Maven Build Processes of the Model-Driven IIoT Platform oktoflow - A Performance Analysis

Holger Eichelberger
eichelberger@sse.uni-hildesheim.de
Software Systems Engineering, University of Hildesheim, Hildesheim, Germany

Abstract

Build processes and continuous integration are central to software development. Their inefficient execution may impact development speed, agility and motivation. In this paper, we identify performance improvements for Maven build specifications and, as an example, analyze the performance of the build processes of the open source model-driven IIoT platform oktoflow. For application builds, we achieve 22%-39% and for the platform about 70% build time improvements.

1 Introduction

Continuous Integration (CI) and build processes are essential for consistently developing complex software systems, even in the age of Artificial Intelligence (AI). However, various reasons may affect build process performance, ranging from superfluous build steps, issues in individual build steps, to slow hardware. As complex build processes may run for hours, performance issues can ripple up and slow down development.

In this paper, we identify performance improvement actions for Maven build specifications and apply them in a performance analysis of the build processes of the open source model-driven IIoT platform oktoflow[1]. This is motivated by the use of oktoflow as central technology in the BMFTR DATIpilot innovation community ReGaP[2], where funded community projects create own oktoflow applications. In this context, build performance issues may significantly affect work and motivation in the community projects.

Various publications discuss the performance of/in build processes, often not aiming to improve the build time. Among them, the works on Moobench and PEASS, e.g., [1], which integrate performance analyses into build flows. The evolution the Palladio build setup from Jenkins to github in [2] provides performance insights into full and incremental builds. Arntzenius et al. [7] and de Medeiros et al. [5] characterize the energy consumption of build processes. Zhang et al. [4] identify performance smells of build process configurations, an overlap with our work, and analyze effects of automated configuration repair.

In this context, we ask which actions can affect Maven build performance and how we can analyze the performance effects of the actions and the build steps in the context of the oktoflow IIoT platform. We contribute the background of our work (Section 2), the performance improvement actions in Section 3 as well as an experimental performance analysis (Section 4).

2 Background

oktoflow [3] is an open source IIoT platform, consisting of more than 80 mandatory, optional and alternative components. At its core, oktoflow is a distributed stream processor with specialized data connectors for Industry 4.0 protocols like OPC UA, MQTT, Modbus/TCP [6] and particularly enables the integration of Python-based AI. To ease the development of streaming applications (apps), oktoflow applies model-driven techniques, leading to the generation of various types of artifacts, including Java and Python code, tests and Maven build specifications, utilizing custom Maven build plugins, e.g., for code generation, Python syntax checks or unit tests.

oktoflow components apply usual compile-test-package-install-deploy build flows. oktoflow apps are built in four stages, each employing an own build specification. The 10 oktoflow example apps utilize a single build specification that orchestrates these stages. In ReGaP, users apply both, UI-based interactive as well as orchestrated builds (usually based on modifying selected example apps) on Windows and Linux.

The CI of oktoflow is realized as incremental build tasks on Jenkins covering all oktoflow components and, as integration-level regression tests, the example apps. Thereby, the CI is authoritative w.r.t. Maven snapshots, i.e., it deploys snapshot versions, but does not update them from other sources, which speeds up the builds and reduces build time variances.

3 Approach

By previous experience as well as observations of our build processes, we first identify a set of change actions that can affect the build processes, including the used Maven plugins that constitute the build steps. Then, we determine candidates for modification. Potential modifications are first validated locally and, if successful, committed and recorded as individual CI

[1] `https://oktoflow.de`
[2] `https://regap`

build timings. Ultimately, we validate the effects in a performance experiment by monitoring the execution times of the individual build steps and the entire build using different oktoflow versions as subjects (cf. Section 4). We focus on the following actions:

A1. **Improve own build plugins:** Custom oktoflow Maven plugins may cause performance issues. In contrast to official plugins, we can much easier modify the code, e.g., add execution logging.

A2. **Skip build steps:** Often, Maven plugins can be disabled through a flag. This happens at build runtime, i.e., the plugin is loaded, properties are transferred into the plugin instance and the plugin is (partially) executed. If skipping can be decided earlier, a plugin can be utilized as part of an optional Maven profile, i.e., it is loaded and executed only if the profile is active.

A3. **Remove build steps** if they are superfluous. If build specifications are generated, e.g., in a model-driven setting, plugins can be included/excluded during generation. This is an earlier variant of A2 as it is enacted before build runtime and without need for a Maven profile.

A4. **New hardware/installation:** Executing build processes on a more recent hardware may lead to significant performance improvements. For Jenkins, separate remote build nodes are recommended, which allow for isolation of software installation and resource usage. A virtualized installation, e.g., as a virtual machine, can ease setup and migration between physical hardware.

A5. **Update build plugins:** Most official Maven plugins are under active development, i.e., using a different version could affect the performance. Some become deprecated and shall be replaced.

A6. **Adjust plugin configuration** based on configuration smells [4], e.g., enable parallelism.

A7. **Update Maven:** Also the build system or the JDK can be updated. However, this may have unexpected impact on the build functionality as well as the compatibility of build plugins.

Build steps can be inherited from parent build specifications. While this enables reuse and concise build specifications, changes on parent level may also cause wider impacts. Further, decisions in particular for A2 and A3, may not solely be made by platform developers. In oktoflow, platform users can enable/disable certain build steps by model settings, thus, affecting the generation of build specifications, e.g., to activate/deactivate creation of code documentation.

Due to length restrictions, we focus now on A1-A4 and defer plugin replacement, version and configuration changes (A5, A6) as well as Maven/JDK updates (A7) to future work.

4 Performance Analysis

Based on the identified candidates, we modified oktoflow build specifications according to the actions A1-A4. In this section, we analyze the effects in terms of a performance experiment.

Subjects: Besides the build processes involving all oktoflow components on our Jenkins CI, we focus (due to space limitations) on one of the example apps. `examples.python` is a simple app often used as master copy by users in ReGaP. It streams artificial data from a Java source through a Python (AI) data processor to a Java receiver. On this subject, we enable the `maven profiler`[3], a build extension that records the execution times of the build steps even in sub-builds, in our case the four orchestrated stages realized via an extension of the `maven invoker`[4].

Changes: As baseline, we use oktoflow/app version 0.8.0 with no changes, while version 0.8.1 realizes the following changes: The code generator for `protobuf` as alternative data transport format as well as the coverage instrumenter `jacoco` are enabled by default in the parent build specification in version 0.8.0. `protobuf` is only needed when the respective protocol is activated in the model, i.e., it can be enacted on demand by code generation (A3). `jacoco` is a typical CI build step, which is better located in an optional Maven profile that is activated on the CI (A2). Further, we improved our code generation plugin (A1), which loads the XText-based models, performs code generation and executes generated build specifications. Optionally, a user may skip documentation generation (A2) through a setting in the model.

Setup: For execution, we use Maven 3.9.7 on Open JDK 21.0.1 installed on the CI server S (Ubuntu 24.04 VM with 8 cores and 64 GB RAM on an Intel Xeon E5-2420 v2 processor with 192 GB DDR3 RAM), the build node B (Ubuntu 24.04 server VM, 8 cores, 62 GB RAM on a AMD EPYC 9255 processor with 128 GB DDR 5 RAM), a server workspace O (Ubuntu 24.04 server VM, 4 cores, 32 GB RAM on the same hardware as B) and a development laptop L with Windows 11 build 26200 on an Intel i7-1265U processor with 64 GB DDR4 RAM. During the experiment, O is used only if B is idle. On O and L we use a dedicated clean user account. On Windows, the local Maven repository and the experiment directory are excluded from virus scanning; we reboot L and wait for stable CPU usage before the experiment.

App execution: After a basic installation of oktoflow and a sparse checkout of the example in both versions, we perform a full build as warmup on O and L. Then, an AI-generated script executes a typical development scenario (re-build of app-specific code and app integration) of the respective subject 10 times with 60 seconds cooldown. We collect the Maven console logs and the raw JSON logs of `maven profiler`.

[3] `https://github.com/jcgay/maven-profiler`
[4] `https://maven.apache.org/shared/maven-invoker/`

action / measurement	v0.8.0		v0.8.1	
	Win	Linux	Win	Linux
protobuf (A3) [s]	1.37	0.25	-	-
jacoco (A2) [s]	1.23	0.31	-	-
code gen (A1) [s]	59.98	17.32	10.69	7.04
javadoc (A2)* [s]	6.30	1.72	4.44	1.72
# build steps	83	83	57	56
min build time [s]	101.22	34.23	63.70	26.90
avg build time [s]	108.74	34.61	66.42	27.02
σ build time [s]	8.86	0.29	3.98	0.08
max build time [s]	130.21	35.06	77.79	27.17

Table 1: App build times for steps and build process.

A script combines the logs and converts them to CSV for further analysis in Excel[5].

CI execution: After a build triggered by a change commit, we archive the Jenkins overview HTML page, which states in hidden fields the build times of individual tasks in milliseconds. A script converts the task information to CSV for combined analysis in Excel[5].

App results: Table 1 shows the results for the example app, in the upper part for the individual build steps. As discussed above, `protobuf` and `jacoco` can be skipped (indicated by "-"). Most improvement stems from the code generation plugin through revised model loading, more focus on model elements effectively required by the app, and subsequently faster build steps. Optionally (marked by "*") skipping `javadoc` through a model setting may further reduce the build time. For the entire build process (lower part), we reduced the number of build steps, the build time (39% on Windows, 22% on Linux) and the variance. Potential outliers occur in no regular fashion, e.g., for v0.8.0 on Windows at the beginning, on smaller scale on Linux during the experiment. Linux builds run 2.4-3.2 times faster than Windows builds.

CI results: We migrated the CI execution from server S to build node B (A4), which reduced the overall build time of oktoflow from 3.8 to 1.8 hours, i.e., 53%. Further, improvements are skipping `protobuf` and `jacoco` 6%, disabling `javadoc` on most example builds 2% and above code generation modification (9%) as part of A1, in sum about 70%. Although not discussed in detail, all oktoflow example apps show an average build process speedup of 35% on the CI[5].

Limitations: Improvements to subject-specific complex build processes may not be generalizable. While the app measurements followed an experimental setup with repetitions, the CI measurements were done in a realistic setting, i.e., without repetition. Measurements on Windows may be affected by various factors such as virus scanner, Windows updates or the user interface. However, actions like hardware upgrades, disabling unneeded build actions and, in particular, improving (invisibly) inherited build steps are generalizable knowledge.

[5]Collected data and replication package: `https://doi.org/10.5281/zenodo.23103423`.

5 Conclusion

Build processes and regression tests are a backbone of developing complex software systems. Inefficient build processes can affect agility and motivation in software development. We discussed seven actions to improve Maven based build processes and applied a selected subset to the open source IIoT platform oktoflow. For oktoflow apps, we improved the build time by 22%-39% depending on hardware and operating system, for the entire platform by about 70% in total.

Further improvements could be achieved, e.g., by incremental code generation. We further investigate how Python or Angular development approaches can be transferred to our case. As early experience, when build processes only re-build relevant artifacts and applications re-load code at runtime, the build time can be reduced to less than 10 seconds.

Acknowledgement

This work is partially supported by the German federal ministry for Ministry of Research, Technology and Space through grant 03DPC1512B.

References

[1] D. G. Reichelt, S. Kühne, and W. Hasselbring. "PeASS: A Tool for Identifying Performance Changes at Code Level". In: *ASE*. 2019, pp. 1146–1149.

[2] L. Alber, N. Boltz, and L. Schmid. "Continuing to Catch Up with State of the Art Continuous Integration Pipelines in Palladio – The Experience Report Strikes Back". In: *STT* 43.4 (2023).

[3] H. Eichelberger and C. Niederée. "Asset Administration Shells, Configuration, Code Generation: A power trio for Industry 4.0 Platforms". In: *ETFA*. 2023, pp. 1–8.

[4] C. Zhang et al. "BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds". In: *ASE*. 2023.

[5] S. Q. de Medeiros et al. "Evaluating the Energy Profile of Tasks Managed by Build Automation Tools in Continuous Integration Workflows: The Case of Apache Maven and Gradle". In: *ICT4S*. 2025, pp. 12–22.

[6] C. Nikolajew and H. Eichelberger. "Industry 4.0 Connectors - A Performance Experiment with Modbus/TCP". In: *STT* 45.1 (2025), pp. 5–7.

[7] R. Arntzenius, X. Liu, and A. Zaidman. "On the Energy Consumption of Continuous Integration in Open-Source Java Projects". In: *SANER-C*. 2026, pp. 181–188.