Software Continuous Integration with Hardware

Brent Clausner
beclausner@sei.cmu.edu

Abstract

Validation and maintaining a stable code base are vital for software to be regarded as a quality product. An open question is how to ensure that the software you are developing will not be broken by another team within your organization making their own bug fixes? With limited hardware and growing business, how do we ensure the product works as expected with thousands of features being developed or already existing while new development is being done? Keeping costs down and software quality up it is necessary to test frequently and often. The overwhelming parallel development of new code along with feature enhancements, bug fixes and other modifications to existing code impedes the task of ensuring a product will function as expected. Business models may not fully support the hardware needs of teams to test a codebase appropriately to sustain high software quality. One good solution is to test periodically, use emulation, and seek out offending “bad” code.

Biography

Brent Clausner is a DevOps Engineer who has been working in the Software Development Life Cycle for 16 years. Within this time, he has worked as a System Administrator, Software Engineer, Software Engineer in Test, Tools Developer, and DevOps Engineer.

Brent is currently a DevOps Engineer at The Software Engineering Institute, Carnegie Mellon University. Here he has worked with new languages to develop prototype applications, web services, and pipelines to secure code bases for various languages. Previously he has developed detailed functional test plans, wrote code to split a program into multiple processes for producing multiple reports, developed tools for migrating data between systems, and setup multiple development pipelines.

Brent currently resides in Trafford, Pennsylvania along with his wife and child. Brent studied computer programming at Pittsburgh Technical Institute and computer science at Point Park University. He is an avid survival crafting, video game player.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM23-0633
Excerpt from PNSQC Proceedings
Copies may not be made or distributed for commercial use

1. Introduction

Executing tests on a regular basis and doing so upon every code submission is a very important part of ensuring an application works as expected. Doing this with software that is tied to a specific hardware can be difficult depending on the size of the workforce for the product. If 100 submissions occur in a short span of time, to keep moving quickly would require either a large amount of equipment or can become a long sequential queue for executing testing. Executing continuous integration in this fashion can miss some cross functional incompatibilities.

Having a lot of equipment is not cost effective. As the product changes new hardware will be required and updating the testing equipment can mean downtime for testing. At this time if code changes are allowed to go in, untested, it can result in code dependencies on “bad” code making the removal of those changes, difficult if not impossible.

Many functional teams working on an application may have dependencies on each other that would not show any problems with alterations to feature specific functionality until later in the release development process. Because of this, it is important to execute tests including all current code changes at once. Also targeting a specific feature area in testing may work most of the time, however dependent functional areas should also be tested. In this document I’ll go over some processes and tools that I’ve experienced use of and will share some of the benefits and pitfalls of doing things in this way.

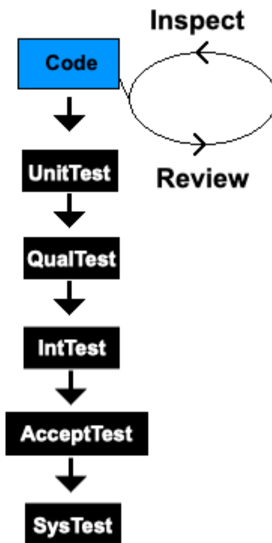
2. Current DevOps Integration Testing

Developing an application that serves a single purpose makes for easier design and focus. Exposing serviceable endpoints, via REST for example, need to be tested for functional completeness and needs to be robust. A single endpoint can touch multiple functional areas. Authentication should be implemented and used by every endpoint. The endpoint itself then may interact with another area, you can think of a creation of an object that gets stored in a database or memory. A few different feature areas will then have been touched, authentication and database.

Most software is designed to work with a specific hardware architecture. This makes development easier as you can implement a DevOps pipeline that covers what the application is expected to do. Cross platform building is also possible and can add a layer of complexity but generally can be virtualized to allow parallel building and testing.

After code is developed, it needs to build. This is the very first step towards quality. Within this phase unit testing can be done within the same language being used. Most languages have testing frameworks that allow this to take place. When an engineer looks to have code submitted, the normal phase of code review and inspection can help to alleviate issues with normal flow and typical pitfalls. Within configuration management, pipelines can be used to automatically run tests to ensure that code meets standards necessary put forth by the organization. This is a location which can include things like static code analysis or enforcement of coding standards can take place.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633



Development Stages. [1]

When coding is typically being done, the normal phased approach for this allows us to have our code inspected and reviewed by peers if the team size allows it. This is then followed by unit, quality, integration, acceptance, and system testing. Unit tests that are written either beforehand or during the coding phase to lock in codes functionality. Quality testing is done to ensure that the code is functioning as expected. Then testing is normally accomplished by other teams where it is worked for integration with the rest of the code base. Acceptance testing allows it to be used by other teams. Finally, system level testing can stress the test with more complex scenarios. [1]

3. Software for Specific Hardware

Developing software for a specific piece of custom hardware does have its own challenges. There are multiple layers to what is needed to test a version of software. The software needs loaded unto the system. Configurations will need to be applied after it's initialized. Testing can then begin. At this point, several different functional areas have already been touched. Each of those need to work for testing of different feature areas to begin.

Getting to a point where the system is ready for testing, can take a long time. The more complex the hardware is the longer it can take and more opportunities for things to go wrong. Using a product like rconsole can allow connections to take place without manually being at the keyboard. This is good for being able to access things like the BIOS. If you must issue any special commands to have the system load in a debug mode can be done this way too. Most automation looks for output to then proceed to the next step. Most times, initialization can have issues with timing. Order of processes coming up need to make sure that there are no race conditions that would cause any failures.

Errors with the hardware may be displayed to the console only and need to be tracked as well as any standard output. Logging the information as you execute automation is very important to be able to determine what occurred with a given system. This also can interrupt automation as the output may look garbled up with characters overwriting each other. Special care is needed when writing automation that handles checking the output with a console connection.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633

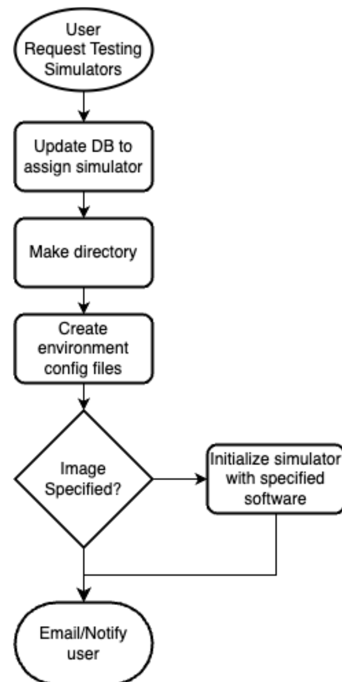
Timing of commands and the results from those may not be instantly complete. Most automation works such that when the output is seen, it immediately executes a command. A good example of this would be the “login:” prompt. When you see that, the system is ready for a username to be inputted. This can be used for ensuring that the system has loaded a specific module necessary for testing. If you work with an asynchronous system and you issue a command that makes a call, your result of the command may not be seen as fast as the automation can check. A lot of time this means it’s necessary to loop over a command to verify that what you did, resulted in the expected output. As an example, if your system takes a backup of the configuration, this job typically happens in the background. The job can include taking a snapshot of an internal database, then compressing it, and finally updating the internal database to indicate the status of the backup for the user interface. This can require any test that issues the creation of the backup to loop over a command to see the created objects, that waits for the desired object to show up. The system should still be available when this job is running. Automation should not wait forever in this case and a reasonable time should be listed in the design documentation for the maximum. This factor plays into the usability of a system that you want it to be responsive. New work may cause additional delays in commands from running and if automation is relaxed to much that when you go to manually use it, it feels that it has too much lag in responding to your inputs.

Integrating hardware into a git pipeline directly can be troubling with this type of product. Typically, it requires having additional server processes running. Having another process required to execute commands on a system can lead to performance degradation and issues like running out of file descriptors. With specific hardware it’s important to keep the product as close to a deliverable as possible. If there is another machine in between the product and the test execution gives another point for failure. The more that we add in between the higher the risk for failure outside of the product and relates to the infrastructure.

4. Emulation is Important

While executing tests on the actual product, having an emulator drastically cuts down on the cost and time to test. Having a product that can be loaded as a Virtual Machine makes it easy to load up a system on the fly. This can cut down time to test at any phase.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633



Sample testbed creation flow.

Emulated hardware allows you to have a farm of bare metal machines that can be used by any team to execute automation or manual testing. When you combine that with continuous integration testing, it can be very powerful. This lets you submit code in and execute tests from any team. Having the proper code coverage is necessary to ensure that every feature area is protected. Implementing a common flow like the figure above, you can automate these to look for available systems and submit test jobs periodically. With the setup being a directory for housing the environment settings it allows users or automation to run from a specific location with everything setup to execute tests or see the connection information to allow manual or hybrid manual automation tests.

A common template for setting the emulator makes developing testing suites for this environment quick as well. Common methods can also be analyzed for speeding up the timing on getting it ready. Shared templates allow that to be done once and every testing suite benefits. As an example, during initialization it is good to look for multiple asynchronous commands that are used to be called one after another and letting the simulated system handle multiple processes at once. Have the automation check at the end for all created objects and command completion rather than issuing a single command and wait for completion.

Having an infrastructure setup like this allows the number of tests that the system can handle to be a lot larger than just a single machine. With a test framework that allows you to split your process into many threads can help to speed up testing a bigger set of tests. Sub-processing out the testing portions of the automation, you can execute tests that do not touch the same functional areas within the software to allow more to go on at once safely. Negative tests that crash any processes mostly should be avoided with shared resource testing, as it can negatively impact other functional areas. When the automation test run is complete, it is important to update the database to allow the emulator or hardware to be provisioned again. This allows for multiple teams to add to the testing automation a lot easier. Scaling up the number of bare metal machines and emulators is very easy with a setup like this. It also does not limit you on only using emulators, but you can add actual hardware to the infrastructure if there are specific things that can only be validated there.

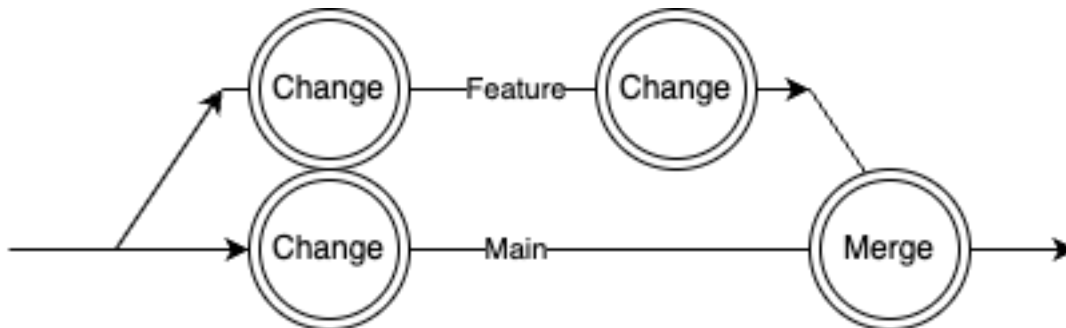
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633

Generally having a bare metal machine like an ESX server offers a good solution in this space. Having a machine that hosts multiple virtual machines like this can offer benefits for hosting as it cuts outs unnecessary resources. This eliminates running the operating system under the virtual machine. [2]

5. Single Main Branch for Developing

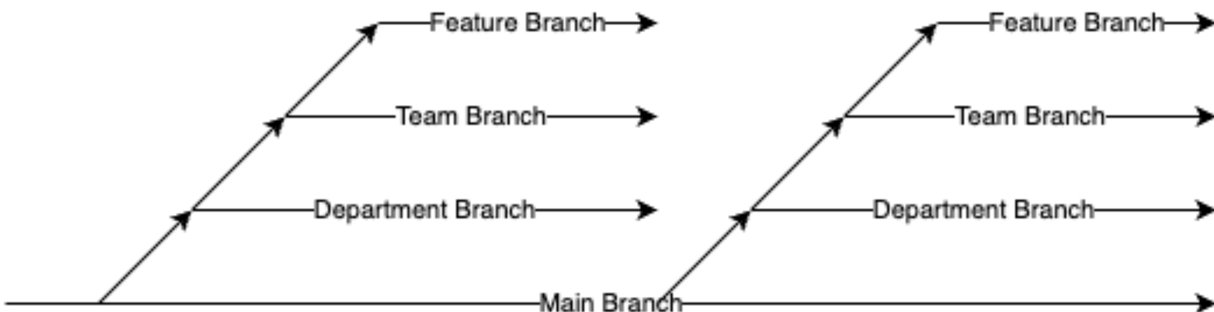
For a small software project, with a single main branch and developing on a branch from that main is a good way to keep the code clean. Having product release branches that are versioned are necessary for supporting a product for the long term. If the configuration management system is setup in a way that each functional area has its own branch, this can lead to a huge delay in changes being propagated throughout the code tree.

Looking at the hardware initialization phase, if a change has been committed that breaks the flow for users having things configured in a normal manner, no testing of other functional areas can take place. Using a single main branch for code submissions can have a large impact to several teams in this case. Having multiple branches not shared between teams can cause long delays in seeing integration problems. With specific tests that look only at the functional area may pass or have specific requirements that other areas do not typically use.



Simple branching of a feature off the main branch.

Branching from main, while working on a feature area may take time and can also require multiple branch synchronization. This can cause issues for a team working on the same areas of code, so it is a better practice to submit smaller incremental code and have it merge in. The code can have longer soak time and will be touched by multiple users.



More complex example of branching.

With a more complex layout of branches as an example looking at the figure above, at each merge between branches testing would be necessary to take place to ensure that each level of automation passes. That will take a long time for it to go from one branch to another, I have seen it take over the course of months to propagate changes in between feature branches like this. Cherry picking changes

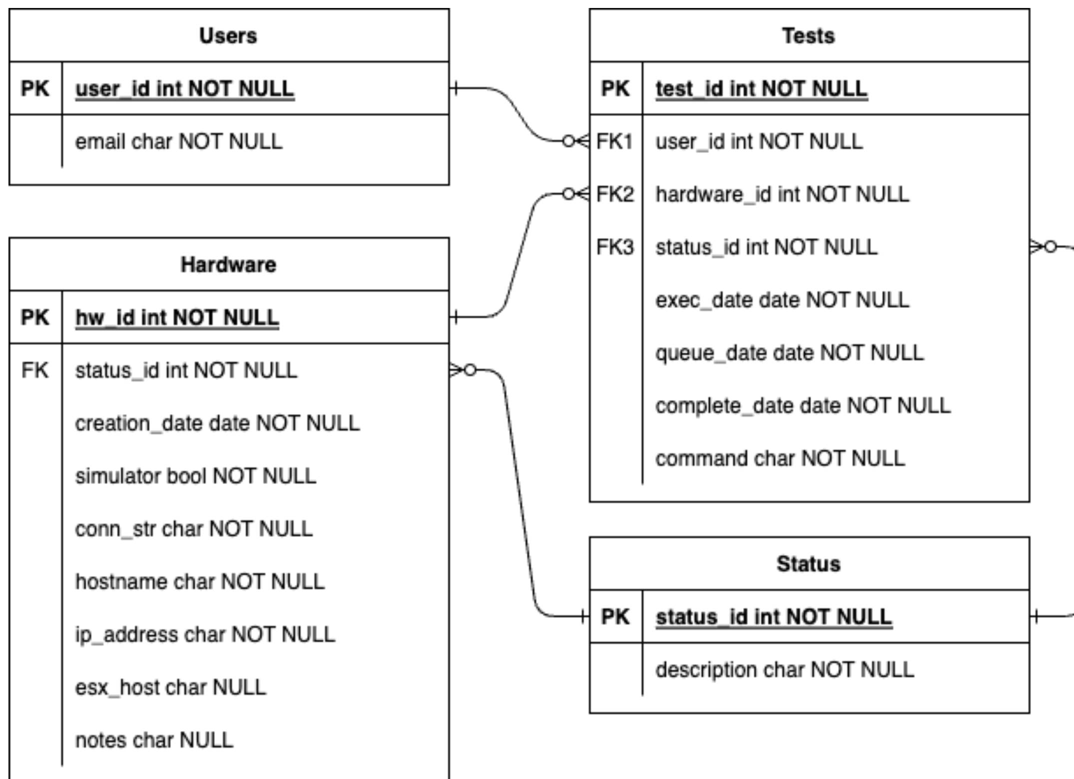
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633

between branches can result in incomplete code as a feature may rely on other code that would be needed. This can be an option, however if the code changes drastically in between picking and developing the feature work it can be problematic with syncing and merging. This way didn't work well from my experience.

If we have a hundred teams working off the main line at any given time, there can be a lot of changes being made. Each submission if it has testing being done upon submission will flood the integration testing infrastructure can result in major delays. Using a scheduled integration testing system can combine all the active changes at once for testing.

When you test periodically and not on every submission, you can get a better picture of quality to the product if the testing being done covers your feature area. You will get failures, but if you have an emulator in use this can allow you to sort out failures. If you have a simple numeric based commit system, any number of sorting algorithms can be used. This lets you try different ones to see what works best for you.

Doing continuous integration testing like this requires multiple tools to get things to work well. For instance, using git normally will fire things off immediately and if you want to schedule you need another system. Something like Jenkins will allow you to schedule things as a cron. Periodically testing like this would need to access hardware and it's handling of the hardware with software running on it as well, so it suffers from similar issues as a gitlab-runner. That requires additional software to be running on a system to execute properly [3]. Any additional process resources may impact how testing is done.



Sample Database Diagram

I haven't found any good product that handles hardware management that doesn't require a database to be setup and managed by hand or written proprietary software. In the above diagram you can see an

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633

example of how a database scheme can be setup to allow automated testing or manually allocated emulator or hardware. The last time that I've looked it was the best practice to have the database manage the hardware with a state machine that can manage if the hardware is in use or not. There are various states that need to be taken into consideration like this such as online, offline, in-use, and in-error. Managing emulators in this way also allows you to store proper connection information with it. This still requires an interface that allows the database to be updated for requesting the simulators and it can be expanded to allow reservation of hardware or for that to happen when testing fails.

Integration testing like this need to still run every test suite to validate all changes. When the offending change has been found, it can be rejected from the system to keep the main branch clean. This is why it is so important for a feature developing team to write good tests to ensure their area will work as designed.

A benefit of doing things this way is that you can write code and submit it after doing your due diligence of validating the functional area the code was written for, you do not need to worry about breaking other teams. This is because the system does the testing for you, and you need only worry about the areas that you are aware of affecting. When it comes to areas that have multiple dependencies on, it can be good to execute the integration tests prior to submitting the code into the main line. That should be done prior to having the code reviewed.

Having new testing suites being developed, you can include them with the normal runs and not allow them to be considered for code failures. It is important to have several passing runs along with a lot of clean intermittent runs being done prior to being allowed to reject code. This gets easier as the templates for the infrastructure and testing suites are developed.

This does not mean you cannot use something like on code submission for executing and it makes sense to have static code analyzers to scan your code for any problems. These types of tools are still a good thing to implement.

6. Drawbacks

Intermittent failures can be devastating and lead to false positives for valid code changes. Executing testing on known good changes, with a lot of the same testing going on can help to identify problems with the tests that are being executed and specific infrastructure pieces. That can be done to help make the integration testing system work well. Sometimes it is necessary to mark test suites as not reliable and to allow code to be submitted that causes failures in those cases.

Delays can occur because of a system like this. Having multiple "bad" code changes that break the same area can cause problems with finding them. It may be necessary to stop allowing code submissions in to sort out the problems. For instance, if a developer submits code that alters an internal replicated database table layout that causes a backup database object to fail creation and another developer submits code in that breaks creation of files for a backup, that testing area will take a while to find the bad changes. It may be necessary to lock the main branch to have time to find and remove all the bad code submissions. Locking the main branch and preventing submissions delays everyone from getting work done but finding the problem change(s) is more important for the sake of the product. This can be minor though, as developers use the system, they learn to execute tests for other areas that are depending on their code prior to merging.

The requirement of custom automation to accomplish this will have drawbacks of its own. The owning organization may need to set a team itself to handle common automation techniques to have the system run reliably.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633

Overall time to get code submitted and marked for approval can be a while. Working up to the last minute is not a good idea. With scheduling periodic runs of testing, it can make it difficult for submitting code prior to going on vacation. If the change is rejected, you may leave people depending on your code to wait on testing or integrating with it.

7. Conclusion

Using a configuration management infrastructure with a single main line and having timed periodic integration tests has led me to have a passion in software quality control. With having some awareness of pitfalls such as timing of commands and commands executing very fast, knowing ahead of time of what to do with polling for objects really helps to produce quality automation quicker. Reducing costs for businesses and using emulation for the product should be a priority for any organization that produces hardware. When it comes to working with many teams on the same product, I’ve seen it have many benefits with product quality. Writing test automation that protects the feature area that I am concerned about and seeing failures for code submitted from other teams when their changes break an area that gets removed is wonderful. Preventing code breaking changes from going into a product and being able to test ad-hoc has led to a faster time to test with as much code changes from an entire product line.

There is a lot of data that can also be captured using systems like this. Organizations can make their own determination of what data is valuable. It can show things like teams that prevent the most breaking code from being submitted into the main branch. Fixing intermittent problems results in a major impact as those are normally the most devastating. For me it is very satisfying to fix issues like this as they are the most difficult to find.

References

1. Tim Menzies, William Nichols, Forrest Shull, & Lucas Layman (Retrieved on 2023, May 29) Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the lifecycle. Retrieved from: https://link.springer.com/article/10.1007/s10664-016-9469-x
2. David Davis (Retrieved on 2023, May 30) What is VMware ESXi Server and Why do I Need It? Retrieved from: https://www.pluralsight.com/blog/it-ops/what-is-vmware-esx-server-and-why-you-need-it
3. Install GitLab Runner (Retrieved on 2023, May 30) Retrieved from: https://docs.gitlab.com/runner/install/index.html

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633

Legal Markings

The following markings MUST be included in work product when attached to this form and when it is published. For purposes of blind peer review, markings may be temporarily omitted to ensure anonymity of the author(s).
Copyright 2023 Carnegie Mellon University.
This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.
NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM23-0633