SysGit

A Discussion on Accelerating Hardware Engineering Through Agile Practices

Treat Hardware as Code.

sysgit.io

Table of Contents

About the Author

Steve Massey is the Co-Founder and CEO of SysGit. As a software engineer at Slingshot Aerospace and mission integration engineer at SpaceX, Steve developed a passion for building tools and processes to support consensus and collaboration across teams, and between organizations.

Steve worked at SpaceX from Falcon 9 flight four to the first Heavy, serving in the only department that managed an interface with an outside organization. He supported the launch of over two dozen spacecraft to orbit. This included serving as Mission Manager for the F9-20 ORBCOMM-2 launch campaign, which successfully delivered 11 OG2 spacecraft to orbit, with the first stage returning to land.

At Slingshot Aerospace, he led the development of the Edge platform, performing sensor fusion across visible light, IR, and RF sensors to provide tactically relevant and actionable insights to the warfighter on remote platforms.

Introduction

The Aerospace industry faces significant challenges in program execution and delivery timelines. Major programs like NASA's SLS and the F-35 have experienced extended development cycles and cost overruns that have drawn scrutiny from Congress and industry observers. Projects like the Sentinel ICBM modernization continue to face oversight challenges around schedule and budget management. These patterns reflect broader systemic issues in how we approach complex systems development.

Program leadership across the industry recognizes these challenges. In response, there has been substantial investment in "digital engineering," "model based systems engineering," and "agile practices" as potential solutions. Enterprise software providers have captured significant market share by positioning their platforms as enablers of these methodologies, though implementation often falls short of transforming underlying development processes.

My co-founder Zeke and I were incredibly privileged to work at Space Exploration Technologies for almost six years among titans of industry and folks that would go on to drive the wider industrial revitalization of America. SpaceX is praised for not only its technical innovations in launch, restoring human spaceflight to American soil, and revolutionizing global telecommunications, but also its ability to craft young engineers into aggressive risk takers due to the extreme ownership, the resulting pressure-cooker environment, optimization to hardware development, and the unique approach to engineering an inherently dangerous and complex platform.

We subsequently have had the privilege of working with an incredible team to build software meeting this need, to grow it to several million in revenue at the first part of this decade, and then to deploy it in service of a major weapons system. It was this deployment in late 2023 that we learned, in detail, where our thesis in serving exclusively one engineering approach will fail outside of our bubble and what significant changes we will

have to drive via software, policy, and culture in order to actually accelerate a multi-billion dollar national investment.

America's continued technological superiority and defense of Western ideals depends on both the success of our major vehicles and weapon systems and our ability to adapt to a changing world. While SpaceX was able to brute-force their technological agility due a singular work environment, a lift-and-shift of these factors is impossible for the rest of the industry without a wholesale reconstruction, which we do not have the time or workforce for. It's clear that there's a large disconnect between the effectiveness of the legacy software vendor's products selling "agility", the practices currently implemented by the major primes and integrators, and the underlying drivers that would actually drastically improve program performance.

In this document, I summarize the differences in systems engineering practices at SpaceX vs legacy industrials and their underlying motivations, the differences between software engineering and systems engineering, and how we can use specific technologies from software engineering to support a federated industrial base while approximating SpaceX-level agility approach.

Any program or company names I reference are purely for illustrative purposes, and are the property of their respective rights holders. No relationship or endorsement of any product is intended or implied by any party.

Orientation to Today

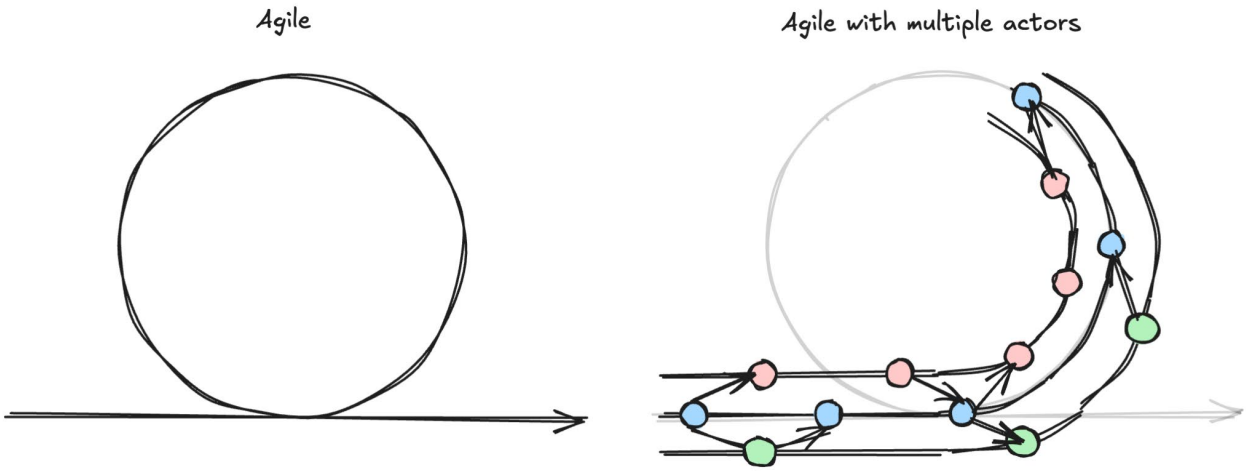
Agile Software Engineering

Agile Software Engineering works because two engineers can independently work on the same thing at the same time and gracefully resolve their differences at a later date. This is done by making an independent copy of their common baseline of work (called a “branch”), working off of that independent copy until the engineer completes the task, and then working through a formalized process of re-inserting their changes into the common baseline (called a “merge”). Cottage industries of tooling have sprouted up around this, resulting in multiple categories of software that allows engineers to collaboratively work through the formalized process (a “merge request”).

The technical specifics of implementing branching and merging are left to the reader, however as a practice we’ve gotten really good at this over the last two decades.

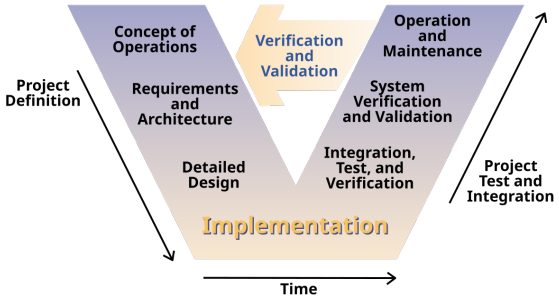
It’s not only about the process, but also the project management. Agile Software Engineering also includes a significant investment in practices like “scrum” or “kanban,” which support teams in how they plan their agile workload, with regular check-ins like a daily standup, or a biweekly product meeting. Much of this really boils down to “ask your customer frequently if they like what you’re doing,” and “reprioritize tasks based on that feedback.” Teams typically want to have some kind of a design document written out to organize thoughts at a strategic level, then implement the work at a tactical level on a bi-weekly cadence.

These frequent checks allow teams to incorporate new information and quickly pivot when required. Furthermore, the revision control processes allow teams to iterate on decisions that may or may not make it into the end product, as they can always choose to *not* merge a branch.

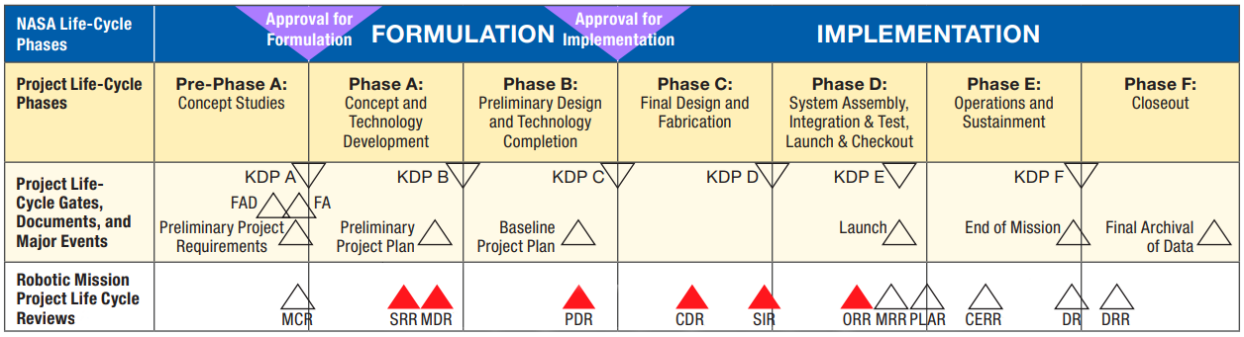


Hardware Engineering

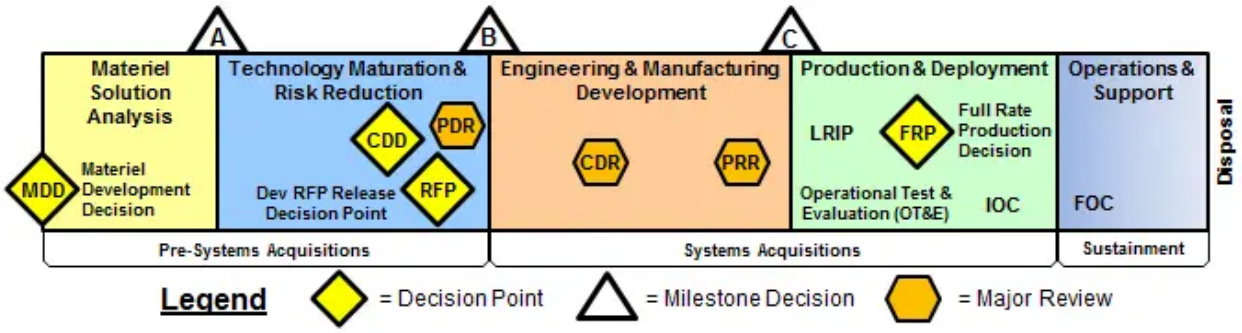
The Hardware Engineering lifecycle centers around a concept of the “Systems Engineering Vee”. At a high level, this maps the various engineering practices and artifacts required to develop a successful program, ranging from “Capture Requirements,” to “Make Hardware,” to “Verify Made Hardware Meets Requirements.”



Each of these stages has a major design review “gate” associated with it (see below for NASA and USAF documents mapping these out). You typically aren’t allowed to progress through to the next layer without passing the gated design review approvals from all parties involved, agreeing that the hardware does or will do what’s intended.

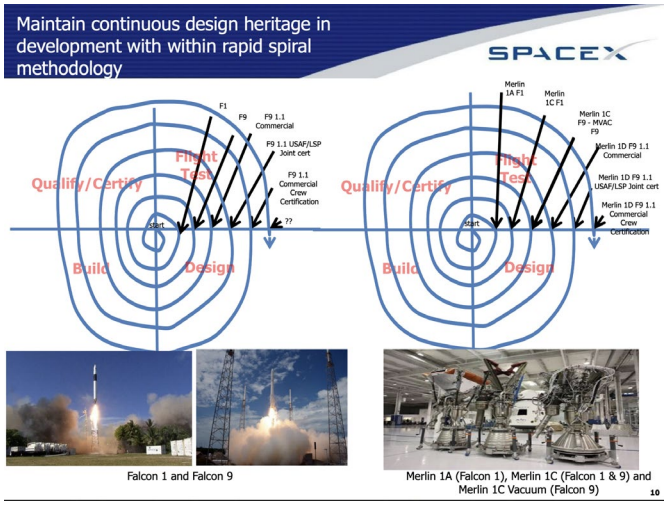


NASA design review cadence

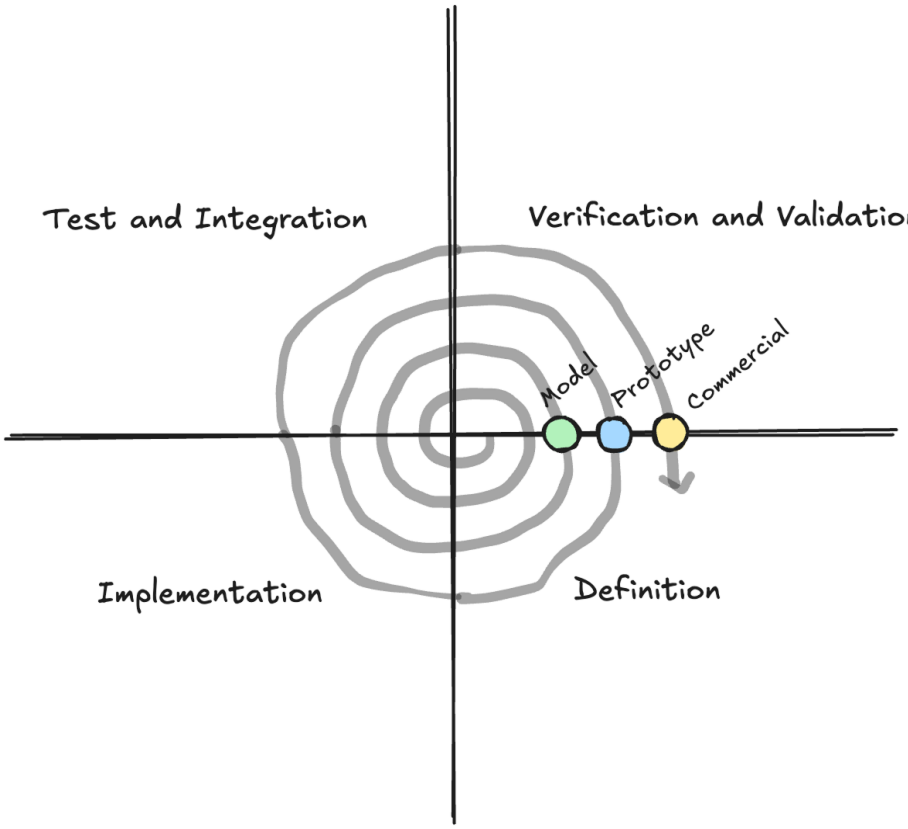


DoD design review cadence

A 2012-era SpaceX presentation is often cited as the basis of most “Agile Hardware Engineering” approaches, which introduced the “Spiral Method. The Spiral Method still acknowledges a Systems Engineering Vee but instead lightly “spirals” through the Vee. Each of the major Vee stages are loosely defined at first pass, and iteratively refined in successive loops over the Vee. SpaceX additionally adopted many of the tactical agile planning tools available at the time to track progression through the Agile Hardware Engineering process.



(Author's note: Having been at SpaceX from 2012 to 2018, I saw this firsthand. Specifically, I was in Mission Management, which needed to map the internal and organic "spiral" approaches to our customer's engineering artifacts that resulted from formal approaches. We had formal requirements and needed to somehow turn them into launches and money.)



Challenges of Agile Hardware vs Agile Software

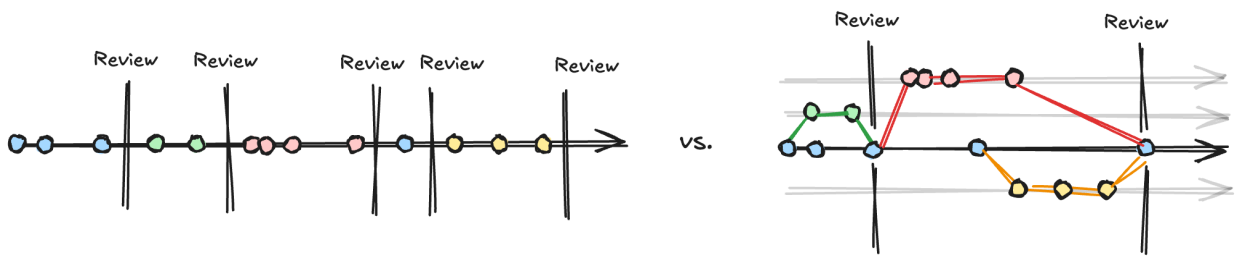
Agile Software is a bit more straightforward than Agile Hardware. All of the engineering artifacts are of the same type, and the engineers roughly use the same lingo. All software engineers are basically working with text files of computer code, and tools that act on this computer code. Hardware is a mix of different files: some binary, some text, some in databases. The engineers also spend anywhere from 4 to 10 years hyper-specializing within a discipline such that a PhD in Fluid Mechanics has no concept of which FPGA IP Cores are controlling his valves, and that MS in Electrical Engineering doesn't really need to

understand the difference between compressible and non-compressible fluid flow through the valve that her work is controlling.

A major difference from software in these files is that these artifacts are not shared within the same repository type. And while a single software effort might be spread across several repositories, there are robust ways of sharing information and context between repositories. Meanwhile, a PLM system has a very different revision control paradigm compared to software, which has a different paradigm than an ERP, which is different from a Teamwork Cloud style MBSE model repository.

Why Is This A Problem

These revision control paradigms are important to review. The dominant approach to revision controlling Requirements and System Models is, at best, closer to that of a PLM. It includes a single thread of historic changes, and a locking mechanism to prevent folks from editing their model or requirement on top of someone else's changes. These systems rely heavily on user and role-based access control to prevent changes and set up reviews. There is rarely any consideration for "parallel path" revision control, where teams are able to work on overlapping subassemblies in parallel. That is, if I'm working on a wing, you absolutely cannot work on the wing at the same time. And if you're working on the fuel tank within the wing, you'd better not make any substantial changes that impact the wing itself.

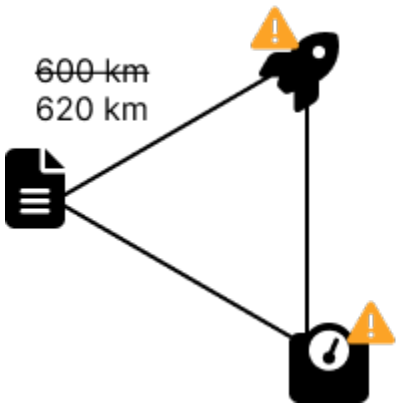


To be honest, our engineering design stack at SpaceX was built on tools that subscribed to this older paradigm. The "agile" component was actually human-to-human negotiation that routed around a tool and allowed folks to work on top of each other and resolve their differences. This worked because of the substantial autonomy granted to the Responsible Engineers who owned a subsystem, were working 80 hour weeks to drive hardware to completion, and could walk across the cubicle farm to real-time negotiate any interfaces as needed (like from our earlier example, increasing the size of the fuel tank in such a way the wing would need to also expand). This was an ecosystem that was able to tactically implement agile practices despite using non-agile tooling.

Tools that fail to understand that distinction are simply cloud-based distillations of a 15-year old workflow. *(Author note: More on my use of cloud as a pejorative in a future post.)* But I

wouldn't want to build a platform that codified the process of people routing around my underlying paradigm. That's not really innovative, or even worth venture-scale investment. It's probably a great lifestyle business, though.

Going a bit deeper into this, I've noticed a massive investment in the "digital thread" as an attempt to enable an agile engineering capability. This is effectively the ability to route engineering parameters across the different tools used by different disciplines at each stage of the Vee. Say you have a Requirement that demands a spacecraft be inserted into a 600 km circular orbit. It's valuable for that "600 km" parameter to show up in the Guidance Navigation & Control team's launch simulator. That simulator then generates a representation of the various engine states of the rocket launch, which would be valuable to show up in the Static and Dynamic Loads Analysis team's simulator (along with the spacecraft model) so they can generate an analysis of the launch loads. This is a linear process, as each discipline has their own model of the rocket to pull from (and again, each model has a different revision control paradigm). This actually approximates the Software world's CI/CD pipelines pretty well, as a change to the source of truth is then automatically propagated into subsequent services, stakeholders, scripts, and automations. However, the underlying engineering source data is rarely revision controlled in a non-linear way, compared to how it is in the software world.



Also, all of these people have to generate PDF reports, PPT slides, and (if you're lucky) technical representations, so that you can send it back to the customer so that you can prove you'll safely place their spacecraft in a 600 km circular orbit.

And meanwhile, this "digital thread" is still connecting tools together in a linear revision control paradigm. You still have to expend great effort to understand what the current system state is, and how to manage changes across different repositories if you want to

iterate in any meaningful way. And how is that negotiated? It's not automatic, since you need to engage with all of the humans along the way that maintain the actual state and its intuitive impacts within their brains and supplementary engineering artifacts. And from a business perspective, each engineering tool in that pipeline is a special snowflake of APIs and file formats, so you end up spending massive amounts of money investing in custom integrations for each variant of a CAD tool or analysis tool, only to barely recoup that investment through the users of that integration.

Okay great. In the best of cases, this approach can make a ton of sense if you have a single textbook rocket design that isn't changing at all. And it'd be great for making a dashboard to show a lot of this routing within your enterprise, so that you can generate a dashboard for review by leadership and customers. But let's talk about a few edge cases:

- What if your rocket design isn't stable? What if you have several variants of your vehicle? Do you just resign yourself to brute-force running many analyses every time the system changes?
- What if your company building this rocket design isn't vertically integrated? What if you have a major component under active development by another entity (say, your entire upper stage) and THOSE teams keep building and iterating on their vehicle. What information do you share with them? How does that impact your ability to generate these analyses in a timely manner? How can you safely make changes to your half of the rocket that might impact the upper stage?

Your tactical Agile paradigm starts to break down at these edges. And these aren't insignificant edges, these are the explicit use cases seen on every major vehicle and weapon system the DoD buys, on every major automobile, and every medical device. In fact, case 1 contributes heavily to why engineers prefer to make small iterative changes on known-good technical baselines instead of fully exploring novel design spaces (see: 737-MAX8 as an iteration of the 737 family). Recall that the only reason SpaceX started to

vertically integrate in the mid-late-aughts is because their vendors could not move fast enough. They didn't start the company expecting to make everything in-house!

(*Note: a friend of mine started a consulting business in the radiation effects space, specifically because he realized that his team of ~10 ex-SpaceXers represented the absolute top of the industry of radiation effects on electronics; however there were way more than 10 space companies. This specialization allows them to support more companies. Another friend did something similar with the hardware design space to great success, and is the secret backbone of creating early hardware components for the best venture backed hard tech companies. I wholly disagree with the simplistic venture-driven pattern-matched-to-SpaceX worldview that we should vertically integrate. Do you know what else vertically integrated? All the legacy aerospace companies. Aren't we trying to disrupt them?*)

Treatise of Change

So let's explore the counterfactual of wanting two things to be true:

- Fully explore an unstable design space and converge on something novel. Nothing is off the table, from violating engine design constraints to finding additional margin, to just slapping on more engines to your rocket.
- Assemble a launch vehicle from "best of breed" hardware, from all your friends' companies. This isn't 2005, this is 2025 and there are literally a hundred companies making hardware using the above first iteration of Agile Hardware Design for far cheaper and better than the previous two decades. Why can't you integrate components like the absolute best IMU, the best rocket engines, and the best composite products?

In the "deep tech" branded space, hardware products in case 1 are probably most effectively built using agile tactics. Scrum, kanban boards, standups, whatever, with a robust product definition document. In fact, violating the design constraints to find margin can be a reasonable use of the Spiral Method, since you're realigning your investment in pre-manufacturing Design Analysis into post-manufacturing Verification & Test. But today, "deep tech" branded companies are still working around the tools they're buying, including the venture-backed darlings that managed to capture an approximation of these agile tactics into a cloud database with linear revision control.

At the more established companies (Lockheed, Boeing, BAE, etc), this practice itself is managed by maintaining a simplified model of the entire vehicle, or "system." This is a System-Level Model, usually maintained by discrete Systems Engineers doing Systems Engineering, and more recently using Model Based Systems Engineering. These SEs preach the Vee as gospel, benefit from the design review gates as an opportunity to update the model they maintain, and spend a lot of time maintaining their own representation of what the individual engineering disciplines are doing. Because this System Model is a

useful abstraction, it's used for everything from proof-of-work at payment milestones to interface management between vendors. And if the System Model has high enough fidelity, a lot of the high-level concerns raised by the counterfactual can be resolved at the front of the Vee.

But you have to be sufficiently motivated to record and maintain this abstraction, and unfortunately most of the current System Modeling tools are arcane and frustrating to use. So the most innovative companies brute force Agile by implementing only tactics working around the existing toolsets, because it is quite literally easier and faster to find out the failure mode in hardware on the test stand than to draw out and maintain the block diagrams correctly.

Agile Hardware Engineering

Using only Agile tactics has gotten us so far. It's clear that approximations don't actually work beyond a controlled setting like SpaceX, and literally everyone else is working on even more complex and regulated programs, and not even vertically integrated. This results in Agile Hardware Engineering being more of a collection of tactics, and not a discrete paradigm shift.

So what if instead of only implementing agile tactics, we were able to adopt the underlying engineering technology pioneered by Software Engineering, and somehow apply it to the hardware product we are developing? How can we maintain parallel path revision control (branching & merging) and enable a new paradigm for hardware engineers? How can we open a "pull request" against the entire rocket with something of the scale of "add legs so it lands?" Or "add propulsive landing to the capsule because aliens wouldn't splash down in the ocean?" Or "propulsive landing is hard, please revert to a known-good earlier state?"

Well, first you need to maintain a cross-discipline abstraction of the entire rocket to even be able to corral these changes. This would be done in a System Model that is maintained somehow and kept in-sync with all the other changes being done at the company.

You would want a System Model maintained in some system that would codify the actual boots-on-the-ground reality of overlapping design work and create strong bumpers to manage these changes. Engineers would work on a common digitized technical baseline where they can collaborate on generating their own understanding of the abstracted system, author system diagrams, study impacts of changes, and share information. If a change needs to be made, they would effectively duplicate the whole baseline, perform their work, study the change impacts, and then compare these changes and impacts against the common baseline.

Consider this network diagram of people working together on an engineering platform (todo fig). Mark might be working on the engine while Susan is working on the fuel tank. The engine needs to plug into the fuel tank at some point. However Mark was told (by Julie, the first stage Responsible Engineer) he needs better fluid flow rates to increase engine performance, so he widens the entry point. At some point he has to tell Susan about this, who cares about how much fuel is flowing out of her fuel tank. So they get together, negotiate the change (Susan now needs to work with Loads to model how the fuel sloshes around differently), and build a report on the impacts.

Susan and Mark then write down what changes they made, and share that with Julie (the RE for their combined assembly, the First Stage) for review. Julie reviews all of the impacts and makes the call on whether or not to accept the update. If she accepts the change, then they continue on with their work.

This work will happen regardless. It's probably being captured in PowerPoint right now. An iterative improvement would be for the team to capture their changes in a common system model and then present their changes in a dynamic interdisciplinary way, so that the design

review process can become significantly more technical and dynamic. This has been the pitch of MBSE evangelists of the past two decades, as it has attempted to be implemented at larger companies like Lockheed, Boeing, and Northrop.

Broadly speaking, if this process worked as promised, then I don't think companies would have found an edge by implementing agile tactics in place of formal systems engineering. So what's going wrong?

1. The process of generating such a system model in a specialized tool is way too difficult, so we assign specialist individuals whose job it is to create a model and attempt to keep it in sync (basically a scribe).
2. The modeling exercise often happens independently of Real Work™, since we've relegated the modeling process to a separate entity.
3. Modeling is contractually mandated and prescribed to use specific tools.

By demanding the creation of system models regardless of utility, and then demanding they be created in specific tools, you've effectively divorced the practice of Systems Engineering from literally everyone else making technical decisions. The dirty secret at these innovative "new entrants" (SpaceX is 23 years old now) is that they have distributed the activity of (lowercase) systems engineering by empowering individual owners of subsystems to make good choices and have built an environment to support the emergent system engineering behavior of the individuals engineering the system.

But again, this really only works if a few employee-centric environmental factors are implemented, as detailed above. If you're one of those literally three companies, please don't buy our software. Don't buy anybody's software marketed explicitly at deep tech companies. Go use Airtable or Excel and implement good tactics. You're also probably a bad customer, and it's better if our competition spin their wheels dealing with you on a net-negative ACV basis anyway.

But how can we capture the promised benefits of Systems Engineering we detailed before? Notably, how can we rapidly iterate on an experiment as significant as “add legs so the rocket lands” without burning the boats and editing everything going forward, only assuming success?

We need to (1) maintain system engineering as a distributed task actively supported by all engineers, and (2) make sure we actually maintain a technical baseline using a system-level model.

We then need to be able to create an indefinite amount of arbitrary duplicates of that model so that an individual empowered engineer can make arbitrary changes and assess their impact. The system-level model also needs to be robust enough to sufficiently represent the work being done in the actual engineering design tools, and potentially support interacting with these tools if the system-level fidelity isn’t good enough. We need an incredibly robust and trustworthy way of reviewing these changes in a collaborative setting. And finally, we need some kind of way of re-inserting the changes into the main technical baseline.

Enter using a few key technologies:

- Git for revision control
- A Domain Specific Language for capturing these models, fully text based so it works well with Git (SysML v2)
- Interpreters, compilers, and visualizers for SysML v2

Putting the text-based models in Git is a great starting point. A motivated and talented software developer can start today by reviewing the changes, and a lot of our earliest supporters fell under that category. This approach doesn’t simply approximate software engineering best practices, it literally adopts software engineering best practices by using the same technology that powers software engineering worldwide.

What is missing in making this more broadly appealing to engineering teams are basically three things:

1. An authoring environment for technical non-software developers (enabling them to point-and-click to draw my block diagrams or edit a table) so that they can create system models.
2. A diff tool, so they can visualize changes to these block diagrams or tables.
3. Some kind of collaborative environment to provide feedback on what they are seeing in the diff tool, and either accept or reject discrete changes to the system model.

And this is great, if you want to maintain a discrete task of Systems Engineering, or train your team to interact with the model directly via a discrete tool. An improvement to the user interfaces of such a platform might develop some progress toward democratizing access to this information. If it's built correctly, then these more agile deep-tech darlings might even adopt it.

But these statements are only part of it. Is there anything we can do to make it easier to build and maintain a more robust model without a huge demand of labor overhead? Is there an improvement we could make such that Susan and Mark could more easily capture their proposed changes? Should this even be in a discrete Systems Engineering tool, as would we expect SpaceX to even use such a discrete tool?

It's still an extra step to generate a new engineering artifact just for a conversation. Teams might have used LucidChart instead to generate a diagram, and then written some Python scripts to model out what they expected to see, or just have done the change and talked about it in a design presentation using PowerPoint. So if there isn't sufficient enough of a "stick" (the federated collaboration environment of contractors, subcontractors, and vendors) to enforce good modeling practices regardless of a tool, the carrot of having a model to make decisions against is not yet appealing enough to adopt a discrete modeling activity within the organization.

Accelerated Agile Hardware Engineering

That said, maintaining the technical baseline in a Systems Engineering tool likely would have accelerated their decision making and review process, assuming we were able to minimize the labor needed to generate and update the system model. So if we were able to use “AI” to take the above engineering artifacts (LucidChart, Excel, CAD, ERP, napkins, etc.) to generate a system model, that might be useful to this audience. We’re still in “scribe” mode, but at least it’s less burdensome to benefit from the features of a systems engineering tool. Perhaps that scribe activity could be, instead of a different human in another department, partially a result from some kind of data processing pipeline guided by the engineer who actually owns the subsystem!

I call this “Clippy-style” AI. Basically, we assume engineers actually want to be doing the work of generating system diagrams but need help interacting with a more robust platform than a commodity diagramming tool. This would occur using a natural language processing (NLP) pipeline that can digitize a PDF of requirements into whatever format your tool expects. There even could be an Agent that can suggest design changes to an engineer actively performing work within the tool.

A few startups are exploring this space, as well as a few companies exploring building plug-ins for legacy SE tools (SysGit also has supplementary capability in this space; it’s pretty much table stakes these days). This might provide some iterative improvements to folks who are model-based practitioners, but isn’t exactly a fundamental paradigm shift in engineering development. I also doubt these platforms will gain much traction in organizations that are again prioritizing a sprint through the Vee towards “find out in hardware,” as they don’t have dedicated staff supporting the old model of a model based scribe.

But let’s instead decide to legitimize the complaints made by the “new entrants” (again, Happy 23rd Birthday to Space Exploration Technologies, born March 2002, you’re actually

a full year older than the seminal *UML for Systems Engineering RFP* published by OMG in March 2003). How can we somehow bring the benefits promised by discrete Systems Engineering activities without demanding the establishment of discrete organizations to perform the more robust modeling work typically seen in the formal environment?

On Agile Design Reviews & AI Agents

Let's first assume we've agreed to store a common baseline, in a common descriptive language. This might have been generated by hand initially, and is then kept up-to-date through the engineering team committing to maintain the baseline possibly supported by the Clippy-style features. This team would also benefit from collaborative revision control features when assessing any change to the system model, and more decisions might be made virtually before committing to a hardware build ("shifting to the left on the vee" some might say).

Now imagine Google Docs. If you have four people working on the same document, you literally have four people editing the same set of text at the same time. What if you wanted to work on the same paragraph as someone else? Typically you'd copy and paste the paragraph to elsewhere in the doc, make separate changes, and talk through resolving the differences. Now imagine if one of those contributors was an AI agent making instant changes, much faster than you could get your thoughts down? That sounds annoying at best and unhelpful at worst.

Now imagine if that document wasn't a Google Doc but instead an Aircraft made in a system modeling tool, with our classical linear revision control paradigm of a single technical baseline. You basically have your AI making changes to your system model at the same time as the humans working on the model. At best, the humans and AI are locking model sets before making changes, so as to not step on each other when doing work, and at worst, are just making changes on unlocked models while overwriting each other.

Instead, let's assume we have our next generation modeling tool, based on git, with a robust diff tool for processing sprawling changes to a model.

Imagine if on Julie's First Stage team, alongside Mark and Susan, we had an AI Agent focused on the outer structure of the rocket. The AI agent isn't in charge of the outer structure of the rocket, but instead monitors changes made by a fourth engineer, Jeffrey, in his various authoring tools of choice (Excel, Confluence, Solidworks, and others). This AI agent is effectively just Jeffrey's scribe. Jeffrey's AI Agent is able to ingest the latest engineering artifacts made in Jeffrey's tools, create a branch of the system model technical baseline, and generate SysML v2 textual notation based on those artifacts, placing them in existing model files or new model files depending on relevance. Once the processing is complete, a Pull Request is opened and then any colleague of Jeffrey like Mark, Susan, or Julie can review the changes, provide feedback, and choose to accept them.

This is only possible because:

1. The teams are working from a common technical baseline.
2. The baseline uses a well-defined industry-supported language, so individuals can introspect the changes from any number of platforms.
3. The changes are captured using Git for revision control, allowing changes to be proposed without impacting the active work occurring elsewhere.
4. The changes can be gated by the review and acceptance process from others.
5. These changes can be gracefully merged back into the technical baseline.

The benefit of having an AI Agent here isn't to make arbitrary changes to the system model. Instead, it's to keep the model up to date based on developments occurring elsewhere. This also aggressively derisks the use of any kind of AI, since *a human is reviewing all proposed changes by the AI before inserting them into the model from a systems level.* Any review functionality built out to assess proposed changes from an AI can be used to assess proposed changes from a human, too! Humans should probably be reviewing all impacts to

the abstraction anyway, since it's that human understanding of design intent that will be so difficult to replicate with machines.

Recall our earlier issue preventing the adoption of formal Systems Engineering processes at other companies:

1. The process of generating such a system model in a specialized tool is way too difficult, so we assign specialist individuals whose job it is to create a model and attempt to keep it in sync (a scribe).
2. The modeling exercise often happens independently of Real Work™ since we've relegated the modeling process to a separate entity.

By smartly distributing the systems-level engineering work between teams of humans and agents, we can offload the role of a scribe to a data processing pipeline monitoring other changes. This then ensures the modeling exercise happens in parallel to Real Work™, without incurring a massive labor cost of establishing a team of scribes to try (in vein) to keep the model up to date.

Let's zoom out again, to our initial problem statements. How can we:

- Fully explore an unstable design space and converge on something novel. Nothing is off the table, from violating engine design constraints to finding additional margin, to just slapping on more engines to your rocket.
- Assemble a launch vehicle from "best of breed" hardware, from all your friends' companies. […] Why can't you integrate components like the absolute best IMU, the best rocket engines, and the best composite products?

These two statements are pretty irrelevant to the existence of an AI agent or not. However, the first statement does assume you have either a robust system-level model or a ton of money to spend on hardware iterations. The second statement is a little more nuanced. Either you have a ton of money to destructively iterate using expensive vendor hardware, or

you maintain a rigid interface boundary around the hardware so you can safely iterate destructively, or you demand your vendors provide robust models that allow you to perform these iterations virtually.

Conclusion

The enabling technology here isn't the system modeling tool, it's not a new modeling language, it's not a platform that stores the language in software infrastructure, it's not a revision control process, and it's not the purpose-built AI supporting these activities. It's actually all of these things.

The point here isn't to share a new fancy widget, the point is to somehow take the agile tactics that have worked under a very narrow set of parameters for the past twenty years, and try our hardest to generalize it for the rest of the Defense Industrial Base. The DIB wouldn't have aggressively leaned in on modeling if it didn't need to somehow figure out how to improve their timing and success in a federated environment.

I wrote this document because this is not an easy problem to describe. It's a niche technical issue that actually impacts every single billion dollar weapons system. I cannot distill this into a 10-slide investor deck, and the nuances require careful explanation to resonate with practitioners across both established companies and new entrants. But it's a problem I have spent thirteen years of my life experiencing, and over half a decade of my life trying to solve at scale. I continue to tackle this because as of June 2025, the solutions available in the market remain fragmented and don't adequately address the fundamental paradigm shifts needed. The stakes are high - America's technological competitiveness and ability to field effective defense systems depends on solving these engineering collaboration challenges.

This is not a time for half measures, and I hope you'll join me in supporting agile engineering for all programs, as soon as possible.