# Frequently Asked Questions

## How is this different from Autotest, kselftest, and so on?

KUnit is a unit testing framework. Autotest, kselftest (and some others) are not.

A unit test is supposed to test a single unit of code in isolation and hence the name *unit test*. A unit test should be the finest granularity of testing and should allow all possible code paths to be tested in the code under test. This is only possible if the code under test is small and does not have any external dependencies outside of the test's control like hardware.

There are no testing frameworks currently available for the kernel that do not require installing the kernel on a test machine or in a virtual machine. All testing frameworks require tests to be written in userspace and run on the kernel under test. This is true for Autotest, kselftest, and some others, disqualifying any of them from being considered unit testing frameworks.

## Does KUnit support running on architectures other than UML?

Yes, mostly.

For the most part, the KUnit core framework (what we use to write the tests) can compile to any architecture. It compiles like just another part of the kernel and runs when the kernel boots, or when built as a module, when the module is loaded. However, there is infrastructure, like the KUnit Wrapper (`tools/testing/kunit/kunit.py`) that does not support other architectures.

In short, yes, you can run KUnit on other architectures, but it might require more work than using KUnit on UML.

For more information, see :ref:`kunit-on-non-uml`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\[linux-master][Documentation][dev-tools][kunit]faq.rst, line 39); *backlink***
>
> Unknown interpreted text role "ref".

## What is the difference between a unit test and other kinds of tests?

Most existing tests for the Linux kernel would be categorized as an integration test, or an end-to-end test.

- A unit test is supposed to test a single unit of code in isolation. A unit test should be the finest granularity of testing and, as such, allows all possible code paths to be tested in the code under test. This is only possible if the code under test is small and does not have any external dependencies outside of the test's control like hardware.
- An integration test tests the interaction between a minimal set of components, usually just two or three. For example, someone might write an integration test to test the interaction between a driver and a piece of hardware, or to test the interaction between the userspace libraries the kernel provides and the kernel itself. However, one of these tests would probably not test the entire kernel along with hardware interactions and interactions with the userspace.
- An end-to-end test usually tests the entire system from the perspective of the code under test. For example, someone might write an end-to-end test for the kernel by installing a production configuration of the kernel on production hardware with a production userspace and then trying to exercise some behavior that depends on interactions between the hardware, the kernel, and userspace.

## KUnit is not working, what should I do?

Unfortunately, there are a number of things which can break, but here are some things to try.

1. Run `./tools/testing/kunit/kunit.py run` with the `--raw_output` parameter. This might show details or error messages hidden by the kunit_tool parser.
2. Instead of running `kunit.py run`, try running `kunit.py config`, `kunit.py build`, and `kunit.py exec` independently. This can help track down where an issue is occurring. (If you think the parser is at fault, you can run it manually against `stdin` or a file with `kunit.py parse`.)
3. Running the UML kernel directly can often reveal issues or error messages, `kunit_tool` ignores. This should be as simple as running `./vmlinux` after building the UML kernel (for example, by using `kunit.py build`). Note that UML has some unusual requirements (such as the host having a tmpfs filesystem mounted), and has had issues in the past when built statically and the host has KASLR enabled. (On older host kernels, you may need to run `setarch `uname -m` -R ./vmlinux` to disable KASLR.)
4. Make sure the kernel .config has `CONFIG_KUNIT=y` and at least one test (e.g. `CONFIG_KUNIT_EXAMPLE_TEST=y`). kunit_tool will keep its .config around, so you can see what config was used after running `kunit.py run`. It also preserves any config changes you might make, so you can enable/disable things with `make ARCH=um menuconfig` or similar, and then re-run kunit_tool.
5. Try to run `make ARCH=um defconfig` before running `kunit.py run`. This may help clean up any residual config items

which could be causing problems.

6. Finally, try running KUnit outside UML. KUnit and KUnit tests can be built into any kernel, or can be built as a module and loaded at runtime. Doing so should allow you to determine if UML is causing the issue you're seeing. When tests are built-in, they will execute when the kernel boots, and modules will automatically execute associated tests when loaded. Test results can be collected from `/sys/kernel/debug/kunit/<test suite>/results`, and can be parsed with `kunit.py parse`. For more details, see "KUnit on non-UML architectures" in Documentation/dev-tools/kunit/usage.rst.

If none of the above tricks help, you are always welcome to email any issues to kunit-dev@googlegroups.com.