# The Unified Build Almanac (TUBA) - History

There have been at least 5 major iterations in how we build and ship:

### Product Construction v0 (.NET Core 1.0 to .NET Core 2.0)

Mists of time…

.NET was built using many inter-related repositories. A rudimentary dependency flow system (Maestro Classic) was used to update the relationships between these repositories. When a repository built officially, it uploaded information about the produced packages to the versions repository. Changes to this repository were monitored by Maestro Classic, which then ran specific Azure DevOps pipelines based on those changed files. Those pipelines then created PRs to consumers of those packages. A build was complete once the set of desired changes to all repositories had propagated along all the dependency flow paths.

This system worked, in a way, but had serious drawbacks. It was very difficult to determine the state of the build. How much had propagated? What was left to do? The dependency flow system lacked any sort of reasonable UX. Historical data was very difficult to find and sift through. The dependency flow mechanics essentially lived within the build logic of the repository and could be broken by changes to the repo. It was very difficult to produce builds with security fixes. All of this led to builds that were slow to produce and prone to errors.

.NET decided to move to a different approach that could produce builds faster and more reliably. Enter Product Construction v1 (informally, this was referred to as “ProdCon”).

### Product Construction v1 (.NET Core 2.1 to .NET Core 2.2)

Product Construction v1 was an attempt to lower the cost of builds and produce them faster. ProdCon, as it was informally called, produced builds of .NET by sequentially launching the official builds of the “core repositories” of .NET and feeding the outputs of each one into the next. For example, coreclr would build, then ProdCon would feed the packages produced into the corefx build, and so on. The key difference between ProdCon and Product Construction v0’s approach is that ProdCon orchestrated the build from the top down, rather than letting repositories orchestrate it from the bottom up. Dependency flow was done without commits to each repo. The orchestration system simply overrode the input package versions of dependent repositories with the outputs of previous legs, and then told the build where to put the outputs.

Again, this system worked, but had serious drawbacks. On one hand, it was easy to determine what the state of a build was. It was easy to deal with security fixes. It produced builds relatively quickly compared to previous systems. On the other hand, it had no way to deal with breaking changes, making it prohibitive for use in early product development phases. Because dependency flow was not implemented with repository commits, building a shipping commit for a repository wouldn't produce the same build that ProdCon created. Reproducing official builds was expensive and required specialized knowledge.

Again, .NET decided to try a different approach (**Product Construction v2**).

## Source-Build (.NET Core 2.1 - Today)

In *parallel* to Product Construction v1, .NET began a separate effort to enable Linux distributions to natively include .NET, beginning with RedHat. There are key differences between how Microsoft traditionally builds its products and how Redhat and other organizations build what they include in their distributions. Linux distributions usually require that all software be built from source alone, using other software that has been built from source, on a single machine, disconnected from the internet. No other inputs. This is fundamentally at odds with how .NET's product is constructed; as a set of independent repositories interconnected by binary dependency flow transport mechanisms, with heavy reliance on orchestration and multiple machines even within a single repository build. There are many places within the product build that join together artifacts built on multiple machines.

To enable RedHat and other organizations to build .NET, we began the "Source-Build" effort. Source-Build re-architects the build to produce artifacts required for Linux distributions (typically the .NET SDK) from a single source layout. Each repository in .NET and its external dependencies (e.g. Newtonsoft.Json) are cloned into a flat source layout, along with associated tooling and scripts. The repositories are built and their outputs are fed (on disk) to the next repository. The build logic is tweaked to exclude package references, build targets, projects, etc. that do not contribute to the required artifacts, or would bring in additional additional dependencies that can't be source built.

While source-build does work, it takes significant resources to maintain. There are significant differences between the source-build methodology and Microsoft's .NET build methodology, and these tend to contribute to build breaks.

## Product Construction v2 (.NET Core 3.0 - Today)

Because the drawbacks of Product Construction v1 made it unsuitable for day-to-day development and because .NET Core 3.0 was a significantly more complex product than 2.1, .NET decided to move back towards the Product Construction v0 methodology for .NET 3.0 and onward. Essentially, orchestrate the .NET build from the bottom up (repo-repo relationships) instead of the top down. To reduce cost, we decided to rewrite the dependency flow system to enable better UX and flow control and added shared tooling via Arcade to enable consistency across repositories. You can read more about these changes in: The Evolving Infrastructure of .NET Core and A Deep Dive into How .NET Builds and Ships.

For most cases, Product Construction v2 is a significant improvement over either previous Product Construction systems. The dependency flow system (Maestro++), in particular, is much more flexible, self-service, and easier to understand than previous iterations. Unfortunately, the ease of dependency flow has actually led to a *more* complex build. Because more repositories could participate and introduce new relationships, they did. This strains .NET's ability to build and ship. A single commit in a repository at the base of the dependency flow graph might lead to *dozens* of eventual commits in downstream repositories. Cross-stack changes are still expensive to make. Maintaining source-build in the face of this complexity is harder than ever. Furthermore, the infrastructure required to produce a *full* .NET build (as opposed to a source-built SDK) really only exists within Microsoft's labs, and even Microsoft's own .NET developers struggle to work efficiently. This is a hindrance to the .NET community.

Again, we need a new methodology.

### Unified Build (Product Construction v3) (.NET 8/9+)

Unified Build for .NET is a combined effort to improve two problematic areas for .NET:

- **True upstream model** – Partners are efficient and can do the same things we can.
- **Product build and servicing complexity** – Taming our incredibly complex infrastructure.

Unified Build will eliminate Microsoft's current method of building the full .NET product, replacing it with source-build across all platforms. This will result in only needing to maintain one build system, shared with the community. The product will be built as an aggregate codebase (a monolithic repository), although individual repositories will still be used for day-to-day development. The goal is simplicity and consistency. We believe that this will not only make .NET easier to develop, build, and ship, but it will also enable .NET contributors to be more productive.