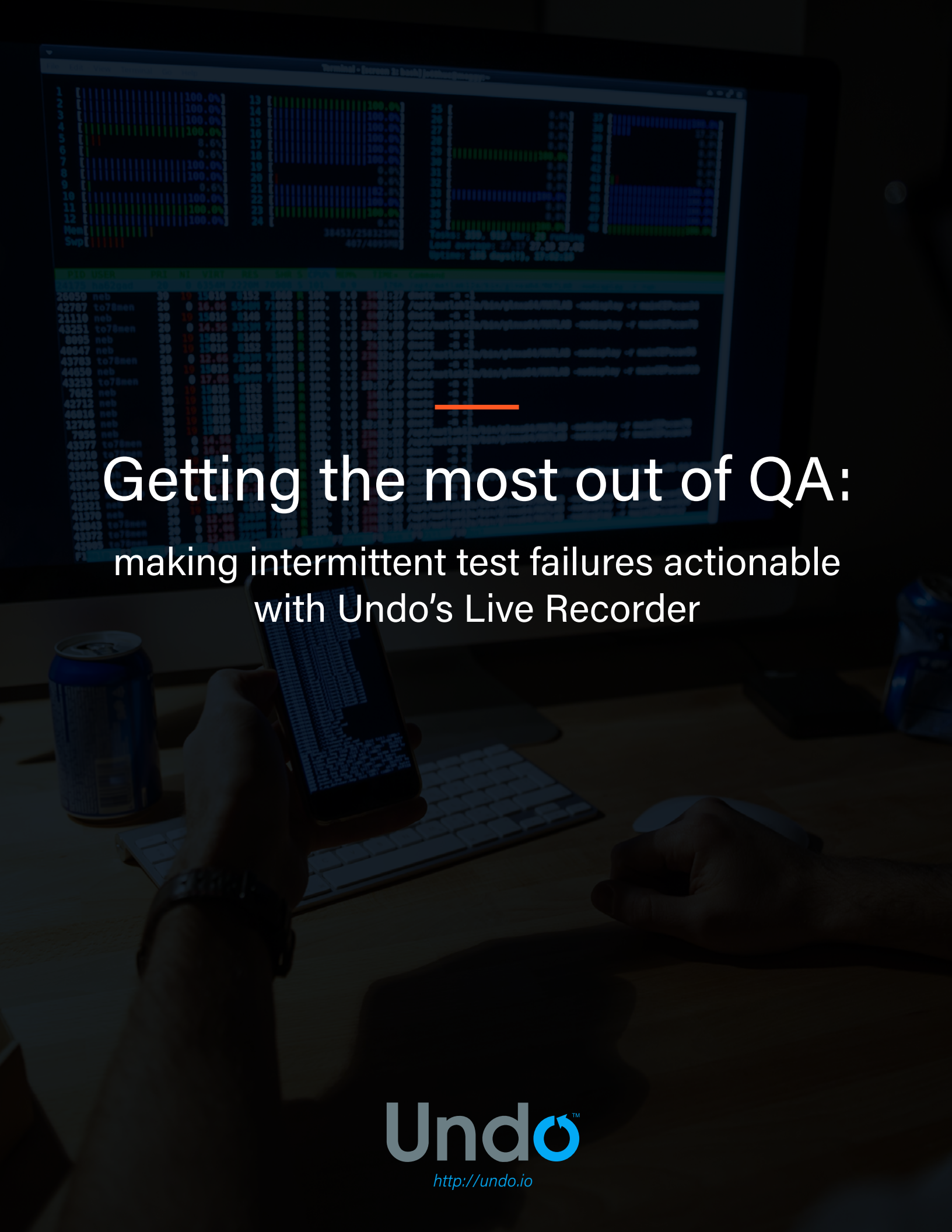


Getting the most out of QA:

making intermittent test failures actionable with Undo's Live Recorder

Undo™

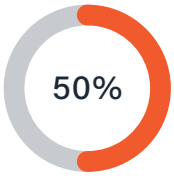
http://undo.io

Introduction

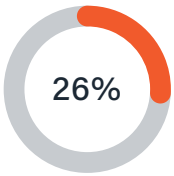
We live in a world in which software is increasingly dominant. Not only do we rely ever more on software, but its complexity is becoming even greater. Projects for which it is feasible to manually test the whole feature set for each release are rare; even when it is possible, doing so is generally wasteful of engineers' time, and manual testing is itself error prone.

With greater complexity, bugs are more likely to be intermittent with failures difficult to reproduce. Sometimes this non-determinism results in failures never getting fixed. Any unfixed bug is potentially a security breach or catastrophic customer outage waiting to happen.

To cope with this complexity whilst still providing some assurance of delivered software quality, the software industry has embraced a variety of automated testing regimes. A development team of a given size is able to run many more tests than it could before and must triage, and attempt to resolve, a correspondingly larger number of test failures. Diagnosing the cause of each test failure can still take a lot of time, particularly if the problem is intermittent or otherwise not easily reproducible.



For these reasons, developers spend on average 50% of their programming time debugging, not developing[1].

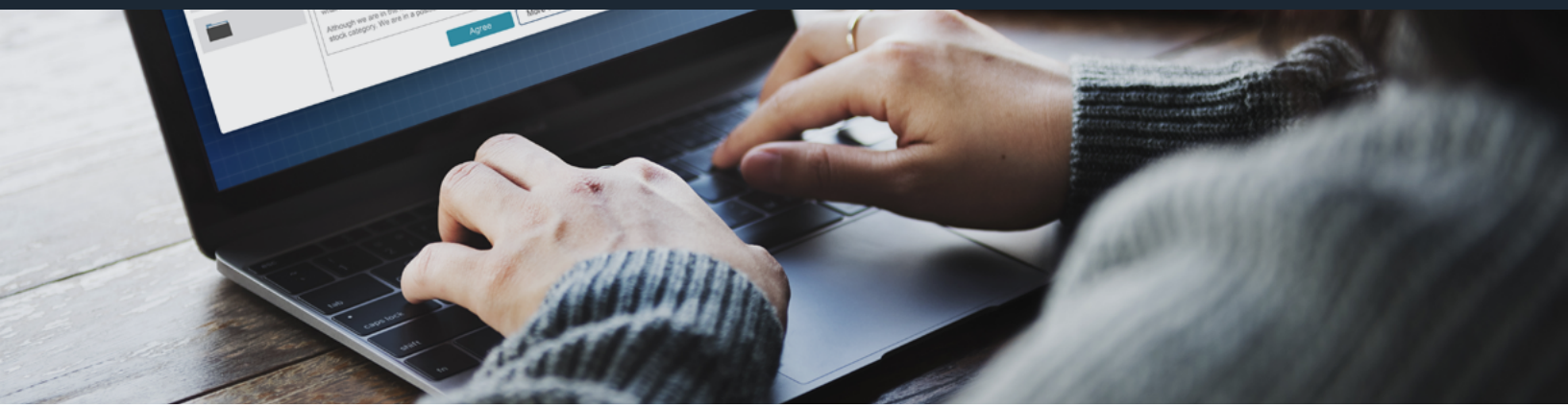


Reversible debuggers are proven to reduce debugging time by 26% [2].

Live Recorder bridges the gap between identifying a failure in the Quality Assurance (QA) environment and creating a reliable reproducer in the developer's environment by generating a recording. The recording harnesses the power of reversible debugging to give the developer the power of going backwards through a program's execution from the symptom of a failure to its root cause, reducing time and effort to diagnose and fix failures.

This Whitepaper will briefly review modern approaches to testing, with a particular emphasis on simplifying interactions between QA and Development teams. It will then explore how Undo's Live Recorder is allowing teams to optimise their existing test systems in addition to capturing and analysing failures using record, rewind and replay.

[1] Judge Business School (2013) - Research by Cambridge MBAs finds 49.9% programming time spent debugging
[2] Judge Business School (2013)



Goals of testing

The ultimate goal of all software product testing is to minimise the issues encountered by users of the software after it is released. The approaches to achieve this objective vary, but the one thing they all have in common is that they identify a failure mode of the software under test - i.e. it did something it should not have done - which is then typically reported back to Development to investigate and resolve.

The relative cheapness of machine time versus human time means that the majority of tests run by modern software teams are automated. Even when testing a new feature, tests aim to be scriptable (as far as possible) and therefore repeatable - with as little interaction as possible. When the new feature is fully tested and accepted into the product, the accompanying scriptable tests are accepted into the product's automated test suite in order to catch regressions caused by further development. The test suite also accumulates tests created in response to bugs that arise in other forms of testing, such as exploratory, ad hoc[3] or fuzz[4], or even in customer deployments. The total library of tests thus consists of individual tests with a broad spectrum of motivations and methodologies.

```
class FastSqroot
{
public:
    FastSqroot(size_t cachesize) :
            cachesize(cachesize)
    {}
    int operator()(int number);

private:
    const size_t cachesize = 100;
    std::deque<std::pair<unsigned char, unsigned char> > data;
};

int FastSqroot::operator()(int number)
{
    auto f = std::find_if(data.begin(), data.end(),
                          [&number] (const std::pair<unsigned char, unsigned char> &p)
                          {
                              return p.first == number;
                          });

    if (f != data.end())
    {
        return f->second;
    }

    /* Cache miss. Find correct result and populate a few cache entries. */
    int sqroot = 0;
    for (int number2=number-1; number2 < number+1; ++number2)
    {
        int sqroot2 = (int)sqrt(number2);
        data.push_back(std::make_pair(number2, sqroot2));
        while(data.size() > cachesize)
```

However, minimising potential production issues may not be at the forefront of QA's mind on a daily basis as testers may be inclined to adopt a more narrow focus on identifying the existence of bugs and reporting them to Development for fixing. This is especially likely to be true in highly functional organisations where a formal division between QA and Development can reinforce demarcation between teams. "I've done my bit, over to you".

[3]Both exploratory and ad hoc testing grant the tester degrees of license in deciding how and what they do at any given time. They are by their very nature unscripted and therefore difficult to reproduce. However precisely the tester's actions may have been noted along the way, the timing at least will differ; this could be a critical element in reproducing the failure.

[4]Fuzz testing consists of testing with an element of randomization in a bid to evoke unexpected code paths.

Test deliverables

Developers and testers are all aware of the deliverables in one direction: Development deliver the untested product to QA, who deliver the tested product. However, there is also an ongoing communication from QA back to Development about bugs found. This is a tacit deliverable in the opposite direction.

A QA department can be viewed as a "friendly customer" that uses **all of your product's features**. They are willing to help but...

- Sometimes they have conflicting priorities *(e.g. maintaining test infrastructure, or testing other features).*
- It may not be easy to give access to that "special" environment for a particular repro.
- Can't always pinpoint a ready reproducer *(e.g. intermittent failure, fuzz testing, or unscripted testing).*
- Silo mentality can creep in.

Tester says:

I've determined that there is a bug, it's Development's job from here on.

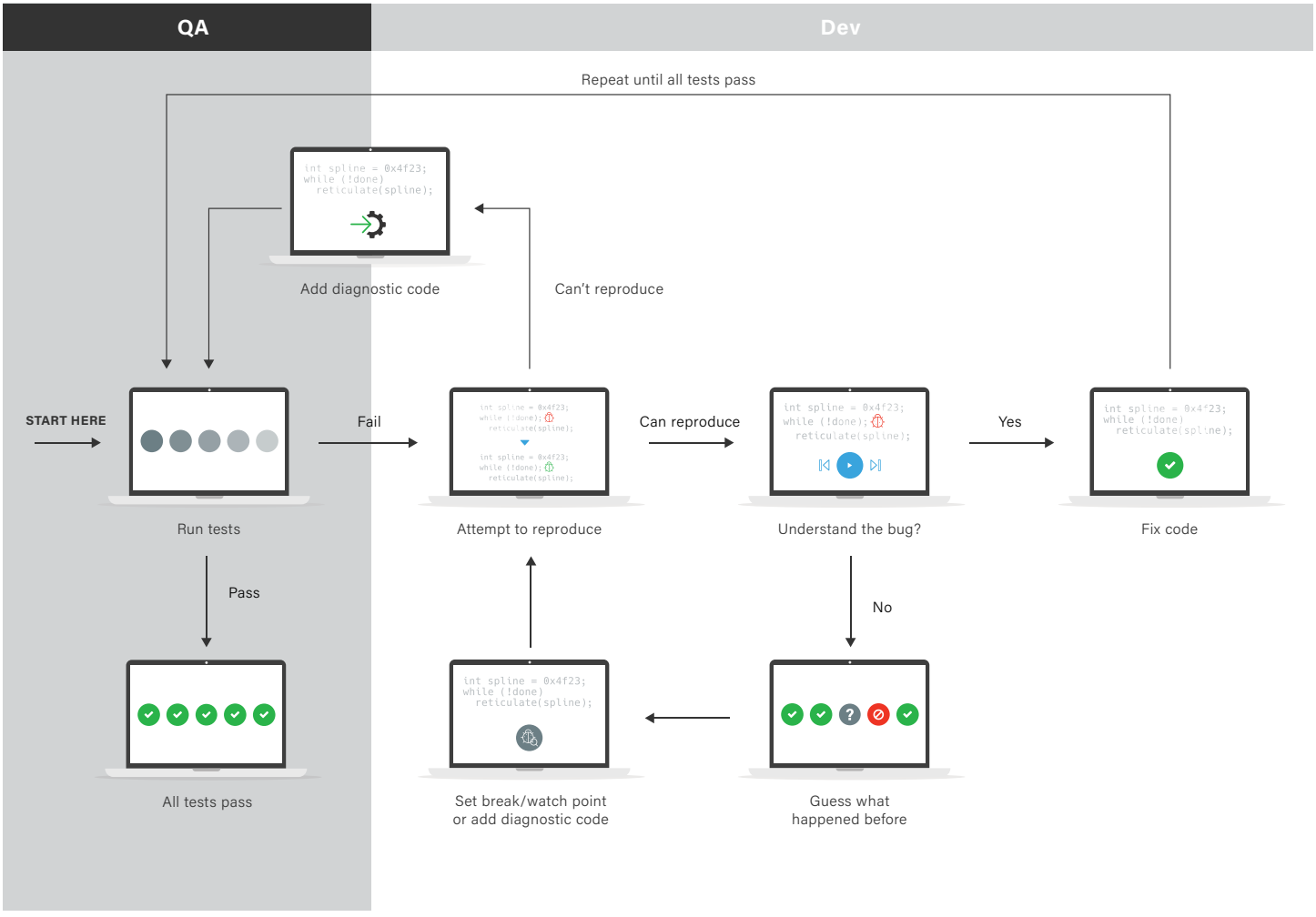
Developer replies:

Works for me, I'm not going to investigate until QA gives me more info on how to reproduce.

The effectiveness of the delivery channel back to Development hinges on the quality of the information about the failure.

Traditional QA

Conventional test systems inform that there is a bug, but offer limited information about the cause. A good QA department will endeavour to include standard information in bug reports to Development such as running environment, standard diagnostics etc.



This flow diagram illustrates a typical interaction between QA and Development teams when a test fails.

Workflows such as Continuous Integration[5] offer a further clue in the form of which commit to the codebase caused the failure. However, there are several cases where CI cannot directly help identify problematic code changes. For example, the triggering code change may be entirely correct, yet cause a pre-existing bug to manifest. The developer cannot simply stare at the modification in the expectation that the bug will eventually reveal itself. Conversely, the failure may not easily reproduce for the developer even running the same test, or a test may start failing for some external/environmental reason. In short, CI is often very good at making the easier bugs very easy to fix, but it typically is of little help with the more difficult bugs, particularly intermittent ones.

When a failure occurs it must be triaged. This is no easy process, requiring the balancing of conflicting tensions such as the desire to deliver a bug-free experience to customers and pressure to deliver on-time. The following considerations must be taken into account:

→ Maybe the failure doesn't represent a genuine bug.

- Has the functionality changed as a result of a new feature such that the product's behaviour is intended to have changed, invalidating the test? If so, must the test be retired, or should it be modified to test the new behaviour?
- Is the failure a false positive, caused by testing infrastructure or non-determinism in the test itself?

→ What is the effort of fixing the bug versus the impact of leaving it in?

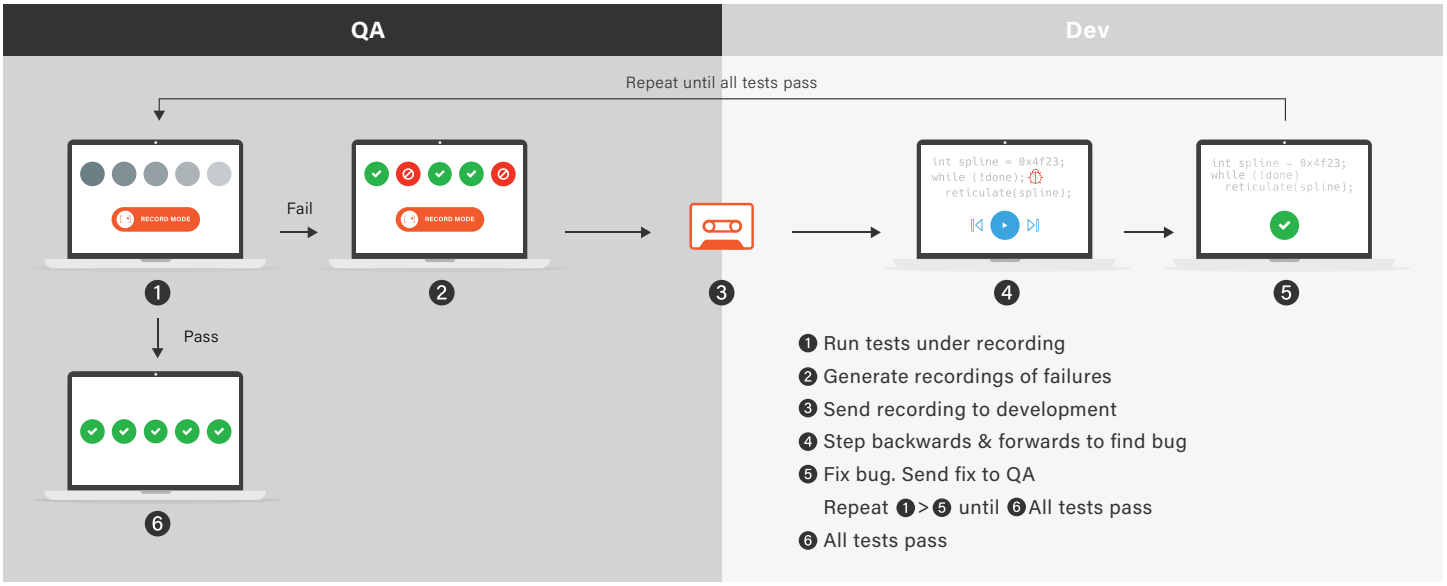
- Some bugs only occur under extremely rare conditions, and may be prohibitively expensive to fix. If the implication to the customer of hitting this bug is deemed to be low, then the bug will typically be logged, but remain unsolved. This is a difficult judgement to make. Any bug may potentially be exploited as a security vulnerability or result in a catastrophic system outage for the customer. At this point it is much harder to resolve than when detected in the test phase.
- Even when a failure must be fixed because it is assessed as high impact, it may have such an esoteric nature that it is impossible to reproduce other than under the conditions of the test environment. In this eventuality, many iterations may be required of a developer adding speculative checks and diagnostics to determine enough information about the root cause to either diagnose or construct a more reliable reproducer.

The developer's starting point, therefore, is often to repeat the failing test on their own machine in a bid to reproduce the same failure. If the test case is simple, and the failure happens reliably, this may be good enough for the developer to diagnose the root cause. If the test case has a long run-time, or the failure is intermittent, then the developer has a choice: they can accept the overhead of a lengthy delay between each reproduction of the failure, or they can attempt to construct a simpler reproducer. The latter may be difficult without first understanding the cause: a typical Catch-22 in the world of software!

[5]Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

QA with Live Recorder

Live Recorder for Automated Test allows a tester to automatically capture test failures as they happen, including everything about the process's execution. The companion UndoDB tool allows a developer to step line by line or run forwards or backwards through the recording to have total visibility into the entire program state at any point in its execution. They can zero in and diagnose the root cause of the failure using breakpoints and watchpoints, forwards and in reverse.



From failure to resolution: how Live Recorder works.

With Live Recorder deployed in the test environment, the deliverable from QA back to Development now includes a recording of each observed failure. A developer can analyse the recording to see what the software did during the failed test. They neither need to reproduce the failure in another environment, nor have access to the original test setup.

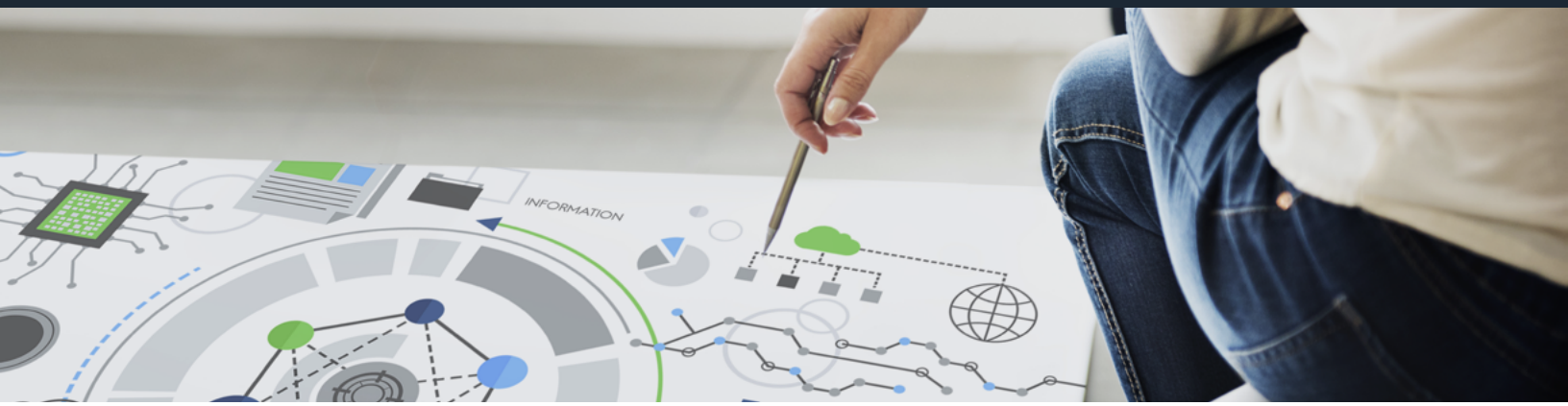
As a consequence of removing the need to reproduce the failure, there is no longer the potential for many cycles of the developer adding test/diagnostic code and requesting the tester to rerun the test. In the case of recorded manual testing, diagnosis of the bug is no longer critically dependent on the tester being able precisely to redo their actions leading up to the failure.

Recordings can be shared between multiple developers, possibly in geographically diverse locations. This makes it easy for experts in different areas of the product to investigate their components as the diagnosis unfolds. Each developer is looking at the exact same execution code path, rather than attempting to independently reproduce something close enough for meaningful discourse.

The ability to analyse a recording also provides all of the benefits of reversible debugging using the UndoDB debugger[6]. The debugger allows developers to see exactly what the program really did, with unlimited precision and detail - inspect any memory, variable or register at any moment in the program's execution (at the granularity of a machine instruction if desired).

Most test failures become much easier, and therefore quicker, to solve. Many failures that were essentially impossible to address, become addressable. As a result, product quality is improved. Having a recording of the failure breaks the interdependence between understanding and reproduction, eliminating communication barriers between and within teams.

[6]Undo - Reversible Debugging Whitepaper / Whitepaper

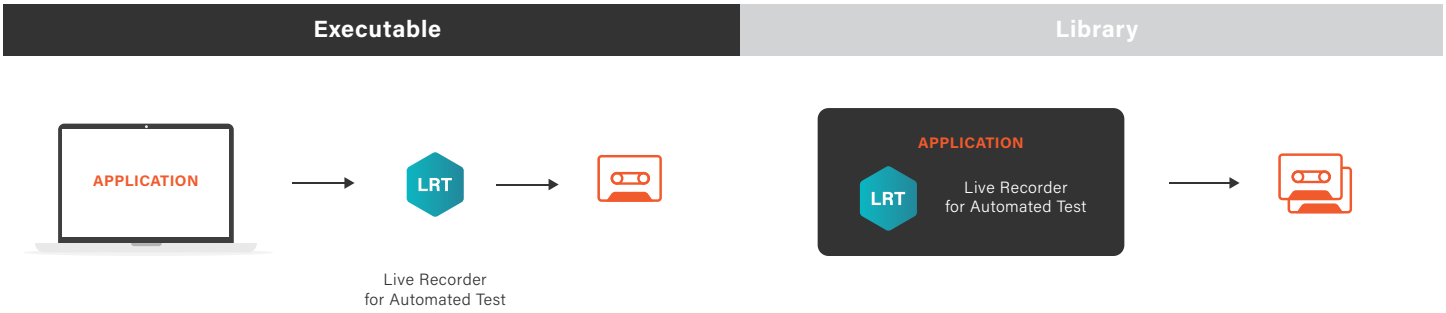


Deploying Live Recorder

Integrating Live Recorder to capture tests

Live Recorder can be used in one of two ways:

- **Executable:** the *live-record* command-line tool is used to wrap the software under test, providing deterministic recording of unmodified code.
- **Library:** the software under test is linked to the *libundolr library*. Minimal changes to the application provide the ability to control Live Recorder.

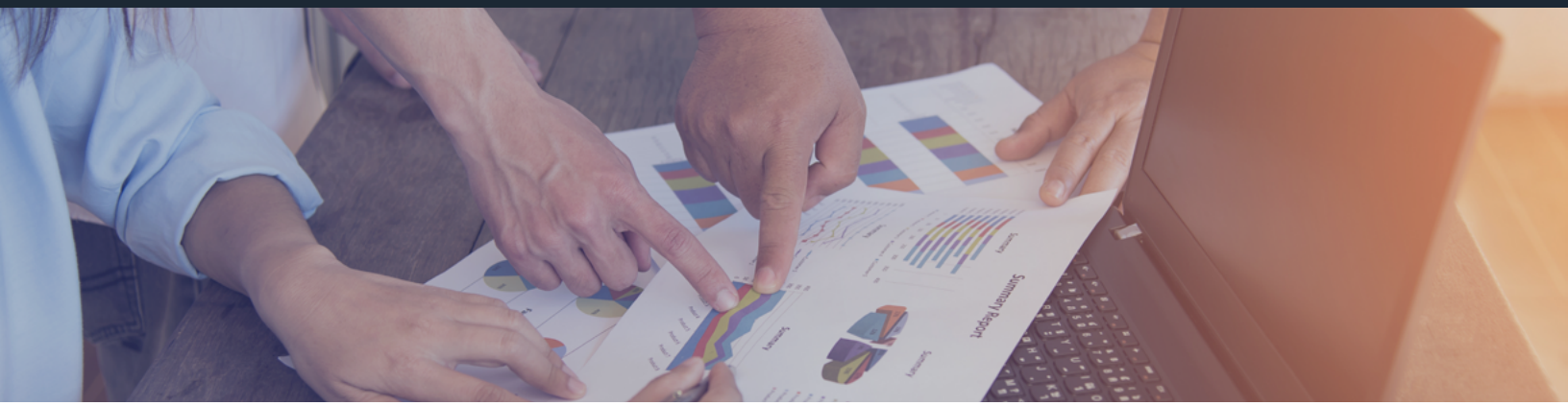


The executable is simplest to deploy. It avoids the need to modify the application's source code and build dependencies; instead the tool is installed on each machine in the test farm.

The test infrastructure needs a small modification to run the test binaries under *live-record*, generating a recording for the entire execution of each test process.

The Library solution is best suited to more demanding deployments, since it permits the most flexible control of recording behaviour. The source code modifications required to use the API are very small and easily integrated within existing configuration interfaces; an additional library dependency on *libundolr* is introduced.

The process itself controls which part, or even multiple parts, of its execution is recorded. For example, preliminary processing that is not directly relevant to the test need not be recorded; a single process which runs many tests can produce one recording for each test.



Planning the scope of test capture

When deploying Live Recorder in test, Undo suggests two complementary approaches.

- **Always-active:** Routinely use Live Recorder to record whole classes of test. This guarantees that any failures will have recordings available but it does impose some slowdown. It's faster to fix a test that has a recording but tests will take longer to run. The slowdown is generally acceptable in the context of over-night tests, for example, but may be problematic for tests that are designed to give rapid feedback to developers.

- **Targeted:** Use Live Recorder to record tests of specific interest. These may be statically chosen, or Live Recorder can be activated automatically when a test is re-run after failure. This approach has minimal effect on test run-time (zero effect for passing tests) but might miss a very rare opportunity to record an intermittent failure.

A typical deployment will employ both approaches, sometimes within a single class of test. For example, SAP, one of the world's leading technology companies, has incorporated Live Recorder into its fuzz testing of SAP HANA[7]. The SAP HANA team is using Live Recorder always-active on a subset of their fuzz testing server farm. This is allowing SAP to phase in recording in parallel to their existing workflow, with their previous testing routine continuing as normal. The recordings allow them to quickly address issues identified by their fuzz testing, leading to ever greater product quality[8].

Summary

Live Recorder for Automated Test allows QA teams to deliver a recording that allows Development to perform a forensic analysis of the failure. It is no longer necessary to convey a lengthy description of environment/ test setup.

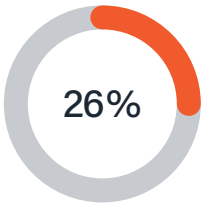
This results in much cleaner communications between QA and Development, with far fewer exchanges required on average for each reported test failure.

One or more developers analyse the exact failure seen in test. Removal of the need to reproduce each failure and the ability to run backwards as well as forwards through recorded program execution leads to rapid diagnosis of previously undiagnosable bugs.

[7]HANA is a scalable, massively multi-threaded database that forms the foundation of the SAP technology stack and is thus the backbone of major businesses worldwide.
[8]Undo - Improving Software Quality in SAP HANA using Live Recorder / Case Study

About Undo

Undo's products are used by thousands of developers to solve complex, real-world problems for leading technology companies, from embedded to enterprise and High Performance Computing to banking. Its unique record, rewind and replay technology enables Linux and Android developers to see exactly what their program did at every step in its execution.



Developers can now rapidly respond to failures in production and test environments, increase their development productivity by at least 26% and dramatically improve software quality.

Undo is a privately held company headquartered in Cambridge, UK. It was one of the winners of the 2016 London Accenture FinTech Innovation Lab and its technology has won numerous awards including Best Software Product at the ARM Innovation Challenge and Gartner's Cool Vendor in Application Development. For more information, see http://undo.io or follow us on Twitter at twitter.com/@undosoft.

Undo

9 Signet Court
Swann's Road
Cambridge, UK
CB5 8LA

http://undo.io

linkedin.com/undoltd

twitter.com/undosoft