Scaling agile development across loosely coupled teams using microservice architecture

ANASTAS STOYANOVSKY, IBM Watson
STEVEN PRITKO, IBM Watson

We observe that teams from different organizations with different incentive structures tend to focus on different system quality attributes and that this difference can impede collaboration. In this experience report, we present a case study of a microservice architecture-based approach to this problem that we adopted during a collaboration between IBM Watson and IBM Research. We propose that the overall framework described is generally applicable to scaling agile development across teams with different incentive structures.

1. INTRODUCTION

The quality attributes (QAs) of a system, such as performance, accuracy, fault tolerance, reliability, and maintainability, are various factors which affect its behavior, usability, and design across all its components. QAs can have tradeoffs with one another and thus their relative importance to a particular system must be weighed and balanced. If a particular QA affects a software system's architecture, it can be an architecturally significant requirement (ASR) [1].

A common trope is that of "research code" and refers to code written for research applications, with the subtext that such code can be difficult to maintain or extend, lack reusable abstractions, and have inexhaustive error handling. While it is true that code written in the course of a research project is often not suitable to be shipped in a product, another perspective on the phrase is that research code is intentionally written with emphasis on certain QAs, such as accuracy, over others that are more important to production code, such as maintainability.

The QAs that a development team focuses on often align with the incentives of the organization that team belongs to. In a research organization whose incentives are built around academic publication and furthering the scientific state of the art, an inclination towards focusing on accuracy at the expense of maintainability can graduate to a norm, while within a software engineering organization performance, usability, and fault tolerance will tend to be first concerns. When two teams from different organizations with different incentive structures and reporting chains are tasked to begin collaboration on a product, this difference in habits and norms can impede the development process or lead to tension.

In this experience report, we recount an approach we took towards preventing this tension by hoisting [3] certain QAs into a microservice based architecture so as to lead two teams to efficiently work towards a common goal while simultaneously following their respective organizations' incentives. In contrast with agile frameworks such as Disciplined Agile Delivery [5], ours is architecture-first perspective that tries to use system design to scale agile processes across teams.

2. BACKGROUND

IBM Watson and IBM Research are separate organizational units, the former being a software engineering organization first and the latter being a research organization first. They routinely collaborate and often a new product or feature is started with a collaboration between teams from each organization that have never worked together before. The difference in habits and incentives between these organizations needs to be resolved every time a new collaboration is formed, which especially comes to fore when considering what to do with a research prototype.

A research prototype is generally not written with maintainability as a focus, nor often extensibility or modularity. What portion, if any, of the existing code can be reused, where new code should be written, and how the ensuing development process will be conducted needs to be solved every time this situation occurs.

Anastas Stoyanovsky, 1710 Murray Ave., Ste. 300, Pittsburgh, PA 15217; astoyan@us.ibm.com
Steven Pritko, 1710 Murray Ave., Ste. 300, Pittsburgh, PA 15217; steven_pritko@us.ibm.com

The problem compounds if product development must start while the scientific work is still being developed: should there be code handoff, a common codebase, a prototype/reimplement cycle, or some other method of collaboration? How does one engineer a product and deliver it on time when the algorithm development has not finished, or is not past the conceptual stage? And, as we focus on here, can the collaborative development process be designed to take advantage of, rather than try to work against, differences between team incentives and norms?

3. CASE REPORT

In June 2017, an engineering team from IBM Watson, distributed between Pittsburgh, PA and Denver, CO, and a research team from IBM Research, based in New York City, NY began partnering on the development of an artificial intelligence application built on top of an existing, productized platform to provide state of the art functionality to solve a novel artificial intelligence task. Though the natures of the inputs to and outputs of the desired solution were defined, the nature of the solution itself was not known.

The core development process problem was to find an efficient way for the engineering team to iterate on the system design while allowing the research team to iterate on the science in order to achieve a production quality, state of the art system within an aggressive deadline. Technical problems to be solved included a data processing pipeline, real-time propagation of and reaction to state changes such as new or modified data, task scheduling and status monitoring of training jobs on a high-performance computing cluster, and trained model management.

3.1 Strategy

Our approach was to not only identify specific QAs as ASRs but to go farther by hoisting them into the system design in such a way as to allow each team to focus on the same QAs it normatively does, without having to trade off between them. We used a microservice architecture to facilitate system design built around this principle. An essential auxiliary activity was the recording of relevant system design decisions using architectural decision records (ADRs) [6] to serve as a record of the design process.

3.2 System Design

Given that the types of inputs and outputs of the desired product were known, we chose an ensembling [4] approach in which any number of different microservices can implement an algorithmic solution for the desired task, so long as they implement a common interface (Fig. 1). User input is asynchronously fanned out to all *Algorithm* backends for their returns to be disambiguated and de-duplicated before being stored and made available for user retrieval. The common interface each *Algorithm* microservice implements evolved over time, but changes were infrequent and made only as necessary because any changes had implications both for the *Ensembler* service and for each *Algorithm* backend.

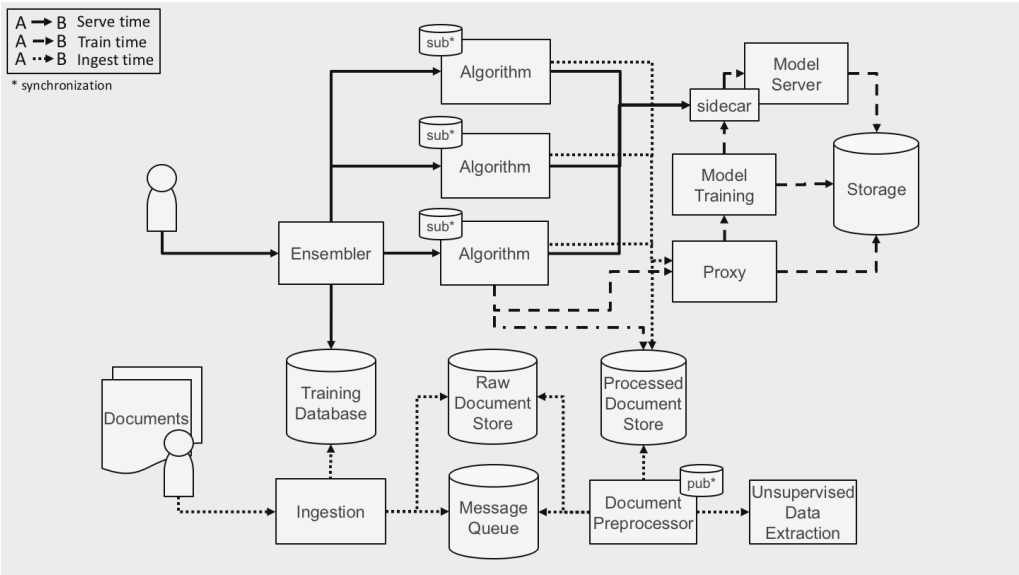

Figure 1. **System Architecture**

As the project developed and the various concerns of the overall application became better defined, we abstracted away each of those concerns common to all *Algorithm* backends behind APIs, leaving each backend itself stateless and only reading from or updating shared state as necessary. References to new or updated data arriving from the user via *Ingestion* are placed on a message queue for the *Document Processor* to pull from. Once *Document Preprocessor* completes and saves the output of its data normalization and annotation tasks to the *Processed Document Store*, it publishes a list of the consequent state changes, and each *Algorithm* backend subscribes to notifications of such state changes to create what is essentially a blackboard pattern [2], (pub*/sub* in the figure). Each *Algorithm* backend then triggers any required new training jobs via the *Proxy*, which abstracts away task scheduling in *the Model Training* service and which maintains the blackboard with training job status for the *Algorithm* backends to monitor. Once any training job completes, the resulting trained model is loaded into memory by the *Model Server* and made available for use. Finally, the user is able to query the system for inferences made based on the processed data, which are ensembled as described above and stored to the *Training Database* to be retrieved at will.

Using this architecture, reliability and fault tolerance are hoisted into the system design, leaving accuracy to be the chief concern of the *Algorithm* backend being developed by the research team and allowing the engineering team to iterate on the usual engineering tasks. The responsibilities for individual components of the system were discussed among and decided upon by both teams to avoid missed requirements and were then recorded in ADRs.

3.3 Summary of Results

Throughout the development process, as the concerns common to each backend became clear, utilizing a microservice architecture allowed the engineering team to iterate on the overall system design and implement individual components to provide the desired QAs, allowing sufficient time for the science produced by the research effort to evolve. For example, one backend was able to provide minimal service quality while initial nontrivial scaling and performance issues in the other backend were characterized and resolved.

This architecture was flexible enough to be adapted to the new business requirements that were added over time and allowed both teams to work with minimal communication and management overhead. In particular, this development process allowed both teams to work towards their own incentives in such a way that the final product had a cohesive design and had all desired properties.

Following this strategy, the teams quickly adapted well to this development process. However, a drawback to that process became apparent upon the discovery that, after two sprints with less communication than there should have been, both teams had identified and approached the same problem with mutually exclusive solutions. These solutions had to be reconciled, delaying development. Furthermore, towards the end of the project there was a significant barrier for one team helping another complete or debug an implementation because of a lack of familiarity with each other's codebases. Nevertheless, and despite significant new requirements having been introduced midway through the project, the overall system met its functional specifications within a few weeks of the aggressive original deadline.

4. CONCLUSIONS

We found that choosing a microservice architecture can facilitate a development process for teams in different organizations and with different incentives to be able to do long-term work in a decoupled manner and having significantly reduced communication and management overhead. The overall process could be improved by increasing cross-team code reviews, having some minimal formal project management role, and/or making more frequent use of architectural decision records in order to avoid loss of cross-team synchronization.

We have examined here a method for scaling agile development across loosely couple engineering and research teams by hoisting specific quality attributes into system design using a microservice architecture. Although this is one specific application of this approach, we believe that the overall framework described here is generally applicable to scaling agile development across teams with different incentive structures.

5. ACKNOWLEDGEMENTS

We would like to thank, in no particular order, Gabe Hart, Mark Tyneway, Manjari Akella, Alfio Gliozzo, Nicolas Fauceglia, Gaetano Rossiello, Michael Glass, and Sarthak Dash for their passion and hard work throughout the project described here. We would also like to thank Tim O'Connor for his critical feedback throughout the writing process.

REFERENCES

[1] Chen, Lianping (2013). "Characterizing Architecturally Significant Requirements". IEEE Software. 30 (2): 38–45. doi:10.1109/MS.2012.174.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1995). "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Fairbanks, George (2014). "Architectural Hoisting. IEEE Software 31 (4): 12-15. doi:10.1109/MS.2014.82

[4] Rokach, L. Artif Intell Rev (2010). "Ensemble-based classifiers". 33: 1-39. doi:10.1007/s10462-009-9124-7

[5] Scott W. Ambler and Mark Lines. 2012. Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise (1st ed.). IBM Press.

[6] Tyree, J. and Akerman, A. (2005). "Architecture decisions: Demystifying architecture." IEEE software. 22 (2): 19-27. *doi:10.1109/MS.2005.27*