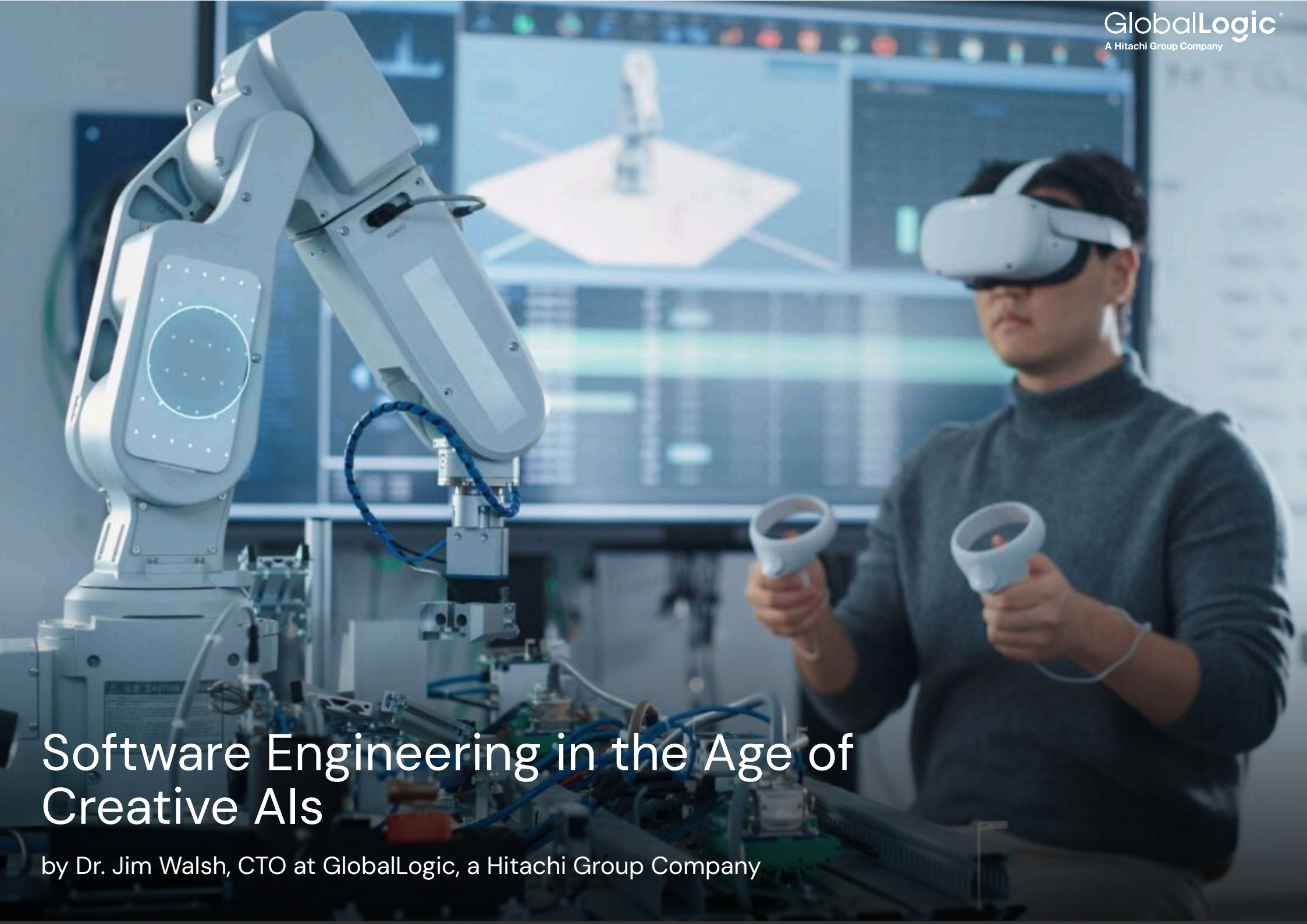


GlobalLogic®
A Hitachi Group Company

Software Engineering in the Age of Creative AIs

by Dr. Jim Walsh, CTO at GlobalLogic, a Hitachi Group Company

Thanks to Mayank Gupta, head of GlobalLogic's Global Agile Practice, for the stimulating discussion that led directly to this train of thought.

I had a great conversation recently with Mayank Gupta, the head of our global Agile Product Development practice, about how software development might change as "creative AIs" such as ChatGPT grow more sophisticated over time.

Getting to the point we envision and describe here will clearly take some time — perhaps a long time.

I'd guess (and it's only a guess) that it will take ten to twenty years of gradual progress to get to widespread adoption of a paradigm similar to the one we'll explore here, though I could be very wrong either direction time-wise.

However, I do believe it will happen, in the lifetimes and careers of many of us.

> Imagine, if you will, an AI that can write software from a more or less natural language 'specification' or 'user story.'

This AI would also have full context of the current state of the system under development — that is, it would know what already exists and how it works.

It will also know the business context, personas (user roles), and user needs the software under development is trying to address. For example:

- Is this a multi-tenant system meant to be deployed "as a Service" for many businesses and business users?
- Is this a 'single user' mobile-type application?
- Is it a game intended for entertainment?
- Is it a banking system?
- Will the UI be deployed in the metaverse?
- And so on.

Finally, this AI is well aware of both the logical and physical architecture of the system under development, including the technologies to be used.

Whether that architecture was itself designed by an AI, and whether the technologies were chosen by AI, we'll leave an open question for this discussion.

For now, let's just assume they exist and that the coding AI is well aware of both.

Today, most software systems are developed by companies like

Dr. Jim Walsh, Chief Technology Officer at GlobalLogic

GlobalLogic and many Silicon Valley as well as world-wide product development shops using an Agile software methodology, which allows rapid adaptation to changing requirements.

This characteristic of Agile will be even more important with AI-generated software than it is today for human-generated systems, I believe—as we'll see in this thought experiment.

Let's begin by imagining a day in the life of a future-world software development team.

We'll assume two things:

- that in this future world, code-generating AIs have been perfected,
- and that any code the team specifies is generated and deployed in a test environment almost instantly by an AI.

A "day in the life" of this new-world software development team might look something like the following:

8:00am – 8:15am: Morning kick-off meeting (standup)

8:15am – 10:15am:

- Review the software generated by the AI overnight.
- Execute the software in the test environment, and evaluate whether it meets the user needs.
- Identify any discrepancies between the way it works and the actually desired behavior.
- Also evaluate the as-built (generated) architecture and insure it conforms to the desired system architecture. Depending on how software work is structured, this may involve filing bugs for resolution by another team, or clarifying the specs and acceptance test cases. We'll assume the latter for this scenario (the person finding the problem does the work).

10:15am – 10:30am: Coffee / ping-pong break

10:30am – noon: Use the AI to generate a new system based on the clarified specs. Repeat the review and clarification process until last night's user stories are completed to satisfaction. Deploy to production. (This is the equivalent of a present-day sprint review and release readiness review).

Noon-1pm: Lunch

1pm – 4pm:

- Evaluate end-user feedback from yesterday's and prior deployments, and create or modify user stories to address customer issues and requests.
- Work on generating new "roadmap" functional and technical user stories, grooming the backlog, and producing new acceptance tests for all of the above—all assisted by the AI.
- Do preliminary system / code generation runs against the new user stories to evaluate the results, receive feedback on ambiguous details from the AI, and refine / expand the user stories accordingly.
- Prioritize the backlog, and decide which user stories will be staged for production tomorrow.
- Choose the appropriate stories for this evening's code generation, acceptance and regression test run. Break for water cooler conversations.

4pm – 4:45pm: Email, meetings, review customer feedback and competitive systems to define new roadmap items, etc.

4:45pm – 5:00pm: Evening stand-up. Launch evening AI run.

5pm: Head home feeling good after a job well done.

Besides being able to head home at 5pm (rare in the software world!), what's different about this AI-centric software development scenario and current development?

1. No coding or coders

The future AI does the coding work based on the 'specs' embodied in the functional and technical user stories.

The AI presumably also does dependency management between software components, ensures conformance to the architecture, and executes other currently manual tasks.

2. Testing against specs is done automatically by the AI...

BUT testing against actual functional and technical requirements has to be done by hand. The (far?) future-state AI can presumably be counted on to generate code that confirms to the specs as given, and to pass the acceptance tests enumerated in the user stories.

However, we can't count on the specs themselves, or the tests — both of which we humans wrote (albeit with AI assistance) — to accurately describe to the AI what the users and internal stakeholders really want.

This means the future equivalents of today's product owners, BAs, architects, engineers and testers will be constantly evaluating the output of the AI against what the product owner and technical architect really wanted, both functionally and technically.

3. Coding and testing is accelerated.

Today, a sprint cycle might take two weeks of coding and testing, followed by a "sprint review" from business and other stakeholders.

Then, depending on the system, deployment to production may happen immediately, or after several sprints (product increments) depending on the specific needs.

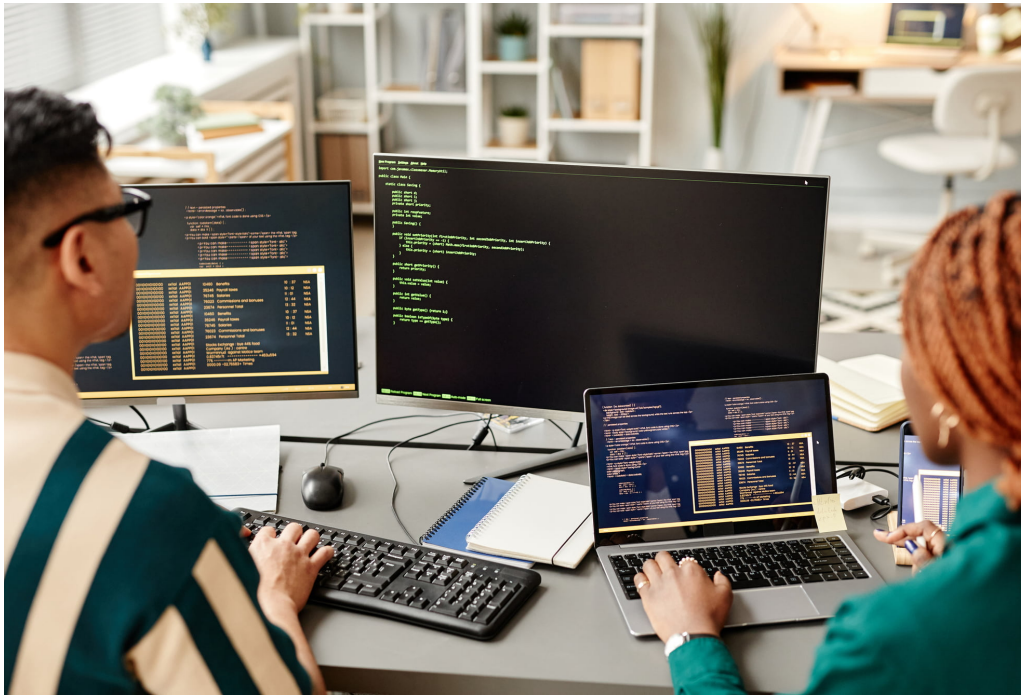
In the future AI-driven paradigm, the coding and testing portion of a sprint will be greatly accelerated.

In the scenario above, we imagine that the equivalent of a 2-week sprint in 2023 takes a day — but it could be more (probably due to human factors) or even less!

This means that as AIs mature, the development team will constantly be in what today we call Agile/ SAFe sprint review, backlog grooming, product increment planning and sprint kickoff mode.

The scenario above tacitly assumes we are developing a system more or less from scratch — that is, custom or "bespoke" software.

While such "Greenfield" system development will certainly continue to be developed, I expect that a lot of what we today consider IT-centric software development will, in the future, begin with a packaged system that a team acquires and modifies.



Today, many enterprises acquire packaged software systems such as Salesforce, ServiceNow, Workday, vertical market packages for Telcos and other industries, as well as many other systems. Companies acquire these software packages, or access to them via a SaaS model, then expend engineering effort to configure, integrate, and — in some cases — customize them.

Even where a customization of a third-party system would theoretically add business value,

though, many CIOs and CTOs limit themselves to the configuration options and external integrations supported within the purchased package.

This is because they have learned, often 'the hard way,' of the difficulty such changes can cause to subsequent upgrades and maintenance.

I believe, in a future of powerful AIs, packaged software will be more of a 'starter kit' providing off-the-shelf functionality — perhaps supported, instead of code, by the formal description of the "user stories" and "acceptance tests" used to generate and test that software in the first place.

Where it adds business value, teams will be free to modify the supplied user stories and add their own, to provide a custom solution for the area addressed by the packaged solution.

Probably multiple kits from multiple vendors can be combined to create a very powerful integrated solution, all supported by AI-generated code.

These customized packaged solutions will be 'safe' to produce because the future AI will:

- automatically manage upgrades produced by package vendor(s),
- ensure compliance to the user-defined enterprise architecture,
- and guarantee you always have a working tested solution.

Recommended: AI's Impact on Software Development: Where We Are & What Comes Next

Read Now

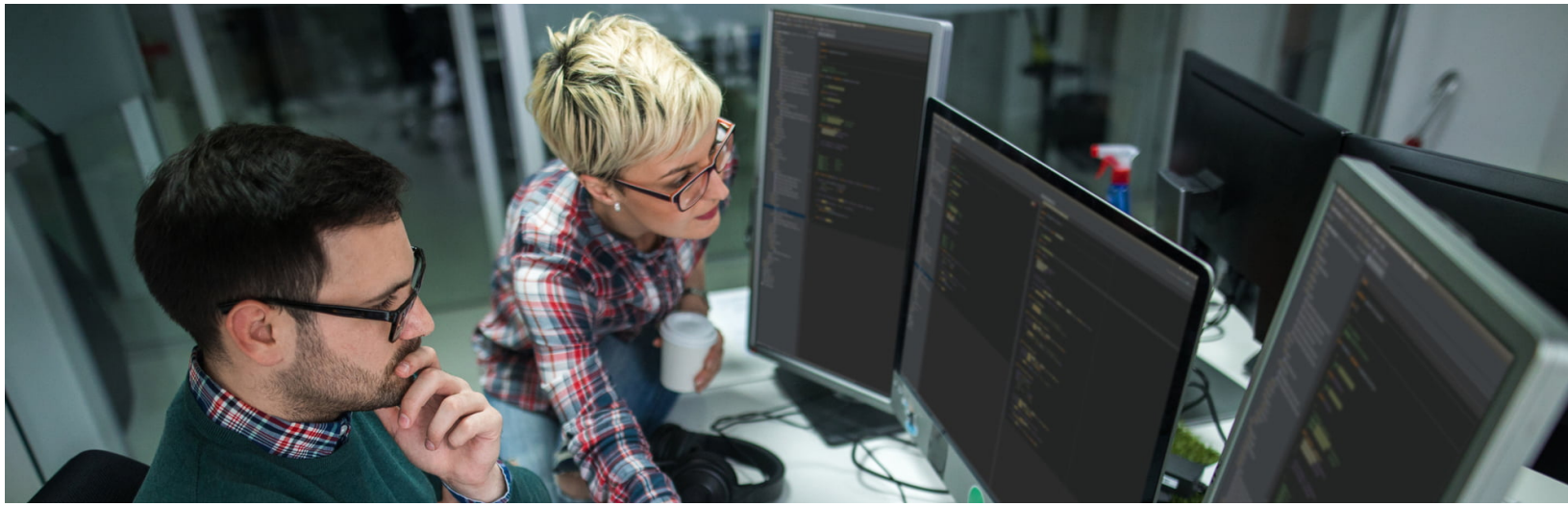
Such development could follow the same "one-day sprint" scenario defined above, but starting from an externally defined, already-existing solution base.

The idea that in the future, an AI will be able to develop all the required code for a complex system starting solely from human-readable "user stories" is, of course, speculation.

Current generation AIs, like ChatGPT, can indeed generate code from verbal or textual descriptions — but today (2023) these tend to be code 'snippets' rather than entire systems.

I believe the scale of the generated code is more a limitation of the framework humans use to describe such complex systems, than an inherent limitation of the AI technology.

If we use an architecture framework such as the legacy Rational Rose to define the desired end-state logical and physical architecture, and Agile "User stories" to unambiguously define the features the system requires, then one can see a clear path to generating a complex software system end-to-end — at least conceptually.



Making this a reality will take a lot of hard work, and may prove to be something the industry doesn't want and won't pay for!

However, I'm convinced it's possible, doable, and will happen, probably in the non-too-distant future, and (I'm pretty sure) within the careers of many current software engineers.

Another facet of the scenario above that may appear unlikely is that humans will still be required to review and correct the output of the future AI-developed software.

> If the future AI were perfected, wouldn't the code and system it generated automatically be perfect?

The idea of generating test cases and code from 'specs' is not new. In fact, I worked with code and test generation from formal specs in the early 1990's, and the subject was not new then.

The problem is that if the code and the tests are generated from the same set of specifications, then all the tests will actually verify is that "the code does what it does."

In other words, you will confirm that the code implements the spec.

In some cases, that's all you want from your tests (for a compliance test suite, or a port, for example).

In most cases, however, you really want to test that "the code does what you WANT it to do."

There are multiple possible sources of error that could cause the generated code from failing to do what you want, the major ones being:

- the code generation process,
- the test generation process,
- and the specs themselves.

If the code and test generation capabilities are correct and interpret the spec in the same way — as we would hope for a future software generation AI — then the remaining major source of error is the specs themselves.

The test cases included with the user story may well add clarity and nuance to that story.

But essentially, that makes the test cases part of the spec, and we end up with the same problem, namely: we can't verify the 'correctness' of the system (that is, whether or not it meets our needs) from within the system itself.

We can, and a future AI certainly will, evaluate the specs for internal consistency, completeness, and against other quality criteria.

In fact, we have AI-based tools that do that today. Such tools can also identify and, in the future will also generate, missing test cases (based on the spec), that ensure the generated system is thoroughly exercised.

An AI could also evaluate the specs of the current system against those of similar or related systems, and determine if a story or requirement is likely to be missing.

However, if the system is to be used by humans, then the system architects and users — or a user's proxy, such as a product owner, BA, designer, engineer, etc. — still needs to evaluate whether the resulting generated system is "as desired" or not.

Assuming the AI followed the instructions in the spec, if there is a discrepancy, it is the spec that will need to be modified and the code regenerated from it until the discrepancy is resolved.

I suspect that this will result in the same sort of iterative development process we use today, albeit at a greatly accelerated pace. This is what we describe above.

Whether this is good news or bad news, I don't know.

What is clear from this thought experiment is that the skills the current generation of engineers and designers are developing in terms of design, backlog creation, test creation, backlog grooming and other "Agile/SAFe" process tasks may very well be in even higher demand in the future than they are today.

In fact, defining what the software is supposed to do and, to an extent, how it does it technically, architecturally and design-wise, may very well become the essence of software development in the age of the "Creative AI."

Using a Future AI to Help Remove Ambiguities from User Stories

We've speculated about a "day in the life" of a future software development team.

In that future state, they will use a highly advanced AI to perform the coding, automated testing and deployment tasks needed to develop and deploy a complex system into production.

As we theorized, the core of such future software development activities is likely to revolve around the creation and correction of the specifications the AI will use to generate the software system.

Let's move on now to focus on that activity and on the work that, in my opinion, will be central to the functional/feature aspect of future software development.

We use the words "functions" and "features" here in distinction to the "technical" aspects of the system which, I believe, will be equally important human-supervised activities.However, we won't address those in any detail.

Today, the foundation of most Agile methodologies is called a **user story.**

As the name indicates, a 'user story' tells a story about something a user wants to do utilizing the software to be developed.

If we were writing a metaverse-based ATM system, for example, a human-generated user story might be:

"As a depositor, I want to securely view the value of the funds in my metaverse bank account denominated in the virtual or physical currency of my choice so that I can see how much I have available to spend."

In the human-centric software development world, our goal prior to any implementation would be to "groom" such a user story to remove as much ambiguity from it as we can.

Otherwise, we will not end up with what we really want, even if we thought we were being pretty specific.

You might think, at first glance, that the user story above is reasonably specific. However, like pretty much every user story ever written, it includes a lot of implicit and unstated requirements and assumptions.

Here are just a few examples:

- **Where do our exchange rates come from?**
  How often are they refreshed? If computed dynamically, what's the buy-sell spread for each currency?

- **Like physical currency, converting virtual currency will probably incur a transaction fee.**
  When showing the user's account balance, do we include this transaction fee so the user truly sees how much they have to spend — which is stated as the intent of the story (in the "so that" clause)? Or do we show the value at the bank's current exchange rate without showing any of the transaction fees the user would incur in actually exchanging the money?

- **Does everyone get the same exchange rates?**
  Do preferred customers or customers who have visited competitive sites get different rates, and different transaction fees, for example?

- **While the story implies that we only show one** target currency type at a time, can the user specify several?

- **Do we always show the entire account balance, or can the user specify a certain amount?**
  If it's a user-specified amount, are there any constraints on the amount or the current currency used to show that amount (for example, is the total amount of the account balance the max we can show?) Is there a minimum?

- **Is the initial currency used to display the default value** used to manage the account, or some other currency?

- **What if there are legal restrictions based on residency on the type (country) or amount of physical currency this user can convert to?**
  Do we still display it even if the user can't convert to the target currency(s) based on his or her country of origin or residence, or do we present a warning or some such?

- **Which currencies are shown for a given customer?**
  The story says the currency "of my choice", but is there a default set of currencies that are always shown, for example? Where do we get the list of currencies that are available?

- **What does the UI look like to the user?**

The process of designing and developing software is, in large part, about identifying and resolving ambiguities in the requirements, and also challenging assumptions to find opportunities for a better technical design or user experience than the one originally envisioned.

With current-day software development technology, this refinement and elaboration process continues right through the point where the engineer keys the code into his or her keyboard.

Even today, though, we try to resolve as many ambiguities as possible before we start coding.

Otherwise, we could waste time developing the desired feature in the wrong way — or the 'wrong' feature entirely.

When the as-built system differs from what the spec-writer had in mind, regardless of what the specs themselves literally say, it can degenerate into an unhealthy dynamic.

The author of the requirements — often the "product owner" — might claim that the engineer's work was buggy or that the engineers 'didn't get it,' while the engineers blame the situation on vague or contradictory requirements.

And in fact, due to time pressure, engineers often do begin work on what they know to be ambiguous requirements.

Even though the engineer might be trying to do the right thing by starting to code even when the requirements are not clear, when that engineer eventually makes a wrong guess and ends up being blamed for 'not getting it' or for 'buggy code', it's a frustrating experience.

In present day Agile/ SAFe projects, we try to combat the inherent tension between clarity and speed of execution using a concept called the “Ready” state.

A user story is said to be “Ready” for work to begin when a team has examined it and removed all the ambiguity that they can identify, through a process called “grooming” the backlog.

That process involves clarifying the story by including architecture diagrams, adding acceptance tests, wireframes, technical clarifications such as sequence diagrams and API specs, and other details.

It may also include splitting the initial user story into several stories that can each be made clear, and which collectively solve the problem.

In an ideal world, and engineer would never even start work on a story unless it was in this well-defined “Ready” state.

Sometimes this works well.

Other times it doesn’t.

In an AI-driven software development world, the AI takes on the ‘burden’ of determining whether requirements are complete and consistent.

This reminds me a bit of the ‘smart’ electric toothbrush I bought last year.

It has a small window that shows facial expressions (emojis) based on how thoroughly you have brushed your teeth.

Similarly, in the future, the AI will be the "bad guy" who takes on the role of insisting on clear requirements.

In fact, in some aggressive implementations of SAFe, the implementation team is sent home until enough stories are in "Ready" state for them to begin effective work.

While at times the ideal of an unambiguous requirement is reached, the norm in most software projects today is for engineers to proceed into development with at least some ambiguities which are not captured and addressed before-hand.

Instead, the developer uses their understanding of the system and of what they believe to be the intent of the user story to either surface the ambiguities for resolution, or else to make a guess about what the story means at coding- and testing-time.

Current-generation AIs, and probably at least those coming in the near-term future, are "text-based." They make inferences based on the language used both in the spec itself, and in a broader context.

Take these phrases, for example: **"view the value of the funds in my metaverse bank account"** and **"denominated in the virtual or physical currency of my choice."**

Looking at the words alone, and with some basic knowledge of the meaning of those terms in the context of the already-existing (or already-specified) system, we begin to spot some potential ambiguities:

- **Phrase "my metaverse bank account"** => Assumption to be validated: context of "account" is the current tenant bank the user is logged in to, in the case that the user has accounts in multiple banks
- **Phrase "my metaverse bank account"** => Clarifying questions: What if the user has multiple accounts in the current tenant bank? Which account should I use? Is this user selectable?
- **Phrase "my metaverse bank account"** => Assumption to be validated: "my" in "my account" refers to the currently logged-in user
- **Phrase "denominated in"** => Clarifying questions: Which exchange rate should be used?

In my case, it generally frowns at me because I don't brush my teeth for the full two minutes.

I told this to my dentist and she said, "That's terrific! Now I don't have to be the bad guy."

Even today it can be — and often is — frustrating to try to get a computer system to do what you want it to do.

However, there is little doubt, in the final analysis, that when things don't work the way you expect, the fault lies with the code you created and not with the compiler or computer system itself.

Similarly, when in the future an AI-generated system fails to do what you want it to do, there can be little doubt that the problem lies with the specs/user stories, not the AI.

This single fact will remove one major source of friction between those who create requirements, and those who implement them.

The requirement-creators will have essentially moved into today's "coder" role (though using much higher-level descriptions than today's code written in C++, Java, etc.).

Some current-day coders may consider this poetic justice, however I suspect that many who write code today will gradually find themselves in the 'requirements creation' role in the future themselves. The rigorous mindset required will be similar.

More Helpful Resources

About GlobalLogic, A Hitachi Company

GlobalLogic, a Hitachi Group Company, is a leader in digital product engineering. We help our clients design and build innovative products, platforms, and digital experiences for the modern world. By integrating our strategic design, complex engineering, and vertical industry expertise with Hitachi's Operating Technology and Information Technology capabilities, we help our clients imagine what's possible and accelerate their transition into tomorrow's digital businesses.

Headquartered in Silicon Valley, GlobalLogic operates design studios and engineering centers around the world, extending our deep expertise to customers in the automotive, communications, financial services, healthcare & life sciences, media and entertainment, manufacturing, semiconductor, and technology industries.

28,000+

employees worldwide

1800+

products launched annually

10 years

average tenure among top 20 clients (FY2020)

How Can We Help You?

And so on. There are many ambiguities to clarify and assumptions to validate before we can implement this seemingly-simple user story.

We assume that in the future AI-assisted 'backlog grooming' process, the AI will ask clarifying questions until the story becomes unambiguous.

The AI will also present and request validation of the 'assumptions' it makes because of learnings with technically similar types of systems—for example, that implicit references are to the logged-in user, the current tenant, and so on.

The end result will be a fully elaborated, unambiguous backlog that is "Ready" for code generation by an advanced AI.

As you can see, as is true today, there will be considerable human involvement required to ensure that the system "as generated" is the one that is desired.

Software development has been an evolution from machine language (binary/hex) to low-level assembly languages to today's higher-level languages like Java, C# and many others.

In addition, a very large set of pre-assembled virtual components have also become available.

Far from ending this progression, I think creative AIs will continue it, allowing software development teams to work at higher and higher levels of abstraction, including natural languages, and bring us to new levels of component and system re-use, as well.

ChatGPT and What Makes Us Human, by Dr. Jim Walsh on the GlobalLogic Blog

Read Next

Browse Events

Blogs, whitepapers, webinars & more on the technologies impacting your business

All Insights

Get In Touch