May 2020

A GENERIC CACHE-BASED DEPLOYMENT PIPELINE OPTIMIZATION TECHNIQUE

HP INC

A Generic Cache-Based Deployment Pipeline Optimization Technique

A deployment pipeline is basically a workflow with steps that are triggered by changes in source code. The idea disclosed here optimizes deployment pipelines in general by showing how they can cache static and dynamic artifacts that can be reused by different pipeline runs.

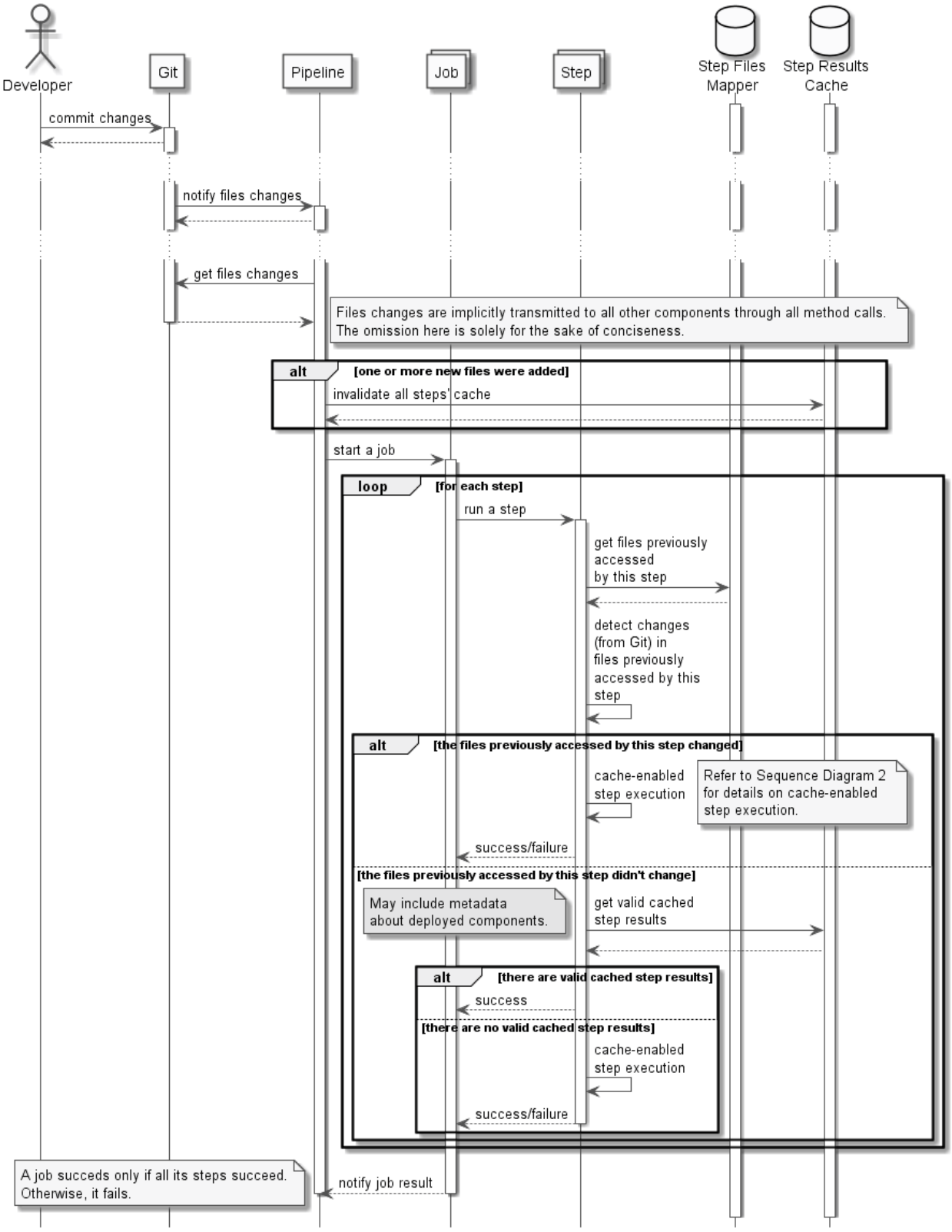
Conventional approaches:

- Cache solely with static artifacts (e.g., processed source files, pre-built images, downloaded dependencies);
- Have a predefined knowledge on the relevance of files for each pipeline step to perform caching decisions;
- Work within the boundaries of specific technologies;
- Don't take cache timeouts too seriously.

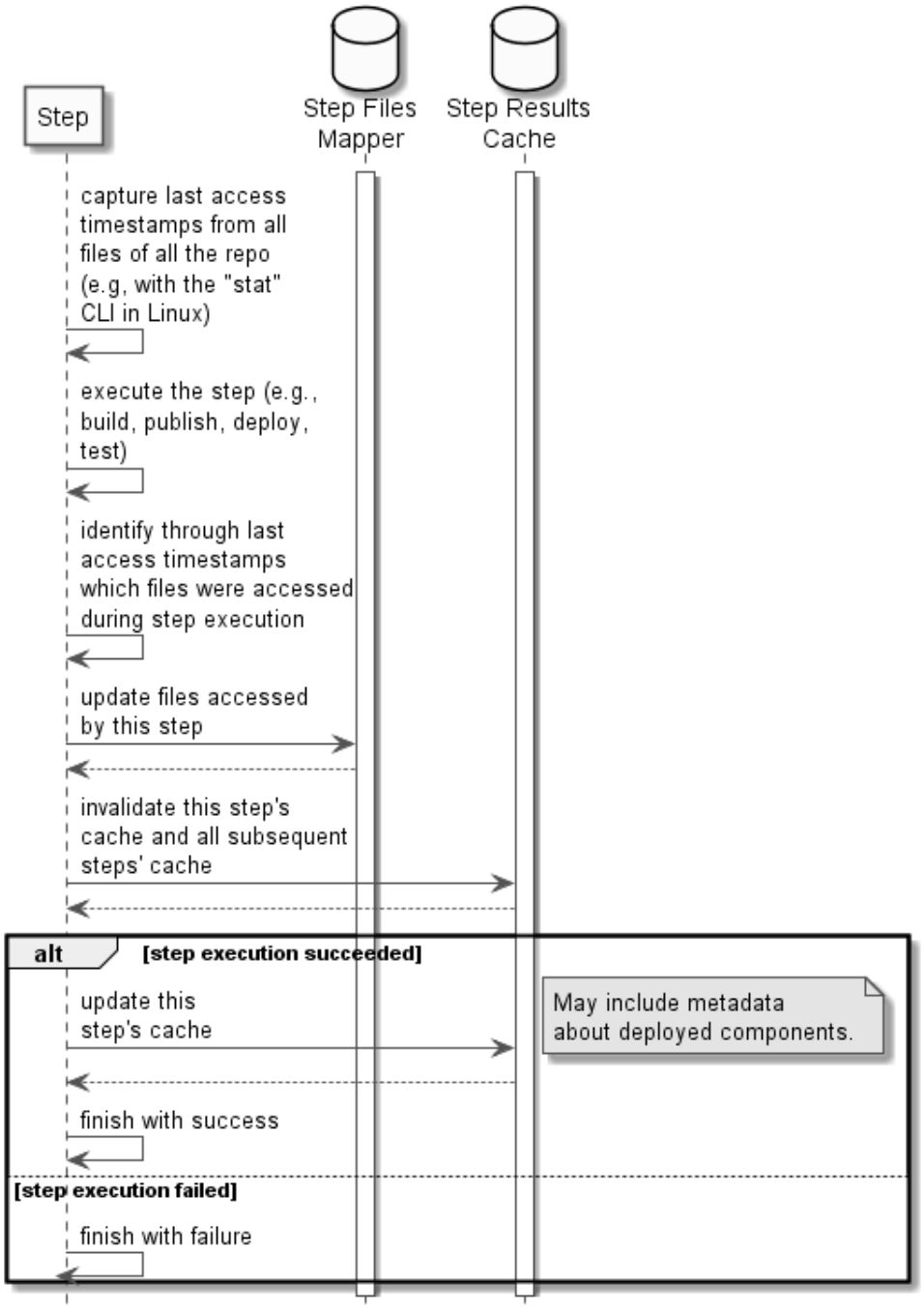
The new approach, detailed in *Sequence Diagrams 1 and 2*:

- Caches dynamic artifacts. For example, it allows a pipeline triggered by a change only in functional test source code to start a new run directly at the functional test step;
- Revalidates its current caching knowledge in each and every pipeline run. Files are dynamically associated or disassociated to steps as the source code evolves;
- Works with any generic workflow that is automatically triggered/executed in response to changes in a source code management tool;
- Don’t rely on valid cached data that is available for too long. At some point in time, the cost of keeping deployments available to be re-used by future pipeline runs would defeat any monetary economy brought by caching pipeline artifacts.

.


Developer
Git
Pipeline
Job
Step
Step Files Mapper
Step Results Cache
commit changes
notify files changes
get files changes
Files changes are implicitly transmitted to all other components through all method calls.
The omission here is solely for the sake of conciseness.
alt
[one or more new files were added]
invalidate all steps' cache
start a job
loop
[for each step]
run a step
get files previously accessed by this step
detect changes (from Git) in files previously accessed by this step
alt
[the files previously accessed by this step changed]
cache-enabled step execution
Refer to Sequence Diagram 2 for details on cache-enabled step execution.
success/failure
[the files previously accessed by this step didn't change]
May include metadata about deployed components.
get valid cached step results
alt
[there are valid cached step results]
success
[there are no valid cached step results]
cache-enabled step execution
success/failure
A job succeds only if all its steps succeed.
Otherwise, it fails.
notify job result

Sequence diagram 1: Cache-enabled pipeline job execution. `Step Files Mapper` is a key-value storage for keeping track of what files are accessed by each individual step. `Step Results Cache` is a key-value storage for keeping re-usable step results for future use.



Sequence diagram 2: Cache-enabled pipeline step execution. `Step Files Mapper` is a key-value storage for keeping track of what files are accessed by each individual step. `Step Results Cache` is a key-value storage for keeping re-usable step results for future use.

What is our proposal?

Fundamentally, we map files to steps, and, through a cache for static (e.g., processed source files, pre-built images, downloaded dependencies) and dynamic(e.g., metadata about deployed applications) artifacts, run only the steps that are impacted by the file changes that triggered a job, if there are valid cached elements that can be reused.

In the example illustrated by *Figure 1*, if the source code changes that triggered `Job 2` impact only `Deploy`, there is no need to necessarily re-run the `Build` and `Publish` steps again, if their results are cached. In other words, in *Figure 1*, `Job 1` executes successfully steps `Build` and `Publish`, but fails at the step `Deploy`, and therefore doesn't even start step `Functional Test`. `Job 2`, though, being able to reuse the cached results of previous successfully executed steps of `Job 1`, starts directly at step `Deploy` (i.e., it doesn't run steps `Build` and `Publish`), right after step `Checkout`, which is mandatory for all jobs.

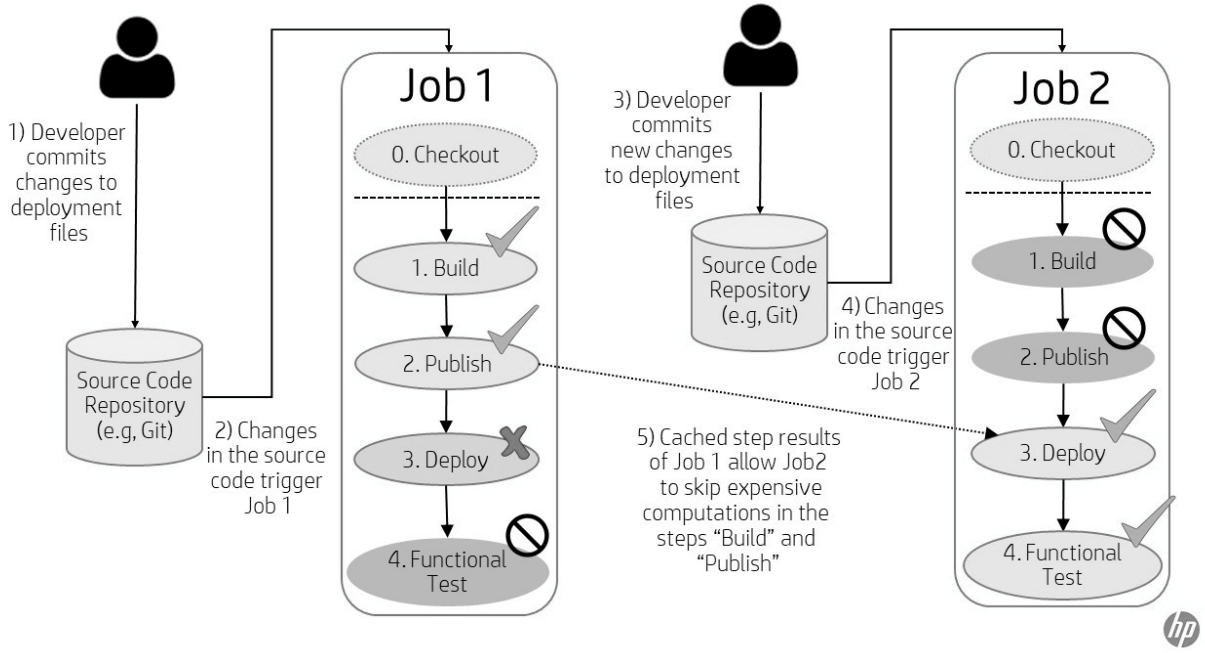


Figure 1: Cache-enabled execution of steps that highlights the re-use of static artifacts.

In the example illustrated by *Figure 2*, if the source code changes that triggered `Job 3` impact only functional tests, there is no need to necessarily re-run the `Build`, `Publish`, and `Deploy` steps again, if their results are cached. In other words, in *Figure 2*, `Job 3` executes successfully steps `Build`, `Publish`, and `Deploy`, but fails at the step `Functional Test`. `Job 4`, though, being able to reuse the cached results of previous successfully executed steps of `Job 3`, starts directly at step `Functional Test` (i.e., it doesn't run steps `Build`, `Publish`, and `Deploy`), right after step `Checkout`, which is mandatory for all jobs.

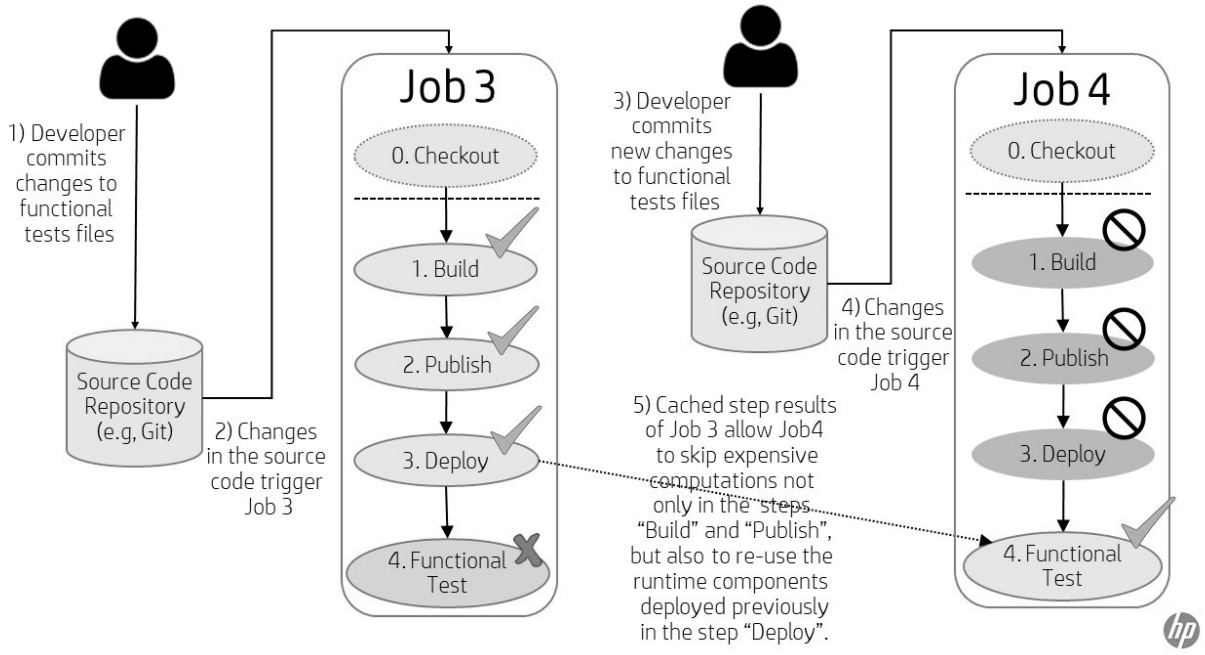


Figure 2: Cache-enable execution of steps that highlights the re-use of deployed artifacts.

To map files to steps, we inspect which files were accessed by each step using a widely available feature provided by most popular operational systems. In Linux, this feature is provided at least by the `stat` CLI tool. The example illustrated in *Report 1* shows what the output of the `stat` CLI tool looks like, with the last access timestamp highlighted in bold.

```
$ stat index.htm

File: `index.htm'
Size: 17137 Blocks: 40 IO Block: 8192 regular file
Device: 8h/8d Inode: 23161443 Links: 1
Access: (0644/-rw-r--r--)
Uid: (17433/comphope) Gid: ( 32/ www)
Access: 2007-04-03 09:20:18.000000000 -0600
Modify: 2007-04-01 23:13:05.000000000 -0600
Change: 2007-04-02 16:36:21.000000000 -0600
```

Report 1: Output of the `stat` CLI tool, which shows in bold the last access timestamp.

The detailed behavior of our proposal is presented in *Sequence Diagrams 1 and 2*. In these diagrams, the component `Git` represents the source code repository. The components `Pipeline`, `Job`, and `Step` correspond to the homonym concepts previously presented in Section *"What is a deployment pipeline?"*. The component `Step Files Mapper` is a key-value

storage for keeping track of what files are accessed by each individual step. Finally, the component `Step Results Cache` is a key-value storage for keeping re-usable step results for future use.

Among other things, *Sequence diagram 1* shows that:

- New files added to source code do not have a previous track of mapping to steps. Therefore, they always invalidate the cache, forcing the immediate subsequent job to run all its steps, as a “conventional” pipeline would do;
- Updated files and/or deleted ones trigger jobs that may make use of the cache, if there are cached step results that can be reused, *which may include metadata about deployed components*.

And, among other things, *Sequence diagram 2* shows that:

- the mapping of files to steps is built on top of tools that allow the retrieval of the last access timestamps of files, which are usually provided by most popular operating systems (e.g., the `stat` CLI tool in Linux);
- the cache of a step is always **updated** by a real step execution, regardless of its result. Moreover, all the caches of subsequent steps are always **invalidated** by a real step execution, regardless of its result.

A note on the required care that developers should have while using our proposal

The potential optimization brought by our proposal may be jeopardized by developers that do not restrict the number of files accessed by each step to the maximum extent possible. For example, an extreme scenario of such jeopardizing is one in which the first step of the pipeline validates license headers of all the files of the source code repository, even if this task does not necessarily result in any cache-able result that is required by the deployment process. From a deployment perspective, such an unnecessary validation of the license headers of all the files of the source code repository will mess with the last access timestamps as described in *Sequence Diagrams* 1 and 2 and will, therefore, invalidate all potentially valid caches that could be reused in subsequent steps.

Our recommendation is that our proposal should be used for deployment purposes only. All tasks not conceptually related to deployment (e.g., the validation of the license headers) should be done in other pipelines (e.g., CI pipelines) associated to the same repository. As the definition of a Continuous pipeline is pretty loose, there is no conceptual limitation for having several pipelines, including the deployment one, for a single source repository.

Disclosed by Mauricio Coutinho Moraes, Jhonny Mertz and Natalia Machado, HP Inc.