Enabling autonomous teams in large-scale agile through architectural principles

Jan Henrik Gundelsby
Knowit
Lakkegata
Oslo, Norway
jhg@knowit.no

ABSTRACT

Teams in large-scale projects and programs need to reach agreement on many decisions with experts, managers and other stakeholders. The need for aligning the work and the process with the rest of the organization reduces team autonomy. Coordination by architecture is one strategy to handle this challenge. Consequently, we did a case study of a large-scale software program consisting of nine teams, to understand how the architecture can enable team autonomy. Initially teams had limited autonomy, because of high task dependencies with other teams and experts. By introducing an architecture based on business domains and APIs, teams got full responsibility for a set of components, and solved the alignment problem by letting other teams access the resources through an API. The new architectural strategy can be understood as structuring by business domains and APIs, instead of features that span the whole code base.

CCS Concepts

•**Software and its engineering** → **Collaboration in software development; Programming teams;**

Keywords

Autonomy, Self-management, System architecture, API-first, Microservices, DevOps, Teamwork

1. INTRODUCTION

According to the agile principles, motivated and empowered software developers, testers and designers - relying on technical excellence and simple designs - create business value by delivering working software to users at regular short intervals. These principles have spawned many practices that are believed to deliver greater value to customers. The core of these practices is the idea of self-managing or autonomous teams whose members work at a pace that sustains their creativity and productivity.

Large projects and programs are increasingly adopting agile development practices and most developers today report to work in agile teams [1]. Using agile practices in large projects creates new challenges because self-managing agile teams need to reach agreement on many decisions with experts, managers and other stakeholders. The teams also need to be highly aligned. If one team breaks the quality, functionality, or change important components, it will affect other teams. The need for aligning the work and the process with the rest of the organization, and for coordinating decisions externally, reduces team autonomy and empowerment.

Coordination by architecture is one strategy for strengthening team autonomy. The software architecture is the fundamental technical organization of a system, and for more than forty years, researchers have argued that system architecture play an important role in coordinating development work [2, 3]. In 1972, David Parnas [3] suggested that modular design enables independent decision making. Ovaska et al. [4] found that participants in a large-scale distributed project coordinated their development work through software module interfaces. Relying on this strategy, each software module could be developed separately, and thus the coordination problems could be mitigated. However, experience shows that the effectiveness of coordination by architecture is limited since system modules are never truly independent. This study aims to answer the following research question:

How can software architecture in large-scale agile initiatives enable autonomous teams?

1.1 Software architecture and team autonomy

A classic definition of software architecture is given by Clements [5]: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." Every software system has an architecture because every system can be shown to be composed of elements and relations among them. The architecture embodies information about how the elements relate to each other. A recent trend within architecture is microservices architecture. It is defined as "an approach to developing an application as a set of small independent services. Each of the services is running in its own independent process" [6]. When re-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

This is the accepted version of the publication. The final version is published in
XP '18: Proceedings of the 19th International Conference on Agile Software Development: Companion
https://doi.org/10.1145/3234152.3234183

lating microservice architecture to the definition of software architecture, each software element corresponds to an independent microservice. We use the label "component" as a synonym for "microservice".

In large agile projects or programs, teams are often assigned the responsibility for a component or collection of components. Guzzo and Dickson [7] define an autonomous team as "employees that typically perform highly related or interdependent jobs, who are identified and identifiable as a social unit in an organization, and who are given significant authority and responsibility for many aspects of their work, such as planning, scheduling and assigning tasks to members, and making decisions with economic consequence." In their study of a large-scale agile program, Dingsøyr et al. [8] found that the system architecture gave individual teams a stable working environment with room for teams to design their own local sub-architectures that contributed to team efficiency. The architecture enabled the teams to take many decisions regarding their work, i.e. increased the level of autonomy.

In the following, Section 2 describes our methods, analysis and our case-study. Section 3 describes results. First through a description of how the large-scale program was organized to mitigate scaling challenges, and then providing details on software reference architecture and team autonomy. Section 4 discusses the results and concludes.

2. METHOD

To answer our research question, we conducted a case study[9] of a large-scale software development program. The software development program consists of nine teams. As this is an ongoing longitudinal case study, we are continuously collecting new data. For this paper, we did a group interview with one team (Alpha) and follow-up interviews of two of the developers and one architect from the program. We observed the team and collected case material such as Jira documentation and strategy documents. The author has since 2003 been one of the key developers and architects in the software development program. The data included in this paper was gathered from March to December 2017.

The author used a variety of strategies to analyze the material. One such strategy was to describe the program and its context in a narrative to achieve an understanding of what was going on in the program. In the analysis, we emphasized how events were interpreted by different participants in the program. In this study, the interviews, documentation and observations were compared to each other. By doing this, some phenomena started to emerge, which were compared to existing theories on autonomy and architecture. We then identified, analyzed, and reported patterns (themes) that emerged within the data by using thematic analysis.

3. RESULTS

This section present challenges related to autonomy when scaling a development organization relying on microservices. We present a new strategy to give the team a higher level of autonomy, and the new organizational and architectural setup. Lastly, we present findings on team autonomy.

3.1 Challenges scaling a development organization

Our case is a large Norwegian municipality with approximately 50 000 employees, and 50 organizational units, all varying in size, responsibilities, domain, IT-competence and funding. The municipality have its own development program, responsible for integrating hundreds of internal systems and is a data hub for communicating to the public and businesses. The platform is an in-house developed service delivery platform (SDP). A SDP enables automation of routine activity associated with building, testing, and deploying services, including the provisioning and ongoing management of infrastructure services. It is the foundation on which deployment pipelines for building, testing, and deploying individual services run [10]. The SDP is built from open source components as a foundation and run on premise. Java, Scala and Python are used as the main programming languages.

Since 2009 the municipality has standardized on a microservices architecture, slowly scaling both the development organization and the number of components. In 2016 there were approximately 200 components running in production. This resulted in several challenges: One being a too large codebase to comprehend, as one of the developers stated: "I don't know what this microservice does?" "What happens if I change X?", "Let's add Y instead". At the time, the teams were organized as feature teams, building a feature making it all the way to production using DevOps-principles [11]. The developers deployed to production themselves, and also monitored the components. Having full responsibility of a feature gave the teams a high level of autonomy. A feature often involved changes in numerous microservices. Often developers from other teams were needed to understand the structure and logic of the components involved, when implementing a feature. Many dependencies made a feature cumbersome to implement and test. As more features were added, the program became even more complex because different components often had different non-functional requirements. One set of components required a particular type of service level agreement. Another set of components had different requirements to scale independently of others and with a higher user load.

While the team was given responsibility to develop and maintain the feature, team autonomy was limited by the software architecture. It had become a big ball of mud. In 2016 the municipality started working on a new strategy to maximize team autonomy, enabling teams to deliver quality software rapidly and independently of each other.

3.2 A new architectural strategy for enabling autonomous team

The new strategy had two goals. First, teams were given responsibility for a collection of microservices within a business domain, instead of having responsibility for a set of features implemented all across the code base. A domain is a structure which comprise software elements [5], with some form of natural grouping. The main idea being that the team can focus on one business domain exposing a defined API, or other natural grouping of the microservices. Due to practical reasons, a team could be responsible for multiple domains.

Second, aligning API-boundaries with team boundaries [10]. By introducing an API-centric architecture (Fig. 1), teams are responsible for one or more domains and they apply explicit service boundaries between domains. The only interface into a domain is through the API-component, no

This is the accepted version of the publication. The final version is published in
XP '18: Proceedings of the 19th International Conference on Agile Software Development: Companion
https://doi.org/10.1145/3234152.3234183

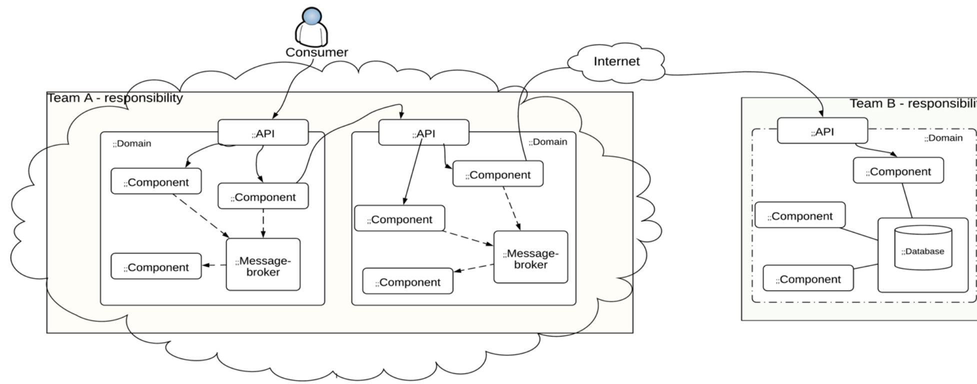

Figure 1: API-centric architecture

exceptions are made. An API-centric architecture, is analogous with the "API-mandate" famously issued by Amazon CEO Jeff Bezoz [12].

3.2.1 Program setup

When the new strategy was decided, the program consisted of a project management team with solution architects and managers, 80 developers organized into nine development teams and one infrastructure/platform-team providing the SDP to the development teams. The goal of the SPD was to enable teams to deploy code to production, create their own test environments and experiment with different tools and technology. Each team had a product owner, was co-located in open office spaces, conducted a daily stand-up meeting [13], but there was is no governance in place for agile practices across teams. Teams followed Kanban, Scrum, or could pick the practices from the agile method best suited for the team. Working remotely (e.g. from home) was allowed, but not encouraged. For task management and planning the program relied on Jira. A Scrum-of-Scrums [14] were conducted every Monday. Every second week three coordination activities were conducted. 1) an internal team lead-meeting discussing the internal workings of teams, 2) and an architect-forum focusing on strategy, platform and other national initiatives relevant for the municipality, 3) a demo day where teams presented their latest progress to everyone interested.

3.2.2 Team setup and autonomy

All teams are run by the mantra "if you build it, you run it" [15], meaning that developers are also doing operations and are in full control of the application lifecycle. A team consists of people with the combined competence to implement, test and deploy into production. Within the domain strategy, all components can be deployed independently and the team is responsible for the internal structure of the domain, planning, scheduling and assigning tasks to members. Each domain has its own version control of the team's components, and only team members can deploy code to production. The code is open to other teams, and pull requests can be created by other teams, but the code has to be approved and deployed by the team responsible for the component. All teams have full responsibility for one or several domains except one team that was responsible for shared functionality, shared libraries and frameworks. This team had many dependencies to others, and therefore experienced problems with task dependencies. Team size is scaled with the number of components they are responsible for, splitting the team and domains if necessary.

Team Alpha is a small team with four members including the team lead. The members work for three different consulting companies. The team has responsibility for 38 components. The team lead has time developing code and mentoring people on the team by e.g. doing code reviews. Team Alpha delivers software regularly, and the product has received a high degree of attention in the organization. At one point they even won a price for their product. On the architectural side of things, the dependencies to other teams are few and well-defined via APIs. When making changes to components almost all changes are done to components well-known and owned by the team. The domain is relatively simple to understand and has a stable API.

4. DISCUSSION AND CONCLUSION

This is an ongoing study where we have studied nine teams, one of them in more depth. We will now discuss our research question

4.1 (RQ) How can software architecture in large-scale agile initiatives enable autonomous teams?

We have described how a large-scale program changed structure of the system (the architecture) to enable autonomous teams. Before the architectural change, teams had the full responsibility of a set of features (development, maintenance and operations). However, teams could not make decisions regarding their plans, schedules or task without discussing and negotiating with other teams or experts. That is, the old architecture resulted in a constant need for alignment and coordination that reduced the team autonomy because of high task dependency. Task dependencies refer to conditions where completion of one task is necessary before the next can begin [16]. The new architectural strategy, organized according to business domains and APIs, enabled the teams to have full responsibility for a set of components, and solved the alignment problem by letting other teams access the domain through an API (Figure 1). The new architectural strategy can be understood as structur-

This is the accepted version of the publication. The final version is published in
XP '18: Proceedings of the 19th International Conference on Agile Software Development: Companion
https://doi.org/10.1145/3234152.3234183

ing by domains and APIs, instead of features that span the whole code base.

By introducing the domain concept, team goals became clearer since a team then handled a business domain. Clarity of organizational goals increases team performance of autonomous teams [17]. While teams became more autonomous we found that the team responsibility for shared functionality, shared libraries and frameworks, still had many dependencies to other teams and therefore low autonomy. While the architectural strategy is important we found that the self-service platform supported autonomy. In addition to deploy code to production, teams could then create their own test environments and experiment with different tools and technology.

4.2 Implications for practice

We believe that our study has the following main implications for practice:

Find the right type of domain. Three strategies are 1) a team is responsible for an end-user product and put all components related to the product in one domain or 2) mapping a team to an organizational unit. 3) Components may have different non-functional requirements, e.g. some have a 24/7 SLA. In this case, it may be necessary to group components with the same service level agreement in the same team domain.

Modify the service boundaries continuously. If a team is not able to deliver features independently, it is essential to continuously modify the service boundaries accordingly. Practitioners often refer to Conway's Law [18] when using this method, suggesting evolving your team and organizational structure to promote your desired architecture. This is also referred to as the "Inverse Conway manoeuvre" [19].

Implement API versioning and management. The external domain API that other teams depend on needs to be stable. Versioning and API-management of an API are critical, meaning that other teams can rely on the stability of external domains and its components. API-management therefore need to include properties such as well-documented APIs, well-defined service level agreements (SLAs), authentication and authorization and knowing who is using the API so it can evolve in a safe manner.

4.3 Conclusion and further work

We found that a architecture organized by domains with API's and service boundaries, and a self-service platform enables team autonomy. Our findings point out a number of directions for future research. We plan to collect data from more teams to understand the coordination challenges and if different types of business domains influence team autonomy. Further, there is a need to understand how to increase autonomy in teams responsible for shared functionality, shared libraries and frameworks. Finally, as domain change, there is a need to understand how such changes influence autonomy.

5. REFERENCES

[1] Viktoria Stray, Nils Brede Moe, and Gunnar R. Bergersen. Are daily stand-up meetings valuable? a survey of developers in software teams. In *Agile Processes in Software Engineering and Extreme Programming (XP2017)*. Springer, (in press).

[2] James D Herbsleb and Rebecca E Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE software*, 16(5):63–70, 1999.

[3] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[4] Päivi Ovaska, Matti Rossi, and Pentti Marttiin. Architecture as a coordination tool in multi-site software development. *Software Process: Improvement and Practice*, 8(4):233–247, 2003.

[5] Paul C Clements. Software architecture in practice. *Diss. Software Engineering Institute*, 2002.

[6] Namiot Dmitry and Sneps-Sneppe Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9), 2014.

[7] Richard A Guzzo and Marcus W Dickson. Teams in organizations: Recent research on performance and effectiveness. *Annual review of psychology*, 47(1):307–338, 1996.

[8] Torgeir Dingsøyr, Nils Brede Moe, Tor Erlend Fægri, and Eva Amdahl Seim. Exploring software development at the very large-scale: a revelatory case study and research agenda for agile method adaptation. *Empirical Software Engineering*, 23(1):490–520, 2018.

[9] Robert K Yin. *Case study research and applications: Design and methods*. Sage publications, 2017.

[10] Jez Humble, Joanne Molesky, and Barry O'Reilly. *Lean enterprise: How high performance organizations innovate at scale*. " O'Reilly Media, Inc.", 2014.

[11] Mike Loukides. *What is DevOps?* " O'Reilly Media, Inc.", 2012.

[12] Martin Fowler. Who needs an architect? *IEEE Software*, 20(5):11–13, 2003.

[13] Viktoria Stray, Nils Brede Moe, and Dag I. K. Sjøberg. The daily stand-up meeting: Start breaking the rules. *IEEE Software*, (in press), 2018.

[14] Maria Paasivaara, Casper Lassenius, and Ville T Heikkilä. Inter-team coordination in large-scale globally distributed scrum: Do scrum-of-scrums really work? In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 235–238. ACM, 2012.

[15] C O'Hanlon. A conversation with verner vogels, 2006.

[16] Diane E Strode. A dependency taxonomy for agile software development projects. *Information Systems Frontiers*, 18(1):23–46, 2016.

[17] Erik Gonzalez-Mulé, Stephen H Courtright, David DeGeest, Jee-Young Seong, and Doo-Seung Hong. Channeled autonomy: The joint effects of autonomy and feedback on team performance through organizational goal clarity. *Journal of Management*, 42(7):2018–2033, 2016.

[18] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[19] Rebekka Parsons. Inverse conway maneuver. https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver, 2015.

This is the accepted version of the publication. The final version is published in
XP '18: Proceedings of the 19th International Conference on Agile Software Development: Companion
https://doi.org/10.1145/3234152.3234183