# CloudBuild: Microsoft's Distributed and Caching Build Service

Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan,
Erik Mavrinac, Wolfram Schulte, Newton Sanches, Srikanth Kandula
Microsoft

## ABSTRACT

Thousands of Microsoft engineers build and test hundreds of software products several times a day. It is essential that this continuous integration scales, guarantees short feedback cycles, and functions reliably with minimal human intervention. This paper describes CloudBuild, the build service infrastructure developed within Microsoft over the last few years. CloudBuild is responsible for all aspects of a continuous integration workflow, including builds, test and code analysis, as well as drops, package and symbol creation and storage. CloudBuild supports multiple build languages as long as they fulfill a coarse grained, file IO based contract. CloudBuild uses content based caching to run build-related tasks only when needed. Lastly, it builds on many machines in parallel. CloudBuild offers a reliable build service in the presence of unreliable components. It aims to rapidly onboard teams and hence has to support non-deterministic build tools and specification languages that under-declare dependencies. We will outline how we addressed these challenges and characterize the operations of CloudBuild. CloudBuild has on-boarded hundreds of codebases with only man-months of effort each. Some of these codebases are used by thousands of developers. The speed ups of build and test range from 1.3× to 10×, and service availability is 99%.

## Keywords

Build Systems, CloudBuild, Distributed Systems, Caching, Specification Languages, Operations

## 1. INTRODUCTION

Microsoft is rapidly adopting an agile methodology to enable a much faster delivery cadence of days to weeks. For instance, Office 365, Visual Studio Online and SQL Azure have release cycles ranging from 3 months to 3 weeks. On the extreme end, Bing's website ships several times a day.

Delivering rapidly requires that builds, which are at the core of the inner loop of any engineering system, are fast and reliable. This challenge has given us an opportunity to design a new in-house infrastructure for continuous integration known as CloudBuild. Its design was motivated by these needs:

*ICSE '16 Companion, May 14 - 22, 2016, Austin, TX, USA*

ISBN 978-1-4503-4205-6/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889222

- execute builds, tests, and related tasks as fast as possible,
- on-board as many product groups as effortlessly as possible,
- integrate into existing workflows,
- ensure high reliability of builds and their necessary infrastructure,
- reduce costs by avoiding separate build labs per organization,
- leverage Microsoft's resources in the cloud for scale and elasticity, and lastly
- consolidate disparate engineering efforts into one common service.

We found it difficult to simultaneously satisfy these requirements. Most build labs, for instance, have only a few dependencies on other systems; they connect to one version control system or build drop storage solution. To avoid separate build labs, CloudBuild supports the union of all lab dependencies and workflows. However, the consequent interoperability issues can lead to lower reliability. As another instance of conflicting requirements, product groups have optimized their builds (and build labs) over many years, often exploiting their particular product architecture and workflow to optimize for incremental and parallel builds. Since CloudBuild offers a generic cloud-based solution across the various product groups, our current speedup improvements are less dramatic (not 100× for example) relative to the highly optimized custom systems.

When architecting CloudBuild, we had to make a crucial decision. We found that existing build tools and specification languages were unsuitable for cached and parallel execution. In particular, the languages would under-specify dependencies and the tools produced non-deterministic outputs. Hence, while single machine (or single threaded) execution would run correctly, distributed execution and caching of task outputs could lead to unpredictable behavior or limited speedup. A *clean* solution would be to design a new build specification language, rewrite all build specifications in this language and force tools to respect additional constraints. Such efforts are underway [5, 27]. CloudBuild, however, is our *quick* solution. That is, here we show how to onboard existing specification languages (and tools) on to a cloud-based parallel and cached build system. The key advantage with this choice is the speed of onboarding. Builds and tests can be sped up and disparate labs can be consolidated into the cloud quickly. We also found CloudBuild to be able to cover a wide range of tools and specifications. The cost, however, is the rather heuristic nature of CloudBuild: our goal is not to function correctly in every possible setting (of specification and tool behavior). We mitigate this with customer education and hardening of CloudBuild to cover frequently recurring issues.

We believe that our design decision has been successful. CloudBuild is currently being used by more than 4000 developers in Bing, Exchange, SQL, OneDrive, Azure and Office. It executes more than 20K builds per day on up to 10K machines spread over several data centers. Build and test speeds typically improved by 1.3 times to 10 times compared to earlier lab based systems, sustaining an availability of 99.9%. Reliability, defined as the number of requests that are served without a CloudBuild internal error, is better than 99%. CloudBuild connects with three different version control and four binary storage systems. REST APIs and Azure's Service Bus [35] are used for integration into multiple workflow systems.

While CloudBuild's service design follows existing designs for large scale data-parallel systems [3, 8, 41], CloudBuild's build engine is unique in that it allows the execution of arbitrary build languages and tools, even in the presence of under-specified dependencies and non-deterministic tools.

Given a build request, CloudBuild evaluates build specifications, typically expressed in Make or MsBuild files, to infer project-to-project level dependencies. This means that CloudBuild's unit of scheduling is a coarse grained project. While not as fine grained as other systems like Google's Bazel [5], the approach allows greater reuse of pre-existing build specifications, without having to comprehend all the runtime semantics of the underlying build tools.

CloudBuild can determine whether to execute a given project or to reuse outputs captured during an earlier execution, and stored in a system-wide cache. The determination is based on the evaluation of the pertinent source files and parent projects which is summarized into a cache lookup key for the project. The approach to compute this key ensures that the build output is deterministic and consistent with other projects in the same build.

Finally CloudBuild schedules work, i.e. the tasks for each project that needs to be built, in a location-aware manner over multiple worker machines. Locality here implies two things: freshness of the cache at the machine as well as the freshness of their enlistment from source control. Without this locality, the build-preparation-time to configure the machine with the right set of SDKs and sources before it can participate in the build will be substantial. CloudBuild uses standard DAG scheduling logic. The tasks themselves have heterogeneous resource demands. The dependencies between tasks are more arbitrary than in data parallel clusters (no shuffles for example). Furthermore, unlike the case of data-parallel jobs where outputs are much smaller than the input (e.g., queries in TPC-DS [16] or TPC-H [17]), the outputs of the median build job are quite large since many libraries and executables are generated and packaged. This requires CloudBuild to more carefully pipeline the fetching of inputs with the execution of tasks.

For a given task, CloudBuild creates a dedicated file system directory structure, on which build tools are to execute. We call this a *sandbox*, and it addresses reliability issues due to under-specified dependencies and allows for multi-tenant builds. This is similar in intent to Unix's chroot jail [34], though it is implemented at the application level. As we will explain later, cache and sandbox play together to guarantee the absence of "Frankenbuilds", i.e. builds where outputs from different build jobs can combine in inconsistent ways due to cache re-use.

Despite allowing so much flexibility for specifications, the resulting CloudBuild system is fast and reliable. At Microsoft, we have observed that 70 to 90% of build artifacts can be shared; this number varies across code bases. And CloudBuild's overhead for distributed builds is small—with enough builders, 95% of builds take no more than 1.05× the total time cost of the most expensive path in the directed task graph described in the build specifications.

CloudBuild's main contribution is thus to allow for reliable, fast builds in the presence of under-specified dependencies and non-deterministic tools. CloudBuild's engineering contribution is that it is easy to use, scales to varying demands and integrates well into Microsoft's engineering fabric.

In the rest of this paper, we describe the design of CloudBuild (§2–§3). Experiential anecdotes are offered throughout the design sections to clarify the various design choices. We then characterize some operational aspects of the CloudBuild service (§4.1) and share onboarding and live-site experiences (§4.2–§4.3). We conclude with a discussion of related work (§5) and lessons learnt.

## 2. PRIMER ON BUILD SERVICES

Today, build systems such as Make, Maven, Graddle and MSBuild execute on one machine, typically the developers' desktops or laptops. As the size of code bases grow, these tools can take several hours for a full build. Delta builds, that is, rebuilding after changes, are faster. However, continuous integration workflows often require packaging binaries and/or running exhaustive batteries of tests.

To speed up the build workflow, we note that there is substantial opportunity to parallelize. While dependencies do exist, they only form a partial order over the build tasks and many build tasks are unordered with respect to each other. As a consequence various efforts are under way to provide multi-threaded builds and distribute the build tasks over many nodes.

Furthermore, there is an opportunity to reuse the outputs of previous builds, similar to delta builds. Build reuse arises fundamentally due to highly modularized software engineering practices. Teams of developers work on specific components and can reuse the latest binaries for the rest. Even across source control branches, there is a high likelihood of sharing due to the use of shared libraries for commonly used or important functionality.

In this context, CloudBuild offers a distributed and caching build service. We offer a quick summary of relevant background.

### 2.1 Source Control

Enlistments are (partial) materializations of source files from a version control system. CloudBuild supports different version control systems , including Source Depot [14], Git [7] and TFVC [15]. New changes registered in source control are a potential trigger of new builds for verification.

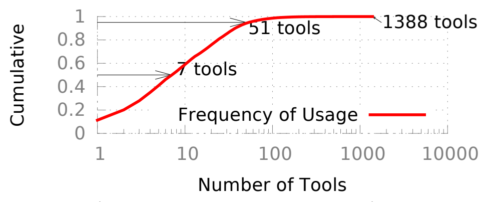
### 2.2 Build Specifications and Tests

Build specifications describe the items that are to built, the dependencies between items and the actions to be taken per item. Example specifications include Makefiles and project files in MSBuild [9]. A codebase typically has multiple file-system directories with many specification files, typically one specification file per *build target*. A build target is a logical entity comprising a binary file and ancillary files (e.g., debugging symbols).

Dependencies between build specifications are expressed by referencing the paths of other build specs, or the paths of files produced by the targets.

Build specifications also describe the details of *how* to build a particular target. This is done either explicitly or by invoking *rules* defined in other files. For example, the rule to compile a dynamically linked library from C++ source code would specify a set of steps using any number of tools (compilers, linkers, etc.) as well as command-line options. Compiler distributions and SDKs often reference rule files that can be reused. However, teams or organizations can customize these rule files to satisfy technical or policy requirements. Further, many aspects of the common rules can be parameterized, extended, or changed from within a build spec.

While CloudBuild supports in principle any build language that declares which sources are read, which project-dependencies they have, and which output directories they use, special handling is



| Tool | Rel. Freq. | Tool | Rel. Freq. |
|---|---|---|---|
| perl | 0.11 | nmake | 0.03 |
| mkdir | 0.09 | buildoutput | 0.03 |
| echo | 0.08 | copy | 0.03 |
| cmd | 0.07 | dfmgr | 0.03 |
| exportls | 0.06 | signtool | 0.02 |
| cl | 0.05 | link | 0.02 |
| binplace | 0.04 | getfilehash | 0.01 |

Figure 1: CDF of tools by usage and the names of the most frequently used tools across all of Codebase A's build specifications.

given to the two major build languages currently in use at Microsoft, nmake [10] and MSBuild [9].

As described in the introduction, CloudBuild is also responsible for the execution of tests that follow compilation and linkage. The test execution is distributed in the same way as the build steps. CloudBuild supports tests written in VSTest [18] and its ecosystem of adapters for different languages and unit test frameworks, including nUnit and xUnit. CloudBuild also supports various code analysis frameworks.

## 2.3 Build Tools

Build specifications happen to use many kinds of tools. Figure 1 lists the top few tool names and their frequency in all of the build specifications of codebase A.[1] A total of 1388 different tools are used in codebase A. The distribution of usage is heavy tailed; 50 tools (3.6% of all unique tools used) contribute 95% of the overall usage. However, several of the most frequently used tools are script invocators: `perl` and `cmd` for example. The latter is Windows' shell invocator similar to `sh` or `bash`. There are also recursive calls such as those to `nmake`. Hence, this characterization of tool frequency under-estimates the diversity of tools. Yet, it suffices to show that a wide variety of tools are used by typical codebases.

As mentioned in the introduction, a key goal of CloudBuild is to onboard existing build specs with minimal additional effort. Hence, CloudBuild has to support all of these tools. As some of the code bases are quite old, the associated tools are equally old. But they are so widely used that rewriting them is not easy. Further, several tools depend on their environment in idiosyncratic and under-specified ways that makes them a challenge to parallelize. Many of these tools are also not deterministic, so reinvoking the tools could create different outputs; a typical cause of non-determinism is that tools may have timestamps or randomly generated bits in their output.

## 3. DESIGN OF CloudBuild

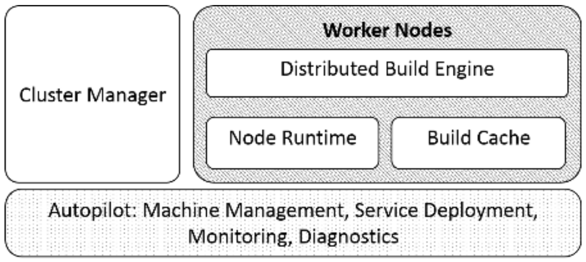
## 3.1 Design Principles

The primary goal of CloudBuild has been to reduce build times and increase build reliability for as wide a range of Microsoft teams as possible with as little investment as necessary. To achieve that, CloudBuild follows the following principles:

[1]While we anonymize the names of codebases, we will relate various aspects of the same codebase by consistently referring to them with the same letter.

| Codebase | Speed-up | Onboarding Time | # Devs |
|---|---|---|---|
| B | 1.6 | 3mm | 1200 |
| C | 7.3 | 3mm | 350 |
| D | 3.1 | 2mm | 100 |
| E | 3.8 | 2mm | 200 |

Table 1: Shows the ratio between the build times with and without CloudBuild. Original build times ranged from 1 to 5 hours. The table also quantifies onboarding effort (mm denotes a man-month), and the number of committers to each codebase.



Cluster Manager
Worker Nodes
Distributed Build Engine
Node Runtime
Build Cache
Autopilot: Machine Management, Service Deployment, Monitoring, Diagnostics

Figure 2: An architecture diagram for CloudBuild

- **Commitment to compatibility:** CloudBuild was designed to replace existing build pipelines, and as such should be substantially compatible with pre-existing build specifications, tools, and SDKs. Given the size, age, and heterogeneity of Microsoft codebases, approaches that require rewrites or major refactoring of build specifications were considered prohibitive by most teams.

- **Minimal new constraints:** The only prescriptions made and changes demanded by CloudBuild are those that would otherwise hinder parallelizing the build, or make caching ineffective. The system provides mechanisms to assist with onboarding legacy specifications and, once onboarded, to maintain those code bases within those constraints.

- **Permissive execution environment:** Safety under adversarial inputs is not a goal. That is, it is possible to write build specifications or tools that result in unpredictable build behavior. However when used as intended, CloudBuild guarantees deterministic build output. Further, CloudBuild is a multi-tenant service and limits the impact on a customer from others that share the same caches or servers.

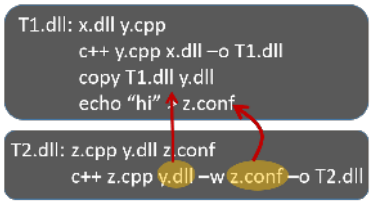
Table 1 provides data from experience to quantify the outcomes of these principles. Each row is a codebase with hundreds of developers that now runs on CloudBuild. Speed-up is very difficult to compute because the underlying build specifications have evolved substantially after onboarding. In every case, the new build specs are much more comprehensive than the historical ones that pre-date CloudBuild. Nevertheless, we compare the build time before onboarding onto CloudBuild vs. the latest build times. The table shows that the speed-ups are sizable. The table also shows that the time to onboard each codebase was a couple man-months. It would have not been possible to onboard as rapidly, i.e., move to a distributed cached build execution, if CloudBuild did not adhere to the above principles.

## 3.2 Architecture Overview

CloudBuild builds on top of Microsoft's cluster management system, Autopilot [33]. Autopilot (AP) manages the deployment of services as well as the monitoring of hardware, operating system and service health. AP applies automatic recovery techniques to deal

| codebase | LOC | build targets | Targets needing... | | Specification files... | | |
|---|---|---|---|---|---|---|---|
| | | | input annot. | output annot. | number | type | parse time |
| B | $2.8 \times 10^7$ | 8263 | 2587 | 214 | 11300 | mostly nmake | 48.8s |
| C | $4.8 \times 10^6$ | 2713 | 178 | 89 | 3539 | mostly nmake | 36.6s |
| D | – | 2076 | 269 | 112 | 2267 | mostly msbuild | 53.7s |

Table 2: Build Targets and those that required explicit dependency annotations. Codebase B uses an automatic build migration tool that leads to a higher-than-typical number of annotations. The table also lists the number of specification files parsed by CloudBuild for dependency inference; this process takes only about one minute, even for some very large codebases.



```
T1.dll: x.dll y.cpp
        c++ y.cpp x.dll –o T1.dll
        copy T1.dll y.dll
        echo "hi" > z.conf

T2.dll: z.cpp y.dll z.conf
        c++ z.cpp y.dll –w z.conf –o T2.dll
```

Figure 3: Example illustrating hidden dependencies that break a parallel or cached build.

with hardware and software faults, and aims to maintain high service availability during maintenance and software upgrades.

CloudBuild consists of a large number of *worker* machines managed by small set of collaborating processes collectively referred to as the *cluster manager*. Once Autopilot deploys the CloudBuild *runtime* to a machine, the worker publishes its presence and state to a Zookeeper [4]-like service. This service maintains the list of available enlistments, machine allocation to ongoing builds, and machine readiness for new builds.

The cluster manager can shrink or grow the number of workers or enlistments per codebase. Workers interact with source control (§2.1) and package managers [11] to download the necessary content into the enlistments.

The cluster manager accepts build-initiation requests. The input includes a build definition that lists configuration options such as: output type (e.g., "build for 64-bit architecture"); parallelization settings (e.g., "use between 5 and 7 nodes"); routing preferences (e.g., "prefer datacenters in US East region"). The input also includes per-request options (e.g., code base revision to be built). Build jobs are logged and added to a *build queue*.

Build jobs remain in a pending state until they are matched up with a sufficient number of workers with matching enlistments. If many workers are available, the cluster manager selects those that performed similar work recently, under the assumption that their enlistments (and local caches) will be freshest. Chosen workers are asked to update their enlistments to the desired source and package revisions. Such preparation happens in parallel. Once enough workers are ready, the build job begins by spawning a *build coordinator* and multiple *builder nodes*.

**Architectural differences:** The architecture is largely similar to other systems that schedule dependent sets of tasks such as Yarn [8], Tez [3] and Cosmos [41]. A key difference thus far is the early choice of machines for builds (maintaining a map from machines to enlistments and caches and using locality). In contrast, data-parallel systems place tasks on *any machine* on the cluster. Since machines have to be prepared (pull relevant sources from source control as well as SDKs and other packages needed to build), CloudBuild is more careful in picking machines for builds to amortize the prep cost and improve cache hit rate. Furthermore, data-parallel systems need not infer hidden dependencies (§3.3) since they are perfectly specified by user queries and do not isolate various tasks at runtime (§3.4). Differences in scheduling are described later (§3.6).

## 3.3 Extracting Dependency Graph

The dependency graph of a build determines the order of execution. Existing build languages such as nmake [10] and MSBuild [9] allow dependencies to be under-specified. Consider the example in Figure 3. The syntax is inspired by make. Each target has a list of explicitly declared inputs and a corresponding set of commands. At first blush, the two targets appear unrelated. However, `T2.dll` uses `y.dll` and `z.conf` which are produced as side-effects of executing `T1.dll`. Sequential builds, which the build spec was written for, will succeed. However, a parallel engine that builds `T2.dll` first will either fail or produce outputs using stale versions of the dependencies. A distributed engine that builds `T2.dll` on a different machine will behave similarly (i.e. either fail or use stale dependencies). Similarly, cached builds would not know to fetch the correct inputs.

The fundamental problem is that widely-used build languages do not enumerate all of the inputs and the outputs of build targets. Neither do they explicitly call out the side-effects and resource dependencies of build commands.

A key component of CloudBuild is its ability to automatically infer hidden dependencies. It does so in a pre-processing step before the build. CloudBuild has a large set of rules to apply against the build specification ASTs that identify patterns for particular build languages and SDKs, and then emit the anticipated inputs and outputs. In some cases, invocations of external commands are also parsed. For example, when `robocopy.exe` [12] commands are used, CloudBuild parses its command line to infer the inputs and outputs. There are invocation parsers for many of the tools listed in Figure 1. These parsers reduce porting overhead; they also substantially reduce the need for manual annotations that are described next.

When implicit dependencies cannot be inferred, users can add annotations. These are designed to blend into the build specification language in a way that is transparent, in case these specifications are executed outside CloudBuild. There are also rules to detect constructs that CloudBuild does not understand, such as invocations of external scripts; the system alerts the user, requesting that annotations be added to describe their inputs and outputs.

In some cases, build tools use runtime context to determine their inputs. Such context is unavailable in the pre-processing phase. Where possible, CloudBuild's parser over-approximates the list of inputs: it may include all files that can be referenced via all branches of the runtime logic; it may include whole directories, where the exact set of accessed files is known only at runtime.

The task dependency graph is constructed with one node per target. An edge is added between two nodes $n_1$, $n_2$ whenever there is overlap between $n_1$'s output and $n_2$'s inputs. Unresolved inputs, that is those not found in source control and not output by any target, are flagged as errors. Cycles in the dependency graph are also flagged as errors.

Besides the task nodes emitted for each target, CloudBuild may emit additional nodes for derived tasks, including: execution of tests and static analysis tools, aggregation of code coverage and analysis reports, and uploading of build outputs to durable cloud storage.

Table 2 offers some experience numbers from three large code-

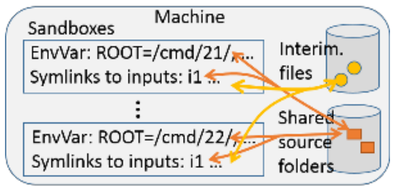


Figure 4: Example illustrating sandboxes

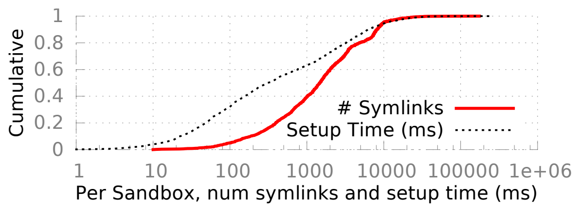


Figure 5: Quantifying sandbox size (number of symbolic links) and the time to setup sandboxes. The gap between the CDFs is due to our recycling of sandboxes. Sandboxes of medium size needs a few thousand symbolic links and takes about 0.3s to setup. Some targets (MSIs) have > $10^5$ inputs.

bases that run on CloudBuild. The data is for a full build of each codebase. We see that parsing the specifications to infer dependencies takes only about one minute, even for large codebases. The parsing happens during a pre-processing step. We also see that the codebases have thousands of targets and about 1.2 times as many specification files. Recall that there can be many common or project-specific specifications that list build rules, in addition to those that refer to the targets. The table also shows the number of cases that require explicit annotation. Most of the work is handled by automatic inference. The remainder are fixed by annotations. Codebase B has a more-than-typical number of annotations because it uses an auto-annotation tool.

## 3.4 Sandboxing Build Tasks

CloudBuild executes each task in a *sandbox*. A sandbox is a directory with symbolic links for all of the task's inputs, where inputs include sources from the enlistment and the outputs of parent tasks. Additionally, the sandbox redefines a number of process environment variables which refer to the particular locations in the enlistment. The redefinition steers the task to locations within the sandbox for inputs, outputs and scratch folder locations. Figure 4 illustrates this design.

Sandboxes are introduced for three main reasons: dependency verification, output collection, and isolation.

**Dependency Verification:** As we saw above, build specifications can have dependencies that are not explicitly declared. The inferences of dependencies made by CloudBuild may therefore be incomplete (§3.3). Mistakes in dependencies will lead to false cache fingerprints and hence false cache hits. To solve this, CloudBuild populates a task's sandbox with only its explicit and inferred dependencies. Thereby, CloudBuild ensures that any missed dependencies will be runtime errors – typically "file not found". Such errors are surfaced to the user, the sandbox's outputs are discarded, and the child tasks are not executed.

Since it is often prohibitively expensive to create symbolic links to every single file, CloudBuild may create symbolic links to the directories that contain the required files. CloudBuild uses Detours [32] binary injection to observe file system APIs, and record the actual files accessed. Accesses beyond those predicted by the dependency parser are surfaced as errors.

**Output Collection:** As also described above, build specifications do not list all of their outputs. To determine the outputs for a task at runtime, CloudBuild enumerates all directory entries in the sandbox's root directory. Any file entry that is not a symbolic link is considered part of the task's output, has its contents hashed, and is moved into a build job-wide location, outside of the sandbox.

**Isolation:** Ideally, to avoid unpredictable behavior during parallel execution, a build task should not mutate its inputs, nor should two tasks produce outputs into the same file path. In practice neither of these criteria were met in the build specifications that CloudBuild aimed to support. The sandboxing approach described above allows the system to detect such conflicting writes, and attribute them to a particular task, even when multiple tasks are executed at the same time on the same machine. CloudBuild has multiple policies to deal with such cases: it can block any such conflicts; it can allow them, provided conflicting accesses write identical content – this is sub-optimal, but deterministic; it can even allow writes with different content, when the resulting non-determinism is acceptable by the codebase owner.

**Recycling:** To minimize costly interactions with the file system, CloudBuild *recycles* sandbox directories. It retains the sandbox directories that have been created for previous build tasks as well as an in-memory representation of their file system state. When creating a new sandbox, CloudBuild uses the in-memory representation to locate the best fitting sandbox directory. Using that as a starting point, CloudBuild adds and removes symbolic links to reflect the dependencies of the incoming task.

**Experience data:** Figure 5 offers some data about sandboxes. It shows the number of symbolic links required and the time to set up as a CDF over all sandboxes. Most sandboxes require between $10^2$ - $10^5$ symbolic links. Targets with many inputs tend to be those that produce deployment packages or layouts. We see that the setup time can often be much smaller than the number of symbolic links due to our recycling of sandboxes. Most sandboxes are set up within a few seconds.

**Alternate designs:** Fundamentally, CloudBuild requires three properties from the file system environment in which the build tool executes. First, build tasks should be isolated. That is, a task should behave as if it is the only one running a machine even though many tasks are running in parallel. Second, all of the inputs and only the inputs should be available to the build task. Third, it should be possible to rapidly setup and tear down such an environment.

Operations like BSD's [34] chroot jails would have been useful, but were not available in our operating system of choice. Moreover, the build specification already expected a root directory to be established through a process-specific environment variable; this allows CloudBuild to do "application-level virtualization" by setting up sandboxes. Virtual machines are expensive to setup and teardown especially since the virtual disk for each sandbox has different files (the inputs). We also considered using Drawbridge [22], a lightweight process isolation container. While it allows for safe detouring of file system operations, the cost to on-board thousands of build tools into Drawbridge was considered prohibitive. Copy-on-write snapshots help reduce performance impact of moving data into and out of sandboxes. Tool compatibility constraints have thus far limited our ability to use a file system that supports this feature; this continues to be an area of further work.

## 3.5 Distributed Cache of Build Outputs

To reuse the work of prior builds, CloudBuild uses a distributed cache. Conceptually, the cache aims to map the entirety of the input values presented to a task execution to the output emitted by the task. If a future execution of the task is presented the same inputs, it can be skipped and the cached outputs used instead. In concrete terms, the cache is implemented as a dictionary mapping `CacheKeys`, which encapsulate the values of task inputs, to *output bags* describing the names and contents of emitted task outputs.

The following sections will explore how the `CacheKeys` are computed, the architecture of the distributed cache, and our strategies to ensure consistent content is retrieved from cache, even in the presence of multiple concurrent builds and non-idempotent tools.

### 3.5.1 Cache Data Structures

Each *output bag* consists of an `ID` plus a set of $(\mathsf{ContentHash}_f, \mathsf{Path}_f)$ tuples. The `ID` is a pseudo-random number assigned to an output bag at creation time to help with race conditions (see §4.3). Each tuple describes the content and path for a file emitted during the execution of the task described by the content bag.

Lookups into the cache are performed based on the inputs of a task. These inputs are encoded into a compact `CacheKey` as described inductively by the following equations. At a high level, `FF` denotes a file or task fingerprint, `GF` is a fingerprint of the global settings and the `CacheKey` of a target depends on the fingerprints of its inputs.

$$\begin{aligned} \mathsf{FF}_s &= \mathsf{ContentHash}_s \oplus \mathsf{Hash}(\mathsf{Path}_s) && \forall \text{ source files } s \\ \mathsf{FF}_t &= \mathsf{CacheKey}_t \oplus \mathsf{ID}_t && \forall \text{ tasks } t \\ \mathsf{CacheKey}_t &= \mathsf{GF} \oplus \left( \bigoplus_{i \in \mathsf{inputs}_t} \mathsf{FF}_i \right) && \forall \text{ tasks } t \end{aligned}$$

The first two equations show how to compute the *fingerprint* `FF` for each input source *s* and build task *t* respectively. File content hashes can be retrieved from source control if available, or computed from file content. The file path is always used as part of its fingerprint, since path values are used as inputs to certain build tasks such as those performing recursive structure-preserving directory copies. For the targets, the fingerprint is derived from their `CacheKey` and `ID`. The third equation shows how to compute the `CacheKey` for a task based on the fingerprints of its inputs (both sources and parent tasks). `GF` encodes global settings configured in a build job, presumed to be inputs for every task.

### 3.5.2 Cache Service Design

CloudBuild's caching layer is implemented in an architecture that is similar to other distributed caching systems. The mapping from `CacheKey` to output bags is maintained by the Global Cache Service (GCS). GCS is persistent and consistent. It is built using Azure Tables. Each worker node runs a Local Cache Service (LCS) which maintains a partial copy of this mapping. Together, LCS and GCS also implement a distributed content addressable store (CAS), with LCS storing replicas of each piece of content available in the cache, and GCS serving as a cluster-wide tracker of such replicas. An LCS keeps local replicas for all content that is referenced or created by tasks on that node.

Reads from the cache are first made to the local LCS. If the `CacheKey` is present on the LCS, the corresponding CAS content is returned to the caller, together with the associated $(\mathsf{ContentHash}_f, \mathsf{Path}_f)$ tuples. Otherwise, the LCS queries the GCS and if present fetches a copy of the contents of the output bag from one of the replicas to the local CAS.

Writes to the cache are handled in a write-through manner. That is, the LCS immediately pushes changes to the GCS and asks that some number of replicas be created. We discuss races in §4.3.

When in active use, popular cache items can have hundreds of replicas. Cache entries with no use are deleted after a period of time (usually a few days). At times, it is possible that items present in the GCS no longer have any valid replicas due to network partitioning, hardware failure, and a small class of maintenance activities. Such issues are detected during cache lookups, and result in the removal of the invalid cache entry.

**Experience data:** The load on the GCS is roughly $O(10^5)$ reads and $O(10^3)$ writes per second. Recall that CloudBuild uses Azure Tables to implement the GCS, thereby receiving a distributed table store with persistence and strong consistency.

## 3.6 Build Scheduling

Each build job is controlled by a build coordinator. It works by assigning individual tasks to builder nodes. For this, it tracks a list of unblocked tasks, and tries to assign these whenever resources on a builder are available.

The build scheduler uses several sets of information:

- The currently available, i.e., unblocked tasks for which all dependencies are completed.
- An estimate of a task's resource profile based on task characteristics inferred during parsing.
- Resource profile and execution time for a task determined from historical build data.
- The longest path for each task to a leaf node.
- The longest path overall, also called the critical path.

The build scheduler uses the *criticality* of a task, defined as the longest sub-path from this task to any output of the DAG. This measure is used as the priority order in which tasks are scheduled.

The scheduler only considers *actionable* tasks in the DAG. Actionable tasks are defined as the subset of tasks who have not yet run, but whose dependencies are met because their parent tasks either finished or had matching `CacheKeys`. The actionable tasks are added to a queue of executable tasks, sorted by the above priority.

The actual assignment of tasks to a specific worker depends on input locality, the task's resource profile, and availability of resources on builders. Per task – going by priority order – the coordinator iterates over the available builders, and through a combination of models tries to evaluate the fastest time to completion for this task.

Several factors influence the completion time. First, the setup phase needs to copy over inputs that are not already available on that machine. A builder that already contains a lot of the inputs will have a shorter setup time than one that needs to copy all of them over. Second, enough resources to run the task may be not available at the machine until some time in the future. The queuing delay is calculated based on the modeled execution times of tasks running already at the builder and those that are in the queue. Tasks are queued at builders with the shortest expected execution time, i.e. the sum of both numbers.

CloudBuild schedules tasks eagerly to overlap fetching of inputs with the execution of other tasks. Such eager scheduling is done carefully however to not make decisions far into the future.

Some classes of task execution errors are deemed retriable, and depending on the nature of the failure, a policy may be set in the task to seek retries in different builder nodes. In this case, the build coordinator excludes the previous execution site from the list of candidates for future placements of the task.

Effectively, CloudBuild uses critical path scheduling [26] but with added attention to multi-resource packing of build tasks and eager queuing of tasks so that fetching inputs can overlap the execution of other tasks. In particular, tasks' have a large number of inputs and in a rather unstructured manner (as opposed to a reduce task reading from every map task [8]). This forces CloudBuild to more carefully account for input locality by actually computing the

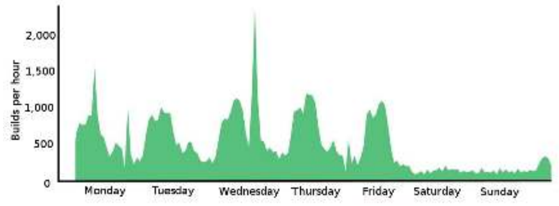


Figure 6: Arrival process of builds

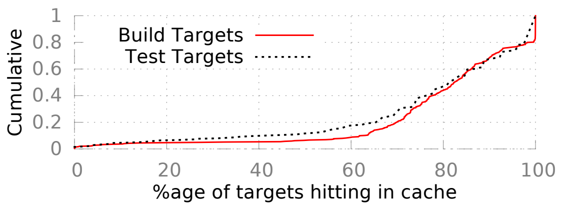


Figure 8: Cache hit rates for 405 builds of codebase B.

time to fetch the inputs; the corresponding issue is simpler in data-parallel systems [3, 8, 21]. Much of our experiences with the build scheduler are documented in §4.1.

## 3.7 Managing Build Outputs

CloudBuild needs to support multiple mechanisms to facilitate access to build outputs. To offer a backwards-compatible build output storage, CloudBuild can persist build outputs into SMB file shares [37]. This has been the primary mode of storage in the old Microsoft build labs that CloudBuild seeks to replace. CloudBuild can also save build outputs into the *Artifact Repository*, an internal content storage system designed on top of Azure Blob Storage [23] and integrated into Microsoft's Visual Studio Online [36] suite. This system allows for progressive differential data uploads during build execution; similarly, it offers customers progressive differential downloads, to reduce the end-to-end time to acquire all of the content produced by a build.

## 4. EXPERIENCES

We want to share some of our experiences and lessons learned from building and operating the service over the last few years.

## 4.1 Characterizing CloudBuild

CloudBuild runs in several datacenters and uses thousands of servers. Roughly twenty thousand builds are launched per day. Figure 6 depicts the arrival process of builds. Demand is mostly diurnal, with peaks at 11AM and 4PM. During business days around 60% of the traffic is due to user requests; around 25% are continuous integration builds that are triggered by a code check-in; 10% are scheduled builds, configured to run at preset times.

Over 95% of the builds use the distributed cache, which is the default option. CloudBuild allows developers the option to not use the cache which is useful for troubleshooting and onboarding new branches. Cache hit rates vary substantially based on the codebase being built. Figure 8 shows the cache hit rates for different builds of codebase B which as described in Table 2 is currently the largest codebase by size and execution volume. For 90% of the builds, at least 50% of the targets are cached, and for 50% of the builds, at least 80% of the targets are cached. The overall cache hit rate is 76% which translates into significant savings. Caching helps both cluster throughput (builds per second) and build latency (time to finish a build).

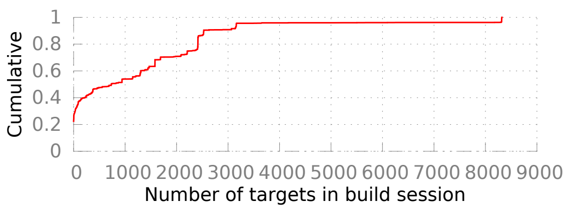


Figure 9: Targets in build job.

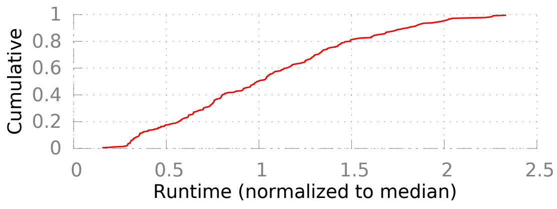


Figure 11: How long does a build take? Above, we show a CDF of the build times of codebase B.

To assess CloudBuild's performance, we compare build execution times with the critical path of the build dependency graph (Figure 10a). The distance in Figure 10a is on average about 20% of the total build time. Some of the difference is due to copying of input and outputs; aspects that are not accounted for in our critical path calculation.

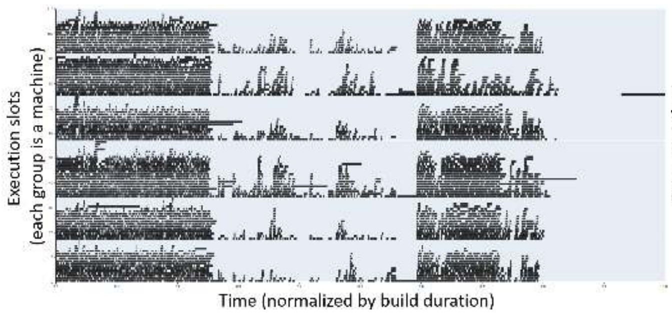
We also compare build execution times with an estimate of the total work (Figure 10b) in the build DAG. This metric is the maximum over all resources, the sum of task duration times the demand for that resource divided by the available capacity of that resource. If there were no other constraints, the total work is a lower bound on the runtime. From Figure 10b, we see that the total work is between 10% and 20% of the actual running time. A look at the timelapse of an example build of codebase B Figure 7a can explain why. The execution schedule shows several voids in which only a few tasks are running. This is because, naively generated dependency graphs can have several "choke points" that limit parallelism. Choke points are instances when all of the unscheduled tasks depend on all of the previous tasks allowing a single outlier to prolong the build [21].

Figure 7b displays a similar view for codebase C. This codebase has been running distributed builds for a number of years; over time its developers have optimized the build dependency graph to increase parallelism, thus reducing the build times. Automatic identification of choke points in our codebases is an area of additional research.

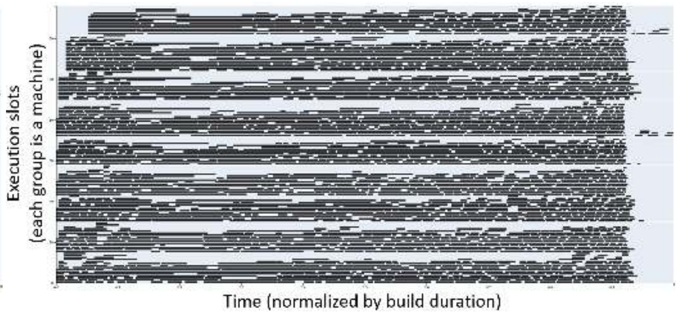
Figure 11 shows that the maximum build time for codebase B is nearly 4× the median build time. The primary factor for this difference is caching. The range between min and max is about 10×. We believe that shaving off some of the fixed overheads would stretch this range further.

Table 3 shows the typical resource utilization of tasks. While on average tasks use relatively few resources, the standard deviation is high. From the maximum values, it can be seen that a one-size-fits-all approach to scheduling would not suffice. Multi-resource packing [28] could improve cluster throughput.

Table 4 takes a deeper look at the correlation between different kinds of task resources. It plots the covariance in usages of different resources. Overall, they do not correlate strongly except for the data read and written by tasks. There is a weaker correlation between memory and data read or written. This is in contrast to tasks in data parallel clusters which for the most part process one row after another (joins are an exception). Many build tasks, such as com-

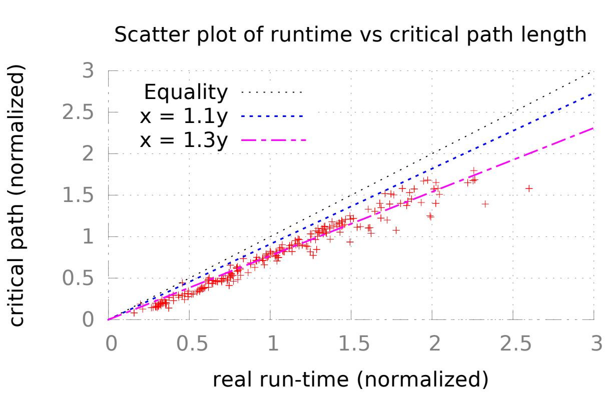


(a) Codebase B (unoptimized)

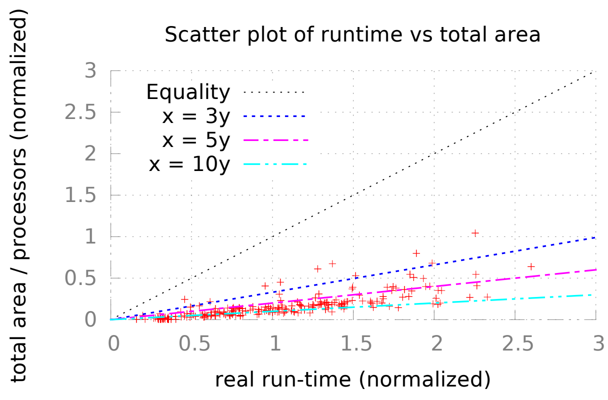


(b) Codebase C (optimized over time)

Figure 7: Depicted above are two actual execution schedules for different builds. Each dark grey block represents a task, and the tasks are sorted by the machine on which they were executed. Figure 7a is an unoptimized build, while Figure 7b depicts a schedule for a different, optimized codebase.



(a) Critical path



(b) Area

Figure 10: The figures above compare the runtimes of codebase B with two lower bounds: length of the critical path, and the area. The figures show that codebase B is dominated by work on the critical path and that CloudBuild's scheduler is nearly optimal.

| Resource | Avg. | Stdev. | max |
|---|---|---|---|
| Cores | 0.52 | 0.23 | 4 |
| Memory (MB) | 105 | 93 | 3622 |
| Data read (MB) | 40 | 141 | 5220 |
| Data written (MB) | 23 | 148 | 5355 |
| # Parents | 9 | 43 | 3484 |
| # Children | 9 | 67 | 2052 |
| Duration (s) | 21 | 25 | 725 |

Table 3: Resource profiles of tasks.

| | C | M | R | W | P | CH |
|---|---|---|---|---|---|---|
| **C** | – | | | | | |
| **M** | 0.415 | – | | | | |
| **R** | 0.178 | 0.459 | – | | | |
| **W** | 0.103 | 0.390 | 0.875 | – | | |
| **P** | -0.011 | 0.039 | 0.443 | 0.320 | – | |
| **CH** | 0.01 | 0.07 | 0.071 | 0.067 | 0.016 | – |

**C**: Cores, **M**: Mem., **R**: Data read, **R**: Data written, **P**: Parents, **CH**: Children

Table 4: Correlation between pairs of resource demands of the various tasks.

pilation, linking and tests may retain more of their inputs (binaries and executables) in memory for the duration of the task. Overall, however, the lack of correlation means that there is opportunity to pack tasks with diverse demands, e.g. I/O intensive jobs with CPU intensive jobs.

## 4.2 People Experiences

**Customer Education:** While CloudBuild was designed to be highly compatible with old build specifications, it does impose additional constraints on what users specify. Because of sandboxing, the execution environment can be somewhat different from that of MSBuild and nmake. Hence, builds can experience unique failures in CloudBuild; even common failures may present themselves differently in a distributed environment. As large engineer populations onboard into CloudBuild, we have had to fund substantial user education efforts, with both proactive documentation and troubleshooting consultancy. We also have implemented functionality to detect, triage and explain failures, and to correlate them with pertinent documentation. Having said that, much work is still needed to identify systems that can automatically surface the most relevant content based on user-stated or implied needs. Another customer education direction was in training to adapt build specifications to forms that can be more effectively parallelized. Figure 7 shows an example where the build on the right is more easily parallellizable than the one on the left (less void cycles, more work off the critical path, etc.).

**Virtuous but spiraling demand:** As codebases are onboarded and users see shorter build times, the demand for builds increases. Builds that would be erstwhile executed on desktops are now offloaded to CloudBuild. The lower cost also means that developers are much more inclined to submit "speculative builds" even where there is little confidence that the builds and tests will succeed. Developers get to wait less and check-in more often, increasing the demand of pre- and post-checkin build jobs. We also saw the number of unit-tests consistently increase. This is less of a problem though since such tests typically add little to the critical path of the build. CloudBuild continues to be substantially scaled out to accommodate all the dimensions of growth.

**Evangelizing the best-of-breed:** By becoming the most widely adopted build solution at Microsoft, our team has exposure to a large variety of codebases. This has enabled the team to identify best-of-breed tools, build specifications, patterns, and practices, and to evangelize them across different groups. We also funnel emerging requirements back to tool owners, and allow them to validate tool changes against their customer codebases on CloudBuild.

## 4.3 Technical Experiences

**Races and Non-idempotence:** It is possible that two concurrent build jobs observe a cache miss and race to execute the same task. Avoiding races requires serialization and hurts performance. Hence, rather than prevent such races, CloudBuild optimistically handles them after the fact. The issue with races is that build tools are not idempotent, i.e., running the same task twice can produce different outputs, for a variety of reasons. The output can contain environmental information such as machine-name or timestamp. Or, it could have some uninitialized strings with random values. To handle such races, CloudBuild only allows the first writer, deemed the winner, to update the cache. The others receive an indication that the task's output is already available causing them to discard their local output and to use the winner's output instead.

Customers expect provenance relationships between outputs of a build job to hold just as if the build had been performed on a single machine without caching. For example, consider a build where a task $t_1$ emits file `raw/file.dll` and a child task $t_2$ emits `compressed/file.dll.gz`. A build output consumer can expect that unzipping `file.dll.gz` yields a file that is bitwise-identical to `file.dll`. Assume now that an earlier build job created cache entries for $t_1$ and $t_2$, but later $t_1$'s entry was deleted for some reason. If $t_1$ is regenerated by a non-idempotent tool and $t_2$'s output is retrieved from the cache, `file.dll` and `file.dll.gz` will no longer match. To avoid this issue, CloudBuild includes the input's content bag *identity* when computing a target's `CacheKey`. In this example, $t_1$'s content bag *identity* is used to obtain $t_2$'s `CacheKey` which leads to a cache miss and thus forces $t_2$ to be re-built avoiding "Franken-builds".

**Build Job Cohorts:** For large codebases, build durations and request arrival processes are such that there are often many concurrent jobs building the same codebase. Most developers tend to work close to the "head revision" in their codebase, so the majority of source code is common across concurrent jobs. The races described above are therefore very common. Later-starting jobs are more likely to catch up since they consume cached content created by the earlier-starting jobs. Once they catch-up, they will pathologically race to execute further tasks (over common sources). This creates high demand for certain `CacheKeys` and causes load on the GCS to spike. This also wastes cluster resources in the work performed by the race loser. Federating schedulers so as to minimize races between different build jobs is an area for further exploration.

**Non-Deterministic Build Outputs:** Prior to CloudBuild, codebases were built using a single machine. While parallelism was possible, task execution order were mostly stable across build jobs since the relatively naïve scheduler had only the limited resources offered by a single machine. Conflicts between tasks can go unnoticed when task execution order is stable. For example, suppose a spec allows two tasks to output to the same file path but this would go unnoticed if the stable execution order ensured that the same task always won in single machine builds. Since CloudBuild schedules over many machines and can receive variable amounts of cached outputs, the execution orders are no longer deterministic. This surfaced several hidden or latent bugs. The vast majority of onboarded codebases had to be patched to remove latent bugs. CloudBuild's consistency policies (see §3.4) helped detect some of these issues. Surprisingly, some codebase owners were happy to allow output non-determinism – in some cases because the affected output content was not used by the application at run-time; on others because any of the possible output values was acceptable.

## 5. RELATED WORK

Google's Bazel [5] system solves a similar problem– parallel and cached builds for large code bases. The key difference is that Bazel requires developers to write specifications and redesign their code layout. It offers a new build specification language and requires a specific directory structure. Both these decisions can simplify dependency inference. In contrast, CloudBuild's primary goal is to quickly onboard teams with minimal changes to their existing build specs and tools. Further distinctions are hard to draw since very little information is publicly available about the operational aspects of Bazel. The released version only targets single machine builds. How they cache or schedule builds, the extent of support for arbitrary build tools and their performance and time-to-onboard new teams would all be very useful topics to compare in the future.

distcc [6] is another publicly available system. The unit of work is a preprocessed source code file that is sent over the network and compiled remotely. A *pump mode* allows preprocessing to also be executed on the distributed host. Compared to CloudBuild, it misses several steps such as distributed linking or test execution. Linking is executed on the main host, and test execution is not part of its design. Further, distcc only supports gcc.

State-of-the-art build systems such as Maven [2], Ant [1] and SBT [13] are neither distributed nor cache build outputs; some offer parallelization at a coarse granularity [2]. The key ideas in CloudBuild improve upon ideas from prior work; our use of sandboxes is similar to [20] and our cache for build outputs is akin to [29].

Some prior work describes new build specification languages with formal properties [25, 30] and techniques to automatically migrate legacy specifications [27]. Other work reduces "debt" [38] or pares down build targets to only the outputs that are actually used [40]. CloudBuild is orthogonal to these efforts. We believe that such efforts are needed and we are working towards these goals as well.

Yarn [8], Mesos [31], and other data-parallel computing infrastructures work with tasks that are already amenable to distributed execution, i.e. tasks with deterministic and idempotent behavior. These attributes also mean that they are easily runnable in parallel. Many of the challenges that CloudBuild solves are towards making build tools and specifications geared for single-threaded / single-machine execution to work in a distributed setting.

The build scheduling problem (§3.6) is closest to scheduling jobs with dependencies (DAGs) as is common in data-parallel frameworks such as Tez [3], Hive [39], and SCOPE [24]. There are a few key differences however. First, due to caching, the subset of the build DAG that needs to be executed varies dynamically. Second, unlike data-parallel jobs where the data-in-flight reduces dramatically with each operation [19], build outputs are numerous and large in size. Hence, scheduling build DAGs needs more careful handling of dependencies.

## 6. FINAL REMARKS

CloudBuild was started to enable Bing's fast delivery cadence. It has since been rapidly adopted by many teams across Microsoft. The primary reason has been that instead of few builds per day, developers could suddenly do many builds and thus run many more experiments. For instance, the number of builds for codebase B doubled within six months, despite the fact that the team did not change its development process. Changing engineering processes can improve agility further. For instance, a team that recently onboarded has 2000 targets and executes 10× more builds per day than they could two months ago using their old build-lab-based system.

Growing CloudBuild to support teams with diverse programming cultures (e.g., Bing and Office) was not without challenge. First, we struggled with the immense scale requirements. For instance, our load exceeded existing binary storage and file-signing solutions. While we could scale up or scale out some of the dependent systems, others had to be redesigned. We created a new content-addressable de-duplicated store in the cloud, which is currently used for drops, symbols, and packages. By exploiting CloudBuild's cache hit rates we were able to reduce the ingress and egress volume for various storage system by over 80%.

Another challenge was the requirement to integrate into the very different engineering workflows that existed across groups. In some cases, CloudBuild did not have proper abstractions for all integration points. When new requirements came up we followed the open source model and allowed developers from other organizations to contribute. The resulting system slowly became fragile, because of unintended feature interactions of the new dependencies. We took various steps to address this issue: Workflow integration now follows a publish/subscribe model, i.e. integration is done externally. For simpler deployment, testing in production, and dependency isolation, we are currently redesigning CloudBuild to follow a more decoupled service design pattern. Finally, we have started to continuously apply Chaos Monkeys to improve the resilience and recoverability of our services.

In spite of the rapid growth of CloudBuild to a service used by thousands of engineers, CloudBuild continues to achieve its primary goal: improve the cycle time for the developers inner loop: build, and test, and do so reliably and quickly.

## 7. ACKNOWLEDGMENTS

The authors would like to thank current and past CloudBuild staff members and contributors: *Stephan Adler*, *Andrew Arbogast*, *Jonathan Boles*, *Nick Carlson*, *Matthew Chu*, *Jacek Czerwonka*, *John Erickson*, *Steve Faiks*, *Maksim Goleta*, *Dmitry Goncharenko*, *Dedian Guo*, *Mario Guzzi*, *Jeffrey Hamblin*, *Cody Hanika*, *Kim Herzig*, *Zach Holmes*, *Jinjin Hong*, *Fei Huang*, *Rui Jiang*, *Paul Jones*, *Jeff Kluge*, *Ben Lieber*, *Eric Maino*, *Joshua May*, *Michael Michniewski*, *Paul Miller*, *Nick Pizzolato*, *Michael Pysson*, *Josh Rowe*, *Zack Runner*, *Vinod Sridharan*, *Stanislaw Szczepanowski*, *Carl Tanner*, *Suresh Thummalapenta*, *Umapathy Venkatachalam*, *Christopher Warrington*, *Rick Weber*, *Martha Wieczorek*, *Yancho Yanev*, and *Jet Zhao*.

## 8. REFERENCES

[1] Apache ant project. ant.apache.org.
[2] Apache maven project. maven.apache.org.
[3] Apache Tez. http://tez.apache.org/.
[4] Apache zookeeper. zookeeper.apache.org.
[5] Bazel. http://bazel.io/.
[6] distcc. https://github.com/distcc/distcc.
[7] Git for visualstudio. http://bit.ly/1vqcDFe.
[8] Hadoop YARN Project. http://bit.ly/1iS8xvP.
[9] MSBuild. http://bit.ly/1Fwk6Ez.
[10] Nmake. http://bit.ly/1NgNzsE.
[11] Nuget. http://bit.ly/1OdeEJA.
[12] robocopy. http://bit.ly/1OdeEJA.
[13] Scala build tool. www.scala-sbt.org.
[14] Source depot. https://en.wikipedia.org/wiki/Perforce.
[15] Team foundation version control. http://bit.ly/1ES8VLm.
[16] TPC-DS Benchmark. http://bit.ly/1J6uDap.
[17] TPC-H Benchmark. http://bit.ly/1KRK5gl.
[18] VSTest. http://bit.ly/1L2nmPO.
[19] S. Agarwal et al. Re-optimizing data parallel computing. In *NSDI*, 2012.
[20] G. Ammons. Grexmk: Speeding up scripted builds. In *Workshop on Dynamic Systems Analysis*, 2006.
[21] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*, 2010.
[22] A. Baumann et al. Shielding applications from an untrusted cloud with haven. In *OSDI*, 2014.
[23] B. Calder et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.
[24] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
[25] M. Christakis, K. Leino, and W. Schulte. Formalizing and verifying a modern build language. In *FM*, Lecture Notes in Computer Science. 2014.
[26] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.
[27] M. Gliboric et al. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *OOPSLA*, 2014.
[28] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource Packing for Cluster Schedulers. In *SIGCOMM*, 2014.
[29] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *PLDI*, 2000.
[30] J. Hickey and A. Nogin. Omake: Designing a scalable build process. In *FASE*, 2006.
[31] B. Hindman et al. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.
[32] G. Hunt and D. Brubacher. Detours: Binary interception of win 32 functions. In *Usenix Windows NT Symposium*, 1999.
[33] M. Isard. Autopilot: Automatic Data Center Management. *OSR*, 2007.
[34] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.
[35] Microsoft. Azure Service Bus. http://bit.ly/1LjqUIf.
[36] Microsoft. Introducing Visual Studio Online. http://bit.ly/1OvpNez.
[37] Microsoft. SMB Protocol and CIFS Protocol Overview. http://bit.ly/1Crljd7.
[38] J. D. Morgenthaler et al. Searching for build debt: Experiences managing technical debt at google. In *Workshop on Managing Technical Debt*, 2012.
[39] A. Thusoo et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, 2009.
[40] M. Vakilian et al. Automated decomposition of build targets. In *ICSE*, 2015.
[41] J. Zhou et al. SCOPE: Parallel Databases Meet MapReduce. In *VLDB*, 2012.