# Automating Build, Test, and Release with Buildbot

Dustin J. Mitchell
Mozilla, Inc.
Email: dustin@v.igoro.us
Tom Prince
Email: tom.prince@ualberta.net

***Abstract*—Buildbot is a mature framework for building continuous integration systems which supports parallel execution of jobs across multiple platforms, flexible integration with version-control systems, extensive status reporting, and more. Beyond the capabilities it shares with similar tools, Buildbot has a number of unique strengths and capabilities, some of them particularly geared toward release processes. This paper summarizes Buildbot's structure, contrasts the framework with related tools, describes some advanced configurations, and highlights ongoing improvements to the framework.**[1]

## I. Introduction

Buildbot is a continuous integration (CI) framework which has been in use for well over 10 years[13]. While it began as a simple build-and-test tool, it has grown into an advanced framework supporting continuous integration testing, continuous deployment, and release engineering. Its design allows an installation to grow with its requirements, beginning with a simple configuration and growing into a multi-master configuration with complex coordination of builds, each containing many sophisticated steps. This flexibility has led to its use in a number of high-profile open-source projects, including Chromium, WebKit, Firefox, Python, and Twisted[10].

Buildbot is but one of many tools commonly used for build, test, and release because, in such an active ecosystem, each tool has strengths that make it the best choice for certain purposes. Buildbot's niche is complex applications requiring extensive customization.

This paper briefly summarizes Buildbot's structure for background purposes then compares its framework to other tools. We also describes some of the tool's unique features and capabilities and finishes with a description of ongoing work in the Buildbot development community.

## II. Buildbot's Structure

At its core, Buildbot is a job scheduling system: it queues jobs (called *builds* for historical reasons), executes the jobs when the required resources are available, and reports the results. Refer to the Buildbot documentation[4] for more detail.

Buildbot's *change sources* watch for commits (generically called *changes*). The built-in change sources support the most common version-control systems with flexible configuration to support a variety of deployments. *Schedulers* react to new data from change sources, external events, or triggers based on clock time (e.g., for a nightly build), and add new *build requests* to the queue.

Each build request comes with a set of source stamps identifying the code from each codebase used for a build. A *source stamp* represents a particular revision and branch in a source code repository. Build requests also specify a *builder* that should perform the task and a set of *properties*, arbitrary key-value pairs giving further detail about the requested build. Each *builder* defines the steps to take in executing a particular type of build. Once the request is selected for execution, the steps are performed sequentially. Each build is executed on exactly one slave.

Buildbot has a distributed master-slave architecture where the *masters* instruct the *slaves* to perform each step of a build in sequence. The slave portion of Buildbot is platform-independent, and many slaves can be connected to each master, running builds in parallel. A slave may execute multiple builds simultaneously, as long as they are for different builders.

Once a build is complete, the framework reports the results—log files and step status—to users and developers via *status listeners*. Buildbot has built-in support for reporting via web, email, irc, Tinderbox, and gerrit.

The Buildbot master and slave are both implemented as Python daemons. Small Buildbot installations are generally composed of one master and tens of slaves while larger installations run tens of masters with hundreds or thousands of slaves.

## III. Related Work

The ecosystem of continuous integration testing tools is a busy one. The Thoughtworks feature matrix[1] lists 28 packages and does not consider cloud-based services such as Travis-CI or TDDium.

### A. A Framework for Continuous Integration

Many CI packages, such as CruiseControl[2] or Jenkins[5], are structured as ready-to-use applications and embody assumptions about the structure of the project and its processes. Users fill in specific details like version control information and build process, but the fundamental design is fixed and options are limited to those envisioned by the authors. This arrangement suits the common cases quite well: there are cookie-cutter tools to automatically build and test Java applications,

Ruby gems, and so on. These tools are not well-suited to more complex cases, such as mixed-language applications or complex release tasks, where those assumptions are violated.

In contrast, Buildbot is designed as a framework: it provides tools for continuous integration and release management, allowing users to implement a system that suits their needs within that structure. While this makes Buildbot particularly well-suited to implementing the complex requirements of release engineering, it also means there is a steeper learning curve than with simpler CI applications.

### B. Configuration Interface

Buildbot does not support web-based configuration but instead requires the user to edit the sample configuration file before the first run. This approach follows the UNIX philosophy of explicit, editable configuration—many Buildbot users keep their configuration in version control—and enables very fine control over the tool. In contrast, the Jenkins web interface invites the user to add a project via the web interface—no configuration file is visible to the user. The Buildbot community has discussed support for web-based configuration, e.g., [3], but no robust implementation has emerged.

### C. Batteries Included

Like many CI applications, Jenkins ships with limited capabilities and uses a plug-in architecture[8] to add additional capabilities. In contrast, Buildbot includes many useful components in the base package and does not support plug-ins. These components address common needs such as version control integration, specialized build tools, and flexible status reporting. In many cases, the first implementation of Buildbot in an organization uses the built-in components, only replacing them with customized implementations as requirements grow. This approach mirrors Python's notion of a "Batteries Included" language.

### D. Pull vs. Push

Job-queuing systems fall into two categories: push, where the central scheduler sends jobs to a worker it knows to be idle, and pull, where workers request jobs from the central scheduler. Tools like Tinderbox[12] operate on a pull model whereby each tinderbox builds the latest revision in a tight loop, reporting status to the central tinderbox server. CruiseControl and Jenkins support both models. Buildbot operates exclusively on a push model: slaves simply perform the operations they receive from the master and then report the results. The advantage of Buildbot's approach is that configuration is centralized on the masters. However, it is not well suited to cases where slave resources also serve other purposes since the Buildbot master lacks the context to determine if a slave is ready to perform a build.

## IV. Advanced Buildbot Configuration

This section highlights some of Buildbot's high-level capabilities and focuses on those that are useful in release management and operation at scale.

### A. Expressive Configuration

Buildbot's configuration file is simply a Python script. While a basic configuration can be written with little knowledge of Python, access to Python's power makes programmatic configuration trivial. For example, a configuration with several identical Windows slaves might include this snippet to automatically generate each slave configuration:

```
for n in range(0, 20):
  c['slaves'].append(
    BuildSlave('win-slave%d' % n, PASSWD))
```

All of the concepts introduced above–change sources, schedulers, slaves, builders, and status listeners–can be configured in this Python script. Basic configurations can use the built-in support while more advanced configurations include custom implementations.

Builder configurations include a list of steps to execute for each build (a *build factory*). These steps can be simple shell commands, built-in source checkout or build commands, or custom implementations. Steps can use and modify the build's properties, treating them as variables to coordinate the progress of the build.

Buildbot supports dynamic reconfiguration without interrupting running builds, allowing adjustments to the build process without downtime.

### B. Multiple Codebases

Buildbot supports the increasingly common practice of building applications from multiple codebases. For example, a mobile application might have three different codebases: "Android", "iOS", and "Common" which contains assets common to both implementations. Buildbot can monitor all three repositories and schedule android builds with source stamps from the "Android" and "Common" codebases when either one changes; likewise for "iOS". Status is reported for the resulting multidimensional revision space.

### C. Scaling and Resiliency

Large-scale Buildbot deployments run with both masters and slaves in a clustered configuration. The masters use a shared relational database (MySQL or Postgres) to coordinate their actions. Multiple identically-configured slaves allow parallel execution of the same type of build for high-volume deployments and resiliency against slave failure.

The same slave can be configured on multiple masters in a cluster and then connect to any one of those masters. Assuming all masters are configured with the same builders, this allows resiliency against master failures. The masters also share the load of processing status and logs, which can be substantial in a very busy cluster.

As an extreme example of Buildbot at scale, Mozilla's Buildbot deployment is comprised of over 40 masters and thousands of slaves. This configuration handled a check-in every 6 minutes and performed 137 hours of build and test per check-in as of August 2012[6].

Only some of Buildbot's data is stored in the database with the remainder stored in on-disk "pickle" files as marshaled

Python objects. Handling these pickles is inefficient, generating high IO load on the masters and considerably slowing processing. More critically, the data on one master's disk is not accessible from another master, so analyzing builds distributed across a cluster is difficult. Section V describes ongoing work to address these shortcomings.

### D. Artifact Handling

For a continuous-integration build, the important result is essentially boolean: did the build or tests fail? For a release, the output is a set of binary artifacts—usually packages—ready for installation by end-users.

Buildbot includes support for a few common package formats, although in most cases the package building process is driven by a project-specific script. Binary artifacts are generated on slaves as part of a build, and are then transmitted somewhere to make them accessible to users (or to the quality assurance team in a larger organization).

Artifact upload steps are included with Buildbot, but the steps transfer the file data using a particularly inefficient mechanism (RPC, rather than a streaming protocol). For any but the smallest installations, they are inadequate, performing poorly and causing performance issues for other builds on the same master. Most installations use SSH (via `scp`) or authenticated HTTP implemented in a shell command instead.

The Buildbot community has discussed integration with external tools[11], but no compelling implementation has yet emerged.

### E. Build Coordination

It is often desirable to perform multiple operations in parallel. This can drastically reduce the end-to-end time (the time between a commit and completion of the last resulting job), and save a great deal of redundant computation.

Because Buildbot runs each build on a single slave, parallel operations must be represented as multiple builds. In programming-language terms, builds form the *basic block* from which more complex control flows are constructed. The remaining operations are to "branch" control into multiple builds, and then "merge" control back when those builds are complete.

The branching operation is accomplished by configuring a scheduler to queue build requests for multiple builders. The resulting set of build requests comprise a *build set* and can be executed in parallel. If slave resources are limited, the resulting builds may not actually execute concurrently, but they will be prioritized by Buildbot's multi-level priority configuration. This technique provides a simple way to branch the control flow, but does not provide a way to perform some further action after all builds are complete.

Buildbot provides two mechanisms for more sophisticated coordination: the *Dependent scheduler*, and the *Triggerable scheduler*. The Dependent scheduler is configured to watch exactly one "upstream" scheduler. When all builds in a buildset initiated by this upstream scheduler complete successfully, the Dependent scheduler initiates its own set of builds. For example, a Dependent scheduler might be used to schedule several test builds after all compilation builds are complete, allowing multiple test builds to share the same compiled artifacts.

The Triggerable scheduler initiates builds for its configured builders when triggered by a step in another build. That step can either wait for the triggered builds to complete, or continue immediately. Waiting for the triggered builds provides a flexible method of merging control flow. In a release build, for example, a Triggerable scheduler could be used to build localized versions of the application, one build per locale, triggered by a step in the compile build for each platform.

The most complex control flows are implemented as a single controlling build which triggers the necessary builds in sequence. For example, such a build might trigger "compile" builds, followed by "test" builds if compile is successful, and followed by "package" builds, with each initiating builds on different slaves.

Buildbot has ample room for improvement in support for build coordination. First, the framework provides no convenient means to summarize the output of a process that is spread over multiple builds. Second, unusual branch and merge conditions, such as continuing when 80% of locales build successfully, are difficult to implement. Finally, a controlling build must be assigned to a slave, even though it never performs any operations on that slave.

In general, creating a concurrent build process is similar to assembly-language programming in that high-level constructs are not available. Ideally, Buildbot would make common structures like functions, loops, and conditionals available to users to structure these operations. This is an as-yet unexplored area for future work.

## V. Ongoing Developments

### A. Nine: Fully Distributed, Scalable Architecture

Buildbot's developers, including the authors, are working to reformulate the tool as a collection loosely coupled modular components. These components are linked by a common database back-end and a message bus. They can be combined arbitrarily across multiple hosts to create build, test, and release systems customized to the needs of the user's organization.

This design allows redundant clustered implementations which are resilient to the failure of any single component and able to scale to meet load requirements. The language-agnostic interfaces between components allow integration with other tools in an organization. For example, Buildbot messages can be duplicated into an enterprise message bus and used to correlate build activity with other enterprise events.

The design also fixes a substantial performance problem in current versions of Buildbot. Web status views are currently generated on the buildmaster using inefficient methods which can take up to 0.8s per page and which block all other activity on the master during generation. This makes it unwise or impossible to expose the Buildbot web interface to a wide audience. The web status views are rendered in the browser

in the new design, significantly reducing load on masters. As a further advantage, the master can send status updates directly to the web browser so that users see updated results without reloading the page.

The architecture is split into three layers: Database (DB), Message Queue (MQ), and Data. The DB layer abstracts a shared relational database, supporting several common database packages. The API presented by this layer is Python-only and is closely tied to the database schema.

The MQ layer implements a plug-able message queuing system. Like the DB layer, it supports several external backends, including RabbitMQ and ZeroMQ. This layer distributes messages among all masters, providing notification of new events. As an example, when a build is complete, the master processing that build sends a message. Message consumers might react to that message by sending email, notifying users on IRC, or updating the display in a user's browser.

The Data layer combines access to stored state and messages, and exposes a well-defined API that can be used both within Buildbot and by external tools. Buildbot components use the data API to communicate with other components. This layer's design is influenced by REST and is available both via HTTP and as a native Python interface.

This reformulation is nicknamed 'nine', because it is targeting Buildbot version 0.9.0. It is a significant refactor of the Buildbot codebase, and work is slow due to the need to maintain compatibility. As of this writing, most of the Data API is complete, and a proof-of-concept web interface has been written, but all status listeners remain unmodified. Time estimates are difficult for any open-source project, but it is likely that the project will be complete in late 2014.

### B. One-Oh: Well-Defined APIs

Buildbot is a fairly mature project and has grown considerably since it was created over a decade ago. Much of that growth had been organic, driven by small contributions from users with little coordination of effort. The documentation, while describing the configuration options, does not give much detail on how to customize Buildbot—hardly ideal for an application framework. Lacking other guidance, users treat the source code as documentation. As a result, the effective API surface of Buildbot has become large and ill-defined. This has led to difficulty for users in upgrading Buildbot and made development of Buildbot itself difficult as quirks of the existing APIs hinder implementation of new features.

The fix is to define new APIs, make compatibility commitments about those APIs, and put tools in place to guarantee correctness and compatibility. These APIs will include the Data API, as well as Python interfaces for custom change sources, schedulers, steps, and status-handling components. The APIs will be fully documented and include thorough unit and interface tests measuring the implementation's behavior and adherence to the documentation. Any changes to the API will be handled carefully and communicated clearly to the Buildbot user community.

It is unrealistic to expect that the first Buildbot release implementing the Data API will get everything right. Rather than commit to strict API compatibility beginning with Buildbot-0.9.0, the developers plan to allow the new APIs to "settle in" throughout the 0.9.x series. Once Buildbot reaches version 1.0.0, the APIs will be fixed, and changes will be handled using accepted semantic-versioning techniques[9].

## VI. Conclusion

Buildbot's flexibility can seem daunting for users with simple requirements, particularly in comparison to other tools. That same flexibility makes it ideal for large, sophisticated build, test, and release implementations. The tool is not without its weaknesses, several of which are indicated above. The 'nine' project is an effort to address some of these weaknesses, and lay the groundwork for addressing others. The 'one-oh' project will create a more hacker-friendly environment which should have a magnifying effect on the rate of development of new features and capabilities.

As an open source project, Buildbot welcomes all potential users and contributors. The organization is pending membership in the Software Freedom Conservancy. It has also participated in the Google Summer of Code for two years and hosts sprints (multi-day programming sessions) at PyCon every year. The #buildbot IRC channel is an active and positive communication channel, as is the mailing list. This environment has encouraged a broad developer community, with 71 contributors committing code in the last year alone and many more offering advice and support to other users.

## References


[1] *CI Feature Matrix* [online] 2013, http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix (Accessed: 3 March 2013).

[2] *CruiseControl Getting Started* [online] 2013, http://cruisecontrol.sourceforge.net/gettingstarted.html (Accessed: 3 March 2013).

[3] Garboden, Mark. *Buildbot config wizard* [online] 2006, http://comments.gmane.org/gmane.comp.python.buildbot.devel/554 (Accessed: 3 March 2013).

[4] *Introduction - Buildbot v0.8.7p1 documentation* [online] 2013, http://buildbot.net/buildbot/docs/current/manual/introduction.html (Accessed: 3 March 2013).

[5] *Meet Jenkins - Jenkins - Jenkins Wiki* [online] 2013, https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins (Accessed: 3 March 2013).

[6] O'Duinn, John. *137 hours compute hours every ˜6 minutes* [online] 2012, http://oduinn.com/blog/2012/08/21/137-hours-compute-hours-every-6-minutes/ (Accessed: 3 March 2013).

[7] *Open Source Overview* [online] 2013, http://www.jfrog.com/home/v_artifactory_opensource_overview (Accessed: 3 March 2013).

[8] *Plugins - Jenkins - Jenkins Wiki* [online] 2013, https://wiki.jenkins-ci.org/display/JENKINS/Plugins (Accessed: 3 March 2013).

[9] Prestoe-Werner, Tom. *Semantic Versioning 2.0.0-rc.1* [online] 2012, http://semver.org (Accessed: 3 March 2013).

[10] *SuccessStories - Buildbot* [online] 2012, http://buildbot.net/trac/wiki/SuccessStories (Accessed: 3 March 2013).

[11] Tardy, Pierre. *ArtifactStorage management design doc* [online] 2012, https://github.com/buildbot/buildbot/pull/547 (Accessed: 3 March 2013).

[12] *Tinderbox — MDN* [online] 2012, https://developer.mozilla.org/en-US/docs/Tinderbox (Accessed: 3 March 2013)

[13] Warner, Brian. *Buildbot 0.3.1* [online] 2003, https://raw.github.com/buildbot/buildbot/master/master/docs/relnotes/0.3.1.txt (Accessed: 3 March 2013)