable configurations are critical to making sure our developers can get their work done. So we follow continuous delivery best practices, just like they do:

OUR CODE IS ALWAYS RELEASABLE

Our master is always "green" and stable, so it can be released at any time.

WE RELEASE FREQUENTLY

This reduces risk, since there are only small changes from release to release, and we can revert easily as needed.

WE FOCUS ON FAST VALUE DELIVERY

Since our users are Atlassian developers, we want them happy. Continuous delivery ensures we get improvements and fixes out to them as quickly as possible.

Embrace infrastructure as code

Simply put, this just means that we execute code to automatically configure servers, apps, and more instead of manually configuring them via other less efficient methods like in-tool configuration screens and wizards. We can literally use code to hammer out commands like "give me N servers configured with apps X, Y, and Z", and then use review and approval workflows to reduce human error significantly.

As a result, we're able to perform 10x more builds, without adding a single person to our engineering team. We can deploy with far higher confidence, and more independence.

> **As a result, we're able to perform 10x more builds, without adding a single person to our engineering team.**

PRO TIP

With more automation, we can keep our team smaller.

That means less communications overhead, and more speed.

05
Handling Incidents at Atlassian

But what about when things aren't working as planned—like when a feature rolls out that isn't performing optimally? That's where our Service Operations team comes in. Our job is to make it easier to spot and fix incidents, and prevent them from happening again in the future.

We use ITIL as the basic framework for our service management practice. It gives us a standard set of terminology and processes that make it easier to communicate and work together. More specifically, ITIL provides a strong foundation for how to classify incidents, define severity, and perform and track investigations into root cause and more.

Let's take a look at how Atlassian handles incidents when the poop (or anything else, really) does eventually hit the fan.

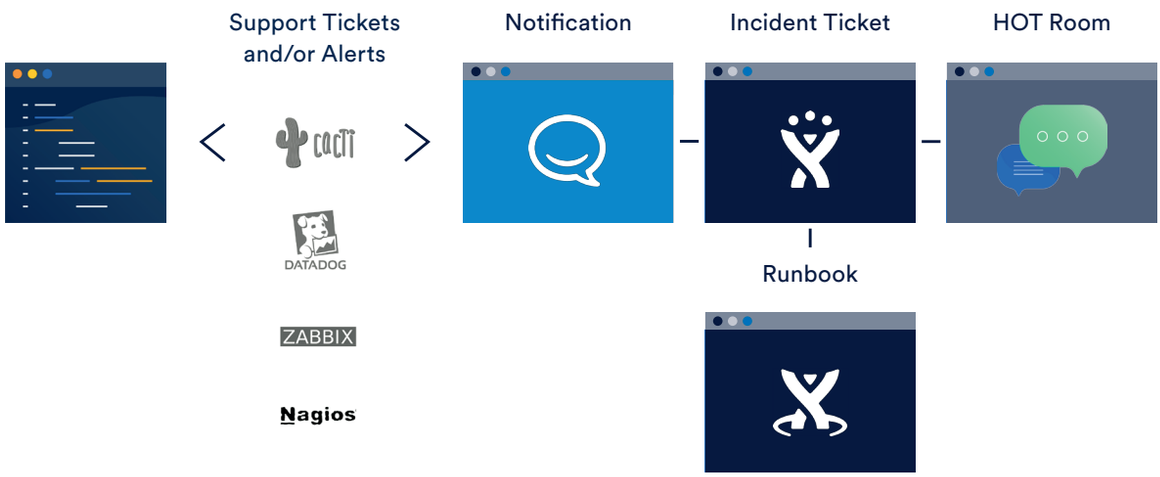
Having a well-integrated incident management tool chain is critical for decreasing context-switching and increasing efficiency during downtime. At Atlassian we rely on Statuspage, Hipchat, Jira Service Desk, and integrations with partners like PagerDuty, xMatters, and OpsGenie for a consistent workflow.

Speedier deployment with DevOps and a greater reliance on outside cloud vendors, creates an increase of incidents. In fact, Statuspage customers opened and resolved nearly 200,000 incidents in 2016 alone for a total of over 1 million hours of downtime!

Someone (or something) reports the incident

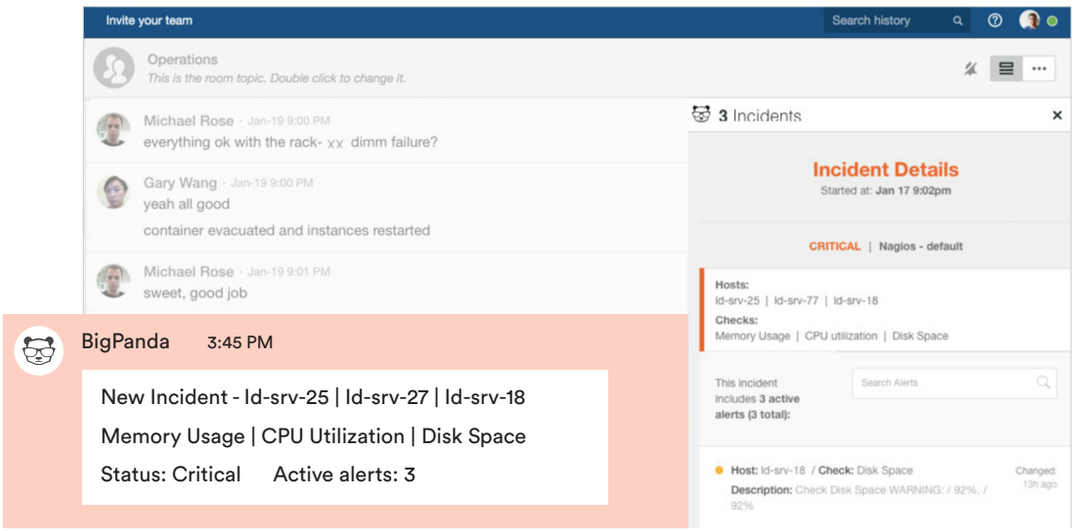
We learn about system outages and other potential performance glitches in two ways:

- Our users raise incidents using Jira Service Desk
- Our monitoring systems (like Cacti, DataDog, Zabbix, and Nagios) send us a notification



We aggregate the alerts into Hipchat

We aggregate all of our incident alerts into a single stream in a Hipchat room, so our teams get directly informed that there is a problem. This can sometimes generate noise, so we turn to tools like BigPanda to help out. BigPanda correlates massive amounts of IT alerts and events, and helps group them together, saving us a ton of time.



We create an incident ticket

Occasionally, a team may know the outage was caused by a change they just made, and they can quickly disable that change. But more often than not, we need to pull a team together to troubleshoot and resolve something. The first step is to raise an incident ticket in Jira Service Desk.

To create a ticket, we enter a few details, like a short name and description of the vent, and then categorize each incident by the impact it could have on a service, the number of users impacted, and how urgently it should be handled.

We notify our users

We use Statuspage to communicate with internal and external stakeholders, and push updates with incident status at regular intervals. Our users are able to subscribe to email and/or SMS notifications so they are alerted about the services they rely on most. Keeping users out of the dark lets us turn a tough situation into a memorable customer experience.

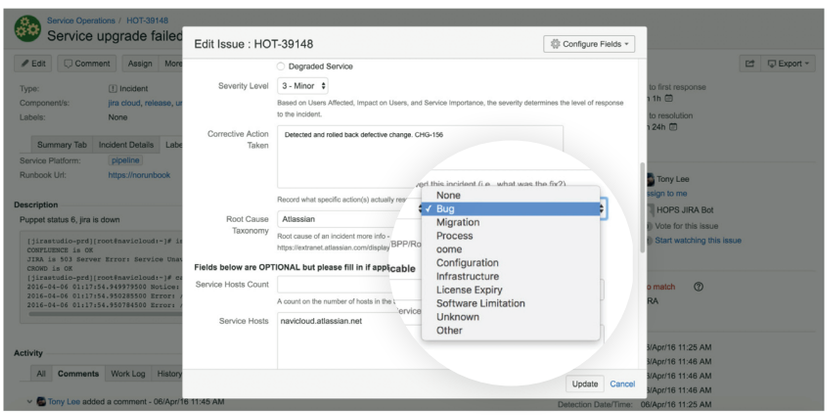
You can see what it looks like on status.atlassian.com

We create a dedicated chat room and swarm to resolve the incident

Within the incident ticket in Jira Service Desk, we use the “create a room” feature to move the conversation to a dedicated Hipchat room and pull in the right team to solve the problem at hand. The team discusses what went wrong, and agrees on an approach for troubleshooting and fixing it. We connect Statuspage with Hipchat to pipe in real-time status updates so the whole Ops team stays on the same page. Other teams – specifically customer-facing – can also pop in the room to get the most up-to-date information on resolution progress.

We resolve and categorize the root cause

ITIL recommends that we categorize each issue (bug, license expiry, infrastructure or configuration issue, etc.) once we’ve identified the root cause and taken corrective action. We also document the correction actions we took as well, and can use all of this information to run detailed reports highlighting our most common incident types and more. This helps us to take a more preventative approach to incident and problem management.



Finally, we conduct a post-mortem and document what went wrong

Possibly the most critical step to resolving an incident is learning from it. At Atlassian, we have a couple of different options for tracking the post-incident review activities: Jira or Confluence. Confluence lets us configure templates for a standard incident report layout, and it's easy to get started quickly. Jira, on the other hand, lets us build structured workflows that guide teams through the post-incident review process, and allow us to track each post-mortem review through to completion.

We've used both successfully. More important than the technology you use in the post-mortem process is making sure that you are able to develop a good understanding of the root cause of your outage. Use that to take the right set of actions to prevent the same outage from occurring again. To complete the feedback loop, we write post-mortems on our status page to give our customers an understanding of what happened and explain what we are doing to make sure it doesn't happen the same way again.

Our top recommendations

COMMUNICATE EARLY AND OFTEN DURING AN INCIDENT

Before an incident strikes make sure you are prepared with a solid communication plan that will keep users informed and build trust along the way. Make sure you know how to identify the incident commander, who owns the comms plan, what channel(s) you will use for different severity levels, what templates you can create for common issues so you can accelerate the time from detection to communication, and how you will follow-up with affected users after resolution.

CAPTURE THE DATA WHILE IT'S FRESH IN YOUR MIND

We use a Jira workflow we developed to walk our team members through the entire incident report process, complete with target timeframes for each step.

MAKE SURE YOU DOCUMENT EVERYTHING IN YOUR KNOWLEDGE BASE

We write all our incident reports in Confluence (and link to them from Jira), so we can refer back to them for future similar incidents and ensure we keep getting smarter (and sharing the knowledge) along the way.

AUDIT YOUR RESULTS REGULARLY

We run reports in Jira to make sure our team is doing a good job of resolving incidents and of documenting the results. By introducing better workflow and diagnosis tools and following a standardized approach to incident and problem management, we've reduced our mean-time-to-diagnosis from 113 minutes to just 23 minutes—and we're committed to cutting it even more.

PRO TIP

We’re big on you build it, you ship it, you run it.

06

Being Proactive and Staying Ahead of the Game

As the saying goes: The best defense is a good offense. While the core responsibility of an SRE team is to ensure reliability and availability, even with the best planning and processes in place, things can always go south. For this reason, our SRE teams at Atlassian believe in being proactive. It's vital that with each and every incident that occurs, we capture key takeaways that will improve our processes and motivate us to take risks and try things differently to drive positive change.

When the dust has settled from an incident, it's time to complete a thorough retrospective and ensure we've identified areas of improvement for next time. We plan, track and assess this work in Jira Software. It's particularly helpful in ensuring our teams across multiple geographies are aligned and always on the same page. Distributed teams are invaluable in providing around-the-clock coverage, but working across different timezones create collaboration challenges. For this reason our team shares one complete backlog of project work that is understood by all team members across regions. We also adopt agile best practices for our proactive work and forecast future work based on capacity and historical velocity.

Here are two agile rituals we follow as a team:

SPRINT REVIEW

Team members educate and showcase the value delivered during the two week sprint to the entire team. This helps members of our team learn from one another, try new and different things to achieve better results.

SPRINT PLANNING

Our team prepares for a sprint ahead of time by considering the priority of work in the backlog, items that are incomplete from the previous sprint, and new stories, tasks or bugs that have been created since last sprint. With a good understanding of the team's overall capacity, team members add stories into the next empty sprint. This helps us have a pre-populated sprint ready and "on deck" at any point in time.

Another key goal for our team has been to evaluate time spent on manual, time consuming processes and find ways to reduce them. Two recent wins that helped us become more productive are: improving the foundation of our monitoring systems and progressive automation with Jira workflows.

Monitoring Improvements

Our team was recently tasked with building our current monitoring platform. The key goals were to reduce median time to resolve (MTTR) and lower the severity of incidents. Early detection via monitoring allows us to detect potential threats before users do and react proactively. We identified four 'golden' signals that help us detect these threats early on.

THE FOUR GOLDEN SIGNALS

We drew inspiration from the four signals that most SRE teams are probably familiar with:

- Latency
- Errors
- Traffic
- Saturation

These golden signals are the bare essential. They are the key aspects that should be monitored as a team tasked with delivering a reliable, user-facing service. Knowing our own team preferences, we expanded the list to include the following:

- Availability
- Saturation
- Application/User
- Reliability
- Latency

We decided to use the above signals based on historical analysis of incidents and a strong understanding of the service level objectives we were hoping to meet for our service. Identifying the right signals was pivotal to our monitoring effort since our goal was to provide monitoring visualization to all dependent teams and give them the right data to make the best decision at any given time. For example, we discovered that the anomaly detection could be as simple as detecting a rapid rate of change as opposed to finding a minor deviation. Another important aspect that helped improve our Monitoring system was to have clear actions around Alerts.

Monitoring vs. Alerts

While our monitoring boards were expansive under the five signal areas, we chose not to receive alerts on everything. This was by choice to ensure that we:

- Wouldn't fall in to the 'alert trap' i.e. not get inundated with alerts at all times
- Had a clear call to action for the recipient of each alert, and
- Identified causes and symptoms of incidents, preferring to alert on a symptom

As a result of these actions, we were able to detect and take appropriate actions on 70% of the incidents ahead of time.

PROGRESSIVE AUTOMATION WITH Jira WORKFLOWS

Jira Software workflows support teams beyond dev teams, and the workflows are capable of supporting more than just the standard stages of:

For example, automation is often a gradual process for our team with several ad-hoc scripts designed to progressively simplify established runbooks. These scripts are then often tied together into an end-to-end service to remove manual intervention and automate the process. Jira workflows help simplify that process, reduce noise in the service request queues, and represent the various ticket stages in a runbook.

Here's one example of how we automated the restoration process of servers post RMA.

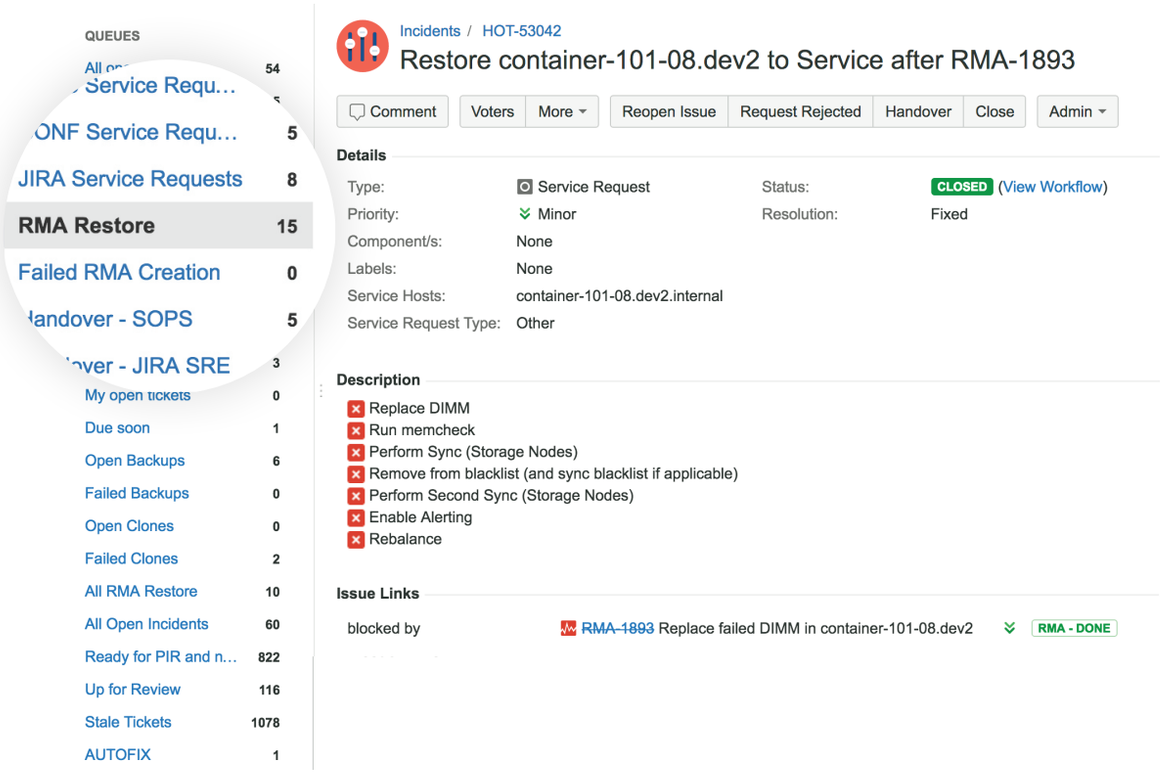
RESTORING SERVERS TO SERVICE AFTER AN RMA

A returned materials authorization (or RMA) is an alphanumeric identifier used by hardware manufacturers to indicate that a user has been authorized by the company to return or repair a defective or broken product.

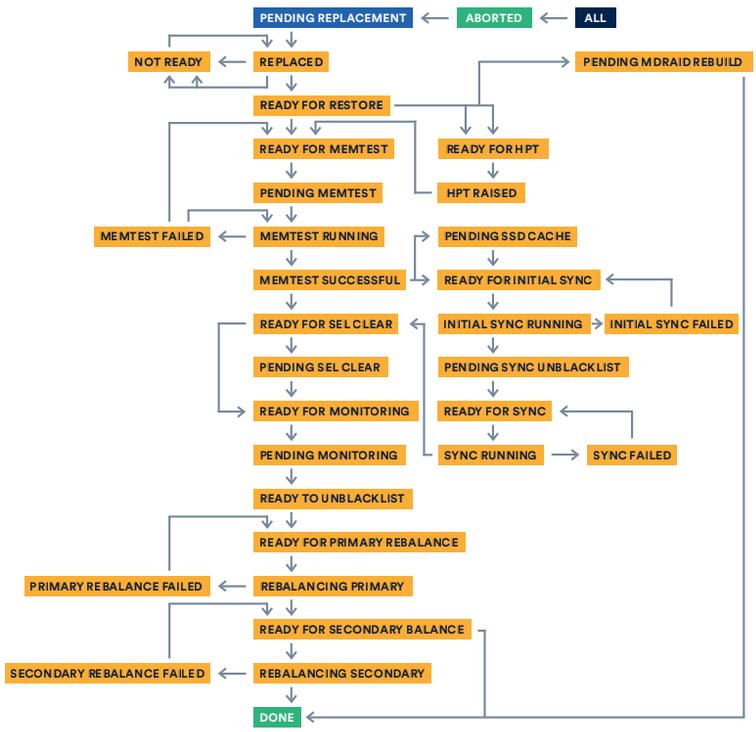
Post RMA i.e. post evacuation, power off, and repair or replacement, servers in our fleet are returned to the cluster. That process was historically complicated, involving several different teams and inconsistent tracking. It was difficult to track the stage of each ticket (of which there were often a dozen at any given point), which ultimately led to tickets being deprioritized and not tracked to completion.

The laborious, manual processing of a ticket looked like this:

1. Opening the ticket queue and selecting a ticket.
2. Checking status of server in ticket. Note: Checkboxes wouldn't reflect actual status of server in the process, meaning we had to either read the comments in the ticket or consult with other team members.
3. Locating the relevant runbook.
4. Locating the relevant step in said runbook. Note: If the step involved waiting for an async process (like a
5. memtest) to complete, we had to ensure that it was complete. If not, this meant starting over from Step 1.
6. Executing steps in runbook as required.



This process was not only tedious and cumbersome, but also unproductive since multiple SREs had to familiarize themselves with the status of each ticket in the queue.



So, it was time to try something different.

Enforcing a Structure

The first step of automating the above process was to establish a structure for automation. Jira workflows not only provide structure, but also let teams see the benefits of this structure even before the automation is complete. By simply creating a specific ticket type and a workflow that has a status for each step of the runbook, we could instantly do the following:

1. See where in the process a ticket was from the queue.
2. Filter queues based on status.
3. Establish a seamless process to transition a ticket to the next status after a step in the runbook was complete.

The next step was to automate this process.

Automating that Structure

With a runbook-like structure to follow, and state stored in the form of ticket status, it was easier to automate ticket management using the Jira API (instead of having to manage state in a database). This allows a stateless microservice to handle any steps that don't require human interaction (like waiting for long-running tasks to complete) and removing any items that don't currently require human interaction from human-visible queues.

The result was the following interconnected components:

1. A stateless microservice that polled the Jira API for tickets with a particular kind of status, and processed them based on that status.
2. Cleaner service request queues that only included things that couldn't be actioned by a human.
3. Tickets that naturally reflected their current status.

But we didn't just stop at automation. Remember staying ahead of the game? That's where our work on improving the un-automatable components comes in.

Improving the Un-automatable

Naturally there are likely to be some steps of any process that are not easy to automate, or are expected to take a significant amount of effort and/or time. You can often still do away with the runbooks in these situations by providing instructions to the persons who will process any manual steps. For example, our restore to service automation does not have access to the IPMI interfaces, so it was easier to have the automation provide instructions as a comment on the ticket, as seen in this diagram. The instructions are clear, easy to follow, and prevent the need to context switch.

HOPS JIRA Bot added a comment - 21/Mar/17 8:27 PM

Action Required

Please run the following commands on manager-0.sm1:

```
$ sudo /usr/sbin/uc_nagios.py container-107-05
$ sudo /usr/sbin/uc_nagios.py container-107-05 -i
```

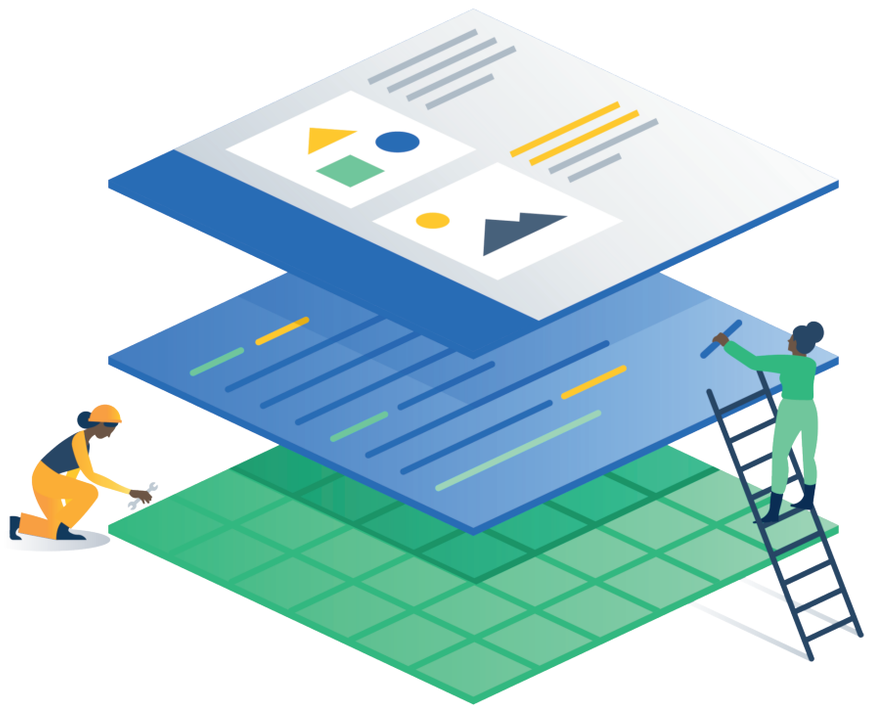
Check the output of the above commands and ensure that all checks are OK. Any checks in WARN or CRITICAL need to be resolved before proceeding.

Once all checks are ok, run the following commands on manager-0.sm1 to enable alerting:

```
$ sudo /usr/sbin/uc_nagios.py container-107-05 -e
$ sudo /usr/sbin/uc_nagios.py container-107-05 -e -i
```

These tickets appear in a Jira Service Desk queue called “Actionable RMA Restores” and we strictly follow this template for clear actions and expected behaviors. This is similar to what would be put in a runbook, but reduces the chance of a transcription error and the friction around performing the actions in this workflow.

Like all agile teams, we are in a constant state of evaluation. Our next goal is to begin to automate more of these manual steps, but for the time being we are focused on continuing to further improve the monitoring systems and progressive automation workflows that have made such a positive impact thus far.

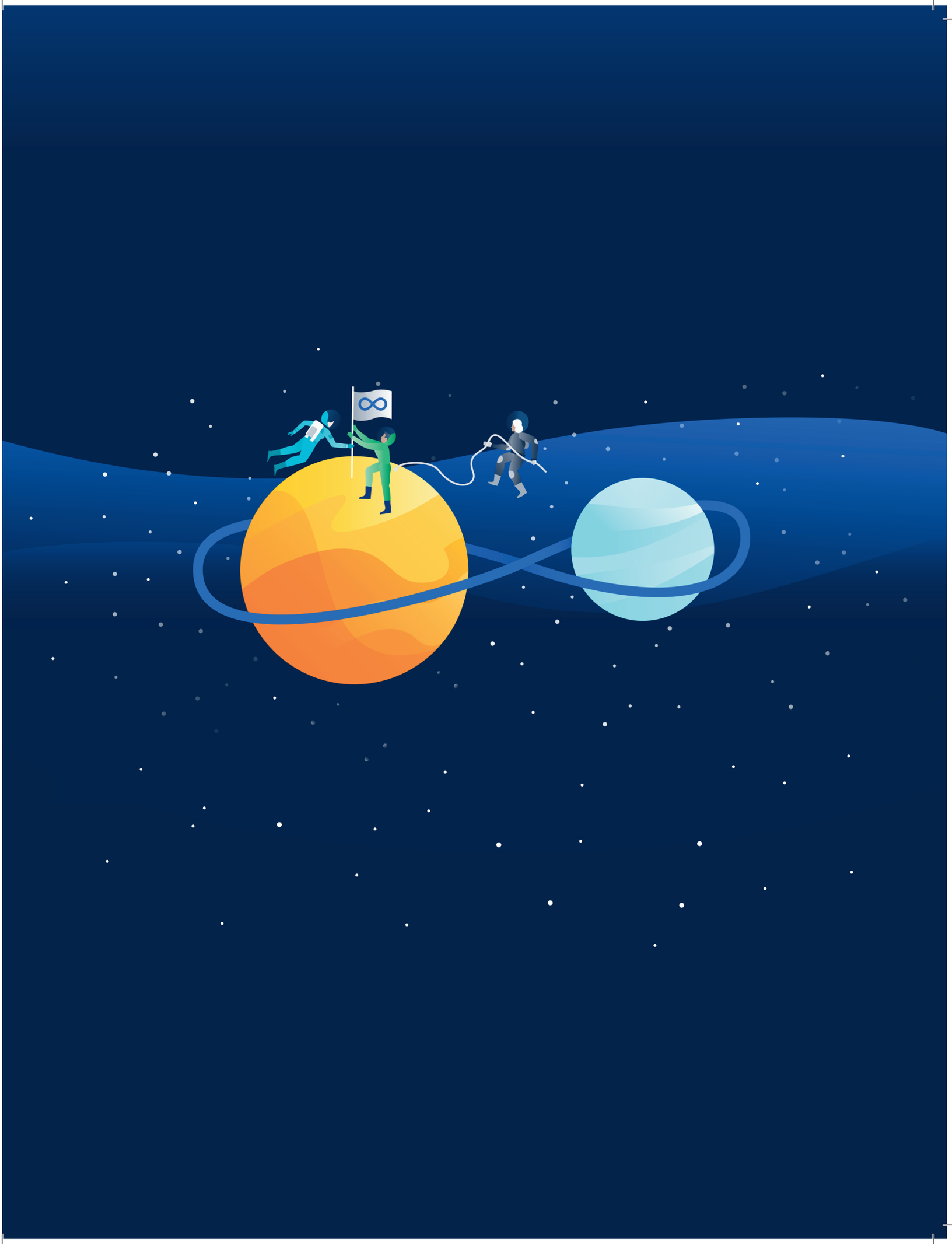


Are You Ready for DevOps?

DevOps is a culture, a philosophy, a methodology. Software Development and Operations teams that practice DevOps are more agile, more innovative, and more profitable. Through increased collaboration and greater visibility across teams, Development and Operations teams can work more productively and efficiently than ever before. Atlassian's mission is to unleash the potential in every team. With the Atlassian suite and our ecosystem of partner integrations, Development, Operations and all associated teams have the tools and processes to: foster a culture of collaboration and trust, release faster, accelerate time to resolution of critical issues, and better manage unplanned work.

Ride the DevOps wave with us.

atlassian.com/devops/start-your-journey



atlassian.com/devops