The Agile Manifesto: A Software Architect's Perspective

This item in japanese

Key Takeaways

- While the role and responsibilities of a software architect can be seen as contradictory to the values of the Manifesto for Agile Software Development, a good architect finds techniques that support an agile development team.
- Working software over comprehensive documentation – An architect's goal must be to support working software. Therefore, the diagrams he creates are beneficial only if they help focus the conversation, provide vision and guidance, and enable development.
- Individuals and interactions over processes and tools – A good architect must actively participate on the agile team by writing and reviewing code, mixing functional user stories with technical tasks, avoiding technical debt, facilitating team agreements, and supporting developers to organically grow their skills set.
- Responding to change over following a plan – The architect should always have a plan that can support and enable future business requirements. He must constantly assess incoming use cases that are on the critical path to evaluate their risks and potential impact on the current design.
- Customer collaboration over contract negotiation – An architect's customers include external stakeholders as well as the development team, each with their own concerns. An architect must combine collaboration and negotiation skills in order to find the right balance between business, technology, and people.

Most software development teams understand the "WHAT" part described by the Agile manifesto. However, for a software architect, the challenge is in "HOW" to be agile, as it is different than how a developer accomplishes the same goals. Besides that, some agile teams may perceive the software architect role as unnecessary, fuzzy, or even running against the principles of agile software development.

Although some agile frameworks try to create a paper definition for the role of a software architect, this can lead to some people following the guidance, but not being as agile and pragmatic as they should. A good architect must continuously adapt their behaviour, not be biased toward one particular framework or methodology, stay relevant to the process, and provide the necessary guidance and vision for the team. When the right mental model is acquired, it implicitly leads towards a natural adoption of Agile Manifesto values, which are the pillars for all agile software development methodologies.

Agile value: working software over comprehensive documentation

In the past, especially in the waterfall era, terms like "astronaut architect" and "ivory tower architect" were coined to define a person who lives too far from the team on the ground. They added no real value to the software itself, being more focused on creating comprehensive documentation, in the form of diagrams and specs, but never creating working software. Unfortunately, this misconception still persists today, even though the way we develop software has changed massively.

Favouring "working software" does not mean the documentation must be neglected. These two facets are not contradictory and they should not exclude each other; instead, they must be wrapped together in the final product. A software architect must remember the final statement of the Agile Manifesto: "While there is value in the items on the right, we value the items on the left more."

Comprehensive documentation might become a trap

Specifications with an architectural impact (in the form of new user stories) should be tracked by the architect and assessed in a pragmatic approach by the whole development team, including experienced developers, test engineers, and devops. Bad habits from the past, when the architect created on paper the full blown technical design for the team, do not fit within modern agile environments. There are multiple flaws with this model, which I also faced in my daily basic work.

First and most important, the architect might be wrong. This happened to me after I created a detailed upfront technical design and presented it to development team during Sprint refinements. I got questions related to cases I did not think about or I failed to take into account. In most of the cases, it turned out the initial design was either incomplete or impractical, and required extra work.

Big upfront design limits the creativity and autonomy of the team members, since they must follow a recipe which is already granted. From a psychological standpoint, even the author might become biased and more reluctant to change it afterwards, trying to prove it is correct rather than to admit its flaws.

The architect will also become a bottleneck for the team since he will be struggling to provide the accurate detailed design upfront. How scalable and autonomous is a team which fully depends on the architect to always provide an upfront design? Experienced developers like to be involved and to contribute in designing the system, hence such a directive style does not fit them. They might say, "Do not provide us the solution upfront, but rather tell us the business requirements and let us (together with the architect) come up with the suitable design". Of course, this does not mean the team can do whatever they want; there must still be some mutual agreements between the architect and team members, including architectural core principles and design guidelines every member must understand and follow.

Just enough documentation as an enabler for developing software

The architect must still consider the overall picture and keep on maintaining the technical integrity of the system, taking care with integration points. There are cases when the architect should perform upfront analysis to assess the upcoming business specifications in order to evaluate the potential risk in regards to current architecture and make it transparent to all stakeholders. This might be crucial when integrating with new external systems or for internal functionalities with an architectural impact. In either case, it becomes easier if the architect prepares a few high-level design scratches for the team. A rough component diagram or some functional use cases can form the basis for further discussions and help the team have an overall understanding across all involved components.

Nevertheless, when it comes to detailed technical design, my advice is to involve the team, brainstorm and agree together. This collaboration should happen neither too early nor too late in the development process. For example, in the case of a regular sprint, the architect and the team could reach consensus during refinement meetings, prior the sprint starting. All implementation details (e.g. internal APIs for intra-components communication, multithreading model, database entity model, etc) which pertain to that technical solution could be further discussed at the beginning of the sprint, as part of the task itself.

Creating and updating technical documentation must be tackled under the same task (in the same sprint), as part of the continuous development process and as a shared responsibility across all team members. Nevertheless, the architect should supervise and ensure this process runs smoothly and becomes part of the team culture, avoiding technical depth and keeping the documentation updated.

When documentation matters

There are a few specific cases when documentation plays an important role and the architect should deliberately consider it.

- In cases of a new or recurrent class of problems within a specific business domain, the architects and senior engineers should collaborate to create a reference implementation, including associated diagrams. This allows the team to focus on the business problem they have to solve, without worrying about the architectural approach and cross-cutting concerns.
- Where there is a need to integrate with external systems, an end-to-end integration flow (e.g. detailed sequence of API calls) should be provided as a guideline implementation to the team.
- When the project needs to be handed over to another team, documentation will speed up the transition process, facilitating a more efficient ramp up.
- In case of a shared libraries or frameworks, the quality and the amount of available documentation, including forums and tutorials, is a fundamental pillar to facilitate community adoption.
- To capture the important architectural design decisions and tradeoffs.

I have previously discussed many additional concerns regarding architectural documentation, including why and how much documentation is really needed, how to identify the real beneficiaries, how to properly structure it, how to maintain it, etc.

Agile value: individuals and interactions over processes and tools

The irony of agile frameworks (e.g. Scrum, SAFe) is that they were developed to define a process, but then people started to use tools to codify the process (i.e. Agile software management tools which follow the Scrum/SAFe process: product planning, release planning, sprint planning, sprint tracking, sprint review, etc.). This might sound in direct contradiction to the agile manifesto principles, however, in reality there is always a mix which depends on the organization, product, teams, etc. What is important is to remember how to "be agile" rather than just "do agile"; valuing communication, continuous collaboration, getting to know each other better, developing trust and bonds, and listening to the other's opinions. Bureaucratic processes along with inappropriate tools add more friction, slowing down a team's velocity.

Examples of how an architect can favor individuals and interactions are perhaps best described using some real-life case studies.

The architect must write and review code

In order to achieve and maintain a good collaboration within the team, and be constantly involved in the development process, the architect must be part of the agile team. This means actively writing and reviewing code, taking care of new feature requests, and supervising the end-to-end delivery process.

An architect might spend less time on coding activities compared to a developer, but it is extremely important to keep writing code as much as possible. For example, on average I spend at least 30-40% of my availability writing code (sometimes even an entire day), while the rest is consumed by meetings and other technical activities (e.g. technical design, assessing future business specifications). The nature of the development tasks could differ; the architect must be primarily focused on writing code on the sensitive or architecturally critical parts of the application. Or, depending on the team environment, they could pick up whatever tasks comes in, just as any other senior developer might.

Besides coding, the architect must also be actively involved in reviewing others' code, which increases the awareness and knowledge of the codebase across different modules. A primary concern must be on reviewing code for those parts of the application which are fundamental for performance, security or other quality attributes. Architectural code reviews should also make sure the system boundaries (e.g. APIs, communication types, messages, etc), architectural core principles and design guidelines are not violated. This helps the architect to ensure consistency and integrity between the paper design and real implementation.

The only way to have a better understanding about real challenges and to provide meaningful technical solutions is to "keep your hands dirty." Writing and reviewing code keeps the architect closer to developers, leading to a better understanding of their real problems and needs, and fostering an in-depth product knowledge.

The architect must constantly facilitate team consensus and eliminate bureaucratic processes

Organizations love processes, and management is keen on creating "rules" for the teams. The bigger an organization is, the more exhaustive and bureaucratic such processes can be. To get a quick grasp of the company's bureaucracy, count the number of management layers above you; most of the time they are directly proportional. When it comes to software development, I would like to talk about two specific processes that provide both an immediate and long-term impact on the team: code reviews and code quality gates (e.g. code coverage).

The architect and the team must be actively involved in assessing what is really necessary for the product to meet business goals and fulfill stakeholders needs. Upon reaching team consensus, document and communicate it back to stakeholders, in a bottom-up manner, not vice-versa. If the team is part of the decision, they will be confident this is the right approach and will support it. Having code review guidelines agreed by the team saves a precious amount of time and prevents over-arguing or over-challenging things during code reviews. Besides that, it leads to a homogenous and consistent source code, since developers will follow a similar set of review patterns, as well as implementation guidelines.

Code quality gates (including code coverage) address the internal quality of the product. Improving the internal quality leads to enhanced external quality, which is visible to the customer and product beneficiaries. Maintaining a good internal quality of the product should be part of the team culture, a culture that everybody embraces. Conversely, imposing arbitrary code coverage thresholds, or any kind of quality gate, on the team will lead to artificial and inefficient activities. Developers will struggle to adjust their code (including developing synthetic test cases) just to meet those metrics. Even worse, they might not take them seriously, since they are not convinced about the real benefits.

Any processes with an immediate impact on the development process must be driven, owned, and constantly reviewed by the team. Rather than imposing top-down rules, use retrospectives to discuss how code review guidelines and internal quality gates can improve the quality of the software. The architect should really challenge and try to prevent situations when they are externally defined and imposed on the team.

The architect must stay closer to developers, being their servant but also their protector

Place your workstation next to developers and be actively involved in conversations, like any other team member. This improves the collaboration and inter-human relationships, creates stronger connections, and a better team spirit. Individuals and interactions are way more efficient than any formal process. Constantly keep on asking team members, especially developers, if they need help, and reflect on what you can do better to support them. Colleagues will appreciate your approach and will perceive you as a real team player. When they have a problem, they won't hesitate to come and ask for your advice, since they know you can support them. They should see you as a leader, not a manager with another hat. Do not label yourself an architect; instead, let others appreciate your contribution and value your role as an architect. Be the facilitator and the servant of the team, not their boss.

Sometimes the architect must shield the team from client, product owner, business analyst or even scrum master pressures. You might argue that methodologies like SCRUM or SAFe explicitly delegate this responsibility to the scrum master. But in reality, I faced situations when even the scrum master becomes demanding and tries to push things over the line, at a potential risk of compromising the internal quality. Also, the client expects more and more features to be delivered, ideally quicker and at a lower cost. Sometimes the product owner or business analyst might want to add or amend user stories during the Sprint, which leads to an increased development effort. In all of these situations, the architect should throttle the external pressure and remove such bad habits. Have discussions with the involved parties and make them aware of the long-term consequences, such as people becoming unmotivated and potentially leaving the project, compromised product quality, or a negative impact on future development costs, etc. Protecting the team from external pressure, offering stability, comfort and predictability are vital ingredients to maintaining a healthy team.

The architect should keep people comfortable in their environments and avoid disruptive activities

Rather than setting formal technical meetings, be in favor of directly approaching team members and discussing problems, whether inside the office at their desks, or going out to a cafe or having lunch together. It is a real fact, technical people do not like meetings. Therefore, try to reduce the number of meetings, and keep developers closer to environments where they feel comfortable (i.e. in front of their computers). Instead of asking a developer for future meeting availability, do it on the spot; have a direct and friendly conversation outside of a formal meeting.

In the case of remote teams, when team members are geo-distributed, things might differ slightly. One way to think about this, but still keep a developer in their comfort zone, is to treat every employee as a remote worker. This means never going into a conference room but prefering to have a video conference at each one’s desk, even if that means some of the people sitting right next to each other with headsets on.

The architect should be careful to make meetings less intrusive; sometimes too much disruptiveness is not good and technical people might lose focus and become inefficient. Besides that, some people are more productive in the mornings, usually during the first part of day, while for others this is totally the opposite. In either case, the architect must find the right moment and a good balance to not jeopardize working hours and keep people focused.

The architect must find room for mixing business functionalities with technical stories

Besides business functionalities, there might be also technical tasks, related to the internal quality of the product, which deserve to be prioritized. Some of them enable the development of business functionalities, directly visible to the customer, while others might not have an immediate tangible effect. Such examples could be related to refactoring or restructuring (to improve code readability, increase cohesion, reduce coupling), upgrading libraries or frameworks versions, enhancing logging format, adding tracing and monitoring, etc.

The architect must keep an equilibrium between adding new functionalities and the amount of required technical tasks. Trying to justify sprint by sprint, the need to prioritize them in front of the product owner, based on my experience, does not work smoothly. On top of that, it is not easy to always justify technical things to non-technical people. They might have a clue, but without a proper understanding of the necessity. The solution which worked for me (in already few projects now) was to agree with the product owner about a quota (or technical buffer) for technical tasks (something around 20% of the average team velocity). Within this quota, the architect, together with team members, has the freedom to decide what is worth it to be included in the next sprint, in order to keep a reasonable progress across technical tasks. This quota might not be rigorously followed on a regular basis, for example there could be sprints where it is not needed, however, it is important to reach consensus with the product owner and leverage this option, if necessary.

The architect should support developers to stay up-to-date with the latest technologies

The architect should discuss with the product owner and scrum master to set aside time for the team to study different technologies they are passionate about or to contribute to open source projects, even though these are not directly linked to their product. When people stay in sync with the latest technologies and are exposed to other ideas, it has a positive impact on the product. There are companies where such activities are highly encouraged and people take them for granted, while at other places it is very difficult to get business approval since there is always a demand to deliver business features. Methodologies like SAFe explicitly assign room for similar activities (one innovation sprint after every other four) which is a good step forward, however, in my opinion, this must be part of all product development teams, independent of the methodology.

A complementary approach is to regularly offer people access to top-notch technical conferences and facilitate access to specialized online courses or training platforms. Try to negotiate these benefits with the budget owner. Team members will feel appreciated and rewarded, and will stay on the project longer. Having knowledgeable people remain on the team leads to higher product quality.

Presentations on different topics, including internal communities of practice, where interested people are regularly meeting and discussing technical subjects, is a good way of spreading the knowledge and best practices, emphasizing human interactions and good collaboration.

Agile value: responding to change over following a plan

Sometimes people on the team might prefer to "Make it work now. Make it prettier later," at the risk of degrading internal quality and increasing the technical debt. Similar guidance may sound like, "*Do not allocate time for finding a proper solution at the moment"* or "*xclude known future use cases and focus only on current sprint requirements*." All of these make sense in the light of agile, however, they can contradict a good architectural standpoint. Underestimated, neglected, and wrong design decisions could become harder or even impossible to be rolled back afterwards, for either technological or economic reasons. Building software without an efficient plan, without a proper vision and realistic expectations is prone to failures. Writing code is costly, developers are not cheap, and throwing away pieces of code and constantly rewriting it causes more troubles and delays. A good architect should not make compromises which deliberately degrade the internal quality, unless the consequences are understood and approved by the business.

From my point of view, the architect should always consider future use cases that are on the critical path, since they might add constraints in the technical design. While an agile team may work in small sprints, the architect is in a position that requires looking ahead to make sure the progress is towards a larger goal. Things go easier when there is a better forecast among business functionalities and clear goals, since the architect could make more accurate architectural decisions and delay things that are unnecessary or unclear at the moment. Assessing future business specifications, measuring their impact in the current architecture, and communicating the risk to stakeholders is critical for the success of the product and one of the key responsibilities for the architect.

For example, I work on reviewing upfront business specifications that will be tackled in the upcoming sprints by the team (e.g. up to six sprints in advance, therefore maximum three months in total), paying attention to the ones with an architectural impact. This approach fits products where there is a high development demand and a consistent backlog, however for others that are predominantly in maintenance or support phases it is almost impossible to have a proper prediction, since things could fly in and out very irregularly.

Nevertheless, responding to change does not mean not having a plan at all, or always being in a position to redesign or re-architect the system. At some point, most aspects of the architecture become largely immutable, especially with older products. The architect should always have a plan, and provide guidance and vision to the team. However, the architectural plan must be flexible enough (i.e. ready to embrace changes) and able to promote future business requirements.

Agile value: customer collaboration over contract negotiation

In agile environments, it is more important to understand the customer's needs than what was negotiated in the contract. Developing software that meets customer needs requires regular collaboration all along the development lifecycle. In practice, this becomes possible at the end of every sprint when feedback is received and priorities might get shuffled, giving the opportunity to ensure maximum developed value very early on in the process.

All stakeholders become the customers for the architect

The architect has to deal with both external and internal requests coming from different stakeholders (e.g. clients, development team, product owner, scrum master, etc.). Despite all having a shared stake in the success of the system, typically they have different specific concerns that they wish the system to guarantee or optimize. These stakeholders become the customers for the architect. Rather than the idea that the customer is always the person who is paying for the software, an architect's customer is anyone with an interest in the realization of the system. Under this circumstance, the architect must find the right balance between business, technology, and people, in a feasible manner, and thereafter, is responsible for communicating this balance back to their customers and stakeholders, to maintain the system's conceptual integrity during development and to facilitate the transition from the model to production system.

An architect should master both communication and negotiation skills

Working collaboratively together is far more important than defining rules of interaction. For the architect, both communication and negotiation skills are extremely important. Up to this point, I've focused on what collaboration means for an architect, in relationship to other stakeholders. Nevertheless, negotiation is also a key ingredient, since the architect will always be in a position to discuss and reach agreements with the client, product owner, business analyst, scrum master or even with developers, on different technical solutions. All examples provided in regards to facilitating a sprint quota for technical tasks, collaborating with the team in order to find appropriate technical solutions, or reaching team consensus while defining code review policies or internal quality gates, do not come for free. As in real life, there are always negotiations and trade-offs.

Final thoughts

The architect should not religiously trust or follow any methodology (e.g. SCRUM, SAFe, etc), as they all have pros and cons, strengths and weaknesses, and are sometimes poorly applied in practice. All methodologies are leaky to some extent. The agile architect is one who does not stick to any schema or dogmatic role. The agile architect must be pragmatic, flexible, and intensively involved in the software development lifecycle process. They must possess good communication but also negotiation skills, continuously search for improvements, and offer guidance and vision to the team.

The Agile Manifesto values are not binaries. It is not only "working software", "individuals and interactions", "responding to change" or "customer collaboration", but rather a scale that an architect, in collaboration with other roles within the team, needs to properly adjust, whether more to the left or right, depending on stakeholders concerns, the development organization, the technical environment, and their expertise.

About the Author

Ionut Balosin is a Software Architect and Independent Technical Trainer. He is a regular speaker at software development conferences and meetups around the world, delivering presentations, training courses and workshops. For more details please check his website.

6