From Concept to Action: How Do Agile Software Teams Decompose User Stories into Tasks?

Delina Ly[1,2]✉[0000−0002−7972−7530], Fatma Başak Aydemir[2][0000−0003−3833−3997], and Fabiano Dalpiaz[2][0000−0003−4480−3887]

[1] Application Lifecycle Management, VX Company, The Netherlands
dly@vxcompany.com
[2] Department of Information and Computing Sciences, Utrecht University, The Netherlands
{f.b.aydemir, f.dalpiaz}@uu.nl

Abstract. In Agile software development, requirements are often captured as user stories (USs) and stored in issue tracking systems (ITS). These USs are typically manually decomposed into tasks that define the necessary steps for their implementation. This decomposition process, which serves as a bridge between requirements engineering and software development, is rarely studied in the existing literature, and empirical knowledge is scarce. We investigate how teams perform user story (US) decomposition in a mid-sized Dutch software development organization. Through content analysis and descriptive statistics, we analyze the decomposition of 373 USs into 1,823 tasks across 10 software systems developed with three different technologies (Java, .NET, and Power Apps). We provide empirical insights into the US decomposition process by examining which roles are involved, how they are involved, and which techniques are used. These findings can inform the development of tools to automate US decomposition and guide further research.

Keywords: Agile Software Development · Requirements Engineering · Requirements Decomposition · Refinement · Issue Tracking Systems

1 Introduction

Information systems are developed amid ongoing technological advances and evolving stakeholder needs. To respond to these changes, many software organizations have adopted agile development practices, in which requirements are specified iteratively and incrementally. These requirements are often captured as USs [5,19], which are frequently expressed through the Connextra template: *"As a <role>, I want <goal>, [so that <benefit>]"* [2,10]. We refer to USs as high-level descriptions of requirements captured in real-world ITS, which may differ from template-based representations.

USs are typically manually decomposed into concrete implementation tasks. These tasks are then assigned to team members [9,13]. We refer to the process

of identifying, refining, and rewriting tasks as *user story decomposition* [3]. In this context, we define *tasks* as concrete activities in ITS. This decomposition helps identify missing tasks and supports more reliable effort estimation [2].

The importance of US decomposition grows as software development practices evolve and technologies such as Large Language Models (LLMs) become more widely adopted. USs are often too abstract to serve as direct inputs for generating useful code, whereas the derived tasks provide a more suitable level of detail for LLM-assisted software development [16].

Despite the importance of US decomposition, empirical knowledge about how practitioners perform this activity remains limited. Previous research has focused on splitting large USs into smaller USs, e.g. [3,10], leaving the finer-grained decomposition largely unexplored. Müter *et al.* [13] provide an initial perspective by analyzing the linguistic structure of the developer-written task labels. We extend this perspective by analyzing the decomposition of USs into tasks as an activity that involves multiple roles within software teams [2].

To address this gap, we conduct an empirical study of how teams decompose USs into tasks at VX Company, a mid-sized Dutch software organization [18].

We report descriptive statistics and conduct content analysis by coding how 373 USs were decomposed into 1,823 tasks across 10 systems developed in Java, .NET, and Power Apps. We pose the following research questions (RQ):

RQ1 Which roles are involved in decomposing user stories into tasks?
RQ2 How are roles involved in decomposing user stories into tasks?
RQ3 What techniques do teams use to decompose user stories into tasks?

This paper provides an empirical analysis of US decomposition practices across 10 software systems in industry. Our results show that US decomposition is performed predominantly by developers and project managers, and typically involves only a single role. Despite the variety of techniques proposed in the literature, practitioners rely on only a small subset, primarily workflow steps and horizontal splitting. This finding reveals a pronounced gap between prescribed and practiced approaches in contemporary Agile development.

The remainder of the paper is organized as follows. Section 2 describes the research setting and method. Section 3 presents and interprets the results. Section 4 provides further discussion of the results, presents comparison to related work, implications, and threats to validity. Finally, Section 5 concludes the paper.

2 Research method

We describe the research setting (Section 2.1) and the data sampling, collection, and preprocessing procedures (Section 2.2). We then outline the descriptive statistical analysis (Section 2.3) and the content analysis approach (Section 2.4).

[3] Other authors refer to this process as *splitting* [3] or *disaggregation* [2]. We use the term *decomposition*, as it aligns with standard software engineering terminology.

2.1 Research setting

We conducted the study at VX Company, a Dutch software organization with approximately 200 employees that develops and maintains customized software solutions. The organization's development activities are organized around three technologies: Java and .NET for high-code software development, and Microsoft Power Apps for low-code development. Azure DevOps is used as the ITS in which requirements are captured as USs and decomposed into tasks. The USs and tasks are written in Dutch, English, or both. We used convenience sampling, since the first author is employed by the organization.

The organization operates with three core teams, each aligned with a different technology. For each sprint, team members are selected from these teams based on availability and expertise required to implement USs. Consequently, team composition varies from sprint to sprint. Teams follow an industry-adapted version of Agile. Each team consists of multiple roles (Table 1) and has an average size of 5.5 members. In some systems, the client serves as the product owner.

Table 1: Team roles and assigned responsibilities in the organization

Role	**Assigned responsibilities**
Product owner	Writes and prioritizes USs, and performs testing to ensure US implementations meet the acceptance criteria.
Project manager	Coordinates planning, monitors progress, and facilitates interaction between development teams and stakeholders.
Business analyst	Elicits and clarifies requirements, writes and refines USs, and acts as a bridge between business and development teams.
Architect	Defines the high-level structure of software systems, oversees design decisions, and provides technical guidance.
Developer	Writes code and unit tests to implement USs and consults with the product owner to clarify requirements when necessary.
Tester	Validates software functionality by executing test cases and identifying defects, particularly in larger systems.
DevOps engineer	Automates and optimizes development and deployment processes, and builds and maintains the infrastructure.

2.2 Data sampling, collection, and preprocessing

We analyzed 10 systems selected by the organization. We used purposive sampling to select project managers and developers involved in these systems. We contacted one project manager and one developer from each core team via email and asked them to identify three sprints they considered representative of their typical approach to US decomposition and overall development activity. We extracted USs and their associated tasks from the selected sprints in Azure DevOps. Table 2 provides an overview of the systems, technologies, team sizes, and the numbers of included and excluded USs and tasks. In the systems S3, S7, and S10, USs spanning multiple sprints were considered a single item in the analysis.

Table 2: Overview of the systems' raw and selected sprint data.

Sys. ID	Tech.	Team Size	User Stories				Tasks			
			Raw	Sprints	Incl.	Excl.	Raw	Sprints	Incl.	Excl.
S1	Java	8	2794	59	50	9	8175	316	238	78
S2		4	591	17	13	4	1201	72	56	16
S3		5	244	46	27	19	440	157	67	90
S4	.NET	12	798	108	98	10	4002	552	507	45
S5		3	640	14	13	1	1678	37	26	11
S6		4	736	21	21	0	3086	123	70	53
S7		6	403	45	42	3	1684	305	222	83
S8		4	387	26	21	5	1065	74	60	14
S9	Power Apps	7	421	68	65	3	2202	420	420	13
S10		2	94	23	23	0	610	190	157	33

For each US and task, we extracted the fields required to address our research questions (Table 3). We excluded USs without tasks and release-related USs titled *"Release <release number>"*, as they do not contain requirements descriptions and consist only of procedural tasks (e.g., Test, Acceptance, Production) that do not represent US decomposition. In ITS, *work items* represent trackable units of work in the software development process, including USs and tasks. Java teams use the User Story work item type to capture releases, whereas .NET and PowerApps teams use a dedicated Release work item type. This leads to more release-related USs and their tasks being excluded for Java teams.

We structured the remaining USs and tasks and imported them into MAXQDA. We validated the individuals and their roles in the sprints with project managers, developers, and a DevOps engineer. These assigned roles were validated by a manager overseeing all teams, who resolved any inconsistencies.

Table 3: US and task fields extracted from Azure DevOps.

Category	**Field**	**Description**
General	ID	Unique identifier of the work item.
	Title	Short textual summary of the work item.
	Description	Detailed explanation of the work item (optional).
	Created By	Name of the creator of the work item.
User Story	Acceptance Criteria	Conditions that must be satisfied for US completion (optional).
	Technical Impact	Expected effects of implementing the US, such as affected components, dependencies, and architectural changes (optional).
	Number of Tasks	Number of tasks linked to the US.
Task	Parent	Identifier of the US to which the task is linked.

2.3 Descriptive statistics

We used descriptive statistics to address RQ1 and RQ2. For RQ1, we computed the number of tasks created by each individual per system and aggregated these counts by role. We visualized the role distributions in a bar chart to examine which roles are involved in US decomposition. For RQ2, we computed the number of tasks created by each individual for each US per system and aggregated these counts by role and US. We visualized role involvement using UpSet plots to examine how the roles are involved in the decomposition of USs into tasks.

2.4 Content analysis

We use deductive content analysis approach, complemented by inductive coding to address RQ3. Given the limited research on task-level decomposition, we constructed a priori coding scheme derived from on US decomposition techniques (Table 4), which are commonly used to split large USs into smaller ones [3,6,8]. This approach enables us to systematically examine whether these techniques also apply when decomposing USs into tasks. The unit of analysis is the set of tasks linked to each US. A US may be assigned multiple codes, while each task is assigned to a single code. All coding was performed by a single author; this limitation is discussed in Section 4.4.

Table 4: Coding scheme for US decomposition techniques.

Technique	**A US is decomposed by**
Horizontal splitting	Separating work across architectural layers, requiring multiple slices to be completed to deliver an increment of value.
Vertical splitting	Separating work into end-to-end slices across all architectural layers, each delivering a complete increment of value.
Workflow steps	Separating work into end-to-end workflow slices.
Business rule variations	Separating work by different business rules or conditions.
CRUD operations	Separating work into create, read, update, and delete operations.
Major effort	Separating work into in high- and lower-effort parts.
Variations in data	Separating work by data or content variations.
Data entry methods	Implementing a simple user interface first and separating more complex variants.
Simple/Complex	Implementing a simple version first and separating complex variations.
Defer performance	Implementing a working version first and deferring non-functional improvements.
Break out a spike	Performing exploratory work to reduce uncertainty.

3 Results and interpretation

We report the results of the analyses. Each subsection presents the findings for the corresponding research question based on aggregated data across all systems. The findings are presented in gray blocks and labeled in ascending numeric order. We then examine whether these findings also hold for the individual systems.

3.1 RQ1: Which roles are involved in decomposing USs into tasks?

We report the percentage distribution of roles involved in US decomposition to facilitate comparison across systems (Figure 1).

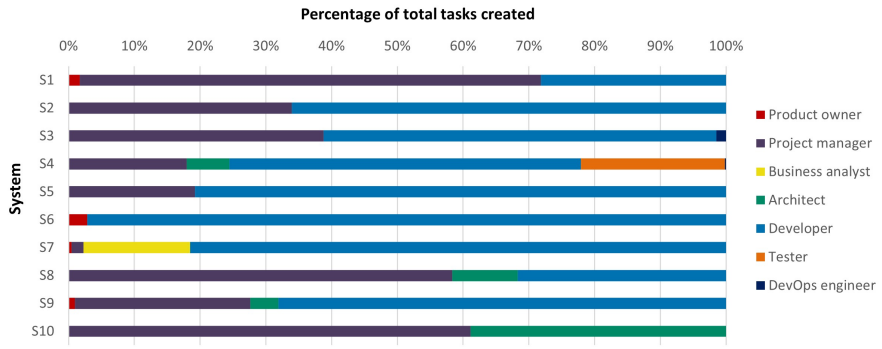

Fig. 1: Distribution of the roles involved in US decomposition across the ten studied systems.

Finding 1: US decomposition is performed primarily by developers (54.3%) and project managers (30.4%). Contributions from architects (6.5%), testers (6.1%), business analysts (2.0%), product owners (0.6%), and DevOps engineers (0.1%) were limited and occurred when USs required their expertise.

Developers were the primary contributors to decomposing USs into tasks. The sole exception was S10, where no developer-created tasks were recorded, as the project manager concurrently served as a developer. Project managers were also involved in many teams, indicating that US decomposition is not exclusively performed by technical roles. However, they were not involved in S6, possibly because the requirements were already well-defined.

Participation from other roles was limited across teams and occurred primarily when USs required their expertise. In most teams, the product owner writes the USs, the project manager facilitates communication, and developers clarify

requirements directly with the product owner, which reduces the need for business analyst involvement. S7 deviates from this pattern, as the project manager shows minimal involvement, which may have required the business analyst to assume a more prominent role in supporting US decomposition. The variation in architects' involvement across teams suggests that they contributed where system complexity or architectural constraints required their input.

Testers were involved in US decomposition in S4, suggesting a stronger emphasis on testing and quality assurance. The system included security-related requirements, increasing the need for dedicated testing expertise. In the other teams, product owners conduct acceptance testing and developers perform unit testing, which reduces the need for tester involvement.

DevOps engineers were minimally involved in US decomposition, possibly because their work is primarily managed through a separate backlog. Their role is primarily to enable the efficient development, testing, and deployment of information systems. Thus, they are not typically involved in US decomposition unless USs address infrastructure or deployment-related requirements.

Prior research positions developers as responsible for decomposing USs into tasks [2,13]. Our findings suggest a shared responsibility between developers and project managers, which may also be influenced by organization-specific responsibilities. The limited involvement of other roles indicates that US decomposition is primarily centralized among developers and project managers in our context.

3.2 RQ2: How are roles involved in decomposing USs into tasks?

To address RQ2, we divide the research question into two sub-questions:

> **RQ2.1.** How many roles are involved in decomposing a user story into tasks?
> **RQ2.2.** Which combinations of roles are involved in decomposing a user story into tasks?

Figure 2 presents an UpSet plot of role involvement in US decomposition across systems. The plot consists of two components: (1) a bar chart at the top showing the number and percentage of USs, and (2) a matrix below, where each row represents a role. A filled circle in a column indicates that the role created one or more tasks for a US, and lines connecting circles indicate that multiple roles contributed to US decomposition.

RQ2.1 How many roles are involved in decomposing a US into tasks?

> **Finding 2:** US decomposition is predominantly performed by a single role (59.5%). Two-role involvement is the second most frequent (33.5%), while three-role and four-role involvement are infrequent (6.7% and 0.3%).

Finding 2 holds for most systems except for S1 and S4, which deviate from this pattern. We report the results for these systems. The remaining system-specific results are provided in the online appendix [11].

In S1, US decomposition is performed by a single role in 44% of USs, while two-role involvement is more common at 50%, with three-role involvement accounting for the remaining 6%. This distribution differs from the overall trend, as S1 shows a higher prevalence of two-role than single-role involvement.

S4 shows the most diverse role involvement (Figure 3). In S4, US decomposition is performed by a single role in 24.5% of USs, while two-role involvement accounts for 55.1%, three-role involvement for 19.4%, and four-role involvement for 1%, indicating broader multi-role involvement than in other systems.

RQ2.2 Which combinations of roles are involved in decomposing a US into tasks?

> **Finding 3:** For USs that are decomposed into tasks by a single role (N=222), developers account for the largest share (54.1%), followed by project managers (38.3%), architects (7.2%), and DevOps engineers (0.5%).

Finding 3 provides further insight into Findings 1 and 2 by showing that, in single-role US decomposition, developers and project managers most often perform the decomposition independently (Figure 2). Single-role involvement by architects and DevOps engineers is infrequent.

In particular, for half of the systems (S5, S6, S7, S9, S10) the developers often perform US decomposition independently, while for the remaining systems

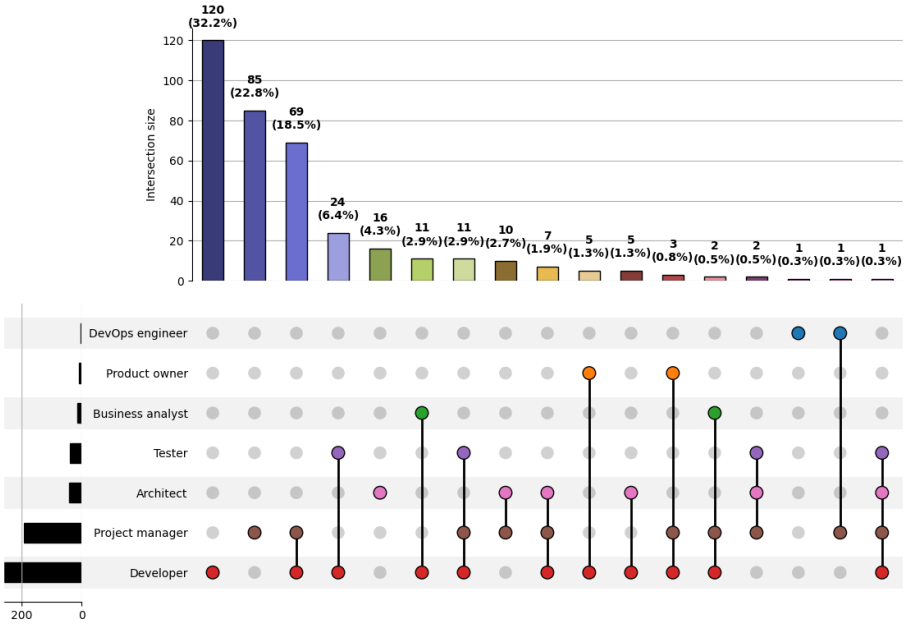

Fig. 2: Role involvement in US decomposition across systems.

(S1, S2, S3, S4, S8), project managers most often decompose USs independently. This pattern is consistent with the overall finding that developers and project managers are the primary roles involved in single-role US decomposition.

Finding 4: In two-role involvement (N=125), the developer–project manager pair is most frequent (55.2%). Other pairs are less frequent: developer–tester (19.2%), developer–business analyst (8.8%), project manager–architect (8.0%), developer–product owner (4.0%), and developer–architect (4.0%). The project manager–DevOps engineer pair is rare (0.8%).

Finding 4 provides additional insight into Findings 1 and 2 by showing that, in two-role US decomposition, the developer–project manager pair is the most frequent. Most other two-role combinations involve developers paired with another role, except for DevOps engineers. The remaining pairs involve project managers: project manager-architect and project manager-DevOps engineer. This pattern suggests that developers and project managers remain central to US decomposition, even when other roles are involved.

This finding holds partly for the systems in which the developer-project manager pair is the most frequent (S1, S2, S3, S5, S8, S9). Among these systems, the

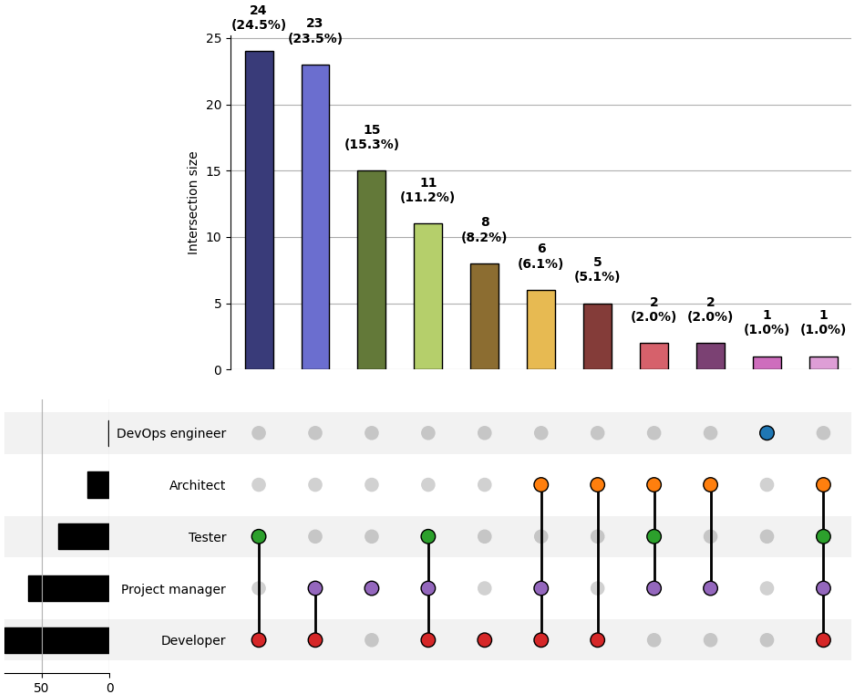

Fig. 3: Role involvement in US decomposition in S4

second most frequent pairs are developer-product owner (S1, S9), and project manager-DevOps engineer (S3). The other systems have one two-role pair.

In contrast, Finding 4 does not hold for the remaining systems (S4, S6, S7, S10). In S4 (N=54), the leading two-role combination is developer-tester (44.4%), followed by developer-project manager (42.6%), developer-architect (9.3%), and project manager-architect (3.7%). This distribution may be attributed to the system's emphasis on testing and quality assurance, as reflected in the prominence of the developer–tester pair.

In S6, the only two-role pair is developer–product owner. In S7 (N=12), the leading pairs are developer–business analyst (91.7%) and developer–product owner (8.3%), possibly due to the project manager's minimal involvement. In S10, the only two-role pair is project manager–architect.

These deviations suggest that system-specific needs shape two-role involvement. Unlike the systems where Finding 4 holds, these systems show more diverse pairing patterns rather than a consistent reliance on the developer–project pair.

Finding 5: USs are rarely decomposed by three or more roles. Three-role involvement (N=25) consists of developer–project manager–tester (44.0%), developer–project manager–architect (28.0%), developer–project manager–product owner (12.0%), developer–project manager–business analyst (8.0%), and project manager–tester–architect (8.0%). Four-role involvement occurs only once: developer–project manager–architect–tester.

Finding 5 shows that three or four-role involvement in US decomposition is uncommon, and when it occurs, typically centers on developers and project managers. These role combinations further reinforce the central role of developers and project managers in US decomposition. In six of the ten systems, there is no three-role or four-role involvement (S2, S3, S5, S6, S8, S10).

In three of the four remaining systems, three-role involvement is limited to a single combination: developer–project manager– product owner in S1, developer –project manager–business analyst in S7, and developer–project manager –architect in S9. S4 has the most three-role combinations (N=19), including developer–project manager–tester (57.9%), developer–project manager–architect (31.6%), project manager–tester–architect (10.5%). There was one instance of four-role involvement: developer–project manager–tester– architect.

These patterns suggest that three or four-role involvement is generally minimal and highly context-specific. S4 stands out as an exception, suggesting that cross-functional involvement is more likely in systems with higher complexity or stronger quality assurance requirements.

Team characteristics, such as culture and team dynamics, may influence the involvement patterns observed in Findings 2-5 [3]. The frequent developer–project manager pairing suggests a culture in which responsibility for decomposition is concentrated within a subset of roles rather than distributed across the team. In addition, personal preferences may influence US decomposition, as some team members favor minimal decompositions while others prefer more detailed ones.

3.3 RQ3: What techniques do teams use to decompose USs into tasks?

We report our results using the coding scheme described in Section 2.4, extended with two codes that emerged from the data: *acceptance criteria*, referring to conditions that must be satisfied for US completion, and *technical impact*, describing expected technical effects of US implementation. Figure 4 presents the percentage distribution of decomposition techniques and artifacts across systems.

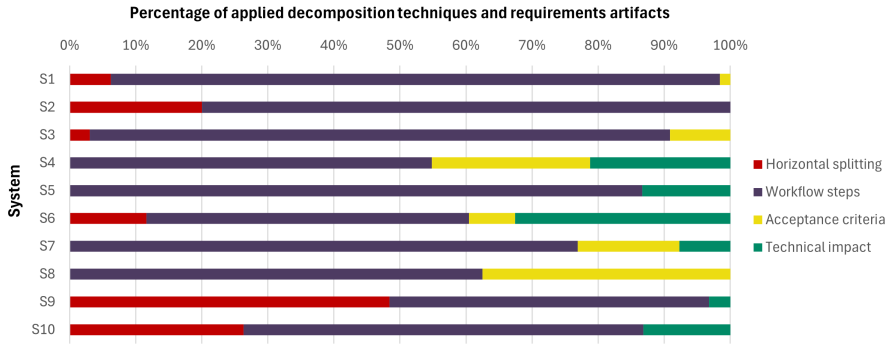

Fig. 4: Distribution of techniques and requirement artifacts across systems.

Finding 6: Among the 11 techniques included in our coding scheme (N=719, reflecting coded labels since each US may be assigned multiple codes), only workflow steps (62.3%) and horizontal splitting (9.5%) were identified. The remaining decompositions were not captured by these techniques but were instead coded as acceptance criteria (14.7%) or technical impact (13.5%).

Workflow steps is the most frequently used technique across all systems except S9. This suggests that teams commonly decompose USs into tasks by following their development process (e.g., build–test–pull request). Thus, US decomposition may serve as a coordination mechanism, where tasks structure the workflow by supporting implementation by developers, testing by product owners or testers, and code review by other developers.

In S9, horizontal splitting is used as frequently as workflow steps. The relatively higher use of horizontal splitting in S9 and, to a lesser extent, in S10 may be influenced by the architecture of low-code platforms, where work is naturally organized into layers, screens, or configuration units. S4 deviates from this overall pattern, with a substantial share of decompositions including requirement artifacts. A possible explanation is that S4 is still under initial development and has not yet been released, which may require more upfront elaboration of requirements, whereas other systems are in maintenance.

S4 is not the only system in which decompositions include requirements artifacts: rather, it highlights a conceptual ambiguity in the notion of US decomposition. The deviations involving requirement artifacts point to an important conceptual distinction. They also indicate that a purely technique-based coding scheme did not capture all US decompositions in our dataset and that additional data-derived codes were therefore introduced.

To clarify this conceptual ambiguity, we distinguish between two types of US decomposition. The first type, *requirement-to-requirement decomposition* (or refinement), involves breaking down a requirement into parts of the same kind, such as acceptance criteria and technical impact. From an ontological perspective, this constitutes proper decomposition, as it preserves the nature of the elements, i.e., a part–whole relation among elements of the same kind [14,17]. The second type, *requirement-to-task decomposition*, refers to the operationalization of requirements into work using techniques such as workflow steps and horizontal splitting. While commonly referred to as decomposition, this transformation does not strictly qualify as decomposition, since it maps requirements to elements of a different kind, i.e., tasks. Under this lens, requirement-to-requirement constitutes proper decomposition, while requirement-to-task decomposition is more accurately described as US operationalization. Despite this distinction, we use the term US decomposition, as it is more widely used in software engineering.

In the context of requirements-to-task decomposition, we observe frequent use of action verbs in task labels, e.g., implement endpoint. This observation aligns with Müter *et al.* [3], who analyzed developer-written task labels and identified a set of action verbs. We also find that nouns are used to denote deliverables in task labels, such as test cases. These linguistic structures, which consist of verbs indicating actions and nouns indicating artifacts, may serve as the basis for new approaches to US decomposition.

4 Discussion

We provide further discussion of the results (Section 4.1), and contrast our work with previous studies (Section 4.2). We present implications for practitioners and researchers (Section 4.3), and discuss threats to validity (Section 4.4).

4.1 On the taxonomy of US decomposition techniques

Our analysis shows that most US decomposition techniques from the literature were rarely applied in our context; only workflow steps and horizontal splitting were used. One possible explanation is that USs do consistently not follow conventional templates. Another explanation is that US decomposition (leading to tasks) occurs at a different level of granularity than US splitting (leading to smaller USs), which may limit the applicability of these techniques. USs may be used at different levels of granularity [9], such that decomposing them into smaller USs might serve the same purpose as splitting USs into tasks. Moreover, US decomposition may be influenced by the accessibility of ITS. For example,

tasks may be structured differently when multiple user interactions (e.g., clicks) are required. In most commercial ITS, USs can be decomposed to (sub-)tasks.

In addition, most practitioners are not formally trained in US decomposition and may rely on intuition or ad-hoc practices. This tendency may be reinforced by time pressure to complete a sprint, leading teams to adopt pragmatic techniques to US decomposition. Consequently, techniques that require more deliberate analysis or depend heavily on system context may be applied less frequently. These results highlight a pronounced gap between the decomposition techniques prescribed in the literature and those used in practice. This points to the need to revisit existing taxonomies and to investigate whether lightweight techniques are more suitable for practitioners in Agile contexts. Future research should examine the linguistic structures of task labels, which consist of verbs indicating actions and nouns indicating artifacts, as they may form the basis for new, lightweight, and suitable approaches for US decomposition in Agile environments.

4.2 Comparison to related work

Studies on the decomposition of USs into tasks are limited. We found only two papers that propose the use of artificial intelligence for decomposing USs [15] and splitting USs [7]. However, these techniques are not evaluated in practice. There are only a few relevant studies that report on empirical research on industry practices for US decomposition, which we review in the following.

Dellsén *et al.* [3] examined US splitting through interviews and observations at five companies. They found that multiple roles were involved, with Scrum masters, product owners, and team leads most responsible for US splitting. Their study focused on US–level splitting rather than task-level decomposition and relied on interview data rather than ITS data. Similarly, we observed that US decomposition involved multiple roles, with project managers and developers most frequently. In their study, practitioners had limited formal training and primarily used vertical and horizontal splitting. In our data, vertical splitting did not occur, likely because USs were already structured as vertical slices.

Müter *et al.* [13] analyzed linguistic patterns in 1,593 developer-written task labels from a 50-employee multinational software organization in the Netherlands. They identified a set of action verbs and propose a task label template. In contrast, our dataset includes task labels written by multiple roles, including non-technical roles such as product owner and project manager. Consistent with their findings, we also observe frequent use of action verbs in task labels. In addition, we find that nouns are often used independently to denote deliverables.

Khanfor [6] studied US decomposition in crowdsourced software development by identifying whether the resulting tasks were split vertically or horizontally based on the implementation technologies associated with each task. In contrast, we examine US decomposition of Agile teams in proprietary software development. Moreover, we use a broader taxonomy of techniques derived from the literature, including, but not limited to, horizontal and vertical splitting.

US decomposition is also discussed in the context of effort estimation. Cao *et al.* [1] conducted an empirical study using task-level data and interviews to

examine how task characteristics influence effort estimation. They reported issues such as incomplete tasks, missing data, and variation in task types (e.g., feature, bug fixing, refactoring). While their focus is on effort estimation, our study examines the decomposition of USs into implementation tasks.

4.3 Implications

Implications for practitioners. Since decomposition is mostly performed by developers and project managers, teams may benefit from introducing a brief review moment in which other roles can contribute to missing tasks or role-specific insights (e.g., during one of the sprint ceremonies, such as a planning meeting). This may help reduce coordination issues and improve shared understanding [4].

Agile is practiced differently across software organizations, and values such as equality among team members, shared ownership, and cross-disciplinary involvement are interpreted and applied in varying ways. We recommend practitioners to assess whether their understanding of these values is reflected in their team's way of working, and decide whether US decomposition should remain a specialist activity or be approached as a collaborative effort.

Our data show that only workflow steps and horizontal splitting are applied by practitioners in our context. Given that many practitioners are not formally trained in US decomposition [3], we suggest that organizations should provide lightweight training to enable more team members to become involved and to improve the consistency of US decomposition without adding overhead. Recent empirical studies [12] show that lightweight interventions may improve US practices. More consistent and granular tasks may also enhance developers' ability to use these tasks effectively in LLM-assisted development.

Implications for researchers. More empirical studies are needed to understand how US decomposition is performed at other software organizations. Future research should examine US decomposition practices in industry by analyzing verb and noun usage in task labels, investigating why they are used, and identifying common verb-noun combinations. This may help determine whether more suitable, lightweight techniques are needed for practitioners in Agile projects.

Although our findings suggest a centralized approach to US decomposition, primarily involving two roles, ITS data may not capture informal participation (e.g., oral alignment). Studies that combine ITS analysis with interviews or observational methods are needed to capture the full process. Currently, little is known about how single-role US decomposition compares with two or more role decomposition, making this an important direction for future research.

We also found that USs often do not follow formats such as the Connextra template [2], which may influence the choice of decomposition techniques. Future research should examine how US quality affects task-level decomposition.

4.4 Threats to validity

We address potential threats to the validity of our study using the categories proposed by Wohlin *et al.* [20]. Although these categories were originally developed for experiments, we apply them to our study context.

Construct validity concerns whether the operational measures used in the study accurately represent the intended concepts and align with the research questions. In practice, USs often deviate from conventional formats, which may affect the applicability of US decomposition techniques. We reduced this risk by analyzing US decomposition at the task level.

Internal validity refers to the risk that uncontrolled factors may influence the relationships observed in the study. To ensure representativeness, project managers and developers identified sprints they considered representative of their typical approach to US decomposition and development activity. However, the sample may reflect their perceptions and not fully represent all sprints. Furthermore, individuals were assigned roles by project managers, developers, and a DevOps engineer. We mitigated potential bias by triangulating with a manager overseeing these teams to confirm role assignments and resolve discrepancies.

Conclusion validity pertains to the reliability and replicability of the data and analysis. The first author performed the coding of the US decompositions, which introduces potential coder bias. Although this threat cannot be fully mitigated, the coding of a subset of US decompositions was reviewed with the other authors to ensure consistency. As the dataset cannot be shared due to confidentiality constraints, we provide the coding scheme and anonymized statistical data in the online appendix [11] to increase transparency and support replication.

External validity assesses the extent to which findings can be generalized beyond the studied context. Our analysis focuses on 10 systems from a single company, each with different team compositions. While this provides some diversity within the organization, we do not claim generalizability to other organizations.

5 Conclusion

There is limited empirical knowledge about how teams perform US decomposition in Agile contexts. In this paper, we performed content analysis, complemented by descriptive statistics, on 373 USs and 1,823 tasks from 10 systems developed with Java, .NET, and Microsoft Power Apps at VX Company to investigate how teams decompose USs into tasks. Our findings show that US decomposition is concentrated in a small set of roles (developer, project manager) and is often performed independently by a single role. Despite the variety of decomposition techniques proposed in the literature, practitioners rely on only two: workflow steps and horizontal splitting. This finding reveals a pronounced gap between techniques prescribed in the literature and those used in practice.

Future empirical studies should investigate whether our findings hold beyond the studied organization, VX Company. Moreover, studies should complement ITS data with interviews or observations to capture informal participation, examine how US quality influences the use of these techniques, and analyze the process of refining high-level requirements (epics) into USs, tasks, and sub-tasks.

Acknowledgements. We thank VX Company for providing the USs and tasks analyzed in this study, and the colleagues who assisted with role assignments.

References

1. Cao, L.: Estimating Efforts for Various Activities in Agile Software Development: An Empirical Study. IEEE Access **10**, 83311–83321 (2022). https://doi.org/10.1109/ACCESS.2022.3196923
2. Cohn, M.: User Stories Applied: For Agile Software Development. Addison-Wesley Professional, Boston (2004)
3. Dellsén, E., Westgårdh, K., Horkoff, J.: Invest in Splitting: User Story Splitting Within the Software Industry. In: Requirements Engineering: Foundation for Software Quality. pp. 115–130. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-98464-9_10
4. Glinz, M., Fricker, S.A.: On shared understanding in software engineering: an essay. Comput. Sci. **30**(3–4), 363–376 (Aug 2015). https://doi.org/10.1007/s00450-014-0256-x
5. Kassab, M.: The changing landscape of requirements engineering practices over the past decade. In: 2015 IEEE Fifth International Workshop on Empirical Requirements Engineering (EmpiRE). pp. 1–8 (2015). https://doi.org/10.1109/EmpiRE.2015.7431299
6. Khanfor, A.: Tasks Decomposition Approaches in Crowdsourcing Software Development. In: International Conference on Human-Computer Interaction. pp. 488–498. Springer (2023). https://doi.org/10.1007/978-3-031-35129-7_35
7. Kumar, B., Tiwari, U., Dobhal, D.C.: User Story Splitting in Agile Software Development using Machine Learning Approach. In: 2022 Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC). pp. 167–171 (2022). https://doi.org/10.1109/PDGC56933.2022.10053226
8. Lawrence, R., Green, P.: The Humanizing Work Guide to Splitting User Stories. https://www.humanizingwork.com/the-humanizing-work-guide-to-splitting-user-stories/ (2025), accessed: 2025-11-19
9. Liskin, O., Pham, R., Kiesling, S., Schneider, K.: Why We Need a Granularity Concept for User Stories. In: Cantone, G., Marchesi, M. (eds.) Agile Processes in Software Engineering and Extreme Programming. pp. 110–125. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-06862-6_8
10. Lucassen, G., Dalpiaz, F., van der Werf, J.M.E., Brinkkemper, S.: Improving agile requirements: the Quality User Story framework and tool. Requirements Engineering **21**(3), 383–403 (2016). https://doi.org/10.1007/s00766-016-0250-x
11. Ly, D., Aydemir, F.B., Dalpiaz, F.: Online Appendix of the Paper "From Concept to Action: How Do Agile Software Teams Decompose User Stories into Tasks?" (2026). https://doi.org/10.5281/zenodo.19403017
12. Molenaar, S., Dalpiaz, F.: Improving the Writing Quality of User Stories: A Canonical Action Research Study. In: Scanniello, G., Lenarduzzi, V., Romano, S., Vegas, S., Francese, R. (eds.) Product-Focused Software Process Improvement. pp. 102–118. Springer Nature Switzerland, Cham (2026). https://doi.org/10.1007/978-3-032-12089-2_7
13. Müter, L., Deoskar, T., Mathijssen, M., Brinkkemper, S., Dalpiaz, F.: Refinement of User Stories into Backlog Items: Linguistic Structure and Action Verbs. In: Knauss, E., Goedicke, M. (eds.) Requirements Engineering: Foundation for Software Quality. pp. 109–116. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-15538-4_7

14. Negri, P.P., Souza, V.E.S., de Castro Leal, A.L., de Almeida Falbo, R., Guizzardi, G.: Towards an Ontology of Goal-Oriented Requirements. In: Workshop on Requirements Engineering (2017)
15. Pavlič, L., Bernsteiner, R., Schlögl, S., Ploder, C.: Splitting User Stories Into Tasks with AI - A Foe or an Ally? In: Uden, L., Ting, I.H. (eds.) Knowledge Management in Organisations. pp. 150–160. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-95901-1_11
16. Ullrich, J., Koch, M., Vogelsang, A.: From Requirements to Code: Understanding Developer Practices in LLM-Assisted Software Engineering. In: 2025 IEEE 33rd International Requirements Engineering Conference (RE). pp. 257–266 (2025). https://doi.org/10.1109/RE63999.2025.00032
17. Varzi, A.: Mereology. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, Spring 2019 edn. (2019)
18. VX Company: VX Company's website. https://www.vxcompany.com/ (2025), accessed: 2025-11-24
19. Wang, X., Zhao, L., Wang, Y., Sun, J.: The Role of Requirements Engineering Practices in Agile Development: An Empirical Study. In: Zowghi, D., Jin, Z. (eds.) Requirements Engineering. pp. 195–209. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43610-3_15
20. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29044-2