Artificial Intelligence and Agile Software Development:

Use Cases for AI-Augmentation

By Scott W. Ambler
https://ScottAmbler.com

Ambysoft
White Paper

Since the release of ChatGPT 3.5 in November 2022 artificial intelligence (AI) has taken the world by storm. Since then, dozens of platforms, and thousands of tools, based on AI have been released. Many of these tools are focused on augmenting (enhancing), and potentially replacing, the activities of software developers. This white paper examines how AI augmentation impacts agile software development.

This white paper explores the following topics:

1. Why should you listen to me?
2. AI in software development: What I would do
3. AI in software development: What I wouldn't do
4. Valid use cases for AI-augmentation in software development
5. Putting the AI-augmentation use cases into context
6. AI tooling for agile software development
7. AI through the agile lifecycle
8. AI and DevOps
9. My concerns about AI and agile software development
10. The future of AI and agile software development

Why Should You Listen to Me?

You likely know me as the person behind the Agile Modeling [1] and Agile Data [2] methods, or the co-creator with Mark Lines of the Disciplined Agile (DA) tool kit [3]. I've done other work in the agile space over the years, but those are the highlights. In short, I have a pragmatic understanding of agile software development. Furthermore, during 2023 and 2024 I earned a Master of Science Degree in Artificial Intelligence from the University of Leeds. I was able to leverage my data and information science experience, which goes back to the late 1980s, while working on this new credential. In short, I have a decent understanding of AI too.

AI in Software Development: What I Would Do

In general, here is how I recommend you apply AI in software development:

1. **Augment people, don't replace them**. Your goal should be to augment what you, and your teammates do, not to replace people. You want to use AI to perform some of your existing work and thereby free you up for other, more rewarding things.
2. **Augment only the activities we understand**. This is a critical point. Because you shouldn't trust what an AI produces, discussed later, you need to have the background to determine the quality of whatever the AI does. Furthermore, you need to understand how everything fits together and when you can safely apply AI in the overall process.
3. **Improve the quality of what people produce**. Many people will create a text-based thing, perhaps source code or written prose, and then submit it to a large-language model (LLM) such as OpenAI's ChatGPT or Google's Gemini and ask it to improve

their work. For example, I will often write a few lines of code, realize that it could very likely be cleaner, so I submit it to Bing and ask for a better version. Given that I'm currently working in Python, my prompt is often along the lines of "Please produce a pythonic version of [MY CODE]".

4. **Reduce repetitive work**. AI can and should be used to automate repetitive, predictable work. For example, in software development a common need is to analyze code to identify potential security flaws in it. That's hard enough with a few hundred lines of code, let alone thousands or millions of lines. An AI won't get bored while it systematically pores through your code.
5. **Work in small increments**. LLMs are good at working on small problems as the bigger and more ambiguous the problem the greater the room for questionable hallucinations. Ask an LLM to write a function, not an entire class. Ask it to write a few paragraphs, not an entire section of a document. Iteratively ask the LLM to do a bit of work, improve upon it until it's acceptable, then move on to the next bit of work.
6. **Recognize that it's about more than just LLMs**. LLMs are only one of dozens of categories of AI models, albeit a category that is easy to work with and readily available. The point is that there are many great AI-based software development tools, some based on LLMs and some on other strategies, available to you. I'll go into this in more detail in the next article in this series.

AI in Software Development: What I Wouldn't Do

I'm also seeing inappropriate applications of AI in software development, or anti-patterns if you like. Here's what I wouldn't do:

1. **Trust anything generated by an AI**. Regardless of what you may have heard, large language models (LLMs) hallucinate exactly 100% of the time [4]. The issue is the quality of the hallucination, which can range from pretty darn good to complete rubbish. You might use AI to generate a first draft of something, but you need humans in the loop to ensure that what is produced meets the actual needs of your context.
2. **Visualize processes/architectures/designs before implementation**. When it comes to software development, the "up front" activities around formulating requirements and strategizing a potential architecture/design are best left to people. Augment where you can, including brainstorming potential ideas, but leave important decisions to people. Having said this, there are many good tools for visualizing existing implementations to help you to understand the "as exists" environment. More on this in the next article in this series.
3. **Generate large swathes of code or documentation**. The more code that you generate, or the more documentation that you generate, at one time the greater the chance that the quality of the hallucination (see point #1) is poor. As pointed out earlier, iteratively generate artifacts such as code and documents a bit at a time.
4. **Create "original" IP that you care about**. First, anything that's generated by AI is not original. Second, when you ask an LLM to improve on your existing work, it isn't

clear how original this work is any more. You did the actual thinking, the AI merely suggested how to make it better. This is an incredibly slippery slope and you need to decide where to draw the line. Third, recognize that not all IP is created equal. Do I really care about a function that loops through some data? Likely not, as long as it's correct. Do I care about the overall system that I'm building that invokes that function. Very likely. Once again, where you draw the line is up to you. And no, I did not use AI to generate any of the text of this article although I did generate the image using Google Gemini.

Valid Use-Cases for AI Augmentation in Software Development

What I believe to be the valid use cases for AI augmentation in software development are summarized in Figure 1. There are several key features of Figure 1:

1. **It's a mapping, not a lifecycle**. The diagram indicates the main categories of work in software development and maps AI-augmentation use cases to them. Yes, it does look a tad waterfallish but don't get your agile knickers in a knot, in the next article I will work through a lifecycle view.
2. **It focuses on software development**. I haven't included all of the typical white-collar worker use cases around writing reports or presentations, only the use cases specific to software development.
3. **The wording is key**. It's important to note how I've worded the use cases, using terms like potential and suggest. This is because you need to "keep humans in the loop", working in such a way that a person validates the output of an AI, updates it as appropriate, then commits it to your overall asset base. An implication of this is that the human in the loop needs to be sufficiently competent to judge the AI's output. When the output of the AI is large the human in the loop may need to use another AI-based tool to help them analyze the output of the first AI-based tool.

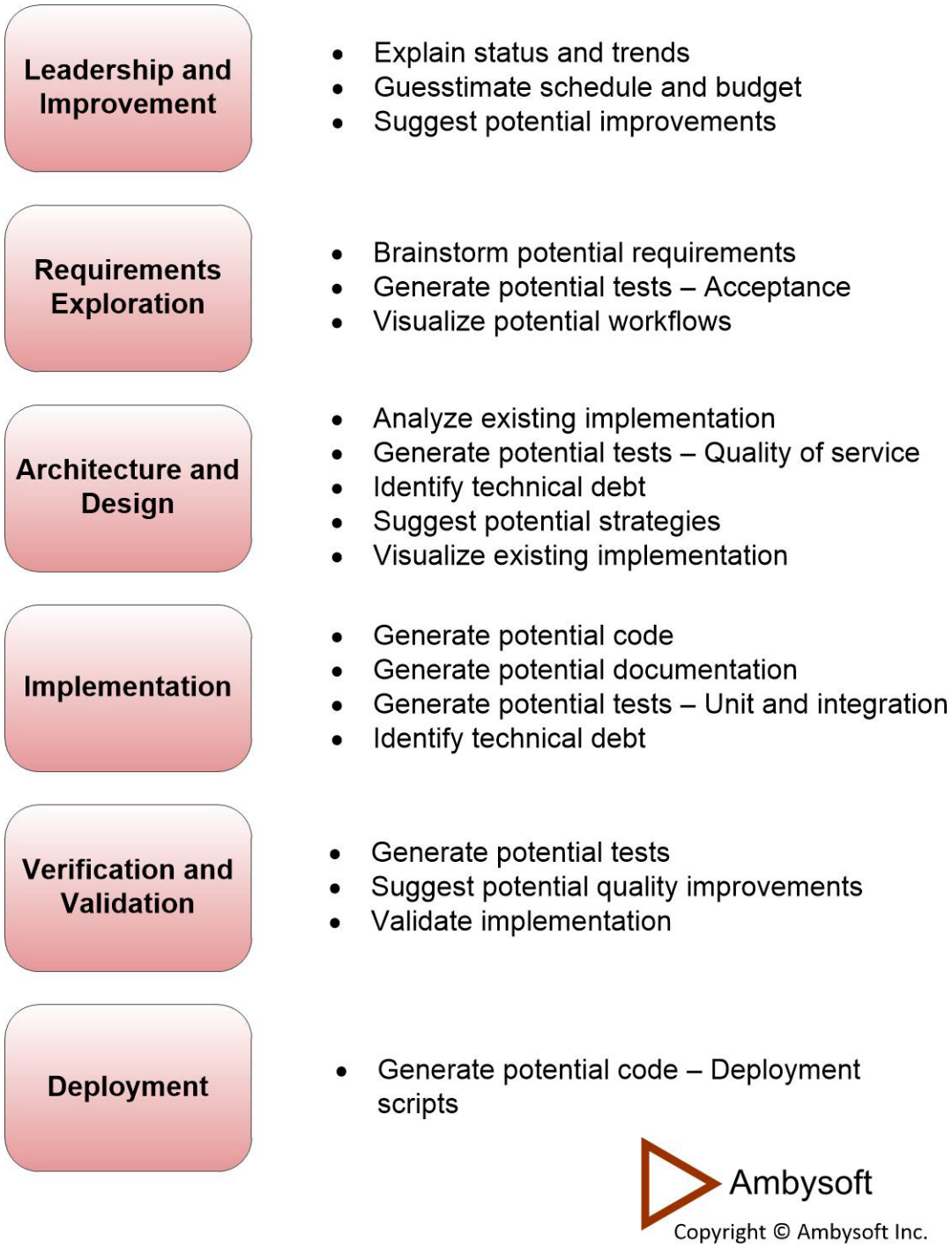


Figure 1. Potential opportunities for AI augmentation in software development.

Let's consider each of these use cases one at a time. I describe each use case and suggest tooling options that support them in the list below, and later in this article I discuss their trade-offs. For most of the use cases there are many tooling options, I'm just suggesting a few exemplars. You will still need to identify tools appropriate for your environment. If I've missed your favorite tool and you'd like to share that with others, please feel free to add a

comment to this article. However, please DO NOT reach out to me directly as I don't want to be inundated with demo offers. The AI-augmented software development use cases are:

1. **Analyze existing implementation**. Software developers often need to work with existing, legacy code that they are unfamiliar with or in some cases have forgotten about even though they wrote it. Tooling such as SonarQube and CAST SBOM Manager help developers to navigate the complexities of existing implementations and to better understand any technical debt that they face.
2. **Brainstorm potential requirements**. I often want to brainstorm requirements with my stakeholders. We'll do this for both potential high-level ideas during scoping as well as later during construction to identify potential details. Asking an AI to suggest ideas is like having another experienced voice in the room. Although large language model (LLM) such as ChatGPT or Gemini will do, an LLM aggregator such as Poe provides a flexible option.
3. **Explain status and trends**. For over a decade software teams have been using dashboard technologies to monitor and report their progress. While this is a great idea in theory, in practice it assumes that people understand the information that is being presented to them. Good luck with that. Luckily AI-based tooling such as Qlik Sense exist to explain the results of your analytics.
4. **Generate potential code**. I will often use an LLM such as Bing to suggest code examples or improvements to a snippet of code that I've written. Many developers will use editors such as Github Copilot, or plug-ins to their preferred IDE, that effectively pair program with you. Note that it isn't just about application code. For example, products like Harness.io create delivery scripts and Flow.bi for creating DataVault2-compliant raw data vaults to enable continuous data warehousing [5]. And I need to give a shout out to Brave CodeLLM which does so privately and provides explanations of the code that it produces.
5. **Generate potential documentation**. First and foremost, do your best to write clean, literate code that requires minimal documentation. Second, you're always going to need some code documentation, particularly why you wrote what you did. Where any LLM will do for this, there are products such as Mintlify that are specifically built to generate documentation from your code.
6. **Generate potential tests**. It is possible, and mostly desirable, to generate tests (or more accurately automate checks) from various artifacts. For example, you can generate acceptance tests and quality of service (QoS) tests from requirement descriptions. Unit and integration tests can be generated from existing code and scripts. Furthermore, tooling such as TestRigor enables you to create acceptance and UI-level integration tests from English prose.
7. **Guesstimate schedule and budget**. Like it or not, software developers are often asked to provide schedule and cost estimates for the current release that they are working on. Products like CostGPT step you through a series of relevant questions and then produce the basis for a guesstimate.
8. **Identify technical debt**. Technical debt refers to quality issues within your system infrastructure, potentially pertaining to architecture, design, or implementation. Technical debt is mistakenly thought of as just a source code problem, but the

concept is also applicable to source data (this is referred to as data debt [6]) as well as your validation assets. Tooling to analyze existing implementations (above) and suggest potential quality improvements (below) effectively identify existing technical debt.

9. **Suggest potential quality improvements**. AI has been used for years to analyze existing systems to identify potential opportunities to improve the quality (to pay down technical debt). SonarQube, CAST SQG and BlackDuck's Coverity Static Analysis statically analyze existing code. Applitools Visual AI dynamically finds user interface (UI) bugs and Veracode's DAST to find runtime vulnerabilities in web apps.
10. **Suggest potential strategies**. Software developers can work with LLMs to suggest potential architecture, technical design, and UI design strategies for what they're building. Modeling tool vendors are building copilot/advisor functionality into their offerings as well.
11. **Suggest potential WoW improvements**. Effective software developers choose to improve their way of working (WoW) based on their learnings over time. Techniques such as retrospectives can be used to identify and explore challenges that you're facing and even identify potential improvements. AI-based tooling such as Effilix will process transcripts from such discussions, transcripts which themselves can be generated via AI-based tooling, to generate potential WoW improvements that your team might choose to adopt. This enables your team to adopt a guided continuous improvement (GCI) [3] strategy and your organization's coaches and project managers to extend their range.
12. **Validate implementation**. Many of us work in organizations that have common guidelines and standards that we are expected to follow, and increasingly in environments where regulatory compliance is required. Many of the products for suggesting potential quality improvements are applicable for this. Furthermore, products such as Compliance.ai can identify and help you to understand your obligations under applicable regulations.
13. **Visualize existing implementation**. Software developers regularly work with legacy implementations that they need to leverage and often evolve. This software is often complex, using a myriad of technologies for reasons that have been forgotten over time. Tools such as CAST Imaging visualizes existing code and ChartDB visualizes legacy data schemas to provide a "birds-eye view" of the actual situation that you face.
14. **Visualize potential workflows**. When software developers extend existing systems they need to ensure that what they are doing fits into the existing workflow appropriately. Visualization tools such as Decisions Process Mining analyze existing systems and visualize the supported workflow. Existing workflow modeling tools are also adding AI-based functionality to do the similar things.

Putting the AI-Augmentation Use Cases into Context

The use cases all make a lot of sense, but that doesn't mean that all of them are applicable to you, at least not right now, given whatever situation that you face. You want to adopt the right strategies, and select the right tooling that supports those strategies, for you and your team. Table 1 puts the AI-augmentation use cases into context by examining their advantages and disadvantages.

Use Case	Advantages	Disadvantages
Analyze existing implementation	• Enables developers to navigate complexities of legacy code. • Pinpoints potential problems in your code. • Thoroughness: The AI is likely to perform a more thorough job than people would. • Trustworthiness: The AI is less likely to be biased than individuals in what it produces.	• Accuracy risk: Although AIs can often be built with greater accuracy than people, they still aren't perfect, and any results shouldn't be taken verbatim. • Intellectual property (IP) risk: Is the AI retaining your IP after use? • Privacy risk: Are you providing personal identifying information (PII) to the AI?
Brainstorm potential requirements	• Provides input into discussions with stakeholders about their requirements. • Greater range: The AI is likely trained on a greater range of examples and contexts than that experienced by the people on your team, potentially giving a more robust result. • Trustworthiness (see above).	• Doesn't remove the need to work closely with stakeholders to understand their actual needs. • The AI isn't producing unique ideas, merely repeating what it has seen before. • IP risk (see above).
Explain status and trends	• Increases the chance that the users of data products such as	• Unless the tooling is specifically trained for software

	dashboards or reports understand the implications of the information being presented. • Greater range (see above). • Thoroughness (see above). • Trustworthiness (see above).	development metrics, you will need to invest time training it. • Accuracy risk (see above). • IP risk (see above).
Generate potential code	• Increases the rate at which code is written. • Potential for developers to become proficient with a language quicker. • Greater range (see above).	• Doesn't turn bad programmers into good ones. • Code quality decreases the more code that is generated at any one time, so my recommendation is to incrementally generate and validate a few lines at a time. • Accuracy risk (see above). • IP risk (see above).
Generate potential documentation	• Increased documentation accuracy when you generate it just in time (JIT), ideally when you consider evolving your implementation. • Can generate documentation for regulatory compliance purposes. • Thoroughness (see above).	• Existing, "legacy" documentation tends to be out of sync with the implementation. You are generally better off writing clean, literate code that doesn't require documentation. • The most important thing to document is why you did what you did, something the AI can't generate. • Accuracy risk (see above). • IP risk (see above).

		• Privacy risk (see above).
Generate potential tests	• Can generate tests to use as executable specifications [7]. • Greater range (see above). • Thoroughness (see above). • Trustworthiness (see above).	• Accuracy risk (see above). • IP risk (see above). • Privacy risk (see above).
Guesstimate schedule and budget	• Likely to consider a more comprehensive range of issues than people do. • Automates bureaucracy that developers typically don't like nor are very good at. • Trustworthiness (see above).	• Very difficult to be accurate because of the bespoke nature of requirements, significant variation in developer skills, and evolution in tooling and technology. • Accuracy risk (see above). • IP risk (see above).
Identify technical debt	• Pinpoints aspects of your implementation that potentially should be improved. • Greater range (see above). • Thoroughness (see above). • Trustworthiness (see above).	• Accuracy risk (see above). • IP risk (see above). • Privacy risk (see above).
Suggest potential strategies	• Greater range (see above). • Trustworthiness (see above).	• Accuracy risk (see above). • IP risk (see above). • Privacy risk (see above).
Suggest potential WoW improvements	• Greater range (see above). • Thoroughness (see above).	• Accuracy risk (see above). • IP risk (see above).

	• Trustworthiness (see above).	• Privacy risk (see above).
Suggest potential quality improvements	• Greater range (see above). • Thoroughness (see above). • Trustworthiness (see above).	• Accuracy risk (see above). • IP risk (see above). • Privacy risk (see above).
Validate implementation	• Greater range (see above). • Thoroughness (see above). • Trustworthiness (see above).	• Accuracy risk (see above). • IP risk (see above). • Privacy risk (see above).
Visualize existing implementation	• Thoroughness (see above). • Trustworthiness (see above).	• Accuracy risk (see above). • IP risk (see above). • Privacy risk (see above).
Visualize potential workflows	• Thoroughness (see above). • Trustworthiness (see above).	• Accuracy risk (see above). • IP risk (see above). • Privacy risk (see above).

Table 1. Software development AI-augmentation use cases in context.

The fundamental challenge is that you need to be realistic about how AI can augment software development. You still need humans in the loop that understand the part of the software development process to validate the AI's output. Furthermore, they need to be able to integrate that output successfully into the overall whole. The smart strategy is to work incrementally in small steps, ideally with active stakeholder participation [8] on a regular basis.

AI Tooling and Agile Software Development

First, let's begin with fundamental advice around AI tooling. In priority order, here is what I recommend for agile software development:

1. **Adopt AI tooling to enable people to focus on collaboration**. A key aspect of agile is on helping people work together more effectively. The implication is that if AI tooling is to support agile software development, it must either enable better collaboration between people OR it must automate away mundane tasks and thereby free up agile developers to focus on better collaborations.

2. **Adopt task-specific tools**. A task-specific tool does one thing well. For example, a tool that helps my team to identify potential requirements for a data product is task specific. A tool that brainstorms requirements for any type of domain is not task specific. Task-specific tools tend to provide better results although require you to adopt and then learn a greater number of tools. A great resource to help find AI-based tools is the There Is An AI For That portal [9].
3. **Adopt orchestration tools**. You need a tool to string together your other tools into your desired workflow. My recommendation is to keep this orchestration tooling slim, plugging your other tools into it as appropriate, and avoid the "one-stop shop" vision of large vendors.
4. **Build a data vault for IT**. Just like your business leaders need a source of high-quality data to make data-informed decisions [10], so do your IT leaders. The implication is that your organization needs a data vault [11], or a hand-jammed equivalent (usually not a great idea), that captures intel from tools and systems across the entire lifecycle from which to report from. Such a "data warehouse/lake house/mesh/fabric" provides the key monitoring and governance benefits of ALM without vendor lock in.

AI Throughout the Agile Lifecycle

Figure 2 maps the AI-augmentation uses cases described above to the phases of the agile software development lifecycle [12]. Three important points about Figure 2:

1. **It depicts explicit phases**. Unlike what the agile purists want you to falsely believe, agile software teams clearly go through phases [13] in their WoW.
2. **Deployment should be an activity**. Deployment is ideally a fully automated activity, not a phase. Not every agile team has reached that level of sophistication yet, unfortunately.
3. **Operations is outside development**. The lifecycle depicted in Figure 2 includes an operations phase, making it more than just a development lifecycle. I do this to make it easier to transition to a discussion of how things map to the DevOps lifecycle in the next section.

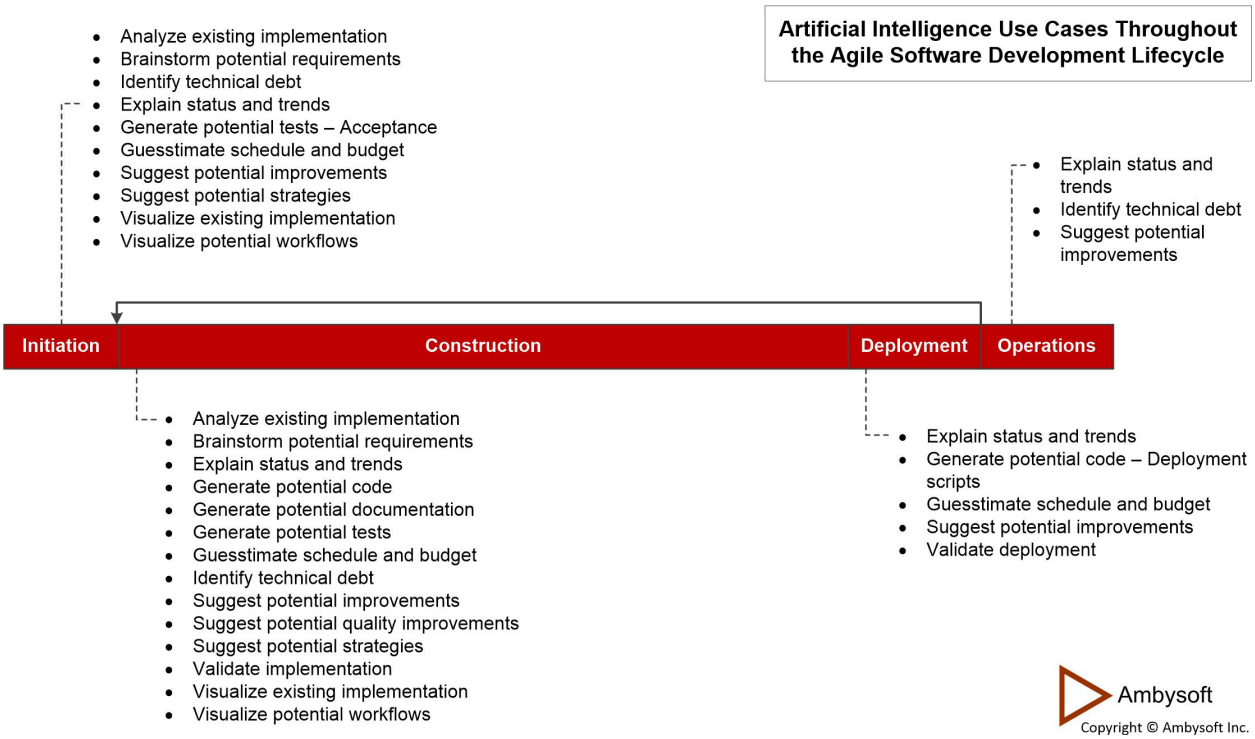


Figure 2. Mapping AI use cases to the agile software development project lifecycle.

Just as your focus changes each phase, so does your usage of AI. During Initiation you'll mostly apply AI tooling to understand the context that you face, to generate ideas, and to support planning at a high level. During Construction much of that continues, albeit now to explore details rather than high-level concepts, but because your focus is now on producing a consumable solution you will adopt many technically oriented AI tools. Deployment is also supported by technically oriented tooling, although frankly the DevOps folks have already done all the heavy lifting in this space years ago. Finally, any AI-augmentation for Operations tends to be on more sophisticated monitoring, particularly around security concerns.

Let's take a more detailed look at Figure 2 a phase a time:

1. **Initiation**. The Initiation phase is sometimes called Sprint 0, Inception, Envision, Ideation, or Start Up. Your goal is to get going in the right direction by coming to an agreement around what you're going to do and how you're going to do it. This requires close collaboration between your team, which is likely still in the process of forming during this period, and your stakeholders (a group that is also evolving). AI can be used to augment the brainstorming of potential requirements, exploring your existing technical environment, initial estimation, team planning, identifying potential architectural strategies, and many more activities. Effective initiation can dramatically reduce the risk of your initiative by working through important concerns before jumping into Construction, but can increase risk if this phase stretches on too long or goes into too much detail [14]. AI tooling offers the potential to stream initiation activities. The focus of AI-augmentation during Initiation is focused on

understanding the context that your team faces, idea generation, and planning support. All of this is performed at a high level, the details will come during Construction.

2. **Construction**. During the Construction phase - sometimes called Development, Engineering, or Implementation - your goal is to produce a potentially consumable solution [3] for your stakeholders. Agile teams typically organize Construction into short timeboxes called sprints or iterations, whereas lean or DevOps teams works in a more continuous manner. AI tooling can be applied for all sorts of technical tasks, freeing developers to focus on understanding the true needs of their stakeholders. Although most AI-augmentation during Construction automates technical tasks, the idea generation and context exploration that was critical during Initiation remains important because now you're dealing with the details.
3. **Deployment**. During the Deployment phase – sometimes called Release, Installation, Transition, Ship, or mistakenly "the hardening sprint" – your goal is to release your solution into production. Earlier I pointed out that deployment is hopefully a fully automated activity rather than a multi-day phase. Classic DevOps tooling for continuous integration (CI), automated testing, and continuous deployment (CD) do most of the heavy lifting here. AI is being leveraged in testing and validation tooling to improve the quality checking that occurs before deployment. After deployment, assuming you keep your team together, you go back and continue with more construction work.
4. **Operations**. During the Operations phase - sometimes called Production, Usage, or Run - your goal is to operate and support your solution in your production environment. Operations is outside the scope of the agile software development lifecycle, but it is an important aspect of the DevOps lifecycle and certainly something that mature developers concern themselves with. AI augmentation is commonly used by operations teams to interpret events as they occur in real time, particularly around security threats and operational outages. Important stuff for operations professionals, and also for anyone in a product-oriented role, such as a product manager.

AI and DevOps

I would be remiss if I didn't perform the same mapping for a continuous lifecycle that I did above for the project-based lifecycle of Figure 2. Figure 3 depicts such a mapping. The detailed explanation for this lifecycle is like that for the agile project lifecycle provided above, albeit organized by activity (envision, implement, ...) rather than phase. I believe it's obvious, but if there's demand for such an explanation then I'm happy to add a fourth article to the series. Feel free to add your thoughts to the comments section.

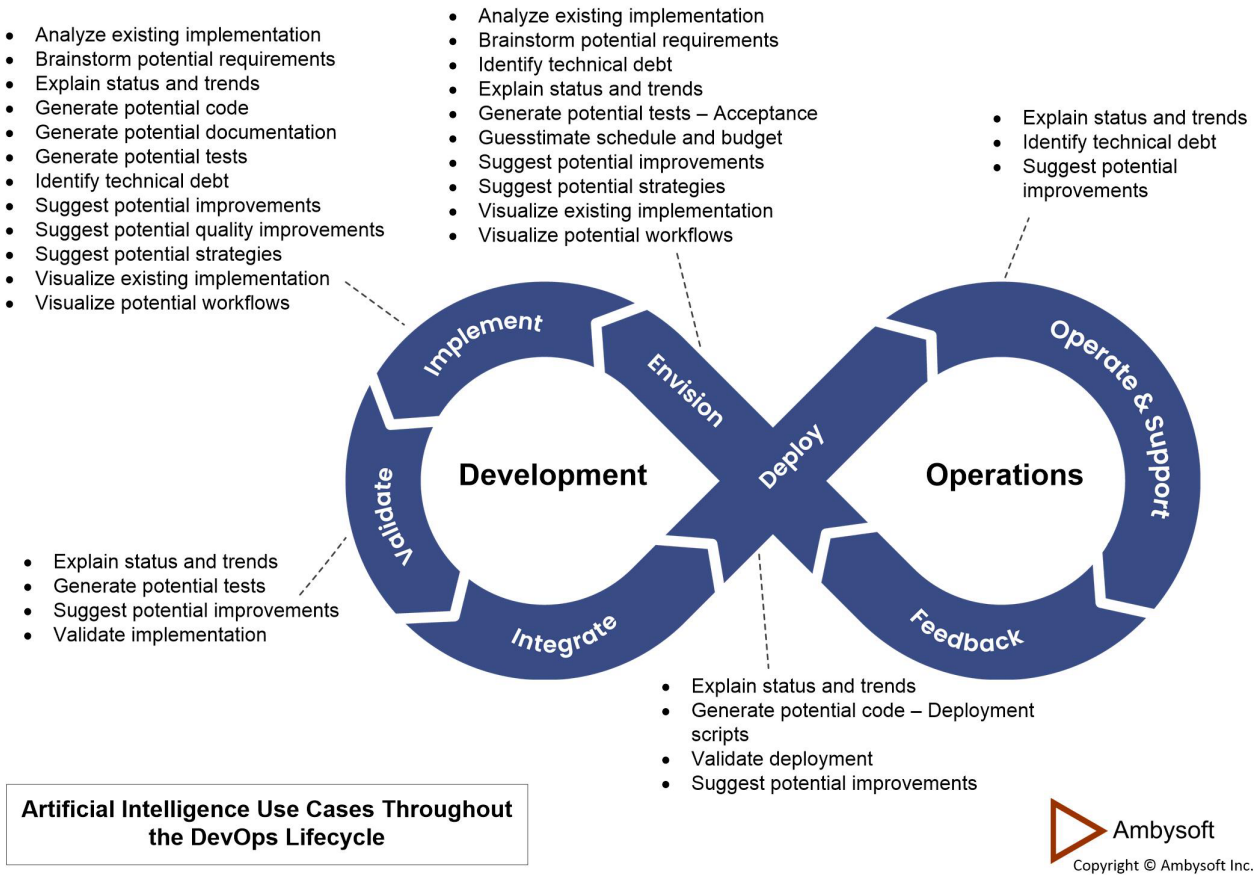


Figure 3. Mapping AI use cases to a continuous DevOps lifecycle.

My Concerns About AI and Agile Software Development

As Robert Heinlein was wont to say "there ain't no such thing as a free lunch", or TANSTAFL for short. There are several significant concerns regarding AI for software development that you need to be aware of:

1. **AI lacks an understanding of your context**. Important aspects of your organizational context include your values, your priorities, your desired direction, your guidelines, your preferred ways of working (WoW), and so on. You can certainly address these issues via strategies such as retrieval augmented generation (RAG), but it takes investment to do so. Does your organization understand and accept this?
2. **Your organization needs to work through the long-term implications**. The unconstrained use of AI in software development runs the risk of creating exponentially more technical debt. If you apply AI in the hopes of reducing IT staff, are you prepared to lose that organizational knowledge? Will you be able to easily and safely update AI-generated software when your stakeholder needs change?
3. **Your leadership may have unrealistic expectations**. Has real, and coherent, strategy work taken place around the application of AI within your organization? If not, then leadership may mistakenly believe one or more of the following: They

won't need software developers anymore; any software developers they do hire can be lower skilled and thus cheaper; or AI can perform all aspects of software development. If any of those beliefs are prevalent within your organizational leadership then consider them very serious red flags. You may want to reach out to me to discuss the possibility of me facilitating a strategy workshop to help to get you on a better trajectory.

4. **Intellectual property (IP) rights are still being worked out**. We're currently in the wild west phase of AI, and will be for several more years, but eventually the adults in the room will get things worked out. The issue is that large language models (LLMs), which many AI tools are based on, were often trained on material that the model vendors don't own the IP rights to. There could be serious legal implications from using that ill-gotten IP to generate what you believe to be your IP just a few years down the road. Does your organization understand and accept this risk?

The Future of AI and Agile Software Development

Looking into my crystal ball, here's what I see for AI and agile software development:

1. **AI is here to stay**. The AI tooling available right now is freaking amazing and it's only going to get better. Strap in, it's going to be a heck of a ride for the rest of the decade.
2. **We'll always need software developers**. AI is going to reduce the need for developers, but that need will never drop to zero. AI isn't going to put you out of a software development job, someone else using AI will.
3. **Higher-level skills will be critical**. The tasks being automated are those that are repetitive and onerous in nature, and that's mostly technical stuff. Higher-level tasks, those that require experience, knowledge, and often "people skills", are being augmented but not fully automated. This includes business analysis, architecture, design, exploratory testing, and team management activities. The days of getting by with a two-day agile certification are long gone and frankly should never have existed.
4. **People skills will be important, but not sufficient**. In point #3 I described the need for experience, knowledge, and people skills. It wasn't just people skills, and people skills wasn't at the top of the list. You must have a solid understanding of the implications of what the AI tooling is doing for you, otherwise it will be doing it to you. A fool with an AI tool is still a fool.
5. **Domain knowledge will be critical**. Always was, always will be.

Finally, some advice for tool vendors, until now the unsung heroes of my story. My advice to tool vendors is to build great tools that are easy to integrate with other great tools. Focus on addressing a specific niche and on ease of integration into tool chains. Having worked for and with tool vendors throughout my career, my experience is that just as software developers must work collaboratively to succeed, so most tool vendors.

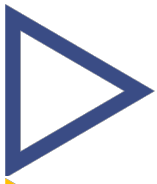
References

1. The Agile Modeling site. https://AgileModeling.com/
2. The Agile Data site. https://AgileData.org
3. Ambler, S.W. and Lines, M. (2022). Choose Your WoW! A Disciplined Agile Approach to Optimizing Your Way of Working – Second Edition. PMI Press. https://scottambler.com/choose-your-wow-2-e/
4. Ambler, S.W. (2024). Large Language Models (LLMs) Hallucinate 100% of the Time. https://scottambler.com/choose-your-wow-2-e/
5. Ambler, S.W. (2017). Continuous Data Warehousing: A Disciplined Approach. https://agiledata.org/essays/disciplinedAgileDW.html
6. Ambler, S.W. (2008). Data Debt: Understanding Enterprise Data Quality Problems. https://agiledata.org/essays/dataTechnicalDebt.html
7. Ambler, S.W. (2004). Executable Specifications: An Agile Core Practice. https://agilemodeling.com/essays/executablespecifications.htm
8. Ambler, S.W. (2002). Active Stakeholder Participation: An Agile Core Practice. https://agilemodeling.com/essays/activestakeholderparticipation.htm
9. There's An AI For That. https://theresanaiforthat.com/
10. Ambler, S.W. (2024). Systems are Data Driven, People are Data Informed. https://scottambler.com/data_driven_vs_data_informed/
11. Data Vault Alliance site. https://datavaultalliance.com/
12. Ambler, S.W. (2007). The Agile System Development Lifecycle (SDLC). https://ambysoft.com/essays/agileLifecycle.html
13. Ambler, S.W. (2010). Agile Phases? Yes, Agile Lifecycles Have Phases. https://ambysoft.com/essays/agilelifecyclephases.html
14. Ambler, S.W. (2002). Big Modeling Up Front (BMUF) Anti-Pattern. https://agilemodeling.com/essays/bmuf.htm

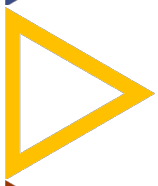
More About Me

I have other writings about AI in my blog on ScottAmbler.com and writings about software development at Ambysoft.com. In my consulting practice I focus on helping teams and organizations to improve their ways of working (WoW) and ways of thinking (WoT), particularly around data activities that are critical for AI/ML development. I also give presentations at conferences, to user groups, and to teams within organizations about a range of topics, including this one.

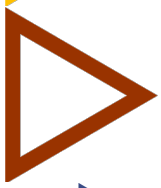
I freely share a lot of ideas and information at the following sites:



AgileData.org



AgileModeling.com



Ambysoft.com

ScottAmbler.com