μXL: Explainable Lead Generation with Microservices and Hypothetical Answers

Luís Cruz-Filipe[†], Sofia Kostopoulou[†], Fabrizio Montesi[†], Jonas Vistrup[†]

Dept. Mathematics and Computer Science, Univ. Southern Denmark, Campusvej 55, Odense, 5230, Denmark.

*Corresponding author(s). E-mail(s): vistrup@imada.sdu.dk;
Contributing authors: lcf@imada.sdu.dk; skos@sdu.dk; fmontesi@imada.sdu.dk;
[†]These authors contributed equally to this work.

Abstract

Lead generation refers to the identification of potential topics (the 'leads') of importance for journalists to report on. In this article we present μXL, a new lead generation tool based on a microservice architecture that includes a component of explainable AI. μXL collects and stores historical and real-time data from web sources, like Google Trends, and generates current and future leads.
Leads are produced by a novel engine for hypothetical reasoning based on temporal logical rules, which can identify propositions that may hold depending on the outcomes of future events. This engine also supports additional features that are relevant for lead generation, such as user-defined predicates (allowing useful custom atomic propositions to be defined as Java functions) and negation (needed to specify and reason about leads characterized by the absence of specific properties).
Our microservice architecture is designed using state-of-the-art methods and tools for API design and implementation, namely API patterns and the Jolie programming language. Thus, our development provides an additional validation of their usefulness in a new application domain (journalism).
We also carry out an empirical evaluation of our tool.

Keywords: Lead generation, Microservices, Explainable AI, Jolie

1 Introduction

Journalists at news media organisations can regularly come across a plethora of available information and events from various online data sources, including social media. Therefore, it is of great significance to explore automated procedures that can support journalists in dealing efficiently with such continuous streams of real-time data. This explains why AI in journalism, or automated/computational journalism, has been intensely studied in the last years.

In this article, we are interested in automated support for *lead generation*, that is, supporting journalists with useful information about what they could report on. Lead generation is connected to trend detection and prediction. Trending topic detection is a problem that has been researched extensively for the specific application domain [1, 2]. In another line of research, there are several works that try to predict trending topics, news, or users' interest in advance. For instance, the authors in [3] aim to predict trending keywords, the work in [4] targets forecasting article popularity, and [5] focuses on the prediction of future users' interests. Automated news generation is another field of research that received much attention by researchers. The authors in [6] present an architecture for automated journalism and in [7] they propose an automatic news generation solution by integrating audio, video, and text information. All the aforementioned works, even though they are closely related, do not tackle the challenging problem of alerting journalists about imminent leads for potential future articles. In this direction, the 'Lead Locator' tool [8] suggests locations relevant to political interest and produces 'tip sheets' for reporters.

Motivation

Our motivation for this work stems from a collaboration with media companies in Denmark,[1] which elicited a number of requirements that are not met by current solutions for lead generation. The first requirement is explainability: the system should present its reasoning for the suggestions that it brings forward, such that the journalist can apply their own intuition as to how promising a lead is. (In general, explanations can be crucial in guiding journalists towards valuable reporting decisions.) The second requirement is flexibility: the system should be designed with extensibility in mind, in particular regarding the future additions of new data sources and processors. The third requirement is reusability: the system should expose its operations through well-defined service APIs, such that it can be integrated in different contexts.

Meeting these requirements is challenging because it requires designing a loosely-coupled system that accumulates different kinds of data. Also, to the best of our knowledge, there are no reasoning tools available for deriving and explaining potentially-interesting scenarios (the leads) from online data streams.

This work

We present μXL, a new lead generation tool that meets the aforementioned requirements thanks to two key aspects.

[1] https://www.mediacityodense.dk/en/

First, μXL is implemented as a microservice architecture. Components are clearly separated and can interact purely by formally-defined APIs. These APIs are defined in the Jolie programming language [9], whose API language is designed to be technology agnostic: Jolie APIs allow only for semi-structured data with widely-available basic values (strings, integers, etc.) and can be implemented with different technologies [10]. Most of our microservices are written in Jolie, but we leverage this flexibility to use Java in our most performance-critical component. In particular, we can use Jolie to lift a simple Java class to a microservice without requiring any additional API definitions or Java libraries.

Second, μXL includes the first implementation of the recent theory of hypothetical answers to continuous queries over data streams [11]. This theory allows for concluding facts (the answers) before all the necessary premises appear in the data, assuming that these missing pieces can still occur later in time. Such *hypothetical* answers are then equipped with clear indications of which premises have already been fulfilled and which are still missing (but might occur in the future). We leverage this idea to provide users with explanations of (i) what has already been confirmed to support a lead and (ii) what needs to happen in the future for the lead to become certain – this is our concept of 'explainable AI' in this article.

Contribution

Our contributions can be summarised as follows:

- a microservice architecture that (i) collects historical and current data relevant for lead generation from various online data sources, and (ii) integrates symbolic AI in the form of a reasoner to generate explainable leads;
- a technology-agnostic description of the APIs used in our system, expressed in Jolie [9], as well as of the underlying API patterns, following [12];
- the first implementation of a reasoning engine producing hypothetical answers – indicating topics that might trend in the near future, with an explanation of why this is the case – and its integration in our architecture;
- an empirical evaluation of our development, which includes a performance evaluation of the lead generation component (the reasoner) and a discussion of our experience with implementing the system in Jolie.

Describing our architecture using Jolie and API patters serves three purposes: (i) for us, these tools were useful guides; (ii) for the reader, this option clarifies our design; and (iii) for Jolie and the collection of API patterns, this serves as additional validation of their usefulness in practice. To the best of our knowledge, this work is also the first validation of both Jolie and API patterns in the journalistic domain.

Our reasoning engine is based on the theory originally presented in [11], so our work also serves as the first experimental validation of its usefulness.

Publication history

This article is the extended version of a previous conference publication [13].

Our reasoning engine has been upgraded: it now supports negation in formulas, and explanations include also the rule applications used in their derivations. The system has also been optimised by better exploiting its database: incoming data is now processed in chunks, allowing us to process much larger amounts of data than previously.

We have also improved the presentation in several ways. Section 2 has been extended with related work in both predictive marketing and API Patterns. Background information on hypothetical reasoning has been expanded and moved to Section 3.1. The new Section 5 discusses practical perspectives on how to use the data output by our reasoning engine, like how to interpret it and some ideas that came out of interacting with early adopters of the tool, and our experience with using Jolie as an architecture implementation tool. We have also expanded our performance evaluation with more details on our setup and new observations on processing data in chunks in Section 6.

Structure

Section 2 presents relevant related work. Section 3 provide background on the reasoning theory that we use and describes our explainable AI engine. Section 4 is dedicated to the system's architecture and our use of explanations from the AI engine. Section 5 reports on our experience with implementing the architecture with the Jolie programming language. Section 6 provides our experimental evaluation. Finally, Section 7 concludes with future work.

2 Related Work

In this section we summarise the works that are most relevant and/or closely related to our development.

AI and Journalism

The use of AI in journalism is seeing increased focus, with the aim of making the journalists' work more efficient.

One of the perspectives that is relevant to this work is automated news generation. In this realm, an early approach [7] proposed an automatic news generation solution with semantically meaningful content by integrating audio, video, and text information in the context of broadcast news. More recently, an automatic news generation system [6] was proposed that is largely language and domain independent.

A practical tool based on similar principles is *News robot* [14], which could automatically generate live events or news of the 2018 Winter Olympic Games. News robot generated six news types by joining general and individualised content with a combination of text, image, and sound.

Another perspective is trending topic detection, where programs try to distinguish trending topics in a wealth of news sources. In that context, *TwitterMonitor* [2] is a tool that detects trends in online Twitter data and synthesises a topic description. The system also offers user interactivity for selection by means of criteria-selection and topic description.

Topic detection on Twitter was also studied in [1], which compared six methods used for detecting major events, with different time spans and churn rates. These authors also developed four new approaches to this problem, showing that they could outperform state-of the-art methods.

An alternative approach [15] designed a novel framework to collect messages related to a specific organisation by monitoring microblog content, like users and keywords, as well as their temporal sequence.

Another noteworthy tool is *CityBeat* [16], which looks for potential news events by collecting geo-tagged information in real-time from social media, finding important stories, and making an editorial choice on whether these events are newsworthy.

The state-of-the-art in lead generation [8] is *Lead Locator*, a news discovery tool that supplements the reporting of national politics by suggesting possibly noteworthy locations to write a story about. In this work, the authors analysed a national voter file using data mining to rank counties with respect to their possible newsworthiness to reporters, from which they automatically produced 'tip sheets' using natural language generation. Reporters have access to these tip sheets through an interactive interface, which includes information on why they should write an article based on that county.

There is also a line of research dedicated to predicting trends using machine learning techniques. One such approach [3] tackled trending topic prediction as a classification problem, using online Twitter data to detect features that are distinguished as trending/non-trending hashtags, and developed classifiers using these features. These classifiers were trained on trending/non-trending hashtags that were posted on Twitter on a specific day, and were used to classify hashtags as trending or non-trending the next day.

Another solution [4] extracts keywords from an article and then predicts its popularity based on these keywords, similarly to popular approaches based on the BERT model and text embeddings.

Predicting future user interests was also explored in the context of microblogging services and unobserved topics [5]. This work showed how to built topic profiles for users based on discrete time intervals, and then transfer user interests to the Wikipedia category structure. In general, predicting user interests is connected to the general topic of predictive marketing [17]. Solutions in this domain are typically based on big data and analytical methods. Our approach, instead, focuses on rule-based methods that can (i) represent and deduce new knowledge, and (ii) offer explanations.

Our explainable AI component in μXL implements a theoretical framework for hypothetical reasoning over continuous datastreams [11], which allows for producing answers that depend on the occurrence of future events. It is the first time that this theory is implemented and used, and thus it is also the first time that it is used in the journalistic context.

Microservices, API Patterns, and Service-Oriented Languages

Microservices are cohesive and independently-executable software applications that interact by message passing. Their origins and reasons for diffusion are surveyed in [18], along with open challenges and future directions. The design of our microservice architecture is based on the recent catalogue of patterns for API design presented in [12]. The catalogue tackles the challenges of microservice and remote API design [19] through peer-reviewed patterns published in the EuroPLoP proceedings between 2017 and 2020 [20–24].

Several languages and tools have been proposed to facilitate the implementation of microservice architectures, like Ballerina [25], Jolie [9], LEMMA [26], and MDSL [12]. We chose Jolie among these because it features an integrated language for programming technology-agnostic APIs – in terms of primitives that have already been used as target for compiling API patterns [12] – and API implementations – in terms of elegant, process algebraic primitives for structuring communications [9, 10, 27–29]. More concretely, there are two aspects of Jolie that are especially relevant for the current work. First, Jolie's algebraic language for composing communication actions facilitates the composition of services by other services. Second, in the definition of services, Jolie's syntax separates APIs, deployment, access points (how APIs can be reached, e.g., with which protocol), and behaviours (service implementations).

Some notable consequences for our work include: (i) Jolie APIs can be implemented with different technologies (we use Jolie itself for some, and Java when fine-tuning performance is important); and (ii) the different parts of our architecture can be flexibly deployed together (communication supported by shared memory), all separate (remote communication), or in a hybrid fashion (some together, some not). Our description assumes all-separate deployment, but adopters are free to change this decision.

Jolie has a vibrant research community [30] that validated the language in several ways and different real-world domains [31–35]. To the best of our knowledge, our application of Jolie is novel in several ways: it implements a microservice architecture designed from scratch with API patterns; it deals with the journalistic domain; and it integrates a reasoning engine component based on hypothetical answers. Previous evaluations of Jolie encompass increasing development productivity [31], the development of security strategies [36], and engineering: Jolie's structures resemble the architectural metamodels found in tools for Model-Driven Engineering of microservices based on Domain-Driven Design [37], which have been tested in the electromobility domain [38].

3 HARP: Hypothetical Answer Reasoning Program

μXL combines several microservices to achieve the global goal of generating leads with explanations. The core AI of the system is the Hypothetical Answer Reasoning Program (HARP), which we describe in this section. The rationale for describing HARP before giving an overview of the whole system is that the design of the APIs for the different microservices is motivated by how the reasoner works.

HARP contains an implementation of the reasoning theory of [11] to perform lead deduction from a program and a datastream. This architecture allows for an arbitrary datastream and an almost arbitrary specification of rules.[2] The core functionalities of HARP are implemented in Java; the resulting microservice and APIs are in Jolie, which wraps the Java code by using Jolie's embedding feature for foreign code [9, 10].

[2]To ensure termination, a program cannot include circular dependency between its rules.

3.1 Theoretical Background

We first review the theoretical background behind our implementation.

The language underlying the theoretical framework that we use in our reasoning engine, from [11], is *Temporal Datalog* [39]: a negation-free variant of Datalog where predicates include temporal attributes. There are two types of terms in Temporal Datalog. An *object* term is either an object (constant) or an object variable. A *temporal* term is either a natural number, called a *time point* (one time point for each natural number), a *time variable*, or an expression on the form $T+k$ where T is a time variable and k is an integer.

Predicates take exactly one temporal parameter, which is always the last one. This gives all atomic formulas, hereafter called atoms, the form $P(t_1,\dots,t_n,\tau)$, where P is a name of an $(n+1)$-ary predicate, t_1 to t_n are all object terms, and τ is a time term. The intuitive semantics of $P(t_1,\dots,t_n,\tau)$ is that predicate P holds for terms t_1 to t_n at time τ.

Programs are sets of rules of the form $\alpha \leftarrow \alpha_1 \land \dots \land \alpha_n$ with α, α_1, …, α_n atoms. The *head* of the rule is α, and the *body* of the rule is $\alpha_1 \land \dots \land \alpha_n$. A predicate that occurs in the head of at least one rule with non-empty body is an *intensional* predicate, otherwise it is an *extensional* predicate.

A *dataslice* consists of a time point τ and a finite set of atoms with extensional predicates, which all hold at time τ, representing representing the information available at the given time. A *datastream* is a set of *dataslices*, one for each possible time point. A *query* is a list of atoms, and an *answer* to a query is a substitution that makes the query valid, i.e., a logical consequence of the program and the currently available prefix of the datastream.

Given a Temporal Datalog program and the current prefix of a datastream, *hypothetical answers* can be computed for a given query. A hypothetical answer is a substitution σ paired with a set of atoms H (the *hypotheses*) such that σ is an answer to the query if the atoms in H appear later in the datastream. The algorithm from [11] is a modified version of SLD-resolution [40] from logic programming.

SLD-resolution computes its answers using SLD-derivations, which work on *goals* – formulas of the form $\leftarrow \alpha_1 \land \dots \land \alpha_n$. SLD-derivations are build from *resolution steps*, where one atom in the body of the current goal G is unified with the head of a rule r in the program. This creates a new goal G' whose body is obtained by replacing the selected atom in G with the body of r. (In particular, the selected atom is simply removed if the body of r is empty.) An SLD-derivation is a finite sequence of goals $G_0,\dots,G_n$ where (i) G_0 is the singleton goal containing the query in its body and (ii) G_{i+1} is obtained from G_i by resolution. The derivation also computes a substitution, which is the composition of the substitutions computed during unification at each resolution step restricted to the variables in the query.

A framework for allowing negated atoms in the body of rules is also formulated in [11]. This framework relies on the idea of negation as failure [41]. Negation as failure permits the deduction of a negated atom, $\neg\alpha$, iff no SLD-derivation of α can be made. In our context, a negated atom $\neg\alpha$ with time point τ is deduced iff there exists no SLD-resolution of α with time point τ given the rules and all possible datastream consistent with the currently available prefix.

The algorithm in [11] computes all SLD-derivations from the given query that end in goals whose body only contains atoms built from extensional predicates, storing hypothetical answers consisting of the computed substitution together, the final goal, and an empty list of *evidence*. These hypothetical answers are then stored and updated continuously. Whenever a new dataslice arrives, the set of hypotheses in each hypothetical answer from the previous iteration is compared to the dataslice. Hypotheses with the current time that unify with atoms in the dataslice are moved to the list of evidence; if there are hypotheses containing atoms with the current time that do not unify with any atoms in the dataslice, then the associated hypothetical answer is discarded. Negative hypotheses are processed afterwards: if there is no hypothetical answer that matches them, then they can never become true, and can also be moved to evidence. Finally, hypothetical answers containing negative hypotheses for which there is an answer (without hypotheses) are discarded.

3.2 Specification of Rules

We illustrate the specification of rules in our implementation by using streams of data that originate from Google Trends. The implementation follows most of the standard conventions for Datalog – variables and predicate names start with an uppercase letter, while constants start with a lowercase letter. Time points must be natural numbers and expressions must be of the form $T+k$ or $T-k$ for a time variable T and a natural number k. In our examples, time points represent hours, and timestamps from Google Trends are rounded up to the next hour.

Our program covers three arguments for why a topic should be considered a lead and one arguments for when a lead is local.

1. If a topic becomes a daily trend in a region and its popularity rises over the next two hours, then it is a popularity lead. The corresponding rule is written as:

```
DailyTrend(Topic, Region, T), Popularity(Topic, Region, Pop0, T),
Popularity(Topic, Region, Pop1, T+1),
Popularity(Topic, Region, Pop2, T+2),
Less(Pop0,Pop1), Less(Pop1,Pop2) -> PopularityLead(Topic, Region, T)
```

2. If a topic is a daily trend in a region, and then becomes a daily trend in another region, then it is a global trend – but only if it continues to spread to new regions every hour for the next two hours. If a global trend remains a global trend for the next two hours, then it is a global lead. This argument is written as two rules: one rule specifying what makes a topic a global trend,

```
DailyTrend(Topic, Region0, T), DailyTrend(Topic, Region1, T+1),
DailyTrend(Topic, Region2, T+2), DailyTrend(Topic, Region3, T+3),
AllDiff(Region0,Region1,Region2,Region3) -> GlobalTrend(Topic, T+1)
```

and one rule specifying how a global trend becomes a global lead.

```
GlobalTrend(Topic, T), GlobalTrend(Topic, T+1),
GlobalTrend(Topic, T+2) -> GlobalLead(Topic, T)
```

3. The third argument uses the notion of *certain leads* – some leads are more certain than others. While our architecture does not capture probabilities, we can specify that both popularity leads and global leads are certain leads.

   ```
   PopularityLead(Topic, Region, T) -> CertainLead(Topic, Region, T)
   GlobalLead(Topic, T) -> CertainLead(Topic, Region, T)
   ```

   If the topics of two certain leads both closely relates to a third topic, then the third topic is a lead, which is derived from other leads:[3]

   ```
   CertainLead(Topic1, Region, T), CertainLead(Topic2, Region, T),
   RelatedTopic(Topic1, Topic), RelatedTopic(Topic2, Topic)
   Diff(Topic1,Topic2)  -> DerivedLead(Topic, Region, T)
   ```

   We differentiate between certain leads and derived leads to avoid derived leads being recursively used to derive other leads. If we allowed derived leads to be used like that, then we would risk generating leads that are very far from the actual data and unlikely to be useful.
4. The next rules denote that both certain leads and derived leads are leads.

   ```
   CertainLead(Topic, Region, T) -> Lead(Topic, Region, T)
   DerivedLead(Topic, Region, T) -> Lead(Topic, Region, T)
   ```

Some arguments relies on the failure of others. For instance, for a topic to be a 'local' lead, it is necessary to prove that the lead is not a global lead. This is an illustrative example of the usefulness of negation: proving that something is false by using only positive observations is cumbersome, since it requires observing that every possible way it can be proven does not occur. This means that adding new rules might invalidate proofs of falsity. Furthermore, it can be computationally intensive: since a topic can become a global lead using any four different regions for n unique regions, then there are more than $\binom{n}{4}$ different ways to prove that a topic is a global lead. Even if regions only denote countries, the body of the rule deducing that a topic is not a global lead would need to contain more than $\binom{195}{4} = 58\,409\,520$ atoms, assuming 195 countries. Instead, using negation as failure (represented as $\sim$) we can define a local lead directly as a lead that is not a global lead, as in the next rule.

```
 Lead(Topic, Region, T), ~GlobalLead(Topic, T)
   -> LocalLead(Topic, Region, T)
```

At each hour, the datastream contains information about topics that are daily trends in a region, `DailyTrend(Topic, Region, T)`, popularity of topics that are daily trends, `Popularity(Topic, Region, Pop, T)`, and which topics are related, `RelatedTopic(Topic1, Topic2, T)`.

3.3 User-Defined Predicates (UDPs)

Predicates such as `Less` and `AllDiff` are not practical to specify using rules, but rather algorithmically. Our implementation allows for specifying such *User-Defined*

[3]This rule captures the idea that if Peyton Manning and Tom Brady are both in the news, then it might be interesting to write an article about NFL Quarterbacks.

Predicates (UDP). An atom whose predicate is a UDP is called a *User-Defined Atom* (UDA). UDAs in the hypotheses are evaluated by running the function defining the corresponding UDP as soon as all variables have been instantiated, after which they are processed similar to other hypotheses. Therefore, all uses of UDAs in rules must be *safe*: any variable that appears in a UDA in a rule must also appear in a non-UDA in the same rule.

UDPs are specified by implementing the Java Interface `UserDefinedPredicate`, whose local path is given in the internal initialisation of HARP. Therefore, adding different UDPs requires updating a configuration file within HARP. The interface `UserDefinedPredicate` includes four methods:

- `id()` returns the textual representation of the UDP;
- `toString(List<Term> terms)` returns the textual representation of a UDA with this UDP and arguments `terms`;
- `nArgs()` returns the number of arguments of the UDP;
- `run(List<Constant> constantList)` returns **true** if the UDP holds for the list of objects (constants) arguments given, **false** otherwise.

The list arguments of both `toString` and `run` must be of length `nArgs()`.

3.4 HARP as a Microservice

HARP allows our implementation of [11] to interface with the rest of the architecture. It maintains an instance of the reasoning framework that can be used after rules and queries are specified. (The original framework [11] only considers a single query, but HARP allows for multiple queries to be evaluated simultaneously.)

When HARP is initialised with a set of rules and queries, it performs a preprocessing step to compute the initial hypothetical answers. Later, it periodically fetches dataslices, rounds the time point up to the nearest hour, and passes them to a database for the reasoner to use for updating the hypothetical answers. The time required for this step depends on the current number of hypothetical answers and the size of the dataslice. Since dataslices are produced every hour, this computation time must be shorter than this limit. This issue is discussed in more detail in Section 6.

3.5 Implementation

Our implementation of the reasoning engine in HARP mostly follows the ideas presented in [11], but with two relevant differences that we describe here.

The first difference is that we have introduced user-defined predicates and atoms. This is a purely practical addition, which is orthogonal to the theory.

The second difference is our implementation of negation. Overall, our algorithm resembles the one described in [11]: we preprocess each query to find out all atoms and negated atoms that need to be proven for the query to hold. Each negated atom generates an auxiliary query that is treated similarly. Then, for each timestamp, we check whether these queries do not hold or have a complete proof.

The difference lies in how we instantiate the time parameters in these queries. In [11], each query answer is instantiated such that the smallest timestamp among its hypotheses becomes the current time. Instead, we instantiate each query answer such that the smallest timestamp in *either* among its hypotheses or the answer itself becomes the current time (whichever is smaller). This detail of considering also the answer itself when instantiating the time means that our system does not miss that a query might hold later because it requires evidence that is still not in the datastream.[4]

Furthermore, compared to the original conference version of this article [13], we have upgraded how our implementation deals with updating answers. In the original version, the dataslice was represented as a string. Parsing such strings has an upper limit to their size given by the memory limitations of the host system. To allow for arbitrarily large dataslices, the updated reasoner instead fetches the dataslice from a database in chunks small enough to stay within the RAM limitations.

4 Architecture

The overall architecture of μXL is shown in Figure 1, and consists of four basic components: Frontend, Data Sources, Data Manager, and Reasoner.

There are two operations that can be executed through the Frontend. The first is the initialisation of the processing pipeline. The user (an administrator) provides the input parameters of the Data Manager microservice and the HARP microservice. The Data Manager takes as input the necessary information for retrieve data from the specified public APIs, e.g., Google Trends. This information will be used to make requests to these APIs. This process takes place recurrently every t seconds, where t is a user-defined parameter. The data received by the Data Manager are stored in a database and the most recent views of data are aggregated, representing the current

[4]This was a minor bug in the theory proposed in [11], which the authors have since fixed (private communication).

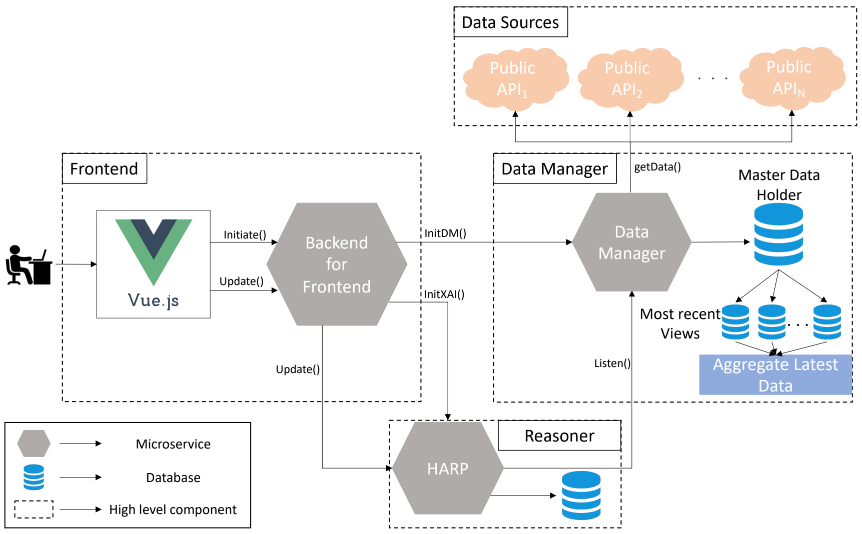

Fig. 1 Architecture overview.

state of the system. At the same time, the HARP microservice is also initialised with the appropriate parameters provided by the user through the Frontend. In particular, the HARP microservice sends a request to the Aggregate Latest Data and receives as response the aggregated most recent views of data. This process takes place recurrently as well. HARP processes these data and returns the current answers.

The second operation retrieves the most recent answers. The user makes a request that reaches the HARP microservice and retrieves as a response the current answers, which are displayed to the Frontend.

The rest of this section is dedicated to an analysis of our architecture in terms of patterns for API design (section 4.1) and a discussion of how the results produced by our tool can be interpreted (section 5.1).

4.1 Application of Patterns for API Design

The architecture of μXL can be analysed wrt the following categories of patterns for API design, as described in [12]:

- The **Foundation Patterns**, which deal with issues like *API Accessibility* (from where APIs should be accessed) and *API Integration* (e.g., whether a client interacts directly with the API or through other means).
- The **Responsibility Patterns**, which clarify the *Endpoint Roles* (the architectural roles of API endpoints) and the responsibilities of their operations.
- The **Quality Patterns**, which deal with the compromises between providing a high-quality service and cost-effectiveness. The Quality Patterns category comprises the following patterns: *Data Transfer Parsimony*, *Reference Management*, and *Quality Management and Governance*.

Data Manager

The Data Manager microservice employs the *Backend Integration* pattern for API Integration: it integrates with two other backends of the same application, the Backend for Frontend and HARP, and multiple other public APIs, by exposing its services via a message-based remote Backend Integration API. We detail the interaction between the Data Manager and these entities below.

- The Backend for Frontend makes `initDM()` makes requests to the Data Manager of the following type.

```
type InitDMRequest { hl: string, /* Host language */
    tz: int, /* Timezone offset in minutes */
    ns: int, geo: string, /* Geolocation */, t: long }
```

  A JSON payload example is

```
{ "hl": "en-us", "tz": 300, "ns": 15, "geo": "DK", "t": "100" }
```

  and the corresponding response has the format

```
type InitDMResponse { ack: boolean /* Acknowledgement */ }
```

- Public APIs, like Google Trends, receive messages of the format

```
type DailytrendsRequest { hl: string, /* Host language */
    tz: int, /* Timezone offset in minutes */
    ns: int, geo: string /* Geolocation */ }
```

and an example of the corresponding JSON payload is

```
{ "hl": "en-us", "tz": 300, "ns": 15, "geo": "DK" }
```

The response is a message of the following format

```
type DailytrendsResponse { entry*: DailytrendsElement }
type DailytrendsElement { query: string /* Trending topic */
    traffic: string /* Approximate traffic in string format */
    urllist*: string /* URLs of articles about trending topic */ }
```

and an example of the corresponding JSON payload is the following.

```
{[{"query":"Detroit Lions", "traffic": "10K",
   "urllist": ["https://en.as.com/resultados/superbowl/detroit_lions", "
       https://www.football-espana.net/superbowl/detroit_lions"]},
  {"query": "Thanksgiving parade", "traffic": "50K",
   "urllist":
       ["https://www.cbsnews.com/news/thanksgiving_parade_2022/"]}]}
```

- HARP sends requests to and receives responses from the Data Manager with the following formats.

```
type HARPRequest { datasource*: string }
type HARPResponse { t: long, facts*: string }
```

A JSON response example is the following.

```
{ "t": 1669306702000, "facts": ["Popularity(detroit lions,10K,463696)", "
    Popularity(thanksgiving parade,50K,463696)", "Popularity(uruguay,5K,
    463696)"] }
```

As concerns API Accessibility, for the Data Manager we follow the *Solution-Internal API* pattern: its APIs are offered only to system-internal communication partners, such as other services in the application backend.

The Data Manager utilises two Endpoint Roles, a *Processing Resource* and an *Information Holder Resource*. The former has to do with the `initDM()` request of the Backend for Frontend, which triggers the Data Manager to start requesting user-specified data from the public APIs. This is a *State Transition Operation*, where a client initiates a processing action that causes the provider-side application state to change. The Information Holder Resource endpoint role concerns the responses of the public APIs, which are data that need to be stored in some persistence and the requests from the HARP, which need to get the aggregated most recent views of data. This is both a *Master Data Holder*, because it accumulates historical data, and an *Operational Data Holder*, because it supports clients that want to read the most recent views of data.

Finally, regarding the Quality Patterns, the Data Manager follows *Rate Limit*, which is dictated both by the user's Data Manager `initDM()` request and the possible rate limits that public APIs might have.

Backend for Frontend

Regarding API Integration, the Backend for Frontend microservice employs the *Frontend Integration* pattern and more specifically the Backend for Frontend pattern, since it integrates the frontend with the backend of our application, by exposing its services via a message-based remote Frontend Integration API. In more detail, the Backend for Frontend integrates with (i) the Data Manager, using operation `initDM()` as previously described; and (ii) HARP, using the `initXAI()` and `update()` operations. We discuss those in the description of the HARP component, coming next.

For API Accessibility, the Backend for Frontend follows the *Community API* pattern, since in the future it is intended to support different kinds of frontends and the integration of our system with other tools used by journalists.

The Backend for Frontend uses two endpoints with one Endpoint Role. Both endpoints are *Processing Resource*s, which has to do with the `initDM()` and `initXAI()` request to the Data Manager and HARP, respectively, and the `update()` request to the HARP. The first two are *State Transition Operations*, where a client initiates a processing action that causes the provider-side application state to change, while the latter is a *Retrieval Operation*, where information available from HARP gets retrieved for an end user.

Finally, regarding the Quality Patterns, the Backend for Frontend follows *Rate Limit* which is dictated by the user's `initDM()` and `initXAI()` requests.

HARP

HARP employs the *Backend Integration* pattern: it exposes its message-based API to two other backend services of the same application, the Backend for Frontend and the Data Manager. The integration between HARP and these two components is described in more detail below.

- The Backend for Frontend makes `initXAI()` requests of the type

```
type InitXAIRequest { name: string, /* Instance name */
    t: long, /* Interval period for each call (in ms) */
    target: string, /* Location of the data source */
    datasources*: string /* Data sources to retrieve data */
    rules*: string /* Rules to be initialised at HARP instance */
    queries*: string /* Queries to be initilizated at HARP instance }
```

  of which the following example illustrates a possible JSON payload.

```
{ "name": "HARP-example", "t": 3600000,
  "target": "getDailyTrends",
  "datasources": ["GoogleTrends_dailytrends"],
  "rules": ["GlobalLead(Topic, T) -> CertainLead(Topic, Region, T)",
           "CertainLead(Topic, Region, T) -> Lead(Topic, Region, T)"]
  "queries":["Lead(Topic, dk, T)","GlobalLead(Topic,T)"] }
```

  The corresponding response has the following format.

```
type InitXAIResponse { ack: boolean /* Acknowledgement */}
```

  The Backend for Frontend can also make `update()` requests to HARP, of type

```
type UpdateRequest { query: string }
```

and an example of the corresponding JSON payload is

```
{"query": "Lead(Topic, Region, T)"}
```

Furthermore, HARP gives a response of the following type.

```
type UpdateResponse { answers*: AnswerElement,
  hypotheticalAnswers*: HypotheticalAnswerElement }
type AnswerElement { answer: string, evidence*: string, explanation*:
    string}
type HypotheticalAnswerElement {
  answer: string, hypotheses*: string, evidence*: string, explanation*:
    string}
```

These types correspond to the (hypothetical) answers explained in Section 2. They also include an *explanation*, which is simply the set of rules from the program that were used to derive the initial hypothetical answer. The following is an example of JSON payload.

```
{"answers": [
   {"answer": "Lead(detroit lions,dk,463694)",
    "evidence": [
        "DailyTrend(detroit lions,dk,463694)",
        "Popularity(detroit lions,dk,8K,463694)",
        "Popularity(detroit lions,dk,9k,463695)",
        "Popularity(detroit lions,dk,10K,463696),"
        "8K<9k","9k<10k"],
    "explanation": [
        "CertainLead(detroit lions,dk,463694) -> Lead(detroit lions,dk,
            463694)",
        "PopularityLead(detroit lions,dk,463694) -> CertainLead(detroit
            lions,dk,463694)",
        "DailyTrend(detroit lions,dk,463694), Popularity(detroit lions,dk,8
            K,463694),
         Popularity(detroit lions,dk,9K,463695),
         Popularity(detroit lions,dk,10K,463696),
         Less(8K,9K), Less(9K,10K) -> PopularityLead(detroit lions,dk,
             463694)"]},
   {"answer": "Lead(thanksgiving parade,dk,463694)",
    "evidence": [
        "DailyTrend(thanksgiving parade,dk,463694)",
        "Popularity(thanksgiving parade,35K,463694)",
        "Popularity(thanksgiving parade,47K,463695)",
        "Popularity(thanksgiving parade,50K,463696)",
        "35K<47K","47K<50K"],
    "explanation": [
        "CertainLead(thanksgiving parade,dk,463694) -> Lead(thanksgiving
            parade,dk,463694)",
       "PopularityLead(thanksgiving parade,dk,463694) -> CertainLead(thanksgiving parade,dk,463694)",
       "DailyTrend(thanksgiving parade,dk,463694), Popularity(thanksgiving parade,dk,35K,463694),
        Popularity(thanksgiving parade,dk,47K,463695),
        Popularity(thanksgiving parade,dk,50K,463696),
        Less(35K,47K), Less(47K,50K) -> PopularityLead(thanksgiving parade,dk,463694)"]}],
 "hypotheticalAnswers": [
  {"answer": "Lead(detroit lions,dk,463695)",
   "hypotheses": ["10k<Pop2",
       "Popularity(detroit lions,dk,Pop2,463697)"],
   "evidence": [
       "Popularity(detroit lions,dk,9K,463695)",
       "Popularity(detroit lions,dk,10K,463696)",
       "9k<10k"],
   "explanation": [
       "CertainLead(detroit lions,dk,463695) -> Lead(detroit lions,dk,463695)",
       "PopularityLead(detroit lions,dk,463695) -> CertainLead(detroit lions,dk,463695)",
       "DailyTrend(detroit lions,dk,463695), Popularity(detroit lions,dk,9K,463695),
        Popularity(detroit lions,dk,10K,463696),
        Popularity(detroit lions,dk,Pop2,463695 + 2),
        Less(9K,10K), Less(10K,Pop2) -> PopularityLead(detroit lions,dk,463695)"]},
  {"answer": "Lead(thanksgiving parade,dk,463695)",
   "hypotheses": ["50k<Pop2"
       "Popularity(thanksgiving parade,dk,Pop2,463697)"],
   "evidence": ["47k<50k",
       "DailyTrend(thanksgiving parade,dk,463695)",
       "Popularity(thanksgiving parade,dk,47K,463695)",
       "Popularity(thanksgiving parade,dk,50K,463696)"],
   "explanation": [
       "CertainLead(thanksgiving parade,dk,463695) -> Lead(thanksgiving parade,dk,463695)",
       "PopularityLead(thanksgiving parade,dk,463695) -> CertainLead(thanksgiving parade,dk,463695)",
       "DailyTrend(thanksgiving parade,dk,463695), Popularity(thanksgiving parade,dk,47K,463695),
        Popularity(thanksgiving parade,dk,50K,463696),
        Popularity(thanksgiving parade,dk,Pop2,463695 + 2),
        Less(47K,50k), Less(50K,Pop2) -> PopularityLead(thanksgiving parade,dk,463695)"]}] }
```

- HARP also makes `listen()` requests to the Data Manager of the form

```
type ListenRequest { datasources*: string /* Data sources */}
```

and receives a response of the following type

```
type ListenResponse { t: long, facts*: string /* List of facts for time t
    */}
```

such as

```
{ "t": 1669306702000, "facts": [
    "Popularity(detroit lions,10K,463696)",
    "Popularity(thanksgiving parade,50K,463696)",
    "Popularity(uruguay,5K,463696)"] }
```

For API Accessibility, HARP follows the *Solution-Internal API* pattern.

HARP uses two Endpoint Roles, a *Processing Resource* and an *Information Holder Resource*. The former concerns the response of the Data Manager, which triggers HARP to produce the current answers. This is actually a *State Transition Operation*, where the Data Manager's response initiates a processing action that causes the provider-side application state to change. The same applies also for the `initXAI()` request. The Information Holder Resource endpoint role concerns the `update()` requests from the Backend for Frontend, which asks for the most recent answers. In more detail, this is an *Operational Data Holder* in the sense that it supports clients that want to read the most recent calculated answers.

Finally, HARP follows the *Rate Limit* Quality Pattern, dictated both by the initial `initXAI()` request and the rate limits that public APIs might have.

5 Discussion

In this section, we discuss some interesting aspects of our architecture and development experience. In particular, we cover how to interpret the meaning of explanations in leads, practical considerations that came to light in discussions with journalists, and our experience in implementing the architecture with the Jolie programming language.

5.1 Explaining Leads

The answer payload example starting on page 15 reports two (complete) answers (in the array `answers`) and two hypothetical answers (in the array `hypotheticalAnswers`). We purposefully chose examples based on similar rules and topics to illustrate the parametricity of our rules and the role of time in our system.

The first two answers are completely supported by evidence. The first, `Lead(detroit lions,dk,463694)`, means that 'Detroit Lions' is a lead in Denmark (`dk`) on 24 November 2022 at 15 o'clock. It was concluded by several pieces of evidence: it is a daily trend on the same day, and its popularity increased steadily for 3 subsequent hours. The user can also inspect which rule applications were used for deriving the answer from the evidence, reported in `explanation`. The most interesting one is the last, which uses the evidence directly to conclude a lead based on popularity. Such a lead is always judged as a certain lead (second rule application), which then translates to something that should be presented to the user (first rule application). The second answer, `Lead(thanksgiving parade,dk,463694)`, means that 'Thanksgiving parade' is lead in Denmark at the same timestamp, and it is similarly concluded.

Besides these two topics being confirmed leads for the specified hour, the system is also informing the user that they represent leads of potential interest for the next hour as well. Specifically, the given hypothetical answers include hypotheses about a still missing (because it is in the future) observation of popularity that, if it occurs, would make the lead confirmed.

5.2 Adjusting for Practice

We report on some considerations that came to light while discussing our tool together with external parties. We broadly separate discussions in two phases.

5.2.1 Phase 1: Co-Design Workshops

In Phase 1, two of the authors participated in two joint workshops with 3 Danish media companies (two of middle size dealing with regional news, one large dealing with national news and television). Participants at these workshops included several journalists, managers, and researchers in both social and computer science. The goal of the workshops was to formulate features for the design of the system that were both feasible and important for adoption.

A key feature of our system is that rules are configurable. Connecting to our example, some journalists might be more interested in a topic being upward trending over a span of days, or even weeks, instead of mere hours. This can be easily achieved by changing the time parameters of our rules, e.g., requiring time differences of 24 hours.

An interesting, and more challenging, point is that different users might want to use different rule sets altogether. For example, a user might want rules that focus on a particular topic and adopt a long time span in order to follow up on a story, while another user might want to focus on recent leads in order to catch new stories. Rules of these kinds can be formulated in our system, but supporting this use case would require extending our architecture to start up different instances of the reasoning engine – each instantiated with a dedicated rule set. In the workshops, we decided to leave this feature to future work but also to design for future extensibility: as a benefit of adopting microservices, all different instances of HARP would be able to get access to the same data manager, if wanted.

Sometimes, users would like simplified information to be displayed to them, instead of full-blown explanations. This simple information can be, for example, displayed in the form of ‘badges’ that express facts like ‘locally trending’ or ‘globally trending’. These badges could even be coloured based on how intense the spike of the trend is. These wishes are easier applications of our methods, since we can just formulate queries that fit these badges.

5.2.2 Phase 2: User Feedback

Phase 2 is a first round of feedback from early adopters. Four practitioners at journalistic companies interested in news from different areas in Denmark have been given access to the tool through a web application hosted at the authors’ institution. These practitioners have experience in both journalism and the use of computer tools. In addition to generally positive feedback on the system, the participants expressed the next three wishes.

Users have expressed particularly interested in getting leads about potential stories of local interest, for example topics that are at the same time (i) getting discussed on an Internet forum about a specific city and (ii) not of global interest. Allowing for such use cases is what motivated us to implement support for negation in this extended version of our work.

While the quality of our leads in terms of presentation was judged positively, their usefulness could be significantly improved by tapping in additional data sources. In particular, users suggested to include reports from meetings at Danish municipalities, since these are hard to track manually. We leave this to future work.

Finally, users wish for integrating our tool with their existing workflows and systems. Some even asked explicitly for data APIs, as we provide. This validates further our choice of using a microservice architecture.

5.3 Jolie Implementation

We now report on our experience of implementing the architecture using the Jolie programming language.

One of the distinguishing features of Jolie is its technology-agnostic interface language. For our work, this characteristic of the language proved to be quite useful, as it simplified establishing clear contracts among the different services and the implementation of the API patterns.

A substantial part of our system deals with data, specifically: (i) invoking an external service to get some semi-structured data (e.g., from Google Trends), and (ii) manipulating this data to clean it, aggregate it, and create the structures that we are interested in. We capitalised extensively on the syntax of Jolie for these tasks.

As a concrete example, consider the task, performed by the Data Manager, that consists of inserting data into the database. The Jolie code responsible for this task is reproduced below.

```
realtimetrends@gtAdapter(rtrequest)(rtresponse)
for( entry in rtresponse ) {
    update@db(
        "INSERT INTO realtimetrends VALUES (:t, :title, :geo)" {
            t = tnow_response, title = entry.title[0], geo = rtrequest.geo
        }
    )()
}
```

This snippet shows how the database is updated with the most recent real-time trends, in lockstep with them being received by the adapter for Google Trends. For each real-time trend in the adapter's response, we issue an INSERT query to the database, where we store the current timestamp, the title of the real-time trend, and the geo-location where this topic is trending. The use of Jolie makes this function very simple to write, requiring only understanding of the type of responses produced by Google Trends.

Another useful feature that we benefit from is embedding, a Jolie feature that allows for loading a microservice inside of another microservice [9]. Specifically, the adapter for Google Trends is embedded in the Data Manager. This allows the Data Manager to use it as a service without it needing to be externally deployed, whilst retaining the conceptual separation between the two components. Furthermore, using embedding in this implementation makes it easy for the programmer to change this choice, for example by switching to a deployment strategy based on an external service if desired.

The most significant challenge that we encountered was the fact that several of the authors were not familiar with Jolie beforehand. This slowed the development of the system in a first phase, as it required a parallel learning process that was not always fully supported by detailed documentation. Examples include understanding how services written in other languages (e.g., Java) could easily be embedded in our system, and how the JavaScript library that we chose for the Frontend could communicate effectively with Jolie. Our experience has been communicated to the Jolie development team for future documentation efforts.

Another factor that slowed some parts of the development was the lack of a full-fledged IDE plugin supporting all stages of the software development cycle. The available tools can highlight syntax errors and suggest available methods, thereby supporting the programmer in the first stages of writing code. Unfortunately, there are currently no IDEs that also include a debugging functionality, which slowed down the testing stage of the project. Hopefully, this issue will be solved in a near future. There is already interesting work that has been recently completed for simplifying testing of microservice architectures with Jolie, which we plan to adopt in the future [42].

A last limitation is that Jolie, being a young programming language, is still somewhat lacking in the domain of libraries. In our case, we would have benefited from the availability of libraries that allowed us to access interesting services, like Google Trends, directly. On the other hand, the adapter that we developed to get data from Google Trends can be the starting point for the development of such libraries.

6 Experimental Evaluation

The bottleneck of our system consists of the preprocessing and update steps in the reasoner. In this section we empirically explore the cost of these computations. Experiments were performed on a machine with an Intel i5-10400 CPU, 64GB RAM, and Windows 11. Our results are shown in Figures 2.

The preprocessing time depends on the rules. (Determining the precise form that rules must have for the worst-case preprocessing time is a task beyond the scope of this article.) In the simple case of a single rule, the preprocessing time increases linearly as the size of the rule's body varies from 0 to 100 000 atoms (Figure 2, left). The general case is known to be exponential [40].

In the right part of Figure 2, we test the time needed for updating the set of hypothetical answers depending on the amount of atoms in the current dataslice and (relevant) previously-computed hypothetical answers. The latter depends on the relevance of the previous data: if data from the datastream matches with atoms in the hypotheses of a hypothetical answer, then more hypothetical answers might be created.

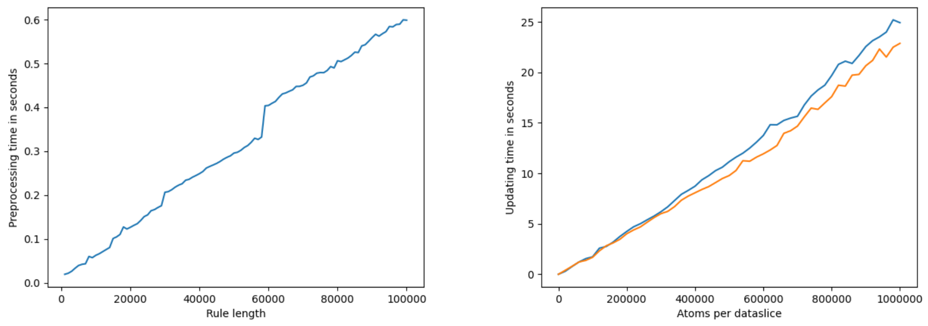

Fig. 2 Left: preprocessing time for a single rule with a body of 0 to 100 000 atoms, at intervals of 1 000, averaged over 10 runs. Right: comparison of the time it takes to update answers using either an in-memory string (orange line) or a database (blue line). The update benchmarks were run with 0 to 1 000 000 different atoms of the form `DailyTrend(Topic,dk,t)` and `Popularity(Topic,dk,t,t)`, at intervals of 20 000 atoms, averaged over 10 runs.

The worst-case scenario is that every data matches with atoms in the hypotheses. The execution time of updates using the rules presented in Section 3.2 has been evaluated in this scenario. We compare the original implementation of the update operation (given in the conference version of this article), which retrieved the entire dataslice at once (orange line), against our new implementation that instead fetches the dataslice in chunks (blue line). For our testing configuration, the former hits the memory limit at around 1 000 000 atoms. Thus, testing was done with data sizes ranging from 0 to 1 000 000 atoms. Execution time peaked at around 24 seconds at 1 000 000 atoms. Getting the entire dataslice in memory at once runs faster compared to processing the data in chunks, as expected. The comparison, however, shows that the price to pay in runtime performance to overcome memory limitations is relatively small (8% on average).

Overall, this evaluation of the performance of our reasoner is satisfactory: service startup is affected only minimally (under a second), and updates can be performed reasonably often. (Note that the Frontend does not need to wait for updates when it asks for the current state, since the latter is cached until a new state is produced by finishing an update.) Nevertheless, we discuss potential improvements in Section 7.

7 Conclusions and Future Work

We have developed μXL, the first extensible system for lead generation that integrates the integration benefits of microservices with explainable AI. Our development is motivated by concrete needs identified in a collaboration with Danish media companies, including (i) integrating our results within existing systems via service APIs, (ii) accompanying answers with explanations of how they were reached, and (iii) and the capability of adding new rules to the system. These needs oriented us towards the adoption of recent theories and tools, in particular API Patterns [12], the Jolie programming language [9], and hypothetical reasoning over data streams [11]. Thus, our work also serves as a practical validation of these methods. Interactions with early adopters have supported our design choices (Section 5), confirming that μXL provides an interesting basis for future developments.

In this work we have focused on the architectural and technical aspects of μXL. However, while early feedback is positive regarding our architectural choices, it would be interesting to evaluate the usefulness of μXL for journalists more thoroughly. In particular, future developments should be guided by: carrying out systematic comparisons against other tools based on different architectures and AI; and conducting controlled user experiments. Other future directions include extending the system to more data sources and integrating more kinds of AI in addition to HARP. Another interesting improvement is parallelising HARP's update operation, such that it can scale to dataslices with sizes of billions or more. A more conceptual extension is to incorporate the possibility of having delays in the data, or allow negation in the specification of rules, as described in [11]. Finally, we plan on exploring procedures that suggest interesting rules, for example based on statistical observations of journalistic behaviour.

Acknowledgements.

We thank Narongrit Unwerawattana for his technical support. Work partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

References

[1] Aiello, L.M., Petkos, G., Martín, C.J., Corney, D.P.A., Papadopoulos, S., Skraba, R., Göker, A., Kompatsiaris, I., Jaimes, A.: Sensing trending topics in Twitter. IEEE Trans. Multim. **15**(6), 1268–1282 (2013) https://doi.org/10.1109/TMM.2013.2265080

[2] Mathioudakis, M., Koudas, N.: TwitterMonitor: trend detection over the Twitter stream. In: Elmagarmid, A.K., Agrawal, D. (eds.) Procs. SIGMOD, pp. 1155–1158. ACM, New York, NY, United States (2010). https://doi.org/10.1145/1807167.1807306

[3] Das, A., Roy, M., Dutta, S., Ghosh, S., Das, A.K.: Predicting trends in the Twitter social network: A machine learning approach. In: Panigrahi, B.K., Suganthan, P.N., Das, S. (eds.) Swarm, Evolutionary, and Memetic Computing. Lecture Notes in Computer Science, vol. 8947, pp. 570–581. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2014). https://doi.org/10.1007/978-3-319-20294-5_49

[4] Pugachev, A., Voronov, A., Makarov, I.: Prediction of news popularity via keywords extraction and trends tracking. In: Aalst, W.M.P., Batagelj, V., Buzmakov, A., Ignatov, D.I., Kalenkova, A.A., Khachay, M.Y., Koltsova, O., Kutuzov, A., Kuznetsov, S.O., Lomazova, I.A., Loukachevitch, N.V., Makarov, I., Napoli, A., Panchenko, A., Pardalos, P.M., Pelillo, M., Savchenko, A.V., Tutubalina, E. (eds.)

Recent Trends in Analysis of Images, Social Networks and Texts. Communications in Computer and Information Science, vol. 1357, pp. 37–51. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2020). https://doi.org/10.1007/978-3-030-71214-3_4

[5] Zarrinkalam, F., Fani, H., Bagheri, E., Kahani, M.: Predicting users' future interests on Twitter. In: Jose, J.M., Hauff, C., Altingövde, I.S., Song, D., Albakour, D., Watt, S.N.K., Tait, J. (eds.) Advances in Information Retrieval. Lecture Notes in Computer Science, vol. 10193, pp. 464–476. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2017). https://doi.org/10.1007/978-3-319-56608-5_36

[6] Leppänen, L., Munezero, M., Granroth-Wilding, M., Toivonen, H.: Data-driven news generation for automated journalism. In: Alonso, J.M., Bugarín, A., Reiter, E. (eds.) Procs. INLG, pp. 188–197. Association for Computational Linguistics, 209 N. Eighth Street Stroudsburg, PA 18360 USA (2017). https://doi.org/10.18653/v1/w17-3528

[7] Huang, Q., Liu, Z., Rosenberg, A.E., Gibbon, D.C., Shahraray, B.: Automated generation of news content hierarchy by integrating audio, video, and text information. In: Procs. ICASSP, pp. 3025–3028. IEEE Computer Society, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331 U.S.A. (1999). https://doi.org/10.1109/ICASSP.1999.757478

[8] Diakopoulos, N., Dong, M., Bronner, L.: Generating location-based news leads for national politics reporting. In: Proc Computational + Journalism Symposium (2020)

[9] Montesi, F., Guidi, C., Zavattaro, G.: Service-oriented programming with Jolie. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) Web Services Foundations, pp. 81–107. Springer, Springer Science+Business Media New York (2014). https://doi.org/10.1007/978-1-4614-7518-7_4

[10] Montesi, F.: Process-aware web programming with Jolie. Sci. Comput. Program. **130**, 69–96 (2016) https://doi.org/10.1016/j.scico.2016.05.002

[11] Cruz-Filipe, L., Nunes, I., Gaspar, G.: Hypothetical answers to continuous queries over data streams. In: Procs. AAAI, pp. 2798–2805. AAAI Press, Palo Alto, California USA (2020). https://doi.org/10.1609/aaai.v34i03.5668

[12] Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., Pautasso, C.: Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. Addison-Wesley Signature Series (Vernon). Addison-Wesley Professional, 221 River Street Hoboken, NJ 07030 USA (2022)

[13] Cruz-Filipe, L., Kostopoulou, S., Montesi, F., Vistrup, J.: μxl: Explainable lead generation with microservices and hypothetical answers. In: Papadopoulos, G.A.,

Rademacher, F., Soldani, J. (eds.) Procs. ESOCC. Lecture Notes in Computer Science, vol. 14183, pp. 3–18. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2023). https://doi.org/10.1007/978-3-031-46235-1_1

[14] Oh, C., Choi, J., Lee, S., Park, S., Kim, D., Song, J., Kim, D., Lee, J., Suh, B.: Understanding user perception of automated news generation system. In: Bernhaupt, R., Mueller, F.F., Verweij, D., Andres, J., McGrenere, J., Cockburn, A., Avellino, I., Goguey, A., Bjøn, P., Zhao, S., Samson, B.P., Kocielnik, R. (eds.) Procs. CHI, pp. 1–13. ACM, PO Box 30777 New York, NY 10087-0777, USA (2020). https://doi.org/10.1145/3313831.3376811

[15] Chen, Y., Amiri, H., Li, Z., Chua, T.: Emerging topic detection for organizations from microblogs. In: Jones, G.J.F., Sheridan, P., Kelly, D., Rijke, M., Sakai, T. (eds.) Procs. SIGIR, pp. 43–52. ACM, PO Box 30777 New York, NY 10087-0777, USA (2013). https://doi.org/10.1145/2484028.2484057

[16] Schwartz, R., Naaman, M., Teodoro, R.: Editorial algorithms: Using social media to discover and report local news. In: Cha, M., Mascolo, C., Sandvig, C. (eds.) Procs. ICWSM, pp. 407–415. AAAI Press, Palo Alto, California USA (2015). https://doi.org/10.1609/icwsm.v9i1.14633

[17] Artun, O., Levin, D.: Predictive Marketing: Easy Ways Every Marketer Can Use Customer Analytics and Big Data. John Wiley & Sons, 111 River Street, Hoboken, NJ 07030 (2015)

[18] Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2017). https://doi.org/10.1007/978-3-319-67425-4_12

[19] Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., Zdun, U.: Introduction to microservice API patterns (MAP). In: Cruz-Filipe, L., Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S. (eds.) Joint Procs. Microservies. OASIcs, vol. 78. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Oktavie-Allee, 66687 Wadern, Germany (2019). https://doi.org/10.4230/OASICS.MICROSERVICES.2017-2019.4

[20] Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U., Stocker, M.: Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In: Sousa, T.B. (ed.) Procs. EuroPloP. ACM, 2 Penn Plaza, Suite 701 New York New York 10121-0701 (2019). https://doi.org/10.1145/3361149.3361164

[21] Zimmermann, O., Pautasso, C., Lübke, D., Zdun, U., Stocker, M.: Data-oriented interface responsibility patterns: Types of information holder resources. In: Procs. EuroPLoP. ACM, 1601 Broadway, 10th Floor New York, New York 10019, USA (2020). https://doi.org/10.1145/3424771.3424821

[22] Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C., Stocker, M.: Interface responsibility patterns: Processing resources and operation responsibilities. In: Procs. EuroPLoP. ACM, 1601 Broadway, 10th Floor New York, New York 10019, USA (2020). https://doi.org/10.1145/3424771.3424822

[23] Stocker, M., Zimmermann, O., Zdun, U., Lübke, D., Pautasso, C.: Interface quality patterns: Communicating and improving the quality of microservices apis. In: Procs. EuroPLoP. ACM, 2 Penn Plaza, Suite 701 New York New York 10121-0701 (2018). https://doi.org/10.1145/3282308.3282319

[24] Zimmermann, O., Stocker, M., Lübke, D., Zdun, U.: Interface representation patterns: Crafting and consuming message-based remote apis. In: Procs. EuroPLoP. ACM, 2 Penn Plaza, Suite 701 New York New York 10121-0701 (2017). https://doi.org/10.1145/3147704.3147734

[25] Oram, A.: Ballerina: A Language for Network-Distributed Applications. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472 (2019)

[26] Rademacher, F., Sorgalla, J., Wizenty, P., Sachweh, S., Zündorf, A.: Graphical and textual model-driven microservice development. In: Microservices: Science and Engineering, pp. 147–179. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2020)

[27] Montesi, F., Guidi, C., Lucchi, R., Zavattaro, G.: JOLIE: a Java orchestration language interpreter engine. In: CoOrg 2006 and MTCoord 2006. Electronic Notes in Theoretical Computer Science, vol. 181, pp. 19–33. Elsevier, PO Box 211 1000 AE Amsterdam, Netherlands (2006). https://doi.org/10.1016/j.entcs.2007.01.051

[28] Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundam. Informaticae **95**(1), 73–102 (2009) https://doi.org/10.3233/FI-2009-143

[29] Montesi, F., Carbone, M.: Programming services with correlation sets. In: Kappel, G., Maamar, Z., Nezhad, H.R.M. (eds.) Procs. ICSOC. Lecture Notes in Computer Science, vol. 7084, pp. 125–141. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2011). https://doi.org/10.1007/978-3-642-25535-9_9

[30] Bandura, A., Kurilenko, N., Mazzara, M., Rivera, V., Safina, L., Tchitchigin, A.: Jolie community on the rise. In: Procs. SOCA, pp. 40–43. IEEE Computer Society, Center, 445 Hoes Lane, P.O. Box 133, Piscataway, NJ 08855-1331. (2016). https://doi.org/10.1109/SOCA.2016.16

[31] Guidi, C., Maschio, B.: A Jolie based platform for speeding-up the digitalization of system integration processes. In: Microservices, (2019). https://www.conf-micro.services/2019/papers/Microservices_2019_paper_6.pdf

[32] Gabbrielli, M., Martini, S., Giallorenzo, S.: Programming Languages: Principles and Paradigms, Second Edition. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2023)

[33] Giaretta, A., Dragoni, N., Mazzara, M.: Joining Jolie to docker - orchestration of microservices on a containers-as-a-service layer. In: Ciancarini, P., Litvinov, S., Messina, A., Sillitti, A., Succi, G. (eds.) Procs. SEDA. Advances in Intelligent Systems and Computing, vol. 717, pp. 167–175. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2016). https://doi.org/10.1007/978-3-319-70578-1_16

[34] Gusmanov, K., Khanda, K., Salikhov, D., Mazzara, M., Mavridis, N.: Jolie good buildings: Internet of things for smart building infrastructure supporting concurrent apps utilizing distributed microservices. CoRR **abs/1611.08995** (2016) 1611.08995

[35] Gabbrielli, M., Giallorenzo, S., Lanese, I., Zingaro, S.P.: A language-based approach for interoperability of IoT platforms. In: Bui, T. (ed.) Procs. HICSS, pp. 1–10. ScholarSpace / AIS Electronic Library (AISeL), 2404 Maile Way, D307, Honolulu, HI 96822 (2018)

[36] Montesi, F., Weber, J.: From the decorator pattern to circuit breakers in microservices. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) Procs. ACM SAC, pp. 1733–1735. ACM, New York, NY, United States (2018). https://doi.org/10.1145/3167132.3167427

[37] Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S.: Jolie and LEMMA: model-driven engineering and programming languages meet on microservices. In: Damiani, F., Dardha, O. (eds.) Procs. COORDINATION. Lecture Notes in Computer Science, vol. 12717, pp. 276–284. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2021). https://doi.org/10.1007/978-3-030-78142-2_17

[38] Rademacher, F., Sorgalla, J., Wizenty, P., Trebbau, S.: Towards holistic modeling of microservice architectures using LEMMA. In: Procs. CEUR, vol. 2978. CEUR-WS.org, Växjö, Sweden (2021). https://ceur-ws.org/Vol-2978/

[39] Chomicki, J., Imielinski, T.: Temporal deductive databases and infinite objects. In: Edmondson-Yurkanan, C., Yannakakis, M. (eds.) Procs. SIGMOD, pp. 61–73. ACM, New York, NY, United States (1988). https://doi.org/10.1145/308386.308416

[40] Kowalski, R.A.: Predicate logic as programming language. In: Rosenfeld, J.L. (ed.) Procs. IFIP, pp. 569–574. North-Holland, Netherlands (1974)

[41] Clark, K.: Negation as failure, pp. 293–322 (1977). https://doi.org/10.1007/978-1-4684-3384-5_11

[42] Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Unwerawattana, N.: Jot: A jolie framework for testing microservices. In: Jongmans, S., Lopes, A. (eds.) Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13908, pp. 172–191. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2023). https://doi.org/10.1007/978-3-031-35361-1_10