Scalable Build Service System with Smart Scheduling Service

Kaiyuan Wang
Google, USA
kaiyuanw@google.com

Greg Tener
Google, USA
gtener@google.com

Vijay Gullapalli
Google, USA
vijaysagar@google.com

Xin Huang
Google, USA
xnh@google.com

Ahmed Gad
Google, USA
ahmedgad@google.com

Daniel Rall
Google, USA
dlr@google.com

ABSTRACT

Build automation is critical for developers to check if their code compiles, passes all tests and is safe to deploy to the server. Many companies adopt Continuous Integration (CI) services to make sure that the code changes from multiple developers can be safely merged at the head of the project. Internally, CI triggers builds to make sure that the new code change compiles and passes the tests. For any large company which has a monolithic code repository and thousands of developers, it is hard to make sure that all code changes are safe to submit in a timely manner. The reason is that each code change may involve multiple builds, and the company needs to run millions of builds every day to guarantee developers' productivity.

Google is one of those large companies that need a scalable build service to support developers' work. More than 100,000 code changes are submitted to our repository on average each day, including changes from either human users or automated tools. More than 15 million builds are executed on average each day. In this paper, we first describe an overview of our scalable build service architecture. Then, we discuss more details about how we make build scheduling decisions. Finally, we discuss some experience in the scalability of the build service system and the performance of the build scheduling service.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Build service system, build scheduling service, build system design

ACM Reference Format:
Kaiyuan Wang, Greg Tener, Vijay Gullapalli, Xin Huang, Ahmed Gad, and Daniel Rall. 2020. Scalable Build Service System with Smart Scheduling Service. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3395363.3397371

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07.
https://doi.org/10.1145/3395363.3397371

1 INTRODUCTION

Building is a central phase in the software development process in which developers use compilers, linkers, build files and scripts to assemble their code into executable units. Modern build tools like Gradle [6], Buck [2] and Bazel [1] support projects in multiple languages and build outputs for multiple platforms. Programming is often described as an "edit-compile-debug" cycle in which a programmer makes a change, compiles, and tests the resulting binary, then repeats the cycle until the program behaves as expected. Slow builds may cause the programmer to be distracted by other tasks or lose context, and reduce the number of code submissions per day. Any delay increases the gap between the programmer deciding on the next change to perform and viewing the effect of that change.

Although developers can run their builds on their local workstation, the ability to build software remotely is also critical. For example, the build environment of the local workstation may be different, e.g. flags, platforms and build speed, and that difference may cause a build to succeed on one machine but fail on the other machine. A remote build allows us to control the environment and make sure the build result is consistent and reliable. Moreover, many critical services rely on a remote build service. For example, the Continuous Integration (CI) service needs to run builds on remote machines. The software release service also needs to build software every day to make sure products can rollout on time.

To provide a fast and smooth software building experience, we have designed a build service system that is able to run tens of millions of remote builds per day. We list some challenges the build service system needs to solve at Google:

- **Scalability**. The system should be able to handle hundreds of build requests per second and run millions of builds per day.
- **Low Latency**. We don't want to slow down the developers productivity, so the build service overhead should be small and reasonable. The build execution time should be reasonably fast.
- **Reliability**. Since the number of builds is significantly higher than the number of machines to run those builds, we need to reliably keep track of created builds and run them whenever a machine is available. Moreover, since the number of machines is large, machine failure is not a matter of if, but when. So we need to tolerate machine failures and be able to retry the build whenever that happens.
- **Priority**. The builds should be prioritized. Some builds are more important and should be scheduled to run sooner than other builds. For example, a human triggered build is often more important than an automation tool triggered build, because it is more costly for a human to wait.

```
java_library(
  name = "Greeter",
  srcs = ["Greeting.java"],
)
java_library(
  name = "HelloWorld",
  srcs = ["HelloWorld.java"],
)
java_binary(
  name = "HelloWorldMain",
  main_class = "HelloWorld",
  runtime_deps =
    [":HelloWorld"],
)
```

```
java_library(
  name = "HelloWorldTest",
  srcs =
    ["HelloWorldTest.java"],
  deps = [":HelloWorld"],
)
java_test(
  name = "AllTests",
  size = "small",
  tags = ["requires-gpu"],
  deps = [":HelloWorldTest"],
)
```

Figure 1: Example build specification

- **Instant Feedback**. We want to provide a remote build service that behaves like the build is running on the developer's local workstations. This means the build output feedback should be streamed back to the user when the build is running. It is unacceptable to deliver the final console output only after the build is completed.

In this paper, we describe the architecture of the build service system used in Google that solves all of the above challenges. Then, we describe more details about our build scheduling service. Finally, we show the scalability of our build service system and the importance of the scheduling service. Specifically, our build service system is able to run 15 million builds on average per day.

The paper makes the following contributions:

- It demonstrates the possibility to implement a build service system that runs tens of millions of builds to support industry-scale development.
- It presents the architecture of the build service system after years of refinement in Google. Our experience could be useful for others to build a scalable build service system.
- It discusses our build scheduling service and its improvement in efficiency on the build service system.

2 BACKGROUND

This section describes the Bazel build tool [1], the Spanner database [9] and the proportional–integral–derivative controller [16] we use in our build service system.

2.1 Bazel

Bazel is an open-source build and test tool similar to Make [5], Maven [7], and Gradle [6]. It uses a human-readable, high-level build language. Bazel supports projects in multiple languages and builds outputs for multiple platforms. Google uses a build tool built on top of Bazel. For simplicity, we refer to Bazel in the rest of the paper as the build tool in Google.

Bazel takes as input a set of targets that programmers declare in build files. Figure 1 shows a Java sample build specification. Each build specification contains a set of targets. Most targets are one of two principal kinds, files and rules. A rule specifies the relationship between inputs and outputs, and the actions to build the outputs. Actions are sets of system calls, e.g. shell scripts, that will be executed to complete the build. Rules can be of one of many different kinds or classes, which produce compiled executables and libraries, test executables and other supported outputs. For example, `java_library` is a rule to compile Java source files into libraries. `java_test` is a rule to execute Java tests. `java_binary` is a rule to create executable files for Java. Targets may have dependencies and each target and its dependent targets form an acyclic dependency graph. For example, the Java library target `HelloWorld` depends on the Java library target `Greeter`. The Java library target `HelloWorldTest` depends on `HelloWorld` target. The Java test target `AllTests` depends on `HelloWorldTest` target and the test execution requires GPU. The Java binary target `HelloWorldMain` depends on target `HelloWorld`. When a programmer issues a command to build a target, the build system first ensures that the required dependencies of the target are built. Then, it builds the desired target from its sources and dependencies.

```
CREATE TABLE BuildTable (
  id STRING NOT NULL,
  state State BLOB NOT NULL,
  build Build BLOB NOT NULL,
  priority Priority NOT NULL,
  request_time TIMESTAMP
) PRIMARY KEY (id);

CREATE INDEX StatePriorityRequestTime
  ON BuildTable(state, priority, request_time);
```

Figure 2: Spanner example schema

A build includes an execution context, a build command, and some metadata. The execution context specifies the workspace (with or without unsubmitted code changes) in which to run Bazel. We use execution context and workspace interchangeably in the rest of the paper. A Bazel build command specifies the command name, the flags and the target to run. For example, the command `bazel run --jvmopt="-Xms256m" :HelloWorld` runs the `:HelloWorld` target with the JVM startup heap size set to 256 MB. Note that `run` is the command name, `--jvmopt="-Xms256m"` is the build flag, and `:HelloWorld` is the target. The metadata stores other relevant information of the build, e.g. the build's unique identifier or priority, etc. A majority of our remote builds is one of the three kinds: (1) executing a dependency graph query; (2) build the specified targets; and (3) build and run the specified test targets.

2.2 Spanner

Spanner is a scalable, globally-distributed database [9]. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [13] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Figure 2 shows an example Spanner database schema that creates a `BuildTable` for storing the build to run. The `id` column stores a universally unique identifier for each build. The `state` column stores an enum that specifies the running state of the build. The `build` column stores an object that specifies the build command, flags and targets to run. The `priority` column stores an enum that specifies the priority of the build. The `request_time` column stores the request time of the build. The primary key of the `BuildTable` is the `id` column, and it lets Spanner automatically index the `BuildTable`.

Spanner allows us to create secondary indexes for other columns. Adding a secondary index on a column makes it more efficient to look up data in that column. For example, Figure 2 shows a secondary index `StatePriorityRequestTime` on the `BuildTable`, which allows us to quickly find all running builds and iterate over those builds in priority order. For builds of the same priority, we order them by their request time in chronological order. This enables an efficient build scheduling algorithm based on build priority and request time.

2.3 Proportional Integral Derivative Controller

A proportional–integral–derivative (PID) controller [16] is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID controller continuously calculates an error value $e(t)$ as the difference between a desired set point (SP) and a measured process variable (PV), and applies a correction based on proportional, integral, and derivative terms (denoted P, I, and D respectively). The input to the process is the output from the PID controller, and it is called manipulated variable (MV).

The proportional, integral, and derivative terms are summed to calculate the output of the PID controller, which is denoted by $ut(t)$ and defined as follows:

$$u(t) = MV(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

where $K_p \geq 0$, $K_i \geq 0$ and $K_d \geq 0$ denote the coefficients for the proportional, integral, and derivative terms, respectively. $e(t) = SP - PV(t)$ denotes the error. t denotes the present time. τ is the variable of integration and it ranges from 0 to the present time t.

In a real implementation, the integral term is discretized, with a sampling time delta Δt, as

$$\int_0^{t_k} e(\tau)d\tau = \sum_{i=1}^{k} e(t_i)\Delta t$$

and the derivative term is approximated as

$$\frac{de(t_k)}{dt} = \frac{e(t_k) - e(t_{k-1})}{\Delta t}$$

The PID controller is designed to make PV smoothly approach SP and finally equal to SP. In this paper, the build service uses the PID controller to make scheduling decisions.

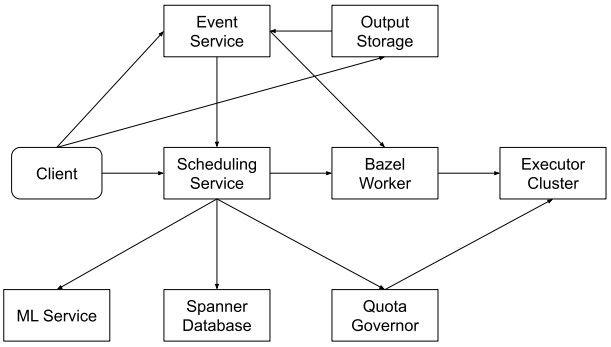

Figure 3: Build service architecture

3 BUILD SERVICE ARCHITECTURE

Figure 3 shows the architecture as microservices of the build service system in Google. Each component talks to its connected component via remote procedure calls (RPC). This allows the system to be loosely coupled, scalable, and highly maintainable and testable. Each arrow points from a component to another dependent component. For example, the build scheduling service depends on the Spanner database but not vice versa. In this section, we first briefly describe a common workflow of running a build. Then, we describe each component in the architecture.

3.1 Workflow

A typical workflow of a build includes the following steps:

- The client sends the build to the scheduling service.
- The scheduling service sends the build to the Bazel worker and returns a build ID to the client.
- The client uses the returned build ID to query the status of the build via the build event service.
- The worker starts a workspace with a Bazel process. The Bazel process analyzes the build and computes the actual actions to run on the executor cluster.
- The executor cluster runs the actual actions and returns the output.
- The output is sent to the build event service and saved in the storage service.
- The client listens to the build event service as if the build is executing on the client's local workstation.

3.2 Client

The client is provided with a set of APIs to interact with the build service system. Table 1 shows the build APIs the client is able to invoke via RPCs. The `CreateBuild` API allows the client to create a build in the system. The build will be queued in the scheduling service and executed some time in the future. The API returns a unique build ID so that the client can query the status of the build via the `WatchBuild` API. The `GetBuild` API allows the client to query the build and its current state. The `CancelBuild` API cancels a previously created build, which frees up some system resources. The `WatchBuild` API allows the client to receive the build output when the build is running. All of `GetBuild`, `CancelBuild` and `WatchBuild` take the build ID as input.

Table 1: Build API

API	Description
CreateBuild	Create a build, queuing it to be executed at some point in the future. A unique ID is returned to the client for querying the build.
GetBuild	Get a build, including its current status.
CancelBuild	Cancel a previously created build. If the build is done or in the process of cancelling, this has no effect.
WatchBuild	Return a stream of events for a build.

3.3 Scheduling Service

The build scheduling service implements the `CreateBuild`, `GetBuild` and `CancelBuild` APIs. It uses a priority queue to hold all created builds. All builds are ordered by their priority which is inferred from the client that creates builds. For example, builds created by humans for code submission may have higher priority than builds created by an automation tool that collects code coverage. A build is dequeued whenever a Bazel worker is available and there exist enough idle executors to run the build. Higher priority builds are strictly dequeued before lower priority builds. The scheduling service generates a global unique ID for a build, and the build ID is returned to the client.

The scheduling service updates the build state during the lifecycle of a build. Table 2 shows all possible states of a build during its lifecycle. A build can be in one of the three states, i.e. `ENQUEUED`, `IN_PROGRESS` or `FINISHED`. When a build is in the priority queue, it is in the `ENQUEUED` state. When a build is dequeued and running on the Bazel worker, it is in the `IN_PROGRESS` state. When a build is finished and the result is returned, it is in the `FINISHED` state. The scheduling service publishes the `BUILD_ENQUEUED`, `INVOCATION_STARTED` and `INVOCATION_FINISHED` events, respectively, to the build event service whenever the build state is set, for the first time, to `ENQUEUED`, `IN_PROGRESS` and `FINISHED`, respectively. These events which are published when a build state is changed are referred to as *lifecycle* events. Normally, a build moves from the `ENQUEUED` state to the `IN_PROGRESS` state, and finally to the `FINISHED` state. But it is possible that the build is lost during the execution, e.g. the worker dies, so in practice the build can move to the `ENQUEUED` state and then move to the `IN_PROGRESS` state repetitively, and finally move to the `FINISHED` state. Whenever a build is dequeued, the scheduling service generates a unique invocation ID for the execution. The build ID is unique during a build's lifecycle but a build can have multiple invocation IDs. The invocation ID can be used to query build execution result from the output storage service.

3.4 Bazel Worker

The Bazel worker is a container with multiple workspaces, and each workspace can run a single Bazel process. Each worker has a limited number of workspaces and thus cannot run more builds in parallel than the workspace capacity. The worker controls the amount of resources, e.g. memory, network and CPU, assigned to each workspace.

The worker sends a long running `GetNextBuild` RPC to the build scheduling service and receives back the build information to start the invocation. The `GetNextBuild` RPC is sent to the scheduling service whenever a workspace becomes available. The selected workspace then sets up the building environment and starts a Bazel process to run the build. During the execution, the workspace periodically renews its lease on the build, i.e. it notifies the scheduling service that the build is in progress. If the worker dies, e.g. hardware maintenance, and the scheduling service does not receive the lease renewal for a while, then the build will be requeued in the scheduling service. If the workspace tries to renew the lease but the build is not in the `IN_PROGRESS` state, e.g. build is requeued or cancelled, then the worker immediately cancels the build to free up resources. Once the build finishes, the workspace sends the build's final result via the `FinishBuild` RPC to the scheduling service which then sets the build to the `FINISHED` state.

Bazel publishes build events to the build event service during build execution. Some build events contain console output which will be printed in the client's terminal. Other build events contain the verbose build logs and these events will be consumed by the output storage service.

Bazel analyzes the build flags and targets to generate a set of actions. Actions may depend on other actions, e.g. the output of one action may be the input of another action. Thus, Bazel generates a directed acyclic action graph, where a node represents an action and an arrow points the output of one action to the input of another action. Bazel sends the actions to the executor cluster in the topological order in parallel to minimize the execution time. Internally, Bazel caches the output of each action by its input digest and only sends the action to the executor cluster if Bazel does not know the output of that action. The actual executions happen on the executor cluster.

3.5 Executor Cluster

The executor cluster executes the actions Bazel sends. The cluster has a global action cache, which can return the action output immediately if the action is executed recently for some other builds. The number of actions running on the executors is very large such that over 99% of actions are cached on average. The executor cluster contains a queue of actions waiting for execution until some executor is free. The action queue length is intended to be small because the scheduling service already has a build queue to hold excessive builds when the executor cluster is full.

The executors have multiple hardware architectures. A majority of the executors are x86 CPUs which are used by all builds. A large set of builds uses Mac machines because Apple's App Store apps must be built on iOS devices. GPUs and TPUs are popular because they can speed up training machine learning models. We use *executor type* to denote different executor architecture in the rest of the paper.

In Google, each product area (PA) is only allowed to use a restricted set of executors so that any PA will not use up all executors. For example, mobile app developers will not see slow builds and be blocked by video website developers who use a large set

Table 2: Build states and corresponding events

State	Event	Description
ENQUEUED	BUILD_ENQUEUED	The build is in the priority queue waiting to be dequeued.
IN_PROGRESS	INVOCATION_STARTED	The build is dequeued to a worker and running.
FINISHED	INVOCATION_FINISHED	The build is finished and the build result is returned.

of executors. However, if executors in a PA is underutilized, then another PA can borrow a limited subset of executors from the PA.

Not all actions are equally expensive. For example, some action may require more CPU/memory to run compared to another action. An action that uses one executor with 5GB memory is more expensive than another action that uses one executor with 2GB memory. In the rest of the paper, we use an executor service unit (ESU) to unify the expense of both memory and CPU. One ESU is equal to 2.5GB of memory or 1 executor. The executor cluster keeps track of the type and amount of ESU each build uses and passes that information to another build service component, i.e. quota governor.

3.6 Quota Governor

In order to govern the quota usage, each PA is assigned to a limited amount of ESU quota. The intention is to avoid the case where builds from a single PA occupy the entire executor cluster while builds from other PAs cannot be started due to lack of resources. We use PA and quota group interchangeably in the rest of the paper. The quota governor computes the ESU each quota group is allowed to use, and the scheduling service uses that to decide the number of builds to dequeue for each quota group.

Section 2.3 introduces the PID controller. In the quota governor, PV maps to the current executor occupancy, i.e. the amount of ESU the executor cluster is using, per quota group and executor type. SP maps to the desired executor occupancy, i.e. the amount of ESU allowed to use, per quota group and executor type. Each quota group is allowed to borrow unused executor quota from all other quota groups, so the SP of each quota group keeps changing as the utilization ratio of the executor cluster changes. MV maps to the target executor occupancy, i.e. the amount of ESU we want to fill up by running more builds. In practice, the PID controller computes the MV and the scheduling service uses MV to decide how many more builds should be dequeued. When PV is larger than SP and MV is negative, the scheduling service simply stops dequeuing more builds and waits until some resources are freed. When PV is smaller than SP again, the scheduling service starts dequeuing more builds. This strategy makes PV smoothly approaching SP and avoids overshooting.

3.7 Build Event Service

The build event service is designed to facilitate build progress reporting. It asynchronously sends build-related events from remote build components (called event publishers) to any number of build watchers. Examples of build events include, but are not limited to, lifecycle events from the scheduling service or console output events from Bazel. A build watcher can watch a build while it's ongoing, or for up to a given period of time after a build has been created. A build watcher can pause the watch session at an arbitrary position, and then resume the paused session.

Lifecycle events are build level events because they are published when the build state is changed. Bazel events are invocation level events because a build can be lost for many reasons and executed multiple times. No Bazel event can be sent to a build watcher unless both `BUILD_ENQUEUED` and `INVOCATION_STARTED` events have already been sent. Likewise, the `INVOCATION_FINISHED` event cannot be sent to a build watcher unless all its Bazel events have been either completely sent, or considered as expired, i.e. the publisher of this stream probably crashed.

Internally, the build event service uses Spanner to store build events. The scheduling service publishes lifecycle events and Bazel publishes Bazel events. Events are published in the order of event occurrence on the client side. The order of lifecycle and Bazel events are checked on the server side. The build event service allows users to query build events via the `WatchBuild` API, and it provides both lifecycle and Bazel events in chronological order at real time.

The build event service allows the build watchers to specify event filters which causes only relevant events to be sent to the build watcher. This is particularly useful because the size of all of the events for a build could be huge, e.g. up to hundreds MBs depending on the build, and clients often do not need all events. For example, a command line tool only needs to print the console output to the users. Showing other outputs will likely overwhelm developers. On the other hand, the storage service needs to save all build outputs to facilitate debugging, so it watches all events.

3.8 Build Output Storage Service

The build output storage service is a centralized repository for build and test results at the invocation level. It also stores test-related metrics, such as running time and failures. The storage service watches all remote builds, aggregates the build results and displays them to human users. The service keeps the build output for a longer period of time compared to the build event service, which allows users to check the build output of an old build or analyze the build status history pattern. Since the total size of the build output per day is very large, the service does not provide a permanent storage solution. The service provides a UI where users can query their build execution results by providing the invocation IDs.

4 BUILD SCHEDULING SERVICE

The clients enqueue builds to the scheduling service and the scheduling service dequeues them whenever some workers and executors are available. The worker keeps pulling builds whenever some workspaces are idle, so the scheduling service only needs to decide if it should send the build to the worker based on the executor availability. The scheduling service dequeues enough builds that fill up the target executor occupancy provided by the quota governor.

In this section, we describe the build scheduling service in more detail. We first describe how to determine some critical properties of the build. Then, we describe why and how the Spanner database is used. Finally, we describe the build enqueuing service, the dequeuing service, and the expiring service.

4.1 Critical Build Properties

The build scheduling service conceptually partitions builds into different subgroups based on some build properties. Some build properties are used to sort builds in the priority queue. Those critical properties include build state, quota group, executor types and build priority. The critical build properties do not change during the lifecycle of the build.

The build state is described in Section 3.3 and only `ENQUEUED` builds are considered for dequeuing.

The quota group is determined by the quota governor, and its value is set to the PA of the actual user that creates the build. The user could be human or an automation tool.

The executor types of a build (before actually running the build) are computed from the build flags (e.g. `--ios` indicating Mac build), the tags that specifies executor type in any of the build targets (e.g. `requires-gpu` tag in Figure 1), and the executor type usage history of the build target rules (e.g. since `swift_binary` targets use a lot of Mac executors in the past, so any build with `swift_binary` target would use Mac executors). Note that all builds use x86 executors but they can use more executor types.

The build priority is derived from the tool used to create the build. Table 3 shows all possible build priorities and some examples. `EMERGENCY` builds take the highest priority and are dequeued immediately. `INTERACTIVE` builds are waited by human users and should be dequeued in few a seconds. `AUTOMATED` builds are important but no human user is waiting at the moment, so these should be dequeued in a couple of seconds to minutes. `BATCH` builds are not important and can be delayed for a long time. In practice, `EMERGENCY` builds are rare. Most `INTERACTIVE` and `AUTOMATED` builds are often created and dequeued at peak hours (9am-5pm). `BATCH` builds are dequeued and executed outside of peak hours.

4.2 Spanner Database

The build scheduling service uses the Spanner database to keep track of any build state change. The reason to use the Spanner database is to avoid data loss or invalid system states caused by server shutdown. In production, we run multiple scheduling servers to enqueue and dequeue builds, and it's not rare that some servers are shut down by the server manager for maintenance [15]. When the server is back up, it can continue enqueue or dequeue builds without worrying about losing builds or handling invalid build state caused by the last shutdown. Thus, the Spanner database allows us to build stateless servers.

We use a Spanner secondary index (Section 2.2) to create a queue of builds ordered by build state, quota group, build priority and request time. Only `ENQUEUED` builds are considered for dequeuing. Conceptually, the scheduling service keeps a queue of enqueued builds per quota group, and the queue is ordered by the build priority from highest to lowest. Within the same priority band, the builds are sorted in chronological order of their requested time. When a build is first enqueued in the database or its state is changed by a transaction, Spanner automatically reindexes the build to the correct position at the transaction commit time.

4.3 Build Enqueuing Service

The enqueuing service provides the `CreateBuild`, `GetBuild` and `CancelBuild` APIs. We run many enqueuing service jobs across multiple geographical locations to (1) handle a large number of requests/queries per second (QPS); (2) avoid multiple machine failures in a single machine cluster; and (3) route build requests to the nearest geographical location to reduce network latencies.

The `CreateBuild` API is the entry point to the build service system. After the client calls `CreateBuild` with the build to run, the enqueuing service first generates a globally unique build ID using type 4 UUID [14]. Then, the service finds all build properties, e.g. quota group and executor types. Next, the enqueuing service uses a machine learning model to predict the estimated ESU of the build. This helps the dequeuing service to know how expensive the build is and decide if there is enough resource to run the build at the moment. Finally, the enqueuing service persists all build information to the database and marks the build as `ENQUEUED`, and returns the unique build ID to the client.

Predicting the estimated ESU occupancy of a build is important to determine if the build can be dequeued. If the sum of the estimated occupancy of all in-progress builds is less than the target occupancy given by the quota governor, then the dequeuing service would dequeue some builds to fill up the gap. Otherwise, the dequeuing service would wait until some builds are finished before dequeuing new builds. This strategy ensures that the executor cluster is busy running actions if there are builds waiting in the queue.

We use a linear regression model in TensorFlow [8] to predict the ESU each build uses. Each executor type has a separate model. We train our models using the data in the past 17 days. The data is split into 95% training and 5% testing. The labels are the actual ESU usage of finished builds, which is stored to the disk by the executor cluster. The training pipeline runs continuously and deploys the new models every day. This helps the system to capture the most recent data distribution, and be able to handle builds with new flags or targets. The feature space includes the followings:

- Build command name and flags. Build command name includes `build` and `test`, etc. For example, the `build` command compiles, links and creates code libraries. The `test` command runs the tests. Build flags include `--copt` and `--android_sdk`, etc. For example, the `--copt` flag specifies the options passed to the C compiler. The `--android_sdk` flag specifies the Android SDK and library used to build Android apps. We treat build flag features as categorical features and each feature can have multiple values. Flags can have out of vocabulary (OOV) values as well. The intuition is that the build flags can affect the actions Bazel generates and the ESU used by the build.
- Build targets and packages. The build target feature is the full path to the target, e.g. `path/to/package:target`. Each target has a unique path which is the path to the package that contains the build specification followed by the actual target name declared in the spec. The build package feature is simply the target prefix without the target name. Targets and packages are multivalent

Table 3: Build priorities

Priority	Description	Example
EMERGENCY	The build is critical and should be executed immediately.	A build that is required for an emergency production bug fix.
INTERACTIVE	The build is important and some human user is waiting for the build.	A presubmit build that blocks code submission.
AUTOMATED	The build is important but no human user is waiting for the build.	A postsubmit build that checks if all tests pass at a given commit.
BATCH	The build is not important and whether it is executed or not blocks nothing.	A nightly build that computes code coverage for some submitted code.

features which can have multiple values for each build. Moreover, we use the target count and package count as numeric features. The intuition is that some targets are significantly more expensive than others, and some packages may contain more expensive targets. Builds with more targets or packages are likely to be more expensive.

- Build properties like quota group and build priority. We found that some PAs may have different ESU occupancy patterns compared to other PAs. For example, the machine learning PA often sends builds that require more GPU/TPU ESUs than other PAs. Build priority is also useful to differentiate build occupancy patterns. `BATCH` builds typically occupy more ESUs than builds of other priorities, because tools that use `BATCH` priority, e.g. coverage analysis tools, often send more expensive builds.

We use an off-the-shelf automated blackbox optimization tool similar to AutoML [12] to search for a set of features from the entire feature space that gives the lowest average loss. The selected feature set also includes synthetic feature crosses [4] of up to 6 basic features per cross. For example, one of the feature crosses we use is the combination of command name, iOS sdk version, XCode version and package, which turns out to be a useful feature for predicting Mac ESUs a build may use.

4.4 Build Dequeuing Service

Conceptually, each quota group has a priority queue of builds. The dequeuing service selects the queue using the weighted random selection. Since x86 executors are used a lot more often than other types of executors, each quota group weight is proportional to the x86 ESUs capacity of each quota group. All quota group weights sum up to 1. This means that builds from the PA with more x86 ESU quota are more likely to be attempted for dequeuing.

Once the build queue of a given quota group is selected, the dequeuing service reads a limited number of builds at the head of the queue. The reason to not read the entire queue of builds is that the total number of builds in a queue could be very large and iterating over all builds in the queue would delay dequeuing builds of other quota groups. Note that if we keep dequeuing the builds from the head of the queue, all builds in the queue will eventually be dequeued.

Given a quota group, algorithm 1 illustrates how the dequeuing service decides which builds can be dequeued. The algorithm takes as input a list of builds from the head of the queue *buildsToDequeue*, the target occupancy for each executor type *targetOccupancyByType*, the total estimated occupancy of in-progress builds for each executor type *inProgressOccupancyByType*. The output is the dequeueable builds *dequeueableBuilds*. The algorithm first sets *dequeueableBuilds* to an empty list and sets *reservedOccupancyByType* to an empty map with a default value of 0. *reservedOccupancyByType* keeps the accumulated estimated occupancy reserved so far for each executor type. For each build in *buildsToDequeue*, we get all executor types of the builds. For each executor type, we get the corresponding target occupancy, total in-progress build occupancy, reserved occupancy so far, and compute the remaining occupancy to fill up. Then, we get the estimated occupancy of the build for the executor type. The estimated occupancy comes from the machine learning model in the enqueuing service. If the remaining occupancy is less than the estimated occupancy of the build, then the build is throttled on that executor type. Next, we reserve the estimated occupancy of the build when considering the next build. If the build is not throttled on any executor type, then it is considered as dequeueable. Finally, we return dequeueable builds.

Algorithm 1: Dequeuing algorithm

Input: List of builds `buildsToDequeue`; Target occupancy *targetOccupancyByType*; Total estimated occupancy of in-progress builds *inProgressOccupancyByType*.
Output: Dequeueable builds.

```
1  dequeueableBuilds = []
2  reservedOccupancyByType = defaultdict(int) // default value is 0
3  foreach build ∈ buildsToDequeue do
4      isThrottled = False
5      foreach type ∈ getExecutorTypes(build) do
6          targetOccupancy = targetOccupancies[type]
7          inProgressOccupancy = inProgressOccupancyByType[type]
8          reservedOccupancy = reservedOccupancyByType[type]
9          remainingOccupancy = targetOccupancy -
             inProgressOccupancy - reservedOccupancy
10         buildOccupancy = getEstimatedOccupancy(build, type)
11         if remainingOccupancy < buildOccupancy then
12             isThrottled = True
13         reservedOccupancyByType[type] += buildOccupancy
14     if not isThrottled then
15         dequeueableBuilds.append(build)
16 return dequeueableBuilds
```

The main idea of Algorithm 1 is that the available ESUs are reserved by the previously iterated builds even if they cannot be dequeued. This avoids the problem when an expensive high priority

build is always throttled and cheaper low priority builds that appear after the high priority build are always dequeued instead.

Once the service finds all dequeueable builds, it needs to choose an appropriate workspace on a Bazel worker, or create one if necessary. A well-chosen workspace can increase the build speed by an order of magnitude by reusing the various cached results from the previous execution. The total number of workspaces of all Bazel workers is very large so it is highly likely that we can find a workspace that previously executed a very similar build, thus reducing the amount of work needed to execute the current build. We have observed that builds that execute the same targets as a previous build are effectively no-ops using this technique.

The dequeuing service has two kinds of workspace selection algorithms as follows:

- The first algorithm computes a hash of the relevant build information and then compares it with the hash of the previously running build in each workspace. The algorithm dequeues the build to the workspace with a matching hash. The build hash is computed from (1) the code repository branch name at which the build is initiated; (2) the build flags and targets, where they can be in different order but still result in the same hash; and (3) the Bazel version required to run the build. It is worth mentioning that the build hash does not depend on the Bazel command name or the base revision of the change. When the service dequeues a build, it sends the build hash to the workspace. The workspace keeps the hash to be able to compare it with the hash of the next build. The intuition is that the target dependency graph of the new build is similar to that of the old build if both builds share the same branch, flags and targets. The reason to include the Bazel version in the hash is to avoid restarting the Bazel process due to version differences.
- The second algorithm computes a set of hashes based on the build's flags and target prefixes. For a build command `bazel build --flag a/b:t1 a/c:t2`, the algorithm computes a set of hashes for each target. For target `a/b:t1`, the hashes are `hash("--flag", "a/b:t1")` and `hash("--flag", "a/b")`. For target `a/c:t2`, the hashes are `hash("--flag", "a/c:t2")` and `hash("--flag", "a/c")`. Then, the algorithm checks if the hashes of each target prefix ever appear in the target prefix hashes in any workspace. If each set of target prefix hashes overlaps with the set of target prefix hashes in the workspace, then the workspace is selected if the Bazel version matches as well. Otherwise, the workspace is not selected. Once a workspace is selected for dequeuing the build, then the dequeuing service unions the sets of hashes of all target prefixes and passes the resulting set of hashes to the workspace. The workspace keeps the set of target prefix hashes for comparison when participating in the next round selection. This method is effective because of the overlap in dependencies of targets that share common prefixes.

The dequeuing service uses the first algorithm to find a workspace, and falls back to the second algorithm if the first algorithm cannot find a match. If the second algorithm cannot find a workspace as well, then the service creates a new workspace to run the build. If the new workspace makes the worker exceed its capacity, then an unused workspace will be shut down.

It is critical that the Bazel process be kept running for as long as possible, because most caches are lost when the process is shut down. The Bazel worker runs a background process which continuously monitors the memory usage on the worker, and shuts down Bazel processes only when the memory usage on the worker exceeds a certain threshold.

After a build is finished, the worker notifies the scheduling service about the final build status, e.g. succeeded, failed, canceled, etc. Then, the scheduling service sets the build state to `FINISHED`, which is the end of the build lifecycle.

4.5 Build Expiring Service

Some low priority builds may stay in the queue for a long time. However, if a build is queued for too long, then the result might be obsolete and no longer valid. The build expiring service expires those builds which stay in the queue for more than a given time period, and sets the build state to `FINISHED`.

Some builds are very expensive and may take a long time to run. We do not want to let a build to run forever, so the build expiring service expires the build invocation if it runs for more than a few hours and sets the build state to `FINISHED`. Often these long running builds are problematic, and requires restructuring the build specification or splitting the build into multiple smaller builds.

Section 3.4 mentions that each Bazel worker periodically renews the lease for the build. This makes sure that the scheduling service knows the build is running normally. If for some reason the build lease is not renewed, the build expiring service expires the lease and sets the build state to `ENQUEUED` again. This allows other workers to get and run the build.

5 EXPERIENCE

In this section, we first describe the production setup of the build service system. Then, we describe the scalability of the system. Finally, we describe more details about the usefulness of our occupancy models and workspace selection algorithm.

5.1 Production Setup

To scale up the build service system, each component is deployed on multiple servers and those servers are widely spread across the world. The load balancer then routes client requests to the nearest group of servers and evenly distributes the traffic to each server.

Google has many datacenters and each data center has many machine clusters. A metro is one or more datacenters that share the same common metropolitan network infrastructure, routing policy, and other associated network resources. Round trip time between any locations within a metro should not exceed few milliseconds. The core components of the build service system are located in a way to minimize the network latency of the build service. Some example configurations are listed below:

- The dequeuing service reads and writes the Spanner database a lot. So the dequeuing servers reside in the same metro as our Spanner database to maximize the dequeuing speed.
- The Bazel process waits until all build events are published to the build event service before finishing the build. So the build event servers reside in the same metro as the Bazel workers to minimize event publishing latency.

Table 4: Service API QPS

Service	Avg	Min	Median	Max
CreateBuild	102-253	0-173	89-244	213-833
GetBuild	526-4325	0-529	308-2712	2276-39466
CancelBuild	0-137	0-1	1-3	0-2841
GetNextBuild	94-223	0-166	89-220	177-529
FinishBuild	93-220	0-172	87-220	169-411
WatchBuild	620-3600	0-1757	735-3926	1165-6331

- The Bazel process interacts with the executor cluster a lot by sending actions and receiving results. So the Bazel workers reside in the same metro as the executor cluster to minimize the accumulative network latencies.

The enqueuing server does not need to be in the same metro as the dequeuing server because both services do not directly interact with each other via RPC. Instead, they share the same Spanner database.

The output storage service does not need to reside in the same metro as the build event service, because the storage service does not block the build execution. Moreover, it is acceptable for developers to wait a couple of more seconds after the build is finished to see the build output, thus the network latency is not critical.

5.2 Build Service System Scalability

The entire build service system runs more than 15 millions of builds on average and up to 25 millions of builds per day. We have not load tested the entire build service system, but we believe that it can handle millions of more builds. The build service system supports more than 50,000 developers' daily activities in Google.

Table 4 shows the average, minimum, median and maximum of queries per second (QPS), per service API, over the last 24 hour window between 2019/10/01 and 2019/12/31. The average `CreateBuild` QPS over the last 24 hours ranges from 102 to 253. On 2019/10/20 Sun at around 3:00am, the `CreateBuild` QPS reaches the minimum of 102. On 2019/11/26 Tue at around 4:30pm, the `CreateBuild` QPS reaches the maximum of 253. Typically, all service APIs reach the minimum value on weekends outside of peak hours and maximum value on weekdays within peak hours. The average `GetBuild` QPS ranges from 526 to 4325. `GetBuild` has a larger QPS compared to `CreateBuild` because it is often invoked multiple times to query the build status during the build lifecycle. The average `CancelBuild` QPS ranges from 0 to 137. `CancelBuild` is not common in practice, but sometimes it is used to cancel problematic builds, e.g. builds that are too big and cause out of memory errors in servers. The average `GetNextBuild` QPS ranges from 94 to 223. The average `FinishBuild` QPS ranges from 93 to 220. The QPS of `GetNextBuild` and `FinishBuild` are comparable to `CreateBuild` because each build often corresponds to a single invocation for each of the `CreateBuild`, `GetNextBuild` and `FinishBuild` APIs. The average `WatchBuild` QPS ranges from 620 to 3600. `WatchBuild` has a larger QPS compared to `CreateBuild` because multiple clients can watch the same build. A single client may also watch the same build multiple times because `WatchBuild` is a long running API and has a higher RPC failure rate. In summary, Table 4 shows that the build service system is able to handle a large amount of throughput.

Table 5: Service API latency in milliseconds

Service	Avg	Min	Median	Max
CreateBuild	111-3k	86-146	114-191	124-48k
GetBuild	15-38	4-16	15-26	28-608
CancelBuild	108-432	26-145	91-208	170-41k
GetNextBuild	106k-235k	28k-102k	98k-166k	199k-2064k
FinishBuild	184-689	131-250	177-297	213-71k
WatchBuild	24k-83k	15k-48k	24k-78k	44k-200k

Table 6: Build count, size and time spent for each priority

Metric\Priority	EMERG	INTER	AUTO	BATCH
Build Count (%)	0.4	40.1	46.0	13.6
Build Size (KB)	1.5	5.8	22.5	18.4
Target Count	10	22	143	201
Build Queuing	2.4	31.0	282.8	2561.8
Worker Process	4.1	3.3	4.1	3.8
Action Queuing	0.1	0.5	2.3	5.2
Execution	127.6	89.8	133.4	176.1

Table 5 shows the average, minimum, median and maximum of latency in milliseconds, per service API, over the last 24 hour window between 2019/10/01 and 2019/12/31. The average `CreateBuild` latency ranges from 111ms to 3s. Typically, `CreateBuild` is designed to have a small latency for better user experience. The average `GetBuild` latency ranges from 15ms to 38ms. The `GetBuild` latency is small because it only involves reading the build information from Spanner and returning it to the user. The average `CancelBuild` latency ranges from 108ms to 432ms. The average `GetNextBuild` latency ranges from 106s to 235s, which is significantly longer than other API latencies as expected. The average `FinishBuild` latency ranges from 184ms to 689ms. The latencies of `CreateBuild`, `CancelBuild` and `FinishBuild` are similar, because they involve read-modify-write [11] Spanner transactions. The average `WatchBuild` latency ranges from 24s to 83s. The `WatchBuild` API needs to send back a long sequence of events to the client, so its latency is larger.

All build service APIs offer high availability, i.e. the successful rate of the APIs, to the clients. `CreateBuild`, `GetBuild` and `WatchBuild` offer 99.9% availability. `CancelBuild`, `GetNextBuild` and `FinishBuild` offer 99.5% availability.

It is worth to mention that the build output storage service has 30,000 to 60,000 daily active human users who access their build output via web browsers.

Table 6 shows the average build count in percentages, the average size in KB, and average time in seconds spent at each phase. The data is broken down by build priority and collected between 2019/10/01 and 2019/12/31. The table shows that `EMERGENCY` builds only account for 0.4% of the total builds. `INTERACTIVE` and `AUTOMATED` builds are comparable and they account for 40.1% and 46.0% of the builds, respectively. `BATCH` builds account for 13.6% of the total builds. The build size reflects the number of flags, targets and size of the metadata. Larger build sizes typically indicate more expensive builds (in terms of occupied ESU). However, builds with a small number of expensive targets could be more expensive than builds with a large number of cheap targets. `EMERGENCY` builds are smaller in size and they are of 1.5KB on average. `INTERACTIVE` builds

are larger in size and they are of 5.8KB on average. AUTOMATED and BATCH builds are comparable and they are of 22.5KB and 18.4KB on average, respectively. EMERGENCY, INTERACTIVE, AUTOMATED and BATCH priority builds have on average 10, 22, 143 and 201 targets, respectively. EMERGENCY builds are the highest priority builds so they only spend 2.4s on average in the scheduling service. INTERACTIVE builds are the second highest priority builds and they spend 31.0s on average in the scheduling service. AUTOMATED and BATCH builds are lower priority builds and they spend 282.8s and 2561.8s on average in the scheduling service. It takes roughly the same amount of time (3.3s to 4.1s) for the Bazel workers to prepare the environment before starting the build execution across all build priorities. Similar to the build queuing time, the action queuing time on the executor cluster is smaller for higher priority builds. The average action queuing time for EMERGENCY, INTERACTIVE, AUTOMATED and BATCH builds are 0.1s, 0.5s, 2.3s and 5.2s, respectively. The execution time spent on the executor cluster for all builds are comparable and roughly proportional to the build size. The average execution time for EMERGENCY, INTERACTIVE, AUTOMATED and BATCH builds are 127.6s, 89.8s, 133.4s and 176.1s. The average execution time of EMERGENCY builds is larger because they are biased by a small number of long running builds that execute tests hundreds of times. INTERACTIVE builds are often human triggered builds, and they are typically smaller in size and faster to execute because developers often only run a small set of targets affected by their change. AUTOMATED and BATCH builds are often tool triggered builds and those builds are larger in size and slower to execute.

The executor action cache hit rate is often around 99% which speeds up the build execution a lot. Without the action cache, the build service system would not be able to support millions of builds.

5.3 Build Scheduling Service Performance

We have discussed some QPS and latency metrics for the scheduling service (Section 5.2). In this section, we mainly discuss the usefulness of the occupancy model (Section 4.3) and the workspace selection algorithm (Section 4.4).

The error of an occupancy model is defined as the difference between the actual ESU and the estimated ESU. The mean square error and the average absolute error of the x86 occupancy model is 312.5 and 5.7, respectively. The mean square error and the average absolute error of the Mac occupancy model is 522.1 and 8.1, respectively. There are more builds that require x86 executors than Mac executors, so the x86 occupancy model has more training data and is more accurate than the Mac occupancy model. Other occupancy models have a lower accuracy compared to the x86 and Mac occupancy models, because only a small fraction of builds require those executor types and those occupancy models have even less training data. Moreover, the executor action cache also makes it hard for the occupancy model to be accurate. Because the same build can use some ESU when there is no cache hit, or 0 ESU when all generated actions hit the cache.

In practice, we find that the occupancy model is useful in making the target executor occupancy more stable and smooth, which helps keeping the executor cluster fully occupied. If we use a default occupancy model which always returns 1 ESU for each executor type per build, the scheduling service would overschedule builds which causes some expensive low priority builds to run and those builds would prevent high priority builds from executing.

The workspace selection algorithm only affects builds that can reuse Bazel's target dependency or action cache. We collect data from 21 days before and after this feature is enabled, and find that the algorithm is able to reduce the average build execution time from 64.1s to 55.4s, which is a 13.6% improvement. Note that the data is not collected between 2019/10/01 and 2019/12/31, and the data excludes builds that are configured to clean the Bazel cache before running.

6 RELATED WORK

CloudBuild [10] is the closest related work to our build service system. It is Microsoft's distributed and caching build service. The main differences are listed below:

- A build in Microsoft contains coarse-grained projects. In Google, a build contains fine-grained targets.
- CloudBuild's distributed cache is similar to the global cache in the executor cluster described in Section 3.5. Microsoft has many codebases and these codebases have fewer shared dependencies. In comparison, Google has a monolithic codebase which results in higher chances in sharing build targets among builds. So CloudBuild's cache hit rate is lower than our executor cache hit rate.
- CloudBuild executes around 20k builds per day and is used by around 4000 developers. In comparison, our build service system executes more than 15 million builds on average per day and is used by more than 50,000 developers.

Another related work is distcc [3], which is a distributed C/C++ compiler publicly available on GitHub. The unit of work is a preprocessed source code file that is sent over the network and compiled remotely. In comparison, our build service system supports many languages and test executions.

Modern build systems like Maven [7], Gradle [6], Buck [2] are not distributed and do not scale to the problem we face in Google. Both Gradle and Buck support local cache which is similar to Bazel in our build service system.

As far as we know, our build service system handles the largest number of builds daily among all build systems developed in other companies. We also introduce the first build scheduling algorithm that uses machine learning models and PID controllers. We believe our work can be beneficial for designing more scalable build service systems.

7 CONCLUSION

In this work we presented what we believe is the most scalable build service system in the world. The system has evolved for many years and is refined with many optimizations. We discuss each component in the build service system as well as the build APIs. More specifically, we discuss the build scheduling service and algorithms to dequeue builds. We show that our build service system handles more than 15 million builds on average daily. We also show the usefulness of our occupancy models and workspace selection algorithm in the build scheduling service. We believe that the architecture and algorithms described in this paper are useful and can help in designing new scalable build service systems.

REFERENCES

[1] [n.d.]. Bazel build tool. https://bazel.build/.
[2] [n.d.]. Buck build tool. https://buck.build/.
[3] [n.d.]. distcc. https://github.com/distcc/distcc/.
[4] [n.d.]. Feature Cross. https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture/.
[5] [n.d.]. GNU Make. https://www.gnu.org/software/make/.
[6] [n.d.]. Gradle build tool. https://gradle.org/.
[7] [n.d.]. Maven. https://maven.apache.org/.
[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.
[9] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.
[10] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 11–20.
[11] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.
[12] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2018. *Automated Machine Learning: Methods, Systems, Challenges*. Springer. In press, available at http://automl.org/book.
[13] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.
[14] Paul J Leach, Michael Mealling, and Rich Salz. 2005. A universally unique identifier (uuid) urn namespace. (2005).
[15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.
[16] John G Ziegler and Nathaniel B Nichols. 1942. Optimum settings for automatic controllers. *trans. ASME* 64, 11 (1942).