DOCUMENTATION

# Continuous Integration

AN OVERVIEW FOR USER-BASED LICENSING

## SUMMARY

This document gives an overview of why moving to user-based licensing for Arm tools and models is very beneficial for users of continuous integration (CI) and other modern development practices. If you already have a grasp of CI and the challenges it can pose for our current licensing, you can skip straight to 'Advantages of User-Based Licensing' on page 4. The preceding section 'Challenges of Usage-Based Licensing for CI' is handy if are less familiar with CI. The final section 'Code Management Systems' is only useful if you're new to software development.

## INTRODUCTION

Arm development tools and models are changing, moving from a usage-based business model built on existing licensing technology, to a user-based business model built around new licensing technology. One of the key reasons for this change is to provide improved compatibility, performance, and an improved experience with modern development processes, including CI.  Whenever our tools and models help developers create great products quickly, efficiently, and effectively, we increase the success not just of our tools, but also of our architecture.

However, it's not always clear what "modern development" or "continuous integration" mean, why they are so valuable to developers, and how our move to user-based licensing drives value for our tools and our architecture. This document explains the key concepts and provides the background information you need to better understand our change in licensing. We look at traditional development techniques and the challenges they can bring, and contrast those with the advantages of more modern development techniques. We then look at the strains that modern development can place on our existing usage-based licensing and business models, and finally explore the benefits of our future user-based licensing.

By the end of this document, you should have the background you need to confidently discuss the advantages of our licensing changes. For simplicity we focus on software development, but many of the same concepts apply equally well to hardware development.

## TRADITIONAL SOFTWARE DEVELOPMENT

Developers start by placing a copy of the latest code from the code management system into a local development area. This is where development happens. When development is complete, the developer's changes must be submitted to the central code repository. However other developers have been submitting changes, and the local development copy has been getting progressively further from synchronization with the code in the central repository. To submit changes, the developer must first update the local development copy with the latest changes from the repository and address any conflicts. This process is called integration, and it's a manual task falling upon the developer.

The effort needed for this integration rises with the size and complexity of the code, the number of active developers, the rate of change in active areas of code, and most importantly, the time between integrations.

Integration costs can be significant and can reach a point known as "Integration Hell," where the time taken to integrate a change is greater than the time taken to create the change itself.

Other than the drop in development efficiency due to integration effort, there can be other less obvious impacts. Conflicts can be missed, leading to latent defects that can remain undiscovered for extended periods or can "escape" into released products. This can introduce significant manual testing and release cost to ensure the quality of each product release.

Missed conflicts can also cause scheduled/automated integration build/test cycles to fail, leading to further (potentially significant) costs in diagnosing and correcting the failure. Feedback to developers is delayed, leading to additional efficiency losses associated with context switching. Developers cannot "move on" from their previous task and fully concentrate on their next development task until a fully successful integration build and test has taken place. Iterative changes (including many defect fixes) become slow and cumbersome as the next iteration cannot start until integration testing of the previous integration is complete.

The biggest hidden cost though, is developer motivation and commitment. Developers did not join your organization to spend significant time fighting code integration, fixing broken builds, and waiting for tests to complete. They joined to create great code, to use their creativity and innovation skills to drive value for your business and success for themselves. Integration overhead, housekeeping, and feedback delays all drain developer motivation and make it more difficult for them to deliver their full potential. As developer motivation drops it become more difficult to attract, engage, and retain the best talent.

## CONTINUOUS INTEGRATION

CI isn't new and was first proposed by Grady Booch in 1991. However, it didn't gain serious traction until changes in technology brought major computing power (and specifically cloud-hosted computing) within easy availability. A significant majority of our tools users either have implemented, or are considering implementing, modern development practices, including CI. This is particularly true of big tools customers with many developers spanning multiple complex software projects.

An essential prerequisite of CI is expansive and automated unit-level testing, with test-driven development now commonplace. The primary advantage for this discussion is the increased probability of catching conflicts. If a change breaks existing code anywhere in the project, comprehensive unit testing should catch that breakage promptly. Developers can be more confident when checking in changes and there's a reduction in release costs as a higher level of software quality is maintained through the development cycle.

Comprehensive unit-level testing has additional implications as code becomes more modular and developers can focus on smaller units of code in isolation. This means that developers tend to check in smaller changes more often, with a corresponding reduction in integration costs.

CI further reduces integration cost by enabling developers to integrate changes on a more frequent basis. The upper bound of integration is generally taken as once per day, but in many cases, integration is much more frequent, tending toward the theoretical maximum of code being integrated continuously. There are about as many implementations of CI as there are teams implementing CI, but a typical implementation might have three stages:

1. During feature creation, developers focus only on the code unit that they are working on. This is made possible by modular code structure and comprehensive unit testing. The developer runs in a very tight loop of create -> build -> test, with build and test covering only the code being changed and being executed on a frequent, sometimes almost continuous basis. When a point is reached where all the tests for that unit pass, changes can be submitted.

2. At this point it's necessary to integrate changes with the latest code in the repository, then build the entire project and run all the project tests. Due to the rapid rate of integration and the wide unit test coverage, this process is lightweight and fast and can even be automated. The cost of integration falling on the developer is either significantly reduced or removed completely. As soon as this process has completed, the developer receives confirmation that the changes did not cause conflicts and have been accepted.

3. It's common for a third level to regularly or continuously build and test the entire codebase, including the latest changes from all developers. It's usual for this process to include additional tasks such:

a. Building and testing the code for a cross-matrix of all possible configurations. For example, a single change may trigger up to 500 separate builds.
b. Analysis of non-functional metrics, including performance.
c. Generation of documentation.
d. Analysis of test coverage.
The advantages of CI are widespread. This list is by no means comprehensive, although it contains the most frequently cited benefits:

- An increase in efficiency through the elimination of developer integration costs.

- A reduction in release costs and latency. Rather than needing to improve quality after development is complete, quality is built in as a core part of development.
- A reduction in maintenance costs through well-structured modular code, with comprehensive unit testing.
- An increase in efficiency through a reduction in expensive failing build/test cycles.
- In increase in efficiency through the removal of developer "context switching."
- Happier, more motivated developers facing a significant reduction in risk and housekeeping tasks.

However, CI is predicated on being able to make high volumes of build and test operations run quickly. Hence the dependence on high-end servers and auto-scaling cloud implementations: build and test must be parallelized to run as fast as possible. Where build and test are slow or delayed, or where failures occur, the system breaks down.

## CHALLENGES OF USAGE-BASED LICENSING FOR CI

As noted above, with CI and related modern development practices we see a significant upscaling of build and test activities. Build and test becomes central to the development and integration cycle, and there's a focus on getting these processes to complete as quickly as possible.

Build and test have to be considerably scalable and results are needed quickly, therefore there is significant focus on getting build and test activities to complete as fast as possible. Easy and cheap access to high-power multicore servers facilitates this by enabling build and test activities to be run in parallel.

The advent of cloud-based computing increases this effect. It doesn't matter if you need one virtual server for 100 minutes or 100 virtual servers for one minute, it's the same price. So, we are increasingly seeing tools users establish development and CI services using auto-scaling cloud infrastructure and taking advantage of the flexibility and scalability of cloud infrastructure to complete significant volumes of build and test operations in the minimum possible time.

These trends put pressure on our existing licensing and business model, with compiler and models particularly affected. This pressure comes in three main areas:

1. It's challenging to run a license server in your cloud because the license server needs a fixed predictable host ID (network address), which is hard to engineer. In addition, it's not uncommon to find that for security reasons, cloud installations have been locked down so that only specifically approved software can run. Just the legal discussions needed to get approval for a license server can be prohibitive. We've already encountered (and needed to create a bespoke solution for) big customers that are unable to use a license server. Our tools fall at the first hurdle: our current licensing solution is technically incompatible with cloud-hosted development.

2. There is license latency—the time taken to check out and check in a license can cause tools to run slowly. This specifically affects the compiler because it has a very short lifespan, making license latency appear significant. There are many variables including the size and complexity of source files being compiled and network capacity and load, so different users see different effects. We have customers who estimate the performance regression to be under 5%, and others who claim closer to 500% (builds take five times as long as they should because of license latency). For some working from home, the effect can be around a 100% slowdown—builds take twice as long as when working in the office and laptop cores are spending 50% of their time idle waiting for a license to be fetched. The biggest real-world impact we've ever seen is a project that

builds 35 times slower than it should due to team accessing a license server on a different continent.

3. Most significantly, our current usage-based business model is focussed on the number of instances of a tool running at once. Sequential compilation using a single core requires one compiler license; parallel compilation using 32 cores requires 32 compiler licenses. Modern development practices that exploit modern infrastructure to process builds and tests as fast as possible require significant license pools, and tools costs can scale quickly. The volume of tools licenses required can become a barrier to adoption of modern development practices.

Following from this there is a fourth, often unappreciated challenge. It is difficult, if not impossible, to calculate your peak license requirements for tools like compilers. Peak requirement is a combination of licenses needed for individual builds and collisions between builds, which is a combination of frequency and length of builds—which is a combination of available hardware, code size complexity, and dependencies. Often the only way to assess peak demand is to trial with unlimited licenses and review usage, but this analysis is time-consuming and often inaccurate, and is voided when changes are made to infrastructure or build patterns.

## ADVANTAGES OF USAGE-BASED LICENSING

User-based licensing directly targets primary challenges to the adoption of modern development practices that can be seen in our existing usage-based business model:

1. User-based licenses can be used offline, disconnected from the license server for extended periods. Once activated, a license becomes deployable. It can be bound into a container or docker for direct use in the cloud, so it's no longer necessary to run a license server in your cloud. As an additional alternative, we can provide host customer-specific cloud-based license servers.

2. Since the license can be used away from the license server for extended periods, tools no longer check the license server for an available license every time they execute. License latency is eliminated completely and tools such as compilers can execute at full speed.

3. Scalability requirements arising from the use of parallel infrastructure are completely removed. User-based licensing is concerned with the number of users, not the infrastructure they are using. A single license enables infinite concurrent tools instances for the same user, so exploiting high-end physical or virtual servers to expedite build and test processes does not require additional tools licenses.

4. Calculation of peak requirement is simplified down to a straight headcount. There is no need to consider development practices, code size and complexity, infrastructure used, or how these factors might change in the future.

In summary, user-based licensing gets out of your way. It enables you to use tools how you choose, adopt the development practices you prefer, and exploit the hardware you like, because these factors are irrelevant to user-based licensing. User-based licensing is only concerned with the number of users, and it delivers the best tools availability, flexibility, and performance possible to those users.

## CODE MANAGEMENT SYSTEMS

For decades now, code has been kept in some kind of code management system. Code is held in a central repository s, developers can fetch a copy of the latest code and submit changes to the repository. There are many different systems available but they all provide a set of core functionality including:

– Resilience: the code store is centrally managed and regularly backed up, so a disk or laptop failure/loss doesn't result in significant loss of code.

- Access management: with read write and management rights being assigned to appropriate individuals.
- Traceability and audit: code changes are timestamped, watermarked with the ID of the person making the change, and paired with explanatory comments. Individual changes can be reversed if needed.
- Version management: the code base can be "tagged" so that product releases can be recreated in the future (for example, for enhancement or for defect investigation).
- Process management: code reviews can be used to gate change acceptance, code branches can be created and merged as needed by product development, and so on.

Code management systems make it possible to manage significant software projects, and for multiple engineers to collaborate on the same code base.