Taming the Variants
Multi-Architecture Continuous Testing at Google

Tim A. D. Henderson*, Sushmita Azad†, Chandrakanth Chittappa, Ali Esmaeeli, Laura Macaddino,
Sam Manfreda, David Margolin, Dharma Naidu, Sabuj Pattanayek, Sachin Sable, Ruslan Sakevych,
Dushyant Acharya, Adrian Berding, Kevin Crossan, Wolff Dobson, Avi Kondareddy, and Abhayendra Singh
Google LLC
1600 Amphitheatre Pkwy
Mountain View, California, USA
(corresponding) *tadh@google.com, †sushazad@google.com

***Abstract*—Enterprises are increasingly adopting multiple general-purpose computer architectures in the data center. This leads to new testing challenges as it creates demand to qualify the software for the additional architectures. Naively double-testing all software for both architectures is costly and unnecessary. Further, reconfiguring CI/CD to take advantage of the new architecture can be non-trivial at scale. This paper introduces CI/CD *variants* and an optimized testing cycle to solve these twin challenges. We empirically evaluate our solution's impact on human and machine expenses using 44k projects at Google on real production data. First, we estimate saving $\sim$25% of machine expenses at the negligible cost of a few delayed breakage detections per day. Second, we estimate a 90+% reduction in human cost for migrating the configuration. All features described in this paper are now Generally Available at Google and we report this as an empirical case study in scaling CI/CD to new architectures.**

I. Introduction

Over the last decade, hyperscalers have introduced multiple general-purpose architectures into the data center [1]. At Google, this creates challenges for our very large scale continuous integration and delivery platforms (CI/CD) [2]. We aim to enable adoption by ensuring the software is appropriately built and tested. We face two challenges.

First, a high performance Continuous Integration [3] system identifies novel build and test failures the moment they are introduced to the source repository. This enables developers and automation to find and fix these failures as soon as possible. The sooner a failure is found, the cheaper it is to fix. For instance, a bug-introducing change can be automatically reverted without review, provided there haven't been further changes that would cause merge conflicts [4], [5]. Finding failures as quickly as possible is in tension with running tests as cheaply as possible (achieved by adjusting testing frequency). Can we add the new architecture without constantly retesting on both platforms?

Second, when a new compute architecture is introduced, all projects need to determine:

(1) Is the project's code build-compatible?
(2) Do the project's tests pass on the new architecture?
(3) Is the project's runtime performance acceptable?

Answering these questions requires modifying the CI/CD configuration to additionally build, test, and deploy the code to the new architecture(s). This mechanical, unrewarding task can be a costly undertaking for an organization. Prior to the start of this project we estimate that it took an *expert* from the central Arm migration team (whose primary job was to perform this qualification) hours of hands-on time just to make the required configuration changes to a single project. Google operates a monorepo with billions of lines of source code and over 100,000 binaries. Making manual changes to every project was impractical [1].

The two challenges are intertwined. We neither want to double test nor do we want our users to manually edit configuration files. Further, manual changes create bespoke additional CI/CD configurations requiring future maintenance. While guides exist, users don't always follow instructions. This means that the CI/CD systems would need complex logic to link functionally-equivalent x86 and Arm configurations.

Without reliably linking configurations, the system cannot optimize the testing process. We solved both problems by introducing *variants*: automatically *re*configured user projects. A variant adapts the original configuration for each targeted platform. This flexible infrastructure also supports variants for compiler optimization level, address and memory sanitization, and other build modifications.

Because the variants are automatically configured, they are also *structured*. This allows our systems to make consistent choices on how building and testing should occur for each variant. Our primary CI system, the Test Automation Platform (TAP) [2], [6], [4], [5], [7], introduced a new testing mode: *Target Comprehensive Cycles*, optimizing the cost of testing across multiple architectures. Our CD system also adopted Target Comprehensive cycles to realize the cost savings. However, to prevent bugs escaping to production, it introduced *Late Stage Testing*. This phase executes the remaining tests prior to production, but at a lower frequency.

A. Contributions

We detail our solution, *Variants*, and empirically analyze its efficacy and performance. We measure the realized human migration savings from variants and the machine costs avoided by *Target Comprehensive* (TC) testing. Finally, because opti-

mizations involve a trade-off, we examine the productivity cost of delaying the detection of bugs by skipping tests.

(1) *Variants* are generally available and used by 44k projects. We present a large-scale empirical case study on the efficacy of our methods using production data. We estimated both the human and machine savings and the productivity cost of delayed breakage detection.
(2) A new *Target Comprehensive* (TC) scheduling mode prevents doubled testing costs. We measured savings of approximately $\sim$25% over the naive double testing strategy.
(3) *Late-Stage Testing* (LST) shifts test load into later, less frequent, parts of the CI/CD pipeline. This executes test configurations skipped by TC prior to user release. Late-stage testing failures occur less than 0.1% of the time.
(4) Automated configuration for building and testing for alternative architectures reduces migration effort by approximately 90%.

II. Background

At Google, developers use multiple tools as part of their CI/CD system. Testing is typically handled by TAP (the Test Automation Platform), which performs testing both prior to code being submitted (presubmit testing) and after code is submitted to the mainline branch of the mono-repository. TAP only handles tests that fit on a single machine and are *version hermetic* (i.e. only use code from a single version and do not make network calls). Some teams need larger integration tests that need multiple machines or need to compare behavior between the test environment or a production environment. These teams additionally use an Integration Testing platform. While our work on variants is also used by the Integration Testing platform, we constrain the focus of this paper to TAP.

TAP runs *testing cycles*, where all projects and their *Bazel targets* are batched together at the same version [4], [7]. Once all the targets for a project have executed, TAP computes a *Project Status* (ex. `PASSING`, `FAILING`, ...). A `PASSING` status enables the next service in the CI/CD system to take over: Rapid.

Rapid is a generalized CI/CD workflow execution engine. It is similar in capability to commercial and open source systems such as GitHub Actions or Jenkins. It allows developers to orchestrate a complex series of processes to build production packages, trigger integration tests, and trigger production deployments. While Rapid's capabilities are very general, all Rapid "releases" start with a *Candidate Creation* phase, during which testing results from TAP and other systems are verified before the production binary package is created. A developer may configure some tests to only trigger in Rapid instead of the more common and recommended method of using TAP.

Due to the general complexity of software development at Google, we have a managed platform that is built on top of all of this called *Urfin*. It provides a streamlined intent-based mechanism for developers to configure their projects. Urfin standardizes the workflows that Rapid executes and the production environments. When a project is configured by Urfin (instead of "Rapid Native"), we are able to automatically determine what workflows modify production environments (versus those used for additional verification or validation with non-production data or users). This structural distinction is relevant for our Late-Stage Testing strategy and empirical evaluation, motivating our introduction of these terms.

A. Glossary of Google Terminology

Piper – The version control system used at Google [8].
Changelist (CL) – A commit in Piper. CLs have integer numbers and submitted CLs have monotonically increasing numbers.
Bazel – The build tool used at Google: https://bazel.build/ [9].
Target – The compilation unit in Bazel. TAP doesn't run individual test methods or even test classes / suites. Instead, it runs Bazel test targets that may contain many test methods and classes.
Forge – The large remote build execution cluster used by Bazel [9].
Build System – A collection of services for batching and running builds in production [9], [10], [11].
TAP – The primary continuous integration platform [12], [13], [14], [15], [16], [2], [17], [6], [4], [5], [18], [7].
Rapid – The primary workflow automation tool use to orchestrate release processes [9].
Urfin – A managed platform built on top of tools like TAP and Rapid that dramatically simplifies configuration and usage of release automation.
Rapid Native – Our term for release projects that don't use the managed platform (Urfin). These projects have non-standard workflows and configurations.
Blueprint File – The common configuration file format used by TAP, Rapid, and many other tools. It is defined in a bespoke configuration language (NCL). See Listing 1 for an example.

B. Configuring TAP and Rapid

TAP, Rapid, and other tools are configured in *Blueprint files* (see example in Listing 1). Blueprints are written in a bespoke, proprietary configuration language (NCL). NCL is a high-level functional language for describing and manipulating protocol buffers. It supports complex data types, functions, and libraries. The power of the language has led to proliferation of complex Blueprints that are difficult to mechanically modify for *Large Scale Changes* [9], even with the support of modern LLM tooling [1].

Users specify in the Blueprints what TAP should run by defining *TAP Projects*. A TAP Project has *patterns* (that match Bazel targets), Bazel flags (which control compilation modes such as building for Arm), a maximum execution frequency, and notification routing. Rapid Projects are also defined in Blueprints. These projects can be extremely complex. We will focus just on how users configure testing. A Rapid Project can be *gated* on one or more TAP Projects, which must all pass. When they all pass it is called a *Shared Green* or a *Golden*. This is shown in Figure 1.

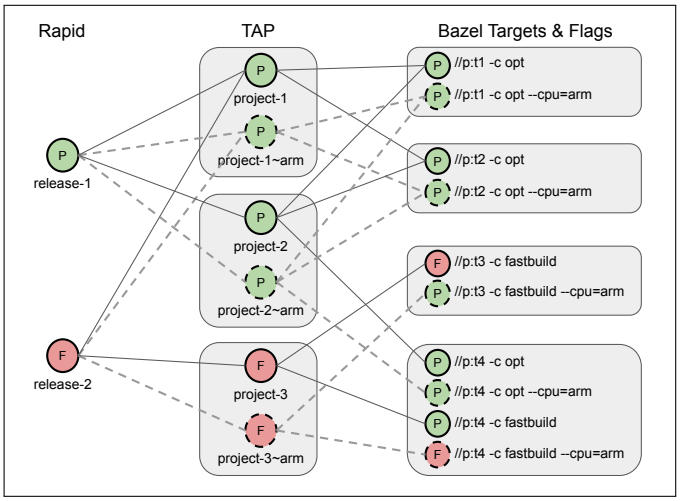

Fig. 1: The structure of Rapid and TAP Projects. TAP projects contain multiple Bazel Targets and they are shown along with the configured Bazel flags. Rapid may be gated on multiple TAP Projects. In order to for a Rapid Project to be passing (P) all targets in all TAP projects must pass. Otherwise the project is failing (F). The automatically generated ∼arm Variant TAP projects are shown.

```
// I need to manually create Arm Buildable Units
local arm_buildable_unit=::blueprint::BuildableUnit(
  name="arm_ex",
  build_flags=["--compilation_mode=opt", "--cpu=arm"],
  test_patterns=[ "//ex/pattern/...", "-//ex/pattern/x86tests/..."]);

// And associate them with a new Arm TAP Project
blueprint_file=::blueprint::BlueprintFile(
  continuous_tests=[
    ::blueprint::ContinuousTest(name="ex", buildable_unit_name="ex"),
    ::blueprint::ContinuousTest(name="arm_ex", buildable_unit_name="arm_ex"),
  ],
  buildable_units=[buildable_unit, arm_buildable_unit]);

continuous_tests=[
    ContinuousTest(name="ex", buildable_unit_name="ex"),
    ContinuousTest(name="ex", buildable_unit_name="ex_config"),
    ContinuousTest(name="ex.arm", buildable_unit_name="ex.arm"),
    ContinuousTest(name="ex.arm", buildable_unit_name="ex_config.arm"),
]
// which I add to my Release project's Golden Info.
golden_changelist=[
    GoldenInfo(tap_build="ex"), GoldenInfo(tap_build="ex_config"),
    GoldenInfo(tap_build="ex.arm"), GoldenInfo(tap_build="ex_config.arm")
]
// create duplicate release with Arm config.
ReleasableUnit(
  name="ex_multiarch_release",
  buildable_unit_names=[ "ex", "ex_config", "ex.arm", "ex_config.arm"])

// and I add --fat_package_cpu=k8,arm to all Release BuildableUnits
BuildableUnit(
  name="ex_package",
  mpm_patterns=["//ex/pattern/..."],
  test_patterns=[ "//ex/pattern/...", "-//ex/pattern/x86tests/..."],
  build_flags=["--fat_mpm_cpu=k8, arm"],
  ...
)
```

Listing 1: From a developers perspective, lines of code needed to manually configure arm-based testing and release automation. Note that all these changes would need to be manually reviewed and approved by a teammate.

```
blueprint_file = ::blueprint::BlueprintFile(
  arm_variant_mode = ::devtools_blueprint:: VariantMode::VARIANT_MODE_RELEASE;
  continuous_tests = [
    ::blueprint::ContinuousTest(name = "example", buildable_unit_name = "
       example"),
  ],

  buildable_units = [buildable_unit]);
```

Listing 2: Simplified variant configuration

Urfin, the managed platform, generates Blueprints (and other configuration) from a higher level "intent"-based configuration format. Urfin's configuration is less flexible than Blueprints and large scale changes are easier to mechanize.

C. TAP Overview

The continuous integration system, TAP, has two phases: TAP Presubmit and TAP Postsubmit. Presubmit runs as part of the developer's "code review loop", executing when they send out changes for review and when they attempt to submit the code to the repository. Because of the massive scale of the repository, TAP Presubmit does not conduct exhaustive testing. Instead, it operates in a "best-effort" mode that attempts to prevent most breakages from escaping into submitted code. TAP Presubmit uses static build dependence-based test selection [12], [16], machine learning driven selection [17], and machine learning driven flakiness mitigation [18]. In this paper, we are focused on TAP Postsubmit's behavior and look forward to analyzing changes to Presubmit in a future manuscript.

TAP Postsubmit runs tests after changes have been submitted. While it originally ran the *affected* tests (based on the build graph) at every single change [13], this quickly proved too expensive. Today, Postsubmit runs periodically and avoids running if its Forge is too full. This strategy effectively batches many changes together, leading to a search problem when breakages occur [4], [5]. We call these "periodic runs" *Fully Comprehensive* (FC) testing cycles since TAP runs all tests in all projects with all Bazel flag configurations that have been affected since the last cycle. Because FC cycles currently run at a maximum rate of once per hour, in 2025 we introduced a faster ML driven cycle to find breakages faster called *Speculative* cycles [7]. Speculative cycles only run targets whose executions are likely to reveal novel breakages.

D. Rapid Overview

Rapid runs user-defined "workflows" that are generally triggered off of a schedule. The first workflow, *Candidate Creation*, always creates a release "candidate", which includes verifying test results and building the binary packages. When Rapid is configured via Urfin, the managed platform, the workflows are standardized and structured such that we know which workflows can change production environments. Rapid's schedule for Candidate Creation defaults to once every 2 hours in Urfin, and production pushes happen once per day on business days. Many Rapid Native projects are configured at a slower cadence.

When verifying test results, Rapid can execute any tests not previously run. Because it evolved independently from TAP, Rapid has always maintained its own execution logic, which historically led to redundant and wasteful testing effort. As part of this project, Rapid now recognizes which tests TAP has already completed; by eliminating these redundant executions, we have also reduced the risk of a release being blocked by flaky test failures.

III. Configuring CI/CD for the New Architecture

In order for a project to take advantage of the new compute platform, it needs to be built and tested for the new platform. Practically, this requires adjusting its Blueprint (the configuration, §II-B) to additionally instruct the CI/CD systems to build and test for the new platform.

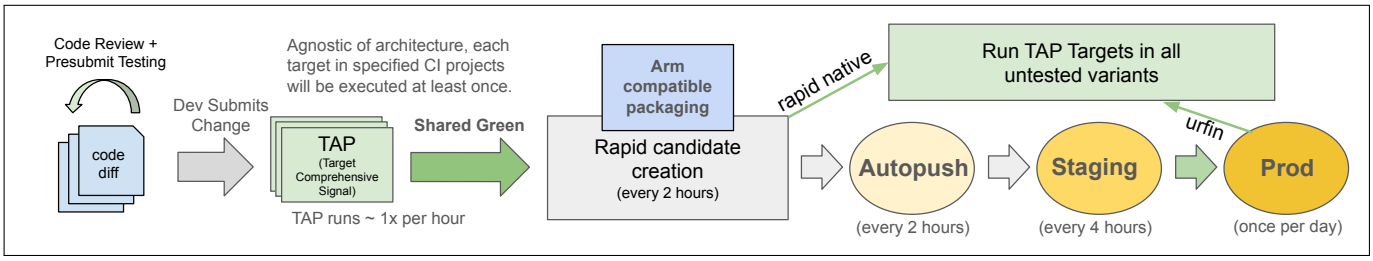

Fig. 2: High level diagram of the CI/CD process at Google for projects building for Arm and x86 using Target Comprehensive testing. The diagram shows the default frequencies for each step.

Listing 1 provides an example of the work it takes to manually create new CI/CD projects to build and test a binary for Arm. As shown in the listing, multiple sections need to be modified and these modifications cannot always be done atomically. Depending on the complexity of a project's configuration, it may require refactoring of the functions or libraries used to generate the various components (such as buildable units) to generate both Arm and x86 versions of those components. Approximately 25% of active Rapid projects have complex Blueprints requiring manual modifications.

Additionally, as the central team gained experience migrating our largest binaries to Arm they discovered a few best practices. For instance, it was better to already have continuous testing results before going to the product team with requested changes. This allowed the product team to better assess the changes. However, setting up a normal CI pipeline for running the Arm tests before notifying the product team is a recipe for friction and pushback, as there are likely to be real and noisy failures. Thus, the central porting team needed a way to run the tests in the same way the product team would.

To solve this problem, we came up with the idea of having "shadow CI projects" that would be duplicates of the production CI projects, but turn off notifications, switch the cost accounting, and add the necessary flags to run the project on Arm. To create these "shadow CI projects", the central team had to create Blueprint files in their own directory that imported and modified the target project's configuration. While the central team had a process that let them drive migrations to increase the migration rate, we needed a more automated solution.

A. Variants

The experiences of the central migration team led us to create *Variant* projects. These are automatically generated CI Projects that take the user-supplied configuration and adjust it to use different Bazel flags. For example, adding the flag `--cpu=arm` will compile and test for Arm. Variants are general and allow us, as the CI system owner, to dynamically define new variants as needed.

Figure 1 shows the structure of how Rapid and TAP interact with the new `~arm` variant. Each TAP project has a copy made (potentially removing or adding targets as necessary), and Rapid utilizes the shared green between the original projects and the variant projects when creating release candidates. The new TAP projects are largely identical to the original project, but we have the ability to globally modify any setting in the variant projects we choose. We used this to implement multiple "modes" for a variant: `DISABLED`, `SHADOW`, and `RELEASE`. We were then able to launch variants in `SHADOW` mode globally with no user involvement or toil from the central migration team. This allowed collection of repository wide testing results to guide porting effort. When a project is ready to adopt Arm, a simple top-level field lets them switch to `RELEASE` mode.

Variant projects were designed to be largely *identical* to regular user configured projects. This meant most portions of TAP, downstream systems, and API clients initially worked unmodified with the variants. However, we eventually introduced differentiated behavior. For instance, we changed the web user interface to explicitly show the user the relationships between the projects when a variant is enabled and has valid targets.

B. Porting Process with Variants

By default, all projects run their Arm variant in shadow mode. Once a project's tests all pass on Arm it becomes eligible for enabling the Arm release (but not necessarily running in production [1]). To enable the Arm release, we added a single top level field to the Blueprint (plus service level opt-outs) that enables Arm for all CI and Release projects defined in the Blueprint. This field `arm_variant_mode` is shown in Listing 2.

This new field drastically simplifies the porting process. It can usually be mechanically added (although not always). In §VI-E, we estimate the total savings of human labor from this simplification. With the automated construction of shadow projects and the ability to mechanically drive changes to Blueprints, we were able to fully automate portions of the porting process.

C. Complexity of Implementation

In §VI-E, we analyze the Return on Investment of this project and find that it was positive. However, the actual implementation of variants was complicated for many reasons that are outside of the scope of this paper to fully describe. For instance, TAP manages a set of projects and their targets at the scale of Google's multi-billion line mono-repository. Those projects and targets form a bipartite graph where a project contains multiple targets but a target may be inside multiple projects. Adding a single variant doubles the size of that graph. There are technical challenges involved with optimizing the variant implementation for computing, storing, and serving this

graph. Note, the graph is dynamic as it changes at practically every submitted change.

Other, more prosaic challenges emerged, such as finding the correct policy for detecting incompatible projects, redesigning data storage layers for project status, and renegotiating the contract between Rapid and TAP, among many other challenges. We regret that we do not provide the full saga of the development.

IV. Target Comprehensive Testing

One benefit of variants we have not yet discussed is their structured nature. When developers manually create new CI and CD projects as shown in Listing 1, there are no constraints on what they might change. We cannot enforce standard naming conventions automatically, nor can we even ensure they create them in the same Blueprint configuration file. Developers may add or subtract additional flags in non-standard ways. They may add or remove sections of their code from the patterns to be built and tested. Because of this, the central services (TAP and Rapid) would need to make conservative choices when qualifying releases.

However, variants have none of the above problems. Their definitions (what flags are added, what kind of targets are excluded, etc...) are centrally defined and audited. This enables direct monitoring (as shown in §VI-C) of behavior divergence between x86 and Arm. It also enables us to make changes to how we schedule tests in TAP and Rapid.

As discussed in §II, TAP has traditionally run all tests in all configurations. We call this testing mode *Fully Comprehensive* (FC). In order to find breakages faster, we then also ran tests in a *Speculative* mode [7]. Now, we introduce a third mode enabled by variants: *Target Comprehensive* (TC) testing. Under TC, we guarantee that targets are run *at least* once in some configuration (x86, arm, or other variant). Because of the automatically generated nature of variants and our central monitoring, results in one configuration are generally *substitutable* for another configuration (see §VI-C).

A. Integration with Release

A TAP project is eligible for Target Comprehensive testing when all release projects that reference it are enabled for Arm (see Listing 2). Under TC, TAP only guarantees to run a target on a single architecture (but it can choose to run it on more). However, before a release goes to production (and potentially executes on an architecture different than the one TAP tested on), we want to qualify it in all configurations. To do so, we run the remainder of the tests that TAP skipped during the release process. Figure 2 visualizes how this all connects together.

In release, there are two types of projects: *Rapid Native* (unmanaged) projects and *Urfin* (managed) projects (§II). By default, Urfin projects "Create Candidates" every two hours and push production releases once per business day (excluding Fridays). Because of the structured nature of Urfin, we know centrally when an Urfin project pushes to production. This allows us to *delay* the finalization of testing to immediately before the production push. We call this *Late Stage Testing*.

In Rapid Native, we do not know when in the release process a developer may be modifying production. Therefore, we need to continue to complete testing in the traditional Create Candidate phase. Luckily, many Rapid Native projects tend to be configured to Create Candidates once per day.

B. Cost Savings

Target Comprehensive testing shifts the load *right* in the Software Development Life Cycle (closer to the user and further away from the developer). This is expected to save machine cost by reducing the frequency at which tests are executed on both architectures. We expect significant savings, since production pushes usually only occur approximately once per day and TAP testing cycles occur once per hour. In §VI-A, we empirically evaluate our realized savings. While lower than the theoretical maximum for a variety of reasons, the results are significant and financially meaningful to us.

V. Empirical Evaluation

A. Research Questions

RQ1: How does the Target Comprehensive scheduling strategy, which executes a union of test targets once across architectures ($T_{\text{x86}} \cup T_{\text{Arm}}$), compare to a Fully Comprehensive baseline ($T_{\text{x86}} \times T_{\text{Arm}}$) in terms of compute resource consumption and project status latency?

RQ2: Does the Target Comprehensive approach maintain release success rate compared to Fully Comprehensive testing? Specifically, what is the impact on breakage detection latency and rate of late failure detections?

RQ3: How does an intent-based configuration abstraction, `arm_variant_mode`, compare to manual Blueprint configuration in terms of upfront engineering cost (measured as saved migration effort)?

B. Note on Statistical Testing

Throughout our analysis below, our primary method for assessing validity and effect size is Bootstrap resampling (with 10,000 samples) to construct confidence intervals at the 95% confidence level using the percentile method [19]. While our sample sizes are large enough to generally "ignore" normality violations, the violations are at times quite severe in our datasets. To build confidence in the conclusions we usually apply a secondary parametric analysis (such as an appropriate t-test or regression model). The results of the parametric tests confirmed the non-parametric Bootstrap method. Finally, the Bootstrap method returns larger confidence intervals than the corresponding parametric method employed. We present these more conservative intervals in the text of the paper.

C. Analyzing the Effect on Machine Costs

The motivating reason for Target Comprehensive testing cycles was to save machine costs as double testing was deemed unsustainable. So, did the cost savings materialize? Forge provides highly-detailed *per action* logs of executor cost and

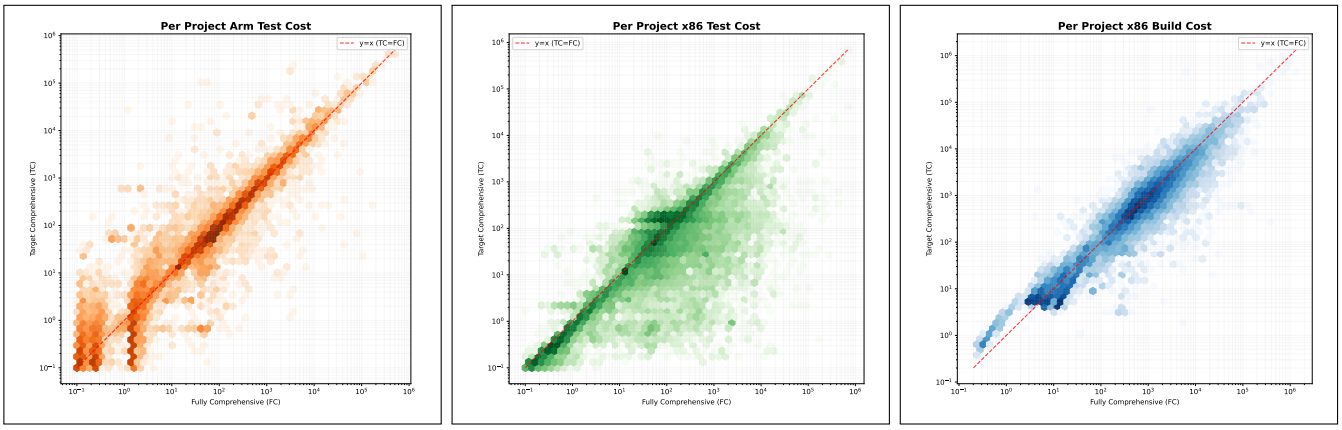

Fig. 3: Hex-binned 90th percentile machine cost (per project) of running projects. Here we see both build costs as well as testing costs. The Y-axis of each plot is the cost for Target Comprehensive. The X-Axis of each plot is the cost for Fully Comprehensive. Bins that are below the dotted red line (y=x) contain projects whose costs decreased with Target Comprehensive. Bins that are above the red line contain projects show costs increased. Bins that are low on the Y-Axis but high on the X-Axis show projects that are expensive to run under Fully Comprehensive but cheap to run under Target Comprehensive. In general, we see the largest savings gained by Target Comprehensive for x86 Test Costs as expected.

time. An action is a single build action that Bazel invokes. For instance, a compiler or linker command. Test actions (when a test is executed) are trivially differentiable from the build actions.

We analyzed Forge usage data from TAP Postsubmit (CI runs occurring after code submission) for 44,243 projects that had Arm testing enabled. We collected data over several weeks, joining Bazel targets with Forge actions to estimate the amortized build cost, and collecting the direct execution cost for each test action. For each testing cycle run by TAP Postsubmit, we summed the amortized build costs and test costs for all targets in the project to create a total cost per cycle. Target costs were normalized when shared by multiple projects in the same testing cycle.

We performed a paired comparison (by project) of the Target Comprehensive (TC) and Fully Comprehensive (FC) costs. We evaluated performance using two distinct metrics: *per-project* savings and *global system* savings. In both cases, savings are defined as $1 - {}^{TC}/_{FC}$.

To estimate the expected "per project" savings, we used the Geometric Mean (GM) of the ratio for all projects: $1 - \text{GM}({}^{TC}/_{FC})$. To estimate the expected system (or total) savings, we estimated the ratio of the sum of costs across all projects in a scheduling cycle: $1 - \sum TC / \sum FC$.

For both estimates, we used the non-parametric method Bootstrap re-sampling [19], which does not require an assumption about normality of the data. As can be seen in Figure 3, the data has heavy tails and multiple peaks violating this assumption. Using Bootstrap, we resampled (for each estimate) 10,000 times and computed the expected savings (the median) and 95% confidence interval using the $2.5^{th} - 97.5^{th}$ percentiles. Finally, we cross checked the per-project estimate using a paired t-test.

D. Analyzing the Effects on Project Status Latencies

A project status includes the results of all targets and indicates if the project is passing, failing, broken, etc. *Project status latency* is the time elapsed between successive calculations of project statuses at each cycle. To account for distribution variability, we utilize a Poisson Bootstrap method with 10,000 resamplings to analyze latency across 2.75 million FC statuses (before launch) and 2.94 million TC statuses (after launch). The cohort consists of 31,996 TAP projects that were Arm-enabled just prior to launch of TC.

E. Analyzing Breakage Detection Latency

Breakage detection latency is defined as the amount of time between when a developer submits a bug that breaks a test and the breakage being detected through a CI execution. We use Bootstrap analysis to compute confidence intervals to compare this breakage detection latency before and after the launch of Target Comprehensive testing. Further parametric analysis was completed that agrees with the non-parametric analysis, which we omit for brevity.

F. Analyzing the Effects on Release Failure Rate

Our second question, RQ2, asks whether the reduction in coverage from Target Comprehensive testing caused additional failures in release. As a reminder, for binaries that will actually be released, we run the remaining tests that were not run during Target Comprehensive. Thus, every test is run on both platforms if it runs in production. Running on both platforms happens much less frequently and enables the savings shown in RQ1.

We gathered data on 91,815 Release Projects, including both projects with Arm enabled and not enabled. Initial normality checks revealed the failure data to be extremely heavy-tailed and zero-inflated: the vast majority of projects never fail, while a small minority fail constantly. We filtered out approximately 130 projects that failed 100% of the time in both periods, as manual inspection confirmed these were abandoned and would otherwise skew volume-weighted metrics.

We analyzed the two project types: "Rapid Native" (flexible workflows) and "Urfin" (managed/structured workflows). For

Urfin, we distinguish between the standard "Candidate Creation" phase and the additional "Late-stage" testing performed immediately before production push.

We compare across these 3 groups the change in test failure rates. We look at the comparison in two ways:

1) *Paired Analysis (System Burden):* We calculated the mean arithmetic difference in failure rate for each project before and after Target Comprehensive was enabled. This measures the expected change in the total failure rate on the system.
2) *Geometric Analysis (Typical Experience):* We analyzed the change in the geometric mean of project failure rates. This metric controls for volume bias (projects with high execution counts) and outliers, answering the question: "Did the failure rate of a typical project change?"

Given the non-normal shape of the data, we utilized Bootstrap re-sampling (10k iterations) to estimate 95% confidence intervals as our primary method. We cross-checked the paired analysis using a Wilcoxon Signed-Rank Test to detect non-random drifts in rank order, and cross-checked the geometric analysis using a Robust Linear Model (RLM) to provide parametric validation robust to outliers.

G. Estimating Engineering Cost Savings

We want to compare the engineering costs of migrating projects using the new variants approach versus a more complex legacy method. We measured effort using the following normalized units that represent the time that a single software engineer spends working.

Definition 1 (SWEy)**.** *A SWE Year is a normalized unit that represents a software engineer working for a year.*

SWEy are treated as interchangeable regardless of which person or how many persons work on the migration. Note that coordination costs between multiple people working on a migration are also neglected. Due to confidentiality constraints, we are unable to report our internal SWEy numbers that we have computed. Instead, *total effort saved* will be reported, which provides a percentage of working time avoided due to the improved configuration mechanism.

To estimate the total effort, we began by estimating the average or expected effort to migrate a single configuration file (i.e., Blueprint). This was estimated using data from the team conducting central high-priority migrations. This team spent approximately 80% of their total time on conducting manual migrations before variants were introduced. Based on their migration output over the course of an entire quarter, we derived a specific (but confidential) estimate of the average time spent per Blueprint.

Definition 2 (BPe)**.** *BPe = "Blueprint Effort", or the time spent to migrate a single Blueprint without variants. (Actual number is confidential).*

Note, the above estimation was computed from expert engineers who were tasked only with conducting migrations. This central team only migrated a small fraction of the total projects. Engineers who don't work on migrations every day are less expert in conducting the migration and they take longer. We estimate that the migration would take twice as long for a non-expert engineer. We use the range $[\text{BPe}, 2 \cdot \text{BPe}]$ to estimate a normal distribution of effort.

In the same way, we can estimate Ve:

Definition 3. *Ve = "Variant Effort" of the time spent to migrate a single Blueprint using variants. (Actual number is confidential).*

Using these estimates, we can estimate total migration costs:

$$\text{baseline\_total}_{(\text{SWEy})} = \text{project\_count} \times 2 \times \text{BPe}_{(\text{SWEy})} \quad (1)$$

$$\text{variant\_total}_{(\text{SWEy})} = \text{project\_count} \times \text{Ve}_{(\text{SWEy})} \quad (2)$$

To compute return on investment, we account for realized machine savings (see Section VI-A) and the development cost (number of SWEy spent on development).

$$\text{ROI} = 1 - \frac{\text{variant\_total} + \text{development} - \text{machine\_savings}}{\text{baseline\_total}}$$

To estimate the return on investment when many of the parameters for the computation are estimates themselves, we used a Monte Carlo simulation parameterized by our ranges of values for quantities such as BPe, VPe, and the machine cost savings. We assumed these values could be drawn from a roughly normal distribution and took 100k samples of each for the MC simulation.

H. Threats to Validity

Developer behavior and resource usage predictably varies throughout the year, with large observed dips during holiday periods. Our comparison accounts for this effect by excluding holidays. Additionally, for our most business critical question (machine cost savings), we were fortunate to be able to conduct a paired analysis by executing a combination of both cycle types after launch. This provided us a robust data set of FC cycles before and after launch and TC cycles after launch, which helps us account for any time based factor. Second, our return on investment computation relies on estimation of effort by experts and may not be perfectly reflective of actual effort. While we attempt to account for this effect, we acknowledge the limitation of both our attempt, and the Monte Carlo simulation used to provide the confidence interval. Further, factors outside the scope of this paper, such as any differences in actual machine cost or other unaccounted costs (either human or machine) could reduce (or strengthen) the actual return.

Finally, this is a case study of one organization conducting a single project to scale their environment to another machine platform. Our results may not generalize. We observe in our own data that the existence of platform specific test behavior is highly dependent on the context of the specific project. We expect other organizations may have a different mixture of code, and if most code is sensitive to the runtime platform then the savings realized here will be unrealizable for that organization.

VI. Results

A. RQ1: Compute Resource Consumption

Figure 3 shows a breakdown of machine costs for running projects under Target Comprehensive and Fully Comprehensive testing cycles. As seen in the figure, there is a high variance in project costs. The plots indicate that x86 Test costs were generally lower under Target Comprehensive (as expected), while Arm Test costs were generally higher (as expected). Table I summarizes the results of our statistical analysis. We observed total savings of $\sim$25%; statistical analysis confirmed the trends evident in the plots, with the most savings derived from avoiding x86 Testing. However, we also saw a substantial 14.8% decrease in build costs attributable to a decrease in double compilation under TC testing.

We validated the per-project savings using a paired t-test on the per-project mean of log-transformed costs. The result was highly significant ($t(44242) = -101.3, p < 0.001$), and the derived confidence interval aligned closely with the non-parametric Bootstrap estimates reported in Table I. This convergence reinforces the statistical validity of our findings. Diagnostic checks (Q-Q plots) confirmed that the data is left-skewed (skewness: -1.46) and leptokurtic (kurtosis: 19.8), exhibiting heavy tails. Despite these deviations from normality, the large sample size supports the use of the parametric t-test, while the Bootstrap analysis provides a robust, assumption-free confirmation.

Summary: Target Comprehensive testing has an expected system wide savings of $\sim$25% versus Fully Comprehensive.

TABLE I: Build and test cost savings of Target Comprehensive scheduling.

Category	Typical Project Savings (Geometric Mean)	Global System Savings (Expected Total Savings)
Total	20.6% (20.3% - 21.0%)	24.9% (19.6% - 29.6%)
x86 Build	16.0% (15.6% - 16.3%)	14.9% (7.7% - 21.4%)
x86 Test	56.9% (56.4% - 57.5%)	66.7% (58.0% - 72.8%)
Arm Test	-25.5% (-26.8% - -24.3%)	-18.4% (-28.7% - -9.3%)

The table summarizes the estimated cost savings comparing Target Comprehensive (TC) to the baseline (FC). We report the median savings and 95% confidence intervals (in parentheses) estimated via Bootstrap resampling. Typical Project Savings represents the geometric mean of the ratio of TC to FC costs per project, GM($^{\text{TC}}/_{\text{FC}}$), reflecting the expected speedup for a typical project. The expected "Global System Savings" (in contrast) estimates $\sum \text{TC} / \sum \text{FC}$ for all projects in a scheduling cycle.

TABLE II: Project Status Latency in minutes (p50 and p90).

	p50 [95% CI]	p90 [95% CI]
FC	88.50 [88.43 − 88.58] min	200.78 [200.43 − 201.12] min
TC	75.08 [75.03 − 75.13] min	176.58 [176.25 − 176.90] min
% Diff	−15.16%	−12.05%

B. RQ1: Comparison of Project Status Latencies

Table II presents the project status latencies and 95% CI for FC and TC, along with the percentage difference between the two periods. Significant architectural changes and varying project definitions occurred between the FC and TC periods. However, our analysis deliberately focuses on the **holistic, end-to-end latency** to capture the net impact on system performance, regardless of these confounding factors.

Summary: Target Comprehensive testing has a median latency savings of 15% and p90 savings of 12%.

C. RQ2: Breakage Detection Latency

Figure 4 summarizes the results of our analysis. We compared the latency of detecting newly-introduced breakages (from submission of the culprit change to the first failure being detected) before and after launch of Target Comprehensive. While the median breakage detection latency was unchanged, the mean and 90^{th} percentile showed significant changes. To understand why, we plotted a breakdown of the number of breakages occurring in ranges of latencies (ex. 17 breakages in 90 minutes – 105 minutes) before and after. We then used Bootstrap re-sampling to estimate the change in the number of breakages likely to occur in each bucket. As visualized in Figure 4, the results indicate that we can expect approximately 4 additional breakages per day that take more than 12 hours to detect. This is a regression and a cost of Target Comprehensive testing as implemented today. We look forward to further optimizations to address this in the future.

Summary: We expect 4 additional breakages per day that take more than 12 hours to detect.

TABLE III: Paired Analysis of Change in Test Failure Rate (Arithmetic Mean).

	δ Failure Rate	Wilcoxon Pval	Verdict
Rapid Native	0.09% $(-0.10, 0.27)$	$p = 0.003$	$-$
Urfin - CC	0.01% $(-0.00, 0.03)$	$p < 0.001$	$-$
Urfin - LS	-0.08% $(-0.11, -0.06)$	$p < 0.001$	$+++$

Verdicts: $(+++)$ Improvement, $(-)$ Potential Regression (CI spans 0).

D. RQ2: Late-Stage Defect Detection Rates

Figure 5 illustrates the distribution of release-time testing failures for projects with Arm enabled versus disabled, distinguishing rates before and after the rollout of Target Comprehensive testing. In general, the Urfin failure rate is approximately $^1/_{10}{}^{\text{th}}$ that of Rapid Native. Notably, Arm-disabled Rapid Native projects (which were unaffected by the rollout) also saw a regression during this period, suggesting a potential global regression in failure rates unrelated to the launch.

Tables III and IV summarize the results of our statistical analysis. We highlight two key findings. First, the paired

TABLE IV: Analysis of Change in Typical Test Failure Rate (Geometric Mean).

	δ Failure Rate	RLM Pval	Verdict
Rapid Native	0.008% $(0.004, 0.012)$	$p < 0.001$	$--$
Urfin - CC	0.001% $(0.000, 0.003)$	$p = 0.159$	$\sim -$
Urfin - LS	-0.003% $(-0.003, -0.002)$	$p < 0.001$	$++$

Verdicts: $(++)$ Small Improvement, $(--)$ Small Regression, $(\sim -)$ Conflict between Robust Linear Model (RLM) and Bootstrap, Prefer Bootstrap Prediction of Small Regression.

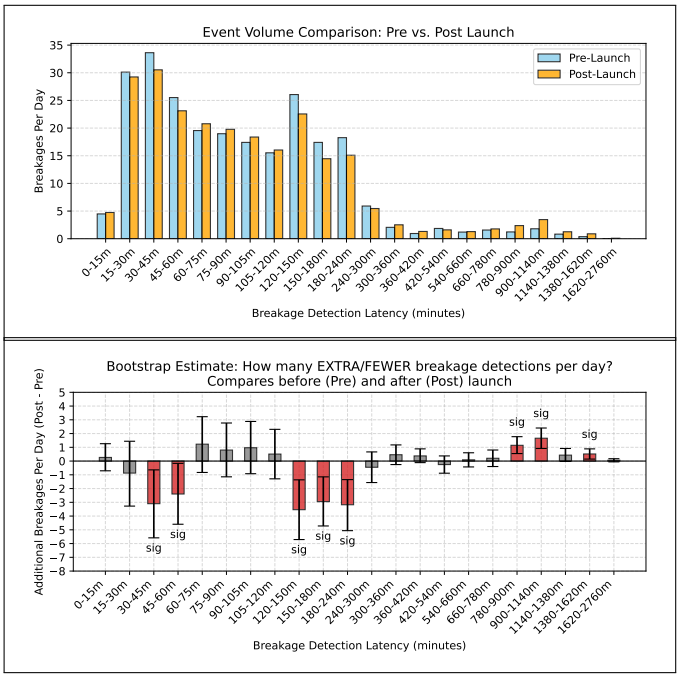

Fig. 4: Latency of detecting newly introduced breakages (from submission of the culprit change to the first failure being detected) before and after launch of Target Comprehensive. The chart on the bottom visualizes a Bootstrap analysis of the additional number of breakages per day for each latency bucket. For instance, approximately 1 additional breakage occurs per day that takes 780-900m to detect. This is a regression. The Bootstrap 95% confidence intervals were are shown and are used to determine significance (shown as "sig" on the graph).

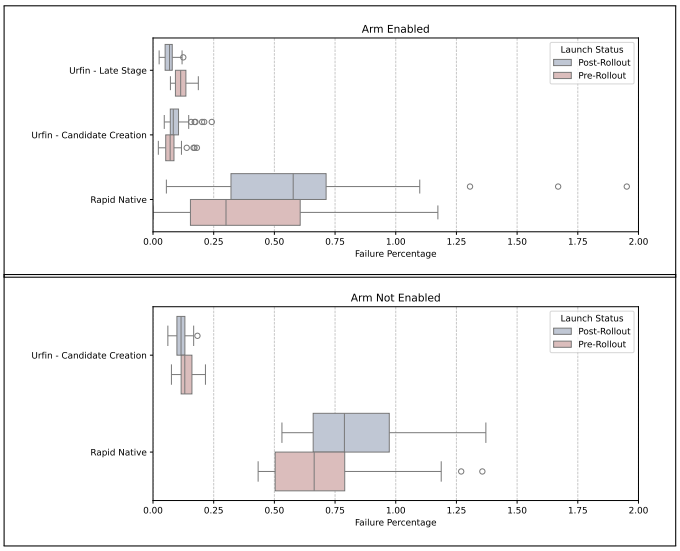

Fig. 5: Failure rate before and after turning on Target Comprehensive testing cycles.

bootstrap analysis for Rapid Native indicates a worst-case regression of 0.27%, but the confidence interval also spans zero, allowing for a possible 0.1% improvement. However, the Wilcoxon Signed-Rank Test reports a statistically significant difference in rank order ($p = 0.003$). Given this conflict between magnitude (CI includes 0) and rank drift (Significant P-value), we classify this as a *potential regression*.

Second, both analytical methods confirm an unambiguous improvement in the failure rate for Urfin Late-Stage testing. Prior to this rollout, Urfin projects executed Arm tests exclusively in the Late-stage phase (a MVP optimization while we waited for Target Comprehensive implementation to complete). The introduction of Target Comprehensive testing effectively *shifted left* this coverage. TC can identify Arm and x86 failures earlier in the TAP Postsubmit CI phase. This operational change explains the significant decrease in Late-Stage failures ($p < 0.001$). This result justifies the strategy of probabilistic early testing (TC) over exclusive late-stage verification (MVP optimization).

Summary: Rapid Native projects show a potential 0.1% regression in failure rates. Urfin projects show a clear 0.08% improvement (decrease) in late-stage failures.

E. RQ3: Engineering Savings

As shown in the Listings 1 and 2, the amount of code that must be added to enable a project to take advantage of Arm is much less with variants. Typically, projects need to change only a single line. We estimate that for the typical Blueprint, variants require only $^1/_{10}{}^{\text{th}}$ of the lines of code to maintain compared to the legacy system. Not all Blueprints needed humans to migrate them.

Many Blueprints could be handled by tooling (either traditional tooling or LLMs); however, there was a large number of Blueprints that did need human effort. We restrict our ROI computation below to those requiring human effort.

The new configuration syntax significantly reduces the effort to change a complicated Blueprint that cannot be automatically changed. We quantified this as percent change from legacy configuration to the modern configuration. With the modern syntax, 55% fewer Blueprints needed to be changed by hand.

Based on the Monte Carlo simulation, we believe our return on investment is approximately 40% and between (10%, 56%) when all these different costs are accounted for. We also estimated the savings in just migration cost. The MC simulation estimates a migration savings of 95%, with an interval of (91%, 98%). As more projects adopt Arm over the coming year, our realized return on investment for this capability will continue to grow, both in terms of saved migration costs and increased savings of machine costs as more projects adopt Target Comprehensive testing.

Summary: Our estimated ROI for this project is 40%.

VII. Related Work

In this paper, we presented an empirical case study examining the cost and benefit of the Target Comprehensive (TC) testing strategy. TC runs each test on either Arm or x86, but not both, for high frequency CI executions. This can be viewed as a trivial case of Combinatorial Interaction Testing (CIT) [20], [21] where we only have 1 parameter. CIT has long been a standard tool for testing the t-way interactions between various options in software testing. The most direct analogy for this paper comes from the world of mobile software testing. With the proliferation of devices,

operating system variants, and manufacturer customizations, mobile application developers have long had to test their software across many devices. This has led to the proliferation of commercial services to facilitate such testing (ex. Firebase Test Lab, AWS Device Farm, Sauce Labs, etc...).

Vilkomir explicitly studied the number of (random) devices one should test on to identify all bugs [22], [23]. They found that 13 devices achieved 100% coverage in their study and 5 devices provided 80%+ confidence that all bugs had been identified. They also examined if there were smarter ways to select the devices such that fewer devices would be required to achieve the same degree of confidence. In 2018, Kowalczyk *et al.* [24] also studied interactions between different properties of devices and test failures. Kowalczyk also found that the applications studied showed a variance in both failure rate and program coverage that depended on the configuration.

As reported by Lin *et al.* in 2023, the difficulties in managing the extensive (and expensive) on device testing strategy led ByteDance to create a "virtual device farm" [25]. Here, we find a direct analogy to Target Comprehensive testing. The Virtual Device Farm mimicked the real devices with emulators, though the fidelity was imperfect. To accommodate this, they would run the majority of their testing effort on the virtual farm. However, final validation tests ran on the physical farm: "Before the app's public releases, we conduct testing on physical devices to uncover the remaining bugs, which are mostly related to vendor-specific system services or drivers (§5.2). Since our continuous testing has captured most app-level bugs (which account for 93% of all bugs), tests on physical devices are much less frequently interrupted by test failures." [25].

In addition to the work in mobile testing, there is a large body of work assessing the feasibility of porting x86 code bases to Arm [1], [26]. We do not attempt to assess Google's overall migration effort here (see Christopher *et al.* paper [1]), but note that there are significant architectural differences between x86 and Arm that lead to correctness violations. These violations require precise changes to rectify, as they often involve synchronization between threads. We note that before a program is (initially) eligible to run on the Arm architecture it goes through additional qualification beyond its test suite [1]. One example of research on automatic migration is Beck *et al.*'s work [26]. They use static analysis to automatically translate synchronization patterns to safe and performant Arm equivalents during compilation.

Finally, our empirical assessment on changes to Breakage Detection Latency (see §VI-C) relied on our past work on TAP [4], [5], [7]. In 2023, we introduced a highly accurate culprit finder, and more importantly, a methodology for automatically assessing its accuracy [4]. Culprit finding simply identifies which changes introduced a test breakage into the repository. This is made challenging by flaky and non-hermetic tests. We conduct extensive reruns (as documented in [4]) to automatically verify the output of the culprit finder. In Kondareddy *et al.*'s 2025 paper, we hand re-verified a sample of the output and determined our verification strategy is $99.2 \pm 3.3\%$ accurate. This verified dataset provides our definitive source of truth ($\pm 3\%$) on breakages and enables us to accurately report Breakage Detection Latency while accounting for flakiness to the best of our ability.

VIII. Conclusion

Automated configuration, *variants*, and *Target Comprehensive* (TC) testing cycles were introduced to enable large scale adoption of the Arm architecture at Google. A naive approach of double testing would have been unsustainable, doubling compute costs and potentially increasing friction from additional flaky failures. Further, requiring all projects to modify their individual configuration files was infeasible.

Our solution enabled the adoption of Arm throughout the organization by controlling machine costs and minimizing human toil. We saved 90%+ human migration effort. We saved $\sim 25\%$ of the machine costs. Even though we don't execute all the tests all the time, the delays in breakage detection were marginal (4 per day) over the total number of breakages across the repository. Finally, our final safety net (Late Stage Testing) only detects a failure in fewer than 0.1% of production promotion attempts, a negligible amount and dwarfed by other reasons a production promotion may be aborted. Collectively, this work demonstrates the feasibility of large scale adoption of heterogeneous computing platforms without doubling the cost of testing.

Contributions and Acknowledgments

Primary Author: Tim A. D. Henderson, *Secondary Author:* Sushmita Azad
Additional authors, alphabetic:

- *Data Analysis for Empirical Evaluation:* Chandrakanth Chittappa, Ali Esmaeeli, Laura Macaddino, Sam Manfreda, David Margolin, Dharma Naidu, Sabuj Pattanayek, Sachin Sable, Ruslan Sakevych
- *Drafting and Editing:* Dushyant Acharya, Adrian Berding, Kevin Crossnan, Wolff Dobson, Avi Kondareddy, Abhayendra Singh

Additionally, we would also like to thank the following people for their technical contributions towards the success of this effort. In alphabetic order: Michael Adelemoni, David Alfonso, Sterling Augustine, Mark Balch, Leeann Bent, Paul Bethe, Lexi Bromfield, Eric Burnett, Saahithi Chillara, Brian Chiu, Dan Cho, Eric Christopher, Eduardo Colaco, Nicolo Davis, Rumeet Dhindsa, Paul Diebold, Bobby Dorward, Pat Doyle, Ákos Frohner, Charlie Fu, Russell Gottfried, Mark Halberstadt, Minh Hoang, Jung Woo Hong, JaJan Hsu, Jin Huang, Xin Huang, Kevin Jiang, Pranav Kant, Danial Klimkin, Gregory Kwok, Leon Lee, Chris Lewis, Drew Lewis, Jordan Lin, Kevin Liston, Sean Luchen, David Michaelson, Dominic Mitchell, Morgan Mitchell, Maksym Motornyy, Katherine Nadell, Danny Nguyen, Denis Nikitin, John Penix, Evelyn Price, Daniel Rall, Emma Rapati, Ashley Richter, Alberto Rojas, Daniel Romero Sanchez, Carly Schaeffer, Aaron Schooley, Alexander Schrepfer, Will Schwarz, Markus Schwenk, Manish Shah, Santanu Sinha, Peter Spragins, Arvind Sundararajan, Xenia Tay, Greg Tener, Marjorie Thibodaux, Dima Tsumarau, Shanid Seerae Valappil, Sri Raga Velagapudi, Andrew Vorwald, Shu-Chun Weng, Michael Wexler, DJ Whang, Jeff Xia, Anthony Xiang, and Jun Xiang Yak.

References

[1] E. Christopher, K. Crossan, W. Dobson, C. Kennelly, D. Lewis, K. Lin, M. Maas, P. Ranganathan, E. Rapati, and B. Yang, "Instruction Set Migration at Warehouse Scale," Oct. 2025. [Online]. Available: http://arxiv.org/abs/2510.14928

[2] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. Piscataway, NJ, USA: IEEE, May 2017, pp. 233–242. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.16

[3] M. Fowler, "Continuous Integration," 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[4] T. A. D. Henderson, B. Dorward, E. Nickell, C. Johnston, and A. Kondareddy, "Flake Aware Culprit Finding," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2023.

[5] T. A. D. Henderson, A. Kondareddy, S. Azad, and E. Nickell, "SafeRevert: When Can Breaking Changes be Automatically Reverted?" in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Toronto, ON, Canada: IEEE, May 2024, pp. 395–406. [Online]. Available: https://ieeexplore.ieee.org/document/10638594/

[6] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco, "Assessing Transition-Based Test Selection Algorithms at Google," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2019, pp. 101–110. [Online]. Available: https://ieeexplore.ieee.org/document/8804429/

[7] A. Kondareddy, S. Azad, A. Singh, and T. A. D. Henderson, "Speculative Testing at Google with Transition Prediction," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Napoli, Italy: IEEE, Mar. 2025, pp. 510–521. [Online]. Available: https://ieeexplore.ieee.org/document/10988976/

[8] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[9] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming over Time*. Beijing: O'Reilly, 2020.

[10] K. Wang, G. Tener, V. Gullapalli, X. Huang, A. Gad, and D. Rall, "Scalable build service system with smart scheduling service," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, Jul. 2020, pp. 452–462. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397371

[11] K. Wang, D. Rall, G. Tener, V. Gullapalli, X. Huang, and A. Gad, "Smart Build Targets Batching Service at Google," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Madrid, ES: IEEE, May 2021, pp. 160–169. [Online]. Available: https://ieeexplore.ieee.org/document/9401973/

[12] P. Gupta, M. Ivey, and J. Penix, "Testing at the speed and scale of Google," 2011. [Online]. Available: https://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html

[13] M. Bland, "The Chris/Jay Continuous Build," Jun. 2012. [Online]. Available: https://mike-bland.com/2012/06/21/chris-jay-continuous-build.html

[14] J. Micco, "Tools for Continuous Integration at Google Scale," Google NYC, Jun. 2012. [Online]. Available: https://youtu.be/KH2_sB1A6lA

[15] ——, "Continuous Integration at Google Scale," EclipseCon 2013, Mar. 2013. [Online]. Available: https://web.archive.org/web/20140705215747/https://www.eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf

[16] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. New York, New York, USA: ACM Press, 2014, pp. 235–245. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2635868.2635910

[17] P. Spragins, "Efficacy Presubmit," Sep. 2018. [Online]. Available: https://testing.googleblog.com/2018/09/efficacy-presubmit.html

[18] M. Hoang and A. Berding, "Presubmit Rescue: Automatically Ignoring FlakyTest Executions," in *Proceedings of the 1st International Workshop on Flaky Tests*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–2. [Online]. Available: https://dl.acm.org/doi/10.1145/3643656.3643896

[19] T. Hesterberg, D. S. Moore, S. Monaghan, A. Clipson, and R. Epstein, "Bootstrap Methods and Permutation Tests," in *Introduction to the Practice of Statistics*, 7th ed. New York: W.H. Freeman, 2012.

[20] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, ser. Chapman & Hall / CRC Innovations in Software Engineering and Software Development. Boca Raton, FL: CRC Press, 2013.

[21] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker, "Chapter one - combinatorial testing: Theory and practice," ser. Advances in Computers, A. Memon, Ed. Elsevier, 2015, vol. 99, pp. 1–66. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245815000352

[22] S. Vilkomir, K. Marszalkowski, C. Perry, and S. Mahendrakar, "Effectiveness of Multi-device Testing Mobile Applications," in *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*. Florence, Italy: IEEE, May 2015, pp. 44–47. [Online]. Available: https://ieeexplore.ieee.org/document/7283025

[23] S. Vilkomir, "Multi-device coverage testing of mobile applications," *Software Quality Journal*, vol. 26, no. 2, pp. 197–215, Jun. 2018. [Online]. Available: http://link.springer.com/10.1007/s11219-017-9357-7

[24] E. Kowalczyk, M. B. Cohen, and A. M. Memon, "Configurations in Android testing: They matter," in *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*. Montpellier France: ACM, Sep. 2018, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3243218.3243219

[25] H. Lin, J. Qiu, H. Wang, Z. Li, L. Gong, D. Gao, Y. Liu, F. Qian, Z. Zhang, P. Yang, and T. Xu, "Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility," in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. Madrid Spain: ACM, Oct. 2023, pp. 1–17. [Online]. Available: https://dl.acm.org/doi/10.1145/3570361.3613259

[26] M. Beck, K. Bhat, L. Stričević, G. Chen, D. Behrens, M. Fu, V. Vafeiadis, H. Chen, and H. Härtig, "AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Vancouver BC Canada: ACM, Jan. 2023, pp. 61–73. [Online]. Available: https://dl.acm.org/doi/10.1145/3575693.3579849