# Protocol Buffer Tooling and Definitions

Although Protocol Buffers are, in many ways, highly standardized, there's a wide range of tooling available to work with them. In general, we've been operating in both the "vanilla" Protocol Buffer world and the "gogo/protobuf" world--two related-but-distinct projects, which use a distinct-but-overlapping set of terms and concepts.

In order to facilitate future discussions and planning about our Protocol Buffer tooling, I wanted to take the time to write down the distinctions between these two projects, and to define the terms they use.

## Glossary

**"Vanilla" Protocol Buffers** refers to the tooling created and maintained by Google. In this universe, we have a few important terms:

- **protoc** is the Protocol Buffer compiler that is maintained by Google. It turns .proto definitions into generated source code.
- **Plugins** are programs which can read "request" protocol buffers from stdin and then write "response" protocol buffers to stdout. This is the approach that Google recommends for third-party code generation. See the official documentation for more.
- **protoc-gen-go** is a plugin for protoc that lets protoc generate Go code. It is also maintained by Google. See the Godoc for more.
- **Extensions** are *fields* in messages defined in .proto files, and are the way that these messages are able to extend one another. For example, if I define a message called *foo* with an extension field, other people can extend *foo* with their own field types. See the official documentation for more.

But in the **gogo/protobuf** world, things are a little bit different. https://github.com/gogo/protobuf is a fork of "vanilla" Protocol Buffers that generates a compatible serialization format but offers more configurable options for code generation in Go. In this world, the important terms are:

- **protoc-gen-gogo** is another Protocol Buffer compiler plugin, which can be used with protoc to enable the extensions included in the gogo/protobuf project. See the Godoc for more.
- **Extensions** are *annotations* in the proto fields which are used by the compiler to inform the generated code. For example, Regen's Cosmos Proto repo contains definitions for extensions which protoc can use to generate Go code that contains Go interfaces. *This is a different use of the word "extension" from above.*

In other words: We always use **protoc** as our *compiler*, but we have a choice of *plugin* (**protoc-gen-go** or **protoc-gen-gogo**). If using **protoc-gen-gogo**, we can also choose from a range of *extensions* which impact the generated code.

## Current State

The Cosmos SDK uses the **protoc** compiler with the **protoc-gen-gogo** plugin and a variety of extensions, including those from Regen's Cosmos Proto repo. The Go code generated this way has several advantages, including being a more natural replacement for the code that was used with Amino.

Meanwhile, Tendermint Core would like to use the **protoc** compiler with the vanilla **protoc-gen-go** plugin.

In theory, it should be OK for different applications to use different proto plugins. As the only difference is that the generated Go code is different, it's fine to use different plugins, or even different compilers, across different projects if those projects are only communicated through serialized data. **In other words, this is fine as long as the generated types are only used internally.** Unfortunately, Tendermint currently exposes generated types in its public Go APIs, which means that we have to use exactly the same generated types as our applications, which means that we must use the same proto compilers, plugins and extensions.

So in order to maintain compatibility with the SDK, we are using gogoproto with the same plugins--for now. We will ultimately switch to the vanilla proto compiler, although this will require us to rewrite our public Go API to use "domain" types rather than generated types.