# Comprehensive Characterization of Concept Drifts in Process Mining

Alexander Kraus[a,*], Han van der Aa[b]

[a] *Data and Web Science Group, University of Mannheim, B6 26, 68159 Mannheim, Germany*
[b] *Faculty of Computer Science, University of Vienna, Währinger Str. 29, 1090 Vienna, Austria*

**Abstract**

Business processes are subject to changes due to the dynamic environments in which they are executed. These process changes can lead to concept drifts, which are situations when the characteristics of a business process have undergone significant changes, resulting in event logs that contain data on different versions of a process.he accuracy and usefulness of process mining results derived from such event logs may be compromised because they rely on historical data that no longer reflects the current process behavior, or because the results do not distinguish between different process versions. Therefore, concept drift detection in process mining aims to identify drifts recorded in an event log by detecting when they occurred, localizing process modifications, and characterizing how they manifest over time.his paper focuses on the latter task, i.e., drift characterization, which seeks to understand whether changes unfolded suddenly or gradually and if they form complex patterns like incremental or recurring drifts. However, current solutions for automatically detecting concept drifts from event logs lack comprehensive characterization capabilities. Instead, they mainly focus on drift detection and characterization of isolated process changes. This leads to an incomplete understanding of more complex concept drifts, like incremental and recurring drifts, when several process changes are inter-connected. This paper overcomes such limitations by introducing an improved taxonomy for characterizing concept drifts and a three-step framework that provides an automatic characterization of concept drifts from event logs. We evaluated our framework through elaborate evaluation experiments conducted using a large collection of synthetic event logs. The results highlight the effectiveness and accuracy of our proposed framework and show that it outperforms state-of-the-art techniques.

*Keywords:* Process mining, concept drift detection, drift characterization

## 1. Introduction

Information systems that support the execution of business processes often generate data in the form of *event logs* [1]. These logs contain sequences of events that describe the execution of process instances over

*Corresponding author
*Email addresses:* alexander.kraus@uni-mannheim.de (Alexander Kraus), han.van.der.aa@univie.ac.at (Han van der Aa)

a specific period of time. They are valuable sources of information for operational analysis, as they describe various aspects of the process, including the activities conducted, their timing, the people involved, and other relevant process details. *Process mining* uses these event logs to facilitate a wide range of analytical inquiries [2]. Organizations can leverage event logs and process mining to extract valuable insights, make informed decisions, and optimize their processes by constructing process models, verifying execution against specifications, and creating simulation, prediction, and recommendation models.

Crucially, business processes are subject to change over time due to various internal and external factors, such as organizational adjustments, process enhancements, policy updates, and technological advancements. These changes can introduce *concept drifts*, which are situations when the characteristics of a business process have undergone significant changes [3] during the period when process execution data has been recorded, resulting in event logs that contain information on different versions of a process. The presence of such drifts in event logs can have a detrimental impact on the accuracy and usefulness of process mining results as these will be (partially) based on historical data that no longer represents the current process.Therefore, to avoid incorrect or even misleadingrocess mining results, *concept drift detection* aims to identify and characterize concept drifts from event logs, striving to understand how a recorded process evolved over time. To achieve this, concept drift detection addresses the following three key tasks [4]: (1) *drift detection*, which involves detecting when drifts occurred, (2) *drift localization*, which aims to describe what was modified in the process, and (3) *drift characterization*, which considers how drifts manifest themselves over time. In this paper, we particularly focus on the latter task, i.e., drift characterization, which seeks to understand how drifts in event logs unfold over time, i.e., whether they occur suddenly or gradually, and whether or not changes jointly form more complex patterns in the form of incremental or recurring drifts.

Despite numerous proposed solutions to automatically detect concept drifts [5], none of the existing techniques can comprehensively characterize drifts in event logs [6]. Specifically, most existing techniques focus on detecting isolated process changes that lead to sudden drifts [7–14], with few others also differentiating between sudden and gradual ones [4, 15–17]. These techniques thus only provide a partial picture of detected concept drifts since they cannot recognize inter-relations between process changes, which lead to more complex drifts in the form of incremental and recurring drifts. However, in many process mining tasks, ignoring such complex drifts can lead to misleading results. For instance, applying existing concept drift detection techniques on a process may reveal that it went through a considerable amount of changes, each leading to possibly different process versions that should be analyzed separately. However, by considering the inter-relations of these changes, it may become clear that this process is in fact subject to a seasonal pattern in which just two process versions alternate. Subsequent analyses can then target each of these versions individually. Similarly, performance analysis may be biased if the event data includes an incremental drift. In this case, proper performance analysis should separately consider recorded process behavior before the incremental drift and after it, while disregarding intermediate behavior that occurs during the period of the

incremental drift.s a result, the state of the art provides only incomplete insights into the actual evolution of business processes over time.

This paper addresses this limitation through two contributions. First, we propose an improved taxonomy that can be used as a basis for the comprehensive characterization of concept drifts, since we recognize that existing works are not just limited in their scope, but are actually grounded on imprecise and incomplete definitions. Second, we propose a three-step framework that automatically characterizes detected drifts in a comprehensive manner, following our proposed taxonomy. Our framework starts with the detection of isolated change points in the event log, marking significant shifts in process behavior. Next, using our change type detection algorithm, we identify actual process changes and categorize them as sudden or gradual. Finally, we determine concept drifts and their types from the detected process changes using our change inter-relation detection algorithm. Conducted evaluation experiments show the accuracy of the developed framework and its algorithms compared to state-of-the-art techniques.

In the remainder, Section 2 provides an introduction to the concept drift characterization, current limitations, and our improved taxonomy. Section 3 explains our framework steps and proposed algorithms. Our evaluation results are summarized in Section 4. Finally, we conclude in Section 5.

## 2. Concept Drift Characterization: Current Limitations and Improved Taxonomy

This section proposes a new taxonomy for the comprehensive characterization of concept drifts in process mining, which overcomes the limitations of existing definitions and can be used to accurately reflect the scope (and limits) of existing concept drift detection techniques. To achieve this, we first introduce preliminaries in Section 2.1. In Section 2.2, we examine the current taxonomy used to characterize concept drifts, along with its limitations, which are detailed in Section 2.3. Then, we presents our proposed, improved taxonomy in Section 2.4, which we use as a foundation in Section 2.5 to provide an overview of previous work on concept drift detection and characterization.

### *2.1. Preliminaries*

**Business process.** A *business process* is a set of activities and constraints between them that are performed within an organizational and technical environment to realize a business goals [18]. Business processes are often visualized using a specific process modeling language. For example, Figure 1 presents a simple business process model created using the Business Process Model and Notation (BPMN) [19], a widely recognized standard for business process modeling. Focusing on the control-flow of a business process, this model features a start event labeled "Order received," five business process activities, and an end event marked "Order dispatched." The activities are carried out in sequence, with a single decision point that allows for a choice between two activities following the first process activity.

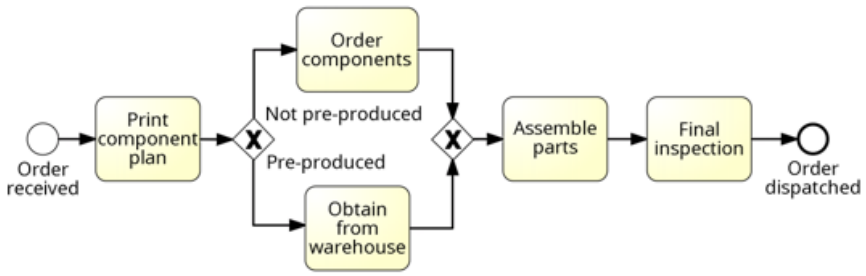



Figure 1: Example of a business process model.

**Event log.** An *event log* $L$ is a collection of events recorded by a process-aware information system during process execution. Each event $e \in L$ is represented as a tuple with at least three attributes $e := (\text{caseID}, \text{activity}, \text{timestamps})$, where caseID is the unique identifier for the executed case, activity indicates the executed process activity, and timestamps denotes the event moment. Table 1 presents a snapshot of an event log generated from the execution of the business process illustrated in Figure 1. In addition to the attributes caseID, timestamp, and activity, the event log also records additional attributes such as resource information and cost.

Table 1: Example of an event log.

| Case ID | Timestamp | Activity | Resource | Cost |
|---|---|---|---|---|
| 1 | 29-12-2010 14:17 | Order received | Susana | 0 |
| 1 | 30-12-2010 11:02 | Print component plan | Mike | 50 |
| 1 | 31-12-2010 10:06 | Obtain from warehouse | Sue | 200 |
| 1 | 05-01-2011 15:12 | Assemble parts | Mike | 100 |
| 1 | 06-01-2011 11:18 | Final inspection | Sara | 200 |
| 1 | 07-01-2011 10:00 | Order dispatched | Sara | 80 |
| 2 | 30-12-2010 15:34 | Order received | Susana | 0 |
| 2 | 30-12-2010 11:32 | Print component plan | Mike | 50 |
| 2 | 30-12-2010 12:12 | Order components | Mike | 100 |
| 2 | 30-12-2010 14:16 | Assemble parts | Pete | 400 |
| 2 | 05-01-2011 11:22 | Final inspection | Sara | 200 |
| 2 | 06-01-2011 11:33 | Order dispatched | Sara | 80 |
| ... | ... | ... | ... | ... |

**Trace.** A *trace* $\sigma$ is a sequence of events from $L$ with the same caseID, ordered by their timestamps. We denote $\Sigma_L$ as the ordered collection of traces, arranged according to the timestamp of their first event. Based on this ordering, each trace is assigned a trace index $i \in \mathcal{I}$ with $\mathcal{I} := [1, 2, \ldots, |\Sigma_L|]$, s.t., $\sigma_i$ denotes the $i$-th trace in $\Sigma_L$.

**Behavioral representation.** A *behavioral representation* of a process is a set of defined behavioral relations and their support (e.g., frequency) that characterize a process based on data from an event log $L$. A

commonly used type of behavioral representation in process mining is the directly-follows relation [20], which captures the frequency with which two activities are observed to immediately succeed one another within the same case. For example, using the two cases shown in Table 1, we can derive the following directly-follows relations with abbreviated activity names: $(OR \rightarrow PCP)$: 2, $(PCP \rightarrow OfW)$: 1, $(OfW \rightarrow AP)$: 1, $(AP \rightarrow FI)$: 2, $(PCP \rightarrow OC)$: 1, $(OC \rightarrow AP)$: 1, $(FI \rightarrow OD)$: 2. Other behavioral representations, such as eventually-follows relations [20], $\alpha$-relations [20], behavioral profiles [21], or declarative process constraints [22], are also frequently applied to capture different aspects of process behavior.

### *2.2. Status Quo Taxonomy*

This section provides a brief overview of concept drift detection and the current definitions (i.e., the *status quo taxonomy*) used to categorize and understand different types of concept drifts in process mining, which is provided by Bose et al. [4].

In the context of process mining, a *concept drift* refers to a situation in which a process is changing while being analyzed [4], which happens when an event log contains data stemming from different process versions. C3 Figure 2 illustrates an example of a concept drift. In this example, the original process version is replaced with the new one that rejects the order if its components have not been pre-produced.

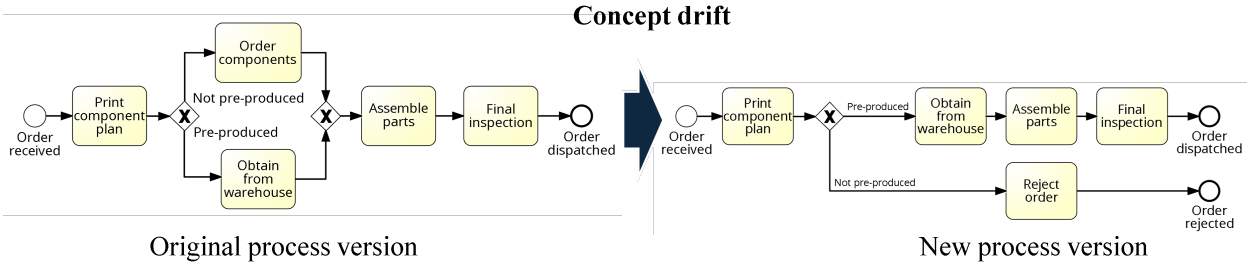



Figure 2: Example of a concept drift.

*Concept drift detection* aims to identify such changes based on event logs to obtain a comprehensive understanding of the overall evolution of a process over time. To achieve this, concept drift detection addresses the following key tasks [4]: (i) *drift detection*, which involves detecting when drifts occurred, (ii) *drift localization*, which aims to describe the process perspective(s) (control-flow, data, time, and data) affected by and specific modifications made to the process during a drift, and (iii) *drift characterization*, which considers how drifts manifest themselves over time (e.g., suddenly versus gradually).

Our work primarily focuses on this latter task, i.e., change characterization, since this is, at best, only partially covered by existing works (see Section 2.5). In the taxonomy by Bose et al. [4], change characterization primarily focuses on classifying drifts into one of four *drift types* [4], which are illustrated in Figure 3:

1. A *sudden drift* occurs when a current process version is entirely replaced by a new one at a specific

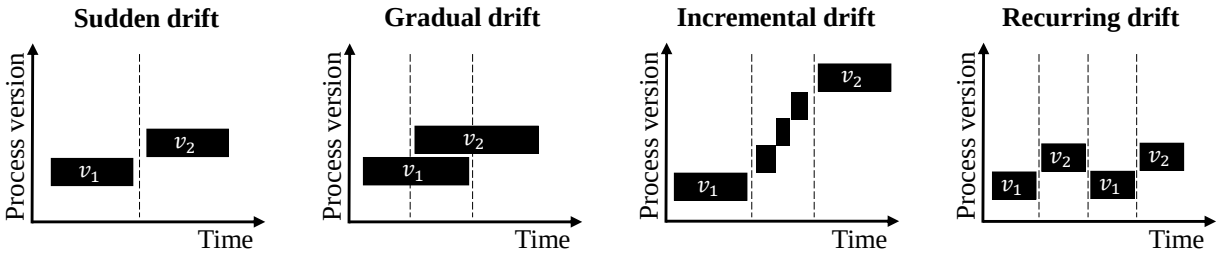



Figure 3: Concept drift types (adapted from [4]).

moment, and the new one takes over all ongoing cases [4]. This type of drift can occur in emergencies or when new regulations must be followed [6].

2. A *gradual drift* occurs when a current process version is replaced by a new one, and both versions coexist during a transition period [4]. Throughout this transition period, an increasing number of process instances begin to follow the new process version until the point at which the new version operates exclusively.
3. An *incremental drift* occurs when a current process version is replaced by a new one via smaller incremental changes [4]. For instance, this drift type occurs in organizations implementing successive business process quality improvements as part of a larger initiative.
4. A *recurring drift* occurs when a set of process versions reappear after some time [4]. Recurring drifts can occur in two forms: periodic and non-periodic. Periodic drifts follow seasonal patterns, such as reduced demand and resource needs during the summer holidays, whereas non-periodic drifts stem from changes that do not recur according to predetermined times, but rather from other conditions, such as a change in a process that is implemented when the workload is too high.

Other aspects, such as momentary and permanent changes [4] or multi-order dynamics [15], are sometimes used for drift characterization but have received comparatively less attention and study.

### *2.3. Limitation of the Status Quo Taxonomy*

The aforementioned status quo taxonomy has several key limitations that hinder its usefulness when aiming to properly characterize concept drifts:

**L1: Non-exclusive drift type classification.** The four drift types defined by Bose et al. [4] are not mutually exclusive because they encompass two different levels of granularity. Specifically, sudden and gradual drifts characterize how individual process changes manifest themselves, whereas incremental and recurring drifts connect several process changes to each other. As a result, a single concept drift can be an incremental or recurring drift, but consist of individual sudden or gradual (or both) changes, as visualized

in Figure 4[1]. Due to this limitation, drifts cannot be properly characterized when using the existing four drift types, since we either lose information at the high level, i.e., how changes are connected, or at the low level, i.e., if individual changes in a recurring or incremental drift occurred suddenly or gradually.

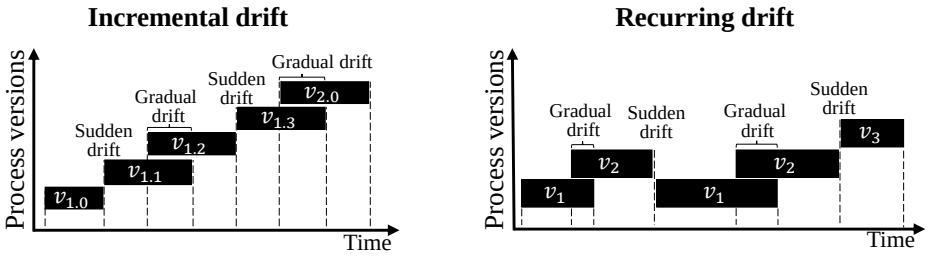



Figure 4: Incremental and recurring drifts can consist of sudden and gradual drifts, leading to non-exclusive classification (L1).

**L2: Imprecise definition of incremental drifts.** The definition of incremental drifts is imprecise, which means that it is not always possible to deterministically assign a drift type to specific observations. Specifically, the existing definition does not specify what makes a change incremental and, therefore, cannot be used to differentiate between a series of unrelated, relatively small drifts and a single, incremental drift, as, e.g., visualized in Figure 5.

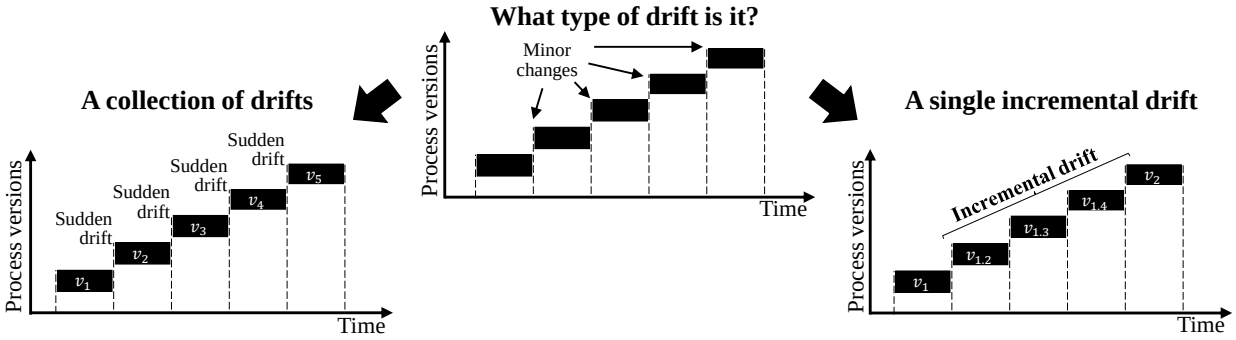



Figure 5: Minor changes can form an incremental drift, but also a sequence of independent sudden and gradual drifts (L2).

**L3: Incomplete definition of recurring drifts.** Finally, the definition of recurring drifts is incomplete (and imprecise) because it does not align with the examples that are used to illustrate these drifts, essentially making the definition too narrow. For example, the definition, which requires a set of process versions to reappear, would exclude situations in which a single process version reappears every so often (whether periodically or not), as, e.g., visualized in case c) of Figure 6. In addition, if there are two recurring drifts in an event log, e.g., as depicted in case d) of the figure, then the existing definition would classify them as a single recurring drift, whereas it would be more precise to state that there are two recurring drifts, one involving versions $v_1$ and $v_2$ and another involving $v_3$ and $v_4$.

[1]Note that Bose et al. [4] indeed remark on this aspect, but it is not picked up by subsequent works.

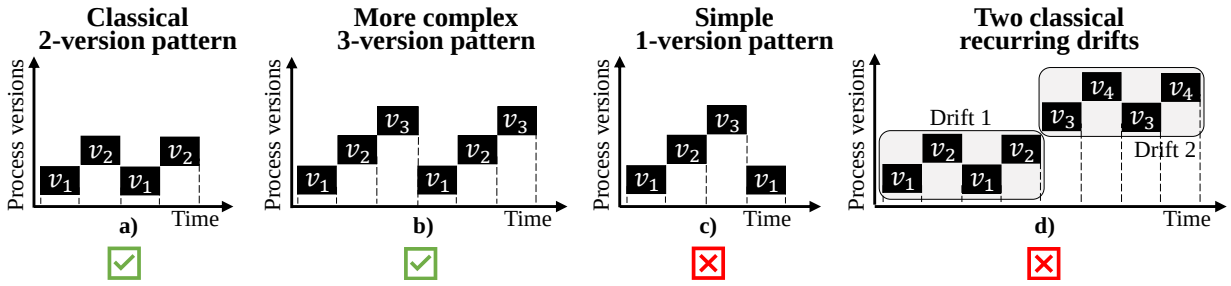



Figure 6: Instances of recurring drifts that align with the current definition (a, b) or do not align (c, d) (L2).

Due to these limitations, concept drift characterization has mainly centered on detecting isolated process changes and their types (sudden vs. gradual), overlooking more "complex" drifts, like recurring and incremental drift types, especially those involving lower-level sudden and gradual changes (see Section 2.5 for an overview).

### *2.4. A New Concept Drift Characterization Taxonomy*

To overcome current limitations, we propose a new taxonomy to achieve a more clear and comprehensive characterization of concept drifts. The novelty of our taxonomy lies in its distinction between simple and complex drifts and its recognition that any change that forms or is part of a drift can be either sudden or gradual. In this manner, it overcomes the limitations observed in the status quo taxonomy.

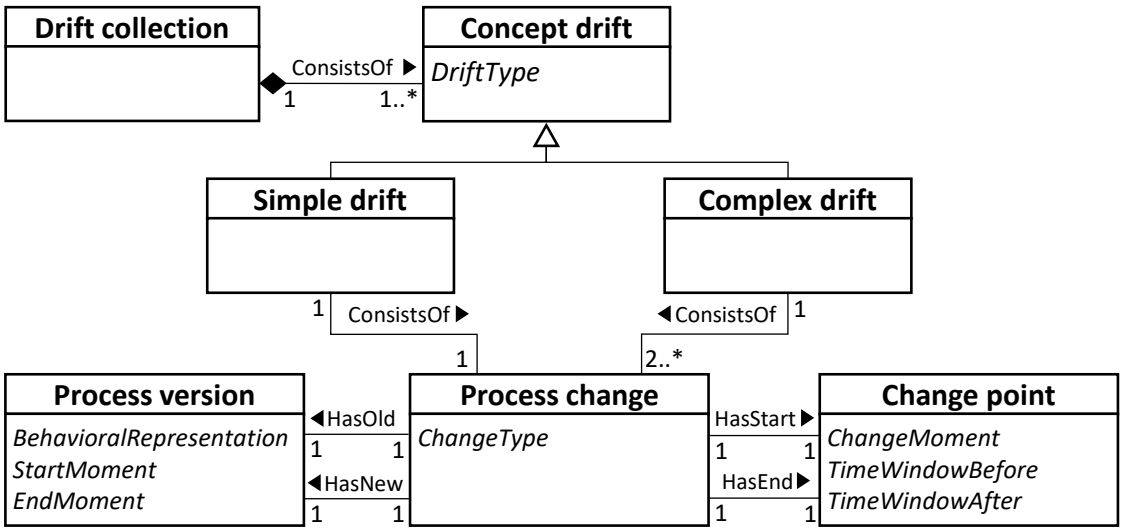



Figure 7: Our taxonomy for concept drift characterization.

In the following, we explain our taxonomy, visualized in Figure 7, in detail, and discuss how it addresses the identified limitations.

*2.4.1. Key Concepts of the New Taxonomy*

As shown in Figure 7, a central concept in our taxonomy is a *process change*. We formalize the concept of process change and its characteristics as follows:

C3

From the behavioral representations of the process versions associated with a process change, we can extract further essential characteristics for drift characterization and other concept drift detection tasks:

- *Change Severity* determines how much a process changed, which can be used, e.g., to detect minor changes when searching for incremental drifts. It can be computed using a function that takes the behavioral representation of old and new process versions as input and produces a value in the range (0, 1). Values closer to 0 indicate low severity, while values closer to 1 signify high severity.
- *Change Localization* identifies specific process modifications when comparing old and new process versions. This information is valuable to address the task of localization changes in concept drift detection. Depending on the behavioral representation, change localization can be quantified by examining sets of behavioral relations or patterns that are removed, introduced, or modified.
- *Change Perspective* reveals the affected process perspectives: control-flow, time, resource, and data. This determination aligns with the concept of perspective of change, as discussed by Bose et al. [4], and relies on the selected behavioral representation techniques.

One or more connected process changes constitute a concept drift, which we formalize as follows:

C3

Given the aforementioned concepts, the objective of concept drift characterization is to establish a drift collection $D$ from an event log $L$, such that $D$ provides a comprehensive characterization of the concept drifts contained in $L$ according to the taxonomy of Figure 7.

*2.4.2. Benefits of the New Taxonomy*

Our taxonomy jointly addresses the limitations of the status quo taxonomy. First, our taxonomy addresses the first limitation (L1) by introducing a mutually exclusive classification of drift types that is based on a process change as a building block for any concept drift and the number of process changes that belong to a drift (simple and complex drifts). Complex drifts (recurring and incremental), by definition, consist of several process changes, where each process change can happen suddenly or gradually, resolving the issues depicted in Figure 4.

We improve the definitions of recurring and incremental drift types, addressing limitations L2 and L3 by specifying how process changes should be related and form complex drifts. In the case of an incremental drift, the proposed notion of a shared business drive, like a BPM initiative to improve the average lead

time of a business process, introduces a clear manner to differentiate an incremental drift from a sequence of standalone disconnected process changes. It also opens new opportunities for how incremental drifts can be detected, i.e., the necessary condition of at least two consecutive process changes with low severity can be extended with further conditions or restrictions. For instance, an additional condition could be related to time, i.e., a sequence of process changes should occur within a certain period, or they should be close in terms of change localization (i.e., which parts of a process are changed). In the case of recurring drifts, our definition of a recurring drift as a pattern of one or more process versions that reappear at least one time covers all possible recurring drift instances, including those depicted in Figure 6.

### *2.5. Related Work*

In this section, we discuss existing concept drift detection techniques in light of our taxonomy, specifically showing their coverage of the different change and drift types. Given the scope of the addressed problem, we focus on offline detection techniques that use event logs as input and yield information about the type of change or drift as output. We exclude online concept drift detection techniques that rely on event streams from consideration, as this problem setting introduces additional constraints and limitations (such as increased computational overhead, real-time processing requirements, and the need for continuous monitoring) that make direct comparisons with our framework and other offline techniques unreasonable.

Table 2: Classification of different concept drift detection and characterization techniques according to our taxonomy.

| **Technique** | **Change point detection** | **Change type detection** | **Drift type detection** | | | |
|---|---|---|---|---|---|---|
| | | | **Simple drifts** | | **Complex drifts** | |
| | | | **Sudden** | **Gradual** | **Incremental** | **Recurring** |
| Various works [7–14] | ✶✶ | | | | | |
| Bose et al. [4] | ✶✶ | ✶ | ✶ | ✶ | | |
| Martushev et al. [15] | ✶✶ | ✶ | ✶ | ✶ | | |
| Maaradji et al. [16] | ✶✶ | ✶✶ | ✶✶ | ✶✶ | | |
| Yeshchenko at el. [17] | ✶✶ | ✶ | ✶✶ | (✶) | (✶) | (✶) |
| Our work | ✶✶ | ✶✶ | ✶✶ | ✶✶ | ✶✶ | ✶✶ |

*Legend: "✶✶" - automated, "✶" - semi-automated, "(✶)" - non-automated.*

Table 2 provides an overview of the scope and automation level of existing drift detection and characterization techniques. Since establishing the problem and importance of concept drift detection in process mining more than a decade ago [23], various techniques have been proposed to detect and characterize them, as highlighted in recent literature reviews [5, 6]. However, the majority of existing techniques detect change points in an event log, aiming to identify a moment when a process behavior significantly changes [7–15]. In terms of drift characterization, they do not contribute to the understanding of the underlying drift types.

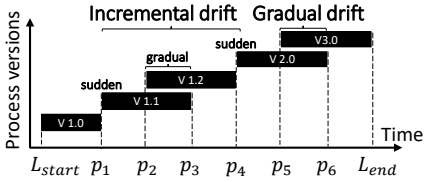
**Change type detection.** When it comes to detecting not only process change points but also process change types, there are two techniques that distinguish between gradual and sudden changes:

Bose et al. [4] introduced concept drift detection in process mining and presented a method for automatically detecting process change types. Their approach uses statistical testing of feature vectors. However, users should indicate which change type should be searched for, i.e., the techniques can only detect sudden or gradual drifts, not both. The method lacks automation, requiring user manual feature selection, assuming prior knowledge of drift characteristics. Additionally, testing all possible activity combinations can also be computationally demanding. Finally, users must specify a window size for drift detection, potentially missing some drift occurrences. To address the window size limitation, Martushev et al. [15] introduced adaptive windowing, which automatically adjusts the window size when searching for drifts. However, this approach requires users to define minimum and maximum window size parameters as upper and lower boundaries for automated window adaptation. Given the mentioned constraints, both techniques have limited capability in detecting process change types, especially in an automated manner.

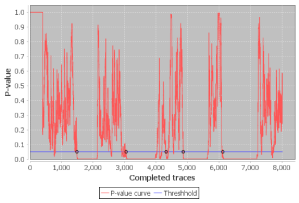
The work by Maaradji et al. [16] introduced an alternative technique for change type detection, addressing the limitations of Bose's technique. Their method offers an automated and statistically grounded solution for identifying both sudden and gradual process change types, representing the current state of the art in simple drift detection. Their approach involves a two-step process: initially, it detects change points to identify sudden drifts (see Figure 11), and then it employs postprocessing on the output of the sudden drift detection algorithm to detect gradual drifts. Specifically, they analyze the behavior within the intervals between two change points by statistically assessing whether it exhibits a mixture of behavior distributions before and after these points [16]. However, this distribution-based method has a significant drawback. In situations where noise is present in an event log, particularly concerning gradual drift detection, their approach experiences a notable decrease in detection accuracy, as demonstrated in our evaluation (Section 4).

**Drift type detection.** Change type detection techniques can only detect simple drifts. Therefore, their usefulness in practice is notably limited when it comes to drift characterization since event logs might include complex drifts, which is apriori unknown. These techniques do not consider the inter-relations between process changes, leading to a collection of simple drifts if the underlying drift in an event log is complex. We demonstrate it using an exemplary concept drift scenario with two drifts: a complex drift of incremental type followed by a simple drift of gradual type, depicted in Figure 8a. The incremental drift consists of three process changes: two sudden drifts at the change points $p_1$ and $p_4$, and a gradual change between $p_2$ and $p_3$. Figure 8b shows the detected change points using the technique by Maaradji et al. [16]. Given the output, it is impossible to conclude that the first four change points belong to an incremental drift and the last one is an independent gradual drift.

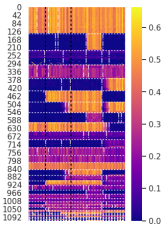
The comprehensive detection and characterization of drift types requires techniques that can simulta-




(a) Concept drift scenario with incremental and gradual drifts.




(b) Outcome of concept drift detection technique for simple drift detection [16].



(c) Outcome of concept drift detection technique for complex drift detection [17].

Figure 8: The outcome of the state-of-the-art concept drift detection techniques cannot comprehensively characterize a complex concept drift.

neously detect simple and complex drifts. The Visual Drift Detection (VDD) technique, introduced by Yeshchenko at el. [17], represents the current state of the art. VDD uses concepts like temporal logic, DECLARE constraints [24], and time series analysis. It groups similar declarative behavioral constraints and automatically identifies change points. The system provides visual aids such as the Drift Map, Drift Charts, and a directly-follows graph. While these visualizations effectively localize change points, it has a significant limitation. The process of identifying simple gradual and complex drifts is not automated and depends on a user's visual interpretation. As a result, recognizing complex drift types or several different drifts within the same event log can be challenging and subjective, limiting the ability to characterize drift types and the overall process evolution. Figure 8c shows the Drift Map, the main output of the VDD technique, for the same concept drift scenario in Figure 8a. Given the visualization, it is impossible to see if detected change points belong to a complex or form a sequence of simple drifts. In our evaluation (Section 4), we further demonstrate these limitations and compare the VDD approach to our work.

Overall, it is thus clear that the comprehensive detection and characterization of concept drifts has not yet been properly addressed. Our framework, described next, overcomes this.

## 3. Concept Drift Detection and Characterization Framework

This section presents our framework for detecting and characterizing simple and complex concept drifts. Section 3.1 introduces the framework at a high level, while Section 3.2-3.4 describe its main steps in detail.

### *3.1. Framework overview*

Figure 9 outlines our proposed framework at a high level, detailing its input, main steps, and output.

**Framework input.** Our framework takes as input an event log $L$. We use $\Sigma_L$ to refer to the set of traces of $L$, where each trace $\sigma \in \Sigma_L$ represents a sequence of events from $L$ with the same case ID, ordered by their


Figure 9: Overview of the main steps of our framework.

timestamps. Finally, we use $p_{first}$ and $p_{last}$ as time points, respectively, corresponding to the timestamps of the first and last events in $L$.

**Framework structure.** Our framework consists of three key steps. Step 1 focuses on the detection of change points in log $L$, for which a range of existing state-of-the-art techniques can be employed In Step 2, our framework turns the sequence of detected change points into a sequence of process changes by differentiating between individual points that correspond to sudden changes and pairs of consecutive change points that indicate a gradual process change. Lastly, in Step 3, we conduct change inter-relation analysis to establish connections between the detected process changes, yielding a collection $D$ of simple and complex drifts.

**Framework output.** Our framework's output is a collection $D$ of identified simple and complex drifts, following the definitions in Section 2.4. Each identified drift is thus associated with a drift type and corresponding process changes, including process change types and the associated change points, providing a comprehensive characterization of the drifts in an event log.

### *3.2. Step 1: Change Point Detection*

The first step of our framework identifies the moments in an event log when process behavior changes, resulting in a sequence of detected change points and the corresponding time windows, as illustrated in Figure 10. This step can be instantiated with any existing technique for change point detection in an event log, as this problem is the most extensively addressed task in concept drift detection, unlike other aspects of our framework, as demonstrated in Section 2.5. Additionally, recent work by Adams et al. [3] underscores the effectiveness of some of these techniques, which we also test and compare in our experimental evaluation in Section 4.

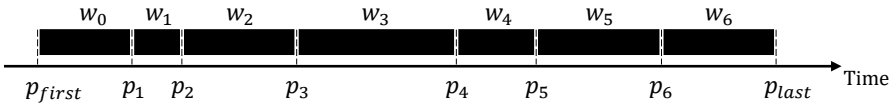



Figure 10: Outcome of the first framework step: change points are detected using an existing change point detection technique, splitting the time frame of an event log into time windows.

Once Step 1 is instantiated using any existing change point detection technique, we obtain a sequence

of detected change points, which we represent as $P := \langle p_1, \ldots, p_N \rangle$.hese change points split the time frame of log $L$ into a sequence of $N + 1$ time windows $W := \langle w_0, \ldots, w_N \rangle$, where $w_0$ represents the time window from $p_{first}$ to $p_1$, and $w_N$ corresponds to the time window from $p_N$ to $p_{last}$. Figure 10 shows the outcome of the first framework step, assuming an event log with six process change points (our running example for this section).

Note that our framework terminates after the first step if $P$ does not contain any change points; otherwise, the framework continues with the next step to reveal process changes from detected change points.

### *3.3. Step 2: Change Type Classification*

The second step of our framework turns the sequence of detected change points into a sequence of sudden and gradual process changes. As illustrated in Figure 11, this involves differentiating between two cases:

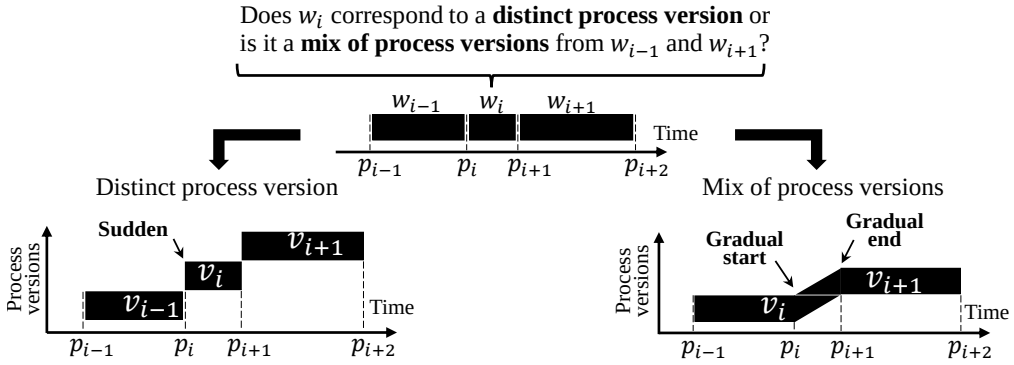



Figure 11: The main idea of the change type classification step.

1. Situations where the behavior that follows a change point $p_i$ (i.e., during window $w_i$) corresponds to a distinct process version, signaling that a sudden change occurred at point $p_i$;
2. Situations where the behavior in $w_i$ reflects a mix of the behavior that occurred before $p_i$ (i.e., in window $w_{i-1}$) and the behavior that happens after the next change point $p_{i+1}$ (i.e., window $w_{i+1}$). This indicates that the behavior observed in window $w_i$ does not correspond to a distinct process version. Rather, it corresponds to a transition period in which the previous process version $v_i$ is shifted out and the new one $v_{i+1}$ is introduced, signifying a gradual process change that starts at $p_i$ and ends at $p_{i+1}$.

To operationalize this idea, we have developed a change type classification technique, presented in Algorithm 1. It determines if a change point belongs to a sudden or gradual process change by considering the evolution of the process behavior before and after the detected change points. It takes an event log $L$ and a sequence of change points $P$, as input and generates a corresponding sequence of $N \leq |P|$ process changes $\overline{C} = \langle c_1, \ldots, c_N \rangle$ as output. Following our definition of a process change in Section 2.4.1, each process change is associated with a change type ("sudden" or "gradual"), two corresponding start and end change points

($p^{start}$ and $p^{end}$), and two process versions ($v^{previous}$ and $v^{next}$). For our running example, the output of this step is shown in Figure 12, which shows that six detected change points describe four process changes.

Next, we explain the main parts of Algorithm 1: behavioral representation and change point classification.

**Behavioral representation.** Our algorithm first computes a behavioral representation to characterize the recorded process behavior during the time windows between detected change points. In our framework, we use directly-follows relations [20] observed within a specified time window, a widely used behavioral representation in process mining for concept drift detection [5]. It involves counting the frequency with which two activities are observed to directly follow one another within a single case.owever, it is important to note that our algorithm's choice of behavioral representation is flexible, provided that it yields a numeric frequency distribution over a predefined set of relations or patterns across the windows. Therefore, it can also cover other types of relations (e.g., eventually follows), sets of relation types, such as those of a behavioral profile [21], or declarative process constraints [22].

To derive a behavioral representation, our algorithm takes an event log $L$ with the sequence of detected change points $P$ as input and then computes a *behavioral matrix*, denoted as $\mathcal{B}$. The behavior matrix consists of columns $\mathcal{B}.windows$ that correspond to the time windows and rows $\mathcal{B}.relations$ that correspond to the behavioral relations. Each cell $\mathcal{B}[b, w]$ of the behavioral matrix corresponds to the relative frequency of a relation $b$ (e.g., a directly-follows relation between two activities) for a window $w$ (line 4). To calculate such a relative frequency, our algorithm first identifies the set of traces $\Sigma_w \subseteq \Sigma_L$ that started during that window (according to the timestamp of the trace's first event)[2]. The algorithm then counts how often relation $b$ is observed in $\Sigma_w$, e.g., how often we observe that an activity $x$ is directly followed by activity $y$ in these traces, and divides that count by the number of traces in $\Sigma_w$, yielding $\mathcal{B}[b, w]$.

Figure 13 illustrates the steps for constructing a behavioral matrix in a simplified example. In this example, an event log containing three change points is first converted into absolute frequencies for each

[2]The choice to compute a behavioral matrix according to the traces that start during $w$ follows existing work on concept drift detection (cf. [3]) and is based on the assumption that the process version of a trace is fixed when it starts. If this assumption does not hold, the behavioral matrix $\mathcal{B}$ should, instead, be computed according to the events observed during $w$.

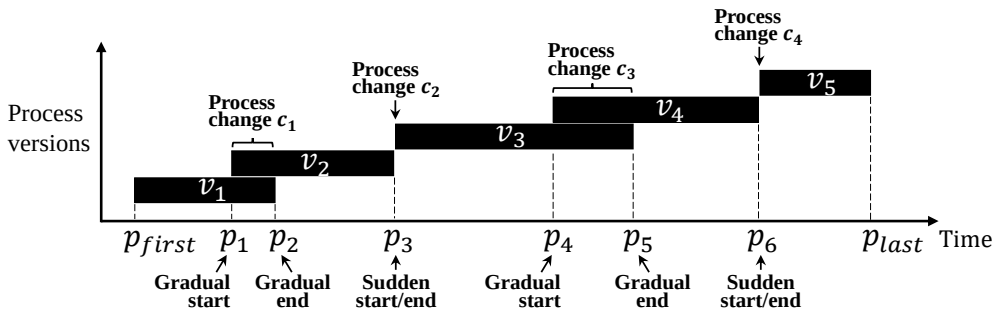



Figure 12: Outcome of the second framework step: each change point $p_i$ is associated with a process change.

**Algorithm 1** Change Type Classification

**Input**: Event log $L$, sequence of change points $P = \langle p_1, \ldots, p_N \rangle$

**Parameter**: Trend percentile $\alpha$

**Output**: Sequence of process changes $\overline{C} = \langle c_1, \ldots, c_N \rangle$

**procedure** ChangeTypeClassification($L, P, \alpha$)
  $W = \langle w_0, \ldots, w_{|P|} \rangle \leftarrow$ `computeWindows`$(L, P)$ ▷ Define window sequence based on $P$
  $T \leftarrow \langle t_1, \ldots, t_{|P|} \rangle$ with $t_i = \perp$ for all $i$ in $[1, |P|]$ ▷ Initialize a change point classification sequence
  $\mathcal{B} \leftarrow$ `getBehavioralMatrix`$(L, P)$ ▷ Derive behavioral matrix
  **for** $i \in [1, |P| - 1]$ **do** ▷ Iterate over all but the last change point in $P$
    **if** $t_i = \perp$ **then**
      $weigth_s \leftarrow 0, weigth_g \leftarrow 0$ ▷ Initialize the weights for sudden/gradual classification
      **for** $b \in \mathcal{B}.relations$ **do** ▷ Iterate over behavioral relations
        $roc_{b,1} \leftarrow$ `rateOfChange`$(b, w_{i-1}, w_i)$ ▷ Compute rate of change from $w_{i-1}$ to $w_i$
        $roc_{b,2} \leftarrow$ `rateOfChange`$(b, w_i, w_{i+1})$ ▷ Compute rate of change from $w_i$ to $w_{i+1}$
        $trend_b \leftarrow$ `classifyTrend`$(roc_{b,1}, roc_{b,2}, \alpha)$ ▷ Classify as sudden, gradual, or unchanged
        **if** $trend_b =$ "sudden" **then**
          $weigth_s \leftarrow weigth_s +$ `calcWeight`$(b, [w_{i-1}, w_i, w_{i+1}])$ ▷ Accumulate weights
        **else if** $trend_b =$ "gradual" **then**
          $weigth_g \leftarrow weigth_g +$ `calcWeight`$(b, [w_{i-1}, w_i, w_{i+1}])$ ▷ Accumulate weights
      **if** $weigth_g > weigth_s$ **then**
        $t_i \leftarrow$ "gradual$_{start}$", $t_{i+1} \leftarrow$ "gradual$_{end}$" ▷ Assign gradual start/end change types
      **else**
        $t_i \leftarrow$ "sudden" ▷ Assign sudden change type
  **if** $t_{|P|} = \perp$ **then** ▷ Handle edge cases where the final point is not assigned yet
    $t_{|P|} \leftarrow$ "sudden" ▷ Assign sudden change type
  $\overline{C} = \langle c_1, \ldots, c_N \rangle \leftarrow$ `setProcessChanges`$(P, T, \mathcal{B})$ ▷ Define process changes
  **return** $\overline{C}$

time window, assuming four distinct relations across all recorded traces. These absolute frequencies are then transformed into relative frequencies, producing the final behavioral matrix.

**Change point classification.** Using the obtained behavioral representation, our algorithm next iteratively goes over the change points in $P$ to classify them. For each index $i \in [1, |P| - 1]$, the algorithm considers a situation such as previously illustrated in Figure 11. Specifically, it considers the behavior observed during window $w_i$, in light of the behavior observed for its preceding ($w_{i-1}$) and succeeding ($w_{i+1}$) windows, in

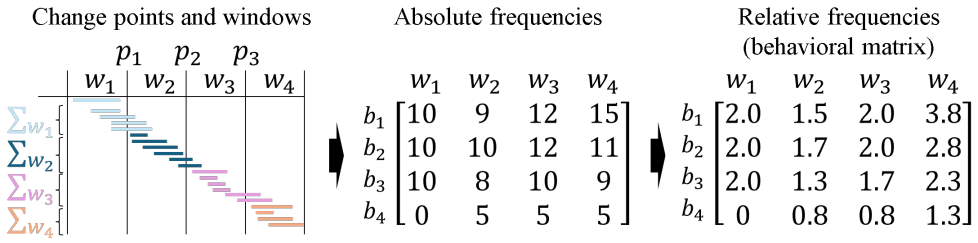



Figure 13: Example of a behavioral matrix obtained from an event log.

order to determine if a change point $p_i$ corresponds to a *sudden* change or to a *gradual start*, i.e., the first change point in a gradual process change from $p_i$ to $p_{i+1}$.

To decide between these two options, the algorithm first considers each behavioral relation surrounding point $p_i$ individually before classifying $p_i$ as being a *sudden* or *gradual start* point:

*Relation-level classification.* Given a change point $p_i$, our algorithm determines for each behavioral relation $b \in \mathcal{B}.relations$ if it was involved in the changes of points $p_i$ and $p_{i+1}$, and, if so, if $b$ changed in a sudden or a gradual manner.

To achieve this, we calculate the *rates of change* (*roc*) [25] of relation $b$ when moving from time window $w_{i-1}$ to $w_i$, denoted as $roc_{b,1}$, and from $w_i$ to $w_{i+1}$, denoted as $roc_{b,2}$ (lines 9–10). For this, we use the following function:

$$\texttt{rateOfChange}(b, w_{i-1}, w_i) := \begin{cases} (\mathcal{B}[b, w_i]/\mathcal{B}[b, w_{i-1}] - 1) * 100 & \text{if } \mathcal{B}[b, w_{i-1}] > 0, \\ 100 & \text{if } \mathcal{B}[b, w_{i-1}] = 0 \text{ and } \mathcal{B}[b, w_i] > 0, \\ 0 & \text{if } \mathcal{B}[b, w_{i-1}] = 0 \text{ and } \mathcal{B}[b, w_i] = 0. \end{cases} \quad (1)$$

In Equation 1, the first case captures the usual computation of a rate of change, according to established definitions [25]. The second and third cases avoid division by zero errors, setting the change rate of a relation $b$ to 100% if it appears in $w_i$ but not in $w_{i-1}$ (second case) and to 0% if it appears in neither $w_i$ or $w_{i-1}$ (third case).

Next to these change rates, we also consider a *trend percentile* $\alpha$, which we use to determine if a change rate falls within the normal variance in a process or if it is part of an actual process change. Specifically, we consider the rates of change of all behavioral relations across all pairs of successive windows, i.e., the distribution of rates of change for a particular log. Given this distribution, we consider *roc* to be significant (i.e., a true process change) if it is lower than the value of the bottom percentile $V_\alpha$ (a significant decrease in the frequency of a relation) or greater than the value of the top percentile $V_{1-\alpha}$ (a significant increase) of all rates of change, otherwise, i.e., if $roc \in [V_\alpha, V_{1-\alpha}]$, it is considered to be part of the normal behavioral variation.

Given the change rates $roc_{b,1}$ and $roc_{b,2}$, and the trend percentile $\alpha$, we then classify the trend of relation $b$ for this particular change point using the following function (line 11):

$$\texttt{classifyTrend}(roc_{b,1}, roc_{b,2}, \alpha) := \begin{cases} \textit{unchanged} & \text{if } roc_{b,1} \in [V_\alpha, V_{1-\alpha}] \text{ and } roc_{b,2} \in [V_\alpha, V_{1-\alpha}], \\ \textit{gradual change} & \text{if } \texttt{sign}(roc_{b,1}) = \texttt{sign}(roc_{b,2}) \text{ and} \\ & roc_{b,1} \notin [V_\alpha, V_{1-\alpha}] \vee roc_{b,2} \notin [V_\alpha, V_{1-\alpha}], \\ \textit{sudden change} & \text{otherwise.} \end{cases} \tag{2}$$

This function classifies the relation $b$ as *unchanged* when both $roc_{b,1}$ and $roc_{b,2}$ represent normal behavioral variation. If $roc_{b,1}$ and $roc_{b,2}$ indicate shifts in the same direction (both positive or both negative) with at least one reflecting a significant increase or decrease, then $b$ is classified as a gradual change. Finally, the change is classified as *sudden* in all other cases.

*Change point-level classification.* Next, the algorithm classifies the change point as either *sudden* or as *gradual start* by considering the relevance of the identified trend types per relation. Specifically, given a relation $b$ and a change point $p_i$, the algorithm assigns a *weight* to that relation. This weight is determined as the average of the relative frequencies of $b$ in windows $w_{i-1}$, $w_i$, and $w_{i+1}$ divided by the sum of all averaged relative frequencies of other relations. Using these weights, our algorithm categorizes the change point as *gradual start* if the cumulative weight of identified relations exceeds that of *sudden*; otherwise, it is labeled as a *sudden* (lines 12–19).

Note that if a point $p_i$ is classified as a *gradual start* point, its successor, $p_{i+1}$ is then immediately classified as a corresponding *gradual end* (see the illustration in Figure 11). Furthermore, if the last change point has not been assigned a type yet when completing the iteration, which happens when $|P| = 1$ or when the type of the second last change point is *sudden*, then the last change point is classified as *sudden* (line 21).

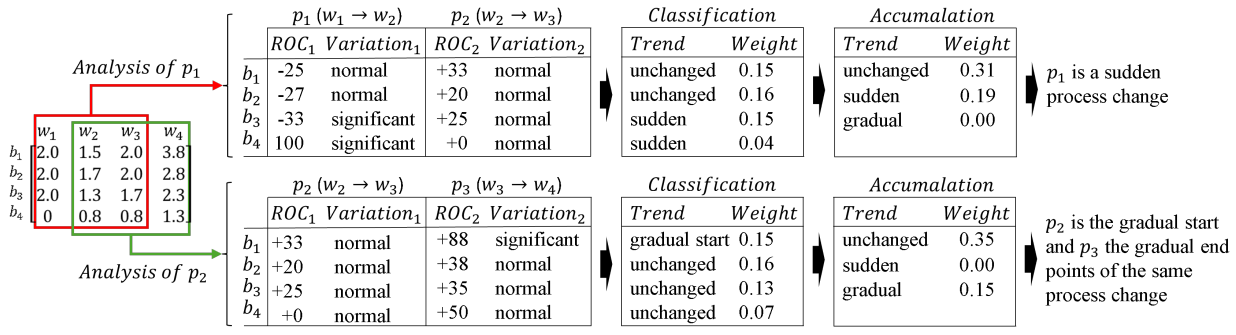



Figure 14: Example of change point-level classification based on the behavior matrix.

Figure 14 illustrate the change point-level classification using the behavior matrix depicted in Figure 13. We consider the first three columns of the behavior matrix to decide whether $c_1$ is a sudden process change

or is the start point of a gradual process change. For this, we calculate the corresponding $roc_{b,1}$ and $roc_{b,2}$ for each relation $b$. Assuming a trend percentile of $\alpha = 3$, we get upper and lower percentiles for $roc$ of 77 and -31, respectively. Given these percentiles, the variations of $roc_{b,1}$ and $roc_{b,2}$ are classified into "normal" or "significant" and then for each relation a trend is defined according to Equation 2. Finally, the weights are accumulated according to the trends. Since the total number of relation weights that belong to a sudden change (0.19) is greater than for gradual (0.0), the change point $c_1$ is classified as sudden. Following the same procedure, the change point $c_2$ is classified as a gradual start, therefore, the change point $c_3$ is a gradual end point. Overall, the three change points form two process changes: sudden ($c_1$) and gradual ($c_2$ and $c_3$).Finally, the algorithm establishes a sequence of detected process changes $\overline{C}$ based on the change points, their classifications, and the behavioral matrix (line 22). Specifically, if a change point $p_i$ is classified as a sudden change, then the algorithm generates a process change instance $c := (sudden, p_i, p_i, v^{previous}, v^{next})$, where $v^{previous}$ and $v^{next}$ corresponding to the behavioral representation recorded in $\mathcal{B}$ for the time windows preceding and following $p_i$, respectively. Otherwise, if two consecutive change points $p_i$ and $p_{i+1}$ are classified as *gradual start* and *gradual end*, then the algorithm creates a process change instance $c := (gradual, p_i, p_{i+1}, v^{previous}, v^{next})$. Here, $v^{previous}$ and $v^{next}$ correspond to the behavioral representation recorded in $\mathcal{B}$ for the time windows $w_{i-1}$ and $w_{i+1}$, respectively.

### *3.4. Step 3: Change Inter-Relation Analysis*

In the final step, our framework analyzes connections among the detected process changes to recognize a collection of simple and complex drifts as output. We illustrate this output in Figure 15, which shows that the four process changes of our running example form three concept drifts. Specifically, the gradual process change $c_1$ does not connect to other changes and is therefore classified as a simple drift. By contrast, changes $c_2$ and $c_3$ have been recognized to jointly form an incremental drift. Finally, since change $c_4$ leads to a previously observed process version ($v_1$), this represents a recurring drift in the process.

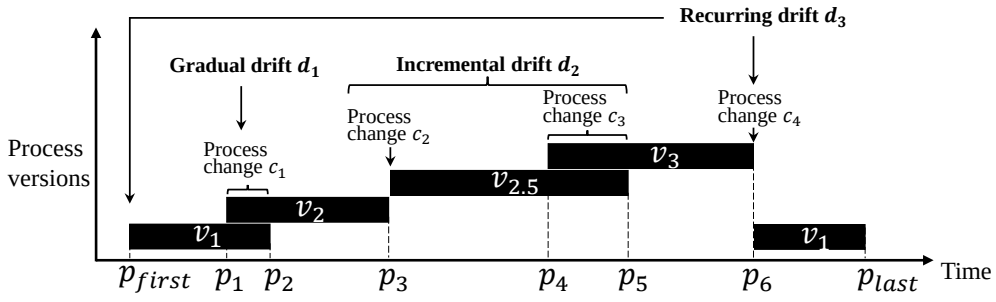



Figure 15: Outcome of the final framework step: every process change is either linked with other changes, creating a complex drift, or exists independently as a simple drift.

To operationalize this step, we have developed a change inter-relation detection algorithm outlined in Algorithm 2. The algorithm takes the sequence of changes $\overline{C}$ stemming from the previous steps as input

**Algorithm 2** Change Inter-Relation Detection

**Input**: Sequence of process changes $\overline{C} = \langle c_1, \ldots, c_M \rangle$

**Parameter**: Thresholds for recurring and incremental behavioral similarity $\theta_{rec}$ and $\theta_{inc}$

**Output**: Collection of detected drifts $D$

**procedure** ChangeInterRelationDetection($\overline{C}, \theta_{rec}, \theta_{inc}$)

▷ **Recurring drift detection**

$\mathcal{S}^{rec} \leftarrow \{\}$ ▷ Initialize a collection to store detected sets of recurring process changes

$c_0 := (sudden, p_{first}, p_{first}, \bot, c_1.v^{previous})$ ▷ Define dummy change for the beginning of log

**for** $c_i$ in $\langle c_0, c_1, \ldots, c_M \rangle$ **do** ▷ For all process changes...

  **if** $c_i \notin \bigcup_{s \in \mathcal{S}^{rec}} s$ **then** ▷ ... that are not already assigned

    $s^{rec} \leftarrow \{c_i\}$ ▷ Initialize a set for changes similar to $c_i$

    **for** $c_j$ in $\langle c_{i+2}, \ldots, c_M \rangle$ **do** ▷ For indirect successors of $c_i$...

      **if** $c_{j-1} \notin s^{rec} \wedge$ `isRecurringChange`$(c_j, \theta_{rec})$ **then**

        $s^{rec}$.`add`$(c_j)$ ▷ Extend $s^{rec}$ with $c_j$

    **if** $|s^{rec}| > 1$ **then** ▷ Check if $s^{rec}$ has more than one process change

      $\mathcal{S}^{rec}$.`add`$(s^{rec})$ ▷ Add $s^{rec}$ to the collection of recurring sets

$D^{rec} \leftarrow$ `detectRecurringPatterns`$(\mathcal{S}^{rec})$ ▷ Turn recurring sets into recurring drifts

▷ **Incremental drift detection**

$\mathcal{S}^{inc} \leftarrow \{\}, s^{inc} \leftarrow \langle\rangle$ ▷ Initialize a collection to store detected incremental process changes

**for** $c_i$ in $\overline{C}$ **do** ▷ For all process changes...

  **if** $c_i \notin \bigcup_{s \in \mathcal{S}^{rec}} s \wedge$ `isMinorChange`$(c_i, \theta_{inc})$ **then** ▷ Check if $c_i$ is not in $\mathcal{S}^{rec}$ and is a minor change

    $s^{inc}$.`extend`$(c_i)$ ▷ Add $c_i$ to the sequence of incremental process changes

  **else**

    **if** $|s^{inc}| > 1$ **then** ▷ Check if $s^{inc}$ has more than one process change

      $\mathcal{S}^{inc}$.`add`$(s^{inc})$ ▷ Add $s^{inc}$ to $\mathcal{S}^{inc}$

    $s^{inc} \leftarrow \{\}$ ▷ Reset $s^{inc}$

$D^{inc} \leftarrow$ `detectIncrementalPatterns`$(\mathcal{S}^{inc})$ ▷ Turn incremental sequence into incremental drifts

$D^{simple} \leftarrow$ `detectSimpleDrifts`$(\overline{C}, D^{rec}, D^{inc})$ ▷ Turn stand-alone changes into simple drifts

$D := D^{simple} \cup D^{rec} \cup D^{inc}$ ▷ Define drift collection

**return** $D$

and evaluates relationships between the changes by comparing behavioral similarities of associated process versions. It produces a collection of concept drifts $D$ as output, where process changes are connected, forming complex drifts, or independent, representing simple drifts. The main part of our algorithm focuses

on detecting complex drifts since simple drifts automatically remain after larger, complex drifts have been detected. To identify complex drifts, our algorithm begins by searching for recurring drifts and then turns to incremental ones. The search for recurring drifts takes priority over incremental ones because it relies on a stronger condition regarding the similarity between different process versions.

Next, we describe Algorithm 2 following its key components: recurring drift detection, incremental drift detection, and simple drift detection.**Recurring drift detection.** Algorithm 2 detects recurring process changes by looking for process changes that lead to highly similar process behavior.

To do this, our algorithm starts by initializing a collection $\mathcal{S}^{rec}$ to store sets of process changes that lead to instances of the same process version (line 2). Since also the initial process version in $w_0$ can reoccur (see e.g., Figure 15), we establish a dummy change $c_0$, corresponding to a sudden change that appears at the start of the event log, i.e., at point $p_{first}$ (line 3). Afterwards, the algorithm iterates over all process changes, including $c_0$ (line 4). For any change $c_i$ that is not yet part of a recurring drift, the algorithm checks if there is any indirect successor $c_j$ that leads to process behavior highly similar to the version following $c_i$, while ensuring that $c_{j-1}$ is not part of a recurring drift. (line 8). If that is the case, it is recognized that $c_j$ is recurring with respect to $c_i$.

Here, the function `isRecurringChange` quantifies the similarity between the two process versions $c_i.v^{next}$ and $c_j.v^{next}$. If the similarity is above a threshold $\theta_{rec}$, then the process change $c_j$ is considered to be a recurring change with respect to $c_i$. The function `isRecurringChange` can be operationalized using any measure that quantifies the similarity between two frequency distributions (over behavioral relations, i.e., the behavioral representation of two windows or process versions), such as the cosine similarity, a vector-based similarity measure, or the Earth mover's distance, which quantifies the distance between distributions, both of which are commonly applied in process mining settings [8, 26]. As a default option, we use cosine similarity, which demonstrates higher sensitivity in detecting added and removed behavioral relationships and achieved the best overall drift detection results in our evaluation.Recurring changes are collected in a previously instantiated set $s^{rec}$ (line 9). Note that our algorithm ensures that $s^{rec}$ will never contain consecutive process changes, as recurring process version instances correspond to process changes that are separated by at least one other process change. Any set $s^{rec}$ that contains more than one process change, i.e., any set that actually forms a recurring drift, is added to the set of recurring drifts $\mathcal{S}^{rec}$ (lines 10–11).

Finally, the set $\mathcal{S}^{rec}$ is turned into a set of recurring drifts $D^{rec}$ (line 12). To do this, our algorithm looks for sequences of consecutive changes in the sets of $S^{rec}$ that repeat two or more process versions in a particular order. Examples of this are seen in Figure 6, where case a) shows versions $v_1$ and $v_2$ in an alternating pattern, whereas case b) shows a repetition of a larger sequence $\langle v_1, v_2, v_3 \rangle$. In both cases, the sets that form these larger patterns are combined into a single recurring drift, whereas any set in $\mathcal{S}^{rec}$ that does not form a larger pattern is turned into a recurring drift by itself, as, e.g., seen for the $\{c_0, c_4\}$

in Figure 15.

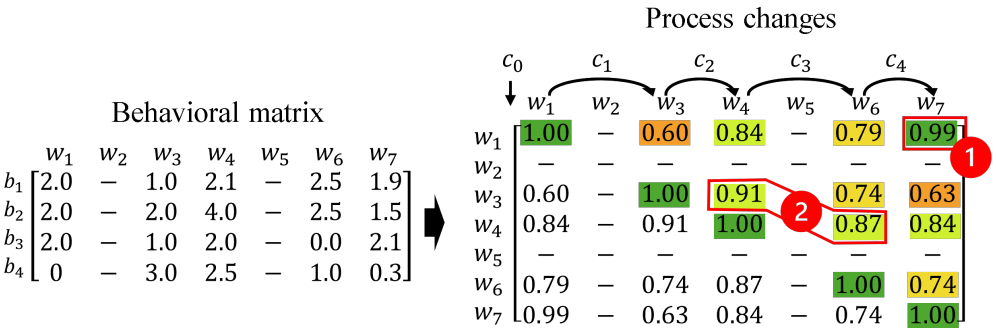



Figure 16: Example of change inter-relation detection using a behavioral matrix.

To illustrate the idea of the recurring drift detection component of our change inter-relation detection algorithm, we present an example shown in Figure 16. This example includes a behavior matrix corresponding to the scenario in Figure 15, along with the matrix displaying the cosine similarity between behavioral representations of different process versions across various windows[3]. Since the process changes $c_1$ and $c_3$ are of a gradual type, the windows $w_2$ and $w_5$ represent transition periods between two different process versions and, therefore, are excluded from consideration. From this similarity matrix, we can observe how the behavior of process versions before and after each change point compares to one another. Assuming $\theta_{rec} = 0.95$, the algorithm detects that the similarity between the process version in $w_1$ and the process version in $w_7$ (i.e., after $c_4$) is 0.99, which exceeds the threshold of $\theta_{rec}$ (see Point 1 in the figure). As a result, the process change $c_4$, along with $c_0$, is identified as a recurring drift.

**Incremental drift detection.** After recurring change detection, we start with the detection of incremental changes by identifying sequences of minor consecutive process changes in the sequence $\overline{C}$.

To do this, we start by initializing a collection $\mathcal{S}^{inc}$ to store sequences of consecutive incremental process changes (line 13). For each process change $c_i$ in $\overline{C}$ that is not already part of detected recurring drifts, the algorithm checks if $c_i$ is a minor change (lines 14–15). The underlying function, `isMinorChange` considers a change to be minor if the behavioral similarity between $c_i.v^{previous}$ and $c_i.v^{next}$ is above the threshold $\theta_{inc}$ (using the same similarity measure as used for recurring drifts). If $c_i$ is indeed a minor change, we add it to the current sequence $s^{inc}$ (line 16) and continue with the following process change. If $c_i$ is not minor (or was already part of a recurring change), we add the sequence of minor changes observed so far, $s^{inc}$, to the set of incremental sequences, provided that $s^{inc}$ contains more than one change (lines 18–19). Then, after resetting $s^{rec}$ (line 20), we continue with the next process change.

[3]Our algorithm performs similarity analysis by iterating over a sequence of detected process changes. For clarity, however, we consider the matrix as a whole to provide a more straightforward explanation.

To illustrate the idea of the incremental drift detection component, we again use the example in Figure 16. Since process changes $c_0$ and $c_4$ are identified as changes of a recurring drift, we need to only consider the remaining process changes and look for a sequence of at least two process changes with a minor change severity. Assuming $\theta_{inc} = 0.85$, a minor process change is given if the similarity between the process version before and after the process change is above $\theta_{inc}$. In our example, among the remaining process changes $c_1$, $c_2$, and $c_3$, both $c_2$ and $c_3$ exhibit similarity values above the incremental threshold of 0.85 (see Point 2 in the figure). Therefore, these changes constitute an incremental drift.

In the end, the algorithm converts the set $\mathcal{S}^{inc}$ into a collection of incremental drifts $D^{rec}$ (line 21), where each detected sequence of consecutive minor process changes within $\mathcal{S}^{inc}$ is turned into an incremental drift.

**Simple drift establishment.** After identifying complex drifts, the algorithm establishes a set of simple drifts (line 22) by turning any process change in $\overline{C}$ that is not part of a recurring or incremental drift into a stand-alone, simple drift, resulting in a set $D^{simple}$. In our example, shown in Figure 16, the only remaining process change is the gradual process change $c_2$. Since it is not part of a complex drift, it is classified as a simple drift.

**Framework output.** Finally, the algorithm returns the collection of detected drifts $D$ given by the union of the detected simple, recurring, and incremental drifts (line 23).

Our framework thus identifies both simple and complex drifts in accordance with the taxonomy presented in Figure 7, specifying their drift types along with the associated process changes, distinguishing between sudden and gradual process changes. In this way, our framework supports the comprehensive detection and characterization of concept drifts from event logs.

## 4. Evaluation

This section describes the evaluation experiments we conducted to test the ability of our framework to detect different types of concept drifts recorded in event logs. Section 4.1 describes the data collection used for this purpose and Section 4.2 presents the general evaluation setup. Afterwards, Sections 4.3–4.5 describe the quantitative experimental results per framework step, whereas Section 4.6 presents a qualitative comparison of our work to a state-of-the-art technique. To ensure reproducibility, we have made the data collection, implementation, configurations, and raw results accessible in our public repository[4].

### *4.1. Data Collection*

To evaluate our work, we require a collection of event logs that contain known (i.e., gold-standard) concept drifts of all types. Since a publicly available collection does not exist, we generated synthetic

[4]Project repository: https://gitlab.uni-mannheim.de/processanalytics/concept-drift-characterization.

datasets using CDLG [27], a flexible tool that produces event logs with known concept drifts. We used CDLG to generate a dataset of 100 event logs. Each log is derived from a randomly generated process tree using PTandLogGenerator [28], which is then modified by a sequence of one to three randomly generated drifts, of different types.

| **PTandLogGenerator Parameter** | | **Process Tree Complexity** | | |
|---|---|---|---|---|
| **Type** | **Name** | **Simple** | **Middle** | **Complex** |
| Number of activities | Minimum | 6 | 14 | 20 |
| | Mode | 9 | 18 | 25 |
| | Maximum | 12 | 20 | 30 |
| Control-flow probabilities | Sequence | 0.70 | 0.25 | 0.25 |
| | Choice | 0.10 | 0.30 | 0.30 |
| | Parallel | 0.15 | 0.25 | 0.25 |
| | Loop | 0.05 | 0.20 | 0.20 |

Table 3: Key settings in PTandLogGenerator [28] used to produce process trees of various complexity levels.

C2

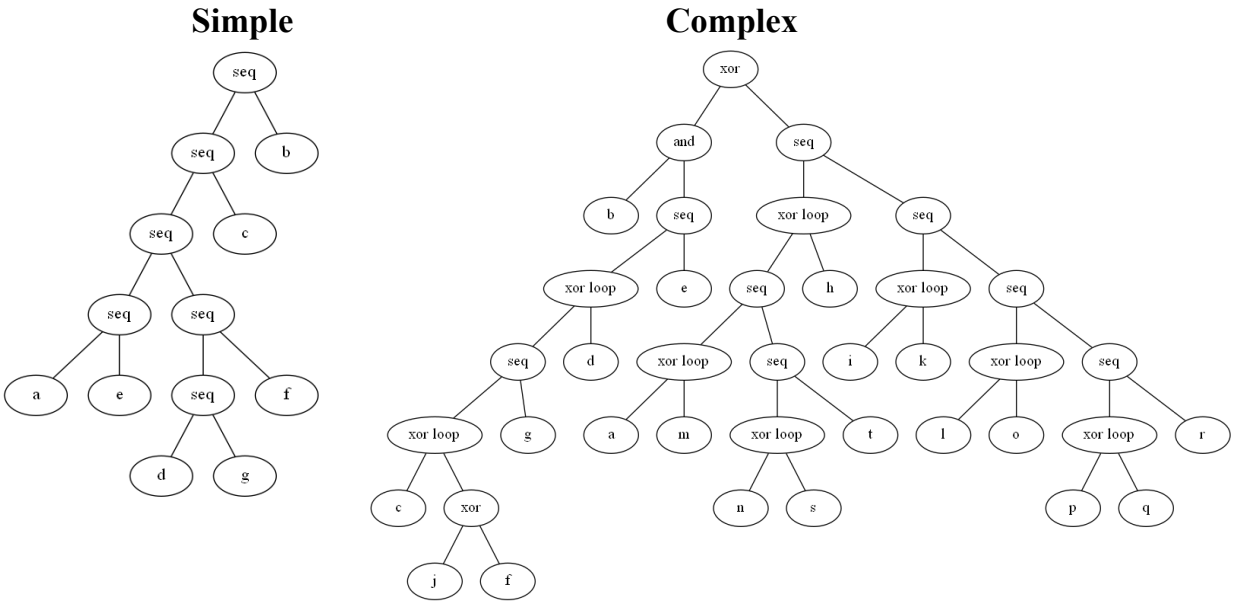



Figure 17: Examples of initial process trees of different complexity levels.

Table 4 provides an overview of the base dataset that contains 100 event logs. The logs contain a total of 198 drifts, i.e., 38 logs with 1 drift, 26 logs with 2 drifts, and 36 logs with 3 drifts. Simple sudden and

gradual drifts involve a single process change, whereas complex recurring and incremental drifts comprise three process changes, resulting in a total of 426 process changes. Each process change can occur suddenly, resulting in a single change point, or gradually, resulting in two change points (gradual start and end), leading to 651 change points in total.

Table 4: Characteristics of the drifts in our base dataset consisting of 100 event logs.

| **Drift type** | **Number of drifts** | **Number of process changes** | **Change points** | | | |
|---|---|---|---|---|---|---|
| | | | **Total** | **Sudden** | **Gradual start** | **Gradual end** |
| Sudden | 35 | 35 | 35 | 35 | — | — |
| Gradual | 49 | 49 | 98 | — | 49 | 49 |
| Incremental | 58 | 174 | 263 | 85 | 89 | 89 |
| Recurring | 56 | 168 | 255 | 81 | 87 | 87 |
| **Total** | **198** | **426** | **651** | **201** | **225** | **225** |

To evaluate the robustness of our detection framework, we generated two variations of the base dataset by introducing noise into the event logs Specifically, we use an existing noise-insertion technique [29] that randomly inserts, removes, and swaps events in a fraction of the traces in an event log, obtaining datasets with 20% and 40% noisy traces (with all other characteristics, such as the number of drifts and change points, the same as depicted in Table 4). Consequently, the data collection used in our evaluation consists of 600 event logs.

### *4.2. General Setup*

**Framework implementation.** We used Python 3 to implement a prototype of our framework, which is publicly available through the aforementioned project repository. Our implementation uses the *PM4Py* [30] library to handle event logs and the python library *Pandas*[5] for data processing steps.

**Framework configurations.** Step 1 of our framework can be instantiated using any of a range of existing change point detection techniques. Therefore, in Section 4.3, we evaluate existing techniques on our data collection and select the one with the best accuracy for the remaining experiments. In Step 2, to build a behavioral matrix, we rely on directly-follows relations over the control flow perspective to obtain behavioral representations of the process execution for each window. In particular, we extract all directly-follows relations from each trace and allocate them to the window containing the first event in the trace. Finally, in Step 3, we use cosine similarity to compare the behavioral similarity between windows in the behavioral matrix when performing incremental and recurring change analyses.

[5]Available at https://pandas.pydata.org

In our algorithms, we set the hyperparameters as follows: the trend percentile is $\alpha = 3$, the incremental threshold is $\theta_{inc} = 0.80$, and the recurring threshold is $\theta_{rec} = 0.95$. We determined these hyperparameters by testing a range of options: $\alpha \in [1, 2, 3, 4, 5, 10]$, $\theta_{inc} \in [0.5, 0.1, \ldots, 0.4]$, and $\theta_{rec} \in [0.50, 0.55, \ldots, 0.95]$ and selecting the combination that yielded the best average score, as reported in Experiment 3.2 (see Section 4.5) using an additional data collection for fine-tuning. This data collection was obtained using the same method as the evaluation dataset and exhibits similar concept drifts and noise levels.

**Evaluation measures.** To evaluate our framework's accuracy, we compare the detected change points, drifts, and drift types against those in the gold standard using precision, recall, and F1-score. Below, we provide a general representation of these measures since their specific operationalization differs per experiment.

*Precision* measures how many detected change points or drifts are correct based on their alignment with the gold standard. It is given as the ratio of true positive detection over the total number of true positive and false positive detections:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{3}$$

*Recall* measures the fraction of gold standard changes or drifts that are detected by our framework. It is given as the ratio of true positive detections over the total number of true positive and false negative detections:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}. \tag{4}$$

Finally, the *F1-score* is the harmonic mean of precision and recall:

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{5}$$

Depending on the experimental setup, we use additional measures that aggregate the results, such as the weighted F1-score, which considers all drift types or change points and their respective support (number of instances).

### *4.3. Step 1: Change Point Detection*

This section discusses experiments conducted to test the performance of various existing techniques that can be used to instantiate Step 1 of our framework, which detects change points in a given event log.

#### *4.3.1. Experimental Setup*

The first step of our framework relies on existing solutions for change point detection. For this reason, we evaluate existing change point detection techniques on our data collection and select the one with the best results for the remaining experiments.

**Change point detection techniques.** We test the same seven change point detection techniques used in a recent experimental comparison by Adams et al. [3]:

1. ProcessGraphs: Seeliger et al. [7] derives process graph features from an event log with the corresponding edge and node frequencies, using a sliding window to extract features. Change points are recognized if p-values from a statistical test fall below a threshold.
2. EMD: Brockhoff et al. [8] use sliding windows where each trace becomes a local multi-activity feature. Earth Mover's Distance measures distribution differences between different windows. Change points are identified as local maxima in a graph.
3. Adwin/J: Martjushev et al. [15] address window size limitations in Bose et al. [4]. They introduce an adaptive window size, using recursive hypothesis tests on smaller windows near low p-values for precise drift location. High p-values lead to adaptive window growth, enabling segment skipping.
4. Rinv: Zheng et al. [9] propose drift detection using a boolean relation matrix. Matrix entries show direct and eventual activity relations within cases. Drifts are identified by change point candidates. These are clustered using DBSCAN, with final change points found based on minimal centroid distance.
5. Lcdd: Lin et al. [10] detect drifts by monitoring changes in directly-follows relations. They use two windows: a static complete window and a sliding detection window. The complete window size is determined based on local directly-follows completeness, ensuring all relations of the active process are included. A drift is reported when the two windows diverge concerning a directly-follows relation.
6. Bose/J: Bose et al. [4] propose activity pair-based features. Using a fixed sliding window, features are extracted locally or globally. Their importance is assessed using J-measure, which quantifies the goodness of a rule. Then, populations are compared using non-parametric tests, and drifts are identified based on the resulting p-values.
7. ProDrift: Maaradji et al. [16] convert traces to partial orders of activities (called "runs"). Runs emerge from ordered traces, handling concurrent activities. Using sliding windows, run populations are extracted and then compared using statistical tests.

To identify the optimal parameter configuration for each technique, we employ the experimental framework [3]. This framework tests these techniques with different parameter options and reports various evaluation measures. Based on the best F1-score, we select the best parameter configuration for each technique, depicted in Table 5.

**Evaluation measures.** We use precision, recall, and F1-score introduced in Equation 3 to Equation 5. In this experiment, a true positive is recorded if a detected change point is correctly assigned to a gold-standard change point. To achieve this optimal assignment, we solve a linear program proposed by Adams et al. [3]. This program finds the best match between detected and actual change points (both are given via trace ID, i.e., the ordinal number of a trace in the log following the change), minimizing the total distance in terms of the number of traces between them. In contrast to Adams et al., who use a fixed absolute deviation of 200 traces, we use a *relative accepted deviation* between the detected and the actual change point. This

Table 5: Optimal parameter configurations for each change point detection technique.

| Technique | Parameters |
|---|---|
| ProcessGraphs | Window size: 300, max. window size: 400, p-value:0.1 |
| EMD | Window size: 150, step size: 1 |
| Adwin/J | Min/max adaptive window: 200/700, p-value: 0.4, step size: 20 |
| Rinv | Minimum relation invariance distance: 600, epsilon: 180 |
| Lcdd | Window size complete/detection: 400/400, stable period: 5 |
| Bose/J | Window size: 150, step size: 2 |
| ProDrift | Window size: 400, step size: 2 |

relative acceptance deviation ensures a fair accuracy analysis for our data collection containing event logs with significant differences in the number of traces. C1 For all assignments, we identify a true positive when the distance between the detected change point and the gold-standard change point is less than or equal to 5% of the total number of traces in the log. Otherwise, the detected change points are classified as a false positive detection. The sum of false negatives and true positives (the denominator in the recall calculation) equals the total number of gold-standard change points.

*4.3.2. Results*

We present the evaluation results of the change point detection techniques, highlighting overall performance and the impact of noise, various change patterns, and different levels of change severity.

**Overall performance.** Table 6 provides an overview of the evaluation results. In terms of overall performance, we can distinguish three clear groups. The first group consists of two techniques demonstrating top performance: ProcessGraphs and EMD. ProcessGraphs is the best performer, with an average F1-score of approximately 0.70 across all datasets, while EMD also shows good performance but achieves a slightly lower average F1-score of 0.67. In the second group, we find Adwin/J, Rinv, and Lcdd, attaining average F1-scores between 0.53 and 0.58. Lastly, the third group encompasses Bose/J and ProDrift, with average F1-scores of 0.31–0.33.

**Noise impact.** Regarding robustness to noise, ProcessGraphs, EMD, Adwin/J, and Bose/J maintain stable evaluation measures across noise levels, while Rinv, ProDrift, and Lcdd achieve lower accuracy for the noisy logs. Specifically, Rinv's F1-score drops by close to 50% between the dataset without noise and those with 20% noisy traces (from 0.78 to 0.41). However, additional noise does not noticeably impact its performance further. By contrast, ProDrift experiences a substantial performance decline, with an 80% drop in F1-score when 20% noise is introduced (from 0.71 to 0.16), followed by an additional 70% decrease for the logs with 40% noise (from 0.16 to 0.05). The significant drop in performance is primarily due to

Table 6: Performance of the change point detection techniques across logs with different noise levels. Bold numbers indicate the best score for the particular column.

| Technique | Logs w/o noise | | | Logs w. 20% noise | | | Logs w. 40% noise | | | Avg. F1 |
|---|---|---|---|---|---|---|---|---|---|---|
| | Prc. | Rec. | F1 | Prc. | Rec. | F1 | Prc. | Rec. | F1 | |
| ProcessGraphs | 0.63 | 0.74 | 0.68 | 0.66 | 0.76 | **0.71** | 0.68 | 0.72 | **0.70** | **0.70** |
| EMD | 0.71 | 0.67 | 0.69 | 0.67 | 0.65 | 0.66 | 0.68 | 0.66 | 0.67 | 0.67 |
| Adwin/J | 0.84 | 0.45 | 0.59 | 0.84 | 0.44 | 0.58 | 0.85 | 0.43 | 0.57 | 0.58 |
| Rinv | 0.72 | **0.87** | **0.78** | 0.57 | 0.32 | 0.41 | 0.49 | 0.37 | 0.42 | 0.54 |
| Lcdd | 0.57 | 0.63 | 0.60 | 0.34 | **0.95** | 0.50 | 0.35 | **0.88** | 0.50 | 0.53 |
| Bose/J | 0.52 | 0.23 | 0.32 | 0.64 | 0.23 | 0.34 | 0.66 | 0.22 | 0.33 | 0.33 |
| ProDrift | **0.99** | 0.55 | 0.71 | **0.89** | 0.09 | 0.16 | **1.00** | 0.02 | 0.05 | 0.31 |

a decline in recall when noise is introduced, decreasing to 0.09 at 20% noise and further to 0.02 at 40% noise. However, precision remains consistently high, exceeding 0.90. Lcdd also appears highly sensitive to noise, often detecting non-existent changes in its results. Its precision is consistently low, while recall increases significantly for the noisy logs, reaching up to 0.95. Interestingly, both ProcessGraphs and Bose/J exhibit an increase in precision as noise levels rise.

Table 7: Performance of the change point detection techniques across different change patterns.

| Technique | Overall | | Recall (by change patterns) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | ⊕ | ⊖ | ⇔ | ⊕&⊖ | ⊕&⇔ | ⊖&⇔ | ⊕&⊖&⇔ |
| ProcessGraphs | 0.65 | 0.74 | 0.68 | **0.73** | 0.59 | **0.90** | **0.88** | 0.82 | **0.97** |
| EMD | 0.69 | 0.66 | 0.66 | 0.69 | 0.64 | 0.57 | 0.66 | 0.74 | 0.75 |
| Adwin/J | 0.84 | 0.44 | 0.54 | 0.51 | 0.34 | 0.42 | 0.25 | 0.17 | 0.28 |
| Rinv | 0.61 | 0.52 | 0.56 | 0.61 | 0.46 | 0.52 | 0.33 | 0.39 | 0.30 |
| Lcdd | 0.38 | **0.82** | **0.86** | **0.73** | **0.80** | 0.86 | 0.87 | **0.91** | 0.94 |
| Bose/J | 0.60 | 0.23 | 0.27 | 0.32 | 0.14 | 0.20 | 0.06 | 0.00 | 0.12 |
| ProDrift | **0.98** | 0.22 | 0.21 | 0.19 | 0.19 | 0.25 | 0.30 | 0.30 | 0.32 |
| Support | | | 555 | 537 | 309 | 279 | 138 | 66 | 69 |

*Legend: "⊕"- Insertion, "⊖" - Deletion, "⇔" - Relocation.*

**Change pattern impact.** In addition to the noise impact, we assess the accuracy of change point detection techniques across different change patterns. We consider 6 different change patterns that are derived from three basic process changes and any combination of them: insertion of new activities (⊕), deletion of existing

activities ($\ominus$), and relocation of activities ($\Leftrightarrow$). Based on all correctly detected change points, we compute recall for each change pattern. As false positives do not correspond to any specific change pattern, we report precision as an overall measure per technique.

Table 7 shows the evaluation results for different change patterns, revealing two main findings. First, detection capabilities vary across techniques for different change patterns. For example, LCDD achieves the highest overall recall of 0.82, showing the best performance for basic process changes. However, PROCESSGRAPHS demonstrates superior change point detection accuracy for complex patterns, outperforming LCDD's recall in 3 out of 4 cases. Second, the complexity of the change pattern impacts detection accuracy differently among techniques. For some techniques, accuracy improves with complex patterns that involve combinations of two or three basic process changes, while for others, accuracy declines as complexity increases. For instance, PROCESSGRAPHS, EMD, LCDD, and PRODRIFT show improved performance with complex patterns, with recall increasing by up to 30 percentage points (as observed for PROCESSGRAPHS) compared to their average recall on simpler patterns with only one change type. Conversely, other techniques experience a drop in recall, indicating challenges in detecting more complex process changes.

Table 8: Performance of the change point detection techniques across different change severity levels.

| | **Overall** | | **Recall (by change severity, in %)** | | | | |
|---|---|---|---|---|---|---|---|
| **Technique** | **Precision** | **Recall** | $(0, 20]$ | $(20, 30]$ | $(30, 40]$ | $(40, 50]$ | $(50, 100]$ |
| PROCESSGRAPHS | 0.65 | 0.74 | 0.71 | 0.70 | 0.75 | 0.78 | **0.81** |
| EMD | 0.69 | 0.66 | 0.66 | 0.64 | 0.71 | 0.68 | 0.62 |
| ADWIN/J | 0.84 | 0.44 | 0.39 | 0.41 | 0.53 | 0.42 | 0.49 |
| RINV | 0.61 | 0.52 | 0.49 | 0.46 | 0.52 | 0.52 | 0.65 |
| LCDD | 0.38 | **0.82** | **0.81** | **0.83** | **0.78** | **0.89** | 0.78 |
| BOSE/J | 0.60 | 0.23 | 0.25 | 0.16 | 0.21 | 0.26 | 0.29 |
| PRODRIFT | **0.98** | 0.22 | 0.20 | 0.25 | 0.24 | 0.20 | 0.22 |
| Support | | | 468 | 495 | 348 | 312 | 330 |

**Change severity impact.** To complement the analysis of change pattern impact, we assess the effect of change severity on detection accuracy. To do this, we assign each change point a severity level, defined as the percentage of behavioral alteration following a process change. This percentage is calculated by comparing the gold-standard process trees before and after a process change. Specifically, we generate all possible traces (setting loop sizes to one) and derive directly-follow relationships. These sets of relationships are then compared using the Jaccard coefficient, a similarity measure between finite sets defined as the ratio of the intersection size to the union size of the sample sets [31]. Finally, we group the resulting values into five intervals with comparable support. Table 8 presents the obtained evaluation results across different levels

of change severity. Regarding peak performance, LCCD achieves the highest recall for change severities up to 50%. However, for process changes that lead to extreme behavioral shifts above 50%, PROCESSGRAPHS surpasses LCCD in performance. When analyzing the impact of change severity on accuracy across techniques, we observe that both PROCESSGRAPHS and RINV demonstrate improved accuracy as change severity increases, while the recall for other techniques remains relatively stable.
Given these evaluation results, we adopt the PROCESSGRAPHS technique to instantiate Step 1 of our framework in the remaining experiments (where applicable).

### *4.4. Step 2: Change Type Classification*

This section discusses experiments conducted to test the performance of Step 2 of our framework, which aims to detect if the detected change points belong to sudden or gradual process changes.

#### *4.4.1. Experimental Setup*

**Experiments.** To comprehensively assess the performance of our framework when it comes to change type classification, we conduct two experiments: In *Experiment 2.1*, we assess the performance of Step 2 in isolation, which tests how well our change type detection technique works when provided with the gold-standard change points as input. Afterward, in *Experiment 2.2*, we assess the combined performance of Steps 1 and 2, i.e., using the change points detected by the PROCESSGRAPHS technique as input for Step 2, which tests how well our framework can recognize sudden and gradual changes in general.

**Evaluation measures.** We use the following measures in the two described experiments:

Experiment 2.1 represents a classical classification problem, where each change point from the gold standard is classified as either *sudden*, *gradual start*, or *gradual end*. A true positive is thus recorded if the detected change point type is correctly assigned to its gold-standard type.

In Experiment 2.2, a true positive is recorded under two conditions: (1) the detected change point is correctly assigned to an actual change point, with a deviation of less than 5% of the total traces in the event log (same as for the evaluation of Step 1), and (2) the detected change point type (sudden, gradual start, or gradual end) corresponds to the gold standard. Otherwise, it is a false positive.

**Baseline.** To put the performance of Step 2 of our framework into perspective, we compare its accuracy against the PRODRIFT technique proposed by Maaradji et al. [16]. We select this technique as a baseline for two reasons: First, it stands out as the only technique that focuses on the automated detection of both sudden and gradual drifts, without requiring a manual indication of the drift type to be searched (e.g., in contrast to BOSE/J [4]). Second, this technique uses a similar two-step procedure that first detects change points and then aims to detect gradual changes by considering sequences of three consecutive windows (see Figure 11). The technique is conceptually different, though, since their second step relies on a statistical

test on distributions of partially ordered runs within sliding windows of traces, while our framework considers individual behavioral patterns.

**Baseline implementation.** To be able to use ProDrift as a baseline for both experiments, we need to decouple its two steps as well, allowing us to provide its second step with the gold-standard change points as input. Since its available Java implementation[6] does not support this, we implemented the second step in Python, following the procedure given in the paper [16, Definition 4]. Instead of using the Java-based JOptimitzer as the solver for the non-linear program, we use the *Optimize* module from SciPy[7] (version 1.10.0), using BFGS as the selected optimization method with the initialization (3, 3).

### *4.4.2. Results*

**Experiment 2.1.** Table 9 presents the change type classification results when using the gold-standard change points as input for Step 2. In this table, we compare the accuracy of our second framework step (see Algorithm 1) against the baseline across datasets with varying noise levels. For each dataset, we report the obtained evaluation measures, including overall measures weighted by the support for each type.Our framework's algorithm greatly outperforms the ProDrift baseline, showing higher F1-scores for all change types, achieving a weighted F1-score of 0.79 for the logs without noise, versus 0.35 of the baseline. Our algorithm maintains a balanced precision-recall ratio across different change types. In contrast, the baseline struggles with identifying gradual change start and end instances, often mistaking them for sudden changes. This is evident in the baseline's high recall (0.86) and low precision (0.36) for sudden changes, along with relatively higher precision compared to recall for gradual start and end change types.

The difference between our algorithm and the baseline is even more pronounced when considering the datasets with noise. The performance of our algorithm even slightly improves for these datasets (from 0.79 to 0.85 and 0.80), whereas the baseline's performance further drops (from 0.35 to 0.18 and 0.27). A particular issue for the baseline is the detection of gradual changes in noisy settings, as, e.g., shown by the recall scores of 0.03 and 0.04 for the logs with 20% noise.

**Experiment 2.2.** Table 10 presents the change type classification results obtained using the change points detected in Step 1. Similar to Table 9, this table provides evaluation measures for different change point types across datasets with varying levels of noise. However, we compare the joint accuracy of our framework's Steps 1 and 2 against both versions of the baseline.First of all, compared to the results of Experiment 2.1, the evaluation results indicate a notable drop in performance for both techniques: For the logs without noise, our framework's weighted F1-score drops from 0.77 to 0.44, whereas the baseline's performance drops from 0.35 to 0.24. This is because the accuracy of change type classification (Step 2) depends on the accuracy

[6] Available at http://apromore.org/platform/tools
[7] Available at https://docs.scipy.org/doc/scipy/

Table 9: Results of Experiment 2.1: Change type classification using gold-standard change points as input.

| Technique | Type | Supp. | Logs w/o noise | | | Logs w. 20% noise | | | Logs w. 40% noise | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Prc. | Rec. | F1 | Prc. | Rec. | F1 | Prc. | Rec. | F1 |
| Framework Step 2 | Sudden | 201 | 0.63 | 0.85 | 0.72 | 0.76 | 0.79 | 0.77 | 0.75 | 0.73 | 0.74 |
| | Gradual start | 225 | 0.88 | 0.75 | 0.81 | 0.88 | 0.87 | 0.87 | 0.82 | 0.83 | 0.83 |
| | Gradual end | 225 | 0.89 | 0.76 | 0.82 | 0.89 | 0.88 | 0.88 | 0.83 | 0.84 | 0.83 |
| | **Overall (weighted)** | | 0.81 | 0.78 | 0.79 | 0.85 | 0.84 | 0.85 | 0.80 | 0.80 | 0.80 |
| ProDrift Step 2 (baseline) | Sudden | 201 | 0.36 | 0.86 | 0.50 | 0.31 | 0.92 | 0.46 | 0.31 | 0.85 | 0.46 |
| | Gradual start | 225 | 0.55 | 0.20 | 0.29 | 0.30 | 0.03 | 0.06 | 0.51 | 0.12 | 0.19 |
| | Gradual end | 225 | 0.49 | 0.18 | 0.26 | 0.35 | 0.04 | 0.06 | 0.53 | 0.12 | 0.20 |
| | **Overall (weighted)** | | 0.47 | 0.40 | 0.35 | 0.32 | 0.31 | 0.18 | 0.46 | 0.34 | 0.27 |

Table 10: Results of Experiment 2.2: Change type classification using change points from Step 1 as input.

| Technique | Type | Supp. | Logs w/o noise | | | Logs w. 20% noise | | | Logs w. 40% noise | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Prc. | Rec. | F1 | Prc. | Rec. | F1 | Prc. | Rec. | F1 |
| Framework Steps 1–2 | Sudden | 201 | 0.27 | 0.41 | 0.32 | 0.29 | 0.42 | 0.35 | 0.31 | 0.39 | 0.35 |
| | Gradual start | 225 | 0.50 | 0.51 | 0.50 | 0.46 | 0.48 | 0.47 | 0.44 | 0.43 | 0.44 |
| | Gradual end | 225 | 0.49 | 0.48 | 0.49 | 0.46 | 0.46 | 0.46 | 0.44 | 0.43 | 0.43 |
| | **Overall (weighted)** | | 0.42 | 0.47 | 0.44 | 0.41 | 0.45 | 0.43 | 0.40 | 0.42 | 0.40 |
| ProDrift Python-based (baseline) | Sudden | 201 | 0.45 | 0.60 | 0.51 | 0.56 | 0.21 | 0.30 | 0.52 | 0.05 | 0.10 |
| | Gradual start | 225 | 0.57 | 0.13 | 0.22 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | Gradual end | 225 | 0.02 | 0.00 | 0.01 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | **Overall (weighted)** | | 0.34 | 0.23 | 0.24 | 0.17 | 0.07 | 0.09 | 0.16 | 0.02 | 0.03 |
| ProDrift Java-based (baseline) | Sudden | 201 | 0.41 | 0.66 | 0.50 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | Gradual start | 225 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | Gradual end | 225 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | **Overall (weighted)** | | 0.13 | 0.20 | 0.16 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

of change point detection (Step 1). If a change point is not detected, it cannot be classified accurately, and if a change point is incorrectly detected, it also leads to classification errors. Therefore, if the quality of available change point detection techniques improves, the quality of our framework's subsequent steps will follow.

Despite the performance drop, it is important to note that our framework consistently outperforms the baseline in precision, recall, and F1-score across all change types. Furthermore, the results remain robust in the presence of noise. Notably, the Python-based implementation of the baseline struggles to identify gradual drifts when noise is present. To verify that this is not due to an implementation error, we provide evaluation results for the baseline using the existing Java-based tool. Using run-based configurations with adaptive window size, the obtained results for sudden process changes align with our Python-based implementation (including the change point detection accuracy of Step 1 corresponding to the results in Table 6). However, the Java-based implementation still struggles to detect gradual process changes, which is consistent with our Python-based results.

### *4.5. Step 3: Change Inter-Relation Analysis*

This section discusses experiments conducted to test the performance of Step 3 of our framework, which goal is to detect concept drifts and determine their drift types from identified process changes.

#### *4.5.1. Experimental Setup*

**Experiments.** To demonstrate the accuracy of the framework's step, we conduct two experiments similar to the experiments in Section 4.4. In *Experiment 3.1*, we assume 100% accuracy of the first and second framework steps to measure the unbiased accuracy of Step 3. Then, in *Experiment 3.2*, we assume 100% accuracy only of the first framework step (since this is based on existing work) to evaluate the joint accuracy of Steps 2 and 3, thus evaluating how well our framework can detect simple and complex drifts based on detected change points.

**Evaluation measures.** For both experiments, we consider evaluation measures at two levels. First, we assess drift type detection accuracy at the *change-point level*, for which we check if change points are assigned to the right drift type (i.e., sudden, gradual, incremental, or recurring). Second, we assess accuracy at the *drift level*, for which we check if change points have been grouped together into drifts (of the right type).

*Change-point level.* To assess drift type detection accuracy at a change-point level, we consider a multi-class classification problem, where each change point is classified into one of four drift types: sudden, gradual, incremental, and recurring. We record a true positive if the detected drift type of a change point is correct given the gold standard; otherwise, it is a false positive for the detected type and a false negative for the gold-standard type. Note that we here report on weighted precision, recall, and F1-score, to account for imbalances in the dataset.

*Drift level.* Assessing drift type detection accuracy at a drift level is more complex than at a change-point level, since, in this case, it involves the (possible) assignment of multiple change points to a single drift, which must also be of the right type. Therefore, we consider a drift to be correctly detected (i.e., a *true positive*) if it has the right drift type and contains the right set of associated change points. If the drift type is correct, but the set of detected change points differs, we use the *Jaccard similarity* to quantify to what degree the set of change points is correct. The Jaccard similarity is calculated by dividing the number of change points in both the actual and detected drifts by the number of observations in either set. For example, if a detected incremental drift $d$ includes three change points $\{c_1, c_2, c_3\}$, while the gold standard specifies four change points $\{c_1, c_2, c_3, c_4\}$, $d$ is considered to be a 0.75 true positive. A drift is considered a false positive if its type is incorrect or if the Jaccard similarity is 0. The denominator in recall, representing the sum of true positive and false negative detections for each drift type, is determined by the total number of drifts that type in the gold standard.

Note that this assessment requires us to establish an alignment between the sets of detected and actual (gold-standard) drifts, i.e., determining which gold-standard drift corresponds to a detected drift, if any. This task is equivalent to solving the *two-dimensional rectangular assignment problem* [32], a well-known problem in operations research. The goal of this assignment is to efficiently distribute a pool of resources (such as individuals or employees) among a limited set of tasks, with the aim of minimizing the total associated cost matrix. In our case, the detected drifts serve as resources, and the actual drifts represent the tasks. The cost matrix is derived from the Jaccard similarity (with a negative sign) between pairs of drifts. We solve this problem using the Jonker-Volgenant algorithm [33], implemented in the Python library *scipy.optimize*.

#### *4.5.2. Results*

**Experiment 3.1.** Table 11 presents the evaluation results of Experiment 3.1 for both measurement levels.

The results at the change-point level show that our framework performs well, achieving an average weighted F1-score of about 0.84 across all datasets. The results also show our framework's robustness to noise, maintaining consistent performance across all noise levels.

With respect to drift types, there are some drift-specific findings. We achieve good accuracy for gradual drifts with precision and recall consistently remains at approximately 0.78 and 0.86, respectively. The detection of recurring drifts yields a perfect recall of 0.98 and a precision close to 0.90. Notably, sudden drift detection displays relatively lower precision, but remains consistently high in recall. In contrast, detecting incremental drifts reveals an opposing trend, indicating that incremental changes are occasionally mistaken for sudden ones. This is closely related to the problem of differentiating a sequence of simple drifts from an incremental drift, as illustrated in Figure 5. Following our definition of a concept drift in Section 2.4.1, our algorithm considers the two necessary conditions for detecting incremental drifts: (1) a sequence of at

Table 11: Results of Experiment 3.1: Drift type detection accuracy using gold-standard change points and types (Steps 1-2).

| Drift type | Support | Logs w/o noise | | | Logs w. 20% noise | | | Logs w. 40% noise | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **Prc.** | **Rec.** | **F1** | **Prc.** | **Rec.** | **F1** | **Prc.** | **Rec.** | **F1** |
| **Change-point level** | | | | | | | | | | |
| Sudden | 35 | 0.46 | 0.91 | 0.62 | 0.46 | 0.89 | 0.61 | 0.46 | 0.89 | 0.60 |
| Gradual | 98 | 0.78 | 0.86 | 0.82 | 0.78 | 0.86 | 0.82 | 0.78 | 0.86 | 0.82 |
| Incremental | 263 | 0.93 | 0.69 | 0.79 | 0.93 | 0.68 | 0.79 | 0.93 | 0.67 | 0.78 |
| Recurring | 255 | 0.90 | 0.98 | 0.94 | 0.89 | 0.98 | 0.93 | 0.88 | 0.98 | 0.93 |
| **Overall (weighted)** | | 0.87 | 0.84 | 0.85 | 0.87 | 0.84 | 0.84 | 0.86 | 0.83 | 0.83 |
| **Drift level** | | | | | | | | | | |
| Sudden | 35 | 0.46 | 0.91 | 0.62 | 0.46 | 0.89 | 0.61 | 0.46 | 0.89 | 0.60 |
| Gradual | 49 | 0.78 | 0.86 | 0.82 | 0.78 | 0.86 | 0.82 | 0.78 | 0.86 | 0.82 |
| Incremental | 58 | 0.93 | 0.66 | 0.77 | 0.93 | 0.65 | 0.76 | 0.92 | 0.63 | 0.75 |
| Recurring | 56 | 0.83 | 0.79 | 0.81 | 0.82 | 0.81 | 0.82 | 0.81 | 0.81 | 0.81 |
| **Overall (weighted)** | | 0.78 | 0.79 | 0.77 | 0.78 | 0.79 | 0.77 | 0.77 | 0.78 | 0.76 |

least two consecutive process changes with (2) low change severity. Improving detection accuracy can be achieved by introducing additional criteria to assess whether or not process changes are part of the same transformation initiative, indicating they belong to the same incremental drifts.

At the drift level, the overall detection accuracy drops by about 8%pt. (from 0.84 to 0.76) compared to the change level. This decline relates to complex drifts (recurring and incremental) since the accuracy for simple drifts remains the same. This aligns with expectations, as complex drifts involve multiple process changes, making them more prone to misidentification at the drift level.

The decline in detection accuracy in recurring drifts is the primary factor contributing to the overall accuracy drop. The average recall over all datasets drops by about 18%pt. (from 0.98 to 0.80). Given the high recall at the change level, this suggests that recurring change points are occasionally assigned to the wrong recurring drifts. Consequently, precision also drops by about 7%pt. Another contributing factor is the drop of around 3%pt. in recall for incremental drifts. This can be attributed to the detection challenge between simple and incremental drifts.

**Experiment 3.2.** Table 12 presents the evaluation results of Experiment 3.2 for both measurement levels.

The results on the change-point level show solid performance with an average weighted F1-score of about 0.79 across all datasets. The results remain robust to noise, ensuring a stable performance across all noise levels.

Compared to the previous experiment, the overall accuracy drops by about 5%pt (from 0.84 to 0.79),

Table 12: Results of Experiment 3.2: Drift type detection accuracy using gold-standard change points (Step 1) as input.

| Drift type | Support | Logs w/o noise | | | Logs w. 20% noise | | | Logs w. 40% noise | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **Prc.** | **Rec.** | **F1** | **Prc.** | **Rec.** | **F1** | **Prc.** | **Rec.** | **F1** |
| **Change-point level** | | | | | | | | | | |
| Sudden | 35 | 0.35 | 0.77 | 0.48 | 0.44 | 0.77 | 0.56 | 0.45 | 0.71 | 0.55 |
| Gradual | 98 | 0.68 | 0.66 | 0.67 | 0.64 | 0.78 | 0.70 | 0.69 | 0.76 | 0.72 |
| Incremental | 263 | 0.87 | 0.61 | 0.72 | 0.90 | 0.64 | 0.75 | 0.89 | 0.69 | 0.78 |
| Recurring | 255 | 0.84 | 0.97 | 0.90 | 0.87 | 0.96 | 0.91 | 0.88 | 0.98 | 0.92 |
| **Overall (weighted)** | | 0.80 | 0.77 | 0.77 | 0.82 | 0.79 | 0.80 | 0.83 | 0.81 | 0.81 |
| **Drift level** | | | | | | | | | | |
| Sudden | 35 | 0.35 | 0.77 | 0.48 | 0.44 | 0.77 | 0.56 | 0.45 | 0.71 | 0.55 |
| Gradual | 49 | 0.70 | 0.66 | 0.68 | 0.64 | 0.78 | 0.70 | 0.70 | 0.76 | 0.73 |
| Incremental | 58 | 0.85 | 0.57 | 0.68 | 0.88 | 0.60 | 0.71 | 0.88 | 0.61 | 0.72 |
| Recurring | 56 | 0.75 | 0.75 | 0.75 | 0.78 | 0.77 | 0.77 | 0.81 | 0.78 | 0.80 |
| **Overall (weighted)** | | 0.70 | 0.68 | 0.66 | 0.71 | 0.72 | 0.70 | 0.74 | 0.71 | 0.72 |

caused by errors from the change type classification algorithm (Step 2). The primary factor contributing to the decline in performance is the decrease in accuracy in simple drift detection. Specifically, we observe an average F1-score drop of about 12%pt. for gradual and about 8%pt. for sudden drifts across all datasets. This is anticipated, as simple drift detection relies on the identified change type from Step 2. If there is an error in identifying the change type, it follows that the corresponding drift type will be inaccurately determined. The errors in Step 2 also impact the accuracy of complex drift detection; however, the drop for recurring and incremental drift types is below 5%pt.

At the drift level, the average F1-score drops by 10%pt. compared to the change-level evaluation. The main drivers and their contributions are proportional to what we observe in Experiment 3.1.

Overall, our algorithms proposed for Steps 2 and 3 demonstrate their effectiveness at detecting drift types, with clearly evident noise resilience, since they maintain consistent results across different noise levels. In addition, our evaluation at the drift level highlights the challenges in accurately detecting and distinguishing complex drifts.

### *4.6. Steps 1-3: Comparison with the Baseline*

In this section, we apply all three steps of our framework and highlight the advantages of the obtained results compared to the state-of-the-art solution.

#### *4.6.1. Experimental setup*

**Baseline and configurations.** We use the Visual Drift Detection (VDD) technique [17] as the state-of-the-art technique. It can detect simple and complex drift types from an event log and is the only (partially) comparable solution (see Section 2.5), although the approach is not automated. In our experiments, we use the online version of the VDD technique[8] with the suggested default parameters: window size 300, slide size 150, cut threshold 300.

**Experiment.** The VDD technique is not fully automated, requiring a user's interpretation of visualizations for complex drift detection. This makes automated comparisons infeasible. Therefore, we illustrate our framework's advantages by comparing results for a specific event log from our data collection (i.e., log number 90, with 20% noise). The selected log contains 64,594 events, 9,169 traces, 1,237 trace variants, and 8 distinct activities. The left-hand side of Table 13 shows that the log contains 8 change points that jointly form 5 changes and 3 drifts. Specifically, there are individual sudden and gradual drifts and a larger incremental drift, which encompasses 3 changes and 5 change points.

#### *4.6.2. Results*

**Our framework.** In Table 13, the right-hand side displays the outcomes of our framework. The first framework step accurately identifies 7 out of 8 change points, with one false positive (identifying a non-existent change point) and one false negative (not recognizing an actual change point), resulting in a F1-score of 0.88. The change type detection step identifies 1 out of 2 sudden change types, with one false negative detection, and successfully identifies 2 out of 3 gradual start/end changes, with also one false positive detection. This leads to a weighted F1-score of 0.67. The errors in the classification of change types appear primarily due to the errors in the previous step. In the final detection step, all change points are correctly assigned to their corresponding drift types, except for the first two, which together form a simple gradual drift, leading to the weighted F1-score of 0.58 (evaluated at the change level).

**Baseline.** Figure 18 presents a primary outcome of the baseline technique, i.e., the Drift Map, Drift Charts, and autocorrelation plots with further measures that are used to support visual analysis. The Drift Map (Figure 18a) shows over 525 detected behavioral rules (y-axis), organized into 55 behavioral clusters (indicated by horizontal dashed white lines). Within each cluster, white vertical lines indicate change points, while black vertical dashed lines highlight global change points spanning over all clusters. The Drift Charts (Figure 18b depicts a drift chart for a selected cluster) categorize drifts, helping to determine if drifts exist, their patterns over time, and the stability or drift in behavior. Finally, Figure 18c depicts autocorrelation, to detect recurring drifts, and erratic measure to differentiate, for instance, gradual drift from incremental.

Next, we use VDD to try to obtain the same types of insights provided by our framework's three steps:

[8] Available online through the URL https://yesanton.github.io/Process-Drift-Visualization-With-Declare/client/build/

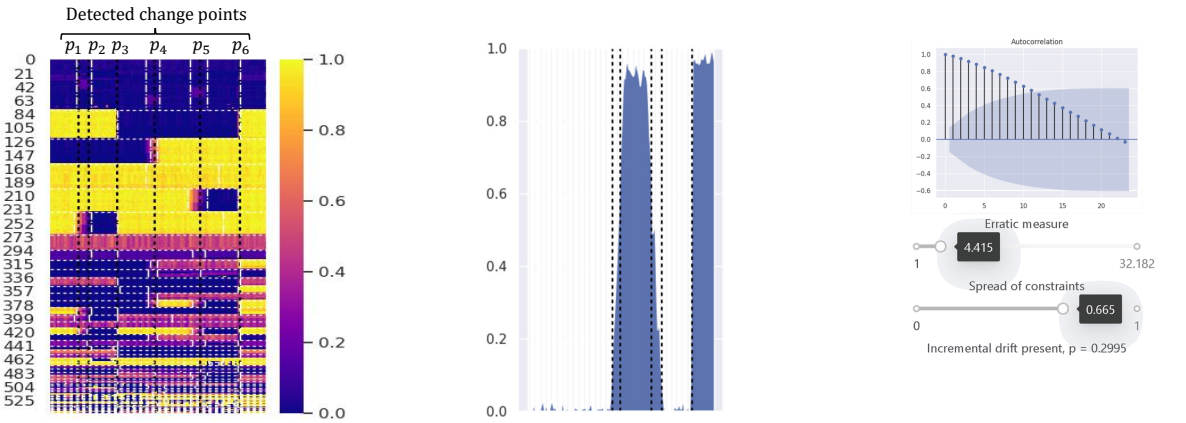
Table 13: Actual concept drift information for the event log 90 with 20% noise vs. detected drift information using our framework.

| **Gold standard** | | | **Detected drift information (our framework)** | | |
|---|---|---|---|---|---|
| **Change point*** | **Change type** | **Drift type** | **Change point*** | **change type** | **Drift type** |
| 1042 | gradual start | gradual | (+) 1429 | (+) gradual start | (–) incremental |
| 1748 | gradual end | gradual | (+) 1879 | (+) gradual end | (–) incremental |
| | | | (–) 2593 | (–) gradual start | (–) incremental |
| 2855 | sudden | incremental | (+) 2989 | (–) gradual end | (+) incremental |
| 4089 | gradual start | incremental | (+) 4195 | (–) sudden | (+) incremental |
| 4623 | gradual end | incremental | | | |
| 5847 | gradual start | incremental | (+) 5976 | (+) gradual start | (+) incremental |
| 6623 | gradual end | incremental | (+) 6694 | (+) gradual end | (+) incremental |
| 7991 | sudden | sudden | (+) 7947 | (+) sudden | (+) sudden |

* We use trace ID to indicate the change point in a log.

*Step 1: Change point detection.* The baseline identifies a total of 6 change points. The change points $p_1$, $p_2$, $p_3$, and $p_6$ align with the gold standard, while the remaining $p_4$ and $p_5$ are within the gradual transition phases. Unfortunately, the figure does not allow for a precise assessment of whether $p_4$ and $p_5$ adhere to the accepted deviation of 5% from the log size. Consequently, our estimation of the accuracy in the change point detection step yields a precision range of 0.67 to 1.00 and a recall range of 0.50 to 0.75, depending on whether or not the change points $p_4$ and $p_5$ are considered correct. This results in an F1-score spanning from 0.57 to 0.86 with an average value of 0.71. Even in the best-case scenario (F1-score of 0.86), the performance is below the accuracy of our framework's Step 1.

*Step 2: Change type classification.* Regarding process change detection, the baseline does not automatically distinguish between gradual and sudden process changes. However, by visualizing different behavioral clusters, we can identify when detected change points suggest a gradual process change. For example, multiple clusters between the first and second change points indicate a gradual shift in certain behavioral patterns. Thus, we can conclude that the baseline correctly identified the first gradual process change, the second sudden process change, and the final sudden process changes. However, change points $p_4$ and $p_5$ are misclassified as sudden process changes due to an error in the change point detection step. Overall, VDD thus leads to a perfect recall and precision of 0.50 for sudden process changes, with perfect precision and 0.33 recall for gradual process changes, resulting in a total weighted F1-score of 0.54. This result is 13%pt. below the accuracy of our framework's Step 2, which, furthermore, does not rely on manual interpretation of various visualizations.




(a) Drift Map. (b) Drift chart (cluster 13). (c) Autocorrelation (cluster 13).

Figure 18: The output of the VDD technique.

*Step 3: Change inter-relation detection.* The identification of drifts and their drift types using VDD is highly challenging. The tool's visualization of clusters does not differentiate between affected behavioral rules that relate to different drifts. Therefore, based on the Drift Map and Drift Charts, it is not feasible to understand the big picture, i.e., that the first two change points as well as the last one are simple drifts that do not belong to the incremental drift in between. Moreover, some visualized clusters indicate a recurring drift pattern, though such patterns are not present in the gold standard. Therefore, drawing conclusions about the overall drift scenario, in terms of exact simple and complex drifts, remains speculative.

**Overall insights.** In summary, the VDD technique visualizes and locates change points well, but understanding the different types of process changes and their relationships, and determining if they form a single incremental drift or a set of unrelated drifts, is still difficult. Consequently, achieving a complete understanding of the overall concept drift is not feasible, even with manual effort. In contrast, our proposed framework enables an automated detection of drifts and achieves better results for all three steps compared to the baselines.herefore, our framework provides an important step towards automated and comprehensive detection of concept drifts from event logs.

## 5. Conclusion

In this paper, we contribute to the more comprehensive characterization of concept drifts recorded in event logs by introducing an improved drift type characterization taxonomy and presenting a three-step framework for the automated detection of drift types.

Our improved drift type classification taxonomy classifies drifts into simple and complex ones, relying on process changes and their properties as the core elements of concept drifts. Our taxonomy addresses existing inconsistencies by providing an exclusive drift type classification. Our taxonomy enhances the

detection and evaluation of concept drifts, especially complex drifts with inter-connected process changes. Following our taxonomy, we proposed an automated, three-step framework for the comprehensive detection and characterization of concept drifts. Our experiments demonstrated that our change type and change inter-relation detection algorithms used in Steps 2 and 3 provide accurate results, offering a more comprehensive understanding of drift types and the overall evolution of the process compared to state-of-the-art solutions.

In future work, we plan to enhance our framework in several directions. First, we aim to refine the change point detection step, which significantly influences the overall framework accuracy. Our evaluation of existing change point detection techniques revealed a need for a more precise approach to detect change points that correspond to sudden, gradual start and end points, especially, in the presence of noise. Second, we plan to improve the identification of incremental drifts by considering additional aspects, like change localization and time aspects, that can improve detection accuracy. Finally, we see the potential to expand this framework with drift localization information that, based on identified drifts and process versions, goes beyond localization for individual change points.

*Reproducibility: The implementation, experimental details, and obtained raw results are available through the repository linked in Section 4.*

## References


[1] M. Dumas, M. La Rosa, J. Mendling, H. A. Reijers, et al., Fundamentals of business process management, Vol. 2, Springer, 2018. doi:10.1007/978-3-662-56509-4.

[2] W. van der Aalst, Process Mining: Data Science in Action, Springer, 2016. doi:10.1007/978-3-662-49851-4_1.

[3] J. N. Adams, C. Pitsch, T. Brockhoff, W. van der Aalst, An experimental evaluation of process concept drift detection, Proceedings of the VLDB Endowment 16 (8). doi:10.14778/3594512.3594517.

[4] R. J. C. Bose, W. Van Der Aalst, I. Žliobaitė, M. Pechenizkiy, Dealing with concept drifts in process mining, Transactions on Neural Networks and Learning Systems 25 (1) (2013) 154–171. doi:10.1109/TNNLS.2013.2278313.

[5] G. Elkhawaga, M. Abuelkheir, S. I. Barakat, A. M. Riad, M. Reichert, CONDA-PM: a systematic review and framework for concept drift analysis in process mining, Algorithms 13 (7) (2020) 161. doi:10.3390/a13070161.

[6] D. M. V. Sato, S. C. De Freitas, J. P. Barddal, E. E. Scalabrin, A survey on concept drift in process mining, ACM Computing Surveys 54 (9) (2021) 1–38. doi:10.1145/3472752.

[7] A. Seeliger, T. Nolle, M. Mühlhäuser, Detecting concept drift in processes using graph metrics on process graphs, in: Proceedings of the 9th Conference on Subject-Oriented Business Process Management, Vol. 9, ACM, 2017, pp. 1–10. doi:10.1145/3040565.3040566.

[8] T. Brockhoff, M. S. Uysal, W. van der Aalst, Time-aware concept drift detection using the earth mover's distance, in: Proceedigns of the 2nd International Conference on Process Mining, IEEE, 2020, pp. 33–40. doi:10.1109/ICPM49681.2020.00016.

[9] C. Zheng, L. Wen, J. Wang, Detecting process concept drifts from event logs, in: OTM Confederated International Conferences "On the Move to Meaningful Internet Systems", Springer, 2017, pp. 524–542. doi:10.1007/978-3-319-69462-7_33.

[10] L. Lin, L. Wen, L. Lin, J. Pei, H. Yang, LCDD: detecting business process drifts based on local completeness, IEEE Transactions on Services Computing 15 (4) (2020) 2086–2099. doi:10.1109/TSC.2020.3032787.

[11] H. Nguyen, M. Dumas, M. La Rosa, A. H. ter Hofstede, Multi-perspective comparison of business process variants based on event logs, in: Conceptual Modeling: 37th International Conference, ER 2018, Xi'an, China, October 22–25, 2018, Proceedings 37, Springer, pp. 449–459. doi:10.1007/978-3-030-00847-5_32.

[12] B. Hompes, J. C. Buijs, W. van der Aalst, P. M. Dixit, J. Buurman, Detecting changes in process behavior using comparative case clustering, in: CEUR Workshop proceedings of the 5th International Symposium on Data-driven Process Discovery and Analysis, Springer, 2015, pp. 54–75. doi:10.1007/978-3-319-53435-0_3.

[13] A. Bolt, W. M. van der Aalst, M. De Leoni, Finding process variants in event logs, in: OTM Confederated International Confer. "On the Move to Meaningful Internet Systems", Springer, 2017, pp. 45–52. doi:10.1007/978-3-319-69462-7_4.

[14] X. Lu, D. Fahland, F. J. van den Biggelaar, W. van der Aalst, Detecting deviating behaviors without models, in: Business Process Management Workshops, Springer, 2016, pp. 126–139. doi:10.1007/978-3-319-42887-1_11.

[15] J. Martjushev, R. J. C. Bose, W. Van Der Aalst, Change point detection and dealing with gradual and multi-order dynamics in process mining, in: Proceedings of the 14th International Conference on Business Informatics Research, Springer, 2015, pp. 161–178. doi:10.1007/978-3-319-21915-8_11.

[16] A. Maaradji, M. Dumas, M. La Rosa, A. Ostovar, Detecting sudden and gradual drifts in business processes from execution traces, IEEE Transactions on Knowl. and Data Engin. 29 (10) (2017) 2140–2154. doi:10.1109/TKDE.2017.2720601.

[17] A. Yeshchenko, C. Di Ciccio, J. Mendling, A. Polyvyanyy, Visual drift detection for event sequence data of business processes, IEEE Transactions on Visualization and Computer Graphics 28 (8) (2021) 3050–3068. doi:10.1109/TVCG.2021.3050071.

[18] M. Weske, Business process management architectures, Springer, 2007.

[19] M. Von Rosing, S. White, F. Cummins, H. De Man, Business process model and notation-bpmn. (2015).

[20] W. Van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE transactions on knowledge and data engineering 16 (9) (2004) 1128–1142.

[21] S. Smirnov, M. Weidlich, J. Mendling, Business process model abstraction based on synthesis from well-structured behavioral profiles, International journal of cooperative information systems 21 (01) (2012) 55–83. doi:10.1142/S0218843012400035.

[22] W. van Der Aalst, M. Pesic, H. Schonenberg, Declarative workflows: Balancing between flexibility and support, Computer Science-Research and Development 23 (2009) 99–113. doi:10.1007/s00450-009-0057-9.

[23] W. Van Der Aalst, A. Adriansyah, A. K. A. De Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. Van Den Brand, R. Brandtjen, J. Buijs, et al., Process mining manifesto, in: International conference on business process management, Springer, 2011, pp. 169–194. doi:10.1007/978-3-642-28108-2_19.

[24] C. D. Ciccio, M. Mecella, On the discovery of declarative control flows for artful processes, Transactions on Management Information Systems (TMIS) 5 (4) (2015) 1–37. doi:10.1145/2629447.

[25] D. Curran-Everett, C. L. Williams, Explorations in statistics: the analysis of change, Advances in physiology education 39 (2) (2015) 49–54. doi:10.1152/advan.00018.2015.

[26] F. Rösel, S. A. Fahrenkog-Petersen, H. van der Aa, M. Weidlich, A distance measure for privacy-preserving process mining based on feature learning, in: International Conference on Business Process Management, Springer, 2021, pp. 73–85. doi:10.1007/978-3-030-94343-1_6.

[27] J. Grimm, A. Kraus, H. van der Aa, CDLG: A tool for the generation of event logs with concept drifts, in: Workshop proceedings of the International Conference on Business Process Management, Vol. 3216, CEUR-WS, 2022, pp. 92–96. URL https://ceur-ws.org/Vol-3216/paper_241.pdf

[28] T. Jouck, B. Depaire, Ptandloggenerator: a generator for artificial event data, in: L. Azevedo, C. Cabanillas (Eds.), Proceedings of the BPM Demo Track 2016, Co-located with the 14th International Conference on Business Process Management (BPM 2016), Vol. Vol. 1789, CEUR Workshop Proceedings, Rio de Janeiro, Brazil, 2016, pp. 23–27.

URL https://ceur-ws.org/Vol-1789/

[29] H. van der Aa, A. Rebmann, H. Leopold, Natural language-based detection of semantic execution anomalies in event logs, Information Systems 102 (2021) 101824. doi:10.1016/j.is.2021.101824.

[30] A. Berti, S. van Zelst, D. Schuster, PM4Py: a process mining library for python, Software Impacts 17 (2023) 100556. doi:10.1016/j.simpa.2023.100556.

[31] N. C. Chung, B. Miasojedow, M. Startek, A. Gambin, Jaccard/tanimoto similarity test and estimation methods for biological presence-absence data, BMC bioinformatics 20 (Suppl 15) (2019) 644.

[32] J. Bijsterbosch, A. Volgenant, Solving the rectangular assignment problem and applications, Annals of Operations Research 181 (2010) 443–462. doi:10.1007/s10479-010-0757-3.

[33] D. F. Crouse, On implementing 2D rectangular assignment algorithms, IEEE Transactions on Aerospace and Electronic Systems 52 (4) (2016) 1679–1696. doi:10.1109/TAES.2016.140952.