Modelling Agile Backlogs as Composable Artifacts to support Developers and Product Owners

Sébastien Mosser[*], **Corinne Pulgar**[†], **and Vladimir Reinharz**[‡]
[*]McMaster University, McSCert, Hamilton, Canada
[†]École de Technologie Supérieure (ETS), Montréal, Canada
[‡]Université du Québec à Montréal (UQAM), Montréal, Canada

ABSTRACT The DevOps paradigm combines (agile) software development and IT operations to deliver high-quality software, thanks to a feedback loop where "ops" feed "devs" and vice versa. In this context, a central challenge is to reduce as much as possible the duration of the feedback loop, allowing stakeholders to reduce their time-to-market and release process duration. This paper describes how to model a product backlog (usually expressed as informal user stories in plain text in an agile context) as a queryable graph-based model. This graph is automatically extracted from existing artifacts thanks to natural language processing techniques. Then, developers and product owners can support their iteration planning process by leveraging the model, enacting a short-range impact analysis feedback loop of their planning decisions. The approach considers the iterative and incremental nature of agile methods through the definition of composition operators to incrementally build the models. We have validated this approach on five industrial scenarios, on top of a reference open-source dataset of 22 product backlogs, representing $1,671$ user stories.

KEYWORDS Agile, DevOps, Backlogs, Graphs

1. Introduction

The agile software development paradigm broke the wall that classically existed between the development team and end-users. Thanks to the involvement of a *Product Owner* (PO) who acts as a proxy to end-users for the team, the *product backlog* (Sedano et al. 2019) became a first-class citizen during the product development. Furthermore, thanks to a set of *user stories* expressing features to be implemented in the product in order to deliver value to end-users, the development teams were empowered to think in terms of added value when planning their subsequent developments. The product is then developed iteration by iteration, incrementally. Each iteration selects a subset of the stories, maintaining a link between the developers and the end-users. The DevOps paradigm (Kim et al. 2016), by considering IT operations (*Ops*) as developers' equals (*Dev*), added another tier to the previous interactions that existed between end-users, POs and development teams. The ops are in charge of the deployment of the product (static code, execution platform) and its monitoring (runtime environment). They are also involved in the product development to join their expertise with the expertise from the others at the product level, instead of thinking in silos inside the company.

Both Agile and DevOps paradigms intrinsically rely on an implicit feedback loop, as depicted in FIG. 1. On the one hand, the PO receives feedback from customers and end-users, translating it into stories related to newly required features that increment the backlog or deprecating stories as the customer's interest in these features are fading. This operation, called grooming, ensures to maintain a good backlog quality. On the other hand, developers receive technical (non-functional) feedback from IT operations related to the performance of their products, or its need for scalability, for example. These feedback loops are essential to support the continuous improvement approach that both Agile and DevOps are advocating (Bordeleau et al. 2020).

The central issue here is the uncomfortable role of the prod-

JOT reference format:
Sébastien Mosser, Corinne Pulgar, and Vladimir Reinharz. *Modelling Agile Backlogs as Composable Artifacts:to support Developers and Product Owners*. Journal of Object Technology. Vol. 21, No. 3, 2022. Licensed under Attribution - No Derivatives 4.0 International (CC BY-ND 4.0)
http://dx.doi.org/10.5381/jot.2022.21.3.a3

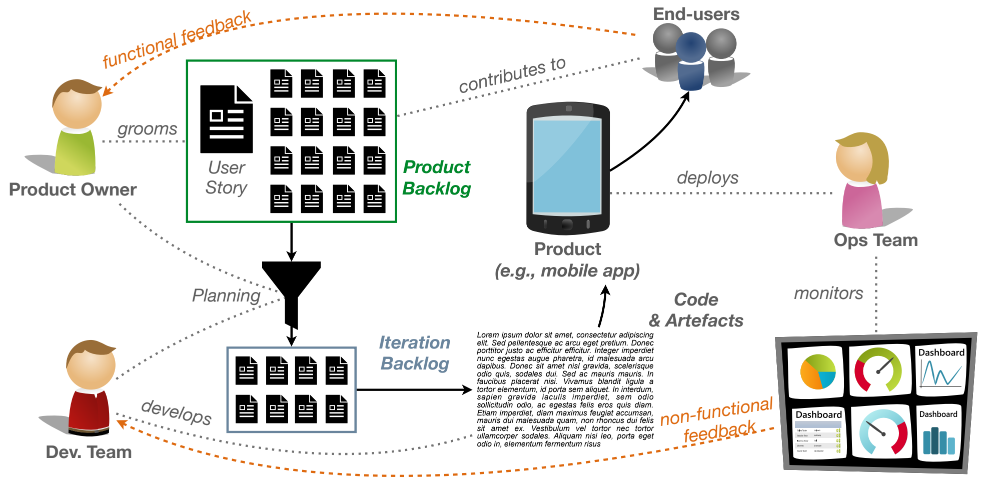


Figure 1 Product and iteration backlogs in the context of a DevOps feedback loop

uct backlog inside this feedback loop. As advocated by the agile approach, it is an essential element. However, according to this paradigm, the product backlog is a collector of feedback from end-users, but it has not been used (yet) to provide immediate feedback to the developers. Thus, the feedback loop used to support its improvement is slowed down, even if the agile and DevOps paradigms aim to reduce it as much as possible. Going back to fundamental DevOps principles, we can link this problem to the "Three Ways of DevOps" (Kim et al. 2016): *(i)* flow thinking, *(ii)* amplifying feedback loop and *(iii)* continual learning. If flow thinking is essential in agile development, the two other ways are not well equipped in terms of automation to work at the backlog level and often focus on more concrete artifacts (*e.g.*, code analysis), slowing down the feedback and continuous amelioration processes.

Taking a step back, it questions the role of modelling in the context of DevOps. The agile community is openly reluctant to models (*e.g.*, the Agile Manifesto[1]). **This paper contributes to the model-driven engineering field by showing how the incremental and iterative nature of backlogs can be taken into account thanks to a lightweight modelling approach that is compatible with agile development**. This modelling approach allows agile teams[2] to automatically extract valuable feedback from backlogs to support their needs. In a nutshell, we propose a model engineering method (and the associated tooling) to exploit a graph-based meta-modelling and compositional approach. The objective is to shorten the feedback loop between developers and POs while supporting agile development's iterative and incremental nature.

After a brief overview of the state-of-the-art concerning backlog modelling (SEC. 2), we define in SEC. 3 a graph-based approach to model product and iteration backlogs in the context of a DevOps team. Our proposition is formalized as a commutative monoid that supports backlogs' incremental and iterative grooming. Then, we explore in SEC. 4 three feedback dimensions that take advantage of this formal model to support teams, applied to 22 backlogs and $1,671$ stories (Dalpiaz 2018). As discussed in SEC. 5, the point here is not to identify the best method to provide feedback to teams but instead to describe a unifying modelling approach for backlogs that supports several alternative definitions of feedback-providing analysis. Finally, SEC. 6 concludes this paper by identifying perspectives to be followed on top of this model.

2. Related Work & State of Practice

2.1. Role of User Stories & Backlogs

From a practical point of view, Sedano *et al.* posited that a *"product backlog is an informal model of the work to be done"* (Sedano et al. 2019), following an empirical study involving 83 practitioners over two years. A backlog implements a shared mental model over practitioners working on a given product, acting as a boundary artifact between stakeholders. This model is voluntarily kept informal to support rapid prototyping and brainstorming sessions. Classically, backlogs are stored in project management systems, such as Jira[3]. These tools stores user stories as *tickets*, where stakeholders write text as natural language. Meta-data (*e.g.*, architecture components, severity, quality attribute) can also be attached to the stories. However, there is no formal language to express stories or model backlogs from a state of practice point of view.

To date, the only reference dataset that is publicly reusable is the one released by Dalpiaz (Dalpiaz 2018). It consists of 22 text files containing stories defined using the Conextra template: "As a <*Persona*>, I want to <*perform an action on entities*> so that <*perceived benefit*>". Unfortunately, this dataset is not qualified or annotated, and no description of the personas or developed products is provided.

2.2. Natural Language Processing (NLP)

Numerous approaches focused on extracting "*models*" out of product backlogs. These approaches classically rely on NLP

[1] https://agilemanifesto.org/

[2] The SAFe framework defines an agile team as "*a cross-functional group of 5-11 individuals who can define, build, test, and deploy an increment of value in a brief timebox*" https://scaledagile.com/.

[3] https://www.atlassian.com/en/software/jira

to extract information out of stories automatically. In a recent systematic literature review released in 2021, Raharjana *et al* have identified 38 NLP primary studies (Raharjana et al. 2021). These studies process user stories to support four main goals: *(i)* detecting defects (6 studies), *(ii)* generating models (14 studies), *(iii)* identifying abstractions (15 studies), and *(iv)* traceability between artifacts (3 studies). Concerning our topic, modelling is covered by the second and third goals. It means that 76% of state of the art focused on using NLP techniques to extract models from stories. This emphasizes the need to provide formal support to the informal nature of backlogs.

Among the NLP approaches available to extract models, 11 approaches (28%) are defined with tool support robust enough to support a quantitative evaluation. Moreover, the kind of considered models is extensive, from tests cases to *Business Process Modelling Notation* (BPMN) ones, including use cases, sequence diagrams and class diagrams. However, only a few of these approaches came with tool support or a quantitative evaluation.

2.3. Role of Modelling

Among the approaches identified by (Raharjana et al. 2021), eight focus on modelling. Three rely on the Visual Narrator tool developed by the Requirements Engineering Lab at Utrecht University (Lucassen, Dalpiaz, van der Werf, & Brinkkemper 2016; Lucassen et al. 2017; Dalpiaz et al. 2019). The tool can extract what is called a *conceptual model* of a backlog in an ontology-like way. The conceptual models are then used to measure user stories quality by detecting ambiguities or defects in a given story. Three other approaches focused on the generation of UML artifacts: class diagrams (Nasiri et al. 2020), sequence diagrams (Elallaoui et al. 2015) and use cases diagrams (Elallaoui et al. 2018). It is interesting to note that, contrarily to the previous ones, these approaches consider a backlog as a means to an end, the end being the availability of UML models to support the development team. However, they do not value the backlog as a first-class citizen in their methodology, breaking its role concerning the Agile/DevOps paradigms. Two approaches proposed to follow an ontology-based approach (Landhäußer & Genaid 2012; Athiththan et al. 2018). The first one creates an ontology from the source code and uses NLP to bind user stories to code locations, ensuring traceability automatically. The second one models the backlog as an ontology and uses it to generate boilerplate code to speed up the development time. None of these approaches leverage the graph structure of the ontologies, and, again, they consider the backlog model as a means to an end.

In addition to the studies based on Visual Narrator that analyze the backlog conceptual model (*e.g.*, ambiguity detection), other approaches focus on stories management. For example, Galster *et al.* uses NLP to extract architectural properties from the stories (Galster et al. 2019). The backlog model is also used to identify duplicated stories (Barbosa et al. 2016), stressing the difficulty for a team to groom a large and complex backlog properly. Finally, stories can be used to generate *executive summaries* for customers and high-level executives (Rodeghero et al. 2017).

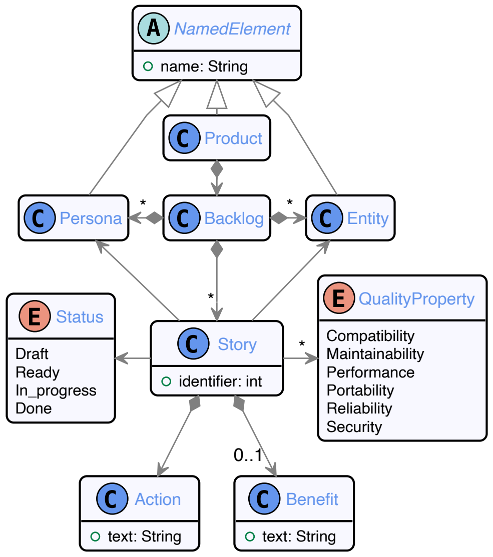


Figure 2 Backlog conceptual metamodel

Interestingly, none of these approaches considered the backlog as a first-class citizen in their tooling support. Excepting Visual Narrator-based approaches, backlogs are transformed into other models (*e.g.*, UML) and never used again. Visual Narrator keeps its conceptual model internal and exposes black-box analysis to users, dedicated to stories' quality in terms of requirements engineering, for example. Thus, it does not support the DevOps team in the development, as the provided feedback focuses on the requirements' expression instead of their role in the software development.

3. Modelling Backlogs as Composable Graphs

This section describes our approach used to create backlog models automatically. To guarantee the soundness of the constructed backlogs, we start by defining the different elements involved in backlog modelling and operators manipulating such elements. Thus, such soundness is ensured by design, thanks to the properties identified over the formal definition of each operator. As the approach aims to be used as a toolchain to support developers and POs, we expose it as an internal DSL.

From a modelling point of view, we can represent the concepts involved in the definition of a backlog in a metamodel, as depicted in Fig. 2. Without surprise, the key concept is the notion of `Story`, which brings a `Benefit` to a `Persona` thanks to an `Action` performed on an `Entity`. A `Story` is associated to a readiness `Status`, and might optionally contribute to one or more `QualityProperty` (*e.g.*, security, performance).

Consider, for example, the following story, extracted from the reference dataset (Dalpiaz 2018):

> *"As a user, I want to click on the address so that it takes me to a new tab with Google Maps."*

This story brings to the user (`Persona`) the benefit of reaching a new Google Maps tab (`Benefit`) by clicking (`Action`) on the displayed address (`Entity`).

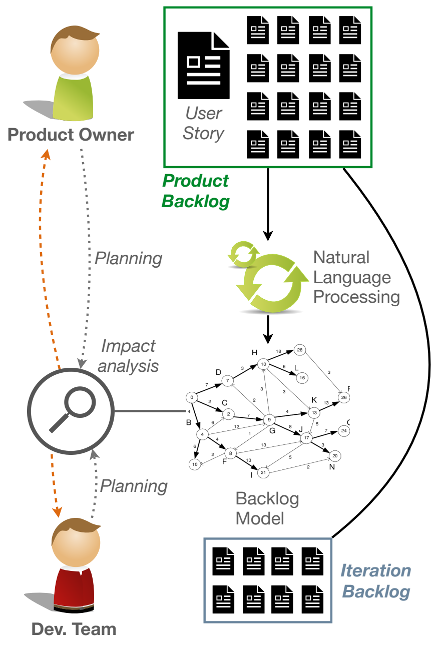


Figure 3 Providing early feedback at the backlog level

As `Entities` and `Personas` implement the *jargon* to be used while specifying features in the backlog, they are defined at the `Backlog` level. On the contrary, `Actions` belong to the associated stories and are not shared with other stories. Finally, a `Product` is defined as the `Backlog` used to specify its features.

The definition of this metamodel supports the characterization of the domain we are working on. It is then possible to address the initial challenge, *i.e.*, leveraging this characterization to provide early feedback to POs and developers while planning a development iteration.

Thanks to a collaboration with architects (IBM France Labs) and DevOps Lead engineers (Instant System), confronted to state-of-the-art and state-of-practice in this context, we identified the following properties a system designed to provide such feedback must hold:

- P_1: rely on existing artifacts (*i.e.*, existing stories);
- P_2: integrate with existing tooling;
- P_3: be as automated as possible;
- P_4: be customizable by the team (no silver-bullet);
- P_5: support an incremental and iterative approach.

To implement such a vision in an actionable way, we propose the system depicted in FIG. 3. As discussed in SEC. 2, NLP approaches are efficient in the context of backlog. Consequently, we propose using an NLP-based extractor to instantiate a backlog model. Then, this model will be used during the planning phase to support teams while selecting the stories to be implemented during the next iteration. Finally, we provide in SEC. 4 five examples of impact analysis that can be performed on top of such a model.

3.1. Composable Backlogs (P_4, P_5)

In order to support team customization (*e.g.*, a given team might want to enrich the backlog metamodel with additional information existing in their product management system), a metamodelling approach based on a closed-world (*e.g.*, EMF, UML) is not appropriate. It also triggers issues concerning the incremental and iterative approach of building backlog, as composing such models together in an incremental way is still an intense research domain (Kienzle et al. 2019). Consequently, we chose an open-world (ontological) representation by modelling backlogs as graphs. The graphs are equipped with constraints (*e.g.*, a story always refers to a persona and an entity) to ensure that the minimal structure captured in the previously defined metamodel is guaranteed.

In this section, we describe how backlogs are modelled in our approach. The approach is supported by a reference implementation, using PYTHON 3.9 and NETWORKX, a reference library for graph modelling in this ecosystem (Hagberg et al. 2008). We provide formal definitions allowing us to construct backlog out of independent stories in a way that is sound, supporting the definition of the analysis described in SEC. 4. As the formal model is provided as an internal DSL, we illustrate how the elements are mapped to language constructions at the implementation level (when relevant).

The choice of relying on PYTHON instead of more classical EMF-based approach[4] is motivated by the integration with the state-of-practice regarding NLP tools and graph algorithms. Therefore, there is no strong dependency for the PYTHON language, and other implementation platforms can be considered (*e.g.*, JAVA for the API and NEO4J for the graph representation).

Definition 1 Story. *A story $s \in \mathcal{S}$ is defined as a tuple (P, A, E, K), where $P = \{p_1, \dots, p_i\}$ is the set of involved personas, $A = \{a_1, \dots, a_j\}$ the set of performed actions, and $E = \{e_1, \dots, e_k\}$ the set of targeted entities. Additional knowledge (*e.g.*, benefit, architectural properties, status) can be declared as key-value pairs in $K = \{(k_1, v_1), \dots, (k_l, v_l)\}$. The associated semantics is that the declared actions bind personas to entities. Considering that story independence is a pillar of agile methods (as, by definition, stories are independent inside a backlog), there is no equivalence class defined over $\mathcal{S}$: $\forall (s, s') \in \mathcal{S}^2, s \neq s' \Rightarrow s \not\equiv s'$.*

We represent in FIG. 4 an example of three independent stories $\{s_1, s_2, s_3\}$ modelled using this definition, as well as the associated code.

Definition 2 Backlog. *A backlog $b \in \mathcal{B}$ is represented as an attributed typed graph $b = (V, E, A)$, with V a set of typed vertices, E a set of undirected edges linking existing vertices, and A a set of key-value attributes. Vertices are typed according to the model element they represent ($v \in V, type(v) \in \{Persona, Entity, Story\}$). Edges are typed according to the kind of model elements they are binding. Like backlogs, vertices and edges can contain attributes, represented as $(\text{key}, \text{value})$ pairs. The empty backlog is denoted as $\emptyset = (\emptyset, \emptyset, \emptyset)$.*

[4] An ECORE binding for PYTHON exists (PYECORE), but would have made the implementation of the formal model less straightforward without adding any value to the contribution of the article, which is based on graphs.

– Backlog excerpt: Content Management System for Cornell University — CulRepo (Dalpiaz 2018).

  - s_1: As a faculty member, I want to access a collection within the repository.
  - s_2: As a library staff member, I want to upload material to the repository.
  - s_3: As a library staff member, I want to create metadata for items.

– Associated model:

  - $s_1 = (\{faculty\ member\}, \{access\}, \{repository, collection\}, \emptyset) \in \mathcal{S}$
  - $s_2 = (\{library\ staff\}, \{upload\}, \{repository, material\}, \emptyset) \in \mathcal{S}$
  - $s_3 = (\{library\ staff\}, \{create\}, \{metadata\}, \emptyset) \in \mathcal{S}$

– Python implementation:

```python
from backlog.model import Story
s1 = Story("s1", {"Faculty Member"}, {"Repository", "Collection"}, {"access"},
           "As a faculty member, I want to access a collection within the repository")
s2 = Story("s2", {"Library staff"}, {"Repository", "Material"}, {"upload"},
           "As a library staff member, I want to upload material to the repository")
s3 = Story("s3", {"Library staff"}, {"Metadata"}, {"create"},
           "As a library staff member, I want to create metadata for items")
```

Figure 4 A set of three independent stories $\{s_1, s_2, s_3\}$

```python
from backlog.model import Backlog
b = Backlog.empty().named_as("b")
```

Listing 1 Empty backlog: $b = (\emptyset, \emptyset, \{(name, b)\}) \in \mathcal{B}$

Here is an example of a backlog containing a single story s_1("*As a faculty member, I want to access a collection within the repository*").

$$\begin{aligned} b_1 &= (V_1, E_1, \emptyset) \in \mathcal{B} \\ V_1 &= \{Persona(faculty\ member, \emptyset), \\ &\quad Story(s_1, \{(action, access)\}) \\ &\quad Entity(repository, \emptyset), \\ &\quad Entity(collection, \emptyset)\} \\ E_1 &= \{has\_for\_persona(s_1, faculty\ member), \\ &\quad has\_for\_entity(s_1, repository) \\ &\quad has\_for\_entity(s_1, collection)\} \end{aligned}$$

At the implementation level, we expose a factory method named `empty()` to build the empty backlog that is used as the entry point for backlog grooming. We represent in LST. 1 the code used to create an empty backlog named b.

Definition 3 Story Promotion ($\sim$). *For type-compliance, we define an operator $\sim$: $\mathcal{S} \to \mathcal{B}$ used to promote an independent story into a backlog that only contains this same story. Considering a story $s \in \mathcal{S}$, we denote as $\tilde{s} \in \mathcal{B}$ the promoted story.*

Calling this operator on a story transforms each element into nodes and links all the nodes together. At the implementation level, we override the $\sim$ operator to allow a developer to promote a story s_1 into a backlog $b_1 = \tilde{s}_1$.

With an automated way to project a story in the backlog space, we can now define an incremental and iterative way of creating backlogs (P_5). The objective here is to support the very nature of Agile/DevOps software development, relying on backlogs built incrementally. Considering that backlogs are built in arbitrary ways by teams, it is essential that a team can work in any order during the backlog construction phase.

```python
from networkx.algorithms.operators.binary \
    import compose

class Backlog:
    # ...
    def __add__(self, other):  # self + other
        result = Backlog.empty()
        result.__graph = compose(self.__graph,
                                 other.__graph)
        return result
```

Listing 2 Implementing backlog merge as graph union

Definition 4 Backlog merge ($\oplus$). *We define the merge of two backlogs $b_1, b_2 \in \mathcal{B}^2$ as the operator $\oplus : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$. Let P_1 the vertices typed as personas in b_1 (resp. P_2 in b_2), E_1 the vertices typed as entities in b_1 (resp. E_2 in b_2), and A_1 b_1's attributes (resp. A_2 for b_2). Considering $b = b_1 \oplus b_2$ the merged backlog, b contains as personas' vertices $P_1 \cup P_2$, as entities' vertices $E_1 \cup E_2$ (equivalence classes among personas and entities are based on names), and as attributes $A_1 \cup A_2$.*

As the previous definition relies on graph unions based on equivalence classes, $\oplus$ is by design idempotent, commutative and associative. It means that backlogs $\mathcal{B}$ equipped with $\oplus$ and $\emptyset$ form a commutative monoid $(\mathcal{B}, \oplus, \emptyset)$. Consequently, by reducing the backlog construction problem to a graph union one, the associativity and commutativity properties of this monoid support the development team by design. Equivalent elements (*i.e.*, entities, personas) are automatically unified in the composed backlog. At the implementation level, we override the $+$ operator (method `__add__`) to support the following syntax: `b2 = b0 + b1` (LST. 2).

Definition 5 Backlog increment ($\leftarrow$). *To increment a backlog b with a story s, we define an asymmetric operator to support*

```python
class Backlog:
    # ...
    def __iadd__(self, story): # self += story
        return self + ~story
```

Listing 3 Implementing backlog incrementation

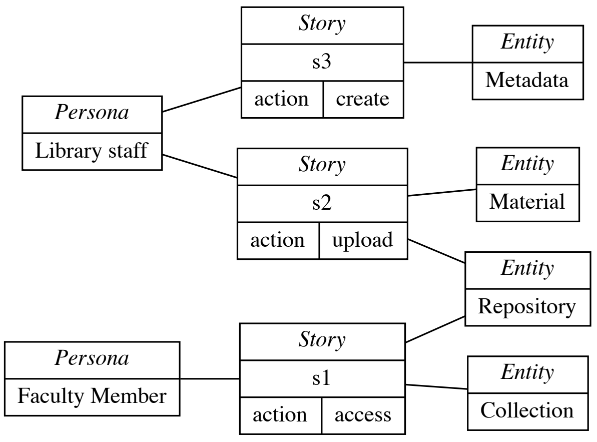


Figure 5 Building $b = \emptyset \leftarrow s_1 \leftarrow s_2 \leftarrow s_3 \in \mathcal{B}$

the addition of a story into a given backlog ($\leftarrow$: $\mathcal{B} \times \mathcal{S} \rightarrow \mathcal{B}$) as syntactic sugar over story promotion and backlog merge. Incrementing b with s (i.e., $b \leftarrow s$) *is equivalent to merging b and* $\tilde{s}$ (i.e., $b \oplus \tilde{s}$)*. The operator is left-associative by design.*

We depict in FIG. 5 how a backlog can be constructed by incrementing the empty backlog with the three stories defined in FIG. 4. The constructed backlog (named b) contains three entities, two personas (merged), and three independent stories. Each story defines its own action, but shares the merged elements with others (*e.g.*, the entity named *repository* is shared by s_1 and s_2). At the implementation level, we override the `+=` operator (LST. 3, method `__iadd__`) to reflect the asymetric and incremental dimensions of this operator.

Definition 6 Backlog Enrichment. *According to the open-world philosophy, backlog elements can be enriched with key-value properties. The enrichment is defined as an operation* $enrich : \mathcal{B} \times \mathcal{K}^n \rightarrow \mathcal{B}$*, where* $(id, \{(k_1, v_1), \ldots\}) \in \mathcal{K}$ *represent the set of key-value pairs to be added to the node identified as id.*

Relying on a graph as the underlying modelling foundation allows teams to customize their models by enriching their proposition with knowledge from their own practice (P_4). For example, one can easily integrate the status of each story in the backlog if needed by enriching it:

$$K = \{(s_1, \{(status, done)\}), (s_2, \{(status, ongoing)\}),$$
$$(s_3, \{(status, ongoing)\})\}$$
$$b = enrich(\emptyset \leftarrow s_1 \leftarrow s_2 \leftarrow s_3, K)$$

At the implementation level, this operation is supported by the injection of a data dictionary into the graph (LST. 4)

```python
class Backlog:
    # ...
    def enrich(self, extra):
        result = Backlog.empty()
        g = self.__graph.copy()
        networkx.set_node_attributes(g, extra)
        result.__graph = g
        return result
```

Listing 4 Enriching backlog with unforeseen knowledge

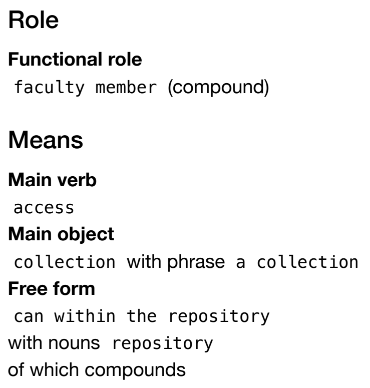
Summary. In this section, we presented a formalization of backlogs as graphs. The objective of this formal model is to demonstrate how our proposition ensures properties P_4 and P_5. Furthermore, we propose an elegant way of modelling such artifacts by reducing backlog construction to graph composition while ensuring their extensibility. Consequently, this formal model is essential to ensure that the constructed graphs are sound and support the development team adequately. Finally, as the model is intended to be used programmatically to define impact analysis tools to shorten the feedback loop between POs and developers, we also illustrated how the model is implemented as an internal DSL using the PYTHON language.

3.2. Automated Instantiation (P_1, P_2, P_3)

To respect P_1 and P_2, the system assumes access to the project management system (*e.g.*, Jira ticketing). This access can be obtained thanks to an API (*e.g.*, Jira exposes a REST API to interact with tickets) or through static exportation (*e.g.*, as XML, CSV, TXT, or JSON files). As a result, we assume that it is possible to have automated access to the content of each story defined in the backlog and their associated metadata (*e.g.*, labels added to the tickets in Jira).

In order to support P_2 and P_3, an automated NLP step is required to automatically extract information from the textual backlog. Considering that many approaches are available in this context, our system is not tied to particular ones. Instead, it only assumes that the NLP-based extraction produces a conceptual model identifying at least personas, actions and entities involved in each story. To demonstrate the versatility of the approach at the implementation level, we have experimented with several tools. First, we considered two NLP tools for story extraction: Visual Narrator (Robeer et al. 2016) and the approach of Gilson *et al* (Gilson et al. 2020). They both produced comparable results in terms of performance and accuracy. Both approaches compute conceptual models (*i.e.*, ontologies, and robustness models as UML artifacts) of the system modelled by the backlog, stored into files that can be automatically processed to produce a backlog compatible with our proposition. The only assumption made by the approaches is that the stories are expressed using the classical *"As a …, I want to …"* template (Lucassen, Dalpiaz, Werf, & Brinkkemper 2016).

We consider here Visual Narrator, applied to the Content Management System for Cornell University (CulRepo) used in the previous section. Taking as input the backlog as a text file (*e.g.*, `dataset/raw/g27-culrepo.txt`), it produces an analysis of each story in various formats (*e.g.*, HTML, JSON, Prolog, Ontology). We represent in FIG. 6 the human-readable result


Role

Functional role

`faculty member` (compound)

Means

Main verb

`access`

Main object

`collection` with phrase `a collection`

Free form

`can within the repository`

with nouns `repository`

of which compounds

Figure 6 Output of VisualNarrator for s_1

obtained when processing s_1. The NLP extraction identifies the persona (*faculty member*), the action (*access*), and two entities (*collection* and *repository*). Based on this extraction process, it is possible to automatically instantiate stories according to the approach described in the previous section.

Each NLP extraction tool will rely on its data format, as there are no standards whatsoever in this field. Therefore, integrating a new extraction tool requires writing glue code that will transform the tool's output into stories that are modelled according to our proposition. Nevertheless, from an abstract point of view, we assume this technical challenge is being tackled by a function named *nlp-extract*, taking as input a textual backlog (*e.g.*, a file) and providing as output stories (as instances of the `Story` class). Then, building the product backlog means using the increment operator on the empty backlog.

$$\textit{nlp-extract}(\textit{file}) = \{s_1, \ldots, s_n\} \in \mathcal{S}^n$$
$$b = \emptyset \overset{n}{\underset{i=1}{\leftarrow}} s_i \in \mathcal{B}$$

Summary. In this section, we demonstrated how the previously defined formal elements and operators could be combined into a toolchain that interacts with a classical DevOps environment to create backlog models automatically.

4. Validation: Models as Feedback Providers

Researching user stories and backlogs becomes complicated when reaching the validation stage. As stories capture the business values, they are considered trade secrets by companies, making the definition of case-study (qualitative) experiments very difficult. Furthermore, as an immediate consequence, the lack of available data prevents performing large-scale quantitative analysis.

Instead, we consider here five validation scenarios identified while working closely with industrial partners during the past five years. We denote scenarios as Σ_i to avoid confusion with stories (identified as s_i). We classified the feedback one can compute over the graph according to three categories: *(i)* Product analysis (Σ_1, Σ_2), *(ii)* Iteration planning (Σ_3, Σ_4), and *(iii)* Portfolio management (Σ_5). Our industrial partners (IBM, Instant Systems) coined these scenarios as essential to support their software development feedback loop in a DevOps

Obstacle (Sedano et al. 2019)	**Validation Scenario**
O_1: Preconceiving Problems	Σ_4, Σ_5
O_2: Preconceiving Solutions	Σ_3
O_3: Pressure to Converge	Σ_1
O_4: Ambiguity	Σ_4, Σ_5
O_5: Time Pressure	Σ_2, Σ_3
O_6: Blocking Access to Users	*N/A*

Table 1 Linking Obstacles (O_i) and Scenarios (Σ_i)

context. These scenarios cover five out of the six obstacles (O_i) to product backlog management identified by Sedano *et al.* in their reference study investigating product backlog management (Sedano et al. 2019). We describe in TAB. 1 how the five validation scenarios cover the obstacles. As our contribution does not consider end-users as part of its scope, it is not possible by design to cover the last obstacle identified.

Our objective in this section is to validate that the backlog models defined in SEC. 3 can be used to answer value-added questions from POs and developers. The point is not to propose a *silver bullet* solution, but instead to provide a tooled model that can be customized to fit various scenarios. We applied these scenarios to the only publicly available dataset of backlogs (Dalpiaz 2018). It is necessary to note that this reference dataset does not *qualify* the stories and only present them grouped by product in a flat representation. Typically, there is no information about the different iterations used to develop the product, and there is no additional description of the artifacts involved. In the remainder of this paper, we denote as g_i a backlog identified under this name in the reference dataset. After being processed by the approach from Gilson *et al* (Gilson et al. 2020; Galster et al. 2019), it results in a set of $1,671$ stories.

Software Artifact. To support this validation section, we provide a companion software artifact, available as open-source software on GitHub at the following address: https://github.com/ace-design/backlog-modelling. The artifact is implemented in PYTHON 3.9 and contains the metamodel implementation, the initial validation dataset (Dalpiaz 2018), and the code of each scenario described in this paper.

4.1. Σ_1: Structural analysis (Product)

Context & Pain point. We consider here an agile team working on a specific product. The team is *pressured to converge* (O_3) and needs to prepare arguments to defend how its backlog compares to the global portfolio in the company.

Scenario implementation. Leveraging the underlying graph structure, it is possible to compute structural metrics (*e.g.*, number of vertices, number of edges, average connectivity) on a backlog to characterize it. Computing a footprint of a backlog at a given point in time is implemented as the construction of a vector containing structural dimensions of the graph (*e.g.*, number of personas, number of actions)

Validation experiment. To validate this scenario, we designed an experiment that analyses the 22 available backlogs according to standard graph metrics. The point is to investigate the

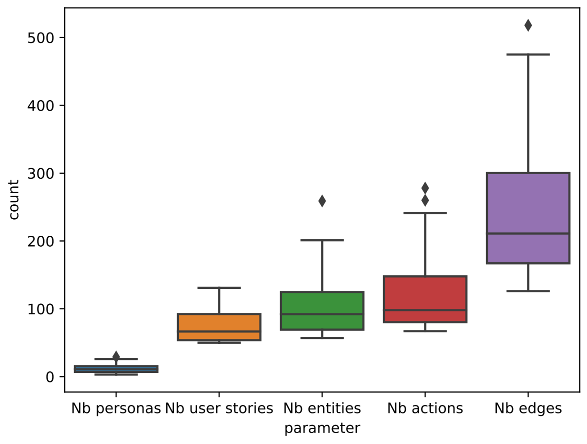


Figure 7 Structural metrics distribution for the 22 considered backlogs

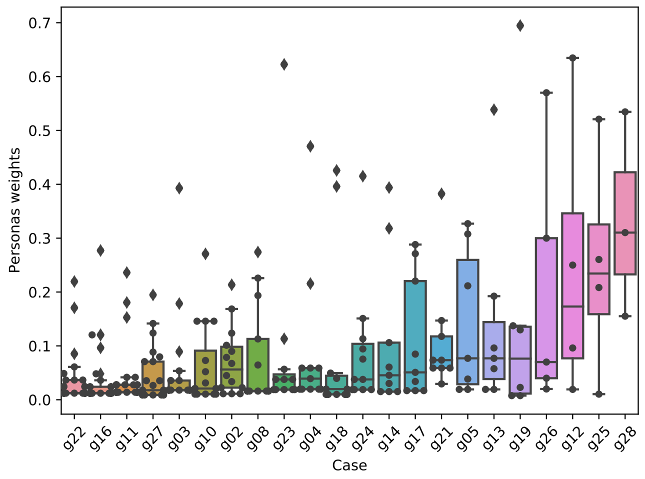


Figure 8 Distribution of normalized weights inside the 22 backlogs (personas)

versatility of backlog structure, emphasizing the need for such analysis. We represent in FIG. 7 the distribution of these metrics for the 22 backlogs.

Conclusions. Without surprises, the experiment exhibits the intrinsical versatility of backlogs. It emphasizes the need for customization support by the team for any automated approach that works on backlogs (P_4).

4.2. Σ_2: Elements' Weights (Product)

Context & Pain point. In this scenario, we still consider a team working on a given product. The team is feeling *time pressure* (O_5) and needs to take a step back on the state of the backlog to properly identify how each element compares to each other in terms of design, facilitating the planning of the upcoming iterations. Selecting a story that involves an entity heavily involved in the backlog might substantially impact the product development. When the team is pressured, it can also face selection bias. For example, the PO might focus involuntarily on a given persona while organizing focus groups with end-users, leading to an imbalanced specification.

Scenario implementation. Such a measurement provides immediate feedback to the team at two different levels: *(i)* by explaining how central an element is in the specification, and *(ii)* by allowing the team to track the evolution of the backlog during grooming and planning phases. To investigate this kind of analysis, we propose to leverage the graph by weighting personas and entities based on their involvement in the graph. For example, considering a given persona p, its weight $w_p \in \mathbb{R}$ can be defined as the ratio of stories involving p over the total number of stories defined in the backlog (a similar ratio is used for entities). Implementing such computation is straightforward when leveraging the underlying typed graph model.

Validation experiment. To emphasize the domain versatility, we computed for each backlog in the dataset the statistical distribution of its (normalized) persona's weights. We depict in FIG. 8 the results.

Conclusions. This experiment leads to two conclusions. First, in addition to the properties associated with the formal model, direct access to the graph structure makes it easy to build such knowledge. Secondly, the centrality of personas inside each backlog does not follow any clear pattern, emphasizing the P_4 property again. It is mandatory for the (meta-)modelling approach chosen in an agile context to adapt to team usage and practices.

4.3. Σ_3: Coverage metrics (Iteration)

Context & Pain points. Planning an iteration is a critical phase in agile development: the team commits to deliver a given amount of value to end-users. Teams rely on informal measurements such as story points or business value and use *yesterday's weather* techniques to prepare their next iteration. If the team has delivered x story points on average during the last iterations, they will commit several stories where story points sum to a value close to x. This approach suffers from issues, such as the *no-estimate* movement recently identified (Sandeep RC 2020; Duarte 2016). In this context, teams has a tendency to face *preconceived solutions* issues (O_2), especially when feeling *time pressured* (O_5). There is a lack of quality-driven appreciation of such estimations. There is no way for the team to know how this value is spread over the different elements constituting the backlog.

Scenario implementation. The team needs guidance in terms of story selection, as the choices made at this stage are technical and political. For example, should the team focus on one given persona (*e.g.*, because of a focus group involving subjects impersonating this persona scheduled for the end of the next iteration), focus on a given entity (*e.g.*, for technical operations reasons), or increment the product in a more balanced way? We leverage here the incremental construction of our backlog model to load the stories into the backlog and call the `enrich` operator to load stories' status into the backlog. Based on this

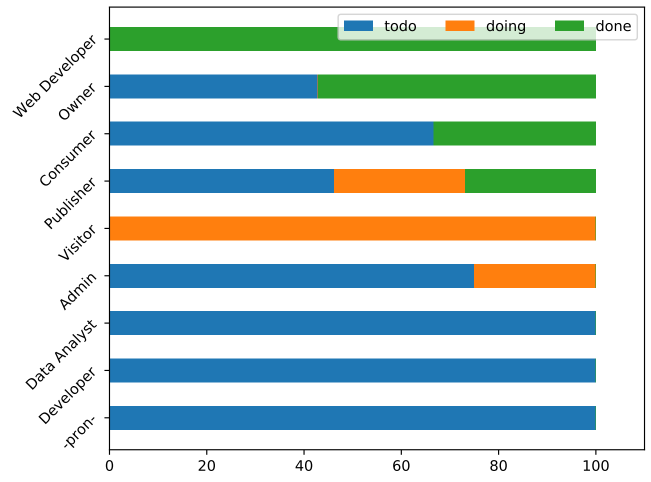


Figure 9 Evaluating personas' coverage for the next iteration

enriched backlog, it is possible to compute a snapshot of the coverage associated with each persona.

Validation experiment. We considered the product g_{14} specified by 67 stories and involving 9 actors. According to our previous definition, $g_{14} = \emptyset \leftarrow s_1 \leftarrow \ldots \leftarrow s_{67}$. We consider a situation where close to 30% of the existing specification has already been implemented in the previous iteration. The status of each story is stored in a CSV file that is used to *enrich* the previously built backlog. For the sake of this experiment (as the reference dataset is not qualified), we asked one of our industrial partners to create this CSV file based on his experience in agile development. Then, we asked a product owner to look at the backlog and computed coverage. Finally, we collected the PO's feedback on the computed pieces of information.

Conclusions. The PO emphasized that, according to the agile teams he would work with, different kinds of coverage would be necessary. It, again, emphasizes P_4 and reinforces the need for a lightweight modelling approach that one can tailor to its needs. The PO also says that coverage could be helpful in terms of personas or entities, but also in terms of non-functional dimensions, *e.g.*, architectural properties (Galster et al. 2019). It emphasizes the need for a model that relies on an open-world assumption to allow teams to feed the model with unforeseen information. Finally, the PO made it clear that some coverage maps such as the one depicted in FIG. 9 representing the coverage of personas at a given moment would be helpful for any team, as the key point of agile development is to deliver value to personas. It comforts the fact that a lightweight modelling approach is the right approach to support the team. First, the scriptable dimension can be leveraged to allow customization. Furthermore, it is still possible to release off-the-shelf model analysis based on commonalities identified during domain analysis.

4.4. Σ_4: Story Recommandation (Iteration)

Context & Pain points. Another difficulty encountered by teams is related to stories selection when working on a large product backlog. It is easy for a team to wear blinders after some time working on a given product, leading to *preconceiving problems* situations (O_4). Another issue is related to identifying *ambiguities* (O_5), at the backlog level, or more globally at the portfolio one. It is challenging to know which stories are similar to a given story s and which ones are outsiders.

Scenario implementation. In a nutshell, the key idea here is to automatically build clusters of stories that the algorithm finds "similar". Then when the team selects a story from a cluster, it is possible to show the ones remaining in the cluster, to guide the selection and provide simultaneous feedback. Modelling the stories' backlog as a graph allows us to deploy similarity algorithms that can work at different levels, from purely structural to more semantic-driven approaches. However, it is worth noting that, in terms of graphs, "similarity" cannot be defined in the general case and has to be tailored to a specific problem (Zager & Verghese 2008). This is compatible with our customization property, allowing a team to try several recommendation algorithms and/or configurations to find the one that suits its development.

Validation experiment. To validate this idea, we have arbitrarily selected two backlogs in the dataset: g_{13} and g_{24} (representing 104 stories, 52 each). For these two backlogs, we created a *Ground Truth* (GT) by manually clustering the backlog into partitions of similar stories. To create such a GT, one author worked in isolation to produce the clustering, and another one reviewed the provided clustering. In case of disagreements, the two authors discussed together to find a way to solve the conflict. If the conflict was unsolvable, the third author was supposed to act as a referee (but this situation had never happened during the experiment). We have then used four different algorithms to create clusters automatically and demonstrate the versatility supported by the formal model:

1. *Fluid:* an off-the-shelf stochastic algorithm for community detection in graphs, which relies on the metaphor of fluid propagation inside the graph to automatically identify neighbourhoods as clusters (Parés et al. 2017).

2. *K-means 3D:* associate to each story a vector of three dimensions: the number of personas (usually one), the number of actions, and the number of entities. Then use a k-means approach to create clusters.

3. *K-means* $(P+E)D$*:* considering $s = (P, A, E, K)$ a story, we associate to s a vector of size $|P| + |E| + 1$, binding each story to a vectorial representation of the actors and entities it contains. The first dimensions represent the involvement of a given persona in the story, followed by the involvement of a given entity. Finally, the number of actions involved in the story is stored in the last dimension. For example, considering a backlog contains two actors and three entities, a story defining four actions involving the first actor and the two last entities is bound to a vector of six dimensions: $[1, 0, 0, 1, 1, 4]$. It results in a vectorial space of 78 dimensions for g_{13}, and a vectorial space of 83 dimensions for g_{24}. Then k-mean is used to identify clusters among the vectors.

4. *Dendrograms:* a tailored implementation of business rules that implements the rationale of our manual similarity selection. A rooted tree is built, clustering step by step stories together based on a dissimilarity metric. A bonus of -1 is given for matching nodes of the same type (persona, action, entity) and the same name. Then maluses are in three categories. $+1$ if nodes of different types are matched, or a node or edge must be created. $+0.5$ if a persona or entity match but with a different name. Finally, most user stories contained a list of actions let say a_1 and a_2, and we penalized their differences by $0.25 \times \frac{|a_1 \Delta a_2|}{|a_1 \bigcup a_2|}$. The tree is then built using the unweighted pair group method with arithmetic mean (UPGMA) method (Sokal 1958). A core assumption is that the distance from the root to each leaf is the same, applicable here because there is no notion of temporality in the backlogs.

We depict in FIG. 10 the results obtained with these four methods. The x-axis represents the granularity of the clustering, *i.e.*, the number of clusters to be computed or the threshold to be applied for grouping the sub-trees in the dendrogram approach. The y-axis represents the "*success*" of the automated clustering compared to our manual clustering, the higher, the better. We consider an automated cluster to be successful if stories that have been manually grouped are found together in the automated one (we will discuss the limitations of this definition in SEC. 5).

Conclusions. It is interesting to notice that off-the-shelf approaches that only rely on the graph structure (*fluid* and *k-means 3D*) do not perform well in terms of clustering to identify similar stories. This can be explained by the constrained nature of the graph, which implies a pre-defined structure for backlogs, jeopardizing the outcome of such approaches. However, algorithms that leverage such a structure while using semantic information (elements involved in the story with *k-means (P+E)D*, and business rules with *dendrograms*) seem to provide a better recommendation to the team. Here, the idea is not to defend one algorithm but to emphasize the versatility supported by the graph structure used to model the backlogs. It is then possible for a team to experiment and use the most suitable recommendation that supports their work. For example, the team working on g_{24} might decide to go for the *k-mean 83D* approach, as it produces correct results concerning the team practices without having to implement specialized business rules like for g_{13}.

4.5. Σ_5: Inter-Product recommandations (Portfolio)

Context & Pain Points. The previous section explored how to support a team to work on their exact product by automating stories recommendation while preparing the next iteration. Here, still related to *ambiguities* (O_4) and *preconceived problems* (O_1), we are interested in the human factor of DevOps to facilitate communication inside a given value stream (delivering a product) and between value streams (delivering a portfolio of products). The idea of transferring knowledge from one product to another one thanks to an analysis of the domain models can also be found in approaches using artificial intelligence for assisting software designers (Combemale et al. 2021)

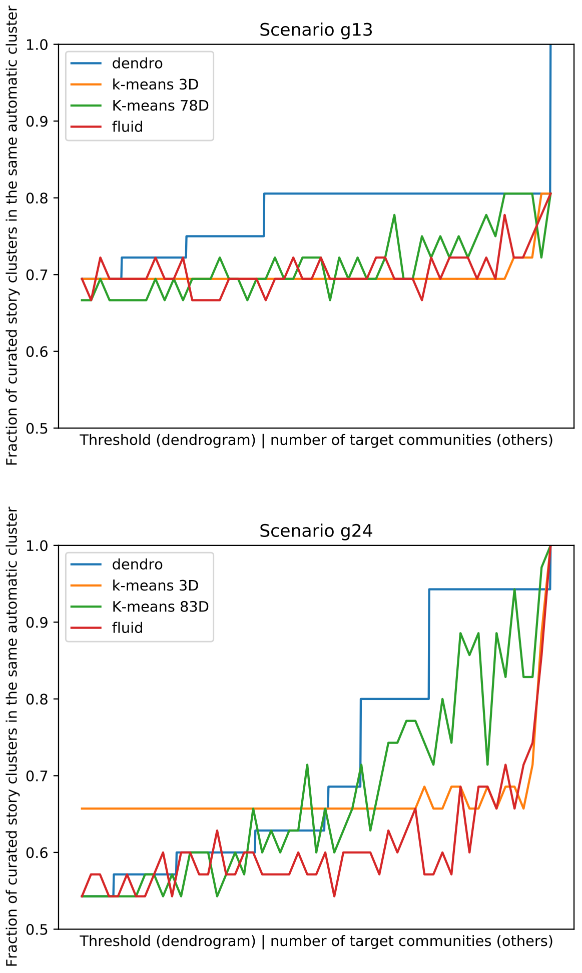


Figure 10 Comparing different recommandation clustering algorithms

Scenario implementation. In this context, the idea is to take advantage of the graph structure to extract relevant information for the teams at the portfolio level, *e.g.*, the team interacting with backlog b is working on specifications similar to the team working on backlog b'. The objective is to exhibit a similarity matrix, which, considering two stories, provides a similarity score between them. A challenge here is to take into account *vocabulary gaps* and *jargon* among teams. By design, persons and entities can be considered fixed points in backlogs, *i.e.*, they model a specific vocabulary used inside the team, with its specificities. For example, what is called a *Repository* by a team might be similar to what is called a *Data Archive* by another one. The idea is to use NLP techniques to identify similarities between concepts across backlogs. As for the previous section, it is worth noting that there are numerous alternatives to compute such similarities. Therefore, this section demonstrates that the model supports such an approach, allowing a company to choose the best fit for its global practices. To exemplify this point, we defined a simple way of computing similarity between stories: we rely on the text-similarity implementation of spaCy (Honnibal et al. 2020), a reference NLP library in Python.

Considering two strings a and b, calling `a.similarity(b)` returns a real value ($\in [0,1]$) to measure the semantic similarity between the two strings, the higher the better. For each backlog, we apply a graph transformation that flatten the graph into triples $(p, A, e) \in \mathcal{T}$, where p is a persona, e an entity, and A the set of actions that p perform on e, whatever the story. Then, we define a similarity operator $\sigma : \mathcal{T} \times \mathcal{T} \rightarrow \mathbb{R}$ as the sum of spaCy's similarities for each string involved in the two triples. Personas and entities are scalar values, so their similarity is immediate using spaCy. As each triple contains a set of actions, we have to normalize the similarities between the elements in each set of actions to produce this value. As a consequence the value returned by σ is a real number in $[0,3]$.

Validation experiment. Using our reference dataset the graph transformation produced $2,973$ triples out of the $1,671$ stories. There are $\mathcal{O}(n^2)$ pairs of triples, leaving us with an exploration space of more than eight million comparisons. We applied three heuristics for the execution of the similarity computation: *(i)* we did not compute intra-backlog comparisons, *(ii)* we relied on the commutativity of the σ operator to only compute half of the measurements, and *(iii)* we parallelized the computation over 40 processors. In the end, it left us with $4,114,765$ measurements, taking close to 122 CPU hours of computation using the Compute Canada cluster (9.3 measurements/second/proc on average). This approach supports the incremental definition of backlogs: adding new stories means only computing the similarity for the impacted subset of triples without changing anything for the already computed ones, and in the worst case, analyzing the complete models for 22 product took three hours in a parallel environment, making it possible to run such an analysis as a weekly task, for example.

Conclusions. We depict in FIG. 11 the distribution of similarities for each pair of backlogs present in the dataset (231 pairs). Using a Kolmogorov-Smirnov test to measure the differences between these distributions, we identified that 97% of the triples are statistically different according to our implementation of the σ measure (p-value < 0.05). This is frustrating but also reasonable considering that the backlogs collected in the reference dataset come from different origins and specify projects that are coming from very different business domains.

However, two pairs of backlogs exhibit a significant similarity. In FIG. 11, they are visible in the *rug bar*, where all the distributions' averages forms a continuum in the $[0.75, 1.5]$ interval, except these two backlogs that exhibit an average higher than 1.5.

- (g_{17}, g_5): The two systems are related to scientific data publication. Scientists can publish the dataset used to support their experiments in an automated or manual way.
- (g_{23}, g_{26}): The two systems are related to archive management. Archivists can moderate digital collections of artifacts to organize their archive and publish it digitally.

Interestingly, these two pairs are close to each other in terms of rationale, *i.e.*, publish a digital collection of "things". It is also interesting to notice that even if the four systems are related to content management, the one involved in the first pair follows an approach where end-users publish data on their own, manually or through an automated API, where the one involved in the second pair relies on moderation processes. It means that even if simple, the σ function defined over the graph model is precise enough to provide this kind of distinction.

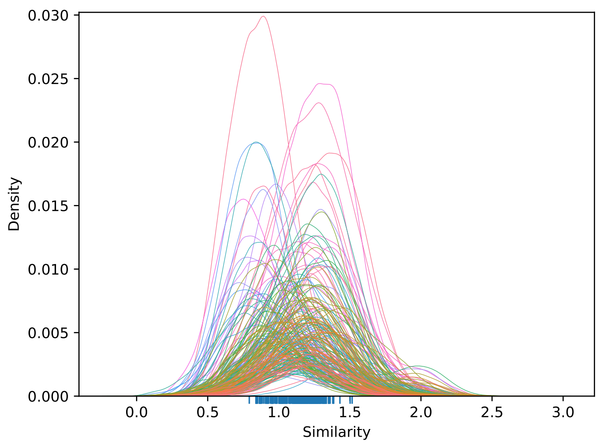


Figure 11 Distribution of similarities for each pair of backlog (231), the rug bars represent the average of each distribution

To win some intuition about what is considered a good match, here is the highest-scoring matching stories between (g_{17}, g_5):

- *"As an API User, I want to have a flexible API using HASC codes for countries, regions and cities, so that I can visualize budget data on maps"*
- *"As an app developer, I want to share a dataset type across multiple applications that include the dataset type's code in their artifacts"*

Both stories have as entity `Code`. The persona in the first one is `API User` and `App Developer` in the second. This highlights the NLP similarity metrics' interest that matched similar personas described with different vocabulary.

5. Threats To Validity

5.1. Threats to Validity

Our objective here is not to demonstrate that a given feedback provider is better than another one or perform an empirical evaluation of the quality of the backlogs available in the reference dataset. To perform such a task, access to the engineers who worked on the different products would have been necessary and access to a qualified version of the dataset (*e.g.*, similarity between stories, iterations planning). Considering that such access does not exist, it is impossible to support such experiments. Moreover, as the products in the dataset are anonymized, it is impossible to reach the companies or open-source projects that worked on them to perform such a qualification.

This is why the ground truths (*e.g.*, the reference equivalence clusters used to measure the story recommendation) were defined by the authors. Unfortunately, this triggers an *external* threat to validity, jeopardizing the result generalization. However, the point here is not to measure a given approach but to show that modelling backlogs can support alternative analysis and let the team choose the one that fits its practices.

Considering this objective of versatility, an *internal* threat is related to the implementation of our proposition, as we relied on tools (*e.g.*, NetworkX, spaCy) that were familiar to the authors. This is mitigated by the fact that the algorithms used would be provided as off-the-shelf components interfaced with the backlog model thanks to a public interface in a production setup. Also, the way we clustered the two backlogs used to demonstrate story recommendation heavily relies on our agile and DevOps development expertise. This is mitigated by the fact that we were not evaluating one approach against the other. Moreover, considering that a backlog reflects the team's practices, designing a customizable solution at this level is essential.

Finally, the approach described in the paper relies on NLP tools to extract the backlog models automatically. As any tool relying on natural language analysis, these tools are not completely accurate and can make mistakes. However, the precision and recall of the tools used to support the experimental part of this paper are among the best of the state of practice. Visual-Narrator exhibits in its reference benchmark precision in the $[92\%, 98\%]$ range, and recall in the $[88\%, 97\%]$ one.

5.2. Discussions & Relations to MDE foundations

This paper describes a modelling approach for backlogs and five applications scenarios built on top of this approach. The application scenarios aim to validate the expressiveness and potential of the formal model. Unfortunately, the agile community is historically hostile to modelling, considering that MDE tools are heavy and unsuitable in an agile context. With this contribution, we aim to demonstrate that models can be integrated inside an Agile/DevOps development cycle by precisely modelling the core of this development paradigm: the backlog.

Taking a step back, it questions our value as model-driven software engineers, more precisely what we take for granted. To integrate modelling inside such a loop and scale top real-life backlogs, we had to give up our classical tooling (*e.g.*, UML models, EMF-based code) and redesign a software stack and approach that was entirely driven by the application domain technological environment. The approach described in the paper goes back to the root of modelling, *i.e.*, identifying the right abstractions. In our case, these abstractions were constrained by the five properties identified in SEC. 3. If P_{1-3} are related to a *technical* level, the last two properties P_4 and P_5 have a strong impact on the modelling choices:

- P_4 advocate for versatility in the processing that will be applied to the models. This *open-world* modelling approach, where models are created with no idea whatsoever of how they will be used for is a new dimension of *design uncertainty* (Famelis & Chechik 2019). It also triggers the need to create model-driven systems that are open to extensions in an easy way, which can be an engineering challenge (*e.g.*, in our case, we had to push all the implementation to a PYTHON environment).
- P_5 insists on a fundamental principle of agility, *i.e.*, being iterative and incremental. This is reflected in the careful formalization of the backlog and the different operators used to manipulate them. However, the link between high-level requirements and technical properties of operators is not always immediate (Benni et al. 2020), and requires tremendous design efforts to ensure that the requirements are fulfilled.

Finally, the "elegance" of the described approach is to solve the particular problem of backlog modelling by going back to modelling fundamentals and considering a model as a graph. This abstract representation provided two immediate benefits: *(i)* being able to switch from one technological ecosystem to another one quickly and *(ii)* being able to rely on a reference graph library to support the low-level implementation of the operators. However, this idea could be reused in other domains. By relying on pure graph structure as modelling backends, it becomes possible to reuse graph-based algorithms, *e.g.*, for recommendation purposes. Relying on standard representation for graphs, *e.g.*, using reference libraries, also foster collaborations and ease the integration of new tools. It is, of course, possible to transform a model (*e.g.*, an EMF model) into a graph and perform computation on top of this graph before coming back to the origin space.

Nevertheless, transforming models back and forth can introduce instability in a toolchain, in addition to the performance costs. We believe that bridging the gap between modelling and computation is essential, as assumed by communities such as the *models@run.time* or the *multi-paradigm modelling* one. Considering the rise of the "artificial intelligence for modelling" topic in the community, pushing our tools and framework to AI-environment instead of relying on transformations might be the next necessary step towards a global acceptance of models recommender for example.

6. Conclusions & Perspectives

This paper showed that reinstating backlogs as first-class citizens allows us to propose early and informative feedback to developers and product owners. Furthermore, we proposed a way to model those product backlogs as constrained graphs formally. Where state-of-the-art approaches considered the backlog a means to an end, we focused on modelling it and considering it as an actionable artifact for analysis purposes. Thanks to this representation, we have shown how to exploit the graph structure to provide immediate feedback to the product team, according to three analysis dimensions: product analysis, iteration planning and portfolio management. To summarize, the contributions of this paper are the following:

1. *A modelling approach for product backlogs*, validated on top of a reference dataset in terms of expressiveness;
2. *A set of scenarios leveraging the model*, demonstrating how versatile it is and making it customizable to a given team's practices;

3. The scenarios are based on *industrial collaborations*, and *covers five out of six obstacles* in product backlog management identified by an external study;

4. A *discussion* of the relation of these two contributions and the model-driven engineering field.

This work opens numerous perspectives, according to different research themes. From a domain point of view, the first theme is related to the *empirical evaluation* of approaches working on backlogs. By having a close relationship with the teams developing the products, it would become possible to deliver to the community a reference standard with automated tooling to evaluate approaches working on the models. The second theme is related to *NLP*. We showed that good results could be obtained using an out-of-the-box implementation of text similarity. A more targeted approach could use constrained syntax trees and natural language modelling techniques (Maupomé & Meurs 2020) to provide more precise model importers. Another interesting research theme is related to the definition of *graph similarity algorithms* on top of the models. To date, we relied on algorithms implemented in PYTHON libraries, but a sustainable approach should rely on something less adherent to code. The research challenges here are identifying the different dimensions of similarity, implementing them efficiently, and exposing such dimensions so that a development team can customize them to fit its practices.

Finally, all the previous themes trigger a *variability management* challenge. The DevOps team is now confronted with many decisions to find the right way to use the model in its feedback loop. Thus, software product lines techniques will have to be explored to guide the team during the configuration process.

Acknowledgments

The work described in this paper is supported by a Discovery grant from the *Natural Sciences and Engineering Research Council of Canada* (NSERC, RGPIN-2020-05791). The authors want to thanks Fabian Gilson (University of Canterbury, NZ) for his initial inputs and dataset availability, as well as Benjamin Benni (Instant Systems), Guilhem Molines (IBM), Enzo Dalla-Nora, Florian Bourniquel, and David Sene, who participated in the implementation of the first proof of concept. This research was enabled in part by support provided by Calcul Québec, Compute Canada and DELL.

References

Athiththan, K., Rovinsan, S., Sathveegan, S., Gunasekaran, N., Gunawardena, K. S. A. W., & Kasthurirathna, D. (2018). An ontology-based approach to automate the software development process. In *2018 ieee international conference on information and automation for sustainability (iciafs)* (p. 1-6). doi: 10.1109/ICIAFS.2018.8913339

Barbosa, R., Silva, A. E. A., & Moraes, R. (2016). Use of similarity measure to suggest the existence of duplicate user stories in the srum process. In *2016 46th annual ieee/ifip international conference on dependable systems and networks workshop (dsn-w)* (p. 2-5). doi: 10.1109/DSN-W.2016.27

Benni, B., Mosser, S., Acher, M., & Paillart, M. (2020). Characterizing black-box composition operators via generated tailored benchmarks. *J. Object Technol.*, *19*(2), 7:1–20. Retrieved from https://doi.org/10.5381/jot.2020.19.2.a7 doi: 10.5381/jot.2020.19.2.a7

Bordeleau, F., Cabot, J., Dingel, J., Rabil, B. S., & Renaud, P. (2020). Towards modeling framework for devops: Requirements derived from industry use case. In J.-M. Bruel, M. Mazzara, & B. Meyer (Eds.), *Software engineering aspects of continuous development and new paradigms of software production and deployment* (pp. 139–151). Cham: Springer International Publishing.

Combemale, B., Kienzle, J., Mussbacher, G., Ali, H., Amyot, D., Bagherzadeh, M., ... Wimmer, M. (2021). A hitchhiker's guide to model-driven engineering for data-centric systems. *IEEE Softw.*, *38*(4), 71–84. Retrieved from https://doi.org/10.1109/MS.2020.2995125 doi: 10.1109/MS.2020.2995125

Dalpiaz, F. (2018). *Requirements Data Sets (User Stories).* Retrieved from https://data.mendeley.com/datasets/7zbk8zsd8y/1 (Mendeley Data (v1)) doi: 10.17632/7zbk8zsd8y.1

Dalpiaz, F., Schalk, I. V. D., Brinkkemper, S., Aydemir, F. B., & Lucassen, G. (2019). Detecting terminological ambiguity in user stories: Tool and experimentation. *Inf. Softw. Technol.*, *110*, 3–16. Retrieved from https://doi.org/10.1016/j.infsof.2018.12.007 doi: 10.1016/j.infsof.2018.12.007

Duarte, V. (2016). *NoEstimates: How To Measure Project Progress Without Estimating (1st edition).* OikosofySeries.

Elallaoui, M., Nafil, K., & Touahni, R. (2015). Automatic generation of uml sequence diagrams from user stories in scrum process. In *2015 10th international conference on intelligent systems: Theories and applications (sita)* (p. 1-6). doi: 10.1109/SITA.2015.7358415

Elallaoui, M., Nafil, K., & Touahni, R. (2018). Automatic transformation of user stories into uml use case diagrams using nlp techniques. *Procedia Computer Science*, *130*, 42-49. Retrieved from https://www.sciencedirect.com/science/article/pii/S1877050918303600 (The 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / The 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) / Affiliated Workshops) doi: https://doi.org/10.1016/j.procs.2018.04.010

Famelis, M., & Chechik, M. (2019). Managing design-time uncertainty. *Softw. Syst. Model.*, *18*(2), 1249–1284. Retrieved from https://doi.org/10.1007/s10270-017-0594-9 doi: 10.1007/s10270-017-0594-9

Galster, M., Gilson, F., & Georis, F. (2019). What quality attributes can we find in product backlogs? A machine learning perspective. In T. Bures, L. Duchien, & P. Inverardi (Eds.), *Software architecture - 13th european conference, ECSA 2019, paris, france, september 9-13, 2019, proceedings* (Vol. 11681, pp. 88–96). Springer. Retrieved from https://doi.org/10.1007/978-3-030-29983-5_6 doi: 10.1007/978-3-030-29983-5_6

Gilson, F., Galster, M., & Georis, F. (2020). Generating use case scenarios from user stories. In *Proceedings of the international conference on software and system processes* (p. 31–40). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3379177.3388895 doi: 10.1145/3379177.3388895

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using networkx. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th python in science conference* (p. 11 - 15). Pasadena, CA USA.

Honnibal, M., Montani, I., Van Landeghem, S., & Boyd, A. (2020). *spaCy: Industrial-strength Natural Language Processing in Python.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.1212303 doi: 10.5281/zenodo.1212303

Kienzle, J., Mussbacher, G., Combemale, B., & DeAntoni, J. (2019). A unifying framework for homogeneous model composition. *Softw. Syst. Model.*, *18*(5), 3005–3023. Retrieved from https://doi.org/10.1007/s10270-018-00707-8 doi: 10.1007/s10270-018-00707-8

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations* (1st Edition ed.). Portland, OR, USA: IT Revolution Press.

Landhäußer, M., & Genaid, A. (2012). Connecting user stories and code for test development. In *2012 third international workshop on recommendation systems for software engineering (rsse)* (p. 33-37). doi: 10.1109/RSSE.2012.6233406

Lucassen, G., Dalpiaz, F., van der Werf, J. M. E. M., & Brinkkemper, S. (2016). Improving agile requirements: the quality user story framework and tool. *Requir. Eng.*, *21*(3), 383–403. Retrieved from https://doi.org/10.1007/s00766-016-0250-x doi: 10.1007/s00766-016-0250-x

Lucassen, G., Dalpiaz, F., Werf, J. M. E. M. v. d., & Brinkkemper, S. (2016). The use and effectiveness of user stories in practice. In M. Daneva & O. Pastor (Eds.), *Requirements engineering: Foundation for software quality* (pp. 205–222). Cham: Springer International Publishing.

Lucassen, G., Robeer, M., Dalpiaz, F., van der Werf, J. M. E. M., & Brinkkemper, S. (2017). Extracting conceptual models from user stories with visual narrator. *Requir. Eng.*, *22*(3), 339–358. Retrieved from https://doi.org/10.1007/s00766-017-0270-1 doi: 10.1007/s00766-017-0270-1

Maupomé, D., & Meurs, M. (2020). Language modeling with a general second-order RNN. In *Proceedings of the 12th language resources and evaluation conference, LREC 2020, marseille, france, may 11-16, 2020* (pp. 4749–4753). European Language Resources Association. Retrieved from https://www.aclweb.org/anthology/2020.lrec-1.584/

Nasiri, S., Rhazali, Y., Lahmer, M., & Chenfour, N. (2020). Towards a generation of class diagram from user stories in agile methods. *Procedia Computer Science*, *170*, 831-837. Retrieved from https://www.sciencedirect.com/science/article/pii/S1877050920306049 (The 11th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 3rd International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops) doi: https://doi.org/10.1016/j.procs.2020.03.148

Parés, F., Gasulla, D. G., Vilalta, A., Moreno, J., Ayguadé, E., Labarta, J., … Suzumura, T. (2017). Fluid communities: A competitive, scalable and diverse community detection algorithm. In *International conference on complex networks and their applications* (pp. 229–240).

Raharjana, I. K., Siahaan, D., & Fatichah, C. (2021). User stories and natural language processing: A systematic literature review. *IEEE Access*, *9*, 53811–53826. Retrieved from https://doi.org/10.1109/ACCESS.2021.3070606 doi: 10.1109/ACCESS.2021.3070606

Robeer, M., Lucassen, G., van der Werf, J. M. E. M., Dalpiaz, F., & Brinkkemper, S. (2016). Automated extraction of conceptual models from user stories via NLP. In *24th IEEE international requirements engineering conference, RE 2016, beijing, china, september 12-16, 2016* (pp. 196–205). IEEE Computer Society. Retrieved from https://doi.org/10.1109/RE.2016.40 doi: 10.1109/RE.2016.40

Rodeghero, P., Jiang, S., Armaly, A., & McMillan, C. (2017). Detecting user story information in developer-client conversations to generate extractive summaries. In *2017 ieee/acm 39th international conference on software engineering (icse)* (p. 49-59). doi: 10.1109/ICSE.2017.13

Sandeep RC. (2020). *Estimation Techniques in Agile Software Development* (Unpublished master's thesis). Høgskolen i Østfold, Norway.

Sedano, T., Ralph, P., & Péraire, C. (2019). The product backlog. In J. M. Atlee, T. Bultan, & J. Whittle (Eds.), *Proceedings of the 41st international conference on software engineering, ICSE 2019, montreal, qc, canada, may 25-31, 2019* (pp. 200–211). IEEE / ACM. Retrieved from https://doi.org/10.1109/ICSE.2019.00036 doi: 10.1109/ICSE.2019.00036

Sokal, R. R. (1958). A statistical method for evaluating systematic relationships. *Univ. Kansas, Sci. Bull.*, *38*, 1409–1438.

Zager, L. A., & Verghese, G. C. (2008). Graph similarity scoring and matching. *Appl. Math. Lett.*, *21*(1), 86–94. Retrieved from https://doi.org/10.1016/j.aml.2007.01.006 doi: 10.1016/j.aml.2007.01.006

About the authors

Sébastien Mosser is Associate Professor at McMaster University (Canada), and a member of the McMaster Centre for Software Certification (McSCert). His research focus on domain-specific languages and distributed systems (*e.g.*, microservices), using software composition as a mean to achieve scalability. You can contact him at You can contact the author at mossers@mcmaster.ca..

Corinne Pulgar is an M.A.Sc. student at *École de Technologie Supérieure* (ETS) in Montréal, Canada, and a member of the Cool lab. Her research interest covers DevOps pipelines and argumentation models. She is a member of the DevOps Industrial Research Chair hosted by ETS, in collaboration with TELUS and Kaloom. You can contact her at You can contact the author at corinne.pulgar.1@ens.etsmtl.ca..

Vladimir Reinharz is professor of Bioinformatics at *Université du Québec à Montréal* (UQAM, Canada), and a founding member of the Cool lab. His research focuses on Computational

Biology and Evolution, particularly the design of new graph-based algorithms to study RNA molecules and viral infections. You can contact the author at reinharz.vladimir@uqam.ca.