diffblue
AI for Code

White paper

The Role of AI-augmented Coding in Accelerating Application Modernization

Driving faster, safer transformation with software that can write software

Introduction

Application modernization is on the agenda of many IT leaders today, and with good reason: as software becomes an ever more integral driver of business success, more modern approaches to design, architecture and operation yield tangible benefits.

A risky business

But despite the considerable upside, application modernization is also time-consuming, risky and comes with a high opportunity cost: developers working on modernization have less time to spend building new capabilities. The risk of change is particularly high where the code is fragile, hard to understand, poorly documented and poorly tested - unfortunately exactly the state of many enterprise applications in use today.

The good news is that cutting-edge AI-augmented coding tools can now autonomously write entire unit test suites that significantly reduce developer effort, help with code understanding and reduce risk by catching regressions as the app is updated.

What is application modernization?

The term ‘application modernization’ covers a range of scenarios, including breaking down legacy code monoliths, refactoring for API-driven microservices, moving to a cloud-native architecture, or upgrading to a new language version and the latest functionality. Some teams may just see it as a process of improving how they work through adoption of automation and CI/CD. Whatever the context, application modernization is an important goal for many enterprises.

But It Was Working Yesterday!

One of the key pain points in application modernization is avoiding regressions or “breaking” the current functionality of the code. This problem is compounded by the length and complexity of integration and functional testing cycles – breakages are often not found until late in the process, and it is slow, difficult work to triage bugs and identify the root cause.

Unit testing offers a way to spot regressions much earlier in development, at the time the code is being worked on by the developer. But too many codebases lack good unit test coverage. Writing enough unit tests can look like an insurmountable mountain to climb in even moderately-sized codebases because of the sheer volume required to get good coverage. Large legacy applications – with perhaps hundreds of thousands, or even millions, of lines of code – present a truly daunting task.

Writing a suite of unit tests for an existing code base means the developer has to study the code to try to identify what it does, and then write a set of tests to an unwritten specification – often without knowing boundary conditions and subtle-but-important behaviors. To isolate the unit test and ensure it runs quickly, realistic mocking has to be implemented too.

The net: how the code behaves today becomes the specification for the unit tests. However, that’s not all: negative tests also need to be written, and these are harder simply because it’s more difficult for us humans to think about failing cases. All these things mean that writing useful unit tests can be a formidable challenge.

Diffblue’s “rule of thumb” based on experience with customer codebases and open source projects is that you need one unit test for every 15-18 lines of code – so a 15,000 line project might need 1000 tests. If it takes 15 minutes on average to write, check and finalize a unit test, that’s 250 hours of work – and then the tests have to be updated and re-checked as the code evolves.

Software Can Now Write Software

AI-powered coding tools are changing the narrative for large scale, tedious, error-prone software development like unit testing. The essential idea is that developers don't have to write 100% of software by themselves. Past efforts at auto-generated coding were crude, mechanical, simplistic and developers hated them. But technology that started out in world-leading AI research groups can now write code to the same standard as a human developer.

There are two main AI code-writing techniques today: large language predictive models trained on open source codebases, used by tools like GitHub Copilot, and reinforcement-learning algorithms like Diffblue Cover that use the same approach as game-winning AI.

These techniques have very different goals, however. Large language models produce fragments of code for human developers to review, using existing code and/or comments as a "prompt" to predict the code that follows. Since these models can't evaluate the code they produce – only a human can do that - they typically offer several completion options for a developer to choose from.

This approach is particularly useful for coding tasks that are time-consuming and exacting, but not especially complicated. Examples include "boilerplate" code such as a Class definition in Java, or calling an unfamiliar API (do you remember all the S3 bucket options?). The advantage is that offering these chunks of code in a few seconds is a lot faster than typing it yourself, visiting StackOverflow or Googling to find some example code and adapting it.

Large language models are an exciting new development, but come with some disadvantages. It's incumbent on the developer to check the predicted code is logically correct and reject erroneous completions that don't satisfy the required intent – and perhaps even more crucially, to spot errors and potential vulnerabilities. As the name suggests, large language models work on word associations and context – they don't code for logic and arithmetic, and it's easy to prompt them into making arithmetic errors.

There's an important reason why GitHub describes this process as automated pair programming: using the tool interactively and with a critical eye is important – it won't work autonomously.

Complicating matters further, the 175 billion parameter predictive model behind Copilot means your code has to be sent to GitHub's Codex cloud service for processing - that's what allows it to quickly return a completion. For many enterprises, sending any code to a 3rd party is an unacceptable security risk, though there are alternatives like TabNine that offer locally-hosted models instead.

Autonomous Unit Test Writing

In contrast to large language models Diffblue Cover uses reinforcement learning, from the unsupervised learning branch of the AI tree, to autonomously write unit tests. Instead of being pre-trained on a large amount of open source code, the learning happens in real time as the model searches the space of possible test programs, following promising trajectories and finding the best possible solution in the time available.

Diffblue Cover writes each unit test by evaluating existing code at a method level and guessing what a good test would be. It then runs the proposed test against the method, evaluates the coverage (how much of the code the test exercises) and other qualities of the test, and then predicts which changes to the test will trigger additional branches to produce higher coverage. Then it repeats these steps until it has found the best test(s) in the time available (around a second).

This is known as “probabilistic search”: a technique where the space of potential solutions is sampled, and the algorithm spends more time searching regions with a greater probability of a good solution.

The return value(s) of the search, and any side effects that were observed, are used to write assertions on a method’s behavior – because it’s no good having good code coverage if the tests don’t check what the code did. The result is a test that is known to work (Cover ran it) and produces specific coverage and assertion checks.

By repeating this for all methods in the code, Diffblue Cover autonomously produces an entire suite of unit tests that reflect the current behavior of the program – just what we know we need for an application modernization project. Any failures of these Diffblue tests identify changes in behavior of the code and a risk of potential regressions. Where the change is intentional and correct, the developer can immediately get replacement tests from Cover without re-creating the entire suite by asking the tool to rewrite only the tests affected by the code diff.

Because Cover knows which tests will reach the code that is being changed, it can also increase the speed and reduce the cost of CI by running just the tests that matter for that PR. Faster feedback from CI runs delivers a big productivity gain: developers can fit more debug cycles in a day, which means faster project completion.

Probabilistic searches cannot guarantee they will find the best solution – or any solution – because by design they don’t evaluate every possible case. But you don’t get gibberish or an incorrect result in that situation: you know when you’ve failed! This makes reinforcement learning a more suitable approach than large language models for fully automated test-writing because it won’t write tests that give you false confidence or cause your pipeline to fail.

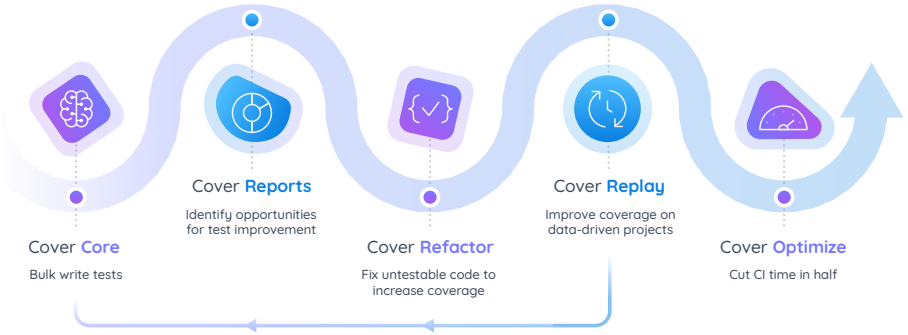
Extending the Power of Unit Testing in Application Modernization

In application modernization, Diffblue Cover can use existing code to write an entire suite of unit tests for very large applications – or large groups of projects – in a matter of hours. Cover Reports can then visualize the current state of unit testing across all projects. It shows you the coverage written by Diffblue, coverage from any human-written unit tests, where there is overlap, which code isn't being tested, and what code is untestable and high complexity. Cover Reports allows you to drill down from a top-level overview to individual Java classes to ensure that development teams can focus effort on high risk areas.

The untestable code highlighted by Cover Reports is a common problem. Two main causes are an inability to call internal methods in a class, and the lack of observer methods that allow tests to make assertions on the state changes made by the code. To solve this problem Diffblue has introduced Refactor, which can determine when observability problems occur and automatically add the missing code to the classes via autonomous refactoring. The project can then be recompiled and Cover re-run to write more tests.

The coverage challenge presented by data-driven applications, where the flow of control and data structures are not known until runtime, presents a further source of risk during modernization. For example, at one investment bank Diffblue came across an application where the definition of the financial transaction being processed is fed into the application at runtime. Typically these applications are tested via integration and functional tests with hand-crafted synthetic data.

The challenge when writing unit tests for such behavior is that there's little in the program to use to write useful tests – all the interesting stuff happens at runtime. A feature of Diffblue Cover called Replay solves for this case. A small agent records data flowing through an application while it is running integration or functional tests; the recording can then be used by Cover to automatically convert those slow functional tests into fast unit tests that run much earlier in the development process.



Java application modernization projects might last months or even years. Diffblue Cover provides a range of AI-powered features that support the fast, effective unit testing essential to success.

Summary

Unit tests are vital to the success of application modernization projects – they're the most fundamental way of ensuring that what works today will work tomorrow, whether that's in new infrastructure, a new application architecture or a new language version.

But achieving the unit test coverage you need can seem like a mountain to climb. A significant amount of developer effort is needed not only to write the tests, but to work out what they should do in the first place.

Diffblue Cover is a tool for AI-augmented development that autonomously writes unit tests for Java applications and can produce useful coverage in a matter of hours. Cover's reporting dashboard allows team leaders to see where they are in terms of coverage and risk, and prioritize work to address coverage gaps and high risk code. Cover Refactor can autonomously refactor code to make it more testable. Cover Optimize dramatically reduces CI cycle times, providing developers with much faster feedback. Cover Replay can shift left functional tests and incorporate them into your unit test suite.

To learn more about Diffblue Cover visit www.diffblue.com/products

About Us

Diffblue is the leading pioneer of software creation through the power of AI. Founded by researchers from the University of Oxford, Diffblue Cover uses AI for Code to solve the problem of effective unit testing. Capable of writing unit tests 250x faster than a human developer, Cover helps software teams improve code quality, expand test coverage and increase productivity, so they can ship software faster, more frequently, with fewer defects.

To find out more visit diffblue.com