Mining Dependencies in Large-Scale Agile Software Development Projects: A Quantitative Industry Study

Katarzyna Biesialska
Universitat Politècnica de Catalunya
Barcelona, Spain
katarzyna.biesialska@upc.edu

Xavier Franch
Universitat Politècnica de Catalunya
Barcelona, Spain
franch@essi.upc.edu

Victor Muntés-Mulero
Beawre Digital S.L.
Barcelona, Spain
victor.muntes@beawre.com

ABSTRACT

Context: Coordination in large-scale software development is critical yet difficult, as it faces the problem of dependency management and resolution. In this work, we focus on managing requirement dependencies that in Agile software development (ASD) come in the form of user stories. *Objective*: This work studies decisions of large-scale Agile teams regarding identification of dependencies between user stories. Our goal is to explain detection of dependencies through users' behavior in large-scale, distributed projects. *Method*: We perform empirical evaluation on a large real-world dataset from an Agile software organization, provider of a leading software for Agile project management. We mine the usage data of the Agile Lifecycle Management (ALM) tool to extract large-scale development project data for more than 70 teams running over a five-year period. *Results*: Our results demonstrate that dependencies among user stories are not frequently observed (the problem affects around 10% of user stories), however, their implications on large-scale ASD are considerable. Dependencies have impact on software releases and increase work coordination complexity for members of different teams. *Conclusion*: Requirement dependencies undermine Agile teams' autonomy and are difficult to manage at scale. We conclude that leveraging ALM monitoring data to automatically detect dependencies could help Agile teams address work coordination needs and manage risks related to dependencies in a timely manner.

CCS CONCEPTS

• **Information systems** → **Data mining**; • **Software and its engineering** → **Software development process management**; **Risk management**.

KEYWORDS

Requirement Dependencies, Agile Software Development, Mining Software Repositories

ACM Reference Format:
Katarzyna Biesialska, Xavier Franch, and Victor Muntés-Mulero. 2021. Mining Dependencies in Large-Scale Agile Software Development Projects: A Quantitative Industry Study. In *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3463274.3463323

EASE 2021, 21 - 23 June, 2021, Trondheim, Norway
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway*, https://doi.org/10.1145/3463274.3463323.

1 INTRODUCTION

Agile software development (ASD) advocates increasing customer value through short and continuous product release cycles informed by feedback. This software development paradigm has been widely adopted in the industry for quite some time now.

User stories are the primary vehicle through which software requirements are communicated within ASD teams. In principle, user stories should be independent, negotiable, valuable, estimable, small and testable – according to the popular INVEST framework proposed by Wake [40]. Here we focus on the first rule, i.e. requiring user stories to be independent. This constraint prescribes non-overlapping scope of user stories and the ability to schedule them in an arbitrary order. Yet, the condition is often not satisfied. It is noteworthy that dependencies that remain unmanaged (either misidentified or ignored) can generate delay due to hindered progress or be a source of serious software and system failures [17, 26, 33, 42], which may result in delayed software delivery, increased costs as well as cause stakeholders' dissatisfaction [10].

Nevertheless, the evidence found in the literature regarding the prevalence of interdependent requirements and the importance to ensure their isolation is inconclusive. On the one hand, practitioners tend to emphasize that dependencies between requirements are critical [18] and try to avoid them as much as they can [26]. On the other hand, software professionals tend to not regard the problem of interdependency of requirements as important and consider it to be related to the context [28]. Some studies indicate that as many as 50% to 80% of all project requirements are interdependent [6, 7]. However, contradictory research findings suggest that the independence of requirements is frequently ensured by practitioners [28]. What is more, there is a lack of research studies addressing the topic of dependencies (especially including cross-team dependencies) specifically in ASD environments [35, 38]. For this reason, our study aims to fill this gap and provide more insights on how dependencies are considered and handled in industry in Agile implementations.

When it comes to work coordination, the size of software organization matters. Small and medium companies struggle because of interdependence between projects and teams, but bigger companies even more so [12]. Consequently, management and communication of dependencies between teams as well as dependency risk management, which aims to mitigate the negative impact of dependencies on team performance, appear as major challenges in large-scale ASD [36]. Therefore, this work addresses the scarcity of scientific evidence on how dependencies are handled in large-scale Agile implementations [20, 38]. In industry, management of large-scale ASD projects is often coordinated through Agile project management (ALM) tools [39]. Such software is a relevant source of information

on how Agile teams assign tasks or perform during iterations, to name just a few insights that might be acquired while mining their usage data.

2 BACKGROUND AND RELATED WORK

At first, Agile development methods were said to be best suited for small, co-located teams. However, inspired by success stories of small Agile organizations, large-scale variants of Agile have been increasingly adopted by large organizations [12, 15, 21, 29].

2.1 Large-Scale Agile Software Development

Unfortunately, there is no unified definition of what large-scale Agile actually is; various explanations have been offered throughout the years. Most of them use the number of people involved, less often they describe large-scale Agile by means of a project budget or the size of software – i.e. requirements, user stories, features, or lines of code [12, 14]. Let us take a closer look at two definitions that seem most universal and summarize well the large-scale characteristic from our perspective. The first definition is an interpretation provided by Dingsøyr et al. [13] which defines size of Agile with respect to the number of collaborating and coordinating teams: where large-scale is when there are 2-9 collaborating teams and as very large-scale above 10 collaborating teams [13]. The second one, by Dikert et al. [12], defines large-scale as a software development organization having at least 50 people or comprising at least 6 teams.

As the size of Agile implementation increases, so does the complexity of coordination. Large-scale ASD projects observe unintended changes, unknown interdependencies, cross-team collaborations that challenge current coordination mechanisms, which in effect evolve with time as project stakeholders need to address new problems (e.g. new mechanisms for coordination emerge) [27, 30, 34]. For instance, as reported by Moe et al. [27], the most popular approach for cross-team coordination in a large-scale setting is to hold regular meetings between representatives of Scrum teams (the so-called Scrum of Scrums).

To address coordination challenges, a wide range of large-scale Agile frameworks has emerged throughout the years, such as Disciplined Agile Delivery (DAD) [1], Large Scale Scrum (LeSS) [22], or Scaled Agile Framework (SAFe) [23]. Particularly SAFe has risen to prominence, becoming effectively the dominating Agile scaling practice [39]. The SAFe framework, by its nature, empowers autonomous teams making teams responsible for how they conduct their work. As demonstrated in [26], for many teams self-directed task allocation is not an easy and straighforward Agile practice to follow. Notwithstanding, any kind of dependency that has an influence on work order is a serious obstruction to upholding the rule. Even more so, if a dependency spans numerous teams and different team members.

2.2 Dependency Management in Software Engineering

The body of research on dependencies is mainly focused on general dependencies in non-ASD environments [38]. Importantly, dependencies in the ASD context are potentially identified at a different stage than in plan-driven software projects [38]. Strode [38] argues that unlike in plan-driven software projects, requirements are not written down in advance in a predefined requirement specification, but are rather continuously generated and processed in short development cycles. Sekitoleko et al. [36] recognizes five main challenges associated with cross-team dependencies in a large-scale ASD: the planning challenge, the task prioritization challenge, the knowledge sharing challenge, the code quality challenge, and the integration challenge. In a similar vein, numerous studies have linked dependencies with the coordination needs within software teams (e.g. [4, 35]). Hence, naturally, the problem of dependency management can be viewed through the lens of coordination theory. According to the *Coordination Theory* framework devised by Malone and Crowston [25], coordination can be defined as "the process of managing dependencies between activities", where dependencies break down to three basic types: (i) task-task, (ii) task-resource, (iii) resource-resource [9]. Where the term *task* refers in this case to a work item or activity to be accomplished. Similarly, Strode [38] devised a dependency taxonomy for ASD projects, which outlines how dependencies can be addressed by different coordination mechanisms.

In SAFe, for programs with many value streams and release trains, dependencies between stories are inevitable. In order to manage dependencies between teams large-scale ASD projects it is often recommended to hold Scrum of Scrums [15], yet some studies provide evidence that questions efficiency of this method in large projects [30, 34]. Stakeholders such as product managers, product owners, or release train engineers need to know about dependencies between teams (including the level on which they occur) to coordinate planning and execution of software. In SAFe, Scrum Masters with the help of their teams are responsible for managing dependencies.

In general, there are many types of dependencies. Therefore, various dependency requirement models [11, 43] have been proposed in the Requirements Engineering domain. In this study, we focus on the simplest case, which indicates whether there is a dependency or not between user stories and distinguishes the order (i.e. predecesssor-successor).

2.3 Tool-Aided Software Engineering

With a growing number of technological tools supporting software development teams in their work and increased interest in data-informed decision-making, the Mining Software Repositories (MSR) field has experienced dynamic growth in recent years.

Interaction data collected from tools such as Integrated Development Environments (IDE) or issue trackers have received a lot of attention in the research community [3, 31]. However, logs of developer actions or issue management data do not reveal the full spectrum of software development activities. In contrast, ALM tools capture the activities of different stakeholders involved in the software development process across different tasks (e.g. requirements engineering, testing, defect resolution). Yet, there exists a very limited set of software engineering studies that have leveraged data coming from that data source (e.g. [24, 37]).

Furthermore, ALM tools are a source of unobtrusively collected information on how Agile teams assign tasks or perform during iterations [24]. Comparing to traditional interview-based research

or controlled experiments, such empirical evidence gathered from regular project activities offers researchers access to unprocessed usage and behavior data, being a “gold-mine of actionable information" [19]. Likewise, such a source of information can supply project teams with up-to-date monitoring data allowing them better understand complex business processes and improve decision making in their organizations [3].

3 CONTEXT AND MOTIVATION

In this section, we describe the industrial context of this study.

3.1 Company Context

We performed this study in cooperation with CA Technologies, now a Broadcom company (for brevity referred to as CA in the rest of this paper). CA is a multinational corporation delivering enterprise software products. The company is a provider of Rally (formerly known as CA Agile Central), an Agile project management software. Rally is one of the leading ALM tools on the market, according to the VersionOne’s Annual State of Agile Survey [39]. CA runs its teams and projects according to SAFe principles and has considerable experience with scaling ASD teams inside and outside the organization. Overall, the company has supported hundreds of enterprises during their large-scale organizational transformations. With acquisition of Rally Software (for brevity referred to as Rally in the rest of this paper) in 2015, CA has adopted *Big Room Planning* for well over a decade on its own products, which is now practised with each line of business, and has helped hundreds of enterprises launch this method for themselves. The company often runs large programs with several SAFe value streams and release trains. Although Rally uses SAFe and its ALM tool in all business units, we purposefully selected only IT-related teams for our study. This ensures that the collected information is relevant to software development activities.

3.2 Scaled Agile Implementation at the Company

Work items are hierarchical. *Epics* decompose to *features*. A *feature* is typically broken into multiple *user stories*. *User stories* are primary work items and may consist of *tasks*. *User stories* and *tasks* are considered intra-team level work items.

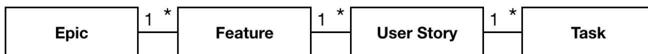

Figure 1: Hierarchy of work items in Rally, where 1:* denotes a one-to-many relationship.

The ALM tool supports creating dependencies between portfolio items of the same type on the lowest level – this means effectively user story’s dependencies. Story-to-story dependencies belong to the *predecessor-successor* relationship type. This type of relationship indicates an ordinal relationship in which user stories are dependent upon each other with regard to their completion. Therefore, a predecessor is a user story which must be completed before a user story identified as its successor. Predecessors and successors can belong to various projects, provided they exist within the same workspace. A user story may have many predecessors as well as successors. In the ALM tool, every user story has a separate attribute to hold information about possible dependencies, as shown in Figure 2. A user can only indicate whether the dependency is preceding or succeeding a given user story.

Within SAFe, product teams may use a combination of practices such as Scrum, Lean, or Kanban. In the studied Rally tool, Kanban stages are optional and customized by a team if it decides to follow Kanban principles.

Iteration is a time-box within which development work is performed. The studied ALM tool facilitates release management tasks by allowing users to create a release plan and adding user stories to a scheduled release. The ALM tool raises a warning if user stories with dependencies are not scheduled. Rally’s teams feature a range of software engineering practitioner roles: developers, product owners, development managers, release train engineers, testers, scrum masters, among others.

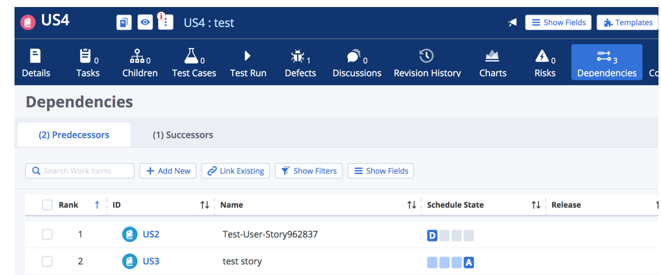

Figure 2: Screenshot of the ALM tool showing dependencies.

4 STUDY DESIGN

This section outlines: methodology that we have followed while conducting our study, research questions that we have posed as well as explains the data collection and analysis of our dataset.

4.1 Research Method

This empirical study leverages two orthogonal research approaches. Firstly, our study falls into a category of an exploratory study; it is an observational study, where the findings are drawn from observations. Secondly, considering two complementary (rather than competitive) methods for conducting empirical studies, i.e. quantitative and qualitative [8, 41], we conclude that this study is a quantitative investigation. Namely, we support our exploratory case study only with quantitative data, which was gathered through a systematic process, as described later in this section. Noteworthy, we leverage the opportunity to study large-scale ASD process in real, industrial setting. In that respect, our study manages to overcome some of the challenges that Easterbrook et al. [16] consider to be inevitable in research conducted in industrial settings, e.g. the *Hawthorne effect*. Concretely, without interfering with the observed ASD processes, we were able to gather data that allowed us to find out what practitioners actually do rather than say they do.

4.2 Research Goal and Questions

In the empirical strategy, we used the Goal Question Metric (GQM) [2, 5], according to which we define the goal of our study as follows: analyze decisions of large-scale Agile teams regarding identification and handling of dependencies between user stories; for the purpose of understanding how dependencies are detected and how they affect projects; with respect to projects' characteristics and team members' activities; from the point of view of software engineering researchers; in the context of data collected from 71 software projects run according to SAFe. We designed three research questions. Based on the above, this study formulates and tries to answer the following research questions (RQs):

RQ1: *How often and when are dependencies identified by members of ASD projects?*
In principle, ASD enforces no dependencies between user stories. These are independent items by default. Our hypotheses is that in reality this constraint does not hold in many cases. We are interested at what stage of an iteration dependencies are identified by teams. Is it at the very beginning, or later on? Are dependencies declared just after the creation of a user story or maybe further down the road?

RQ2: *Do team characteristics influence the identification and number of dependencies?* In this RQ, we aim to understand how teams in a large-scale Agile organization cope with dependencies. Does the team size, composition or distribution play a role? What is the ratio of internal to external team dependencies? Our hypothesis is that with increasing number of teams and people involved in the development project, the number of cross-team dependencies usually increases. Yet, dependencies at the cross-team level have received little attention from software engineering researchers to date [35]. In this RQ, we aim to shed some light on that matter.

RQ3: *How often are dependencies closed before the iteration ends?* This question provides insights on the timeliness of the dependency resolution and coordination within teams. When are dependencies closed? Does it happen that a user story is closed with an open dependency? We investigate if there is a relationship between the stage of a user story and the type of dependency that is being closed.

4.3 Data Retrieval and Analysis

Data for this study is protected by a nondisclosure agreement; therefore, the information presented in this paper is sanitized and contains only necessary level of detail (e.g. we identify user roles rather than individual persons) to draw scientific conclusions for research purposes. At Rally, projects are organized in a hierarchy. We collected data from 71 projects from Rally. All these projects were actively maintained at least for 5 years. All selected projects are IT-related, some related to the development of software products, while other devoted to IT support and maintenance activities. The selected projects involved users in different roles such as architects, developers, product owners or testers, among others.

With respect to co-location of the teams, Rally uses a hybrid approach. The data from selected projects is related to the activity of teams located in 7 distinct locations: 5 in the United States and 2 in other countries (i.e. India and the EMEA region). Some team members worked from *home office* which was indicated as a separate office location and did not provide detailed whereabouts.

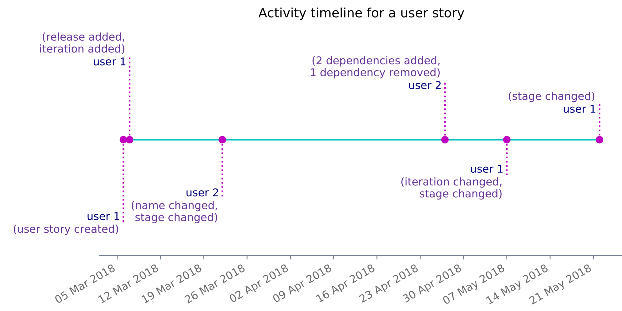

Figure 3: Example of user story activity and duration retrieved from the ALM tool.

Similarly, for some team members detailed location was not specified. However, in some cases we were able to find out timezone in which they worked, hence we made an educated guess and assigned an appropriate location with a reasonable degree of precision.

For efficient querying of the current state of work items, we used the Representational State Transfer (REST) API provided by Rally. Using the API we were able to collect the current objects in JavaScript Object Notation (JSON) format. To retrieve historical data, we leveraged another type of REST API provided by the company. The data collected through this interface included revision history of user stories as well as snapshots of those work items. An illustrative example of information retrieved from revision histories is shown in Figure 3. Snapshots provide information on the state of the artifact at the time it is changed. Noteworthy, dependencies between user stories do not have explicit identifiers (similar to work item artifacts) in the ALM tool. Hence, we needed to process the revision history of a user story to extract the dependency assigned to the user story. Based on that we created our own dataset of likely dependency-related data. We also used the company's internal database to enrich our dataset with historical information, such as the project hierarchy or user data.

The summary of collected dependency-related data from the studied ALM includes:

- *project-related data* (e.g. project hierarchy, team members, user stories within a project, iterations, releases);
- *user story-related data* (e.g. creation date, volume and nature of activity within the user story, dependencies, iterations and releases to which the user story belongs);
- *user-related data* (e.g. user roles, user story owners, person identifying dependency, contributors of the user story with a dependency, members of the team to which the user story belongs).

Based on the retrieved data, we defined our own metrics to aggregate related data. In Table 1, we characterize our dataset and provide descriptive statistics for selected metrics.

5 RESULTS

We have structured our empirical evaluation based on research questions presented in the previous section.

Table 1: Metrics descriptions and descriptive statistics for the dataset used in the study.

Metric	*n*	Unique *n*	Type	Skewness	Kurtosis	Min	Q1	Median	Q3	Mean	Std. Dev.	Max
#predecessors-proj	1181	n/a	number	5.45	43.45	1.00	1.00	1.00	1.00	1.28	0.84	11.00
#successors-proj	1050	n/a	number	5.70	54.44	1.00	1.00	1.00	1.00	1.43	1.16	18.00
#team_members-proj	1032	588	number	5.00	39.00	1.00	3.00	6.00	10.00	8.00	9.00	82.00
#outside_team_contributors-proj	66	32	number	1.00	0.00	1.00	6.00	12.00	24.00	17.00	14.00	55.00
days_from_iter_change-dep	4225	n/a	number	1.02	0.16	1.00	14.00	125.00	317.00	190.29	196.92	770.00
days_to_iter_change-dep	2249	n/a	number	0.89	-0.65	0.00	6.00	58.00	387.00	206.96	248.80	776.00

#(predecessors,successors)-proj: number of predecessors/successors per project; #team_members-proj: number of team members per project; #outside_team_contributors-proj: number of contributors from outside the team per project; days_(from,to)_iter_change-dep: number of days that passed from the last iteration change/remained to the next iteration change for a user story with a dependency;

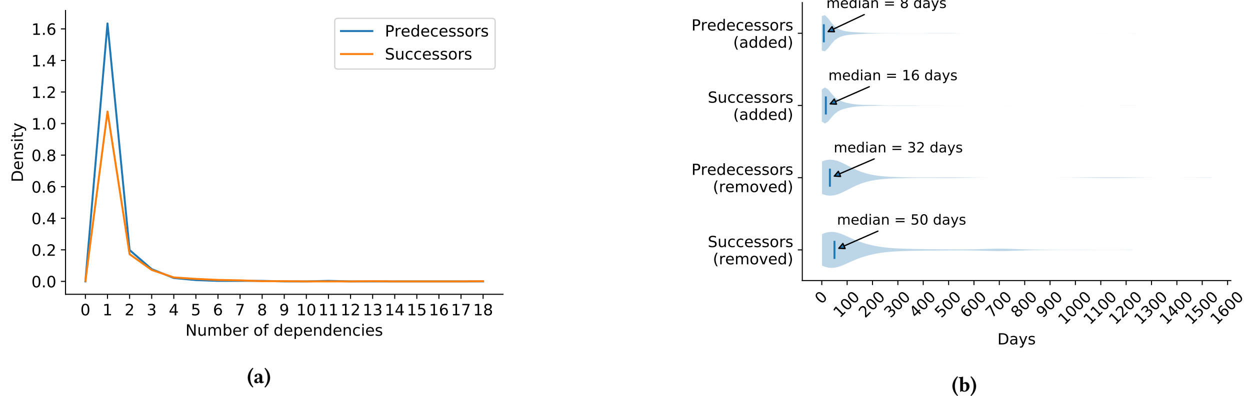

Figure 4: a) Density showing the number of dependencies per user story; b) distribution of dependencies w.r.t. to their duration.

Table 2: Stages for a user story in Rally - adapted from [32]

Stage	Description
Idea	In this first state, business requirements are gathered for the particular user story. No development effort is required at this stage.
Defined	In this stage a user story is in the team's backlog of work. Its development has not started yet.
In-Progress	A first active working state for a user story, when actual development work starts along with the cycle time clock.
Completed	This state informs that all development and testing have been finished and the work is waiting for a product owner (or a person in a similar role) to verify and approve it.
Accepted	This stage means the user story has met the acceptance criteria ("definition of done"). The accepted date is set and the cycle time clock stops.
Released	At this stage the user story is released.

5.1 RQ1: *How often and when are dependencies identified by members of ASD projects?*

We found that in the 71 investigated projects, users declared explicitly a number of dependencies. In our dataset, around 10% of user stories have declared at least one dependency (of any type). In Figure 4a, we depict the number of dependencies per user story using density as a measure of a frequency of occurrence. Most user stories with declared dependencies have only one predecessor or successor. Having over 2 predecessors or successors for a user story is very rare. The maximum number of predecessors for one user story is 11, while for successors it is 18.

We found that dependencies are identified between the same day of a creation of a user story or even up to 1238 days after the creation of a user story to which they are assigned. Mean equals almost 41 days for successors, and 35 days for predecessors. However, 25% of dependencies are created in 2 or less days. There are six different states of a user story scheduled in an iteration or release, known as *schedule states* in the Rally software (which we will refer to as stages). Their definition can be found in Table 2.

Teams identify dependencies usually in the *Idea*, *Defined* or *In-Progress* stages, as shown in a stacked bar chart in Figure 5. Our data analysis suggests, that predecessors are added in earlier stages than successors. 82.9% predecessors are added in the first two stages (i.e. *Idea* and *Defined*), while for successor this percentage is considerably lower and equals 51.5%. Yet, 17.1% predecessors are added after the *Idea* and *Defined* stages.

Summary of RQ1: Dependencies are explicitly declared for every tenth user story. More than one in six declared predecessors are added in the *In-Progress* or later stages of user stories.

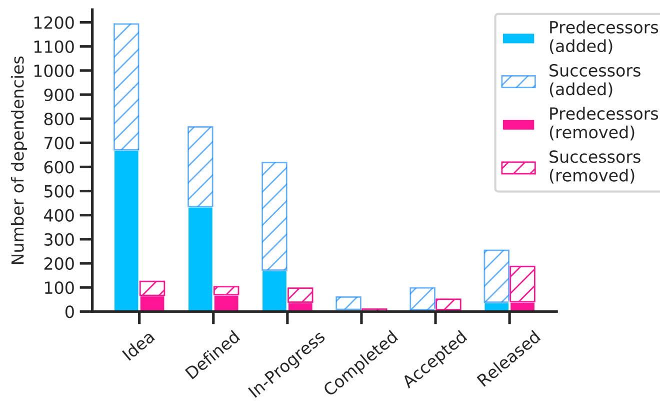

Figure 5: Number of dependencies added or removed w.r.t. stage.

5.2 RQ2: *Do team characteristics influence the identification and number of dependencies?*

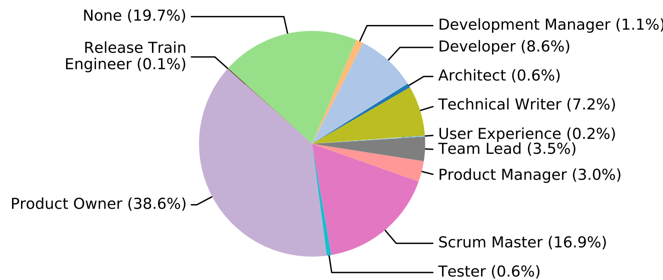

Figure 6: Distribution of user roles among dependency owners. *None* means there is no information regarding the user's role.

Users who add or remove dependencies for the user story (we refer to them as *dependency owners*) hold different roles (see Figure 6). Usually the two most active users within user stories, based on the whole activity history of the user story are dependency owners. Further, dependencies are usually created or removed either by an owner of the user story, or one of the 4 most active users for a given user story. Figure 7 illustrates this as it shows that dependency owners cover the whole spectrum of user roles in our projects and they are mostly ranked between places 1 and 4. Concretely, we can see that release train engineers who declare dependencies are usually either the second most or the fourth most active users in the user stories where they identify dependencies. On the other hand, dependency owners in the scrum master roles usually come first or second in terms of their activity among the user story contributors. The number of dependencies created by the user story creator (owner) is almost equal to the number of dependencies created by other users. Concretely, in 1737 cases a dependency is created by its owner, and in 1898 cases a dependency is created by some other user.

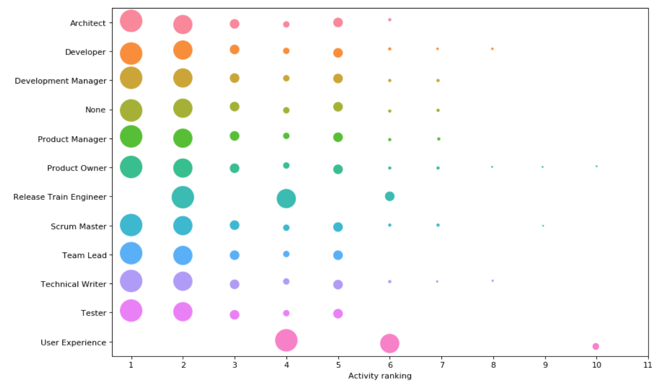

Figure 7: Distribution of dependency owners' user roles w.r.t. their activity ranking spots in user stories. The more users in a given role are classified in the respective rank, the bigger the size of the circle per role.

Furthermore, the team composition and size differ much from team to team, as illustrated in Figure 8. As shown in Figure 9, 59.1% users stories linked with a dependency belonged to the same projects (indicated as 0 in the figure). 19.3% of such user stories shared the same parent project. In other words, they belonged to different projects, but at the same level of hierarchy (indicated as 1 in the respective figure). 21.6% of user stories linked with dependencies are created within projects at different level of hierarchies, being up to 5 levels of project hierarchy away. Hence, with regard to the volume of cross-team dependencies, we identified that the biggest group is formed by user stories that are 1 or 2 levels of project hierarchy away, they constituted 37.7% of all user stories linked with dependencies. In ASD, physical location of team members plays an important role. In principle, co-located teams, where functional teams work in the same workspace are preferable. However, due to several reasons (size of an organization, costs etc.) a company following Agile principles may decide to work in a distributed setting. Members of the analyzed projects worked from 3 different countries in 3 different continents.

Summary of RQ2: Over a half of the dependencies belong to the same project, hence cross-team dependencies are not prevalent. However, if they are declared, they tend to belong to related projects (e.g. having the same parent project). Team compositions may differ from team to team, but dependencies are usually indicated by product owners, scrum masters or developers. Users identifying dependencies are one of the most active contributors to users stories with declared dependencies.

5.3 RQ3: *How often dependencies are closed before the iteration ends?*

Dependencies can be closed in two ways: either the user does it explicitly by removing the dependency, or the user story is closed. In the latter case, the most frequent scenario assumes that a dependency is closed as a consequence of a user story (to which it belongs) being completed. To summarize, a stacked bar chart in

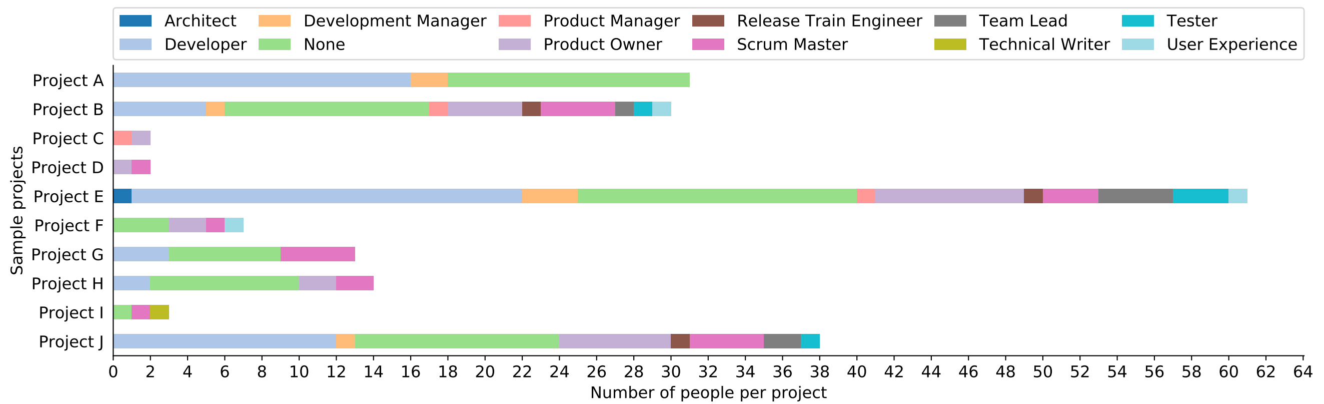

Figure 8: Example of team compositions. *None* means there is no information regarding the user's role.

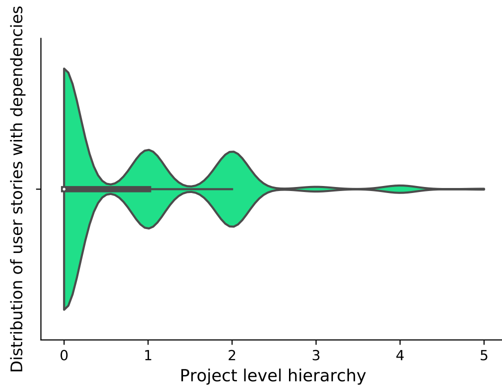

Figure 9: Distribution of user stories with dependencies across projects w.r.t. the project level hierarchy. The numbers on the x-axis should be interpreted as follows: 0 - user stories belong to the same projects; 1 - user stories belong to different projects but share parent projects (they are 1 level of project hierarchy away); 2 - user stories are 2 levels of project hierarchy away, etc.

Figure 5 shows that the total number of created dependencies is considerably higher than the number of removed dependencies. Now, let us focus on the first case, where a user deliberately removes the dependency. Our data analysis unfolded that dependencies are removed between the same day of a creation of a user story or even up to 1535 days after the creation of a user story to which they are assigned. Mean equals almost 121 days for successors, and 80 days for predecessors. However, 25% of dependencies are removed between 8 and 15 days, see violin plots in Figure 4b for more details. As it can be seen in Figure 5, predecessors are predominantly removed in the *Defined* and *Idea* stages, and to lesser extent in the *In-Progress* stage. Successors, on the other hand, are removed much more often in the *Released* stage than predecessors.

Summary of RQ3: Predecessors are mostly removed in the first two stages of a user story to which they are assigned, while successors are removed mainly in the last stage.

6 DISCUSSION AND IMPLICATIONS

In this section, we offer our explanation of the results and suggest implications drawn from our analysis.

We found that a larger number of successors is identified in late stages (such as *Completed, Accepted, Released*) than predecessors. This difference can be interpreted in two ways: (i) before the actual development work starts predecessors may be easier to identify than successors, because the latter may not exist yet; (ii) a user story may be created as a byproduct of another user story (e.g. breaking down and isolating scope of user stories), hence one knows the newly created user story is linked to the latter from the start. In the same vein, in the *In-Progress* stage 27.0% successors are added. This can be perhaps explained by the fact that while a user story progresses through development phases, new user stories emerge which happen to be dependent and naturally they become successors of the user story at hand. Importantly, in the *In-Progress* stage 12.9% predecessors are declared. It is worth mentioning that in general, an implementation of a predecessor user story must be completed before work on a successor user story can begin. Hence, predecessors added in user stories for which development has already started, can be considered a violation of this rule. We could interpret a predecessor declaration for a user story in the *In-Progress* or later stages as delayed. The nature of such late declarations remains to be clarified and could be a stepping stone to uncovering hidden (implicit) dependencies.

Moreover, our results suggest that users – who are frequent contributors and possibly have the best understanding of the nature of a work item at hand – are most likely candidates to identify the dependencies and have knowledge of their impact on other user stories.

The ratio between dependencies created and removed by users suggests that teams focus more on the creation of a dependency in a user story than the removal. Nevertheless, as predecessors are usually removed in the *Defined* and *Idea* stages, we conclude that

this is a desired behavior. Furthermore, our data analysis shows that there are several predecessors removed at later stages, which suggests that either user story contributors forgot to remove predecessors in a timely manner or started working on the successor user story while the implementation of the predecessor user story is still not completed. Noteworthy, the predecessor-successor relationship type is the only type of dependency that can be applied in the investigated ALM tool. This dependency type imposes a relatively strict rule on the order and timeline of the user story completion. For instance, other tools [31] consider more types of dependencies (with varying levels of completion time strictness).

Implication #1: The large majority of user stories do not have declared dependencies. Yet, the volume of unidentified dependencies is not known. It remains as an important and open problem, calling for further investigation by researchers as at the moment it is not a well-researched topic.

Implication #2: The declaration of dependencies is frequently made by people familiar with user stories and within the same project. It remains to be clarified if the knowledge sharing practices within SAFe teams ensure that the scope of work in other projects is not dependent.

Implication #3: ALM monitoring data turns out to be a promising data source, yet it has received very limited attention in research so far. Researchers should strive to exploit more diverse set of data sources [3], including ALM data.

7 THREATS TO VALIDITY

In this section, we discuss possible threats that may have a negative impact on the results of this study.

Threats to internal validity: ALM systems track more information that we could process including complex relations between defects, features and tests. The analysis of the latter data are neither the subject of this study nor we found a good reason to include them. Nevertheless, as we could not analyze all possible sources of information that could carry a dependency threat, we add this caveat that the dataset of dependency-related data identified by us may not be exhaustive. Also, data for several artifacts is incomplete or there is noise introduced through data quality issues.

Threats to external validity: We however acknowledge that our dataset may not be representative of all kinds of software projects, including commercial settings (along with open source projects, which are similar to commercial projects in many aspects). However, to minimize this threat, we selected a large pool of long-running projects – over 70 projects maintained for at least 5 years. Both the ALM tool and the company which data we used may not be representative of how other companies use that specific tool or, in general, cope with dependencies. Further investigation to verify our findings using other industrial as well as open source projects would be desirable. Also, in this work we focus on an organization that follows the SAFe methodology in its projects. Carrying out a study on projects that use other forms of large-scale ASD than SAFe would be beneficial. Further, we study the usage of the Rally software, which – although popular among large-scale practitioners [39] – may not represent the whole spectrum of ALM tools and their usage. This may be especially evident in the case of dependency indication. Numerous factors may impact the usage of the dependency detection feature in the software (e.g. knowledge of the tool, or the usability of user interface and user experience). Hence, our results, which are based on the usage of a concrete tool, may be skewed. However, the usage data covered in this study is collected from teams working in the company that developed the said software, hence their understanding of the tool should be sufficient to use its features (e.g. dependency declaration) proficiently. Lastly, due to limited access to Rally's software development teams, this study does not use qualitative input to interpret the results from the practical perspective. Yet, two authors of this paper used to work at CA, and they were closely working with Rally's software development teams, which helped in understanding company's datasets and tools.

8 CONCLUSION AND FUTURE WORK

Work coordination in large-scale ASD is a non-trivial task and its complexity considerably increases with dependencies. Yet, the scientific evidence on large-scale Agile implementations is very limited. For instance, SAFe (used by the studied company), while maintaining the top position in Agile scaling practices for some years now [39], still lacks empirical evaluation and academic research describing it [12, 20]. In this work, we provide scientific evidence on how the framework is adopted in the industry, focusing on the particular case of dependency management. For that purpose, we studied over 70 industrial, large-scale Agile projects using quantitative methods. Our paper also contributes to the MSR field, as it studies software development patterns through data-informed analysis of software repositories. Importantly, we utilize a relatively rarely used source of data, i.e. an ALM tool. Furthermore, in this work, we show that the problem of dependency management should be viewed through the lens of coordination theory. Self-management of software teams is an inherent characteristic of ASD, but is also a major challenge at scale. We are of the opinion that automatic detection of dependencies can help teams to largely preserve their authority as to how they organize their own work, and hence eliminate the need for incorporating more strict and traditional mechanisms of coordination at the organization level. Furthermore, understanding implicit dependencies and being able to uncover them could considerably help project teams in managing dependencies. Although in our work we only focus on the explicitly indicated dependencies through the dependency attribute, descriptions of user stories and comments provided in natural language leave room for future research.

ACKNOWLEDGEMENT

This work is supported in part by the Catalan Agencia de Gestión de Ayudas Universitarias y de Investigación (AGAUR) through the FI PhD grant and the project 2017 SGR 01694. The research is also partially supported by the Spanish Ministerio de Economía, Industria y Competitividad through the GENESIS project (grant TIN2016-79269-R). We would like to thank the anonymous reviewers and Broadcom employees for their insightful comments on the earlier version of this paper.

REFERENCES

[1] Scott W Ambler and Mark Lines. 2012. *Disciplined agile delivery: A practitioner's guide to agile software delivery in the enterprise.* IBM press.

[2] Victor R Basili and H Dieter Rombach. 1988. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on software engineering* 14, 6 (1988), 758–773.

[3] Katarzyna Biesialska, Xavier Franch, and Victor Muntés-Mulero. 2020. Big Data analytics in Agile software development: A systematic mapping study. *Information and Software Technology* 132 (2020), 106448.

[4] Kelly Blincoe, Giuseppe Valetto, and Daniela Damian. 2013. Do all task dependencies require coordination? the role of task properties in identifying critical coordination needs in software projects. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* 213–223.

[5] Gianluigi Caldiera, Victor R Basili, and H Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.

[6] Pär Carlshamre, Kristian Sandahl, Mikael Lindvall, Björn Regnell, and J Natt och Dag. 2001. An industrial survey of requirements interdependencies in software product release planning. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering.* IEEE, 84–91.

[7] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2015. Predicting delays in software projects using networked classification. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 353–364.

[8] John W Creswell. 2009. *Research design: Qualitative, quantitative, and mixed methods approaches.* Sage publications.

[9] Kevin Crowston. 1997. A Coordination Theory Approach to Organizational Process Design. *Organization Science* 8 (1997), 157–175.

[10] Karina Curcio, Tiago Navarro, Andreia Malucelli, and Sheila Reinehr. 2018. Requirements engineering: A systematic mapping study in agile software development. *Journal of Systems and Software* 139 (2018), 32–50.

[11] Åsa G Dahlstedt and Anne Persson. 2005. Requirements interdependencies: state of the art and future challenges. *Engineering and managing software requirements* (2005), 95–116.

[12] Kim-Karol Dikert, Maria Paasivaara, and C. Lassenius. 2016. Challenges and success factors for large-scale agile transformations: A systematic literature review. *Journal of Systems and Software* 119 (2016), 87–108.

[13] Torgeir Dingsøyr, Tor Erlend Fægri, and Juha Itkonen. 2014. What is large in large-scale? A taxonomy of scale for agile software development. In *International Conference on Product-Focused Software Process Improvement.* Springer, 273–276.

[14] Torgeir Dingsøyr and Nils Brede Moe. 2014. Towards principles of large-scale agile development. In *International Conference on Agile Software Development.* Springer, 1–8.

[15] Torgeir Dingsøyr, Nils Brede Moe, Tor Erlend Fægri, and Eva Amdahl Seim. 2018. Exploring software development at the very large-scale: a revelatory case study and research agenda for agile method adaptation. *Empirical Software Engineering* 23, 1 (01 Feb 2018), 490–520.

[16] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering.* Springer, 285–311.

[17] Martin S Feather, Steven L Cornford, and Mark Gibbel. 2000. Scalable mechanisms for requirements interaction management. In *Proceedings Fourth International Conference on Requirements Engineering. ICRE 2000.(Cat. No. 98TB100219).* IEEE, 119–129.

[18] Xavier Franch, Daniel Mendez, Andreas Vogelsang, Rogardt Heldal, Eric Knauss, Marc Oriol, Guilherme Travassos, Jeffrey Clark Carver, and Thomas Zimmermann. 2020. How do Practitioners Perceive the Relevance of Requirements Engineering Research? *IEEE Transactions on Software Engineering* (2020), 1–1.

[19] Jin Guo, Mona Rahimi, Jane Cleland-Huang, Alexander Rasin, Jane Huffman Hayes, and Michael Vierhauser. 2016. Cold-start software analytics. In *Proceedings of the 13th International Conference on Mining Software Repositories.* 142–153.

[20] Rashidah Kasauli, Eric Knauss, Jennifer Horkoff, Grischa Liebel, and Francisco Gomesde Oliveira Neto. 2021. Requirements engineering challenges and practices in large-scale agile system development. *Journal of Systems and Software* 172 (2021), 110851.

[21] Lina Lagerberg, Tor Skude, Pär Emanuelsson, Kristian Sandahl, and Daniel Ståhl. 2013. The Impact of Agile Principles and Practices on Large-Scale Software Development Projects: A Multiple-Case Study of Two Projects at Ericsson. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement.* 348–356.

[22] Craig Larman and Bas Vodde. 2016. *Large-scale scrum: More with LeSS.* Addison-Wesley Professional.

[23] Dean Leffingwell, D Jemilo, M Zamora, C ONeill, and A Yakuma. 2014. Scaled agile framework (SAFe). *Haettu* 27 (2014), 2014.

[24] Jun Lin, Han Yu, Zhiqi Shen, and Chunyan Miao. 2014. Studying Task Allocation Decisions of Novice Agile Teams with Data from Agile Project Management Tools. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14).* ACM, New York, NY, USA, 689–694.

[25] Thomas W Malone and Kevin Crowston. 1994. The interdisciplinary study of coordination. *ACM Computing Surveys (CSUR)* 26, 1 (1994), 87–119.

[26] Zainab Masood, Rashina Hoda, and Kelly Blincoe. 2020. How agile teams make self-assignment work: a grounded theory study. *Empirical Software Engineering* 25, 6 (2020), 4962–5005.

[27] Nils Brede Moe, Torgeir Dingsøyr, and Knut Rolland. 2018. To schedule or not to schedule? An investigation of meetings as an inter-team coordination mechanism in large-scale agile software development. (2018).

[28] Mirosław Ochodek and Sylwia Kopczyńska. 2018. Perceived importance of agile requirements engineering practices–a survey. *Journal of Systems and Software* 143 (2018), 29–43.

[29] Maria Paasivaara, Benjamin Behm, Casper Lassenius, and Minna Hallikainen. 2018. Large-scale agile transformation at Ericsson: a case study. *Empirical Software Engineering* (11 Jan 2018).

[30] Maria Paasivaara, Casper Lassenius, and Ville T Heikkilä. 2012. 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.* 235–238.

[31] Mikko Raatikainen, Quim Motger, Clara Marie Lüders, Xavier Franch, Lalli Myllyaho, Elina Kettunen, Jordi Marco, Juha Tiihonen, Mikko Halonen, and Tomi Männistö. 2021. Improved dependency management for issue trackers in large collaborative projects. arXiv:2102.08485 [cs.SE]

[32] Rally Software. 2018. What does each Schedule State mean for a story in Rally? https://community.broadcom.com/communities/community-home/digestviewer/viewthread?MID=764423#bm9055b0b6-994a-4324-aeb7-acc04fb7489e Online; accessed 15 February 2021.

[33] William N Robinson, Suzanne D Pawlowski, and Vecheslav Volkov. 2003. Requirements interaction management. *ACM Computing Surveys (CSUR)* 35, 2 (2003), 132–190.

[34] Knut H Rolland, Vidar Mikkelsen, and Alexander Næss. 2016. Tailoring agile in the large: Experience and reflections from a large-scale agile software development project. In *International Conference on Agile Software Development.* Springer, Cham, 244–251.

[35] Alexander Scheerer, Saskia Bick, Tobias Hildenbrand, and Armin Heinzl. 2015. The Effects of Team Backlog Dependencies on Agile Multiteam Systems: A Graph Theoretical Approach.. In *HICSS*, Tung X. Bui and Ralph H. Sprague Jr. (Eds.). IEEE Computer Society, 5124–5132.

[36] Nelson Sekitoleko, Felix Evbota, Eric Knauss, Anna Sandberg, Michel Chaudron, and Helena Holmström Olsson. 2014. Technical dependency challenges in large-scale agile software development. In *International conference on agile software development.* Springer, 46–61.

[37] Junji Shimagaki, Yasutaka Kamei, Naoyasu Ubayashi, and Abram Hindle. 2018. Automatic topic classification of test cases using text mining at an Android smartphone vendor. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 1–10.

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

[39] VersionOne. 2018. The 12th annual state of agile report. https://explore.versionone.com/state-of-agile/versionone-12th-annual-state-of-agile-report Online; accessed 25 March 2020.

[40] Bill Wake. 2003. INVEST in Good Stories, and SMART Tasks. 2003. *URL: xp123.com/articles/invest-in-good-stories-and-smart-tasks* (August 2003).

[41] Claes Wohlin, Martin Höst, and Kennet Henningsson. 2003. Empirical research methods in software engineering. In *Empirical methods and studies in software engineering.* Springer, 7–23.

[42] Denise M Woit. 2005. Requirements interaction management in an extreme programming environment: a case study. In *Proceedings of the 27th international conference on Software engineering.* 489–494.

[43] Wei Zhang, Hong Mei, and Haiyan Zhao. 2005. A feature-oriented approach to modeling requirements dependencies. (2005), 273–282.