Amazon's Build System

amazon.md

Prologue

I wrote this answer on stackexchange, here: https://stackoverflow.com/posts/12597919/

It was wrongly deleted for containing "proprietary information" years later. I think that's bullshit so I am posting it here. Come at me.

The Question

Amazon is a SOA system with 100s of services (or so says Amazon Chief Technology Officer Werner Vogels). How do they handle build and release?

Anyone know what they use or the overall structure? I'll bet it would be very interesting to read about their lessons learned. I worked on Amazon's Build Team for approximately 3 years, from 2006 to 2009. Amazon has produced a world-class build and deployment system which is pretty difficult to imagine and unmatched in open source. It is really sad (and a personal goal of mine to change) that there is nothing even close that is freely available.

My Answer

It is my assertion that the details included in this post are, though possibly not obvious, inevitable discoveries of any sufficiently large engineering organization and the only way to build and deploy quality software in a massive SOAservice-oriented architecture (SOA) environment where one team blocking on another is considered more harmful than integration pain (for an alternative perspective, see any organization which hates topic branches and insists everyone commit directly to trunk and "integrate early" - works less well for SOA). As such, I do not consider this level of detail to be proprietary information, especially combined with the fact that it is 3 years old.

1. Reproducibility - They should guarantee the ability to reproduce any artifact that has ever been produced in the past (or any artifact tagged as "released" at a minimum). Furthermore, they should guarantee the ability to produce any artifact with a known delta - i.e. "this exact version, but with only this one bug fixed". This is critical to know you are only making the minimum change, and not introducing risk into a production system.
2. Consistency - In the past (though to a much lesser extent these days) Amazon had a lot of C/C++ code. Regardless of the language you use (but especially in languages where ABI compatabilitycompatibility is "a thing") you need to guarantee consistency. This means knowing that a particular set of artifacts all work together and are binary compatible. In C/C++ this means ABI compatibility and in Java this might mean that for all libraries on the classpath there is exactly one specific version which works with all the other jars on the classpath (i.e. spring 2.5 or 3.0 but not both, they have different APIs). Ideally, this also involves running unit tests (and possibly other acceptance tests) to confirm nothing is broken - when I was at Amazon tests were far less common than I HOPE they are now...
3. Change Management - The two features above mean a whole lot less if you cannot manage your changes. 100s of services run by individual teams with shared library dependencies inevitably means owners of shared code will need to perform migrations. Team X may need version 1.0 of your library, but team Y might need version 1.1, and the two versions might not be ABI or API compatible. Forcing you to make all your versions compatible forever is an undue burden. Forcing all your clients to suddenly migrate to your new API all at once is also an undue burden. Therefore, your build pipeline must allow some teams to consume older versions, while others consume newer versions. From the bottom up, each piece of your dependency graph must migrate to the new version, but anyone still on the old version must continue to be able to reproduce their build (including making bug fixes). This usually leads to having "major versions" and "minor versions", where minor version changes are non-breaking changes that are "automatically picked up" but major version changes are "picked up by request only". You

then run a migration by having each piece of your dependency graph, from the bottom up, migrate to the new major version.

Amazon's build system is called Brazil (haha! Lots of things are called Brazil within Amazon, it is a crazily overloaded term)

The main build driver is a mess 'o Perl scripts that generate makefiles. The build system is bootstrapped by a minimal Perl script which assumes only base Perl deps and GCC are available, and downloads all other dependencies.

The build system is "data driven", meaning there are configuration files which explain what to build. Code is broken up into units called "packages", each has a configuration file which says what to build, what artifacts are produced, what the package depends on, and frequently details about how it is deployed as well. The build system can be run on the desktop to develop and test, and in the package builder as well (see next point)

The build system ensures that nothing is depended upon besides GCC/Glibc, and your explicitly listed dependencies, by ensuring nothing else is on your linker line / classpath / PERL5LIB / whatever for your language. THis This is critical to reproducibility. If some randorandom library in your home directory was accidentally depended upon, builds would succeed locally but then fail later for other people, and if those dependencies were not available elsewhere, could never be reproduced.

There is a massive scheduling build system called "Package Builder" which (rather than continuously) developers can request makes a release build. Every release build is reproducible and its artifacts are kept effectively forever (backed by S3 last I heard).

Each build requested is built "in a version set", which is a list of package versions that are known to be consistent together. To change the versions in a versionset, you "build package X against versionset Y" - then using all the dependencies listed in versionset Y, the new version is X is built. X could be multiple packages for lock-step changes (called a coordinated build). Only after all packages are build successfully (and, should they have tests, all tests run successfully), the artifacts are published and the version set is updated so future builds against that version set will use the new package versions as dependencies. Builds are serialized on version set, so only one build can occur against each version set at the same time, otherwise you might have two concurrent changes break something without the build system noticing. This means that if each team has their own version set (which makes sense) no one team is blocked on antoheranother team's builds.

There is a "default versionset" called live which, when you create a new versionset, is used as the source to figure out what versions to take. So if you own shared code, you "publish" it by building it against the live version set.

There is a deployment system run by a completely different team called Apollo (which was written to replace Houston - as in "Houston, we have a problem!" - it's a pretty funny story). Apollo is probably the single most critical piece of infrastructure. A deployment system such as Apollo must takes artifacts from a consistent set of versions produced by the build system (in Brazil's case, a version set) and transform them to artifacts ready to deploy (usually a trivial transform), then put them onto hosts, be they desktops or servers in data centers. Apollo has probably deployed petabytes of data since its inception.

Apollo uses network disk tricks to efficiently move bits around so it doesn't have to copy to each and every box, but then builds symlink trees to get the files symlinked into the place they need to be. Generally, applications will live in /apollo/ENVIRONMENT_NAME/{lib, bin, etc}. Most applications use a wrapper that adjusts the dynamic link path to include the environment's lib dir, etc. so only dependencies in your environment are used. In this way multiple apps with different dependencies (at different versions) can all run on the same machine, so long as they are in different "environments". This, like in step 4 above, is CRITICAL because if applications depend upon things outside the environment, then those applications are not running reproducibly and future deployments of the same packages may not behave the same way.

Apollo has startup and shutdown scripts which can be run before/during/after a deployment of an environment - think of it as a re-implementation of a domain-specific init.dinit.d. It's very similar and not particularly special, but important because you want to version your startup/shutdown proceedureprocedure just like you version the rest of your application.

I've heard descriptions and seen blog entries about many other large companies build systems, but to be honest, nothing even comes close to the amazing technology Amazon has produced. I would probably argue that what Google, Facebook, and most other companies of comparable size and larger do is at best objectively less good and at worst wasting millions of dollars of lost productivity. Say what you will about Amazon's frugality, their turnover, and their perks, but the tools available at Amazon make it a world-class place to build software. I hope to bring a similar environment to my current company some time soon =)

terabyte commented on Dec 6, 2017 Author full URL was: https://stackoverflow.com/questions/3380795/what-does-amazon-use-for-its-build-and-release-system/12597919#12597919

Sorry for some lack of formatting, I copy/pasted the diff from the answer history and missed cleaning up some things.

terabyte commented on Dec 6, 2017 Author

My "new thing" I mention has *actually happened*, and can be viewed here: https://www.qbtbuildtool.com

avesus commented on Sep 8, 2019

"Data driven" is considered harmful in the world of solved security issues.

ScatteredRay commented on Jun 10, 2020

@terabyte So would you say a VersionSet is a moving tag? such as a branch tag 'master' in git can track a branch, whereas the SHA itself distinctly refers to a specific version?

So, a VersionSet is a name, and refers to the most recent versions of all packages (successfully?) built against that VersionSet? Does brazil have nomenclature for the immutable thing that represents the newest versions built against that VersionSet?

hohle commented on Jun 27, 2020

Version Sets have "event" ids, which are named versions for that set and a pointer back to the parent (much like a git commit). When you build, by default the latest event is chosen to build against (if you're building locally, the "latest" is periodically updated) or you can choose to build from a different event (for example, if you need to patch a release).

The version set itself will have a name (e.g. "MyTeamsGreatService" as well as a link back to where the VS was created from "live@1593214096"). As already hinted at, events in the version set are delimited by an `@`.

When you create your version set, you supply one more more "root" packages that act as anchors for all of the other packages. As dependencies get discared, they are removed from the version set event.

You can also merge version sets. live typically has a lot of shared packages used by lots of teams (JVM, JDK, shared config, third party libraries, etc.). Instead of picking and choosing specific versions, you can get the latest that are built into live by importing which will pull all of the new package versions in and rebuild anything that depended on them and create a new version set event for your version set (this is such a common operation that it's possible to schedule it on a regular cadence to avoid stale dependencies). You can import from version sets other than live though, which is useful when sharing internal frameworks across close teams that are not intended to be published outside of that group of teams. That makes libraries shared within a multi-team group easy to build once and push out to all consumers.

Hinted at above, but not detailed is the separation of interface version and build version. When you take a dependency on another package in Brazil, you specify its name *and* interface version. You never take a dependency on a specific build version (in fact, you can't). The package config provides the graph of packages and interface versions necessary to build, but the version set is what resolves those interface versions to specific build versions.

If you create a package, say the IDL for your service API or config for clients to connect, you specify the interface version (say, "1.0"). Every time you build, the generated package is given a concrete build version ("1.0.1593214096"). That build version is what's stored in the version set for each package node. Developers never manage the build version either as an author or consumer.

There's an implicit contract, then, that within an interface version all changes are backwards compatible. Internal tooling (service definitions, config, etc.) has all been built with this in mind. APIs, object fields, enumeration values, etc. can all be added without concern about affecting consumers (at build or runtime), but if APIs need to be modified in some incompatible way, the interface version is updated (maybe "1.1") and no existing consumers are affected, only consumers that update their dependency to the new interface version. Regardless, when you build a package all consumers are rebuilt to ensure build-time consistency.

I left Amazon about a year before the OP was written, but the conclusion is spot on. Once you understand the build and deployment tools you first wonder how you ever did anything before and then start to fear how you'll do anything once you leave. One major issue, in my experience, is the Brazil packaging concept doesn't have critical mass or would require boiling the ocean to get there. At Amazon, there's one package manager - Brazil. You want a Gem, NPM package, *.so, or JAR dependency? You add it to Brazil. Sometimes you go through great pain to add it to Brazil (especially when importing a new build system). But once the package is there, everything just works. If you look at something like BSD ports, you get the comprehensive set of packages from varying package managers merged into one, but you don't get version sets. MacPorts can get pretty close since you can point to a specific SHA of package definitions along with fuzzy dependency definitions, but there's one primary set. Other package managers with fuzzy versions can't produce repeatable builds. In the Java space most dependency managers use strict versioning, which means its extraordinarily difficult to regularly pick up small minor version or patch version changes.

Going into a bit more detail, Brazil gets dependency types right as well as relatively straightforward version conflict resolution. As OP said, Brazil is a thin set of bootstrap tools which forward control to a "build system" as soon as possible. A build system is just another package in Brazil with an executable in a consistent location, so it's versioned along with the rest of your version set. Brazil itself isn't versioned, but it is really so small that in practice it doesn't matter. Build tool dependencies are scoped to work with each other but are not added to the library path during compile by default. Compile-time dependencies are exactly that. Runtime dependencies are packages that will be deployed, but not added to the library path during compile. Both build and runtime dependencies ultimately get deployed. Test dependencies are just that, a separate test set of dependencies which also include compile dependencies. Dependencies are transitive.

Because `brazil-build` is primarily a package manager and defers to other build systems for generating the bits, package configuration is super simple. Primarily dependencies and maybe some additional metadata describing the produced artifacts that will be added to any resolved library path. Unlike something like Maven where dependency management and build logic are intermingled, language native build tools are then used. As OP said, make or cmake for C/C++, rake for Ruby, ant for Java. By the time these tools are used and dependency paths are already added, so LD_LIBRARY_PATH or class paths don't need to be mucked with by developers. It also means that if you don't like your build tool, you can switch it out and none of your other dependencies need to change.

When version sets are imported into Apollo, all of the test and build dependencies are stripped away and the remaining graph is passed in.

ScatteredRay commented on Jun 27, 2020

Awesome, thanks for this detail!

ScatteredRay commented on Jun 27, 2020

@hohle Just to clarify:

> MyTeamsGreatService" as well as a link back to where the VS was created from "live@1593214096

live in this case is the name of another version set that this one was say... branched off of? And say, MyTeamsGreatService will have a whole set of links MyTeamsGreatService@16983628 -> MyTeamsGreatService@16983627 all the way back to some other VersionSet, e.g live@1593214096

Is that an accurate understanding?

hohle commented on Jun 29, 2020

That sounds about right. A lot like a git "commit" object, without being content addressable. `live` was a somewhat special cased version set, since it was "public" within the company. IIRC there weren't any "root" package in live, since you didn't really deploy from it.

OP wrote:

> Builds are serialized on version set

This isn't completely true. Parallel builds could happen within a version set as long as their packages didn't overlap. If a version set was used for a single service it was likely that nearly all builds would be serialized since they all ultimately impacted a root package. In live, however, there were many smaller paths which didn't impact one another.

For example, I maintained the internal `tmux(1)` package. It had a small number of consumers for which hit was a runtime dependency (primarily a packages which was a collection of nice developer tools, and a few other packages to make sure you had previous versions when the tux protocol changed and you upgraded while a session was still running). If I rebuilt tmux, only that small set of 3 or 4 packages was affected so any other builds against live that didn't affect those packages could run in parallel. A widely shared package, like Guava which had a lot of consumers (including transitive consumers) would require rebuilding a lot, which would then block anyone else who wanted to build a package in that set.

Later on "dry run" builds were introduced so that if you did decide to build something that would affect a lot of other things, you could perform a non-blocking build where all of the artifacts would just be sent to `/dev/null` but at least you would know that everything compiled and unit tests passed before potentially blocking others for some significant period.

Another great feature was that Brazil was tightly integrated with the VCS being used (Perforce, Subversion, or Git). All artifacts would included links back to the commit (in perforce there were a few large depots broken up into individual packages, in git every package was its own repo). When viewing a deployed package there was always a link back to the raw artifacts and the commit the changes were built from.

OP also mentioned that deployments were namespaced (not dissimilar to containers, but prior to CNAMEs). Workspaces were namespaced as well. A workspace consisted of a version set to track and any packages that were checked out. When working locally, any checked out packages would override the versions in the version set and everything else would be resolved from the previously built version. You could have as many of these namespaces as you wanted, they were "private" to your computer. It wasn't uncommon for people to have a large workspace for all of their team's projects, smaller workspaces for individual projects, experimental workspaces for trying things out without breaking a longer lived workspace. Since these were all namespaced, and *everything* is modeled in the VS, you never had to worry about not having the right RPM installed or working on two projects with conflicting dependencies. Each lived in its own namespace free to evolve independently.

Apollo would also allow you to "override" packages individually so you could deploy an application to your workstation and override an application, library, some config, etc. without needing to touch the rest of the packages in the version set. This was convenient for development, but was not "repeatable" in a strict sense, but pretty nice. When I worked on a monolithic C++ service, for example, I could check out the repo for my team's SO, build it, override it and restart the service to see my changes locally.

It's been many years, so I'm sure I don't have all of the details perfect.

terabyte commented on Jun 29, 2020 • edited Author

@hohle wrote:

> This isn't completely true. Parallel builds could happen within a version set as long as their packages didn't overlap. If a version set was used for a single service it was likely that nearly all builds would be serialized since they all ultimately impacted a root package. In live, however, there were many smaller paths which didn't impact one another.

You are right, of course, I forgot this detail. Thanks very much for expanding this answer and filling in a lot of blanks! Thanks also for confirming so much of what I said and making it feel just a little less like I am some lone crazy person screaming into the void =)

> When working locally, any checked out packages would override the versions in the version set and everything else would be resolved from the previously built version. You could have as many of these namespaces as you wanted, they were "private" to your computer. It wasn't uncommon for people to have a large workspace for all of their team's projects, smaller workspaces for individual projects, experimental workspaces for trying things out without breaking a longer lived workspace. Since these were all namespaced, and everything is modeled in the VS, you never had to worry about not having the right RPM installed or working on two projects with conflicting dependencies. Each lived in its own namespace free to evolve independently.

THIS is one of the most important aspects for local development and you did an excellent job of describing it. The build tool I mentioned above, QBT, does this very similarly, and it is the most important feature IMO, from a lone developer's perspective.

terabyte commented on Jun 29, 2020 Author

CONTEXT: I was on the Amazon build team from 2006-2009, wrote this answer in 2012, QBT was extracted from Palantir and open sourced by **@amling** in October of 2015, and I created this gist in 2017.

QBT is basically unused by anyone for anything except for core developers **@amling** and myself, for personal projects, but it is extremely high quality and stable and could be used in an enterprise environment by anyone with sufficient resources and motivation. Using it from the ground up would be fairly easy, but moving a significant codebase to it would be extremely challenging (just like moving a codebase to Brazil and Apollo, as **@hohle** put it, "would require boiling the ocean to get there").

quettabit commented on Dec 31, 2020 • edited

@terabyte is that why blaze was permuted as bazel by google to sound very similar to brazil 😄 ?

brimworks commented on Mar 2, 2021

Hi Carl! Nice write-up on Brazil! I hope you are doing well.

terabyte commented on Mar 3, 2021 Author

> **@terabyte** is that why blaze was permuted as bazel by google to sound very similar to brazil ?

Probably a coincidence...but who knows? =)

terabyte commented on Mar 3, 2021 Author

> Hi Carl! Nice write-up on Brazil! I hope you are doing well.

Hey man, thanks! I am doing well, and I hope you are too! Drop me an email some time if you want to chat or catch up!

adrianosela commented on Jun 21, 2021

This should all be re-written in a much more ranty way imo... 😂

uri-canva commented on Jul 11, 2021

This all sounds very similar to https://nixos.org/ and https://guix.gnu.org/ (when used as package managers, not as distros).

olebedev commented on Nov 16, 2022

@uri-canva, yeah, that is pretty much Nix's concept of how to manage dependencies of packages.

There is a missing bit in Nix - the deployment system since Nix is "pure". However, after experimenting with Nix I made it work as a deployment system using "side-effects" that I created on top of fixed-output derivation.

But yeah, that amazing to see the idea is getting converged over years into a solid proven-by-many-use-cases and implementations approach.

s-debnath commented on Oct 10, 2023

As someone who has worked in the industry before joining Amazon, Brazil always such an amazement to me. To my coworkers who have only worked at Amazon I can never get through how great it is compared to what's out there. Definitely going to be one of the top things I miss when I leave.

DavidSouther commented on Nov 16, 2023

> Once you understand the build and deployment tools you first wonder how you ever did anything before and then start to fear how you'll do anything once you leave.

You could almost find-replace “Brazil” for “Blaze/Bazel”, “Apollo” with “x20", “smithy” with “chubby/grpc”, and “Amazon” with “Google”, and this post would be exactly correct. The biggest difference between Amazon and Google that took me getting used to (started from Google) was Brazil's source-first philosophy, directly opposite Brazil's package-first philosophy. I don't know enough about BuildXL, but Microsoft has clearly "solved" this as well.

I get that this post was originally written the better part of a decade ago To @terabyte's statement that these are "inevitable" conclusions of a large engineering organization, that is 100% correct then as it is now. And today, a decade later, we're finally getting Bazel, Nix, Docker, and even the language-specific tools (NPM, pip, Gem, as long as you stay within their languages) to provide Reproducibility and Consistency, while coordinating with git to provide Change Management. Yay!

sureshadapa18 commented on Apr 30, 2024

Good Information lot of new words/buzz words. The ultimate aim is to have better build system with re-producibility (from snapshot) and consistency "when your peer / customer trying to run/ execute".
We at Collabnet **@2005** used to call the build system as "ducati-build system" where everything is built from the source with custom bindings and custom patches. It was a unique and great experience.
Consistency: Hopefully most of them (build&release enginners) understand this. It is very loosely used these days in many DevOps projects. The depth is very important. Like m32/m64, Intel/AMD, Kernel dependencies, Environment Packages, Software bindings, Versions, Dependency*....etc .
Profiling: IaaS, PaaS and SaaS.