Riker Evaluation Guide for ATC ‘22 Artifact Evaluators

Thank you for taking the time to evaluate our artifact. This guide provides instructions to set up a virtual machine image with Riker preinstalled, to compile and test the Riker build tool, and to run the suite of benchmarks we used while evaluating Riker for the paper.

The accepted version of the paper is included in the artifact package with this README.

# Table of Contents

[Table of Contents](#_ff4uwpcasigh)

[The AEC Test Environment](#_1u4hkc2osn3)

[Prerequisites](#_kxxgnux0btnk)

[Setup Instructions](#_qmgwubt9wapl)

[Kicking the Tires](#_i4dmabckuzg1)

[Building Riker](#_7dai0yfvwi67)

[Running Riker's Test Suite](#_g40e1ii5vegu)

[Generating Graphs](#_v0dhdtyrfmx6)

[Running a Short Benchmark](#_7jmefphrd8hh)

[Full Evaluation](#_byasm1mqy6gr)

[What to Expect](#_pm28hsz5ojww)

[The Benchmarking Script](#_mmtbtelkzuex)

[Full Builds (to measure Riker's overhead)](#_qc6y24rafvor)

[Case Studies (to measure Riker's incremental build performance)](#_qa8opt2wsaym)

[Scalability](#_fsq2arms8p6y)

[Setting Up](#_s0e20kejdlfc)

[Measuring Scalability of the make Build](#_hg3c32gwxopb)

[Measuring Scalability of Riker's Build](#_zgeg5d7sdcqd)

[Using Riker](#_1aj7459klig)

# The AEC Test Environment

In our paper, we evaluated Riker on a bare-metal Linux machine with an Intel Core i5-7600 processor, 8GB of RAM, and an SSD running Ubuntu 20.04 with kernel version 5.4.0-80. We have since deployed Riker to a variety of other Linux kernels and platforms (e.g., arm64), and we observe that Riker’s state modeling is sensitive to minor changes in system call semantics. To eliminate such concerns for the purposes of artifact evaluation, we have packaged Riker in a virtual machine with a configuration we know Riker supports.

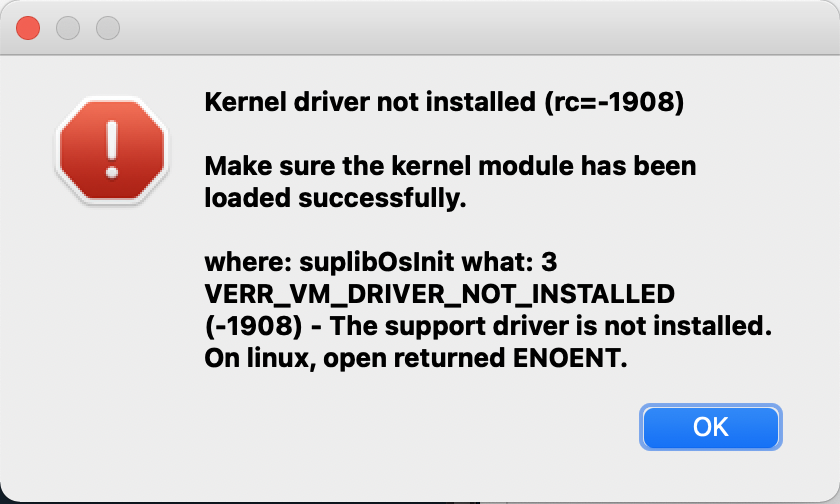
# Prerequisites

We used VirtualBox 6.1 running on Linux and macOS hosts (both with Intel x86\_64 processors) to test our virtual machine. We recommend and provide instructions for reviewers to use VirtualBox. However, since the VM is packaged as an OVA file, experienced reviewers may use a different virtual machine.

The latest version of [VirtualBox can be downloaded here](https://www.virtualbox.org/wiki/Downloads), or you can install it with your preferred package manager. Note that (for some reason) Oracle no longer lists macOS binaries on their website, but that [a version for Intel Mac can be downloaded from Oracle here](https://download.virtualbox.org/virtualbox/6.1.34/VirtualBox-6.1.34-150636-OSX.dmg).

Once reviewers install VirtualBox they may proceed to the next section.

# Setup Instructions

1. Download the riker-aec.ova file from the artifact package that contained this README.
2. Start VirtualBox.
3. Click the “Import” icon on the Oracle VM VirtualBox Manager window.  
   
4. Select “Local File System” as the source, click on the  icon near the File field, then navigate to the location where you downloaded the riker-aec.ova file.
5. Click the Continue button.
6. We recommend that you accept the default Appliance settings by clicking the Import button.
7. (optional) The Riker VM is configured to use NAT networking. If you switch to bridged networking you may be able to reach the virtual machine at the domain name riker.local, depending on your network configuration. This may be more convenient if you prefer to use your host's terminal and an SSH connection to access the VM, but bridged networking may require some additional configuration in your host OS.
8. Start the VM by clicking on the Start icon (it looks like a green arrow).
   1. Note: if you encounter a “Kernel driver not installed (rc=-1908)” error on a macOS host, the issue is that the macOS System Integrity Protection (SIP) mechanism is preventing the VM from running.  
      
   2. SIP does not need to be disabled, however, we do need to allow the VirtualBox application to run. See [the following instructions](https://medium.com/@Aenon/mac-virtualbox-kernel-driver-error-df39e7e10cd8).
9. After the Riker VM starts, you should see a login prompt that prints:  
   Ubuntu 20.04.4 LTS riker tty1  
   riker login:
10. Login with user aec and password aec.
11. The Riker VM is now set up and you should be able to proceed to reproducing Riker benchmarks.

On some computers, the VirtualBox console is too small to read. To fix this, look for an icon that looks like  on the bottom of the riker-aec VM window, left-click it, and then choose Virtual Screen 1 and then a scale factor that looks good to you (e.g., 200%).

The aec user has superuser privileges and an appropriately-configured /etc/sudoers file so that you can just do $ sudo <whatever> to install your preferred packages on the virtual machine.

# Kicking the Tires

To test the basic functionality of the Riker artifact, we will walk through steps to do the following:

1. Build Riker
2. Run Riker's test suite
3. Generate graphs from the paper's experimental results
4. Run a single benchmark to ensure the benchmarking scripts all work properly

## Building Riker

1. Log in to the virtual machine and navigate to the riker directory:  
   $ cd ~/riker
2. Build Riker's debug target, and then its release target  
   $ make debug  
   $ make release
3. There should now be rkr binaries in ~/riker/debug/bin/ and ~/riker/release/bin/, along with a small number of other targets.

## Running Riker's Test Suite

1. In the riker directory, use make to invoke the test suite for the debug and release builds:  
   $ make test-debug  
   $ make test-release
2. Each test directory contains one or more test cases. Successful tests are indicated with a '.' character. Skipped tests appear as an 's', and failures output '!'. You should see the following summary at the end of the test suite output:  
   Ran 113 tests, 107 passed, 6 skipped, 0 failed.

The pdflatex tests depend on a TeXLive distribution, which is quite large so we have not included it in the VM. To run these tests, install TexLive:

$ sudo apt install texlive

You should then re-run the test suites. The first pdflatex test will fail on the first run because pdflatex bypasses output redirection on the very first run after installation. This mysterious behavior leads to a spurious test failure, but it is unrelated to Riker. The tests should all pass on the second run.

## Generating Graphs

The VM includes results collected for the Riker paper. The following steps produce the paper's graphs from those results:

1. In the riker directory, use a script to gather results from each of the benchmark subdirectories:  
   $ scripts/results.py  
   This script generates several files in the results directory, and prints out summary statistics (which are also saved in results/summary.txt)
2. Generate the overhead graph:  
   $ graphs/overhead.r
3. Generate the incremental build savings graph:  
   $ graphs/savings.r

There should now be two PDF files in the graphs subdirectory. You can use transfer.sh to conveniently move these graphs off the VM so you can see them:

$ curl --upload-file graphs/overhead.pdf https://transfer.sh

$ curl --upload-file graphs/savings.pdf https://transfer.sh

Each curl command should print out the URL where the PDF is available for download (for a limited time).

## Running a Short Benchmark

Finally, we will use the included benchmarking script to run one of the faster benchmarks. This should only take a minute or two to complete.

1. In the riker directory, use the benchmarking script to run the xz benchmark:  
   $ scripts/runbench.py full-build rkr xz
2. The benchmark script stores the build times for xz in the file benchmarks/xz/full-build-rkr.csv (replacing the old results).
3. Riker also measures the "nop" build time. This is the time it takes the build system to determine that no work must be done immediately after the full build. These results are saved in benchmarks/xz/nop-build-rkr.csv.
4. (optional) Repeat the steps in the previous section to regenerate the overhead graph using the newly-collected build times. The results will only be meaningful if you also first run the xz benchmark with its default build system (make), as your baseline build time is likely very different than ours.  
   $ scripts/runbench.py full-build default xz

# Full Evaluation

All of the graphs used in the paper can be produced using the Riker automated benchmarking tool. This tool downloads prerequisites as necessary and builds them repeatedly (five times for all except LLVM, which builds just three times) in a timing environment. Benchmark results are written to files in each benchmark's subdirectory, and are gathered and turned into plots using the procedure outlined in the [Generating Graphs](#_v0dhdtyrfmx6) section above.

## What to Expect

Your test environment may differ substantially from the “typical developer workstation” we used in our evaluation. Those differences, combined with the added overhead of a VM, means you should expect to see different absolute time measurements and overhead (figure 3 in the paper). However, we expect most virtualization overhead will apply equally to builds with Riker and the other build systems we compare to. It is possible Riker's tracing may be affected more by VM overhead than other build tools, but we have not observed this in our trial runs. However, we have not re-run the entire evaluation in a VM.

When performing incremental builds, Riker should run the exact same incremental builds on the AEC VM as it did on our evaluation machine; this is because the set of commands that must run to update a build depends on which files change, not the runtime of the build. If overhead is slightly higher in your test environment than it was on our evaluation machine then we would expect Riker to save slightly less time doing incremental builds, but incremental build performance should still be much closer to that of the project's default build system than it is to running a full build on every commit. In short, the incremental build savings graph (figure 4) should be very close to what appears in the accepted paper.

One area where we expect virtualization to have a larger impact is on scalability. Riker's primary evaluation is on serial builds, but the results we report for scalability on parallel builds may be distorted by virtualized multi-CPU systems.

## The Benchmarking Script

In the top-level riker directory, run the benchmarking script to view a help screen that lists available experiments, build tools, and benchmarks.

$ scripts/runbench.py  
Usage: ./runbench.py <experiment> <build tool> <benchmark>...  
Experiments:  
 case-study Record time and command counts for incremental builds  
 full-build Record the time to complete a full build  
 all Run both experiments  
  
Build Tools:  
 default Use each benchmark's default build system  
 rkr Build with riker  
 rkr-parallel Build with riker in parallel mode (not included in 'all')  
 all Run with all build tools  
  
Benchmarks:  
 autoconf Supports full-build  
 calc Supports full-build  
 coreutils Supports full-build  
 llvm Supports full-build  
 lsof Supports full-build  
 lua Supports full-build  
 make Supports full-build  
 memcached Supports case-study and full-build  
 protobuf Supports full-build  
 redis Supports case-study and full-build  
 riker Supports case-study and full-build  
 sqlite Supports case-study and full-build  
 vim Supports case-study and full-build  
 xz Supports case-study and full-build  
 xz-clang Supports case-study and full-build  
 all Run all benchmarks

We will walk through both of the experiments the benchmark script runs in subsections below.

## Full Builds (to measure Riker's overhead)

The full-build experiments measure the time it takes Riker and a project's default build system to build the entire project. These times are used to calculate Riker's absolute overhead. As a byproduct, this experiment also measures the time required to perform an incremental build immediately after the full build (a build that does no work).

You may choose to run every benchmark's full build experiment. Note that this took **more than 11 hours** on our bare-metal evaluation machine.  
$ ./runbench.py full-build all all

You can instead update results for a subset of benchmarks to limit evaluation time. We break benchmarks into rough categories by build time in the items below:

1. To run builds that complete in under one minute (per build), run the command  
   $ scripts/runbench.py full-build all autoconf calc lsof lua make memcached redis xz xz-clang
2. To run longer builds that take more than a minute (per build), but less than 5 minutes:  
   $ scripts/runbench.py full-build all coreutils riker sqlite vim
3. And finally, to run the longest-running builds:  
   $ scripts/runbench.py full-build all llvm protobuf

Once you have run full builds of some or all of the benchmarks, follow the steps in [Generating Graphs](#_v0dhdtyrfmx6) to produce an updated plot of Riker's overhead compared to each project's default build system.

## Case Studies (to measure Riker's incremental build performance)

Case studies compare the performance of Riker's incremental builds to each project's default build system over the course of 100 real commits to each project's version control system. Benchmarks built for these case studies use a Rikerfile that replaces the project's existing build specification, rather than simply wrapping the project's Makefile as we were able to do for many of the full-build benchmarks. The Rikerfile does not describe any incremental build steps (see benchmarks/redis/files/Rikerfile for an example), so speedup on incremental builds is entirely due to Riker's build algorithm.

Case studies take a long time to run. A case study for a single benchmark will run 100 builds with Riker, 100 builds with the project's default build system, and another 100 builds with Riker wrapping the default build system to capture all commands run during each build. Many of these builds are faster than full builds because both Riker and the default build systems can save work via incremental builds. This experiment is meant to compare that time savings between the two build systems.

The redis benchmark is a good starting point for collecting new incremental build times. To run the redis case study, run the following command:

$ scripts/runbench.py case-study all redis

Other case studies will likely take longer, with the exception of xz. However, it appears the xz repository history has been manipulated (possibly via squashed commits) since our initial evaluation. The case study with xz's default build tool will fail on older commits because the repository does not contain a CMakeLists.txt file over all 100 revisions, but Riker's builds still work as expected across all 100 commits.

Once you have run one or more of the case studies, follow the steps in [Generating Graphs](#_v0dhdtyrfmx6) to produce an updated plot of Riker's incremental build time savings compared to the default build system.

## Scalability

The scalability results reported near the end of our evaluation do not have an automated benchmark script. However, the results are relatively easy to collect manually. Our evaluation used LLVM for scalability because it is a particularly large build that we know is easily parallelized via make -j, but any build that uses make and offers sufficient parallelism could be substituted.

The purpose of this experiment was to show that Riker's tracing does not excessively degrade parallel scalability of a build that already runs commands in parallel. To do this, we run a build of LLVM with one thread (make -j1), and compare the runtime to a build with Riker using one thread. We repeated this process for two, three, and four threads. We only stopped at four because that was the number of cores on our evaluation machine, but there is no reason you could not explore greater degrees of parallelism. Evaluating with a larger number of cores would actually reduce Riker's impact; Riker typically occupies one core, so the more cores there are the less Riker takes away from the build (as a percentage).

We now walk through each step to collect the scalability results for LLVM that appear in the paper.

### Setting Up

1. Download a copy of LLVM release 12.0.0:  
   $ wget https://github.com/llvm/llvm-project/releases/download/llvmorg-12.0.0/llvm-12.0.0.src.tar.xz
2. Extract the LLVM release  
   $ tar -xf llvm-12.0.0.src.tar.xz
3. Create an llvm-build directory and move to it  
   $ mkdir llvm-build  
   $ cd llvm-build
4. Configure the build with cmake  
   $ cmake -DCMAKE\_C\_COMPILER=clang -DCMAKE\_CXX\_COMPILER=clang++ -DCMAKE\_BUILD\_TYPE=Release -DLLVM\_TARGETS\_TO\_BUILD=X86 -DLLVM\_ENABLE\_BINDINGS=0 ../llvm-12.0.0.src

### Measuring Scalability of the make Build

We will now run four builds of LLVM, increasing the number of threads on each iteration.

1. Stay in the llvm-build directory. Build LLVM with one thread and record the time.  
   $ time make -j1
2. Clean up after the build  
   $ make clean
3. Now run a build with two threads and record the time.  
   $ time make -j2
4. Clean up after the build  
   $ make clean
5. Repeat the above steps for 3 and 4 threads. Calculate reduction in build time with N threads using the formula , where is the time to build with threads. Perfect scalability would yield reductions in build time of 50%, 66.7%, and 75% for two, three, and four threads, respectively.

### Measuring Scalability of Riker's Build

Next, we will run the exact same builds as above, but this time with Riker's tracing and filesystem model.

1. Create a Rikerfile with the following contents:  
   make -j1 --always-make  
   This Rikerfile will run a make build with one thread. When Riker runs this build it does exactly what make would, with the added overhead of tracing and modeling the filesystem.
2. Run a Riker build with the Rikerfile above and record the time it takes.  
   $ time ~/riker/release/bin/rkr
3. Remove the remnants of the previous build  
   $ rm -rf .rkr  
   $ make clean
4. Update the Rikerfile so it has the following contents:  
   make -j2 --always-make
5. Run a Riker build again and record the time for two threads.  
   $ time ~/riker/release/bin/rkr
6. As above, repeat the process for three and four threads. Compute reduction in build time.

We expect that you may see significantly different scalability results for Riker in a virtual machine compared to what we find in the paper. These results are sensitive to the degree of parallel execution during the build, and there is no guarantee that your virtual machine host is actually dedicating four cores to the VM just because the emulated machine reports that much available parallelism.

# Using Riker

We hope that reviewers can use the examples included in Riker's test suite to better understand the tool and experiment with its functionality.

Riker is not installed globally, so you should run the following command in any login session to add Riker to your PATH.

export PATH=$PATH:$HOME/riker/debug/bin

The Riker test suite contains a large number of functional tests that reviewers can use as examples. For example, the paper-example test (which resides in tests/paper-example) contains source files and a Rikerfile for the illustrative example from the paper's introduction. The file 01-build.t in the same directory is written for the cram test tool, and is meant to be human-readable; unindented lines are comments, indented lines starting with $ indicate commands to run, and all other indented lines are expected output. It should be possible to run through this example using the same commands that appear in the test file.

Nearly all of the tests are self-contained, with interesting Rikerfiles that cover important or surprising behavior. The hello, build-error, and skip tests illustrate basic functionality that goes beyond the example from the paper, and the tests that use pipes or manipulate soft and hard links yield some particularly surprising results.