How We Adapted in a World of Volatility, Uncertainty, Complexity, Ambiguity (VUCA)

ADAM CETNEROWSKI, Intel

Over the past years my engineering team evolved multiple times to better deliver our changing products in an evolving ecosystem. In this paper I highlight factors that prompted the change and document the decisions we made—those that were successful and ones that ended up being mistakes.

1. INTRODUCTION

In this paper I'd like to present key turning points in the way our team practiced Agile over the past six years. Each turning point was prompted by one or more changes in the ecosystem we work with in—both changes in the organizational structure around us as well as the products we were asked to support. For the purpose of this paper I have divided the covered timespan into "eras" and much like historical era, each one starts with a major change to which we needed to adapt. I share our learnings at the end of each era and summarize at the end.

I started my own Agile journey over ten years ago. When I first learned about Agile, I thought it was a good idea that would never work in the context of Intel. As I learned more about the various facets, I found myself becoming an evangelist. By the time the story starts, I was recognized as a trainer and coach, but my role had never been full time in these aspects.

Background

We are a development team within a wider organization responsible for delivering and maintaining software for Intel Graphics (both integrated and discrete). Our team specifically is responsible for the Level Zero and OpenCL drivers available on Microsoft Windows and Linux operating systems[1]. Our direct team is in Gdańsk (Poland), but we have daily interactions with teams across the world.

2. THROUGH THE AGES

For the purpose of this experience report I divided the years I'm covering into eras like human history. Each transition was prompted by a major change in the environment we worked in pushing us to adapt.

2.1 Classical Era

I joined my current team mid 2014 in the dual role of Agile Coach and Program Manager (PM). From the organization's standpoint Agile was an experiment and the PM piece paid for the coaching part. It was a challenge, but one I was willing to take.

At that time, the direct organization (VPG Compute) I would be able to influence was about 100 engineers. It was spread across multiple offices located primarily in Poland and on the US West Coast. We had multiple teams that were split across multiple aspects:

- Functional—developers working implementing the driver, validation engineers creating and executing test content
- Product lifecycle—pathfinding, IP implementation, customer enabling, maintenance
- Operating Systems—MS Windows, Linux, Android
- Features—runtime, compiler, OpenCL, Renderscript

This created a complex environment with many dependencies. To better spend my energy, I decided to focus on the four development teams that I was co-located with. In my opinion, the best way to start engaging

[1] You can learn more about our product on our GitHub page: https://github.com/intel/compute-runtime

Adam Cetnerowski, c/o Intel Technology Poland, ul. Slowackiego 173, 80-298 Gdansk, Poland; adam.cetnerowski@intel.com

with a new team is with a retrospective. It allows the coach to understand the team's situation, their problems and vision, and to simply get to know each other better. As is typical in these situations, the key issues were around visualizing work and communication. To address these, I suggested using Kanban to manage & visualize the work supplemented by daily standups ("walk the board"), while maintaining a regular cadence of retrospectives.

I had expected that my main challenge would be juggling the orthogonal roles of coach and program manager. This turned out to be doable—with the right scheduling and support from my management chain, I was able to find time for both. And I could keep the viewpoints separated by clearly indicating in which role I was acting at a given time.

What was most difficult was the different ways the various teams approached the ideas I was presenting—from cautious acceptance to direct naysaying. The Android team—with which I was closest on account of my PM duties—saw the most changes and most eager adoption of both the methods and then the mindset of Agile. The biggest change was pulling in validation engineers to be a regular participant of all the rituals, which was uncommon for the organization.

On the other end of the spectrum, I found the Windows team, which was very established in its ways. After trying the same approaches that I used with the other teams (different retrospective exercises, adjusting ceremonies, speaking with the team and individually with the team members), I decided to "cut them loose" to devote more time and energy to the responsive teams.

The other two teams had their own quirks, which we worked through. The Linux team first tried more "techy" approaches—using tools such as IRC in place of daily standups, but based on evidence from its sister team, got convinced to meet face to face. The compiler team would grumble at the rituals, but found value in regular meetings, which would evolve into whiteboarding sessions and motivated me to find creative exercises for their retrospectives.

Toward the end of this "era," Android work started to dry up and Linux work to pick up. To react to this, we decided to merge the Linux and Android teams, to prioritize work across the domains. What surprised me was how the team self-organized across this change and fought to have a structure that worked for them (one big development and validation team) and how they decided to keep the Agile practices adopted earlier.

What I learned in this period:

- Every team is different. You can start working with a new team in the same way but must adapt and pivot to meet their specific needs and quirks.
- Apply your energy, where it brings the most good. This is both true for coaching—if a team resists strongly, let them be. If a product is waning, have the team work on something of greater value.
- When first introducing a practice, agree on a timebox for testing it (e.g. four weeks) and "early exit" criteria. It makes it easier to accept the experiment and give it a fair shot.
- Good practices will survive reorganizations and diffuse across sister teams.

2.2 The Middle Ages

The year 2016 saw a major upheaval caused by significant reduction of the workforce at Intel. This overlapped with the graphics division getting a new leader and a major reorganization. Also, our team got shrunk and (from an organizational standpoint) rolled up with our 3D driver teams. While this caused anxiety in our team (new bosses at three different levels), reduction of bandwidth without a significant reduction of scope, it also yielded some new opportunities.

After the reorganization all the engineers working on the compute runtime would be in Gdańsk. (The compiler team merged with its sister compiler teams). We decided to follow the good practice of having one team deliver the same driver across two Operating Systems (OS), to build on the learnings from the previous year. We would still have specialists for both OSes (Linux and Windows), but the expectation was to be able to work cross-functionally. With smaller teams it was possible to get closer with the validation team (also co-located) and build the "one team" mentality.

We also addressed one of the biggest impediments mentioned by engineers—meetings. For local meetings we did a mental experiment—trying to picture a world without any meetings, except for 1:1 between a manager and direct report (used for career discussions and sundry administration). We also saw a need to have a weekly backlog grooming (to discuss new work items and keep a short TODO list) and the ever-important retrospectives. Since we foresaw ad-hoc, topic driven one-off meetings would be necessary, we reserved a two-hour block per week for these meetings (this was a promise that we would spend no more than two hours per week and made sure everyone and a big room would be available). Within a month we identified

one more need—a monthly "all-hands" meeting to provide updates (both on product direction and of administrative nature) and as a chance to ask questions, debate items not covered elsewhere. These did fit in the two-hour reserved space, so they weren't an additional burden.

For meetings with other teams, especially in other time zones, we did an honest assessment of the value of the meeting and communicated that we would not attend in person. Where necessary we would provide and receive updates asynchronously (email, shared documents).

On a personal note, I was offered the opportunity to become a people manager. After the changes, me and my manager would support 18 engineers. In the spirit of a cross-functional team, we clearly indicated that we would set direction together and that who your manager was on paper did not dictate your scope of work or your career opportunities.

These changes would not have been possible if we did not have another ace up our sleeve. For the past year, a few engineers were working on a side project. A new runtime (codenamed NEO) that was written using improved techniques (modern C++, unit tests) with the upfront assumptions of being cross-OS, scaling across multiple generations of hardware, and easy to opensource. These founding features meant that it was easy to ramp up new developers, easy to make changes, easy to contribute as a cross-functional team.

Big changes did not come without a cost. We were combining two teams that had different ways of organizing their work and asking people to move strongly out of their comfort zone (new OS, new driver, new ways of coding). Dividing time between work on the new driver and doing maintenance on the old one (still in production) was also a concern.

Additionally, I could not comfortably act in the role of a coach, since I was their direct manager. In hindsight it's hard to tell how much of it was my hesitation—both not to abuse my new position, especially in relation to the engineers who previously rejected my coaching attempts—and how much it was the team's difficulty in assessing what I was saying as a manager versus coach versus equal participant. The engineers were also reluctant to give negative feedback about their colleagues' actions.

We tried a Kanban system with two major swim lanes (for each product) and the idea that developers would interleave tasks (finish a bug on the old driver, then implement a feature on the new one). We soon saw tasks stalling (or maybe they were just big tasks that took longer than expected), developers gaming the system to take more interesting tasks, important tasks stagnating in the backlog without a good way to move them forward, and trouble with understanding progress.

We also had a hard time with finding ways to synchronize in the team. Daily standups did not work out—partially due to the size of the team, partially as the people who previously did not use them were not willing to give them a fair chance. Retrospectives also kept turning into arguments between vocal minorities and silent majorities.

We realized we had a big team, with different skillsets, different "chemistry", and without clear focus. If we could get the team members to form cohesive teams that would take responsibility for the team's work, not just the individual deliverables, things should improve. And that's what we set out to do.

What I learned in this period:

- Seize the opportunity given. Even a negative change can let you make things more "your way" and provide a good excuse.
- Building a cohesive team out of many people is hard. It requires time that may not be available. Find ways around it. Sometimes, it's enough to have a common goal. In our case, we had to dig deeper.

2.3 Renaissance

Having identified the negative behaviors, we decided to list the behaviors we wanted to promote and build a structure to support that. The positive behaviors we listed:

- Joint (i.e. team) responsibility for delivering work items[2]
- Completing work items in the priority order
- Transparency about progress and completion
- Some predictability about completion trends
- A good mix of getting things done versus spreading knowledge and skills across team members

[2] I prefer "work item" as these include not only Stories (in the Scrum meaning), but also bugs, infrastructure work, internal improvements, etc.

What we wanted to avoid:

- Micromanagement, including too frequent status updates
- Leaving individuals with a problem ("that's not my task")
- Uncontrolled interruptions
- Building silos

This set of requirements pointed us to Scrum. With the number of engineers on the team, we needed to have multiple Scrum teams that would plan together, do their day-to-day work independently, and synchronize in cross-team retrospectives. We had of course discovered Large Scale Scrum (LeSS).[3]

Before adopting this way of working, we wanted to have buy-in from the team, so we gathered the developers that had demonstrated the most interest in the way we work, presented the concept, and asked for feedback. After a discussion without us (i.e. the managers), the proposal was accepted with some tweaks. This is what was agreed upon:

- Three teams working in three-week sprints (two weeks was deemed too short for a significant number of tasks).
- We would have a common planning session, separate sprint reviews, separate retrospectives, and a cross-team retrospective with delegates.
- Day to day work would be governed by the team, updates on tasks in progress would be expected in the tracking tools once per week.

It's interesting that in the beginning the teams were not sold on daily standups (or their equivalent), but after the first sprint it was obvious that the team that decided to have them was the most successful. This practice would be adapted by all the teams and although the format and content would vary, it continues to this day. A need to communicate between teams also came up and got resolved by a team-driven weekly meeting to discuss technical issues.

To avoid building silos around the teams, we agreed that every three sprints (~9 weeks), we would rotate team membership with the goal of making sure everyone got to work with everyone. This highlighted that we were one team and allowed for an improved dissemination of skills and knowledge. It did uncover some incompatibilities between various pairs of people and some of the configurations created lopsided teams (when a mix of people lacked enough technical depth or leadership), overall, it was a good experiment to conduct.

One thing to notice is the lack of a Scrum Master role. The Product Owner role resided jointly with the engineering managers (me and my direct superior) and we assumed we could handle the Scrum Master role as well. In hindsight this was not the optimal decision with the biggest impact being on retrospectives, which were declining in quality—few important issues were brought up, there was no conclusion on follow-up actions or volunteers to implement them, people were less engaged in the meeting. People stopped coming or would declare that there was nothing to discuss or we would push them out due to "urgent work". Finally, we stopped scheduling them.

One late addition to our processes was the Joker (or "Wild card") concept. We found that we still had interruptions—external (critical customer bugs) and internal (infrastructure issues)—and no good way of handling things that could not wait for the next sprint. To handle these, we would designate one team per sprint to handle these interrupts (in exchange they would plan less work up front). Initially a rotating role (each team would take one sprint in round-robin fashion), the teams switched to a model, where being the Joker was part of the planning process (a virtual task in the backlog taken by one of the teams as part of their available bandwidth).

I'd like to highlight two of the most positive aspects from this time period. One was the positive energy around the planning sessions. We would include all the team members, layout the work in priority on a big conference table, discuss the work that was planned and let teams choose and haggle of what needed to be done. This really reinforced the one team mentality and made sure priorities and implementation directions were clear. The other was around teams taking joined ownership of the work. The greatest example I remember was when an engineer got hit by a hard regression (an important change that caused a major issue

[3] At that time, I had not known about LeSS and only discovered the term a few months later when exploring different methods of scaling Agile

in another part of the product), someone on his team volunteered to debug it, so that the engineer could move forward with the next task.

What I learned in this period:

- Teams will gladly adopt even revolutionary changes, if they have significant input in how the organization will look after the change.
- Having someone, who can focus on facilitating learning in the team (e.g. dedicated Scrum Master) outside the normal power structure is a great catalyst (this learning would be proved by later events).
- It's easier for small teams to care for each other.
- Getting Scrum right is hard!

2.4 Industrial

As we settled into our routine of regular sprints, we observed some negative symptoms—work planned for a sprint would not be finished and would "roll over" into the next one, sometimes multiple sprints in a row. The list of things that "had" to be included in the next sprint continued to grow sprint over sprint causing a dangerous feedback loop. We tried to figure out a way to fix these issues but could not come up with a good solution.

What really pushed us to reconsider changing the way we work was Intel's decision to enter the discrete graphics market (Xe Graphics). This decision added a new stream of complex technical work that had an immediate impact on the team, including:

- Implementing support for parts of the new hardware required exploratory work, very close coordination with hardware and other software teams, using simulators with a slow feedback loop (due to the workload execution time)—all of which made scheduling work in two to three week increments with an intention to complete a feature not viable.
- A lot of these work items were tracked at the organizational level requiring new forms of reporting, coordination around planning and integration of large hardware and software stacks.
- Multiple senior team members had to spend multiple weeks in another location and time zone focusing their attention on collaboration with those teams rather than within our team.
- Knowledge sharing was delayed due to the volatile and complex nature of the new IP as it solidified.

The direction we chose was to go back to Kanban, while maintaining the structure of (sub)teams. This allowed the teams to pull the work as they had bandwidth or as priorities changed fluidly (e.g. hardware was not ready for a feature, changing implementation order would make coding easier). We still wanted to keep a "heartbeat", so we rebranded sprint reviews to checkpoints. Now we would meet every two weeks to look at what was finished and in progress. This removed the pressure of having to finish a work item (that could be longer than the sprint) to meet an arbitrary deadline. At the same time, we would look at items blocked by dependencies and impediments and those that started to exhibit scope creep. These checkpoints also doubled as a chance to provide guidance on the next few weeks and made sure directions were aligned.

This high-pressure situation also highlighted that not everyone on the team was suited to this exploratory, intense work requiring very quick ramp of technical knowledge. They shined more in areas such as debugging and drifted toward those types of tasks. We would soon use this insight to adjust how we worked.

What I learned in this period:

- It's not a failure to abandon a practice, because it no longer suits you. For a while, I felt bad that we had to give up on Scrum, until I saw how much more efficient the team became both around execution, but also short-term planning and self-organizing around the work
- Play to the strengths of your team members in selecting what they work on—sometimes this may require reorganizing the team or having a team member move to another team
- When switching methods, it's still viable to keep what worked—like keeping the concept of a "sprint review", but applying it to Kanban

2.5 Modern

As 2018 came to a close, we saw internal and external forces pushing us to the same solution. Internally, we recognized people's preference toward specific type of work. Looking at the work coming, we could clearly see two streams of work—customer issues involving mature platforms and concerns around opensource (adoption by distributions, tool chain differences); development of support for new hardware and software feature on Intel's evolving portfolio of Xe products. We clarified the primary charter of each team—one would

focus on incoming bugs and customer queries, while the other two would focus on new features (but help the first team with knowledge sharing and hands-on support as needed) and moved some team members to better suit their strengths and synergies with peers.

Our (and here I include both managers as well as senior team members) decision proved to be fortunate as in early 2019 Intel announced the OneAPI initiative. Our team would be significantly impacted, as a key ingredient would be a new compute driver written to a new, open specification—Level Zero. Building a new driver from scratch to a specification that was being developed at the same time was a challenge. Meeting the tight deadline requested by senior leadership to prove the viability of the concept only magnified the effort required. To take it on, we changed the charter of one of the feature teams to focus 100% on Level Zero development.

We understood that even a non-constraining system as Kanban would be too much, so the team adapted a "startup" mentality. A clear goal was set for each quarter (the first milestone would be a functioning ray tracer over the new API) with the teams having a lot of leeway on the order of tasks, experimentation, and prototyping. We allowed looser standards in code that we understood would be discarded while rapidly prototyping. At the same time, the thought and effort we put into developing NEO thus far served the new effort well—allowing good reuse of code and development infrastructure. Unlike developing for new hardware, at an early stage Level Zero was a pure software effort, meaning that there were less hard dependencies. These were mostly definitions of interfaces, driver behavior, and expected customer usage models.

To better handle the workload, we paired up with an experienced (though not in graphics) team on the West Coast. This would be the first time in multiple years, when we would do day-to-day joint development across time zones (and for the newer members of our team—their first experience). This came with all the struggles that come at a time like this—different working habits, organization of work, culture. What helped us get through (and still does when disagreements arise) was foremost a common understanding of our goal—deliver a great user experience with the Level Zero driver on Intel GPUs. It also helped that we deferred convergence on anything that didn't require it (minutiae of task management, Scrum vs Kanban, code review practices) as opposed to common items (e.g. coding style, common code repositories and build infrastructure).

One interesting thing that happened during this time involves a fellow manager. He had divested himself of his previous team (citing burn-out and rebalancing his work / life) and had about a month "on the clock" before setting out on a sabbatical prior to coming back to a new full-time position. He proposed that he would audit (in the academic sense) our team, to provide a fresh viewpoint (and maybe to join us after the sabbatical). As both an experienced manager and someone outside the team's power structure, he was able to make keen observations and restarted something we were sorely lacking (as it turned out)—retrospectives. Even though he chose not to return to our team ("looking for bigger challenges"), his legacy is permanent retrospectives run by another manager from outside of our immediate team.

What I learned in this period:

- Cross-functional teams are great as they maximize value and improve knowledge flow. Yet, when faced with multiple streams of work, the cost of task switching may negate this and having dedicated teams that still know how to work together (and think of themselves as "one team") improves throughput and maximizes peoples' strengths.
- Having someone external to the team come in and facilitate change can go a long way with even a little investment—be it a coach, a scrum master, or even just a friend.
- When two teams set in their ways start working together, find the minimum number of parameters that need to be agreed upon. Focus on delivering or achieving something together as soon as possible to share a sense of victory. If you need to converge on more practices, wait until you have built up trust and checked that it's still needed.

2.6 Future

Early 2020 Intel unveiled the details for OneAPI including the (still evolving) specification for Level Zero. This was a signal for us to move our development to opensource and start including a binary release of the driver alongside OpenCL. It was great to see the results of a year's work available to the public.

This also marked a transition for developers contributing to Level Zero as they adopted to the more rigorous processes that applied to a driver that would no longer be considered a prototype, but well on its way to becoming a final product. To my surprise, this shift in processes went overall well—possibly a testament to the growing trust and respect between our teams, including in the third sister team that joined us, this time from India.

Looking at the rest of the year, we see a few challenges:

- Bringing up new hardware platforms
- Achieving production quality with Level Zero
- Closing out the feature delta between OpenCL and Level Zero
- Finding bandwidth to focus more on proactively seeking performance optimizations

Will these require new ways of working, a different organization of our teams, renegotiation of charters across teams (both in Gdansk and across our sister teams)? We don't know yet, but we have learned to ask these questions.

What I learned in this period:

- Nothing yet, but I'm looking forward to it.

3. WHAT WE LEARNED

Since I summarized my learnings at the end of each, I would like to reflect on the overall experiences I had and the high-level ideas I formulated:

- Don't assume that the way you work (team structure, framework, cadence) that suits the team today will be the best solution in the future. Over time expect to adjust, pivot, and even backtrack.
- These changes can be driven by multiple factors, including team composition, the wider organization, and the nature of product or service you're delivering.
- Engage your team in these decisions. Even a coach with a lot of experience needs buy-in from the team and the sooner they are involved and the greater their opportunity to provide input (though not everyone does), the easier the sell.

4. ACKNOWLEDGEMENTS

I'd like to start off by thanking Piotr Rozenfeld, who is not only my manager, but also my partner in brainstorming and introducing the changes I described in this paper. A big thank you to all current and former team members, who took in these changes sometimes with optimism and at other times with a lot of hesitation, but always voicing their opinions. Last, but not least, a big shout out to Jutta Eckstein—I was overjoyed when I learned I would have a friendly face as my shepherd, and I know that this paper would be much worse without your valuable input.