slack engineering (https://slack.engineering/)

Image by Freepik

BuildRock: A Build Platform at Slack

 **Joel Bartlett** Software Engineer I

 **Pamela Gluss** Sr. Software Engineer, Build

 **Pash Kaushik** Manager, Software Engineering

 **Tory Payne** Senior Manager, Software Engineering

 **Adam Compton** Staff Software Engineer, Internal Tools

 **Jerry Shen** Software Engineer, Desktop

 **Arlis Halcomb** Sr. Software Engineer, Developer Productivity

 **Sandeep Baldawa** Senior Staff Software Engineer, Backend

11 minutes ● Written 1 year ago

Our build platform is an essential piece of delivering code to production efficiently and safely at Slack. Over time it has undergone a lot of changes, and in 2021 the Build team started looking at the long-term vision.

Some questions the Build team wanted to answer were:

- When should we invest in modernizing our build platform?
- How do we deal with our build platform tech debt issues?
- Can we move faster and safer while building and deploying code?
- Can we invest in the same without impacting our existing production builds?
- What do we do with existing build methodologies?

In this article we will explore how the Build team at Slack is investing in developing a build platform to solve some existing issues and to handle scale for future.

Slack's build platform story

Jenkins (https://www.jenkins.io/) has been used at Slack as a build platform since its early days. With hypergrowth at Slack and and an increase in our

product service dependency on Jenkins, different teams started using Jenkins for build, each with their own needs—including requirements for plugins, credentials, security practices, backup strategies, managing jobs, upgrading packages/Jenkins, configuring Jenkins agents, deploying changes, and fixing infrastructure issues.

This strategy worked very well in the early days, as each team could independently define their needs and move quickly with their Jenkins clusters. However, as time went on, it became difficult to manage these Snowflake Jenkins clusters, as each had a different ecosystem to deal with. Each instance had a different set of infrastructure needs, plugins to upgrade, vulnerabilities to deal with, and processes around managing them.

While this wasn't ideal, was it really a compelling problem? Most folks deal with build infrastructure issues only once in a while, right?

Surprisingly, that isn't true—a poorly designed build system can cause a lot of headaches for users in their day-to-day work. Some pain points we observed were:

- Immutable infra was missing, which meant that consistent results were not always possible and troubleshooting was more difficult
- Manually added credentials made it difficult to recreate the Jenkins cluster in the future
- Resource management was not optimal (mostly due to static ec2 Jenkins agents)
- Lots of technical debt made it difficult to make infrastructure changes
- Business logic and deploy logic were combined in a single place

- Strategies were missing for backup and disaster recovery of the build systems
- Observability, logging, and tracing were not standard
- Deploying and upgrading Jenkins clusters was not only difficult but risk prone, coupled with the fact that the clusters weren't stateless, so recreation of the clusters was cumbersome, hindering regular updates and deployability
- Shift-left techniques were missing, which meant we found issues *after* the build service was deployed as opposed to finding issues earlier

From the business perspective this resulted in:

- Incidents and loss of developer productivity, mostly due to the difficulty of changing configurations like ssh-keys and upgrading software
- Reduced person-cycles available for operations (e.g. upgrades, adding new features, configuration)
- Non-optimal resource utilization, as un-utilized memory and CPU on current Jenkins servers is high
- Inability to run Jenkins around the clock, even when we do maintenance
- Data loss pertaining to CI build history when Jenkins has downtime
- Difficult-to-define SLA/SLOs with more control on the Jenkins services
- High-severity warnings on Jenkins servers

Okay we get it! How were these problems addressed?

With the above requirements in mind, we started exploring solutions. Something we had to be aware of was that we could not throw away the existing build system in its entirety because:

- It was functional, even if there was more to be done
- Some scripts used in the build infrastructure were in the critical path of Slack's deployment process, so it would be a bit difficult to replace them
- Build Infrastructure was tightly coupled with the Jenkins ecosystem
- Moving to an entirely different build system was an inefficient utilization of resources, compared to the approach of fixing key issues, modernizing the deployed clusters, and standardizing the Jenkins inventory at Slack

With this in mind, we built a quick prototype of our new build system using Jenkins.

At a high level, the Build team would provide a platform for "build as a service," with enough knobs for customization of Jenkins clusters.

Features of the prototype

We conducted research on what large-scale companies were using for their build systems. We also met with multiple companies to discuss build systems. This helped the team learn—and if possible

replicate—what some companies were doing. The learnings from these initiatives were documented and discussed with stakeholders and users.

Stateless immutable CI service

The CI service was made stateless by separation of the business logic from the underlying build infrastructure, leading to quicker and safer building and deploying of build infrastructure (with the option to involve shift-left strategies), along with improvement in maintainability. As an example, all build-related scripts were moved to a repo independent from where the business logic resided. We used Kubernetes (https://kubernetes.io/) to help build these Jenkins services, which helped solve issues of immutable infrastructure, efficient utilization of resources, and high availability. Also, we eliminated residual state; every time the service was built, it was built from scratch.

Static and ephemeral agents

Users could use two types of Jenkins build agents:

- Ephemeral agents (Kubernetes workers), where the agents run the build job and get terminated on job completion
- Static agents (AWS EC2 machines), where the agents run the build job, but remain available after the job completion too

The reason to opt for static AWS EC2 agents was to have an incremental step before moving to ephemeral workers, which would require more effort and testing.

Secops as part of the service deployment pipeline

Vulnerability scanning each time the Jenkins service is built was important to make sure secops was part of our build pipeline, and not an afterthought. We instituted IAM (https://aws.amazon.com/iam/?trk=ced3f61b-256e-4936-a949-66a0aee40a02&sc_channel=ps&sc_campaign=acquisition&sc_medium=ACQ-P%7CPS-GO%7CNon-Brand%7CDesktop%7CSU%7CSecurity%20ID%20Compliance%7CSolution%7CUS%7CEN%7CDSA&ef_id=CjwKCAjw_tWRBhAwEiwALxFPoSJcPh-sUeS9ETDMfeuonhtGyUw9QJuKBu3qjSeOuG0fui7JXkvsaRoCueQQAvD_BwE:G:s&s_kwcid=AL!4422!3!579296534079!!!g!!) and RBAC (https://kubernetes.io/docs/reference/access-authn-authz/rbac/) policies per-cluster. This was essential for securely managing clusters.

More shift-left to avoid finding issues later

We used a blanket test cluster and a pre-staging area for testing out small/large impact changes to the CI system even before we hit the rest of the staging envs. This would also allow high-risk changes to be baked in for an extended time period before pushing changes to production. Users had flexibility to add more stages before deployment to production if required.

Significant shift-left with a lot of tests incorporated, to help catch build infrastructure issues well before deployment. This would help with developer productivity and significantly improve the user experience. Tools were provided so that most issues could be debugged and fixed locally before deployment of the infrastructure code.

Standardization and abstraction

Standardization meant that a single fix could be applied uniformly to all Jenkins inventory. We did this through the use of a configuration management plugin for Jenkins called casc (https://plugins.jenkins.io/configuration-as-code/). This plugin allowed for ease in credentials, security matrix, and various other Jenkins configurations, by providing a single YAML configuration file for managing the entire Jenkins controller. There was close coordination (https://gitter.im/jenkinsci/configuration-as-code-plugin?at=5d6796b5ce6f764db2f92083) between the build team and the casc plugin open source project.

Central storage ensured all Jenkins instances used the same plugins to avoid snowflake Jenkins clusters. Also, plugins could be automatically upgraded, without needing manual intervention or worrying about version incompatibility issues.

Jenkins state management

We managed state through EFS (https://aws.amazon.com/efs/). Statement management was required for a few build items like build history and configuration changes. EFS (https://aws.amazon.com/efs/) was automated to back up on AWS at regular intervals, and had rollback functionality for disaster recovery scenarios. This was important for production systems.

GitOps style state management

Nothing was built or run on Jenkins controllers; we enforced this with GitOps (https://about.gitlab.com/topics/gitops/). In fact most processes could be easily enforced,

as manual changes were not allowed and all changes were picked from Git, making it the single source of truth. Configurations were managed through the use of templates to make it easy for users to create clusters, re-using existing configurations and sub-configurations to easily change configurations. Jinja2 (https://jinja.palletsprojects.com/en/3.0.x/) was used for the same.

All infrastructure operations came from Git, using a GitOps model. This meant that the entire build infrastructure could be recreated from scratch with the exact same result every time.

Configuration management

Associated metrics, logging, and tracing were enabled for debugging on each cluster. Prometheus (https://prometheus.io/) was used for metrics, along with our ELK stack for tracking logs and honeycomb (https://www.honeycomb.io/). Centralized credentials management was available, making it easy to re-use credentials when applicable. Upgrading Jenkins, the operating system, the packages, and the plugins was incredibly easy and could be done quickly, as everything was contained in a container Dockerfile (https://docs.docker.com/engine/reference/builder/).

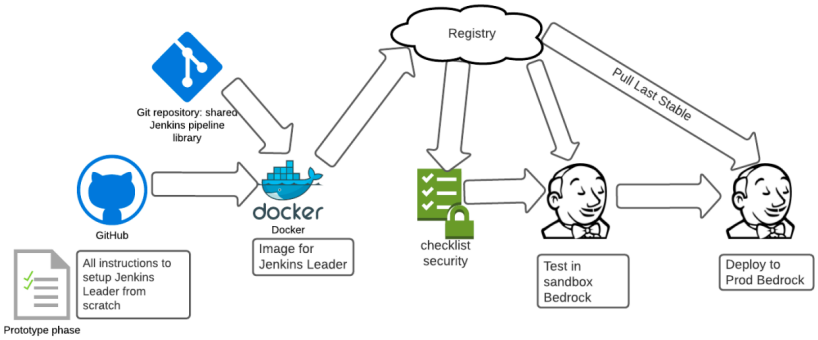
Service deployability

Individual service owners would have complete control over when to build and deploy their service. The action was configurable to allow service owners to build/deploy their service on commits pushed to GitHub if required.

For some use cases, moving to Kubernetes wasn't possible immediately. Thankfully, the prototype supported "containers in place," which was an incremental step towards Kubernetes.

Involving larger audience

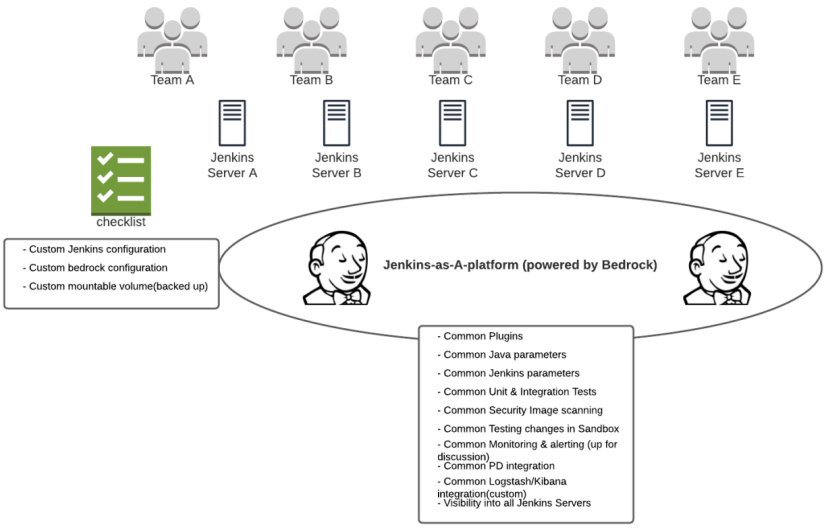
The proposal and design were discussed at a Slack-wide design review process where anyone across the company, as well as designated experienced developers, could provide feedback. This helped us get some great insights about customer use cases, design decision impacts on service teams, techniques on scaling the build platform and much more.



Sure, this is good, but wouldn't this mean a lot of work for build teams managing these systems?

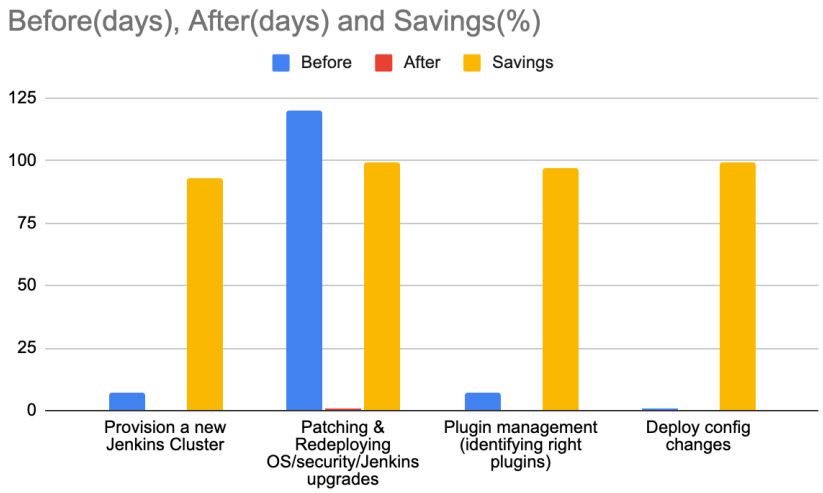
Well, not really. We started tinkering around with the idea of a distributed ownership model. The Build team would manage systems in the build platform infrastructure, but the remaining systems would be

managed by service owner teams using the build platform. The diagram below roughly gives an idea of the ownership model.



Cool! But what's the impact for the business?

The impact was multifold. One of the most important effects was reduced time to market. Individual services could be built and deployed not just quickly, but also in a safe and secure manner. Time to address security vulnerabilities went significantly down. Standardization of the Jenkins inventory reduced the multiple code paths required to maintain the fleet. Below are some metrics:



Infrastructure changes could be rolled out quickly — and also rolled back quickly if required.

Wasn't it a challenge to roll out new technology to existing infrastructure?

Of course, we had challenges and learnings along the way:

- The team had to be familiar with Kubernetes, and had to educate other teams as required.
- In order for other teams to own infrastructure, the documentation quality had to be top notch.
- Adding ephemeral Jenkins agents was challenging, as it involved reverse engineering existing EC2 Jenkins agents and reimplementing them, which was time consuming. To solve this we took an incremental approach, i.e. we first moved the Jenkins controllers to Kubernetes, and in the next step moved the Jenkins agents to Kubernetes.
- We had to have a rock solid debugging guide for users, as debugging in Kubernetes is very different from dealing with EC2 AWS instances.
- We had to actively engage with Jenkins's open source community to learn how other companies were solving some of these problems. We found live chats like this (https://gitter.im/jenkinsci) were very useful to get quick answers.
- We had to be incredibly careful about how we migrated production services. Some of these services were critical in keeping Slack up.
  - We stood up new build infrastructure and harmonized configurations so that teams could easily test their workflows confidently.
  - Once relevant stakeholders had tested their workflows, we repointed endpoints and switched

the old infrastructure for the new.
    - Finally, we kept the old infrastructure on standby behind non-traffic-serving endpoints in case we had to perform a swift rollback.
- We held regular training sessions to share our learnings with everyone involved.
- We realized we could reuse existing build scripts in the new world, which meant we didn't have to force users to learn something new without a real need.
- We worked closely with user requests, helping them triage issues and process migrations. This helped us create a good bond with the user community. Users also contributed back to the new framework by adding features they felt were impactful.
- Having a GitOps mindset was challenging initially, mostly because of our traditional habits.
- Metrics, logging, and alerting were key to managing clusters at scale.
- Automated tests were key to making sure the correct processes were followed, especially as more users got involved.

As a starting step we migrated a few of our existing production build clusters to the new method, which helped us learn and gather valuable feedback. All our new clusters were also built using the proposed new system in this blog, which significantly helped us improve delivery timelines for important features at Slack.

We are still working on migrating all our services to our new build system. We are also trying to add features, which will remove manual tasks related to maintenance and automation.

In the future we would like to provide build-as-a-service for MLOps (https://ml-ops.org/), Secops (https://saltproject.io/the-power-of-secops-redefining-core-security-capabilities/#:~:text=SecOps%20(Security%20%2B%20Operations)%20is,risk%20and%20improve%20business%20agility.), and other operations teams. This way users can focus on the business logic and not worry about the underlying infrastructure. This will also help the company's TTM (https://www.tcgen.com/time-to-market/#:~:text=Time%20to%20market%20(also%20called,the%20first%20unit%20is%20sold.).

If you would like to help us ***build*** the future of DevOps, we're hiring (http://slack.com/careers/)!

#automation #collaboration #developer-productivity #devtools #engineering