Beautiful Code is Good Code: A Developer's Guide to Co-dfns

Aaron W. Hsu

October 24, 2025

Abstract

This is the best that we can do so far in providing a "Developer's Guide" to Co-dfns. It turns out to be a surprisingly difficult problem, but we hope that this document will provide an anchor point for getting started with the project and understanding why it may feel strange and different to work in the Co-dfns code base as a developer, as well as what you can do to get started and begin demystifying the Co-dfns development method. This is not a document that will have any import for the user of the Co-dfns compiler, and it is strictly intended to provide clarity for those who are interested in understanding how to get started working as a developer on the Co-dfns compiler. We divide this into a set of principles and a set of rituals. The principles, by far, are more important than the rituals, and as such, if you read anything, read the principles, which will make the rituals obvious.

1 Introduction

Even among APL projects, Co-dfns is a little strange. For a long time, people have shyed away from exploring the compiler as a developer (as opposed to an user) of the system. We have had calls for more documentation, or various other systems, to provide for some sort of familiar foothold into the project. Each time we have tried to explore these options, they have felt insincere and highly dubious. They almost seem to do more harm than good to the project overall. We have found it exceptionally difficult to provide any sort of written document that provides clarity as to the "way to do Co-dfns development" and how to get started on the project. Despite this, we have had a number of successful people work with the Co-dfns codebase, and generally learn how to work with the compiler as a developer of the system. Still, we have always felt that it would be nice if there was something that could help situate a new developer into the Co-dfns "way of working."

Until recently, we believe that such a guide would have been impossible: we ourselves just didn't have the words to properly describe what we are doing in a way that would allow the new developer to get on the right track without introducing very bad habits that would have to be rectified at great cost. However, we have recently had a couple of people working very deeply in the compiler for an extended period which has allowed us to make some important external observations as to the nature of our work and our collaboration. I have also personally learned of some frameworks that have allowed me to articulate the way that I tend to frame the work that we do in a way that I can actually talk about it authentically. This guide is the result of these experiences and our efforts.

Unfortunately, in having written this guide, it just becomes more clear that in many ways, Co-dfns is strange even down to the core of its method of doing business. A part of me hesitates to release this guide because even reading it myself, I can't help but feel that the overall picture it paints comes off as pretentious and aloof, almost elitist. While I sincerely believe that this couldn't be further from the truth, I also have to acknowledge that it certainly feels pretentious on the front of it. My only defense is that, when I look and discuss the work that I do with my colleagues who work on the compiler as well, the reality is that *we really do work this way.* This isn't some pretentious attempt at designing some guide that avoids the concrete nature of our work; this really is an authentic and direct expression of the way that we do work in the compiler, and it is the method which we have found to be the most effective in actually shipping working code that does what we intend it to do.

While the principles that we use to get our work done might seem strange to some, we have tried many other methods of development and documentation, but each failed to actually achieve the results that we wanted. These principles did work for us, though, and they are consistently the things that have worked each time we have used them, and consistently are the signs that something won't work when we stray from them. I wish that they were more concrete (I really, really do!), but I haven't found anything more concrete that accurately represents what we do day-to-day. In our day-to-day work, this is how we talk about the work. We don't have some style guide that we reference, or some set of rules that we appeal to, or any sort of development process outside of these principles.

Thus, despite the strangeness of this document, I hope that the uninitiated reader will accept our humble offering in our sincerest hope that they will find it useful for them to understand how work is done on the Co-dfns project and how they can join in this process with us. We welcome you!

2 The Principles

These principles are not exhaustive of our way of doing work, but they represent the clearest model we have right now for understanding the nature of how we develop the Co-dfns compiler and what has "worked" for us. The framing of our development method as principles is not directly influenced by the Agile programming methods, since they tend to place a stronger emphasis on the concrete results of these principles in the forms of rituals as a means to define the specific way in which work takes place. For us, though, these principles are the direct frame and lens through which work is done. They aren't abstract derivations of the work that we do, but represent the actual working view and practice "on the ground" of how we do work on the compiler, how we collaborate, and how we evaluate our work.

2.1 The Code is the Specification

This is perhaps one of the most important foundational aspects of the way that we work that differentiates us strongly from other projects, and may be one of the hardest things to understand or internalize coming into the project when working on the code, having come from other programming projects or cultures. It's also easy to misunderstand. We do *not* mean by this that the code is "self-documenting" or that the code is a "reference implementation" of intended behavior. Nor do we mean that the code is "infallible" (more on that later in the latter sections). We also do not mean that the *tests* are the specification, as you may sometimes hear when working on some Agile projects. We also don't mean something like "working software over comprehensive documentation" (citation needed).

To explain this, let's point out the primary implication of the above statements. The first implication is that there is some other artifact written down somewhere that defines what "correct" means in the case of this code, and the code itself is to be evaluated relative to this other artifact. That could be an user story, a requirements document, a test case, or any other sort of formal artifact. This is the usual way to think about code. Code isn't usually the specification; it is usually the implementation of the supposed specification. Self-documenting code may lack documentation, but it usually still presumes an external specification of some sort. A reference implementation may sometimes be used as a reference for correct behavior, but it isn't usually the specification, but rather, a black-box oracle from which correct behaviors can be "tested" or, as is more often the case, where behaviors may be cross-checked against another written specification. Test cases or a test suite are often referenced as a specification of what the code should do, and this often goes hand-in-hand with test first development of various forms, including test-driven development. But this still defines the correctness of the code relative to another artifact which is developed independently, though often concurrently or iteratively with the code itself.

Moreover, we do not say that the code is infallible, despite it being a specification. This is because the code represents only a part of the entire understanding that exists, and it also represents only a present and historical snapshot of understanding of reality. As such, it is a specification, but a specification that is presumed to only represent present knowledge, which may change or grow in the future, sometimes quite drastically. We discuss more about this in the following sections.

Another way of saying that the code is the specification is to understand that a specification is usually the thing that people look to in order to understand what to expect the system to do and to understand what the desired behavior of the system is. For us, we are saying that we write the code such that we intend it to be the primary means of communicating this intent and desired behavior, not some other document.

What then do we mean? By saying that the code is the specification, we are saying that the code represents the highest artifact by which we express our understanding of the problem domain and what the solution to the problem domain should be. It is the statement of what we think of as the correct behavior of the system. In a sense, it represents the highest written authority in the project. This implies some remarkable effects that may be quite unintuitive or surprising to those who are not used to working with code as a specification.

2.1.1 No Developer Documentation

Perhaps the first obvious thing is that there is little to no developer documentation in the project. Having developer documentation, which we have tried in the past, actively undermined this principle. Put another way, we write worse specifications when documentation exists than if it did not. Why is this? Because, without documentation, there is nothing that we can do to make the code easier to understand other than literally making the code easier to understand. If we start to misunderstand a part of the code and are tempted to write developer documentation to help us remember some aspect of it, then the solution to that is not to write documentation, but rather, to write code that is easier to remember and perceive.

What we do allow for is comments which help to communicate intent and some relatively clear meaning within the code itself. We do so sparingly, and with what we hope is reasonable care, but it does help to introduce into the code base certain vocabulary that would otherwise not show up in the code, and this in turn helps us to make sure that we are more coherent in our talking and discussing the code, but the main reason that we do this is that comments sometimes help to serve in place of what other codebases would use function abstraction for, and we find that a comment is a much better way to introduce such information than a mechanical function abstraction.

Likewise, if the architecture of the system isn't clear by reading the entry points of the compiler, then we have done something wrong in the code, instead of having missing documentation. The shape of the code should clearly communicate the actual behavior of the system, becuase the *code is the specification.*

2.1.2 You Read the Code First

The code is written to be the primary point of discussion, understanding, and learning. It is *the* core tool in this project for discussing behavior and results. As such, the code is not the last thing that one reads when trying to understand something, it is the first. In many projects, you read the developer documentation first, then you read various other things, such as the test cases and the like, and then you finally may go into the specific area of the code that you want to play with and try to make the changes you want to a piece of code. But rarely in other projects will you start by reading the code to understand how it works, at least for “good projects” that are “well designed.” There is an implicit value in the broader programming community that by the time you get to the code, you should already basically understand how things work, and that you shouldn’t expect to learn things through the code first. At best, the code serves as the final clarifying point about some specific implementation minutiae, and it shouldn’t be seen as the primary means of understanding what the code should be doing.

In our project, we take a wholly different and contrarian view. When we work, or collaborate, or talk about anything, we almost always start by pulling up a diff of the code, the code itself, or both at the same time. We generally do not develop the compiler or decide what behavior we want by doing anything other than reading and talking about the code itself. This include tests. We will rarely write out tests first as a means of discussing the behavior of the code. We work and read the code first, and then once the behavior and specification is clear, we write tests if appropriate.

2.1.3 Architecture, Boilerplate, and Coherency Really Matter

This all means that anything which makes it harder to read and understand the specification is a very bad thing for us. Therefore, we care a lot about ensuring that the codebase is coherent, predictable, consistent, architecturally clear and transparent, and that we have removed as may possible sources of confusion as possible.

One of those biggest points of confusion turns out to be any kind of excessive implementation-level abstraction or boilerplate. In either case, we often make statments about things which are not related to specification, and instead are related to something other than the problem domain. The more of this exists in the system, the more likely it is that we will confuse what the problem and specification is and what the incidental elements of the system implementation are. If you confuse these two, then it is hard to understand the code as a specification. To that end, we almost obsessively work to ensure that boilerplate and excessive implementation level abstraction is removed from the system. We introduce such elements only in so far as they are necessary, and where they do not harm the codebase as a specification, as much as we can reasonably do, and we continue this process over time.

2.1.4 The Code is Meant to be Understood Holistically

While any good specification allows you to answer specific questions about the behavior of the system, a problem that you often have when going from a specification to an implementation is that you get very bad clarity on the holistic elements of the project, such as the architecture, how data flows through the system, or what sort of overall invariants exist in which places of the project. It can also be hard to understand the impact of changes to a specification, which is why many ad hoc specification methods run afoul of being challenging to change because they can have carry on effects elsewhere (user stories can do this, as well as tests).

Instead, we strongly prefer to bias our specification in favor of making it as easy as possible to see the entire picture of how the system functions while also being able to freely "zoom in and out" at any level of detail without losing your place or losing the sense of the big picture. The more big picture we can provide, the more we like it. The easier it is to reason at a minute level while having a large picture of the whole system, the better.

2.1.5 You Cannot Prove the Code Correct

Another really surprising outcome from treating code as the specification is that you cannot prove that the code is correct. This was something that the developers struggled with for a while. We personally love proofs, and we love the assurance of correctness. However, the more we worked on the codebase and worked to improve the system, the more we realized that this was simply a definitionally impossible problem.

When one proves code correct, one does so by proving equivalence to some specification that is asserted to be correct. However, we do not have this. The code is the specification, so how do we prove that a specification is correct? You can't. By definition, the difference between an implementation and a specification is that the specification represents the "ground truth" of behavior, while an implementation is implementing some ground truth that is specified—often implicity—somewhere else.

Specifications are created by collaborative, incremental, and explorative development in the problem domain, and only the users of the system or the owners of the product can actually assert whether a specification is the correct one. Even worse, it is not always clear that a specified behavior is the one that is really intended in the broader context, even if it was the behavior that the owner thought was the right behavior. Therefore, developing a specification is always a human activity in which the correctness of said specification is an evolving thing as you work towards ensuring that the behavior described is in fact the desired one, and only the owners and users can make the call as to whether a specified behavior is the desired one or not.

2.2 Our Understanding is Built on Tradition and History

Given that our code is the specification, does that mean that all that matters is the code? No. That's because the specification itself doesn't emerge in a vacuum. It grows and changes over time, sometimes by quite a lot. The specification as it is written now, therefore, does not exist independently nor stand on its own. Rather, it exists as the tip of a long history and tradition, and in order to fully understand the code as it is now, it is necessary to appreciate its historicity and the way in which the code is situated and contextualized by past experiences and understandings.

In some sense, then, one might make the very real claim that the entire specification is in fact the current code in the source tree as well as the history of all prior iterations of that code throughout the entire history of the project.

In practice, knowledge gets refined over time and the current specification encapsulates so much of the existing history that it is not necessary nor even desirable to try to relive the entire past just to gain an understanding of the present. It is, however, important to understand just how much the past influenced the present code.

For us, this takes the form of two additional artifacts. First, you have the test suite. While the test suite represents a sort of pseudo-specification in other projects, in our project, we see it more as the historical "mind" of the project, in which we are able to compare the present sentiment of our developers against the whole history of past experiences, and to find and understand points of contension, where the current developers may have forgotten something or may not fully understand something that was momentarily forgotten from the past. This is a sort of "experience meter" that allows us to gauge how authentically and sincerely we are taking into consideration the past when we work towards the future.

Second, we have the commit logs in our source management system. If anything in our world might be called developer documentation, it is our commit logs. The commit logs are the written record and log of our thoughts and intention throughout the process of developing our specification. We use the version control system as a means of creating a historically situated record of what we are doing and thinking at any given time. Therefore, when we read the code, and a particular part feels strange to us, we look back into the commit logs that introduced this line of code and read the history of what was happening at the time, and this usually makes things abundantly clear why a line of code is the way that it is. This then gives us a clear path for moving forward into the future.

Any developer working on the project should make a strong and heavy use of these resources, as well as the various papers and talks that have been given about the project over time, as they provide the clearest context for understanding the code today as it stands in the current repository.

2.3 Knowledge is Personal

Just like the code is simultaneously a specification and also situated within the context of the entire history of the project, it is also understood within the context of the developers who work with it, and cannot strictly be divorced from those developers. We say that knowledge is personal as a way of identifying that the specification can't rightly be understood in its own right. The design, layout, intent, and the bigger picture of what the code is saying is something that is really only understood fully by the developers who work on the code. The code itself represents our best attempt to efficiently and effectively reify our understanding into an artifact that helps us communicate this shared understanding and makes this understanding executable. But attempting to put every possible idea and concept into the code just doesn't make sense. The code itself is built on a huge amount of prior experience, skill, and knowledge that the developers bring to the table and that the code implicitly depends on in order to make sense.

Just as an example, the code doesn't try to explain itself. It doesn't try to explain why we structure the code the way that we do, or why we handle things in one way and not another. Understanding the code requires background knowledge in how compilers are written, and how things like Nanopass style compilers are written. Without this knowledge, you will tend to misread or not quite understand how to think about the compiler passes, or even how to properly think about what is and is not a pass within the code. You also need to know APL. The code isn't designed to explain APL as such. One can learn APL by reading the code, but the code itself doesn't teach APL, and understanding the code means learning APL.

We have chosen to optimize our development processes for producing the end results that we want, and to facilitate clear and rapid manipulation of the code and understanding of the code by developers who are working with the code. This necessitates trade-offs in the accessibility of the code to the uninitiated, and we make that trade-off consciously. It is more important to us that the people who work with the code day in and day out are able to work with the code effectively than it is for people who may pass through to be able to feel like they understand the code without having to feel like they are trying.

Thus, in some ways, understanding the code means internalizing things in such a way that you, in some fashion, build up the implicit world that the code mirrors, but which only fully lives inside of the mind share of the developers that work on the compiler along with the code.

2.4 Development is Participatory and a Process

We do not see the code or our work as having any finality to it. The code only dimly reflects our ideal reality, and our goals and aims shift over time to expand and move towards bigger and better things. We develop the code not from a static sense of what should be, but from a continual refinement process that is incremental and evolutionary. As such, all the work that we do is seen as a continual process of clarification and refinement. We also find that much of what we see as the goals of the project literally emerge from our process of working on the project itself. The project and the compiler, in the course of development, reveals the future of the development. This is what we mean by participatory. It's impossible to separate the act of development from the process of setting goals or targets to reach, or the set of features that we want to implement. The process of development shapes our focus on what is on the horizon and what our goals for the project are at any given moment.

Our development is continually moving forward, and the process can be seen as an incremental development process that is driven by its own development, instead of a separate product manager or the like. Our experiencing using and working with users of the system is a part of this development process.

2.5 Code Must be Used to be Known

There is a general feeling among many programmers that you can read the documentation for a project, maybe use the program a little bit, and then read the code, and in this way you will "know" the code. There is a sense in which you can "observe" or "learn" as something of an outside observer looking in on an artifact, studying it from the outside as it were.

We do not think that this makes sense for Co-dfns. It is true that you need to run the code and see how it works to learn the code. However, when we say that the code must be used to be known, we mean not only running the code, but manipulating and working with the code. It is our experience that many people form opinions about the source from the outside, often in stark contrast to our observations of the code on the inside. Our experience suggests that until one actually works with the code, tries to do things with it, and has to develop the code as a developer, it is very hard to understand the code's "hidden zeitgeist" for lack of a better term. So much of the design decisions and communication mechanisms employed in the code only begin to really sink in and make sense *after* you have spent some time working with the code, at which point, these patterns, disciplines, and rituals begin to make sense.

Even more, we have at times tried to turn these "hidden learnings" of the code into something that we could externalize into some style guide or ruleset, but each time, we found that these rules rang hollow and insincere. They did nto represent what we were actually doing in the compiler, and so attempting to learn the code via these rules didn't make sense. Thus, it is our general approach that the best and most recommended way of learning the code of Co-dfns is to simply work with and use it, play with it, and generally attempt to spend time with it as a developer. Anything less than this and you will likely be missing the boat. This is especially true for truly grasping the style and patterns and why they make sense in our codebase.

2.6 Code Should Move towards Reality

We have one primary mechanism by which we think about how we should change the code: reality. Is the code in alignment with our understanding of what Co-dfns is? Put more snarkly, is Co-dfns actually Co-dfns? There is a desire that the user of Co-dfns has, and the developer of Co-dfns has, of what it should do, and how it should work. There is a sort of idealistic Co-dfns that is the perfect reality of the compiler. We assess the current state of the project compared to this ideal, and then we see how we can bring the code into further alignment, however incrementally, with this ideal vision.

We say that the code becomes more real when we do this. That is, it more closely matches the anticipated reality of the user and developer. Or, another way of saying it is that it more accurately reflects the desires and intent of the users and developers. It should become more accurate, more capable, and more generally deliver the sort of value that is possible by such a system. The code should become more able to communicate subtle issues about APL, and should more clearly explain and detail various aspects of working with and executing APL code. It should embody more knowledge and understanding of APL and of the use of APL in the wild. All of these things make the compiler "more real."

2.7 Aesthetics is the Surest Guide

Perhaps the most difficult thing in the project to communicate to the new developer of Co-dfns is how to judge good code from bad code or whether or not a particular thing that you are doing is working or not working. We have tried to make rules and style guides in the past for this work, but it has failed each time. Because there is no formal specification outside of implementation for APL as we are implementing it, there is no real way to properly understand whether or not the compiler is doing the right thing, either.

Given this, what can we use to guide our movement towards reality in the source? We have landed on the concept of aesthetics, of the "beauty" of the code. "Beauty can be truth-bearing." This doesn't just mean "pretty" code. It means a cultivated sense of elegance, simplicity, architecture, clarity, and meaning. It means the cultivated taste that identifies when code is beautiful and when it is ugly. This is not just personal opinion. It has a fundamental grounding in reality and in the working with the code on a day-to-day basis. Ugly code hides the truth and leads the reader away from the right answers. Beautiful code simply and elegantly clarifies intent, semantics, and purpose. It communicates not only the past, but the future. When you see it, you understand clearly, simply, and directly. You "get it" and the code points you right at the truth of the problem, the solution, and the way forward. It sends clear signals about bad and good architecture, it highlights and makes uncomfortable any sort of hack or obfuscation. Beautiful code makes it easier to understand the code, harder to make mistakes, and it creates a sort of visceral feeling about the rightness of code that tends to indicate when you've done right and when you're probably doing something wrong.

Unfortunately, we have not yet found a way of turning beauty and aesthetics into a set of rules that we can use or even a set of guidelines for writing good code. Like so many things in the Co-dfns project, this is something that you must develop through experience and working through the code directly. You have to reflect and observe the way that the code is, and cultivate a certain sense for things, and why they are this way, until you begin to sense the places where things can be improved, and why those improvements make the code more beautiful.

This is in extremely stark contrast to many other projects, which have a sort of direct confrontation with beauty, and they tend to care little for beauty in favor of a sort of brutish pragmatism. Does the code work? Ship it, and avoid having to change it. That's the underlying spirit of many projects. We find that such a model does not work well for us, for a few reasons. First, Co-dfns as a piece of source code, provides a kind of exemplar on how to do and approach certain kinds of problems which are hard to learn about in APL. It shows how to use APL for domains that aren't always obvious, and as such, we aren't just concerned with the executable artifact of the compiler, but also in the ability of the compiler to show people how to "do APL." Second, because we are simultaneously developing both our understanding of correctness and the reification of that correctness in code, introducing code that "works" but which obscures or otherwise reduces our ability to understand the problems that we are working with in the code itself just makes our lives very difficult. It would make it more likely that we would need to maintain an entirely separate specification for the APL language, at least tripling our workload. It would mean that we would have to work to ensure that our understanding of the code matched and was aligned with the specification. It would mean a fragmented and discordant understanding of the way that the system works, making it more difficult to make systems level and architectural level changes.

Another way of saying this is that "ugly" code might be expedient when you already have a good idea of what it means for code to "work." But we don't have that, and we use the code in order to figure out what we even mean by "works" in the first place, and in such a world, ugly code is only a detriment, never an expedient.

In short, we have found that beautiful code is the only fundamentally consistent metric, cultivated as it may be, which gets us good results. Every time we have tried something else, it has come at extremely high costs and generally been a failure. However, we have generally found that a cultivated sense of aesthetics leads to very rapid identification of where code is likely to be incorrect, misunderstood, slow, or generally mismatched with our intent and reality. There has been a very consistent correlation between the two, and this is why we tend to use this as our main metric when developing code.

3 The Rituals

If you haven't already read through the principles, don't bother reading these rituals. If you have read through the principles, then the rituals should be self-explanatory.

1. *Use the Compiler.* Working with the compiler is best achieved by simply running the compiler and using it as an user first. Get used to it, figure out how it behaves, and get a sense for the overall flow.

2. *Run the Tests.* Running the tests helps ensure that you have a system that works as intended and gives you an entry point into exploring the behavior of the system.

3. *Read the Code.* You need to explore how data flows through the system and the overall flow of the compiler. You need to start getting your feet wet in the code and seeing how things are layed out.

4. *Read the Logs.* You may find some things strange while you are reading. Learn to reference the logs in the VCS.

5. *Trace the Code.* Tracing the code, and learning how to explore the state that exists in the system at any given point is crucial to really becoming competent with the code. We greatly encourage the use of the debugger in exploring the state of the AST and the compiled object through the compiler passes.

6. *Watch the Talk/Read the Papers.* We make no attempt to keep our papers and talks updated relative to the latest changes in the compiler, but these papers and talks help to understand the spirit of the project much better and also help to provide critical context in seeing how the whole system works.

7. *Contact the Developers.* At this point, it should be obvious that Co-dfns is not a project that is intended for you to "go it alone." It is really a project built by humans, and designed to be interacted with at the human level. To that extent, reach out to the developers and begin integrating yourself into the community if you want to really learn how to work with the code. You can certainly "go it alone" if you want, and there are people who have done so, but at some point they found it necessary to reach out and get in touch, and each who did was able to get a feel for things much better afterwards.

I hope that this helps to make the overall system more clear, and I hope that you are able to at least get a sense of why Co-dfns is the way that it is.

4 The Development Cycle

Every project has a sort of development cycle that governs, implicitly or explicitly, how they tend to do work. Here's our general approach:

1. *Identify a gap.* This is a gap in reality. It represents something in which the present system is unaligned with what we imagine the ideal system should do.

2. *Refine our vision.* After identifying a gap, we tend to begin refining our ability to express this gap, using the existing codebase as a staging point, we try to identify what we would want the state of the world to be, or what it should be. This will often involve identifying the points in the code which represent clear violations of this vision. It is not essential at this point to fully reify the future vision, but merely to anchor it against the present code.

3. *Correct the spec.* We then proceed through the code and work to correct the code in any place that fails to appropriately represent what we are trying to do. We generally work on the code until we are satisfied that it says what we want it to say. We do not tend to follow a TDD style development loop.

4. *Add tests.* We examine whether our tests adequately account for the gap that we have identified. Generally, we will try to find a way of using the tests to test key invariants that we expect to be true, or to double check our understanding in areas which are unclear. We see these tests as the checks, not as the spec. They help to verify that the spec is what we thought it was, but are not themselves the spec. It is important to reify the gap that we have experienced into some kind of test so that it can always be kept in the shared history and can be checked again automatically.

5. *Unify our understand through iteration over tests and spec.* Often, our tests verify our spec rather quickly. But sometimes they reveal fundamental misunderstandings. We work in an iterative process to refine the spec and the tests to reach an alignment that matches our vision.

6. *Release.* We do not try to hold on to features. When we fix our understanding, we tend to make a release.

This cycle takes place directly on the *master* branch. We do not make use of other branches to do our work. We make no guarantees of stability on the master branch, as it is the branch that represents our latest understanding, and it is always changing. We make releases when we have reached points of alignment. Commits to the system should be coherent and should help to record our thinking throughout a given change. Preserving the historical record is most important in VCS. Commits are the node on which we can create annotations and discussions about specific work and thoughts that we have, so we leverage them for this purpose when appropriate.

Good luck, and we look forward to hearing from you!