Architecting a Globally Distributed Software Development Organization for Continual Development

ANASTAS STOYANOVSKY, IBM Watson[1]

As an agile development project grows and distributes over different places and timezones, communication overhead increases. If its communication systems are not carefully designed, friction and miscommunication can cripple it to the point of being less effective than a single team working alone. Well-synchronized distributed development can derive benefits such as working hour continuity, diverse skill sets, and varied perspectives. On the other hand, with significantly differing timezones, architectural decisions often need to be made without being able to include all stakeholders or even core developers, and often without including those most experienced with a particular subsystem or component.

This report describes a case study of a working global development model at IBM Watson that was founded for a greenfield project and grew to span seven sites throughout both hemispheres.

1. INTRODUCTION

In response to market demand in 2018 and 2019, IBM Watson prioritized offering on-premise versions of its cloud products. If not tackled, this decision would have brought Watson Discovery, a cloud offering developed by the IBM Data & AI organization, into competition with Watson Explorer, an on-premise offering developed by the IBM Analytics organization. Therefore, in April 2019, the executive leadership team decided to consolidate the Discovery and Explorer offerings into a new, multicloud offering, and eventually release it as Watson Discovery v2. This brought United States and Tokyo based teams into a single organization. The teams were given an ambitious three month deadline to release the new consolidated product.

Due to the diametric opposition of working hours across hemispheres, opportunities for real-time cross team communication were minimal. Communication and alignment issues across teams could have caused the overall global organization to be slower than a single team working on its own. In order to maximize the chances of success, we intentionally adopted a collection of development practices and philosophies; some were chosen at the beginning of the project and others emerged over time. This report covers the philosophies and practices employed to create a global development model that succeeded past its initial goals and grew to five teams across the globe.

2. CASE STUDY

2.1 Preparation

I had been in a technical lead role for the query-time information retrieval functionality in Watson Discovery for several years. Upon learning that the US and Japan organizations were likely going to merge, I performed a literature search that yielded consistently recommended best practices for globally distributed development: initial face to face meeting in order to establish trust between the involved teams, daily handoffs of work items over phone or video conference, shared infrastructure (artifact management, source code repositories, continuous integration infrastructure, &c), and shared expectations for development practices (Treinen & Miller-Frost, 2006) (Verner et al., 2014).

I presented a synthesis of this literature review to my management. As that literature repeatedly emphasized the importance of an initial face to face meeting when forming a globally distributed organizational structure, I emphasized the same; the importance of this investment would be illustrated multiple times throughout development, as will be mentioned below. The overall list of recommendations from the literature were a starting point for how I would try to foresee, prepare for, and prevent challenges for my team.

[1] Affiliation at the time of the work reported; affiliation at the time of writing is Kiavi Inc.

2.2 Project Launch

Upon the conclusion of several weeks of executive discussions in April 2019, myself and three other lead engineers from Discovery were asked to fly to Tokyo in order to kick off product consolidation. We were to spend a week working on-site with Tokyo teams in order to establish working relationships and to form an initial technical plan for product consolidation. While there, we had a daily conference call with the US team at approximately 9pm Tokyo time.

Throughout the course of a week of design and planning with the onsite teams and nightly conference calls, we collectively made a series of key high level technical decisions. The use of architecture decision records (ADRs) (Nygard, 2011) was already an established practice on our team (Keeling & Runde, 2017) (Keeling & Runde, 2018), and so we used that format to propose, record, and communicate individual technical decisions. As opportunities for real time communication were few, this took the form of composing an ADR during the Tokyo day and submitting it as a GitHub pull request (PR) for asynchronous review and discussion from the US team.

This initial practice set a precedent of what would become a main decision making mechanism across the two teams. My team in the US and our Tokyo counterparts from the Explorer product would collaborate on greenfield development of real time query functionality for the new product. Another US team that had been focused on machine learning model training infrastructure would port those services onto the new product's platform, and the two sets of functionality would have well-defined integration points.

The literature on global development models repeatedly highlighted the importance of having shared expectations for development processes from the beginning of a collaboration. For that reason, towards the end of our week on-site, I took the opportunity to obtain buy-in on a set of development practices to be shared across our teams. I synthesized existing practices on my team with those from the same literature review into a proposal:

- *"Atomic issues"*: task management would aim to break work down into the smallest reasonable work units in order to make handoff between teams as easy as possible. The goal was for each work item in our shared issue tracker to be able to be completed in one work day by one team, allowing the other team to pick up the next sequential work item. (In practice, this was rarely accomplished, though it did set a precedent for breaking work down into smaller items.)
- *Daily handoffs*: At the end of a team member's work day, any in progress work would be submitted as a PR, with the intention of being continued by a colleague in the other hemisphere. A daily video conference call would be had at the end of the day in US Eastern time / Tokyo morning time in order to coordinate and communicate any context not obvious from code changes.
- *ADRs*: Any technical or design decision that required either investigation or discussion would be documented as an ADR, subject to approval by the other team.
- *Diagramming*: We would make use of visual software architecture modeling whenever possible in order minimize the risk of miscommunication. In order to decrease the effort involved in maintaining the practice, I suggested two specific diagram types: sequence diagrams and component/connector diagrams.
- *Functional testing*: Every new feature would be covered by component level functional testing.

I presented this proposal as a starting point for our new global collaboration, with the stated intention of adjusting our practices as necessary; under this understanding, both teams agreed to adopt them and course correct over time.

A Minimal Set of Architecture Diagram Types

Software architecture modeling was already a common practice on my team and had well demonstrated its value for communication and alignment. In our teams' new global development model, I expected that miscommunication around software architecture would be more than twice as expensive as for a single team due to the at most twice daily feedback loop between teams.

I had several goals for establishing the role of software architecture modeling as a shared practice:

1. maintain a shared understanding of the system under development;
2. minimize the likelihood of miscommunication of design intent, especially between teams; and
3. incur the minimal time and effort burden in order to maximize adoption as a daily practice.

I proposed two diagram types as a necessary but sufficient set: component/connector and sequence diagrams. More sophisticated modeling approaches such as C4 (Brown, n.d.) or 4+1 (Kruchten, Nov 1995) were selected against due to goal (3), and the use of diagrams in ADRs was strongly and routinely encouraged towards goals (1) and (2). The challenge of cross cultural communication is outside the scope of this paper but did factor into emphasizing visual communication.

The combination of these two diagram types was effective at achieving the above goals. As the initial codebase was being created and organized, component/connector diagrams ensured shared understanding of what software components would exist, and sequence diagrams ensured shared understanding of how those components would interact. Taken together, they helped minimize opportunity for mistakes during coordination and planning of new development.

Distributing Design Authority using ADRs

In this new, globally distributed organization, having a single architect would be a bottleneck that would prevent the realization of the full benefits of global development. Decision making would have been localized to one team and prevented the other team from contributing as much as it could have. Furthermore, the natural course of every team member's work would entail research, critical thinking, and impactful choices, and so each individual should have license to make decisions as they need to be made.

In order to effectively distribute design authority, we needed a mechanism for inclusivity during decision-making, as well as for communication of decisions that had been made. This need had already been well served within my team by ADRs for years (*Distribute Design Authority With Architecture Decision Records*, n.d.), which led me to seek to expand the practice across both teams. An in depth discussion of the various benefits of this practice is outside the scope of this report, but a brief list would include:

- They concisely capture decisions made that required research and/or discussion.
- Once recorded (with context), those decisions can be communicated asynchronously and referenced in the future.
- The vehicle of submitting a PR containing an ADR allows for evidence-based disagreement via the Context and Consequences sections.
- The Consequences section encourages critical thinking in all team members and as a team habit.
- They are a vehicle for maintaining shared understanding across all team members - the "surprise factor" of technical decisions is minimized.
- Everyone is held to design decisions that have already been made unless a strong technical case is made to amend those decisions.
- Individual autonomy becomes the norm, which can contribute heavily to a high performing team.

My own team's usage of ADRs had intially taken weeks to months to initially become established, and a similar timeframe for adoption happened with our Tokyo-based colleagues. As time went on and as the above-described benefits became more evident, the practice did become more routine. I wonder to what degree a language barrier may have played a role, but I would be surprised if it were negligible.

2.3 Forming and Storming: The First Weeks

For the first two weeks, we had a daily 9pm US Eastern Time video conference with both teams lasting 30-60 minutes. Aside from this being a tool for coordination and planning, I also had the goal of the teams building up to the same level of trust that we had built when going on-site in Tokyo. Unfortunately, this did not seem to succeed as much as I would have liked, bringing further support for the importance of an initial face to face meeting between teams.

The daily conference call was an exhausting practice for everyone involved; after two weeks, we moved to a weekly cadence.

Development Practices

The first weeks of collaborative development revealed the extent to which the two teams' development practices differed, almost entirely along agile vs. waterfall lines which will not be restated here. I attempted to take advantage of the global video calls to steward the combined team to a base of shared practices; unfortunately, this approach was insufficient. My team repeatedly expressed frustration at the contrast in development processes, as well as concerns that a lack of agile and/or cloud development practices would put the project at risk. To address these, I encouraged the team to adopt the approach of "demonstrate value".

Adoption through Demonstration of Value – My approach to resolving the tension over development practices was to demonstrate value in practices by "doing instead of talking". A characteristic example was our advocacy for the adoption of component level functional testing: contributed functional tests were able to identify software bugs missed by unit testing, such as an API endpoint failing to serialize a response when called by an external client. I also encouraged my team to take this opportunity to reevaluate whether the practices we considered essential were, indeed, essential; for example, one takeaway from doing so myself was to doubt the value of strict code formatting (unless automated).

The "demonstrate value" approach required time and patience. In the same way that it took time for the value of this practice to become evident on our team as we transitioned from waterfall development to agile cloud development, it took time to successfully advocate for its adoption on this new project. There were weeks during which test writing largely displaced implementation work on my team, which led to low morale that required encouragement to work through. After repeatedly demonstrating situations in which this testing caught bugs that were missed in unit testing, the practice did become common to both teams, justifying the demonstration-based advocacy effort.

Communication and Collaboration

"20x5 Development" – The initial intention was for each work item to be worked on by people from both sides of the ocean as we tried to "pair" across teams, with a hand-off at the end of each timezone's work day. To describe the practice, the term "20x5" development was coined by an executive. In practice, due both to the aggressive deadline and to variation in individuals' working hours, this was effectively "24x5" development.

The idea of every single work item being worked on by both teams was quickly found to be in tension with encouraging individual autonomy. Expecting every work item to be collaborated on in this way was found to not come naturally and to thus be an overly idealistic mandate. As a result, it was abandoned as a goal. While it became routine for a pair of developers across the two teams to collaborate effectively on some high priority feature, it also became routine for smaller work items to be completed by a single individual. This led to strong design cohesion in high priority functionality with some siloing in other essential features. The initial idealism did serve, however, as one motivation for establishing a concrete, effective handoff mechanism: the "TCES" format.

Handoff Mechanism – The "TCES" Format - The initial handoff mechanism was to leave a comment on an active work item on our shared tracker, containing two sections: *What has been done* and *What*

needs to be done. As development ramped up, this proved to be prohibitively time consuming during daily standup as we had to open each individual item to check for updates. During a weekly global conference call, the teams agreed to my suggestion to transform the asynchronous "YTB" update format (Radigan, n.d.) into something suitable for continual, global development in which there is no common "yesterday" or "today". We settled on a "TCES" format to use to post to Slack at the end of the work day:

- *Today*: A summary of what was done during one's work day, with a link to the relevant work tracker item(s).
- *Current problems*: An identification of problems remaining to be solved.
- *External concerns*: Any dependencies on external teams or factors.
- *Stakeholders*: A list of people who are working on the same or related work item(s), using "@" tagging in Slack to notify them.

It is difficult to overstate the effectiveness of this simple change in communication and coordination. I believe it was crucial to the project's success for the following reasons:

- The *Stakeholders* notification mechanism transformed information propagation from a pull-based model to a push based one - the relevant people receive notifications rather than sifting through a work tracker for new information.
- The declarative *Current Problems* transformed the *What needs to be done* section from prescriptive to descriptive. Whereas before it could feel that one person was prescribing what should be done next, it was now a simple statement of what problems remain to be solved, implicitly trusting others to work on the problem and solve it on their own. This fostered trust and collaboration.
- The *External Concerns* section generalized *Blockers* in the YTB format. In contrast with only raising blocking issues, this became a routine point of collaboration with other teams throughout the organization and likely replaced a significant number of hours spent in meetings.
- Reading the updates from each team member on the other side of the globe in the morning replaced a 30+ minute daily morning team-wide coordination and planning meeting with a few minutes of individual, asynchronous reading.

2.4 Norming and Performing: Towards the Finish Line

After the formative weeks, development settled into a very effective and efficient mode:

- As the value of some practices and insignificance of others was demonstrated to everyone, both teams settled on a largely common way of working. Code review on incremental PRs became the norm; together with CI/CD, a functional test suite, and the aspects below described, the software development lifecycle became very rapid.
- ADRs were routinely used to effectively capture a proposed decision based on investigation and discussion on one team, receive/incorporate feedback from the other, and then finalize/commit. Trust and understanding improved to the point of being able to gauge which decisions should be raised for inter-team discussion, versus which could be acted on immediately without issue.
- Routine architecture modeling helped us to maintain shared understanding of the evolving system across both teams. When a new component was found necessary, assessing its role with respect to other components, updating the diagrams, and implementing it involved little controversy.
- The TCES communication format allowed for well-coordinated global collaboration and for low overhead in overall project management.

There was one remaining practice that emerged organically as development proceeded and that I refer to as "inverting the testing pyramid", which merits its own discussion.

Inverting the Testing Pyramid

There were still differences across teams in coding style, unit test organization/scope, package structure, and other internals. Our emergent focus on functional testing as the golden standard made those differences largely inconsequential. The underlying philosophy was that if the inputs and outputs of the application were *ceteris paribus* correct (e.g. using test mocks), then a minimum acceptable confidence in software correctness could be achieved.

This inversion of the testing pyramid emerged organically, and its displacement of unit testing as the predominant testing paradigm allowed for rapid and routine large-scale refactoring. It is possible that the confidence in the functional testing allowed team members to feel comfortable relaxing usual standards of unit testing. An additional contributing factor might have been that different methods of organization of unit tests between the two teams might have led to frustration, perhaps leading some to simply not contribute to unit tests organized in an unfamiliar way.

However it evolved, this approach significantly lowered the barrier to and cost of large refactors, although unit testing did remain an essential tool for deep internals. Generally, as long as the functional test suite was both comprehensive and passing, rapid iteration was enabled, accelerating the pace of development.

2.5 Release

In the end, the US and Tokyo teams functioned as one large, globally distributed team. All required functionality was completed and thoroughly covered by functional testing with weeks to spare, leaving enough time to build a comprehensive end-to-end test suite on top of a framework developed by another, dedicated team in the organization. We were also able to dedicate time to conduct performance tests to demonstrate that performance targets were either met or significantly exceeded. The end-to-end tests were able to verify the integration of our deliverables with the ML lifecycle infrastructure that had been ported from the cloud Discovery services by another team. This result established an additional standard of always requiring new functionality to be accompanied by end-to-end testing.

The complete delivery of all required functionality, performance testing, end to end testing, &c was celebrated, but the cost of the achievement is noteworthy.

The Cost of Success

Although the release was successful by all of our measures, it was accompanied by fatigue, if not burnout. Aside from the pressure of the aggressive deadline, maintaining this development model required constant vigilance from all team members. The demands on each team member were those of normal development, combined with: handing work off at the end of the day, reviewing another team's output every morning, and often having early morning and late night conversations with the other team. Although I am confident that all team members recognized and appreciated the project's success on an intellectual level, at least my team and our executive leadership agreed that this model was simply too exhausting to be sustainable long-term.

After the development model had succeeded in its goals, it was no longer necessary to develop a single codebase across both hemispheres for effectively 24 hours a day. A foundation had been laid and different teams could now specialize on different functionality going forward.

Indeed, that success in laying the foundation of a production codebase led to three more teams joining the information retrieval organization. This new organizational scale brought new challenges.

2.6 Growing to Five Teams

The initial release accomplished by the US and Tokyo teams was a success by all measures: functionality, performance, stability, test coverage, global relations, documentation, etc. Building upon this foundation, the information retrieval organization grew to five teams across seven sites:

- the initial US based team focused on incorporating more sophisticated query-time information retrieval functionality,
- the initial Tokyo based team switched priorities to incorporating a set of data mining functionality,
- one new team was tasked with adding the ability for end users to hand curate certain behaviors,
- a second new team was brought in to incorporate an IBM Research asset for addressing a novel use case; and,
- a third new team was to adapt another legacy product to address an experimental use case.

While I had previously adopted a democratic approach for spreading practices to new teams (Stoyanovsky & Chaparro 2019), the success of the project so far justified wholesale adoption via a `CONTRIBUTING.md` (*Setting Guidelines for Repository Contributors*, n.d.). In other words, the development practices developed for the initial release were adopted wholesale by all five teams, and were expected to be maintained. In the case that some new deliverable again required global development, the institutional knowledge for how to do so had been created; until then, maintaining the established practices would keep the organization ready to execute. With these structures in place, the foremost remaining challenge was to effectively scale development of this new production codebase to these five teams.

Fast-Forwarding Conway's Law

With a proven set of development practices in place and documented, the biggest risk to delivering across these five teams would be in integration of code changes that would overlap in high-touch code sections. This risk was potentiated by the strategic technical debt we had accrued in order to meet the initial deadline and that existed in areas of the codebase that all teams would be likely to modify.

I had contributed some and steered other abstractions in the codebase to tend towards increased modularization. After initial release, I formalized those abstractions into a pipeline framework[2] whose goals were to:

- allow technical internals to be abstracted away when not needed,
- explicitly modularize disparate functionality,
- enable orthogonal features to execute concurrently without needing careful coordination,
- manage resource usage across all users of the framework,
- allow detailed visualization of pipeline execution, and
- increase transparency of internal execution such as timing and order of execution.

Refactoring the codebase paid off a significant amount of the mentioned strategic technical debt while also explicitly modularizing the codebase. This effort took some time and stalled the new teams from beginning development, which caused tension with those teams and pressure on myself. Having made it clear ahead of time to management and to product owners that we had to pay off our strategic technical debt in order to be able to keep growing helped justify the investment.

This delayed start was more than made up for as the codebase structure almost immediately came to reflect organizational structure, allowing all five teams to proceed without encumbering each other to any significant degree. The abstractions over technical internals of the software also lowered the learning curve for the codebase, for which at least one of the new teams expressed appreciation. All five teams met their deliverables for the next, again ambitious release, and the development practices we had established for initial release were successfully adopted by all involved teams.

[2] Available at: https://github.com/IBM/cusp

3. CONCLUSION

The task of consolidating an on-premise and a cloud product into a new multicloud offering in three months and across globally distributed teams with paradigmatically different practices brought many challenges. In this experience report, I have described how we met those challenges. Recapitulating, those were:

- a minimal set of software architecture modeling types, namely sequence and component/connector diagrams;
- distributing design authority using ADRs;
- adoption of development practices by demonstrating their value;
- self-organization of collaboration in continual, "20x5" development;
- minimal communication and coordination overhead via a "TCES" format;
- inverting the testing pyramid; and
- "fast-forwarding" Conway's law.

The combination of these tactics – and, of course, the hard work of everyone involved – enabled the architecting of a highly effective global development model. Two teams spread across two different countries were able to combine into a single international team, resulting in successful around the clock development. Furthermore, we were able to build on that success to expand our new organization to span five teams, without compromising the standards that we had developed. Presented with a similar situation in the future, I would apply the same nature of approach; the main area in which I would seek improvement would be long-term sustainability.

4. ACKNOWLEDGEMENTS

I would like to thank Yulia Pieskova for her extremely thorough and thoughtful feedback and advice during the writing of this report. I would like to thank Hiroaki Kikuchi, the lead of the Tokyo team in the collaboration described above, without whose close collaboration we would not have succeeded, as well as both teams' members, who were (in no particular order): Charles Gala, Chris Nolan, Yi-Shiuan Tung, Jennifer Manning, Daiki Tsuzuku, Takashi Fukuda, Takahito Tashiro, He Yanxia, and Tominaga Yasuyuki. I would like to thank William Chaparro for his guidance and advice throughout the described project.

REFERENCES

Stoyanovsky, A., & Chaparro, W. (2019). Accelerating Organizational Growth with Inspiration from Parliamentary Procedure. (n.d.). XP. Montreal.
Brown, S. (n.d.). Software Architecture for Developers (Vol. 2).
Keeling, M., & Runde, J. (2017). Architecture Decision Records in Action. SATURN. Denver.
Keeling, M., & Runde, J. (2018). Distribute Design Authority with Architecture Decision Records. Agile. San Diego.
Kruchten, P. B. (1995, Nov). The 4+1 View Model of Architecture. IEEE Software, 12(6), 42-50. 10.1109/52.469759
Nygard, M. (2011, November 15). Documenting Architecture Decisions. Cognitect. Retrieved May 11, 2022, from https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
Radigan, D. (n.d.). Standups for agile teams. Atlassian. Retrieved May 11, 2022, from https://www.atlassian.com/agile/scrum/standups
Setting guidelines for repository contributors. (n.d.). GitHub Docs. Retrieved May 11, 2022, from https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/setting-guidelines-for-repository-contributors
Treinen, J. J., & Miller-Frost, S. L. (2006, Oct-Dec). Following the sun: Case studies in global software development. IBM Systems Journal, 45(4), 773-783. 10.1147/sj.454.0773
Tuckman, B. W. (1965). Developmental sequence in small groups. Psychological Bulletin, 63(6), 384-399. 10.1037/h0022100
Verner, J. M., Brereton, O. P., Kitchenbaum, B. A., Turner, M., & Niatzi, M. (2014, Jan). Risks and risk mitigation in global software development: A tertiary study. Information and Software Technology, 56(1), 54-78. 10.1016/j.infsof.2013.06.005