



Archivo Digital UPM houses in digital format the academic and scientific documentation (theses, pfc, articles, etc.) generated at the institution and makes it accessible through the Internet, within the framework of the Budapest Open Access Initiative and the Berlin Declaration, of which the Universidad Politécnica de Madrid is a signatory.

El **Archivo Digital UPM** alberga en formato digital la documentación académica y científica (tesis, pfc, artículos, etc..) generada en la institución y la hace accesible a través de Internet, en el marco de la Iniciativa por el Acceso Abierto de Budapest y la Declaración de Berlín, de la que es signataria la Universidad Politécnica de Madrid.

► **To cite this version:**

P. Rico, F. Cuadrado, J. C. Dueñas, J. Andión and H. A. Parada G., "Business Process Event Prediction Through Scalable Online Learning," in IEEE Access, vol. 9, pp. 136313-136333, 2021, doi: 10.1109/ACCESS.2021.3117147

© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Received July 30, 2021, accepted September 22, 2021, date of publication October 1, 2021, date of current version October 11, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3117147

Business Process Event Prediction Through Scalable Online Learning

PEDRO RICO^{ID}, FÉLIX CUADRADO, JUAN C. DUEÑAS^{ID}, (Senior Member, IEEE),
JAVIER ANDIÓN, (Member, IEEE), AND HUGO A. PARADA G.^{ID}

Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, 28040 Madrid, Spain

Corresponding author: Pedro Rico (ricopinazo@gmail.com)

This work was supported in part by the Ministerio de Ciencia, Investigación y Universidades de España, project PRECON-I4, Retos de la Sociedad, under Grant TIN2017-86520-C3-2-R.

ABSTRACT Predictive process monitoring techniques aim to forecast outcomes of running business process instances. These techniques are based on using predictive models built from past observed behavior, i.e., in an offline setting. However, business process behavior usually changes over time and predictive models are therefore at risk of becoming obsolete. Because of this, the definition of systems that build predictive models through an online setting has recently gained attention. Nevertheless, the scalability of this kind of setting within a context where the amount of data available is experiencing rapid growth is an outstanding issue. To solve this problem, this paper aims to define a framework for event sequence prediction capable of taking advantage of modern distributed processing platforms. An implementation over this framework based on Apache Flink is presented and it is tested upon two different case studies to prove its validity and its capacity to scale.

INDEX TERMS Data mining, distributed processing, monitoring, predictive models.

I. INTRODUCTION

Event data is everywhere, from isolated devices to entire organizations, and its extraordinary growth over the last decades will keep an exponential trend [1]. In the case of organizations, information systems take care of gathering business process event data to build the so-called event logs. Every event within an event log refers to a case and a particular activity executed for that case. Thus, the sequence of events associated with a particular case can be seen as one execution of the business process managed by the event log. As a result, event logs offer a significant source of information and indeed they are the starting point for process mining [2].

Process mining is a research discipline that sits between data mining and process modeling and analysis. The techniques framed within this context are based on extracting knowledge from the mentioned event logs. Thus, they offer new tools to improve and support operational business processes [3].

There exist three main categories of process mining techniques [4]:

- **Process discovery** techniques are the most noticeable among the literature. These techniques aim to learn a model from an event log in order to explain most of the observed behavior. Models usually consist of a graph such as a Petri net.
- **Conformance checking** methods compare the modeled behavior and the observed behavior, represented by the event log. Here, the model may have been produced externally or through process discovery techniques. The starting point of this comparison usually consists of the alignment between the model and the event log to check the conformance.
- **Model enhancement** intends to improve or extend existing process models. Non-fitting models can be improved through the results obtained from conformance checking alignments. Moreover, the alignment between the process model and the event log enables extending the model using additional information from the data set.

One of the main limitations of the techniques framed within these categories is that they are usually used in offline settings. They feed from the historical information offered by event logs to provide knowledge about processes without considering running process instances. However, process mining

The associate editor coordinating the review of this manuscript and approving it for publication was Zhan Bu^{ID}.

techniques can play an important role for operational decision support [5].

Indeed, the applying of process mining techniques over running cases has gained much attention recently. This specific sub-field within process mining is referred to as PPM (predictive process monitoring) [6]. This methods often rely on the capacity to generate predictive models from process logs that can be further used to predict the behavior and performance of running process instances [7]. This has a crucial role in supporting organizations on proactive management tasks to improve process performance and detect risks [8].

Several objects of prediction can be found among PPM approaches. The most recurrent one, the remaining execution time of process instances, reduces to a regression problem. Methods to predict a single attribute about instance performance are also widely addressed and are usually tackled as classification problems. This is also the case of risk probability estimation and next event prediction, which have received less attention [8]. Concretely, predicting the entire sequence of events to completion is one of the least covered categories. Nonetheless, the knowledge of such outcomes can be especially useful to support business processes and can serve as the first step to subsequent checks on process monitoring. For this reason, through this paper, sequence prediction is considered as the main object of prediction.

As for the correctness of these predictions, it depends on the performance of the predictive models. This point is especially challenging in real-time process monitoring systems, since process behavior often evolves along time. This can be caused by external factors, such as the appearance of new regulations or new market needs [9], or internal factors, such as variations of the own process performance [8]. Predictive models therefore need to be retrained periodically to ensure that its performance does not deteriorate with this evolution [8]. Indeed, the definition of online learning settings has become a matter of interest to tackle these scenarios [10], [11].

Additionally, another challenge for PPM settings is the processing of high volume real time event streams. As an example, the size of these event streams can range up to millions of transactions every minute in a stock exchange corporation [12]. This challenge has been barely tackled within PPM field and can be addressed by using modern distributed stream processing platforms [8], such as Apache Spark [13] or Apache Flink [14].

Taking into account all above, the main contribution of this paper is the definition of a framework to tackle event sequence prediction in an online learning setting. Particularly, given a running process instance, the framework proposed addresses the prediction of the remaining activities and the times when these activities are executed. Besides, an implementation example based on a distributed processing system is provided in order to show the capacity of the proposal to be mapped into this kind of systems and therefore to scale to large event streams.

The remainder of this paper is structured as follows. Section II reviews related work in approaches similar to the presented one. Section III formalizes and develops a framework to carry out the prediction of event sequences. Section IV describes the implementation carried out to validate the framework proposal and provide several guidelines to map it to a stream processing platform such as Apache Flink. Section V presents the results obtained from applying the implementation over two different case studies in order to evaluate the proposal. Section VI concludes this paper and points out several future lines of research to extend this work.

II. RELATED WORK

From a conceptual perspective, business process instances can be described through event sequences. An event within a process instance is fundamentally defined by the kind of activity executed and the timestamp fixing the time when this activity was executed. Given an input sequence of events known about a running process instance, the general goal of PPM methods is to predict future outcomes for that instance.

Among the approaches to predict the next event, a number of methods can be found. There exist works that evaluate the use of classical methods, such as transition matrix [15], HMM (hidden markov model) [16], [17], or PFA (probabilistic finite automaton) [18]–[20]. Machine learning methods are also evaluated to tackle next event prediction, such as decision trees [21] or random forest [22]. Finally, there exist several works on the use of deep learning methods, such as recurrent neural networks [23]–[26].

Among these proposals, only a few tackle the prediction of complete event sequences. In the following, these proposals and others that tackle sequence prediction as the main object of prediction are analyzed and differences with this proposal are discussed. Among these works, two different approaches are considered: iterative predictions and path searching.

Iterative predictions are built on top of a model capable of predicting both the next activity and timestamp. The model is fed iteratively using every prediction as the input of the next iteration. The sequence of events is therefore formed as the concatenation of these predictions. To implement these methods, additional logic is needed in order to identify those activities that mark the end of the current sequence. The main counterpart of this approach is that infinite loops might be generated when an already traversed activity is again predicted. Therefore, it is also necessary to implement additional logic to handle these situations.

This approach is proposed in [23], [24], and [25]. The three of them are based on a model to predict the next activity and timestamp using LSTM (long short-term memory) networks. Besides, [25] propose a method to avoid infinite loop situations. This consists of artificially lowering the probability of an already executed activity to occur. However, it may alter the nature of the underlying models.

Path searching refers to those techniques that try to find the most likely path inside a graph that describes the process behavior. This graph can be built by using process discovery

techniques. In [27] and [28], this is tackled by translating the problem of finding the path with greater probability to a shortest path problem. The probability of a path is computed as the product of the probability of every edge. Here, a lower probability is associated with a greater cost. Therefore, the shortest path is that maximizing this product.

However, this approach leads to some problems when handling cyclic behavior. For example, assume that a graph is built based on three observed sequences of activities that are identical, $a \rightarrow b \rightarrow b \rightarrow c$. The result may be the graph depicted in FIGURE 1. As shown, the probability of activity b after itself is 0.5 and is equal to the probability of c after b because both transitions happen three times within the observed sequences. Therefore, after activity a , the probability of path $\rightarrow b \rightarrow c$ is $1 \cdot 0.5$, whereas the probability of path $\rightarrow b \rightarrow b \rightarrow c$ is $1 \cdot 0.5 \cdot 0.5$. Hence, the path predicted by the model after this activity is $\rightarrow b \rightarrow c$, which does not match the observed behavior.

Of course, this may be solved through a smart encoding of the observed activities. The model may distinguish between activity b when coming from a and activity b when coming from b . However, this approach requires previous knowledge about process behavior and does not allow for handling cycles with an arbitrary number of repetitions. Moreover, previous knowledge of the process is also required because this approach needs the definition of accepting nodes where the paths end, c in the case of FIGURE 1.

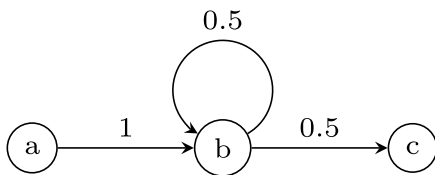


FIGURE 1. Graph generated by the repetition of the event sequence $a \rightarrow b \rightarrow b \rightarrow c$ three times.

Additionally, these works tackle the prediction of the time to completion for the entire trace as an isolated problem, but do not provide a way of producing timestamps for individual activities.

Recently, the use of process trees has been proposed in order to predict future activities in [29]. However, this approach implies a more complex treatment, meaning a barrier for scalability. For this reason, this method has been left out of the scope of this work.

All these proposals forecast sequences of events using information about isolated transitions to build complete paths. Moreover, they all need a priori knowledge about what events mark the end of a case or additional logic to handle the ending point of a prediction. In contrast, the proposal presented in this paper consists of mapping the problem of predicting a sequence to a classification problem. Thus, the sequence predicted consists of a class, and information about complete sequences instead of isolated transitions is

recorded. Therefore, reproducing loops but avoiding infinite loops to arise at the same time is supported by definition and there is no need for additional logic to handle sequence finalization.

In addition, all mentioned works assume that there exists an event log to create the model from. Moreover, they assume that the monitored process behavior does not change over time. Whereas it can be true most of the time, there exist certain scenarios where this cannot be ignored.

In general, all these approaches do not take into account considerations about scalability. Indeed, as mentioned earlier, processing real-time event streams in the context of PPM making use of state-of-the-art big data platforms is a current challenge [8]. However, there are proposals within the process mining field that tackle this. For example, [30] develops a way to tackle process discovery techniques using a general map-reduce approach. For process discovery purposes as well, [31] presents a framework that is implemented on top of Apache Flink. Therefore, one of the main goals of this paper is to define a solution that can be mapped to a big data platform in the context of PPM and particularly focused on event sequence prediction.

III. PREDICTIVE PROCESS MONITORING FRAMEWORK

This section formalizes a solution to predict the future sequence of activities until completion and their timestamps given an ongoing process instance. In order to do this, this method follows two steps, consisting of the prediction of the entire sequence of activities remaining and the subsequent forecasting of their timestamps. The rest of this section is structured as follows. Subsection III-A defines the kind of scenarios that this work focuses on. Subsection III-B formalizes several concepts needed in the rest of this paper. Finally, subsections III-C and III-D define the two prediction steps mentioned before and the latter presents the final solution combining them.

A. CONTEXT ASSUMPTIONS

As a matter of fact, defining a general PPM approach that suits all kinds of scenarios is extremely complicated. The scope of this work is slightly bounded to the characteristics of the case study. As a result, the proposed framework is especially suited for processes that fulfill the conditions listed below.

- Time has a cyclic influence over the delay between activities. This can be caused by a human factor, as in the case studies analyzed in this paper, since human activity is bounded to fixed schedules such as workdays or workweeks. However, other factors can cause this behavior.
- Control flow does not strongly depend on time, so there is no need to include time as an input variable in order to make next activity prediction.

It should be noted that the application of this approach to scenarios that do not fulfill these conditions can lead to poor

results. In return, the proposal offers a couple of generalizations that can be mapped to many other cases.

- It can handle cyclic behavior of arbitrary complexity.
- It does not need a priori knowledge about possible events, nor start and finish events. As a consequence, the solution does not need additional logic to handle special events and is robust against noise caused by the loss of this kind of events.

B. PRELIMINARIES

In order to define the prediction of future events, a definition of several general and domain-related concepts is needed. The following definitions are based on those from [32], but with noticeable differences, including the consideration of stream handling.

First of all, for a given set X , X^* stands for the set of all sequences over X . A sequence $x \in X^*$ is a partial function $x: \mathbb{N}^+ \mapsto X$. It is denoted as $\langle x_1, x_2, x_3, \dots \rangle$, where $\forall i \in \mathbb{N}^+$, x_i states for the element of X in position i , $x(i)$, if defined. Besides, $\|x\|$ states for the set of elements within the image of x , i.e., it refers to the set of elements in the sequence.

Throughout the definition of this framework, as streaming scenarios are tackled, it is necessary to distinguish between bounded and unbounded sequences, referred to as lists and streams, respectively. The set of all lists over X is denoted as $X^\dagger$ and they are functions in $\{1 \dots n\} \rightarrow X$, where $|x| = n$ is used to denote the length of the list, for coherence with the set cardinality notation $|X|$. Meanwhile, the set of all streams over X is denoted as $X^\ddagger$ and they are functions in $\mathbb{N}^+ \rightarrow X$.

Several functions are defined in order to allow sequence handling. A function $hd^i \in X^* \rightarrow X^\dagger$ to extract the head of a sequence is defined, where $hd^i(x) = \langle x_1, x_2, \dots, x_{\min\{i, |x|\}} \rangle$. Likewise, the tail of a list is extracted through a function $tl^i \in X^\dagger \rightarrow X^\dagger$, defined as $tl^i(x) = \langle x_{\max\{1, |x|-i+1\}}, \dots, x_{|x|-1}, x_{|x|} \rangle$. In contrast to hd , this function cannot be applied over general sequences in X^* , because it needs the definition of length provided by lists in $X^\dagger$. Note that, if $i \geq |x|$, $hd^i(x) = tl^i(x) = x$. Besides, the concatenation of lists is expressed through the use of $\#$ operator, i.e., $\langle x_1, \dots, x_n \rangle \# \langle y_1, \dots, y_m \rangle = \langle x_1, \dots, x_n, y_1, \dots, y_m \rangle$.

On the other hand, $\mathbb{B}(X)$ stands for the set of multi-sets or bags over X . A bag $\mathcal{X} \in \mathbb{B}(X)$ is a mapping $X \rightarrow \mathbb{Z}$ where $\mathcal{X}(x)$ denotes how many times $x \in X$ is included in $\mathcal{X}$. This approach is similar to the one provided by [33]. Given two bags $\mathcal{X}, \mathcal{Y} \in \mathbb{B}(X)$, addition and subtraction operations can be defined as follows. $\mathcal{A} = \mathcal{X} + \mathcal{Y}$ means $\mathcal{A}(x) = \mathcal{X}(x) + \mathcal{Y}(x) \forall x \in X$ and $\mathcal{A} = \mathcal{X} - \mathcal{Y}$ means $\mathcal{A}(x) = \mathcal{X}(x) - \mathcal{Y}(x) \forall x \in X$, where $\mathcal{A} \in \mathbb{B}(X)$ in any case. For example, given $X = \{a, b, c\}$, $\mathcal{X} = [a, b^2, c^{-1}]$ denotes a bag where a is included $\mathcal{X}(a) = 1$ time, b is included $\mathcal{X}(b) = 2$ times and c is included $\mathcal{X}(c) = -1$ times. Moreover, $\mathcal{X} + [a^{-1}, b^2] = [b^4, c^{-1}]$ and $\mathcal{X} - [a^{-1}, b^2] = [a^2, c^{-1}]$.

On top of that, within the process domain, an event usually refers to an activity within a process and is related to a specific

process instance. It has usually a timestamp associated with it and a set of features. Every process instance has a case identifier associated with it.

Let C be the set of case identifiers, A the set of possible activities, $\mathcal{T}$ the time domain, and F the set of feature tuples. Then the universe of events is $E \subseteq C \times A \times \mathcal{T} \times F$, i.e., an event is defined by a case identifier, an activity, a timestamp, and a tuple of features. Notation $\pi_x(e)$ is used to refer to the different fields of an event. Given an event $e = (c, a, t, f)$, $\pi_c(e) = c$, $\pi_a(e) = a$, $\pi_t(e) = t$, and $\pi_f(e) = f$. Additionally, the timestamp of the given event can be referred to as $\bar{e} = \pi_t(e) = t$.

On the other hand, to handle the cyclic influence time has over the process, it is convenient to allow expressing time in terms of the current cycle. Let T be the period defining the cycle. Then $\hat{t} = t - \lfloor t/T \rfloor \cdot T$ states for the time elapsed from the start of the current interval. Here $\lfloor x \rfloor$ refers to the greatest integer less than or equal to x , so $\lfloor t/T \rfloor$ is equal to the number of intervals of length T elapsed so far.

This framework aims to address streaming scenarios and, for this reason, it is necessary to define a global event stream, $\phi \in E^\dagger$, that states the order in which events are ingested into the system:

$$\phi = \langle e_1, e_2, e_3, \dots \rangle$$

The order in which events are ingested into the system is referred to as ingestion order and the addition of each new event is referred to as ingestion step. These ingestion steps are defined over $\mathbb{N}$, where 0 (excluded from the domain of ϕ) states for the initial step where no events have arrived, 1 states for the step where the first event, e_1 , has arrived, and so on.

On the other hand, it is important to differentiate between ordered and unordered event streams. An ordered event stream, for any pair of events ϕ_i and ϕ_j , where $i, j \in \mathbb{N}$ and $i < j$, satisfies that $\bar{\phi}_i < \bar{\phi}_j$. Moreover, this differentiation can be considered casewise. Thus, an event stream is said to satisfy casewise ordering if $\bar{\phi}_i < \bar{\phi}_j \forall i, j \in \mathbb{N}, i < j \wedge \pi_c(\phi_i) = \pi_c(\phi_j)$. This constraint is very useful since it is quite easy to fulfill under situations where each process instance takes a long time and at the same time it makes computing much easier, as seen later. Indeed, the framework proposed addresses casewise ordered streams, although it is able to handle casewise unordered even streams under certain conditions that are discussed.

In parallel, a trace r can be defined as a sequence of events of finite length, $r \in R \subseteq E^\dagger$, such that its timestamps are nondecreasing:

$$R = \{r \in E^\dagger \mid \bar{r}_i < \bar{r}_j \forall i, j \in 1 \dots |r|, i < j\}$$

By extension of the event notation, when working with traces π_a is used to refer to the list of activities, i.e., $\pi_a(r) = \langle \pi_a(r_1), \pi_a(r_2), \dots, \pi_a(r_{|r|}) \rangle$.

Usually, the set of all event traces for every case in a process is known as an event log. In contrast to typical approaches, as an event stream is used as the main input, the cardinality of the event log is unbounded and therefore a global view

of this event log is not available. As time goes by, a partial view of the event log composed by the events ingested into the system so far keeps growing. To obtain this partial view, it is necessary to define a translation from known part of the global event stream into individual event traces for every case. This is done through the use of a function $\sigma_n \in C \rightarrow R$, where n stands for the ingestion step, satisfying the condition:

$$\|\sigma_n(c)\| = \{(c', a, t, f) \in \|hd^n(\phi)\| \mid c' = c\}$$

As seen, the trace $r = \sigma_n(c)$ only contains events observed until step n . This makes explicit the uncertainty about case completion because all events referring a case c in E may not have been ingested into the system by that step. For a given ingestion step n , a trace with all the received events referring to a particular case c can be built, but there is a possibility for this trace to change in the future. This fact constraints the way the framework is developed.

To properly address this fact, it is convenient to introduce the definition of the complete trace of a case, $\sigma_\infty(c)$. Given a case c , there exists an ingestion step m such that $\sigma_n(c) = \sigma_m(c) \forall n \geq m$. Then, the complete trace of c is $\sigma_\infty(c) = \sigma_m(c)$, i.e., the final version of the trace.

An example of a global event stream can be seen on Table 1. This example includes events from three different cases, c_1 , c_2 , and c_3 . For instance, the known event trace associated with the case c_1 at the ingestion step 4, i.e., at the moment in which event e_4 is ingested into the system, is given by $\sigma_4(c_1) = \langle e_1, e_3 \rangle$. Once events e_5 and e_6 arrive, the known event trace becomes $\sigma_6(c_1) = \langle e_1, e_3, e_6 \rangle$. Here, the event e_1 is a tuple $(c_1, \text{New}, t_0, (\text{problem}, \text{yes}))$ and $(\text{problem}, \text{yes}) \in F$ is the feature tuple associated with e_1 . It is important to note that a specific event log can contain an arbitrary number of features for every event and the number of possible values for each one can be large. However, for this framework to work properly, it is necessary to consider only a subset of these features, so the number of features does not grow excessively and these have a small number of possible values. If it is not guaranteed, efficient encoding may become infeasible.

TABLE 1. Event stream example.

Event	Case	Activity	Timestamp	Category	IsUrgent
e_1	c_1	New	t_0	problem	yes
e_2	c_2	New	t_3	question	no
e_3	c_1	Assigned	t_4	problem	yes
e_4	c_2	Resolved	t_7	question	no
e_5	c_3	New	t_{10}	question	yes
e_6	c_1	Resolved	t_{12}	problem	yes
e_7	c_3	Assigned	t_{23}	question	yes

In order to make predictions from the known part of a given trace, an encoding that represents the current state to make predictions from is needed. These encodings are referred to as states. Thus, given a trace r , a state s is obtained through a representation function $s = \rho(r)$, $\rho \in E^\dagger \rightarrow S$. This allows for much flexibility in the way a state can be defined.

Nonetheless, the proposal constraints the form of the function ρ , and consequently the composition of S is as follows. Given a trace $r = \langle e_1, e_2, \dots, e_n \rangle$, ρ is defined as follows.

$$\rho(r) = (\pi_a(h^{h+1}(r)), \pi_f(e_n) \uparrow G) \quad (1)$$

Here, if the feature tuple f has the form $f = (f_1, f_2, \dots, f_n)$, then $G \subseteq \{1, 2, \dots, n\}$ and $f \uparrow G$ stands for a projection of the feature tuple including only the features stated by G , e.g., if $G = \{1, 3\}$ then $f \uparrow G = (f_1, f_3)$.

Thus, states are defined as tuples, $S \subseteq A^\dagger \times F'$, where F' contains the feature tuples in F after the projection with G and h fixes the event horizon, i.e., the number of previous events affecting the definition of a certain state. It should be noted that a large value for h or the inclusion of features of the feature tuple that have a large number of possible values can lead to an overfitted model.

Additionally, given a state $s = (\langle a_1, a_2, \dots, a_n \rangle, f)$, the last executed activity, which led therefore to this state, is denoted as $\pi_a(s) = a_n$ and the tuple of selected features is denoted as $\pi_f(s) = f$, by extension of event accessors notation.

This way, the state reached by the case c_1 in Table 1 after the execution of the three events, assuming $h = 1$ and $G = \{1, 2\}$, is $s = (\langle \text{Assigned}, \text{Resolved} \rangle, (\text{problem}, \text{yes}))$. This state is taken as input to make predictions, so in this case, the prediction model takes into account that the last two executed activities are Assigned and $\pi_a(s) = \text{Resolved}$, in this order, and the last feature tuple is $\pi_f(s) = (\text{problem}, \text{yes})$.

This approach to define states is similar to the so-called maximal-horizon abstraction described in [32]. In contrast, instead of using an annotated transition system to define a new state from the previous state and a new event, this work defines new states from the updated trace. This enables handling unordered event streams, although the effects of these cases are left out of the scope of this paper, as this is not the main goal of this work.

Finally, from (1), another definition is given to support the combination of the prediction steps. Given a trace $r = \langle e_1, e_2, \dots, e_n \rangle$, a path is defined as $\mu \in S^\dagger$, $\mu = \langle s_1, s_2, \dots, s_n \rangle$, where $s_i = \rho(hd^i(r)) \forall i \in \{1 \dots n\}$.

C. PATH PREDICTION

Every time a new event ϕ_n associated with case $c = \pi_c(\phi_n)$ is ingested into the system a new trace $r' = \sigma_n(c)$ is known. If the event stream is casewise ordered, as it is assumed to be, this trace is the header of the complete trace associated with that case $r = \sigma_\infty(c)$, i.e., $|r'| \leq |r|$ and $r' = hd^{|r'|}(r)$. Hence, the goal of this framework is to predict the complete trace with the known information.

The sequence of states, i.e., the path, traveled by r is $\mu = \langle \rho(hd^1(r)), \dots, \rho(hd^{|r'|}(r)), \dots, \rho(hd^{|r|}(r)) \rangle = \langle s_1, \dots, s_{|r'|}, \dots, s_{|r|} \rangle$. Therefore, after the state defined by the known part $s_{|r'|} = \rho(r') = \rho(hd^{|r'|}(r))$ there is in turn a path that continues to the end $\langle s_{|r'|+1}, s_{|r'|+2}, \dots, s_{|r|} \rangle$. Note that if $r' = r$ then the path after $s_{|r'|}$ is the empty list. Taking this into account, the goal of the first step of this

framework is to predict the path following the known state $s_{|r'|}$. Moreover, states are handled instead of isolated activities because they add useful information to append timestamps to these sequences in the next prediction step, described in subsection III-D. As a consequence, the encoding scheme defined by ρ influences the way the predictive model is built, since it defines the possible state values.

In order to predict the path following a state, a function $\omega_n \in S \rightarrow S^\dagger$ has to be defined. This function is built from the samples available in the global event stream ϕ . First of all, it is necessary to define how to obtain those samples from a single trace r assumed to be complete. For every state within that trace, a sample of the remaining path to the end is taken. These samples are then grouped into a bag. This way, a function $sl \in (R, S) \rightarrow \mathbb{B}(S^\dagger)$ that associates every state s within r with its bag of samples $\mathbb{B}(S^\dagger)$ is defined as follows.

$$sl(r, s) = \sum_{\substack{1 \leq i \leq |r| \\ s = \rho(hd^i(r))}} [\langle \rho(hd^{i+1}(r)), \dots, \rho(hd^{|r|}(r)) \rangle] \quad (2)$$

Every element in the summation defined in (2) is a bag with a unique sample, so the result consists in a bag containing all these samples. The samples obtained from all traces observed in ϕ after every state can be grouped to form a global multi-set. This is described through a function $P_n \in S \rightarrow \mathbb{B}(S^\dagger)$ that associates every state s with its bag of samples.

$$P_n(s) = \sum_{c \in C} sl(\sigma_n(c), s) \quad (3)$$

Once these samples are known, function ω_n can be defined. The main goal of this paper is to provide a scalable framework. Therefore, the predictive model has been reduced to the simplest alternative: for a given state, the path predicted after it is simply the one that has appeared more times after that state. This way, for a given state s and a given ingestion step n , the prediction is the element on the bag $P_n(s)$ with greater multiplicity. Hence, the mentioned function ω_n is defined as follows.

$$\omega_n(s) = \operatorname{argmax}_{\mu \in S^\dagger} \left(\mathcal{P}(\mu) \Big|_{\mathcal{P} = P_n(s)} \right) \quad (4)$$

It is noticeable that leveraging from next event prediction to future sequence prediction eliminates the need of additional logic to handle trace finalization. Instead, when no remaining events are expected, the prediction consists of an empty path $\langle \rangle$.

The main problem to address here is related to the uncertainty about process completeness. Since there is no a priori knowledge about which events mark the end of a case (or even more, this kind of event does not even exist), there is no way to know when a given process instance has actually ended. Clearly, a timeout can be set every time a new event arrives to trigger the collection of the gathered information about the trace. The problem with this approach is that there is a trade-off between setting the timeout long enough to be

sure an instance actually ended but short enough to be able to use new information under scenarios with high concept drift.

Instead of finding a point within such a trade-off, the proposal here is to incorporate new information as it comes, assuming traces are complete and correcting errors as soon as they arise. This approach is very similar to the one followed by [31] and it is able to handle unordered event streams by definition.

The incorporation of new information means that a new model is generated every time a new event arrives. In order to address scalability concerns, it is necessary to define $P_n(s)$ in terms of the previous version, $P_{n-1}(s)$. This way, incremental updates are identified.

Let ϕ_n be the n -th event ingested into the system, associated with case $c_n = \pi_c(\phi_n)$. Then, an updating expression can be extracted from (3) by splitting the cases as follows.

$$\begin{aligned} P_n(s) &= \sum_{c \in C - \{c_n\}} [sl(\sigma_n(c), s)] + sl(\sigma_n(c_n), s) \\ &= \sum_{c \in C - \{c_n\}} [sl(\sigma_{n-1}(c), s)] + sl(\sigma_{n-1}(c_n), s) \\ &\quad + sl(\sigma_{n-1}(c_n), s) - sl(\sigma_{n-1}(c_n), s) \\ &= \sum_{c \in C - \{c_n\}} [sl(\sigma_{n-1}(c), s)] + sl(\sigma_{n-1}(c_n), s) \\ &\quad + sl(\sigma_n(c_n), s) - sl(\sigma_{n-1}(c_n), s) \\ &= \sum_{c \in C} [sl(\sigma_{n-1}(c), s)] + sl(\sigma_n(c_n), s) \\ &\quad - sl(\sigma_{n-1}(c_n), s) \\ &= P_{n-1}(s) + sl(\sigma_n(c_n), s) - sl(\sigma_{n-1}(c_n), s) \end{aligned}$$

This holds since $\sigma_n(c) = \sigma_{n-1}(c) \forall c \neq c_n$.

This updating process is represented on FIGURE 2, where $P_3(s)$ is computed. Here, the three first events of the event stream are assumed to be $hd^3(\phi) = \langle e_1, e_2, e_3 \rangle$. Here, the three events are associated with the same case identifier, c_{id} . $\pi_a(e_1) = a$, $\pi_a(e_2) = c$, and $\pi_a(e_3) = b$ and $\bar{e}_1 < \bar{e}_3 < \bar{e}_2$. As noted, an unordered event stream is considered to showcase its management. Besides, no features are considered. For the sake of simplicity, traces are represented as

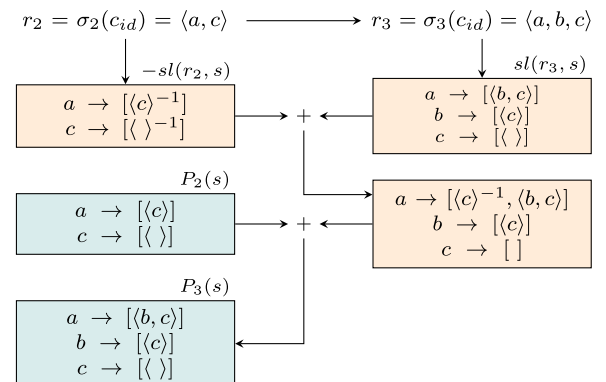


FIGURE 2. Updating process for $P_n(s)$ when a new event with activity b arrives.

lists of activities and the representation function is reduced to $\rho(r) = \pi_a(r|_r)$ ($h = 0$ and $G = \{\}$), so paths are also represented as lists of activities. At the end of the process, it can be observed that the predicted paths after every state correspond to the observed behavior in $\sigma_3(c_{id})$, which includes all known information so far. It is important to note that these updates do not grow as the number of traces received increases, i.e., they are scalable.

D. TIME PREDICTION

As mentioned before, once the entire list of activities has been predicted, it needs to be labeled with timestamps. This is tackled by estimating the delays between activities iteratively until the trace is completely timed.

As stated in subsection III-A, time has a cyclic influence over the delay between activities in the scenarios addressed. Hence, it is necessary to model the delay behavior along the given cycle, which has to be known. Besides, this solution aims to support the handling of concept drift. Both problems can be addressed at once by recording and maintaining a list of delay samples labeled with timestamps. For that, given a trace r and a certain position i within the trace, a timing function, tm , to obtain a tuple containing the timestamp of the event at this position and the delay to the next event is defined as follows.

$$tm(r, i) = (\overline{r(i-1)}, \overline{r(i)} - \overline{r(i-1)}) \quad (5)$$

Let $\phi_n = (c_n, a, t, f) = e$, $\sigma_n(c_n) = r$, and $r_i = e$. Using (5), a function τ_n to obtain a list of all samples for a given previous state s_p and next state s_n can be defined recursively.

$$\tau_n(s_p, s_n) = \begin{cases} \tau_{n-1}(s_p, s_n) \# tm(r, i) & \text{if } i > 1 \\ & \wedge \rho(hd^{i-1}(r)) = s_p \\ & \wedge \rho(hd^i(r)) = s_n \\ \tau_{n-1}(s_p, s_n) & \text{otherwise} \end{cases} \quad (6)$$

For this definition to work properly, it must be ensured that $\tau_0(s_p, s_n) = \langle \rangle$.

On the other hand, if the input event stream does not satisfy casewise ordering, the samples included in the sequence might not be correct. This happens when the state produced by a new event ϕ_n , $c_n = \pi_c(\phi_n)$, is defined before previous events of the same case c_n arrive. Let $r = \sigma_\infty(c_n)$ be the complete trace of that case. Then, the mentioned state is given by $s = \rho(hd^i(\sigma_\infty(c_n)))$ where $r_i = \phi_n$. If ϕ is not casewise ordered, it is trivial to prove that $s = \rho(hd^i(\sigma_n(c_n)))$, as defined in (6), can not be guaranteed. In that case, $\rho(hd^i(\sigma_n(c_n)))$ would be a fake state, i.e., a state that does not take place in the real process but appears in the partial view of the process provided by the event stream because of the lateness of previous events.

As a result, working with casewise unordered event streams can lead to the presence of noise among the samples, since fake states can be introduced into the list defined by (6). This could be solved through corrections. However, whereas

corrections are necessary to record path samples even though the input event stream is ordered, as seen in subsection III-C, they are only necessary for this prediction step under this scenario. Besides, casewise ordering is not as restrictive as total ordering criteria. Moreover, the presence of noise can be handled by processing those samples appropriately to a certain extent.

It must be noted that the growth of the list defined by (6) is unbounded. Of course, this prevents the streaming system supported by this framework from being scalable. Therefore, only the last w samples can be maintained in order to keep the list manageable, defining a window. If there exists a time factor that has a cyclic influence over the delay between activities, this window has to be long enough to go over several of those cycles. For example, if the workweek is the time factor to consider, the window has to go through several weeks in order to model the cyclic effect produced by this factor. Taking into account all this, a delay estimation function λ_n that takes into consideration the start time of a transition between two states is defined.

$$\lambda_n(s_p, s_n, t) = dl(tl^w(\tau_n(s_p, s_n)), t) \quad (7)$$

Here dl states for an arbitrary computation of the delays on $tl^w(\tau_n(s_p, s_n))$. t can be considered to weigh the samples or even select a subset of them to use similar samples within the cycle or handle concept drift.

Finally, in order to allow the combination of the two prediction steps, given the next state s , a function $ev \in (R, S) \rightarrow E$ to build a synthetic event following a given trace r through the use of (7) is defined.

$$ev(r, s) = (\pi_c(r|_r), \pi_a(s), \overline{r|_r} + \lambda_n(\rho(r), s, \overline{r|_r}), \pi_f(s)) \quad (8)$$

Then, from (4) and (8), a function γ capable of producing an entire event trace from an uncompleted list of events is defined as follows.

$$\tilde{\gamma}_n(r, \mu) = \begin{cases} \gamma_r(r \# ev(r, \mu_1), tl^{|\mu|-1}(\mu)) & \text{if } |\mu| > 0 \\ r & \text{otherwise} \end{cases} \quad (9)$$

$$\gamma_n(r) = \tilde{\gamma}_n(r, \omega_n(r))$$

Ideally, given a case c and an ingestion step n , it must be satisfied that $\gamma_n(\sigma_n(c)) = \sigma_\infty(c)$.

The delay estimation function dl used in (7) can be defined in many ways. The simplest approach would be to ignore the time the delay starts from and make a computation using all samples regardless of the timestamps associated with them. For example, the delay estimation may consist of the average of all delays in the window. In that case, the delay estimation function would be defined as follows.

$$dl(w, t) = avg(\{d \mid (d, t) \in \|w\|\}) \quad (10)$$

Nevertheless, the use of the average in the presence of noise can lead to poor results. This can take place under scenarios with arbitrary time distributions, as the case under study in section V, or if input event stream is not casewise ordered. As an alternative, the median can be used as a simpler and safer option, since noisy samples get filtered. This is

especially important since, as explained in subsection III-B, the computing of casewise unordered event streams can produce noise. Thus, the delay estimation function would be as follows.

$$dl(w, t) = med(\{d \mid (d, t) \in \|w\|\}) \quad (11)$$

However, to take into consideration the cyclic influence time can have on delay behavior, it is convenient to take into account only samples that occur in a similar interval within the cycle that the one being predicted. K-nearest neighbors has shown a good performance for this purpose, as seen in section V. Given a list of samples, w , the k-nearest neighbors is a set given by a function nn_k such that the set cardinality is constrained as $|nn_k(w, t)| = \min\{k, |w|\}$ and the elements in the set are defined as follows.

$$nn_k(w, t) = \{(t_1, d_1) \in \|w\| \mid |\hat{t} - \hat{t}_1| \leq |\hat{t} - \hat{t}_2| \\ \forall (t_2, d_2) \in (\|w\| - nn_k(w, t))\} \quad (12)$$

Using (12), the delay estimation function can be defined through an arbitrary computation of these samples. Again, the previously seen estimators, median and average, can be used.

$$dl(w, t) = avg(\{d \mid (d, t) \in nn_k(w, t)\}) \quad (13)$$

$$dl(w, t) = med(\{d \mid (d, t) \in nn_k(w, t)\}) \quad (14)$$

Note that equations (13) and (14) reduce to (10) and (11), respectively, if $k = \infty$.

Using these estimators, based on k-nearest neighbors, behavioral changes along a given cycle of the process can be accurately predicted. Moreover, in this case, median estimator becomes especially useful, since it is capable of forecasting feasible delays as it takes a real delay from the past.

As an example, a target transition whose ending timestamp has to be predicted is assumed to start on Friday and three past samples of this transition are used, one of them ending within the same day and the others ending on Monday. If the average is used as estimator, the end of the transition could fall on Sunday, which may violate workweek criteria. In contrast, the output of a median estimator would fall on Monday, which is feasible in terms of the underlying time behavior of processes influenced by the workweek.

Table 2 provides a quick reference for the concepts introduced in this section.

IV. IMPLEMENTATION

This section describes the implementation carried out to validate the framework proposal and provide several guidelines to map it to a stream processing platform such as Apache Flink. First of all, in order to properly understand the details of the proposal, it is important to know several ideas related to Flink's architecture and API. These are presented in subsection IV-A. Then, the system design itself is described in subsection IV-B. Finally, since one of the main goals of this work is to provide a scalable solution, subsection IV-C details the considerations to have into account when preparing this system to scale.

TABLE 2. Notation summary.

Concept	Symbol	Description
Generic objects		
Case	$c \in C$	Case identifier
Event	$e \in E$	Execution of an activity associated to a case
Trace	$r \in R$	Finite sequence of events, usually for a case
State	$s \in S$	Representation of the state of a case
Path	$\mu \in S^\dagger$	Finite sequence of states traversed by a case
Global framework objects		
Events	$\phi \in E^\dagger$	Stream of events ingested into the system
Log	$\sigma_n(c)$	Trace related to a given case
Paths	$P_n(s)$	Multi-set of paths following every state
Delays	$\tau_n(s_p, s_n)$	List of delay samples
Framework functions		
Path	$\omega_n(s)$	Predict a path following a state
Delay	$\lambda_n(s_p, s_n, t)$	Predicts the delay between two states
Trace	$\gamma_n(r)$	Predict the final version of a known trace

A. FLINK CHARACTERISTICS

To understand the implementation scheme presented in this section it is important to know several ideas behind Apache Flink [14] for supporting streaming pipeline execution.

As a distributed processing engine, the architecture of Apache Flink's deployments is cluster based. A cluster is made up of processes that can run in different machines. There are two kinds of processes: JobManagers and TaskManagers. A JobManager process takes care of distributing the work onto the TaskManagers instances. There is only one JobManager working at a time, although there can be more than one in stand-by mode for high availability purposes. Meanwhile, TaskManagers are the processes where the actual work is carried out.

Apache Flink uses JobGraphs as the main abstraction to represent streaming pipelines. These graphs are composed of operators and intermediate results. They are obtained by translating them from the application code. Hence, in order to deploy an application, some client has to make this translation and submit the JobGraph in to a JobManager. Each one of the operators a JobGraph is made up of can be implemented by several instances, which allows to execute them in parallel. Thus, each instance of a certain operator handles a subset of the data the operator has to manage.

Regarding its internals, the basic units of execution within Flink's engine are tasks. Each operator instance is executed in a different task, so it is isolated from the rest of instances of the same operator. This means that if the parallelism of a given operator is four, there have to be necessarily four different tasks to handle these instances.

At the same time, each task can handle instances from one or more chained operators. This optimization is important regarding the way tasks intercommunicate. Since every task is executed by a different thread, they use buffering in order to interchange data between them. Thus, in general, every task

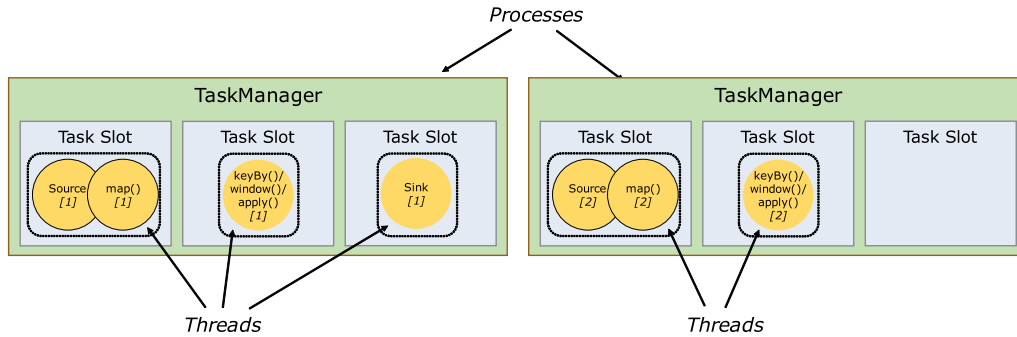


FIGURE 3. Example of task distribution among several TaskManagers and task slots in Apache Flink [14].

has an input and an output buffer. By assigning several operators to the same task, resources are used more efficiently.

The task concept is used to define the way execution resources in Flink are handled. Every TaskManager, which is a JVM process, has one or more so-called task slots. Each task slot represents a fixed subset of resources of the TaskManager. Each one of them can be in charge of the execution of one or more tasks. When there are more than one, this is called slot sharing. Hence, through this idea, one unique slot may hold an entire pipeline of the job. A diagram showing this idea is provided in FIGURE 3.

From a conceptual perspective, operators are schemes that control and transform the data flowing through the pipeline based on a provided algorithm. In the following, among the operators implemented by Flink, the most relevant ones for the proposed implementation are described.

- `map` generates a new stream by transforming elements one by one.
- `flatMap` produces a new stream by concatenating lists obtained from each one of the elements
- `keyBy` creates a *keyed stream*, i.e., it splits the stream into groups based on a provided key for each element.
- `reduce` takes a *keyed stream* or a *windowed stream* as input and aggregates the elements of each one of the groups/windows starting from the provided definition of how two elements are added.

Here, *windowed streams* are streams that are split into windows. Windows are chunks of the datastream that can overlap or not. They are called sliding windows or tumbling windows, respectively. This splitting can be made by event time (the real time where data was created), processing time, or simply the position within the stream. The last approach is indeed used within the pipeline proposed in subsection IV-B. They are called *count windows* and are generated through the `countWindow` operator.

On top of this design, Apache Flink is focused on stateful stream processing. For that, it uses what are called *rich functions*. *Rich functions* are abstractions offered by Flink's API that allow to define algorithms that depend on a given state, which can be updated in every computation. This enables operators to change their behavior along time, based on the

observed data. Moreover, when working with *keyed streams*, this state is local to the group being processed. This is known as *keyed state*. *Rich functions* can be accepted by the previously seen operators, producing stateful versions of them.

From a utility perspective, Apache Flink platform offers several functionalities that fit a wide range of streaming usecases. Some of these functionalities are offered through special operators different from the ones mentioned so far. These operators are discussed in the following.

For those usecases depending on the definition of iterations, such as the one presented in this paper, Flink offers interesting features. The `iterate` operator can contain an execution graph by itself in order to implement iteration steps. This is achieved through the creation of iteration source and iteration sink tasks connected by a feedback channel [34].

On the other hand, especially in incremental learning applications, it is very common to have two streams where one of the streams carries the rules to apply over the other. These rules can represent a machine learning model and they can evolve along time, so the implementation is not trivial. For these usecases, Flink provides a `connect` operator, which converts a pair of streams into so-called *connected streams*. Then, following operators applied over these connected streams have access to both streams at the same time, so they can keep updating the rules received and apply them to the other stream.

Finally, it is a common pattern to take a stream to produce several output streams. Usually, this is carried out through the definition of different operators. Everyone is executed isolately over the input stream and produces a different output. It may impact efficiency, since each one of these operators belongs to a different task, consuming therefore different resources. To tackle these cases, Flink provides a `process` operator that, apart from its main output, allows to define an arbitrary number of additional outputs. These additional outputs are called *side outputs*. In this way, it is possible to share resources, increasing efficiency.

B. SYSTEM IMPLEMENTATION

The ideas introduced in subsection IV-A have been used to translate the framework presented in section III into a real

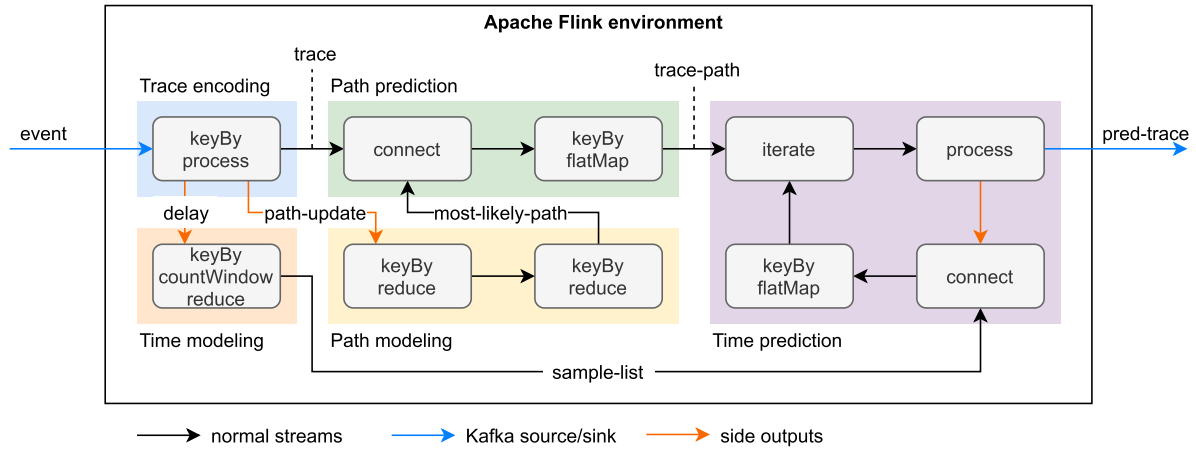


FIGURE 4. Implemented pipeline in Apache Flink to predict future event traces taking every new event as input.

system based on Apache Flink 1.13.0. This system has an input stream, ingested from Kafka, and an output stream, which is sent back to a different Kafka topic. The data pipeline implemented can be observed in FIGURE 4. This pipeline can be seen as the conjunction of five different high level stages. In the following, each one of the stages the pipeline is composed of is described as a mapping from the framework introduced in section III.

Trace encoding stage takes the system input stream, which is the global event stream, ϕ . This stage consists of the combination of two operators, a `keyBy` that hashes events by process instance identifier and a `process` operator that maintains in its *keyed state* the previous events of every case. Using these events, it creates three different streams: *trace* as the main output and *path-update* and *delay* as side outputs. The first one of these streams carries the updated trace for every observed case. Hence, for every input event ϕ_n , $c_n = \pi_c(\phi_n)$, the element generated for this stream is $\sigma_n(c_n)$. The *path-update* stream carries all updates to compute $P_n(s)$. Thus, for every ϕ_n , $c_n = \pi_c(\phi_n)$, and every state s , the `process` operator produces the update $sl(\sigma_n(c_n), s) - sl(\sigma_{n-1}(c_n), s)$. Finally, the *delay* stream is composed of the delay samples computed from the last received event and the previous one. For that, for every ϕ_n , let $c_n = \pi_c(\phi_n)$, $\sigma_n(c_n) = r$, and $r_i = \phi_n$, the new sample generated for the stream is $tm(r, i)$.

Path modeling stage is composed of two steps. First, a `keyBy` groups the observed paths that are identical and come from the same starting state. Then a `reduce` operator computes the number of times each path occurs after the given state, which corresponds to the cardinality of that path within path model $P_n(s)$. This model is then offered to the next step. Therefore, the second block receives a stream containing all observed paths after every state along with their number of occurrences. A `keyBy` is used to group the paths that have the same starting state. Then, a `reduce` operator is used again. This time, given two different paths following the same

state s , it selects the one with a greater number of occurrences. This way, the output stream of this stage consists of the most likely path for every state, which models the function $\omega_n(s)$.

Path prediction stage takes care to predict the paths expected in the future for the traces to continue down. In order to do this, a `connect` operator is used to make available for next operators the observed traces along with the most likely path after every state, i.e., $\omega_n(s)$ function. Then, `keyBy` operator groups together the most likely paths $\omega_n(s)$ for every state s with those traces, r in general, whose state $\rho(r)$ is identical, i.e., $s = \rho(r)$. Then, the next operator, a stateful `flatMap`, fills its keyed state with the last received most likely path. This produces no output, which is the reason why `flatMap` operator is used instead of a `map`. Meanwhile, the input traces received from the other stream are coupled with the most likely path forming pairs to build the output stream, *trace-path*.

Time modeling stage, meanwhile, uses the delay samples received to create lists with the last w samples to send them to the time prediction stage for every pair of states. It uses a `keyBy` to group together those samples that share the same previous state s_p and next state s_n . Then, a sliding `countWindow` operator selects the last w samples. Finally, a `reduce` operator takes all samples and makes a list of them, which is sent.

Time prediction stage receives *trace-path* pairs from path prediction stage and uses the delay sample lists from time modeling stage to predict the timestamps associated with every step within the predicted path. It does so iteratively until the path end is reached. This iterative processing is built through the use of an `iterate` operator that defines an iteration source which receives those pairs. For every iteration step, given a *trace-path* pair, with a trace r and a predicted path μ , the goal is to predict the delay between the previous state defined by the trace $s_p = \rho(r)$ and the next state, i.e., the first state in the path $s_n = \mu_1$. Thus, in order to execute a step, it must be ensured that $|\mu| > 0$, i.e., there are remaining states

to append to the trace. This condition may not be met even before the first iteration if the prediction is that the case has already ended. To ensure that, a `process` operator receives the pairs from the iteration source and checks that condition. Whenever a trace is completed, it is pulled apart through the main output of the operator. This output stream becomes the system output stream. The remaining pairs are sent through a side output. Then, in a similar way to the path prediction stage, a `connect` operator is used to manage trace-path pairs and delay sample lists in parallel by the same operators. Particularly, every pair whose delay from s_p to s_n has to be predicted has to be grouped along with the delay sample lists that model that transition, $\tau_n(s_p, s_n)$. This is achieved through a `keyBy` operator applied over the *connected streams*. Then, a `flatMap` operator retains the last sample list $tl^w(\tau_n(s_p, s_n))$ in its *keyed state* and uses it to predict the delay between s_p and s_n through an estimator dl . Once the next timestamp is predicted, the next event is created through $e = ev_n(r, s_n)$. This way, a new trace-path pair is created, where the new trace is $r \# \langle e \rangle$ and the new path is $tl^{|\mu|-1}(\mu)$. This new pair is ingested into the iteration sink to start a new cycle.

C. SCALABILITY CONSIDERATIONS

The system aims at enabling scalability for scenarios with a high rate of events per second. That is the main reason why the system is built on top of Apache Flink. By designing a framework that is able to be mapped onto a distributed processing platform, it is straightforward to allow the system to scale horizontally to support high throughput streams.

Particularly, this horizontal scalability is achieved by splitting the operators into parallel instances. Each one of these operator instances takes care of processing a subset of the samples handled by the operator. Besides, every operator instance can use at most all the resources provided by a task slot. Hence, as long as the samples can be split into an arbitrary number of subsets, the system can increase the number of total task slots in order to scale.

However, this is not always fulfilled. When a `keyBy` operator is followed by a stateful operator, the latter has access to the so-called *keyed state*, i.e., a state that is local to the group for a certain key. There is a single operator instance for every group keeping its corresponding local state. Hence, the number of operator instances processing the samples is limited by the number of keys generated by that `keyBy`. As a consequence, if a certain `keyBy` operator followed by a stateful operator splits a stream in a small number of groups, that operator can become a bottleneck.

The concatenation of a `keyBy` operator and a stateful operator appears in three different places along the pipeline shown in FIGURE 4.

- In the single block within trace encoding stage, keying is carried out casewise to maintain all previous events for a specific case in the state of the stateful `process` operator.

- Inside the path prediction stage, the second block uses a stateful `flatMap` that keeps track of the most likely path for every state. For that to be possible, the keying process creates a group for every possible state.
- Within the `keyBy-flatMap` block inside time prediction stage, groups are keyed considering every possible transition between states, i.e., state pairs that happen one after the other. This allows maintaining the last window of samples to make estimations from.

Among this keying objects, it is clear that cases are the less restrictive ones, since the number of case identifiers within the processes addressed by this work is always high.

On the other hand, as the number of possible states or transitions between states can be much lower, they can really involve a constraint for scalability. The number of states is usually lower than the number of transitions between states. However, the amount of computation for every predicted trace carried out by the `flatMap` operator in the time prediction stage is considerably larger since it holds the computation of k-nearest neighbors algorithm. Therefore, either of these two variables, i.e., the number of possible states or the number of possible transitions between states, may constraint the scalability of the system. It depends on the configuration used, e.g., the size of the window of samples processed by the time prediction stage, or the complexity of the state encoding scheme.

To avoid problems in this sense, it is convenient to properly select the encoding scheme. If the states are encoded more granularly, then the number of possible states increases and so does the number of possible transitions between states. This way, these variables are less likely to produce bottlenecks.

V. EVALUATION

The present section aims to prove the validity of the framework proposed. For that, a deployment has been defined to test the implementation upon a couple of case studies. This deployment is described in subsection V-A. Besides, subsection V-B defines the main metrics used to evaluate the results. Then, subsections V-C and V-D present the case studies and the results obtained. Particularly, they analyze the influence of different encoding schemes, i.e., different versions of the function ρ defined by (1). Several values of h and different cardinalities of G (different number of considered features), $n_f = |G|$, are tested. On the other hand, for time prediction, the average (*avg*) and the median (*med*) are used as defined by (13) and (14), respectively, for different values of k . Particularly, $k = \infty$ is used to refer cases where all samples are used to compute the estimated delay, i.e., there is no time awareness. Finally, subsection V-E discuss the tests carried out in order to check the capacity of the implementation to scale.

A. EVALUATION SETUP

In order to carry out the processing to obtain the results presented throughout this section, a setup has been defined

through docker-compose. This setup is mainly composed of two sections.

First of all, a section based on Kafka that acts as the interface of the system has been defined. This section is responsible to ingest messages into and pull the results out of the system. It is composed of two services: Kafka and Zookeeper, which is necessary for Kafka to work. An only instance of Zookeeper based on version 3.6.2 of its official image in Docker Hub has been used. Similarly, a unique Kafka broker is deployed using the image `bitnami/kafka:2.8.0` from Docker Hub.

The main section of the setup is the Flink's cluster itself. This cluster consists of a leader instance and four follower instances. Both of these kinds of instances are based on a custom image. The image has been built from the official image of the last stable version by the time this is written, `flink:1.13.0-scala_2.11` concretely. It is a multi-stage build where the first stage compiles the project inside the same environment that the application is going to run later on, and the second stage takes the generated `.jar` file and places it in `/opt/flink/usrlib/` folder. This way, the code is accessible for all instances. The leader runs by issuing the command `standalone-job`, i.e., the cluster works in application mode. Meanwhile, followers run by issuing the command `taskmanager`. Each one of the task managers are set up with two task slots and the default parallelism is set to 8.

Using the deployment through its Kafka interface, the results presented in this section have been obtained by sending the events from several datasets to the system and reading the predicted traces. These results have been stored in files in order to analyze them afterwards. All this process has been carried out using Python and its Kafka client.

B. EVALUATION METRICS

The present section aims to measure the quality of the predictions made by the described implementation. As the global event stream ϕ is simulated from unbounded datasets of length N , only the header $hd^N(\phi) = \langle e_1, e_2, \dots, e_N \rangle$ is considered known. Hence, for a given event $e_n \in \|\phi\|$, belonging to case $c = \pi_c(e_n)$, the complete trace associated with that case is assumed to be $r = \sigma_\infty(c) = \sigma_N(c)$. Let i be the position of the event within the trace, $r(i) = e_n$, if the event stream is casewise ordered, with the addition of that event, the first i events of the trace become known, i.e., $hd^i(r)$. This known part of the trace is used by the system to produce a prediction about the complete trace $p = \gamma_{n-1}(hd^i(r))$. Ideally, the predicted complete trace is equal to the real complete trace, i.e., $p = r$.

Of course, $hd^i(p) = hd^i(r)$. In order to assess the predictions, the tail following the last received event, e in this case, is what needs to be compared, i.e., $tl^{|p|-i}(p)$ and $tl^{|r|-i}(r)$.

Hence, the object of evaluation is a list of tuples, ξ , $|\xi| = N$. For every event e_n and the considerations made so

far, this list is defined as follows.

$$\begin{aligned}\xi &= \langle \xi_1, \xi_2, \xi_3, \dots \rangle \\ \xi_n &= (\overline{e_n}, tl^{|r|-i}(r), tl^{|p|-i}(p))\end{aligned}$$

For convenience, in the following, the elements in the tuples are referred to as (t, r, p) , but is important to note that here r and p state for the tail of the real and the predicted trace, not the whole trace. Besides, t is the time used as the starting point of the sequence of predicted times and is used later to normalize the delay errors.

Taking into account the above definition, metrics to address two different evaluations over that stream are needed. The two dimensions to evaluate correspond to the two steps the prediction model is composed of. Firstly, a metric to measure the quality of activity prediction is needed. Secondly, for those activities that were predicted successfully, a metric to measure the quality of the timestamps predicted is needed also.

In general, since time predictions are carried out iteratively once the sequence of activities has been predicted, if one of the predicted activities is wrong, then the following activities and its timestamps are not worthy for this evaluation, for coherence. This idea constraints the way metrics presented in this subsection are conceived.

Taking into account these considerations, an accuracy metric is defined to tackle the first of the two mentioned dimensions, since it consists of a classification problem. However, rather than the accuracy of the whole path predicted, this metric is parameterized to assess the accuracy of the header composed of an arbitrary number of events, which is referred to as the prediction offset, o .

Thus, given a set of tuples $W \subseteq \|\xi\|$, an accuracy metric depending on o is defined.

$$ac(W, o) = \frac{|\{(t, r, p) \in W \mid \pi_a(hd^o(r)) = \pi_a(hd^o(p))\}|}{|W|}$$

Here, the dividend states for the number of tuples (t, r, p) within W whose first o predicted activities are correct. Indeed, the accuracy with $o = \infty$, $ac(\infty)$ is used to refer to the global accuracy, treating the whole trace as a class.

Secondly, a metric to measure time prediction quality is also needed. For this, a function ae is defined to compute the absolute error yielded by the timestamp of the event o steps ahead of the prediction point. For a given tuple $(t, r, p) \in \|\xi\|$, it is defined as follows.

$$ae(r, p, o) = |\overline{r_o} - \overline{p_o}| \quad (15)$$

In the same way as with accuracy, a metric to measure the quality of the time prediction from a set of prediction tuples $W \subseteq \|\xi\|$ can be defined. Among these samples, only the valid ones to evaluate time prediction have to be considered. Particularly, only those that contain enough number of events to evaluate the prediction offset o have to be considered. Moreover, because of the way this framework is defined, time prediction depends on the path prediction accuracy. Hence, considering the time error between two events only makes

sense when the path prediction has been successful so far. Thus, only those samples whose activities have been successfully predicted o steps after the prediction point have to be considered. Given a window of samples W , the set containing the valid samples can be obtained as follows.

$$v(W, o) = \{(t, r, p) \in W \mid |r| \leq o \wedge |p| \leq o \wedge \pi_a(hd^o(r)) = \pi_a(hd^o(p))\} \quad (16)$$

From (15) and (16), a function to compute MAE (mean absolute error) in terms of W and the offset o is defined.

$$MAE(W, o) = \frac{1}{|v(W, o)|} \sum_{(t, r, p) \in v(W, o)} ae(r, p, o)$$

However, this approach to time evaluation has some counterparts. The delays between activities change considerably from one kind of activity to another. As a consequence, those kinds of activities that produce a bigger delay after them have a greater impact on MAE, since the ae is likely higher in those cases. To solve this in a unified way, with a metric that considers all kinds of activities, it is necessary to normalize the values. Given a prediction tuple (t, r, p) , the problem of predicting the timestamp of the event o steps ahead of the prediction point, $\overline{r(o)}$, is tackled by estimating the delay between the base time and that timestamp, $\overline{r(o)} - t$. Ideally, this coincides with the predicted delay, $\overline{p(o)} - t$. Therefore, one of these two delays might be used in order to normalize the time error. However, either approach presents problems. Normalizing the error using the former delay means that predicting a null delay yields a relative error equal to 1. Likewise, using the latter delay means that predicting an infinite delay yields a relative error equal to 1.

To avoid both distortions, the chosen option here is to use the minimum value between the two, so either predicting a null delay or an infinite delay results in an infinite relative error, and predicting a delay that doubles or is half the real one results in a relative error of 1, which may be considered a more balanced approach. Of course, the best definition to assess the error depends on the business needs, but for research purposes, the expression of the relative error re used in the rest of this paper is as follows.

$$re(t, r, p, o) = \frac{|\overline{r_o} - \overline{p_o}|}{\min\{\overline{r_o} - t, \overline{p_o} - t\}}$$

C. CASE STUDY A

With the aim of assessing the performance of the proposal, the implementation has been tested upon an event stream generated from a real event log of an incident management system. The management of this system is based on tickets. These are generated every time a user reports an incident and track the whole process until the incident is resolved. Hence, every ticket represents a process instance and every step in the ticket process is recorded as an event.

The process behind this system, as defined by the company, can be observed at FIGURE 5. As can be seen, six possible activities are associated with the events in the event log. After

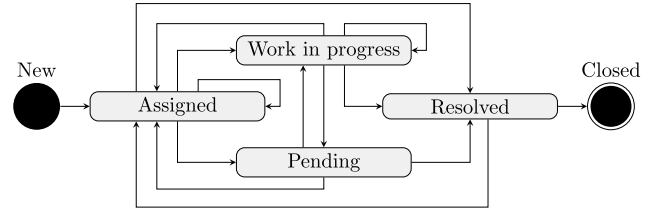


FIGURE 5. Process model of the incident management system under study.

a new case appears (*New*), it is assigned to a group (*Assigned*). As soon as the group confirms it can handle the case, it starts working on it (*Work in progress*) marking it as *Resolved* when it finishes. Finally, once the quality assurance team checks the validity of the solution, it is marked as *Closed*. Besides, if the user did not provide enough information to tackle the problem, it is marked as *Pending* until new details are received. Nevertheless, as seen, tickets can require reassignment if the group chosen cannot handle the case or the solution is not validated by the quality assurance team. Because of this, cyclic behavior is widespread in the dataset under study.

The event log recorded from the incident management system execution contains a total of 423954 events and records the traces of 95919 process instances. For every one of those events, the event log defines a case identifier, an activity (whose possible values were already explained), a timestamp, and a tuple of features. This tuple contains three fields named *isUrgent*, with possible values {yes,no}, *category*, with possible values {applications, infrastructure, IT services, user platform}, and *priority*, with possible values {P1,P2,P3,P4,P5}. This dataset required a cleaning process to erase duplicates or events with null timestamps. Furthermore, the dataset shows behavior diverging from that allowed by the graph in FIGURE 5. This can be observed in Table 3, where the most

TABLE 3. Trace count.

Count	Trace
30638	<Assigned, Resolved, Closed>
12618	<Assigned, Work in Progress, Resolved, Closed>
12449	<Assigned, Assigned, Resolved, Closed>
11035	<Assigned, Assigned, Work in Progress, Resolved, Closed>
6110	<Assigned, Resolved, Assigned, Resolved, Closed>
4306	<Assigned, Assigned, Assigned, Resolved, Closed>
2044	<Assigned, Closed>
1101	<Assigned, Assigned, Assigned, Assigned, Resolved, Closed>
...	
254	<Assigned, Work in Progress, Pending, Closed>
236	<Assigned, Work in Progress, Assigned, Assigned, Resolved, Closed>
230	<Assigned, Assigned, Work in Progress, Pending, Assigned, Resolved, Closed>
216	<New, Assigned, Assigned, Assigned, Resolved, Closed>
214	<Assigned, Assigned, Assigned, Assigned, Work in Progress, Resolved, Closed>
201	<Assigned, Assigned, Assigned, Assigned, Assigned, Assigned, Resolved, Closed>
178	<Assigned, New, Assigned, Assigned, Resolved, Closed>
...	

repeated trace patterns are shown jointly with the number of occurrences throughout the event log. The most notable difference is that almost all traces start from *Assigned* activity instead of *New*, as stated in the graph. Besides, although not visible in the table, where all traces end with activity *Closed* as it is supposed to be, there are cases that do not satisfy this rule. A 0.13% of the traces does not end properly, which corresponds to 124 traces. In Table 3, only 216 traces fulfill the ideal behavior, the trace pattern, (New, Assigned, Assigned, Assigned, Resolved, Closed). This highlights the need of a system capable of handling event traces with no previous knowledge about their morphology or special events such as start or finish events.

The delay between activities is influenced by time cycles. This is a common time behavior and it has been one of the main factors to define the way the prediction model is built. In this particular case, as observed in FIGURE 6, the time cycle to consider is a week. The delay between an event within a trace r_i and the next event r_{i+1} in that trace, $\Delta t = \bar{r}_{i+1} - \bar{r}_i$, is shown in terms of the time within the week for the first event $\hat{t}$, $t = \bar{r}_i$. The plot is the result of extracting 1000 random samples from the event log. It can be observed how the delay between an event and the next one changes along the week period. This is caused by the workweek schedule.

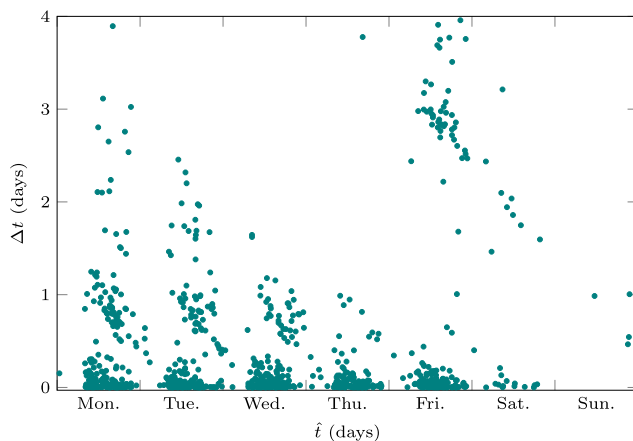


FIGURE 6. Distribution of delays Δt between events as a function of the time within the week $\hat{t}$ of the first one.

First of all, the activity accumulates over business days, whereas during weekends there is little activity, especially on Sunday. Moreover, the density is not clearly represented in this graph, but the events concentrate between 9:00 AM and 6:00 PM on weekdays and with a delay lower than ten hours. Indeed, this delay is lower at the end of the workday. This happens because if the management process of an incident updates within a workday, the following event is likely to happen within the same workday. That leaves a margin up to the end of the workday that decreases along the workday itself. Above these clusters of dots, there is a second set, especially noticeable from Monday to Wednesday, which relates to cases where the following event occurs the next

workday. Therefore, the delay is around one day at the start of the workday and decreases along the day because the start of the next workday is closer. Finally, the accumulation of samples with a delay of approximately three days over Friday especially draws attention. It means that when an event happens on that day, the next event is likely to occur three days later, i.e., on Monday.

This analysis of the data makes clear the importance of taking into account not only the time in terms of the day, but the time in terms of the week, in order to model the delay between events.

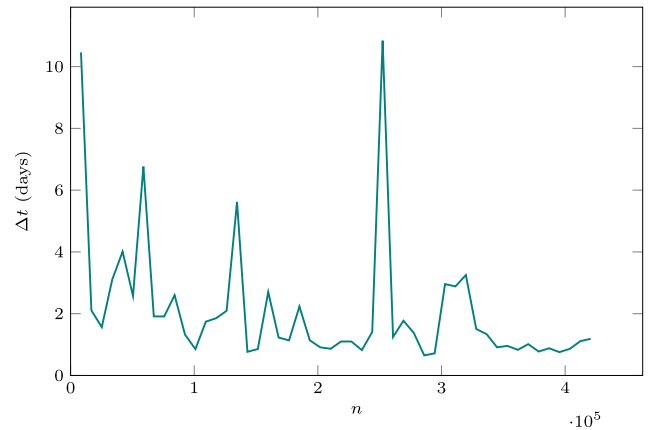


FIGURE 7. Evolution of $\text{avg}(\Delta t)$ over ingestion order n .

Finally, FIGURE 7 depicts the evolution of the average delay between events over the ingestion order n , i.e., over the order the events would be sent to the system. This allows for later comparison with the obtained results. As seen, the delay shows great fluctuations over time. A peak around $n = 2.5 \cdot 10^5$ is especially noticeable, but there are also another peaks throughout the entire sequence. There can be many reasons behind, such as vacation periods, service outages, etc. Even so, this kind of behavior can be found in real situations like the one presented and therefore needs to be handled. Indeed, the presence of this behavior in the case study allows for proving the robustness of the system against these kinds of scenarios.

As for the results obtained, first of all, FIGURE 8 shows the evolution of the accuracy in terms of the ingestion order n for different offset values o . To compute this evolution, the evaluation list ξ is split in sliding windows and the metric $ac(W, o)$ is applied over every window W . In general, it can be observed how the accuracy increases at the start of the processing of the input event stream ϕ from a cold start. On the other hand, the accuracy drops as the offset increases, since as the distance between the prediction point and the predicted event enlarges, so do the chances of failing at some prediction step. As most of the traces are made up of three events, once the first of them arrives, only two events to predict remain, and so the accuracy does not improve beyond $o = 2$.

Similarly, FIGURE 9 shows the evolution of the accuracy, but in this occasion for a fixed offset value $o = 2$

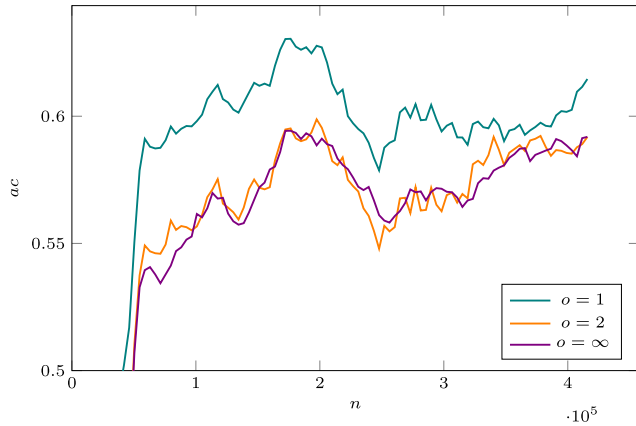


FIGURE 8. Evolution of accuracy $ac(W, o)$ upon sliding windows over ingestion order n for encoding scheme $(h, n_f) = (0, 0)$ and different values of offset o .

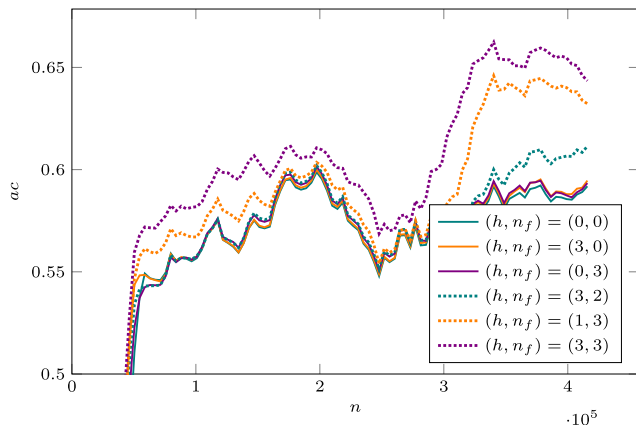


FIGURE 9. Evolution of accuracy $ac(W, o)$ upon sliding windows over ingestion order n for offset $o = 2$ and different encoding schemes (h, n_f) .

and different encoding schemes, i.e., different versions of the representation function ρ . Those have been obtained by setting different values of h and n_f . To start with, three simple encoding schemes are shown: $(h, n_f) = (0, 0)$, with no added information, $(h, n_f) = (3, 0)$, with memory of three previous activities but no features, and $(h, n_f) = (0, 3)$, considering the three features available but no memory to encode every state. The results obtained for these three encoding schemes are very similar, showing that no relevant information is added in terms of path prediction. However, several combinations of memory and features have proved to obtain better results. The encoding schemes $(h, n_f) = (3, 2)$ and especially $(h, n_f) = (1, 3)$ notably improve the accuracy about the end of the evolution. Furthermore, the encoding scheme $(h, n_f) = (3, 3)$, which is the most granular encoding under analysis, leads to even better results.

It should be noted that using more granular encoding schemes is the only way for this framework to improve the path prediction stage. However, encoding is useful beyond that. As seen later, they are also very useful to improve time prediction.

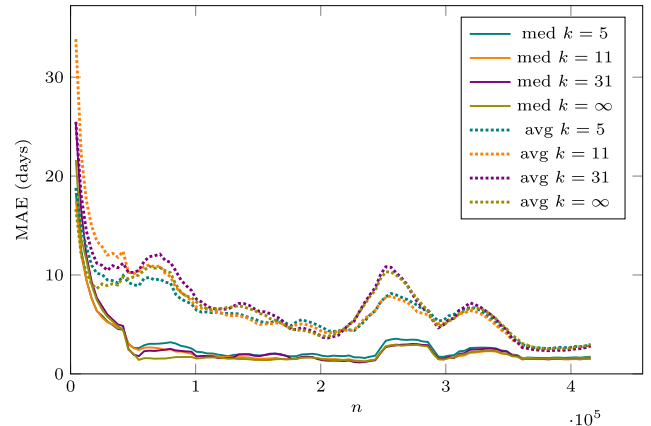


FIGURE 10. Evolution of MAE (W, o) upon sliding windows over ingestion order n for offset $o = 2$, encoding scheme $(h, n_f) = (0, 0)$, and different delay estimators.

Regarding time prediction, in FIGURE 10 the evolution of MAE over ingestion order n is shown. The windows used in order to compute MAE are the same as those used above. Particularly, the evolution of this metric is presented for different delay estimator dl versions. Results using average and median over k -nearest neighbors for different values of k are shown. As happened with accuracy, MAE values improve fast from the beginning. However, the results obtained using the average are considerably worse than the results obtained through the median. This is likely related to the inability of the average to filter noise. Indeed, around several points such as $n = 2.5 \cdot 10^5$, MAE performance deteriorates. This worsening is especially noticeable in the curves for the average estimator.

These results show the capacity of the system to keep delivering reasonable results over time despite large delay variations. This is possible thanks to the use of a windowing scheme over the delay sample stream. The use of a window of arbitrary size can prevent the system from indefinitely degrading its predictive model.

Besides the mean error, it is important to know how this error is distributed. The ability of an estimator to limit its error under a given threshold with a certain probability is a property of interest for many business needs. In this sense, FIGURE 11 shows the CDF (cumulative distribution function) over the absolute error ae obtained from a window W for an offset $o = 2$. The window considered discards the first 10^5 samples in order to ensure the initialization process is not taken into account. Thus, $W = tl^w(hd^N(\xi))$, where $w = N - 10^5$.

The curves shown in FIGURE 11 using this approach bring out several insights about the performance of the different estimators. These curves enable identifying how good are different estimators at predicting times with an absolute error lower than a certain threshold for ae . Hence, the faster the curve grows, the better the performance is. The most obvious insight is that median estimators notably outperform the results obtained using the average because of the noise filtering. This is the consequence of the effect observed previously

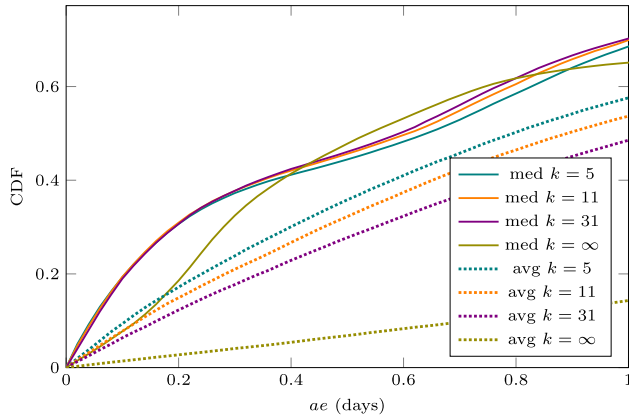


FIGURE 11. Cumulative distribution function of ae upon a window $W = tI^W(\xi)$ for offset $o = 2$, encoding scheme $(h, n_f) = (0, 0)$, and different delay estimators.

in FIGURE 10, where the average has shown a great vulnerability under the presence of noise.

Among the results obtained by using average, the influence of different k values becomes clear. The results for different average estimators improve as k decreases probably because closer and so more precise samples are used. However, this is not so clear for the case of the median. Although the curve obtained for $k = \infty$ grows more slowly than the others at first, it ends up showing better results around $ae = 0.5$.

It is important to note that the use of the average of all samples ($k = \infty$) is proposed by [32] to predict the remaining time until completion. This can be seen as a similar problem and serves as a baseline to illustrate the improvement offered by the rest of the proposed estimators.

Regardless, as explained before, the absolute error metric applied for all kinds of activities at once can show distorted results. Hence, to provide a unified view, in the following, the relative error metric is used instead.

FIGURE 12 presents the results for the same window of samples as before but using the relative error definition provided to compute the CDF.

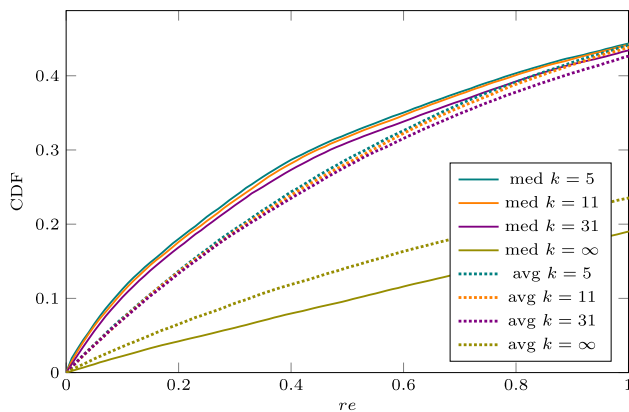


FIGURE 12. Cumulative distribution function of re upon a window $W = tI^W(\xi)$ for offset $o = 2$, encoding scheme $(h, n_f) = (0, 0)$, and different delay estimators.

Using this unbiased metric, the differences between the estimators become clear. In this case, the results obtained by time aware estimators, either average or median, outperform the results obtained for $k = \infty$, as expected. This highlights the importance of using the k -nearest time samples to the time being predicted from. Besides, as seen before, the median achieves better results than the average.

Taking again the relative error as metric, FIGURE 13 shows the CDF for different offset values o . This time, the median is used for $k = \{11, \infty\}$. It should be noted that, as they are relative values, it is perfectly feasible that results improve for greater offset values, as it is the case.

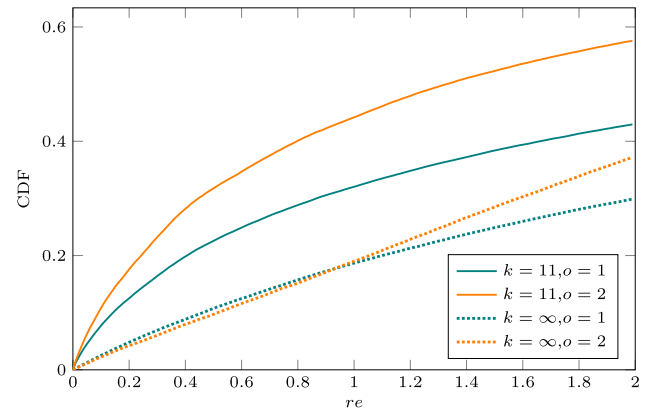


FIGURE 13. Cumulative distribution function of re upon a window $W = tI^W(\xi)$ for encoding scheme $(h, n_f) = (0, 0)$, median estimator and different values of k and offset o .

On the other hand, the contribution of different encoding schemes is analyzed. For that, FIGURE 14 shows the CDF obtained for different versions of ρ . Particularly, different combinations of the number of activities in memory, h , and the number of features considered, n_f , have been tested. All these results have been obtained using the median over the 11 nearest samples as the delay estimator. The worst results are obtained for $h = 0$ and $n_f = 0$, i.e., no previous activities

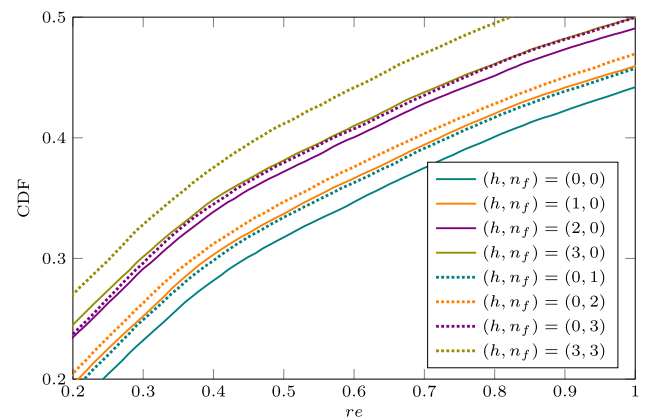


FIGURE 14. Cumulative distribution function of re upon a window $W = tI^W(\xi)$ for offset $o = 2$, delay estimator med with $k = 11$ and different encoding schemes (h, n_f) .

nor features are taken into account to encode the states used for prediction. As the number of activities or features considered increases, the results obtained are better. It makes sense since more precise encodings allow for handling different time behaviors isolated. Indeed, the best results are achieved by an encoding that takes into account the three previous activities and the three features, which is the most granular encoding under analysis.

The influence of different estimators, offset values, and encoding schemes over time prediction has been analyzed so far. However, the error threshold offered by different approaches is not the only characteristic to consider about the performance of the system. Depending on the business needs, it may be important that predictions fulfill certain time schedules, as real events do. For this particular case study, the time schedule consists of the workweek. As a consequence, real events have a much greater presence over the central hours of every business day.

Scenarios where there exist shared resources are good examples of the problems that may arise. For example, if the predicted events tend to extend evenly over the week duration, the likelihood of two events that need a certain resource colliding decreases. However, this behavior violates workweek schedules, therefore distorting further analysis based on these predictions.

In order to study this factor, FIGURE 15 shows the comparison between the histograms produced by time in terms of the week $\hat{t}$ from events predicted through different estimators and events generated by an ideal estimator with the same path accuracy but with no time errors. In this sense, the average, even for a small k value, shows a poor performance. It completely violates the schedules observed for real events, although the predicted activity peaks coincide with the real ones. On the other hand, the median for $k = \infty$ shows different results, being able to capture time cycles caused by days. Nevertheless, these cycles are shifted from the real

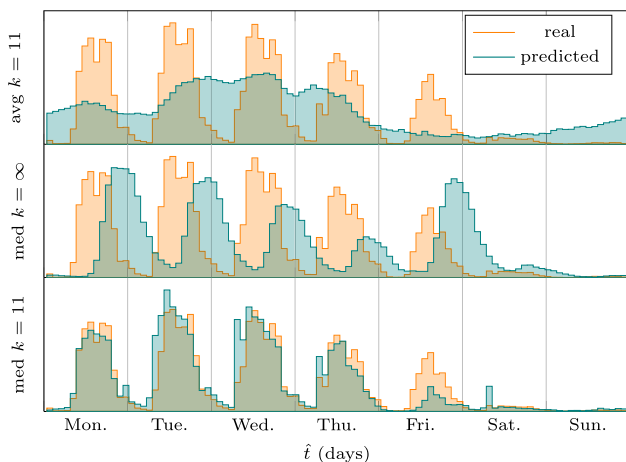


FIGURE 15. Histograms produced by predicted timestamps and their real values in terms of the week, $\hat{t}$, for offset $o = 2$, encoding scheme $(h, n_f) = (0, 0)$ and different delay estimators.

cycles, probably because of the time unawareness, so again the results produce a poor performance. Finally, the results obtained using the median with $k = 11$ further improves the two previous estimators. As the median is used, feasible delays are predicted and so it produces a cyclic histogram. Moreover, as it is time aware, these cycles are not shifted and coincide with those produced by real events. All these events have been generated with an offset $o = 2$ to bring out greater differences.

Finally, in FIGURE 16 the influence of different encoding schemes over histograms is brought out. Here, an offset $o = 2$ has been used again to obtain more clear differences. The first encoding scheme, as seen, is the most basic one, used for previous visualizations. Then, as an intermediate stage, an encoding considering only two previous activities is used, showing slightly better results. One of the most notable differences is the disappearance of the activity peak over Saturday produced by the most basic encoding. In the last place, the histogram produced by the encoding that showed better results in FIGURE 14 is placed. As expected, it presents slightly better results than the other two encodings, capturing more precisely the behavior over Friday.

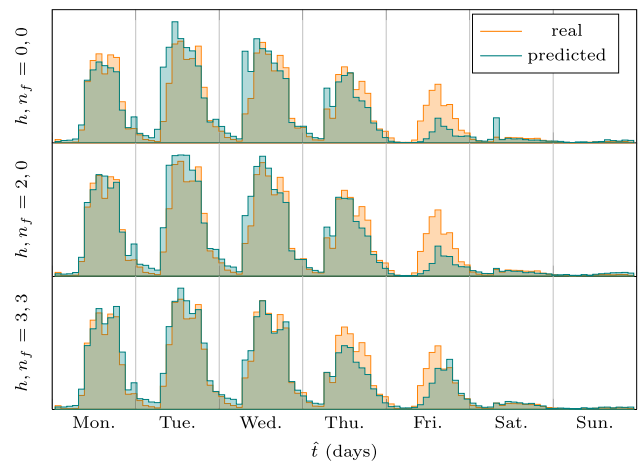


FIGURE 16. Histograms produced by predicted timestamps and their real values in terms of the week, $\hat{t}$, for offset $o = 2$, delay estimator *med* with $k = 11$ and different encoding schemes (h, n_f) .

D. CASE STUDY B

As a second case study, an event log from the business process intelligence challenge in 2014 [35] has been used. It contains the activity records related to an incident management system. It is composed of 46616 cases and a total of 466737 events. Its feature tuple for the experiments carried out is composed of an unique field, *AssignmentGroup*, which stands for the team responsible for the management of the incident. Hence, the possible values for n_f are 1 and 0 (no features considered).

Same as before, the evolution of the accuracy over ingestion order is presented in FIGURE 17. As observed, the level of accuracy is lower than in the previous case study. From

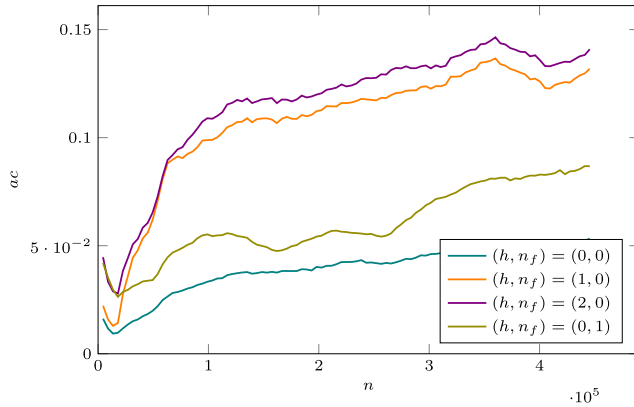


FIGURE 17. Evolution of accuracy $ac(W, o)$ upon sliding windows over ingestion order n for offset $o = 2$ and different encoding schemes (h, n_f) .

the perspective of a classification problem, this case accounts for a total of 22703 classes, one for every activity sequence pattern observed from the start of every trace. Among them, there are four classes that appear more than 1000 times, i.e., represent a fraction of the samples higher than 0.044. As can be observed, these values are similar to the accuracy levels presented. Among the codification schemes under analysis, the addition of memory with $h = 1$ and especially $h = 2$ produces the best results. For this reason, the encoding scheme $(h, n_f) = (2, 0)$ is used in the following to analyze the time prediction quality.

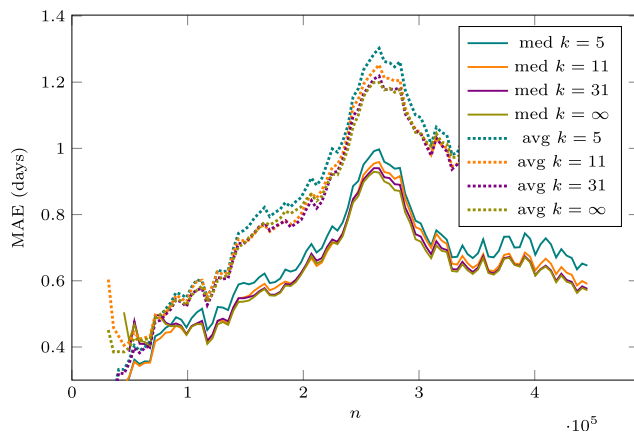


FIGURE 18. Evolution of MAE(W, o) upon sliding windows over ingestion order n for offset $o = 2$, encoding scheme $(h, n_f) = (2, 0)$, and different delay estimators.

FIGURE 18 shows the evolution of MAE for this case study. In this case, delays between activities are lower than in the previous case study. As a consequence, MAE values are also lower, although once again the results obtained using the median outperform those obtained through the use of the average as estimator.

This fact become clear in FIGURE 19, which shows the CDF of the absolute error ae for time prediction. Additionally, here the effect of the lower delays between events becomes clear. On the one hand, the error values are considerably

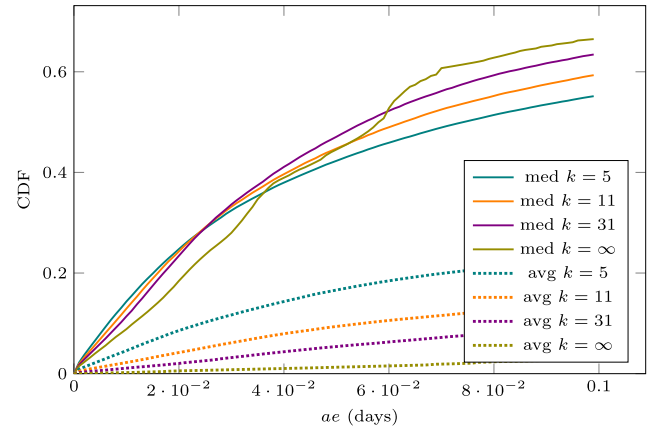


FIGURE 19. Cumulative distribution function of ae upon a window $W = t^W(\xi)$ for offset $o = 2$, encoding scheme $(h, n_f) = (2, 0)$, and different delay estimators.

lower than in the previous case study. On the other hand, the effect of the workweek schedule becomes unnoticeable from the perspective of the events, because the delays are negligible compared to the range of the workweek. Since k -nearest neighbors aims to model the behavior over this schedule, the improvement caused by its use is less obvious in this case. However, they are still better to predict times with the lowest error threshold. This effect is confirmed by FIGURE 20, which shows the CDF of the relative error for the same scenarios.

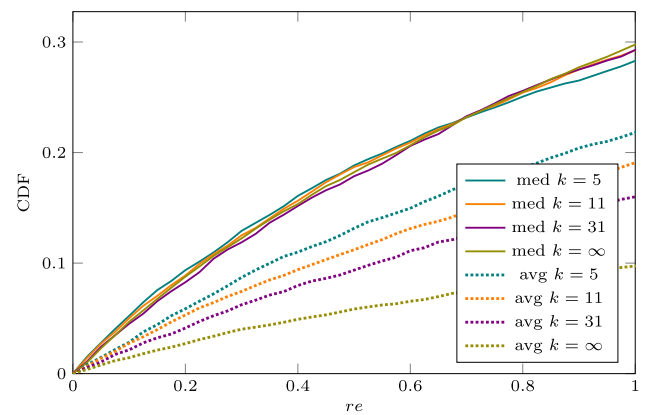


FIGURE 20. Cumulative distribution function of re upon a window $W = t^W(\xi)$ for offset $o = 2$, encoding scheme $(h, n_f) = (2, 0)$, and different delay estimators.

Additionally, FIGURE 21 depicts the CDF of the relative error for different encoding schemes. It proves a general improvement with the use of more sophisticated encoding schemes, although $(h, n_f) = (2, 0)$ shows worse results for lower error thresholds. In contrast, the encoding scheme $(h, n_f) = (0, 1)$, i.e., the consideration of the unique feature available, *AssignmentGroup*, produces the best results.

Finally, FIGURE 22 shows the histograms produced by real and predicted events. It proves the ability of the median to produce feasible results. However, in contrast to the previous

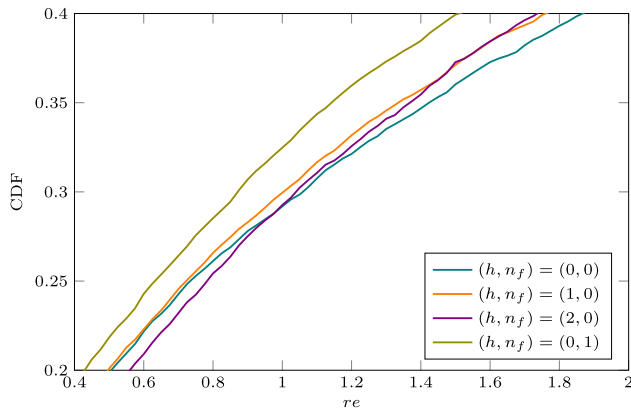


FIGURE 21. Cumulative distribution function of re upon a window $W = tI^W(\xi)$ for offset $o = 2$, delay estimator med with $k = 11$ and different encoding schemes (h, n_f) .

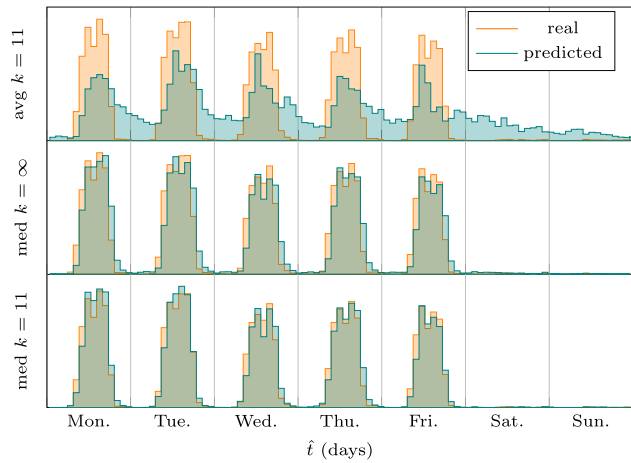


FIGURE 22. Histograms produced by predicted timestamps and their real values in terms of the week, t , for offset $o = 2$, encoding scheme $(h, n_f) = (2, 0)$ and different delay estimators.

case study, although the use of k -nearest neighbors shows a slight improvement, differences are minimal. As explained before, this is the consequence of the lower delays between events.

E. SCALABILITY TESTS

Finally, this subsection aims to check the ability of the proposed implementation to properly scale. In order to do this, several settings with the same scheme that the one used to obtain the results but with a variable number of followers have been tested. It has been deployed in a Docker Swarm cluster composed of eight machines, everyone with 32 GB RAM and an Intel Xeon E5-2630 v2. The maximum amount of CPU available for every follower container has been set to four cores, so adding each new instance involves adding a fixed amount of compute resources to the total capacity.

The encoding scheme used has been the most complex one under study, $(h, n_f) = (3, 3)$, and the estimator to compute

the delays has been the median of the k -nearest samples with $k = 11$.

To measure the number of events processed per second, the $4 \cdot 10^5$ first events of the dataset used in subsection V-C are ingested into the Kafka service. Once available, the Flink cluster is deployed with a given number of followers and processes the events from the beginning. The results are sent back to a different Kafka topic. This service, by default, adds a timestamp to every sample it receives with the current time. Therefore, once $4 \cdot 10^5$ predictions have been received, the difference between the timestamp of the first prediction and the timestamp of the last one enables computing the elapsed time and therefore the number of events per second.

Results in FIGURE 23 show how the processing capacity of the system scales as the number of followers grows. Of course, this increase is not proportional to the number of instances because of the overhead produced by the addition of new followers, e.g., the network usage is increased. Nevertheless, it is linear in the range tested.

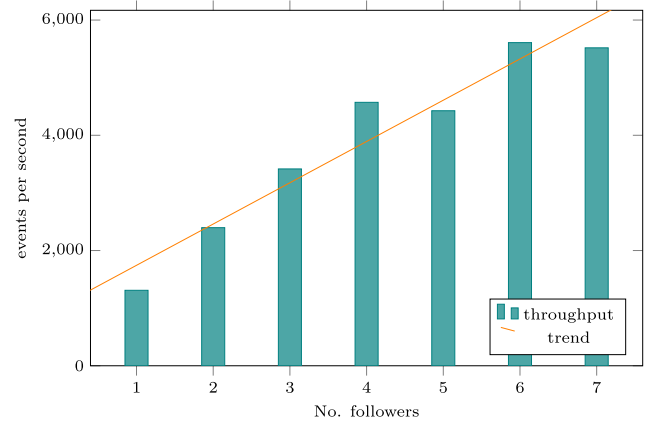


FIGURE 23. Number of events processed per second using a Flink cluster with a variable number of followers.

VI. CONCLUSION

The main contribution of this paper is the definition of a framework to predict the remaining sequence of activities and their timestamps for a given business process instance in an online and scalable setting. This framework is composed of two main stages, where the first one carries out the prediction of the remaining sequence of activities and the second one takes care of adding timestamps to these activities. Both stages, alongside the updating process of the predictive models, have been defined so they can be mapped to big data platforms. Moreover, an implementation of this framework based on Apache Flink has been tested upon two different case studies. That way, the capacity of the framework to scale and to successfully tackle event sequence prediction has been proved.

To conclude, there are several future lines to continue the work presented in this paper. Based on the proposed framework, more sophisticated estimators to predict the delay

between events should be studied. Particularly, estimators capable to detect concept drift, i.e., abrupt changes in the process behavior, would be of special interest. On the other hand, the prediction scheme proposed by this framework to make path forecasting highly relies on the state encoding scheme, because it simply predicts the most repeated path after every one of them. A more complex approach that takes event features into account but does not use these features to encode states could be of interest.

REFERENCES

- [1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, A. H. Byers, *Big Data: The Next frontier for Innovation, Competition, and Productivity*. New York, NY, USA: McKinsey Global Institute, 2011.
- [2] W. van der Aalst, "Process mining: Overview and opportunities," *ACM Trans. Manage. Inf. Syst.*, vol. 3, no. 2, pp. 1–17, Jul. 2012.
- [3] 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, and A. Burattin, "Process mining manifesto," in *Proc. Int. Conf. Bus. Process Manage.* Berlin, Germany: Springer, 2011, pp. 169–194.
- [4] W. Van Der Aalst, "Process mining," *Commun. ACM*, vol. 55, no. 8, pp. 76–83, 2012.
- [5] W. M. Van der Aalst, M. Pesic, and M. Song, "Beyond process mining: From the past to present and future," in *Proc. Int. Conf. Adv. Inf. Syst. Eng.* Berlin, Germany: Springer, 2010, pp. 38–52.
- [6] I. Teinemaa, M. Dumas, F. M. Maggi, and C. Di Francescomarino, "Predictive business process monitoring with structured and unstructured data," in *Proc. Int. Conf. Bus. Process Manage.* Cham, Switzerland: Springer, 2016, pp. 401–417.
- [7] D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal, and M.-C. Shan, "Business process intelligence," *Comput. Ind.*, vol. 53, no. 3, pp. 321–343, Apr. 2004.
- [8] A. E. Márquez-Chamorro, M. Resinas, and A. Ruiz-Cortés, "Predictive monitoring of business processes: A survey," *IEEE Trans. Services Comput.*, vol. 11, no. 6, pp. 962–977, Nov./Dec. 2018.
- [9] W. Fdhila, S. Rinderle-Ma, D. Knuplesch, and M. Reichert, "Change and compliance in collaborative processes," in *Proc. IEEE Int. Conf. Services Comput.*, Jun. 2015, pp. 162–169.
- [10] M. Maisenbacher and M. Weidlich, "Handling concept drift in predictive process monitoring," *SCC*, vol. 17, pp. 1–8, Jun. 2017.
- [11] J. Carmona and R. Gavalda, "Online techniques for dealing with concept drift in process mining," in *Proc. Int. Symp. Intell. Data Anal.* Berlin, Germany: Springer, 2012, pp. 90–102.
- [12] X. Liu, Y. Yang, D. Cao, and D. Yuan, "Selecting checkpoints along the time line: A novel temporal checkpoint selection strategy for monitoring a batch of parallel business processes," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 1281–1284.
- [13] *Apache Spark Project*. Accessed: May 28, 2021. [Online]. Available: <https://spark.apache.org/>
- [14] *Apache Flink Project*. Accessed: May 28, 2021. [Online]. Available: <https://flink.apache.org/>
- [15] M. Le, B. Gabrys, and D. Nauck, "A hybrid model for business process event prediction," in *Proc. Int. Conf. Innov. Techn. Appl. Artif. Intell.* London, U.K.: Springer, 2012, pp. 179–192.
- [16] M. Unuvur, G. T. Lakshmanan, and Y. N. Doganata, "Leveraging path information to generate predictions for parallel business processes," *Knowl. Inf. Syst.*, vol. 47, no. 2, pp. 433–461, May 2016.
- [17] G. T. Lakshmanan, D. Shamsi, Y. N. Doganata, M. Unuvur, and R. Khalaf, "A Markov prediction model for data-driven semi-structured business processes," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 97–126, Jan. 2015.
- [18] D. Breuker, P. Delfmann, M. Matzner, and J. Becker, "Designing and evaluating an interpretable predictive modeling technique for business processes," in *Proc. Int. Conf. Bus. Process Manage.* Cham, Switzerland: Springer, 2014, pp. 541–553.
- [19] D. Breuker, M. Matzner, P. Delfmann, and J. Becker, "Comprehensible predictive models for business processes," *MIS Quart.*, vol. 40, no. 4, pp. 1009–1034, Dec. 2016.
- [20] J. Becker, D. Breuker, P. Delfmann, and M. Matzner, "Designing and implementing a framework for event-based predictive modelling of business processes," in *Enterprise Modelling and Information Systems Architectures—EMISA*. Bonn, Germany: Gesellschaft für Informatik, 2014.
- [21] M. Ceci, P. F. Lanotte, F. Fumarola, D. P. Cavallo, and D. Malerba, "Completion time and next activity prediction of processes using sequential pattern mining," in *Proc. Int. Conf. Discovery Sci.* Cham, Switzerland: Springer, 2014, pp. 49–61.
- [22] I. Verenich, M. Dumas, M. La Rosa, F. M. Maggi, and C. Di Francescomarino, "Complex symbolic sequence clustering and multiple classifiers for predictive process monitoring," in *Proc. Int. Conf. Bus. Process Manage.* Cham, Switzerland: Springer, 2016, pp. 218–229.
- [23] N. Tax, I. Verenich, M. La Rosa, and M. Dumas, "Predictive business process monitoring with LSTM neural networks," in *Proc. Int. Conf. Adv. Inf. Syst. Eng.* Cham, Switzerland: Springer, 2017, pp. 477–492.
- [24] M. Camargo, M. Dumas, and O. González-Rojas, "Learning accurate LSTM models of business processes," in *Int. Conf. Bus. Process Manage.* Cham, Switzerland: Springer, 2019, pp. 286–302.
- [25] C. Di Francescomarino, C. Ghidini, F. M. Maggi, G. Petrucci, and A. Yeshchenko, "An eye into the future: Leveraging a-priori knowledge in predictive business process monitoring," in *Proc. Int. Conf. Bus. Process Manage.* Cham, Switzerland: Springer, 2017, pp. 252–268.
- [26] J. Evermann, J.-R. Rehse, and P. Fettke, "Predicting process behaviour using deep learning," *Decis. Support Syst.*, vol. 100, pp. 129–140, Aug. 2017.
- [27] M. Polato, A. Sperduti, A. Burattin, and M. de Leoni, "Time and activity sequence prediction of business process instances," *Computing*, vol. 100, no. 9, pp. 1005–1031, 2018.
- [28] S. Van Der Spoel, M. Van Keulen, and C. Amrit, "Process prediction in noisy data sets: A case study in a Dutch hospital," in *Proc. Int. Symp. Data-Driven Process Discovery Anal.* Berlin, Germany: Springer, 2012, pp. 60–83.
- [29] G. Bernard and P. Andritsos, "Accurate and transparent path prediction using process mining," in *Proc. Eur. Conf. Adv. Databases Inf. Syst.* Cham, Switzerland: Springer, 2019, pp. 235–250.
- [30] J. Evermann, "Scalable process discovery using map-reduce," *IEEE Trans. Services Comput.*, vol. 9, no. 3, pp. 469–481, May 2016.
- [31] A. Awad, M. Weidlich, and S. Sakr, "Process mining over unordered event streams," in *Proc. 2nd Int. Conf. Process Mining (ICPM)*, Oct. 2020, pp. 81–88.
- [32] W. M. P. van der Aalst, M. H. Schonenberg, and M. Song, "Time prediction based on process mining," *Inf. Syst.*, vol. 36, no. 2, pp. 450–475, Apr. 2011.
- [33] J.-P. Banâtre, P. Fradet, and Y. Radenac, "Generalised multisets for chemical programming," *Math. Struct. Comput. Sci.*, vol. 16, no. 4, pp. 557–580, 2006.
- [34] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, pp. 28–38, Jan. 2015.
- [35] B. F. van Dongen, "BPI challenge 2014," in *4TU.ResearchData*, Apr. 2014, doi: [10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35](https://doi.org/10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35).



PEDRO RICO received the B.S. degree in electronic systems engineering from the Universidad de Málaga (UMA), Spain, in 2019, and the M.S. degree in telecommunications engineering from the Universidad Politécnica de Madrid (UPM), Spain, in 2021. From 2018 to 2019, he was a Research Assistant at UMA. He is currently a Research Assistant at UPM. His research interests include process mining, distributed systems, and software engineering.



FÉLIX CUADRADO received the Ph.D. degree in telecommunications engineering from the Universidad Politécnica de Madrid (UPM), Spain, in 2009. He is currently a Senior Distinguished Fellow (Beatriz Galindo scheme) with the Universidad Politécnica de Madrid, a Visiting Reader at the Queen Mary University of London, and a fellow of the Alan Turing Institute. He has numerous publications in top tier journals and conferences, including IEEE TRANSACTIONS ON SOFTWARE

ENGINEERING, IEEE TRANSACTIONS ON CLOUD COMPUTING, Elsevier JSS, Elsevier FCGS, IEEE ICDCS, and WWW. His research explores the challenges arising from large-scale data-intensive applications through a combination of software engineering, distributed systems, and mathematical approaches.



JAVIER ANDIÓN (Member, IEEE) received the Ph.D. degree (*cum laude*) in telecommunications engineering from the Universidad Politécnica de Madrid (UPM), Spain, in February 2021. He worked as a software developer in several startups or research groups. Since 2015, he has been part of the research group Real-Time Systems and Architecture of Telematic Services (STRAST) at UPM, where he has been as an Assistant Professor, since 2020. He is the author or coauthor of four

scientific publications in high-impact journals and two registered software for the management of failures in computer networks.



JUAN C. DUEÑAS (Senior Member, IEEE) received the Ph.D. degree in telecommunications engineering from the Universidad Politécnica de Madrid (UPM), Spain, in 1994. He is currently a Professor at the UPM's Escuela Técnica Superior de Ingenieros de Telecomunicación and the Deputy Vice-President of research at UPM. His research interests include services engineering, internet services, service oriented architectures, and software engineering.



HUGO A. PARADA G. received the Ph.D. degree in telecommunications engineering from the Universidad Politécnica de Madrid (UPM), Spain, in 2010. Since 2015, he has been an Assistant Professor with the Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación, UPM. His current areas of research interests include machine learning, big data, and cloud computing.

...