

Instruction Set Migration at Warehouse Scale

ERIC CHRISTOPHER, KEVIN CROSSAN, WOLFF DOBSON, CHRIS KENNELLY, DREW LEWIS, KUN LIN, MARTIN MAAS, PARTHASARATHY RANGANATHAN, EMMA RAPATI, BRIAN YANG, Google, USA

Migrating codebases from one instruction set architecture (ISA) to another is a major engineering challenge. A recent example is the adoption of Arm (in addition to x86) across the major Cloud hyperscalers. Yet, this problem has seen limited attention by the academic community. Most work has focused on static and dynamic binary translation, and the traditional conventional wisdom has been that this is the primary challenge.

In this paper, we show that this is no longer the case. Modern ISA migrations can often build on a robust open-source ecosystem, making it possible to recompile all relevant software from scratch. This introduces a new and multifaceted set of challenges, which are different from binary translation.

By analyzing a large-scale migration from x86 to Arm at Google, spanning almost 40,000 code commits, we derive a taxonomy of tasks involved in ISA migration. We show how Google automated many of the steps involved, and demonstrate how AI can play a major role in automatically addressing these tasks. We identify tasks that remain challenging and highlight research challenges that warrant further attention.

1 Introduction

Migrating large codebases to a new Instruction Set Architecture (ISA) is a major engineering challenge. Examples include Apple’s migration from PowerPC to x86 and later to Arm [4], as well as the adoption of Arm by major hyperscalers (such as Amazon, Google, Microsoft). While there are anecdotal claims regarding the complexity and efforts required for such migrations [2, 18, 19], to our knowledge, there is no systematic analysis of what these ISA migrations entail, and how they are impacted by modern technologies such as improved software engineering tools and artificial intelligence (AI). In this paper, we perform such a systematic analysis for the migration of a multi-billion line codebase from x86 to Arm at Google.

Historically, the conventional wisdom has been that the biggest challenge in ISA migration involves translating machine code between ISAs [2, 19]. Correspondingly, there has been a significant amount of work on static [23] and dynamic [12] binary translation that automatically rewrites binaries compiled for one ISA to another. Binary translation was the main problem when software was distributed as binaries and source code was not usually available. However, modern ISAs are generally well-supported in upstream compilers, runtime libraries, and the Linux kernel. As a result, modern compilers mostly “just work” for a new ISA, and previous ISA migrations have smoothed the path to packages supporting cross-compilation by default. For example, 98% of Debian packages build for RISC-V, although it only became an official Debian architecture in 2023 [1].

Perhaps surprisingly, this does not mean that ISA migration is no longer a challenge. While code translation is not the main issue anymore, we find that modern ISA migration involves many usually-simple, repetitive, automatable tasks such as updating build scripts or fixing floating-point issues, which AI can increasingly facilitate. In this paper, we analyze a large-scale ISA migration at Google that added Arm support alongside x86. We focus on the following research questions:

- (1) What are the tasks that are involved in a modern ISA migration?
- (2) Which tasks can be automated and how can modern AI help?
- (3) Which tasks are difficult and are good targets for future research?

To answer these questions, we provide what we believe is a first-ever detailed breakdown and taxonomy of large-scale ISA migration tasks. Using state-of-the-art LLMs, we analyze and categorize a corpus of 38,156 commits that constitute our real-world migration. We quantitatively evaluate the capability of current tools, including AI models, to perform these tasks automatically. We

systematically identify the strengths and weaknesses of current automated tools, and highlight areas of future work and improvement. We believe that this work highlights research opportunities for the academic community and revisits long-standing assumptions around ISA migrations.

Specifically, we contribute the following insights: 1) The complexity of ISA migrations is not in code translation but involves a number of different tasks, many related to rewriting BUILD and configuration files; 2) Many of these tasks are highly automatable; 3) Many of the tasks that are not automatable only need to be performed once when going from a single ISA to multiarch; 4) Of the remaining tasks, many can be performed by modern AI, but some challenges remain.

2 Background & Related Work

There are a number of reasons why organizations have performed large-scale ISA migrations. First, many ISAs have gone extinct over the years (e.g., Alpha, MIPS, SPARC, Itanium, VAX). Second, with the adoption of Android and iOS, more codebases were ported to Arm to be used in mobile applications. Third, Apple Macs went through successful migrations from PowerPC to x86, and most recently from x86 to Arm to support custom Apple Silicon. Finally, major cloud hyperscalers have been migrating large codebases from x86 to Arm as well.

In this context, migration does not only refer to the act of getting software to build on a new architecture but to reaching parity in terms of performance, security and stability. The most closely-related work in the academic community falls into two categories.

First, there is a significant amount of work on static and dynamic binary translation from one ISA to another [12, 23]. Static or dynamic binary translators can serve as a bridge to handle the long tail of a software ecosystem, but eventually they must be phased out to avoid carrying forward technical debt (e.g., Apple will actively phase out its Rosetta dynamic binary translation system in 2027 [3]). In cloud computing, where compute is commoditized, developers seek as much efficiency as possible, and recompiling onto the new ISA is the best route to maximize the compiler’s options for performance. We thus forewent binary translation altogether in our deployment.

Second, there is a significant amount of work on automatically applying edits to code, such as for performance optimization [16], fixing security issues [14], or correcting bugs [5]. As we will see, these are common tasks that are part of a successful ISA migration.

3 Google’s x86 to Arm Journey

We now analyze Google’s multi-year effort to port a substantial portion of Google’s server application ecosystem from x86 to Arm, enabling simultaneous support for both. We start by describing Google’s environment and provide a step-by-step analysis of our ISA migration.

3.1 Google’s Software Ecosystem

Google’s codebase is organized as a monorepo containing billions of lines of code [20]. Individual applications and libraries reside in various directories. These folders also contain metadata files, e.g., to indicate code owners or configure continuous integration (CI) testing [27].

Building with Bazel. Builds use Bazel [7], a highly configurable build system. BUILD files describe how binaries, libraries, and tests are built from source files. Most code is covered by our primary continuous integration system, “TAP” (Test Automation Platform [7]), and it is standard for TAP to gate releases. Bazel’s builds and tests (including TAP) run on a shared set of machines called Forge. Forge’s scale and cache enables Google to compile everything needed for a binary from scratch on every build, including fundamental dependencies like the Python interpreter.

Creating releases with Blueprints. Google distinguishes between binaries and releases (named “MPMs” for the “Midas Package Manager” that stores them globally [7]). Releases are bundles

of binaries and data that are ready to be deployed in Google’s clusters, akin to a package in a Linux distribution. Releases are defined by Blueprint files, which are handwritten or managed by other systems that standardize releases and configurations. A system called Rapid [6] consumes Blueprints, runs CI tests, including TAP, and builds releases for server-side packages.

Running applications on Borg. Borg is a custom cluster management service for the Google fleet that runs nearly all Google services [26]. Applications are deployed to Borg through configuration files that define the MPMs needed to run a service, runtime parameters, and scheduling constraints. MPMs can be rolled forward and backward safely because they are almost entirely hermetic.

Multiarch Support. Borg was heterogeneous even before the Arm migration. Borg has had dozens of different types of CPUs over the years, and services run on machines that can be as much as ten years old. Unless an owner adds specific constraints, a job can be scheduled on any machine with an architecture-compatible binary. During development, engineers can request builds for multiple architectures (e.g., Arm, K8, Haswell) and expect multiarch MPMs. They can also request tests to run on each target hardware. At release time, owners of packages can also configure Blueprints to target one or more ISAs. Owners expect to get a mix of different kinds of machines with different performance profiles and, in some cases, ISAs.

Shifting down and Large Scale Changes (LSCs). Google has moved to a “shift down” approach to development [11], where developers only focus on one level of the stack and other issues are abstracted and/or automated for them. For example, Bazel, TAP, and Rapid mean that developers do not generally need to worry about the specifics of CI. In addition, a healthy automated testing culture means everyone can change everyone else’s code without frequent breakages. This enables Large Scale Changes (LSCs) [27] which change code owned by many different teams and can affect thousands of files at once. In cases where an LSC is considered low risk, it can be approved centrally and submitted efficiently without asking individual teams. To get approval from many owners at once, Google has developed Rosie [27] which allows engineers to create a very large commit and shard it into tens, hundreds, or thousands of smaller commits split up by owner.

3.2 Life cycle of an ISA migration

Moving an individual package from single arch (x86) to multiarch support requires several steps:

- (1) **Test:** Fix tests (and builds) that break when run with the new ISA. Since anyone can build and test any code in our monorepo, it is easy to identify tests that break and require fixes.
- (2) **Set up multiarch CI:** This requires modifying the corresponding Blueprint files to ensure that no additional regressions are introduced (often simultaneous with the next step).
- (3) **Configure releases:** This modifies Blueprint files to make releases multiarch by default.
- (4) **Roll out new binaries:** Run the multiarch packages on machines of the new ISA and assess performance and stability, addressing issues as needed.
- (5) **Full production:** Allow production jobs to be scheduled on machines of the new ISA.

While these steps are the same for all packages, the issues encountered within each step vary widely across applications and throughout the different phases of our ISA migration from x86 to Arm. Often, this involves performance-optimizing code for the new platform, which can happen in parallel to these steps. We observe parallels to Uber’s reported porting workflow [17].

3.3 Phase 1: Large users

We started our Arm migration with a small set of large users, such as Spanner [9], BigQuery [15] and Bigtable [8]. These migrations were hands-on with a small team, and required weekly meetings

and tracking bugs. Once tests passed, rollout was manual, with very careful performance and load testing, and gradually removing scheduling constraints on a per-job basis.

During this phase, a number of issues in these workloads were surfaced and addressed. Examples include: 1) Replacing x86-specific intrinsics; 2) Replacing `long double`, which differs between x86 and Arm, with `absl::float128`; 3) Brittle tests (e.g., due to exact floating point equality checks); 4) x86-specific flags; 5) Memory ordering issues hidden by x86; 6) Out-of-memory errors, often due to heap limits being tuned for x86; 7) Multiarch MPMs exceeding the capacity limits of our infrastructure; 8) Unsupported dependencies, and loading of unsupported dynamic libraries; 9) Jobs not getting scheduled due to unsatisfiable scheduling constraints in Borg configurations.

This list was surprising to the teams involved—initially, there had been a perception that porting these large and mature codebases to Arm would be a herculean task, and that the very different toolchains would result in myriad difficulties. However, most issues involved simple changes or fixes, many of them in configuration files. At the same time, these changes were surprisingly pervasive, as evident by the large number of commits. It is therefore not the case that most software compiles and runs on Arm without modifications; it is that these modifications are of a different kind than expected initially. For example, it is not unusual that, for a given software package, almost *nothing* builds initially, suggesting that large and pervasive changes are required. However, simple fixes to a number of shared dependencies often unblocks many of them at once.

3.4 Phase 2: Everybody else

To take full advantage of Arm in the data center, migrating only the largest workloads is insufficient. To make maximum use of available capacity, Borg needs to be able to schedule workloads flexibly across platforms, packing large and small users onto machines efficiently. If only a small subset of services can run on Arm, it will result in underutilization of those machines. We note that the distribution of workloads at Google is very flat: Although our top 50 users are very large, they only represent $\approx 60\%$ of running compute [13]. Addressing this long tail requires porting over 100,000 packages and billions of lines of code. This makes the Phase 1 approach of working directly with customer teams infeasible. In fact, even just talking to each team would be prohibitively expensive. The second phase of the x86 to Arm migration therefore focused on automating and scaling the migration of these workloads, while minimizing involvement from the teams themselves. It is this phase of Google’s x86 to Arm migration that we mostly focus on. So far, we have ported about 30,000 packages, accounting for a significant portion of CPU cycles. We found that effectively making use of Arm hardware did not require porting all workloads.

4 Analyzing an ISA Migration

To fully understand what is involved in an ISA migration, we now analyze the full range of tasks involved in Google’s x86 to Arm migration (RQ1). As a monorepo, any change – be it to code, configurations or documentation – is tracked as a commit in our repository’s history. Further, these relevant commits were marked with a keyword that indicates they were part of this migration, allowing us to extract them after the fact. We thus identified a relevant set of 38,156 commits.

Analyzing these commits manually would have been cost-prohibitive. We instead used a variant¹ of Gemini 2.5 Flash to analyze these commits at scale. We passed the commit messages and code diffs into the LLM’s 1M token context window in groups of 100 at a time. We prompted the model to pick a set of 20 categories for each batch. Then, we took all 400×20 categories and asked Gemini to consolidate them into 50. Further manual iteration over model outputs led to a final list of 16

¹We use Gemini 2.5 fine-tuned on a corpus of internal data, including code and documentation. This corpus may include material related to our Arm migration, but this represents a sufficiently small subset that recitation is not an issue.

Change Category & Description	Commits	Total LoC	LoC per Commit	Automation
Code Adaptation & Correction				
1. Platform-Specific Conditional Code: Introducing or modifying code blocks that are conditionally compiled or executed (e.g., using <code>#ifdef __aarch64__</code> , runtime CPU feature checks). Examples: Different syscall usage, platform-specific API calls.	236	16,235	9 [1, 330]	22% / 28%
2. Intrinsic and Assembly Code Porting: Replacing or providing alternatives for x86-specific intrinsics (e.g., SSE, AVX) with Arm equivalents (NEON), or rewriting assembly language sections.	74	4,454	10 [1, 209]	1% / 0%
3. Data Representation & Alignment Fixes: Modifying code to handle issues arising from differences in data type sizes, alignment requirements, or byte order (endianness) between x86 and Arm.	38	2,574	9 [1, 324]	0% / 0%
4. Memory Model & Concurrency Adjustments: Fixing code that makes assumptions about memory ordering, atomicity, or thread synchronization behavior that differ on Arm.	12	49	2 [2, 12]	0% / 0%
5. Performance-Driven Code Optimization for Arm: Refactoring algorithms or code patterns specifically to improve execution speed, reduce latency, or improve efficiency on Arm microarchitectures. This is beyond basic correctness.	18	2,080	13 [4, 337]	0% / 0%
Test Adaptation & Configuration				
6. Test Logic, Data, and Assertion Modifications: Changing the test code itself, updating golden files, or adjusting test assertions to be compatible with Arm, reflecting valid behavioral differences rather than bugs.	276	37,052	5 [1, 147]	3% / 1%
7. Test Execution Environment & Scope: Configuring which tests run on Arm, adjusting test timeouts, memory/CPU limits, and sandboxing; excluding tests not suitable for Arm.	1,303	13,783	1 [1, 37]	53% / 8%
Build, Deployment & Infrastructure Configuration				
8. Build, Packaging & CI/CD Configuration: Changes to BUILD files, Bazel settings, genmpm rules, Blueprints, TAP, and release platform configs to support multi-architecture builds, testing, and releases.	32,204	139,611	1 [1, 9]	95% / 63%
9. Borg & Runtime Environment Configuration: Modifying Borg configurations, allowlists, admission control, resource allocation within jobs, and service enablement for running on Arm in non-prod and production.	757	26,581	10 [1, 136]	1% / 0%
10. Infrastructure Resource Management & Provisioning: Managing quota, dedicated machines, security policies, storage, network, and scaling for the Arm migration. Includes cluster state management and kernel/platform rollouts.	381	32,159	8 [1, 163]	1% / 0%
Supporting Processes & Tools				
11. Monitoring, Alerting & Performance Analysis: Setting up dashboards, alerts, collecting metrics, analyzing performance benchmarks, and classifying errors for the Arm migration.	645	115,343	16 [1, 525]	0% / 0%
12. Migration Tooling & Automation Development: Creating and enhancing scripts, tools, and automation to assist with any stage of the Arm migration.	644	125,536	29 [1, 808]	2% / 1%
13. Documentation & Knowledge Management: Creating and updating guides, best practices, and debugging information.	369	24,115	13 [1, 260]	0% / 0%
14. Rollbacks & Cleanup: Reverting changes and removing obsolete code, configurations, or data from the migration process.	940	163,042	2 [1, 200]	58% / 5%
15. Specialized Service Configuration: Platform-specific adaptations for databases, experiment frameworks, or other unique services.	119	22,501	7 [1, 747]	3% / 0%
16. Release Qualification & Validation: Defining and implementing processes to ensure multi-architecture releases meet quality and performance standards on Arm, including using CHAMP data.	123	8,178	14 [1, 284]	5% / 2%
17. Uncategorized	17	1,328	39 [2, 223]	65% / 84%

Fig. 1. Categories of commits in Google’s x86 to Arm migration. LoC per commit shows median and 90% CI. Automation shows the fraction of commits/LoC generated using large-scale changes (Section 5.1).

categories (Figure 1)². Once this list was finalized, we ran the model on all commits again and had it assign one of these 16 categories to each of them (as well as an additional “Uncategorized” category, which improved stability by catching outliers). Figure 2 shows examples of each category.

Commits fall into four overarching groups: 1) Code changes, 2) Test changes, 3) BUILD files and configurations, and 4) Supporting processes and tools. In total, our commits updated around 700K lines of code. While the vast majority (84%) of commits are related to updating build or configuration files (Category 8), these commits account for only 19% of lines of codes updated. We also see a substantial number of lines (17%) spent on migration tooling. A large portion of this work is only required once and can be reused in future ISA migrations.

All categories contain a meaningful number of commits, supporting our claim that ISA migration is a multifaceted engineering challenge where no single type of task dominates. We also see that code related commits (Categories 1-5) only account for 1% of commits and less than 4% of lines of code, refuting the conventional wisdom [2] that code translation accounts for most of an ISA migration. In Section 5, we analyze how automatable these commits are.

We also analyze the timeline of our ISA migration (Figure 3). We observe that at the start of the migration, most commits were in tooling and test adaptation, aligned with Phase 1 (Section 3.3). Over time, a larger fraction of commits became around code adaptation, which can be seen as a phase when there is still a need to update code in common dependencies and address common issues in code and tests. Eventually, the fraction of these kinds of commits declines and in the final phase of the process (Section 3.4), almost all commits are configuration files and supporting processes. We also observe that in this later phase, the number of merged commits rapidly increases, capturing the scale-up of the migration.

²The descriptions in Figure 1 are almost entirely the model’s output, with some minimal edits to remove internal information.

1. Platform-Specific Conditional Code <pre>#define _CONFIG_H_ // distro #define SOLIB_EXT ".so" #if defined(__aarch64__) #define ARCH "aarch64" #elif defined(__x86_64__) #define ARCH "x86_64" #else #error Unsupported architecture #endif Note: Multiarch a config header</pre>	6. Test Logic, Data, and Assertion Modifications <pre>-EXPECT_THAT(distribution.mean(), DoubleEq(event_set.SpaceUsedLong())); +EXPECT_THAT(distribution.mean(), DoubleNear(event_set.SpaceUsedLong(), 1e-10)); Note: Adjust the floating point sensitivity</pre> 7. Test Execution Environment & Scope <pre>tags = { "cpu:4", "requires-mem:20g", }, deps = { Note: Increase a test's memory limit.</pre>	12. Migration Tooling & Automation <pre>... +EXPORTED_ACTIONS = { + + automation_structure_pb2.ACTION_PARSE_BLUEPRINT: + parse_blueprint, + automation_structure_pb2.ACTION_RUN_TEST: + run_test, + # Used for testing + automation_structure_pb2. + ACTION_WRITE_OUTPUT_FROM_INPUT: + write_output_from_input, + automation_structure_pb2. + ACTION_WRITE_OUTPUT_FROM_CONFIG: + write_output_from_config, +} ... Note: Part of a new Blueprint parser</pre>
2. Intrinsic and Assembly Code Porting <pre>#ifdef __aarch64__ #include "third_party/sse2neon/sse2neon.h" #else #include <emmintrin.h> #endif ... #ifdef __aarch64__ #define GETROUND() (_MM_GET_ROUNDING_MODE()) & VM_SSE_ROUND_MASK) #define SETROUND(x) (_mm_setcsr(x) (_MM_GET_ROUNDING_MODE()) & ~VM_SSE_ROUND_MASK)) #else #define GETROUND() (_mm_getcsr()) & VM_SSE_ROUND_MASK) #define SETROUND(x) (_mm_setcsr(x) (_mm_getcsr()) & ~VM_SSE_ROUND_MASK)) #endif Note: Port code using intrinsics</pre>	8. Build, Packaging & CI/CD Configuration <pre>cc_library(name = "vector", hdrs = ["vector.h",], copts = ["-mavx2", "-mavx2", "-mavx2",], visibility = [Note: Remove x86-specific build flags</pre>	13. Documentation & Knowledge Management <pre>... + * Borg config managed by the pilot dev team + specifically selects the pilot + machines for specific pilot jobs. ... - The allowlist of users permitted to run on the - pilot machines is managed via - the 'logs' usermap: ... Note: Evolving the docs for the Logs test</pre>
3. Data Representation & Alignment Fixes <pre>-char suffix_bytes[kFileSuffixSize]; // NOTE: alignas(uint32_t) is used to ensure // proper memory alignment when // reinterpreting the char pointer to a uint32_t // pointer. alignas(uint32_t) char suffix_bytes[kFileSuffixSize]; Note: Add an alignas modifier</pre>	9. Borg & Runtime Environment Configuration <pre>access_level = - + pa_proto.PlatformAllowlistAllowedCollection. + INITIAL_LIMITED_QUALIFICATION + + pa_proto.PlatformAllowlistAllowedCollection. + EXPANDED_PROTECTION }, { collection_key = { user = 'ul-mixer' } Note: Change allowlist to extend machine exposure</pre>	14. Rollbacks & Cleanup <pre>package_name = "archive", args = [{"archive"}], mpm_tags = [{"rapid"}], platforms = [{"/borg/all"}], deps = [{"oneday", "borgfiles"}],) Note: Rolls back a problematic data MPM change</pre>
4. Memory Model & Concurrency Adjustments <pre>stripCalcLatch = new CountdownLatch(1); calcAwaitingLatch = new CountdownLatch(1); ranStripCalc = new AtomicBoolean(false); appKey = new PlainKey("takeKey"); calculationContext = new CalculationContext(new FakeExtraData()); calcEnv = new InProcessCalcEnv(appKey, ritzInfo, calculationContext); ... - ranStripCalc = true; + ranStripCalc.set(true); ... Note: Switch to atomic boolean to avoid races (Java)</pre>	10. Infrastructure Resource Management <pre>borg_pool = arm-dedicated-pool {} borg_pool = arm-dedicated-pool {} borg_pool = arm-dedicated-pool {} +borg_pool = arm-dedicated-pool {} Note: Add a new cell for testing</pre>	15. Specialized Service Configuration <pre>+ +// a dedicated borg pool consists of and + machines for A/B testing +// - : for x86 100% +// environment +// - : for ARM 100% +// environment + _info arm-dedicated- - + _default_info({}) { + cell = 'arm-dedicated-' + user = 'arm-dedicated' + scratch_path_prefix = + '/cns/arm-dedicated' + remote_path = '/cns/home/' + platform = + } + Note: Add a configuration for a Spanner pilot</pre>
5. Performance-Driven Code Optimization for Arm <pre>... template <> inline uint64_t MathUtil::SafeRound(float x) { uint64_t result; __asm__ ("fcvtaw %x0, %x1 : "=x"(result) : "w"(x)); return result; } ... template <> inline int64_t MathUtil::SafeRound(double x) { return vcvtdw_s64_f64(x); } ... Note: Implement math routines for aarch64</pre>	11. Monitoring, Alerting & Performance <pre>+perf_workflow_test { + name = + "borg_workflow_arm_one_build_multi_runs", + configuration = { + "postprocessing_config": { + + "microbenchmark_data_uploading_config": + DEFAULT_MICROBENCHMARK_ + DATA_UPLOADING_CONFIG, + }, + }, + # To run workflows on ARM, need to have + "-cpu-arms" here. + extra_postprocessor_blaze_flags = + ["-cpu-arms"], + phases = { + + + + + Note: Add a new Arm-specific testing infrastructure</pre>	16. Release Qualification & Validation <pre>... WHEN champ_state = 'Incompatible' - THEN PrioritizedStatus(1, + 'NON_COMPATIBLE_VIA_CHAMP') + THEN PrioritizedStatus(0, + 'NON_COMPATIBLE_VIA_CHAMP') WHEN champ_state = 'Compatible' - THEN PrioritizedStatus(1, + 'QUALIFIED_VIA_CHAMP') + THEN PrioritizedStatus(0, + 'QUALIFIED_VIA_CHAMP') ... Note: SQL change for improved tracking of qualification</pre>

Fig. 2. Specific code examples for each category.

Finally, we want to understand how these different categories of commits differ from one another. We observe that median commits in most categories are less than 20 LoC, with many single-line commits. However, we also observe very large individual commits that change 10,000+ LoC and skew the averages. We manually inspected these commits to understand their origin and found that these commits do not typically represent more work since they are conceptually similar to large numbers of simple, one-line commits. Overall, there are 19 commits that cumulatively account for 238,289 LoC (32.4%) of total lines changed and that are trivial. Examples include:

- Remove a porting tool once it was no longer used – 57K LoC (Category 12)
- Update a list of microbenchmark targets – 23K LoC (Category 11)
- Add several very large test vectors for a coverage tool – 15K LoC (Category 6)

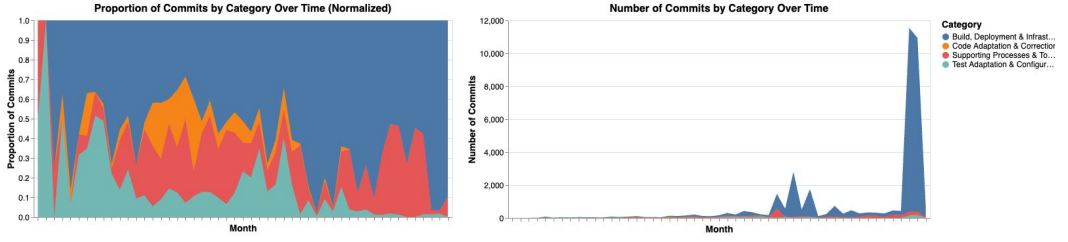


Fig. 3. Categories of commits over time.

In summary, we find that most commits related to migration are small, and that the largest commits often change very large lists or configurations and are not inherently complex. We also find that size alone does not measure difficulty. Finally, we observe that some of the commits (particularly in “Supporting Processes & Tools”) could likely be reused in a subsequent multiarch migration.

5 Automating ISA Migrations

Now that we have established the tasks that are part of an ISA migration, we can explore how automatable each of these tasks is (RQ2) and how novel automation approaches can facilitate them.

5.1 ISA Migration Automation at Google

We already employ a number of automation tools at Google today that automate a large portion of the ISA migration process (83.82% of commits and 14.15% of LoCs).

Large-Scale Changes (LSCs). The key piece to ISA migration automation is Rosie (Section 3.1), which allows us to programmatically generate large numbers of commits and shepherd them through code review. This includes running affected TAP projects, requesting code reviews by code owners, and submitting each commit once all tests pass. We find that 31,984 of our commits were generated by Rosie, signaling automation. However, we note that these commits only account for 14.15% of lines of code, indicating that most of these commits are very small. Figure 1 shows the fraction of each category that was generated using automated tools. For example, one major LSC adds the following line to Blueprint files of projects, configuring all of its tests and releases for Arm:

```
arm_variant_mode = ::blueprint::VariantMode::VARIANT_MODE_RELEASE,
```

Sanitizers & Fuzzers. While not limited to ISA migrations, fuzzers and LLVM sanitizers such as AddressSanitizer [21], MemorySanitizer [24] and ThreadSanitizer [22] are key enablers of our migration. Even before Arm adoption, Google routinely ran all TAP tests with these tools enabled, which turn latent errors such as a memory corruption, memory leak, or race condition into a debuggable fault. Application owners regularly triage and fix these faults, causing us to sidestep many common differences in execution between x86 and Arm (e.g., a data race may be hidden by x86’s TSO memory model). Catching these kinds of issues ahead of time avoids debugging non-deterministic and hard-to-debug behavior when recompiling to a new ISA.

Continuous Health Monitoring Platform (CHAMP). The final step in our automation is CHAMP, which assesses Arm-built applications on Arm server hardware. It continuously monitors health metrics to detect whether behavior differs from x86 instances of the job (e.g., significantly higher RPC error rates or crashes). If so, it automatically marks the job as ineligible for Arm, files a bug for its owners to follow up, and automatically retries in 30 days. It scales up the fraction of Arm instances of the application task by task, job by job, and cell by cell following Google’s production

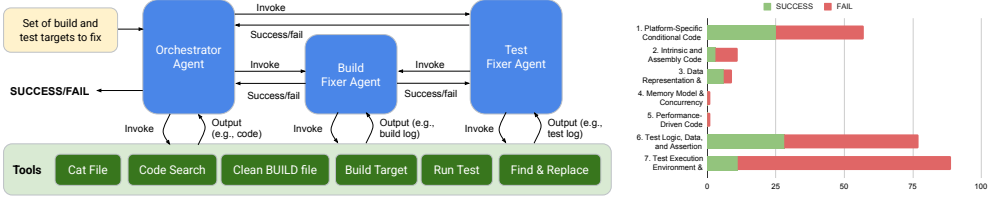


Fig. 4. Agentic flow and the success rate of the agent.

principles to limit SLO risk. CHAMP is not needed for new microarchitecture deployments (either x86 or Arm), as the behavior, performance differences, and associated issues are relatively minor. However, auto-qualification of Arm binaries was necessary due to the increased incidence of issues.

Using CHAMP, it is no longer necessary to manually shepherd every binary through qualification. Instead, after updating project configurations to build Arm releases, this process is now automatic.

5.2 Reliability of the Automation Approach

Combined, these tools allow for a mostly-automated approach where LSCs enable Arm for different builds and releases, which are then automatically qualified using CHAMP. To understand the stability of this approach, we analyze LSCs targeting a standardized release management system. These LSCs modified release configurations to bring this system’s percentage of Arm-qualified applications from 4.8% to 59.6%. The rate of applications that were rolled back in early testing was 1.8% (which dropped to 1% after fixing bugs), and less than 0.8% in the final phase.

Early in the migration, after ≈ 300 MPMs, we had a 5% refusal rate (code owners deciding not to migrate). During scale up, this dropped to 0.6% after ≈ 600 additional MPMs. In the final phase, the commits were globally approved, with no refusal. We found that acceptance rate was strongly influenced by careful workload targeting, users gaining trust in the automation, and messaging that anticipated worries and objections.

6 Automation of ISA Migrations with AI

While LSCs and CHAMP automate a large part of the porting process, there are limits to this approach. They can edit build and configuration files, as well as automatically qualify Arm binaries for deployment. However, standard LSCs are fixed-function pipelines. They are not flexible to respond to unexpected errors or other issues that occur at any stage of the process, be it during testing or in production.

Modern generative AI techniques represent an opportunity to automate the remainder of the ISA migration process. We built an agent called *CogniPort* which aims to close this gap. CogniPort operates on build and test errors. If an Arm binary does not build or a test fails at any point in the process, the agent steps in and aims to fix the problem automatically.

The agent consists of three nested agentic loops (Figure 4). Each loop executes the LLM from Section 4 to perform one step of reasoning, followed by a tool invocation—i.e., a function call (the loop terminates once the agent emits a special ‘finish’ call). This tool executes, and its outputs are attached to the agent’s context. For example, there are tools for building and returning the build log, running test(s) and returning the test log, and running a tool that fixes errors in BUILD files. The agent also has tools to search through code and make modifications.

The outermost agent loop is an *orchestrator* that repeatedly calls the *build fixer agent* and/or the *test fixer agent* depending on the state of the workspace. The build fixer agent tries to build a given

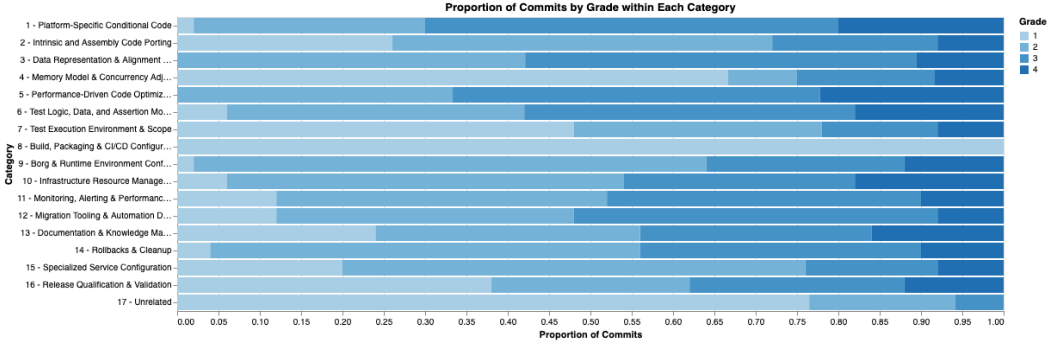


Fig. 5. AI-assessed automatability of each category (1 = trivial, 5 = probably unsolvable), as well as actual fraction of commits and LoCs in each category that were generated using automated tools.

target and makes modifications to files until the target builds successfully. The test fixer agent tries to run a given test and makes modifications until the test passes. In both cases, the agent can time out via a step limit or give up early by calling ‘finish’.

To evaluate the agent, we take historic commits from our data set, revert them and then evaluate whether the agent is able to fix them. We note that not all of our categories are suitable for this approach—it only applies to Code & Test Adaptation (categories 1-8). To evaluate the agent, we further narrow down our data set by only picking commits that can be cleanly reverted and that have identifiable build or test targets. This results in a benchmark set of 245 commits.

We see that the overall success rate is 30%, with test fixes, platform-specific conditionals, and data representation fixes having the highest success rates. Memory model, test execution environment, and performances fixes are most difficult (though based on a very small sample size). Overall, however, this indicates that AI achieves a reasonably high success rate. We note that we consider these results *directional*: While we analyzed an arbitrarily chosen subset of outputs to ensure the validity of results and confirm that we are not observing recitation from training, the evaluation is not perfect and may miss cases where, e.g., a fix is incorrect but not caught by a test or where there is other information leakage (e.g., a subsequent commit made the original fix easier or reverting only the commit itself makes it easier to root cause an issue than if the entire fix was reverted).

7 Discussion & Research Challenges

Overall, we see that ISA migrations involve a wide range of different tasks. Many of these tasks are highly automatable. BUILD file and configuration changes are almost fully automatable, while code changes and tests are partially automatable with AI. In addition, there are many changes in areas like build and test infrastructure, spinning up a new hardware platform whether or not it is a new ISA (e.g., categories 11 and 15), or deprecating old code (category 14) that needed generalizations for multiarch support, but these scale across many architectures and will not be required again.

This refutes the conventional wisdom that the main challenge of ISA migration is translation, and also that migrating a large software ecosystem to a new ISA is a prohibitively large amount of work, particularly with modern AI. This raises the question: What are the remaining challenges, and are there research opportunities in closing the remaining gaps (RQ3)?

To answer this, we analyze our data set of commits to identify changes that are challenging to automate. Once again, we use AI. We used our LLM to assess how automatable different categories are by sampling up to 50 commits in each category and using an LLM to grade them (1 = trivially automatable, 5 = difficult, even for advanced AI). This is not a conclusive result, but directionally

tells us how difficult each category is, according to the LLM itself (Figure 5). By manually inspecting the outputs, a picture emerges of which changes remain the most challenging.

First, we find that the LLM confirms the categories that are already automatable today—i.e., BUILD and configuration files, test execution environment—as automatable. Second, categories 1-7 (code & test adaptation) stand out as problems that have a significant fraction of commits ranked as 3 and 4—indicating problems that are hard but not impossible. Examples of these include:

- **ISA-Specific Vector Code:** Writing ISA-specific, performant vector code is a hard problem, and is actively investigated by the research community [25]. While current AI can translate simple routines, generating complex kernels exposes a complex search space that goes beyond a simple build-repair loop.
- **Deep Performance Optimizations:** Performance optimizations sometime require major refactors, algorithmic changes and intrinsics. There is significant existing work towards applying LLMs to these problems. At Google, we have a system called ECO that is used for automating some of these optimizations [16], including for code running on Arm.
- **Difficult Corner Cases:** We saw a number of corner cases that require obscure knowledge beyond the code itself. For example, we saw commits that worked around an Arm compiler bug, addressed a hash function behaving differently on Arm and x86, and fixed bugs that were unrelated to the Arm migration but were not previously triggered. It is plausible that an agent that can search documentation and the wider web could perform better on these.
- **Performance Tuning:** Hyperparameters and feedback directed optimization (FDO) profiles sometimes have to be regenerated for a new platform. This is potentially automatable, but requires an agent to be able to run workloads and perform performance measurement.

Meanwhile, we also found a number of examples where it is difficult to tell whether they are automatable using AI or whether they fundamentally require human involvement:

- **Multiarch tooling:** A significant portion of this work includes implementing the automation itself (e.g., CHAMP), as well as simulation tools and dashboards. While AI can help in the development flow of these components, many commits involve addressing feature requests by users, and thus require human involvement. On the other hand, this work only needs to be done once and is not required for future ISA migrations.
- **Resource provisioning:** These are changes to configuration that follow human engineers installing hardware in data centers and subsequent management by Borg. They should not be performed by AI, and are being obviated via improvements to Borg and adjacent systems.
- **Documentation:** LLMs have already proven useful at generating some kinds of documentation [10], and future LLMs may improve the ability of maintaining it and generating user-focused narrative documentation (and chat agents) that require context beyond code.

Taken together, this demonstrates that there are opportunities for further closing the gap and that future ISA migrations may require a limited amount of manual work, mostly focused on making new hardware available and adding the new ISA to the automated multiarch tooling.

8 Conclusion

By analyzing a large-scale ISA migration at Google, we refute several long-standing assumptions about ISA migrations. First, code translation is only a small portion of the ISA migration, and mainly focuses on intrinsics and vector code. Second, merely recompiling available code is not sufficient. Third, the tasks required for an ISA migration are multifaceted, with no single task dominating. Fourth, many of these tasks are highly automatable, particularly using modern AI techniques. Finally, even with AI, there remain a number of challenges that currently require human involvement and represent opportunities for future work on AI for ISA migration.

Acknowledgments

We thank Arvind Sundararajan, Dushyant Acharya, Ahmed Alansary, Shelah Ameli, Owen Anderson, Sterling Augustine, Sushmita Azad, Nupur Baghel, Antoine Baudoux, Patrick Bellasi, Vincent Belliard, Kyle Berman, Paul Bethe, Gopu Bhaskar, Raymond 'Princess Sparklefists' Blum, Marshall Bockrath, Harsha Vardhan Bonthalala, Lexi Bromfield, Jean-Luc Brouillet, Eric Burnett, Marcelo Cataldo, John Cater, Kristine Chen, David Cheng, Ilya Cherny, Saahithi Chillara, Rob Chilton, Chandrakanth Chittappa, Brian Chiu, Daniele Codecasa, Eduardo Colaço, Pavithra Dankanikote, Nicolo Davis, Rumeet Dhindsa, Zhuoran Diao, Bartosz Dolecki, Ian Dolzhanskii, Pat Doyle, Elian Dumitru, Ali Esmaeeli, Samuel Foss, Ákos Frohner, Neeharika Gonuguntla, Shruti Gorappa, Russell Gottfried, Manoj Gupta, Benjamin Gwin, Yanyan Han, Jitendra Harlalka, Milad Hashemi, Tim Henderson, Daisy Hollman, Jung Woo Hong, Jiawei Huang, Jin Huang, Talha Imran, Victoria Juan, Pranav Kant, Lera Kharatyan, Joonsung Kim, Danial Klimkin, Sree Kodakara, Avi Kondareddy, Danila Kutenin, Gregory Kwok, Pavel Labath, Leon Lee, Sungkwang Lee, Li Li, Sha Li, Xu Li, Zhongqi Li, Jianyi Liang, Kevin Liston, Haiming Liu, Li Liu, David Lo, Sean Luchen, Albert Ma, Laura Macaddino, Anthony Mai, Jennifer Mansur, Simon Marchuk, David Margolin, Alexander Midlash, Dominic Mitchell, Karthik Mohan, Albert Morgese, Maksym Motornyy, Katherine Nadell, Ngan Nguyen, Denis Nikitin, Stoyan Nikolov, Nicolas Noble, Chester Obi, Andri Orap, Habib Pagarkar, Dasha Patroucheva, Sabuj Pattanayek, Vaishakhi Pilankar, Saranyan Vangal Rajagopalan, Daniel Rall, Majid Rasouli, Salonik Resch, Alberto Rojas, Annie Rong, Jesse Rosenstock, Michal Sapinski, Yashnarendra Saraf, Aaron Schooley, Alexander Schrepfer, Manish Shah, Alexander Shaposhnikov, Stan Shebs, Guangyu Shi, Oscar Shi, Santanu Sinha, Anna Sjövall, Ben Smith, Justin Smith, Jairaj Solanke, Fangrui Song, Raman Subramanian, Xenia Tay, Toni Thompson, Dima Tsumarau, Cassie(Yitong) Wang, Tommy Wang, Shu-Chun Weng, DJ Whang, Hailong Xiao, Andy Xu, and Jason Yuan for their contributions to Google's Arm porting efforts and the work described herein.

References

- [1] 2025. RISC-V - Debian Wiki — wiki.debian.org. <https://wiki.debian.org/RISC-V/>. [Accessed 11-09-2025].
- [2] E.R. Altman, D. Kaeli, and Y. Sheffer. 2000. Welcome to the opportunities of binary translation. *Computer* 33, 3 (2000), 40–45. doi:10.1109/2.825694
- [3] Apple. [n. d.]. About the Rosetta translation environment. <https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment/>. [Accessed 10-09-2025].
- [4] Babbage. 2023. Apple's Mac Transitions : 68k to PowerPC to Intel to Apple Silicon. The Chip Letter (Substack). <https://thechipletter.substack.com/p/apple-transitions-68k-to-powerpc> Accessed: 2025-09-10.
- [5] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. doi:10.1145/3360585
- [6] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site reliability engineering: how Google runs production systems*. " O'Reilly Media, Inc".
- [7] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara, and Stephen Thorne. 2018. *The site reliability workbook: practical ways to implement SRE*. " O'Reilly Media, Inc".
- [8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA) (OSDI '06). USENIX Association, USA, 15.
- [9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*.
- [10] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A Comparative Analysis of Large Language Models for Code Documentation Generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software* (Porto de Galinhas, Brazil) (AIware 2024). Association for Computing Machinery, New York, NY, USA, 65–73. doi:10.1145/3664646.3664765

- [11] Google Cloud Events. 2025. Shift down: A practical guide to platform engineering. YouTube. <https://www.youtube.com/watch?v=FqBuYUeJkxE> Accessed on 2025-09-22.
- [12] Redha Gouicem, Dennis Sprockholt, Jasper Ruehl, Rodrigo C. O. Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 107–122. doi:10.1145/3567955.3567962
- [13] Svilen Kanev, Juan Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2014. Profiling a warehouse-scale computer. In *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.
- [14] Jan Keller and Jan Nowakowski. 2024. *AI-powered patching: the future of automated vulnerability fixes*. Technical Report.
- [15] Justin Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugam, and Yuri Volobuev. 2024. BigLake: BigQuery’s Evolution toward a Multi-Cloud Lakehouse. In *SIGMOD*.
- [16] Hannah Lin, Martin Maas, Maximilian Roquemore, Arman Hasanzadeh, Fred Lewis, Yusuf Simonson, Tzu-Wei Yang, Amir Yazdanbakhsh, Deniz Altinbüken, Florin Papa, Maggie Nolan Edmonds, Aditya Patil, Don Schwarz, Satish Chandra, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. 2025. ECO: An LLM-Driven Efficient Code Optimizer for Warehouse Scale Computers. arXiv:2503.15669 [cs.SE] <https://arxiv.org/abs/2503.15669>
- [17] Andreas Lykke and Jesper Borlum. 2025. Adopting Arm at Scale: Bootstrapping Infrastructure. <https://www.uber.com/blog/adopting-arm-at-scale-bootstrapping-infrastructure/>. [Accessed 10-09-2025].
- [18] MulticoreWare. 2024. *Why Porting Applications Across Architectures Isn’t Simple*. MulticoreWare Inc. <https://multicorewareinc.com/why-porting-applications-across-architectures-isnt-simple/>
- [19] Peter Phillips and George Phillips. 2003. No Source Code? No Problem! What if you have to port a program, but all you have is a binary? *Queue* 1, 6 (Sept. 2003), 50–57. doi:10.1145/945131.945155
- [20] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59 (2016), 78–87. <http://dl.acm.org/citation.cfm?id=2854146>
- [21] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) (USENIX ATC’12). USENIX Association, USA, 28.
- [22] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer – data race detection in practice.. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. NYC, NY, U.S.A., 62–71. <http://doi.acm.org/10.1145/1791194.1791203>
- [23] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wu Yang. 2012. LLBT: an LLVM-based static binary translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Tampere, Finland) (CASES ’12). Association for Computing Machinery, New York, NY, USA, 51–60. doi:10.1145/2380403.2380419
- [24] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. San Francisco, CA, USA, 46–55.
- [25] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K. Lahiri. 2025. LLM-Vectorizer: LLM-Based Verified Loop Vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) (CGO ’25). Association for Computing Machinery, New York, NY, USA, 137–149. doi:10.1145/3696443.3708929
- [26] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems* (EuroSys). Bordeaux, France.
- [27] Titus Winters, Tom Manshreck, and Hyrum Wright. 2020. *Software engineering at google: Lessons learned from programming over time*. " O’Reilly Media, Inc."