Thoughts on WMF CI architecture

Lars Wirzenius, Release Enginering

work in progress, first draft being written

Abstract

The release engineering team is working on replacing the WMF CI system. The existing CI system needs to be replaced, since it relies on a customized version of Zuul, which has long been obsolete. The current upstream version of Zuul is entirely different, so whatever happens, we need to make significant, user-visible changes, and are taking advantage of the opportunity to re-think the whole CI system from scratch to make it server our various stakeholders better. This isn't change for the sake of change, this is opportunistic redesign.

This document describes what we in the release engineering team are thinking for the new CI system, and why. This is part of the second phase of evaluation of new CI tooling. The first phase reduced the potential options from several dozen to three. The second phase looks as the options in more detail.

This document is in the middle of being written. Nothing is finalized yet. Anything might change.

Feedback on this document is most welcome. Direct feedback to its author or the RelEng team in general.

Contents

Chapter 1

Introduction

CI WG plans replacement of its current WMF CI system with one of Argo, GitLab CI, Zuul v3. These were selected in the first phase of the CI WG.

We aim to do "continuous deployment", not only "continous integration" or "continuous delivery". The goal is to deploy changes to production as often and as quickly as possible, without compromising on the safety and security of the production environment.

This document goes into more detail of how the new CI system should work, without (yet) discussing which replacement is chosen. A meta architecture if you wish.

It is assumed as of the writing of this document that future CI will build on and deploy to containers orchestrated by Kubernetes.

An important change is that we aim to change things so that as much as possible, all software deployments are to containers orchestrated by Kubernetes

Vision for CI

This is Lars's personal opinion, for now, but it's based on discussions with various people while at WMF. It's not expected to be new, radical, or controversial, compared to status quo.

In the future, CI at WMF serves WMF, its developers, and the Wikipedia movement by making software development more productive, more confident, and faster. The cycle time of changes (the time from idea to running in production) is short: for a trivial change, as little as five minutes. At the same time, the safety and security of production is protected: malicious changes do not get deployed, mistakes are rare, and can easily be fixed or the problematic change reverted.

Production here means all the software needed to run all the sites (Wikipedias in different languages, Commons, etc), as well as supporting services, including tooling and services that supports development.

Overall solution approach

The overall approach to the architecture of the CI system, and the workflow supported by it, is to keep all changes in version control (git), which includes code, configuration, and scripts for building and deploying. When a change to version control is pushed, CI builds and tests the change, humans review the change, and if all seems to be in order, CI deploys to production.

Stakeholders

Stakeholders in the WMF CI system include:

- RelEng, who are responsible for keeping CI running
- SRE, who are responsible for the infrastrcture on which CI runs
- MediaWiki developers, who develop MW and its extensions, and will (eventually) be doing MW releases for external MW deployers
- staff and volunteer developers of anything else built, tested, and deployed by CI
- WMF, who pays for CI using donations
- the Wikipedia movement, who use sites and services operated by WMF
- all of humanity, who benefit from having knowledge freely disseminated

Chapter 2

Requirements

This chapter lists the requirements we have for the CI system and which we design the system to fulfil.

Each requirement is given a semi-mnemonic unique identifier, so it can be referred to easily.

The goal is to make requirements be as clear and atomic as possible, so that the implementation can be more easily evaluated against the requirement: it's better to split a big, complicated requirement into smaller ones so they can be considered separately. Requirements can be hierarchical: The original requirement can be a parent to all its parts.

FIXME: We may want to have a way to track which requirements are being fulfilled, or tested by automated acceptance tests. Need to add something for this, maybe a spreadsheet.

These requirements were originally written up in the WG wiki pages and have been changed a little compared to that (as of the 21 March 2019 version).

Very hard requirements

These are non-negotiable requirement that must all be fulfilled by our future CI system.

- **SELFHOSTABLE** Must be hostable by the Foundation. It's not acceptable for WMF to rely on an outside service for this.
  - **FREESOFTWARE** Must be free software / open source. "Open core" like GitLab is be good enough, as long as we only need the parts that provide software freedom.

    This is partly due to the **SELFHOSTABLE** requirement, but also because a WMF value is to prefer open source.

- **GITSUPPORT** Must support git. We're not switching version control systems for CI.
- **UNDERSTANDABLE** Must be understandable without too much effort to our developers so that they can use CI/CD productively.
- **SELFSERVE** Must support self-serve CI, meaning we don't block people if they want CI for a new repo. Due to **PROTECTPRODUCTION**, there will probably to be some human approval requirement for new projects, but as much as possible, people should be allowed to do their work without having to ask permission.
  - **SELFSERVE2** Should allow the developers to define or declare at least parts of the pipeline jobs in the repository: what commands to run for building, testing, etc.

Hard requirements

These are not absolute requirements, and can be negotiated, but only to a minor degree.

- **ROLLBACK**: If a change is deployed to production and turns out to cause problems, it is easy and quick to revert the change.
- **FAST** Must be fast enough that it isn't perceived as a bottleneck by developers. We will need a metric for this.
  - **SHORTCYCLETIME** Must enable us to have a short cycle time (from idea to running in production). CI is not the only thing that affects this, but it is an important factor. We probably need a metric for this.
- **TRANSPARENT** Must make its status and what-is-going-on visible so that its operation can be monitored and so that our developers can check the status of their builds themselves. Also the overall status of CI, for example, so they can see if their build is blocked by waiting on others.
- **FEEDBACK** Must provide feedback to the developers as early as possible for the various stages of a build, especially the early stages ("can get source from git", "can build", "can run unit tests", etc.).

  The goal is to give feedback as soon as possible, especially in the case of the build failing.
  - **FEEDBACK2** Must support providing feedback via Gerrit, IRC, and Phabricator, at the very least. These are our current main feedback channels.
- **SECURE** Must be secure enough that we can open it to community developers to use without too much supervision.

- **MAINTAINED** Must be maintained and supported upstream. The CI system should not require substantial development from the Foundation. Some customization is expected to be necessary.
- **MANYREPOS** Must be able to handle the number of repositories, projects, builds, and deployments that we have, and will have in the foreseeable future.
- **METRICS** Must enable us to instrument it to get metrics for CI use and effectiveness as we need. Things like cycle times, build times, build failures, etc.
- **GERRIT** Must work with Gerrit as well as other self-hostable code-review systems (e.g., GitLab), if we decide to move to that later. This means, code review happens on Gerrit, after building and automated tests pass, and positive code review triggers deployment to production.
- **NOREBUILDING** Must promote (copy) Docker images and other build artifacts from "testing" to "staging" to "production", rather than rebuilding them, since rebuilding takes time and can fail. Once a binary, Docker image, or other build artifact has been built, exactly that artifact should be tested, and eventually deployed to production.
- **LOCALTESTS** Must allow developer to replicate locally the tests that CI runs. This is necessary to allow lower friction in development, as well as to aid debugging. For example, if CI builds and tests using Docker container, a developer should be able to download the same image and run the tests locally.
- **AUTOMATEDEPLOYMENT** Must allow deployment to be fully automated.
  - **AUTOMATEDSELFDEPLOYMENT** Must be automatically deployable by us or SRE, onto a fresh server.
- **HSCALABLE** Must be horizontally scalable: we need to be able to add more hardware easily to get more capacity. This is particularly important for build workers, which are the mostly likely bottleneck. Also, probably environments used for testing.
- **PROGLANGS** Must be able to support all programming languages we currently support or are likely to support in the future. These include, at least, shell, Python, Ruby, Java, PHP, and Go. Some languages may be needed in several versions.
- **OUTPUTLINKS** Must support HTTP linking to build results for easier reference and discussion. This way a build log, or a build artifact, can be reference using a simple HTTP (or HTTPS) link.
- **ARTIFACTARCHIVE** Should allow archiving build logs, executables, Docker images, and other build artifacts for a long period.

  - **RETENTION** The retention period should be configurable based on artifact type, and whether the build ended up being deployed to production.

- **CONFIGVC** Must keep configuration in version control. This is needed so that we can track changes over time.
- **GATING** Must support gating / pre-merge testing. FIXME: This needs to be explained.
- **PERIODICBUILDS** Must support periodic / scheduled testing. This is needed so that we can test that changes to the environment haven't broken anything. An example would be changes to Debian, upon which we base our container images.
- **POSTMERGETESTS** Must support post-merge testing. FIXME: This needs to be explained.
- **CIMERGES** Must support tooling to do the merging, instead of developers. We don't want developer merging by hand and pushing the merges. CI should test changes and merge only if tests pass, so that the branches for main lines of development are always releaseble.
- **TESTVC** Must support storing tests in version control. This is probably best achieved by having tests be stored in the same git repository where the code is.
- **BUILDDEPS** Must have some way to declare dependent repositories / software needed for testing. FIXME: This needs to be explained.
- **TESTSERVICES** Must support services for tests — i.e., some PHPUnit tests require MySQL. These are most important for integration tests. Proper unit tests do not depend on any external stuff. However, integration tests may well need MediaWiki, some specific extensions, and backing services, such as databases, "oid" services, and possibly more. CI needs to be able to provide such environments for testing.
- **OTHERGITORTICKETING** Must allow changing git repository, code review, and ticketing systems from Gerrit and Phabricator. We are not currently looking at switching away from Gerrit and Phabricator, but the future CI solution should not lock us into specific code review or ticketing solutions.
- **PROTECTPRODUCTION** Must protect production by detecting problems before they're deployed, and must in general support a sensible CI/CD pipeline. This is necessary both for the safety and security of our production systems, a higher speed of development, and higher productivity. The protection brings developer confidence, which tends to bring speed and productivity.
  - **ENFORCETESTS** Must allow Release Engineering team to enforce

tests on top of what a self-serving developer specifies, to allow us to set minimal technical standards.

- **CACHEDEPS** Must support dependency caching – we have castor, maybe we could do better? Maybe some CI systems have this figured out? This means, for example, caching npm and PyPI packages so that every build doesn't need to download them directly from the centralised package repositories. This is needed for speed.
- **GOODOPS**: The CI system needs to maintained in a proper way, with monitoring, multiple people who can get notified of problems and can fix things.
  - **MONITORING**: The CI system needs to have monitoring to automatically alert of problems.
  - **ONCALL**: There must be multiple, named people who can be alerted to deal with problems in CI that users can't deal with themselves. These people need to cover enough time zones to be available at almost all times, and have enough overlap to cover for each other when they're not availble due to illness, vacation, travel, or such reasons.
  - **SLA**: RelEng provides a promise, similar to a service-level agreement, for how well CI serves WMF and the movement.

Softer requirements

These requirements are even more easily negotiated.

- **HA** Should be highly available - can restart any component without disrupting service.
- **LIVELOG** Should have live console output of build.
- **MAXBUILDTIME** Should have build timeouts so that a build may fail if it takes too long. Among other reasons, this is useful to automatically work around builds that get "stuck" indefinitely.
- **CLEANWORKSPACE** Should provide a clean workspace for each test run - either a clean VM or container.
- **RATELIMIT** Should have rate limiting - one user/project can not take over most/all resources.
- **CHECKSIG** Should support validation and creation of GPG/PGP-signed git commits
- **SECRETS** Should support secure storage of credentials / secrets.

Would be nice

These are so soft they aren't even requirements, and more wish list items.

- **LIMITBOILERPLATE** Would be nice for test abstractions to limit boilerplate, i.e., all of our services are tested roughly the same way without having to copy instructions to every repository.
- **PRIORITIZEJOBS** Would be nice to prioritize jobs.
  - Use case: if there is a queue of jobs, there should be some mechanism of jumping that queue for jobs that have a higher priority.
  - We currently have a Gating queue that is a higher priority than periodic jobs that calculate Code Coverage.
- **ISOLATION** Would be nice to support isolation / sandboxing.
  - Jobs should be isolated from one another.
  - Jobs should be able to install apt-packages without affecting dependencies of other jobs.
- **CONTROLAFFINITY** Would be nice to have configurable job requirements/affinity.
  - Be able to schedule a job only on nodes that have at least X available disk space/ram/cpu/whatever OR try to schedule on nodes where a current build of this job isn't already running.
- **POSTMERGEBISECT** Would be nice to post-merge git-bisect to find patch that caused a particular problem with a Selenium test.
- **DEPLOYWHEREVER** Would be nice to have a mechanism for deployment to staging, production, pypi, packagist, toollabs. We could do with a way to deploy to any of several possible environments, for various use cases, such as bug repoduction, manual exploratory testing, capacity testing, and production. FIXME: what do pypi and packagist do in the list?
- **MATRIXBUILDS** Would be nice to have efficient matrix builds.
  - E.g., we currently run phpunit tests and browser tests for the Cartesian product of {PHP7 PHP7.1 PHP7.2 HHVM} x {MySQL, SQLite, PostgreSQL} x {Composer, MediaWiki vendor}, but we perform setup/git clone for all of those tests. Doing that in a space and time efficient way would be good.
- **MOBILE** Would be nice to support building and testing mobile applications (at minimum for iOS and Android).
- **EMBARGO** Would be nice to be able to run for secret/security patches. This means CI should be able to build and deploy changes that can't be made public yet, for security embargo reasons.

Chapter 3

Important use cases

These are some of the important use cases for the CI system, and how we plan CI to implement them.

Chapter 4

Architecture

The WMF development ecosystem

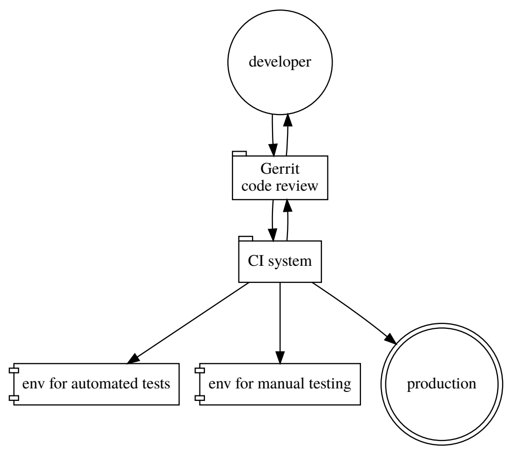

Figure 4.1: The WMF development ecosystem, roughly

The figure above is simplistic, but gives the general idea of what happens when a developer is finished with a change:

1. developer pushes a change to Gerrit, which trigger CI
2. CI builds and tests change (commit stage)
3. CI deploys to a test environment, runs tests against that (acceptance test stage); if everythins is OK, Gerrit is notified and requests code reviews from relevant parties
4. testers can request CI it deploy the change to an environment dedicated for manual testing
5. after a successful code review, CI merges changes to the master branch, runs all automated tests again, and deploys to the production environment

The commit and acceptance stages are triggered as soon as developer pushes changes to be reviewed. Human reviews won't be requested until the two stages pass, as there's no point in spending human attention on things that are not going to be candidates for deployment to production. The two stages may be re-run after code review, to make sure nothing unforeseen has changed while the review took place.

Other stages may run in parallel with code review, and if they fail they may nullify the release candidacy of the change. For example, stages for manual and capacity testing, and security test/review; depending on the change and the component in question, some or all of these may be necessary.

Normal change to an individual component

- a developer pushes a change to one program that runs in production
- the change is indepent of other changes and no other component depends on the chage
- e.g., bug fix, not a feature change
- the governing principle is that with commit stage and acceptance stage passing, plus a positive code review, the changes can be deploed to to production in most cases
- developer pushes change, this trigger commit and acceptance stages, which pass, which triggers code review requests to be sent to reviewers
- reviewers vote +2, which triggers a deployment to production
- this is the simplest possible use case for CI

Interdependent changes

- changes to two or more components that must all be applied at once or not at all, e.g., to mediawiki core and an extension
- in this scenario the change to MediaWiki core and the change to an extension may depend on each other, so that if either is deployed without the other, the system as a whole breaks; thus, either both changes get deployed, or neither
- Lars's opinion: this seems like a bad way of managing development. It seems better to be careful with such changes so that they can be disabled behind a feature flag in the configuration, or by autodetection of the other component, so that if only one component has been changed, it can stillbe deployed. Only when both components have been changed in production is the feature flag enabled, and the new feature works.

Security embargoed change

- change can't be public until it's deployed or manually made public

- this is typically part of "responsible disclosure"
- the change will be made public, but CI should be able to use it even before it's public, so that when it's time, there's no need to wait for CI to build/test the change and it can just be merged and deployed
- this means some builds and builds artifacts need to be locked away from public

Log storage

- We want to capture the build log or "console output" (stdout, stderr) of the build and store it. This is an invaluable tool for developers to understand what happens in a build, and especially why it failed.
- Ideally, the build log is formatted in a way that's easy for humans to read.
- It'd also be nice if the build log can be easily processed programmatically, to extract information from it automatically.
- We may want to store build logs for extended periods of time so that we can analyze them later. By storing them in a de-duplicating and compressing manner, the way backup software like Borg does, the storage requirements can be kept reasonable.

Artifact storage

- Artifacts are all the files created during the build process that may be needed for automated testing or deployment to production or any other environment: executable binaries, minimized Javascript, automatically generated documentation from source code (javadoc).
- We basically need to store arbitrary blobs for some time. We need to retrieve the blobs for deployment, and possibly other reasons.
- We may want to store artifacts that get deployed to production for a longer time than other artifacts so that we can keep a history what was in production at any recent-ish point in time.
- We will want to trace back from each artifact which git repository and commit it came from.
- We can de-duplicate artifacts (a la backup programs) to save on space. Even so, we will want to automatically expire artifacts on some flexible schedule to keep storage needs in control.
- We need to decide when we can make these artifacts publically accessible.
- Artifact storage must be secure, as everything that gets deployed to production goes via it.

- There are some artifact storage systems we can use.

Credentials management and access control

- FIXME: This is unclear as yet, the text below is some incoherent preliminary rambling by Lars which needs review and fixing.

Credentials and other secrets are needed to allow access to servers, services, and files. They are highly security sensitive data. The CI system needs to protect them, but allow controlled use of them.

Example: a CI job needs to deploy a Docker image with a tested and reviewed change as a container orchestrated by production Kubernetes. For this, it needs to authenticate itself to the Kubernetes API. This is typically done by a username/password combination, but might be an API token of some kind (though it doesn't really matter; it's all just secret bits at some level). How will the future CI system handle this?

Example: for tests, and in production, a MediaWiki container needs access to a MariaDB database, and MW needs to authenticate itself to the database. MW gets the necessary credentials for this from its configuration, which CI will install during deployment. The configuration will be specific for what the container is being used: if it's for testing a change, the configuration only allows access to a test database, but for production it provides access to the production database.

Builds are done in isolated containers. These containers have no credentials. Build artifacts are extracted from the containers and stored in an artifact storage system by the CI system, and this extraction is done in a controlled environment, where only vetted code is run, not code from the repository being tested. The build environment can't push artifacts directly to the artifact store.

Deployments happen in controlled environments, with access to the credentials needed for deployment. The deployment retrieves artifacts from the artifact storage system. The deployments are to containers, and the deployed containers don't have any credentials, unless CI has been configured to install them, in which case CI installs the credentials for the intended use of the container.

Note that credentials should not come directly from the source code of the deployed program. CI deploys configuration when it deploys the software. This way, the same software (build artifacts) can be deployed to different environment. (This may be complicated by the way MediaWiki is configured, using a PHP file in the source tree. This will need discussion.)

Tests run against software deployed to containers, and those containers only have access to the backing services needed for the test, and may even be firewalled to not have access to any other network locations.

Suggestion: Deployments will be done dedicated deployment environments, which run a "pingee" service. When a pipeline executes a deployment stage, deploying to any environment, the stage runs in a suitable container, but doesn't actually do the deployment itself. Instead, it "pings" a deployment service, with information of who is deploying, what, and where, and the deployment service inspects the change, and if it looks acceptable, does the actual deployment to the desired environment. The deployment service has access to the credentials it needs for accessing the artifacts and doing the deployment. There may be several deployment services, for deploying to environments with different security needs.

Chapter 5

The CI pipeline

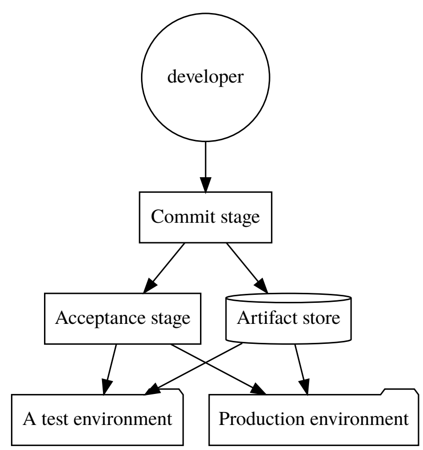

Figure 5.1: The default pipeline

CI will provide a default pipeline for all projects. Projects may use that or specify another one.

The pipeline will be divided into several stages. Mandatory stages for all changes and all projects are commit and acceptance stage. Other stages may be added to specific changes projects as needed.

The goal is that if the commit and acceptance stages pass, the change is a candidate that can be deployed to production, unless the project is such that it needs (say) manual testing or other human decision for the production deployment decision. Likewise, if the component or the change is particularly security or performance sensitive, stages that check those aspects may be required. CI will have ways of indicating the required changes per component, and also per change. (It is unclear how this will be managed.)

If the commit or acceptance stage fails, there is not production candidate. The pipeline as a whole fails. Any artifacts built by the pipeline will not be deployable to production, but they may be deployable to test environments, or downloaded by developers for inspection.

The commit stage

The commit stage builds any deployable artifacts, such as executable binaries, minimized Javascript, translation files, or Docker images. It is important that artifacts don't get rebuilt by later stages, because rebuilding does not always result in bitwise identical output. Instead the goal is to build once, test the artifacts, and deploy the tested artifacts, instead of rebuilding and maybe deploying something different than what was tested.

The commit stage also runs unit tests, and any other tests that can be run in isolation from other parts of the system, and that also are quick. The commit stage does not have access to backing services, such as databases or other components of the overall system. For example, when the pipeline processes a change to a MediaWiki extenasion, the commit stage doesn't have access to MediaWiki core or the MariaDB MediaWiki uses. Integration or system tests should be done in the acceptance test stage.

The commit stage also runs code health checks.

The commit stage is expected to be fast, aiming at less than five minutes, so that we can expect developers to wait for it to pass successfully. This will be a new requirement on our developers.

The commands to build (compile) or run automated tests are stored in the repository, either explicity, or by indicating the type of build needed. There might be a `.pipeline/config.yaml` file in the repository, which specifies that `make` is the command that builds the artifacts. Otherwise, the file may specify that it's a Go project, and CI would know how to build a Go project. In this case we can change the commands to build a Go project by changing CI only, without having to change each git repository with a Go program.

Only the declarative style will be possible for building Docker images, as we want control over how that is done (**SECURE** requirement).

CI may enforce specific additional commands to run, to build or test further things; this can be used by RelEng to enforce certain things. For example, we may enforce code health checks, or to enable (or disable) debug symbols in all builds. Such enforcement will be done in collaboration with our developers.

Any build dependencies needed during the commit stage must be specified explicitly. For example, the minimum required version of Go that should be installed in the build environment would be a build dependency. If a project build-depends on another project, it needs to specify which project, and which artifacts it needs installed from the other project. Explicit build-dependencies is more work, but results in fewer problems due to broken heuristics.

The acceptance stage

During the acceptance stage CI deploys artifacts built in the commit stage to a production-like system that has the same versions of all sofware as production, except for the changes being processed by the pipeline. CI will then run automated acceptance tests, and other integration and system tests, against the deployed software. The test environment is clean and empty, and well-known, unless and until the test suite inserts data or makes changes.

The acceptance stage can take time. Developers are not expected to wait until it is finished before they move on to working on something else.

Manual tests

Testers may instruct CI to deploy any recent built set of artifacts to a dedicated test environment, and can use the software in that environment where it is isolated from others, and won't suddenly change underneath them. The details of how this will be implemented are to be determined later.

This feature of the CI can also be used to demonstrate upcoming features that are not yet ready to be deployed to or enabled in production.

Capacity tests, non-functional requirements

Capacity tests, and other tests for non-functional requirements, will also be done in dedicated, isolated production-like environments. RelEng will work with the performance team to sort out the details.

Chapter 6

CI implementation

FIXME This needs to be written, but it needs a lot of thinking first

Chapter 7

Automated acceptance tests for CI

- FIXME: This chapter will sketch some automated acceptance tests using a Gherkin/Cucumber-like pseudo code language. Or in some other way that can be automatically executed.

  The goal is to have CI deploy itself, and as part of the pipeline run acceptance tests defined here.