Let’s say you have a massive C/C++ project that started out with make files and shell scripts and grew more and more. At some point you realised that building the project takes ages.

At this point someonewill come out and say "we used this thing called blaze at Google, and it was super fast, so let’s rewrite everything to use bazel".

Cool! No objections here, go and do that.

But know that things are probably not going to be as easy as that one small library you converted to bazel would have you believe.

As mentioned earlier, distributing a build has a hidden cost, the shuffle stage in a distributed systems has to pay to cost of transfering artifacts from one machine to another byte by byte.

So just by the virtue of distributing your build, you’re adding an overhead to your build.

Let’s talk about a concrete example.

If you have **hello.c** which compiles to **hello.o** and the only thing **hello.o** needs to link with is **libc** then you are probably better off using the same task context for both compiling and linking.

In simpler terms if your intermediate file has one consumer, then you are better off using the same task context and skipping the new sandbox creation and copying the bits in to that sandbox.

Only whwen you have 2 concurrent upstreaam consumers you’ll want to create a new sandbox and copy the results.

This is obviously from a pure performance perspective, sandboxing actions indiivdually is a good idea even when the over head is taken in to account as it increses observability and debugability.

So if the performance is the only concern, then bazel might not be the best option, maybe dedicated build machines with more memory and more processing will do just fine.

IE, if you have a huge project and the linking stage takes a long time, bazel isn’t going to help bring down that number, but a bigger machine will.

All of this to say, Bazel is usually a good bet, but not always.

So what must one do?

Measure the performance of your build system. See what’s taking a long time.

If you are repeating a lot of work, then you might want to use a distributed build system.

But first and foremost, you need to understand where the bottlenecks are.

Dealing with bottlenecks

"Progressive enhancement" was coined by Steven Champeon & Nick Finck at the SXSW Interactive conference on March 11, 2003in Austin.

Imagine you're working for a massive airline and you are tasked with modernizing the booking system.

The bit of the page that searches for flights was built by a team of contractors for the airline group 3 years ago, but that team has been disbanded after the work was complete.

The bit of code that does the seat selection is securely stored in a CVS instance so that piece of code is lost completely to the sands of time.

The teams themselves were all formed on a contract basis to do specific feature work on the page and were disbanded once the work was completed.

All the information on how to build the projects and even some of the tools that have been used to build them are obsolete.

And since rebuilding the entire page from scratch is not an option one would need to get creative.

This was a very common occurance as we were transitioning off of Flash and on to a more modern HTML5 systems.

Progressive enhancement came out of people trying to make pages more usuable without throwing out the old code.

Progressive enhancement is a design philosophy that provides a baseline of essential content and functionality to as many users as possible, while delivering the best possible experience only to users of the most modern browsers that can run all the required code.

source

The progressively distributed apps are a way of taking a large application and incrementally breaking down it's funtionality where bottle necks are identified.

In simpler terms, if you start with rules_foreign_ccand work your way up to rules_ccas needed, you'll be progressively distributing the application.