μXL: Explainable Lead Generation with Microservices and Hypothetical Answers[⋆]

Luís Cruz-Filipe$^{1\,[0000-0002-7866-7484]}$, Sofia Kostopoulou$^{1\,[0000-0002-5557-1295]}$, Fabrizio Montesi$^{1\,[0000-0003-4666-901X]}$, and Jonas Vistrup$^{1\,[0000-0001-7704-3656]}$

Dept. Mathematics and Computer Science, Univ. Southern Denmark
{lcf,fmontesi,vistrup}@imada.sdu.dk,skos@sdu.dk

Abstract. Lead generation refers to the identification of potential topics (the 'leads') of importance for journalists to report on. In this paper we present a new lead generation tool based on a microservice architecture, which includes a component of explainable AI. The lead generation tool collects and stores historical and real-time data from a web source, like Google Trends, and generates current and future leads. These leads are produced by an engine for hypothetical reasoning based on logical rules, which is a novel implementation of a recent theory. Finally, the leads are displayed on a web interface for end users, in particular journalists. This interface provides information on why a specific topic is or may become a lead, assisting journalists in deciding where to focus their attention. We carry out an empirical evaluation of the performance of our tool.

Keywords: Lead generation · Microservices · Explainable AI

1 Introduction

Background. 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 [1,13]. 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 [5] aim to predict trending keywords, the work in [18] targets forecasting article popularity, and [20] focuses on the prediction of future users' interests. Automated news generation is another field of research

⋆ Work partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

that received much attention by researchers. The authors in [12] present an architecture for automated journalism and in [10] 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 [6] 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.

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 [15], 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 [14]. 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 [4]. This allows our system to present potential leads given the facts that are currently available, and accompany them with rule-based explanations that clearly distinguishes observed facts from hypotheses about the future.

The contributions of our paper 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 artificial intelligence (AI) to generate explainable leads.

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

- A technology-agnostic description of the APIs and patterns used in our system, which are respectively expressed in Jolie [15] and the API patterns recently exposed in [21]. This serves three purposes. For us, Jolie and API patterns were useful guides. For the reader, it clarifies our design. And for Jolie and the collection of API patterns, it is an additional validation of their usefulness in practice. (For both Jolie and API patterns, it is the first validation in the journalistic domain that we know of.)
- The first implementation of a hypothetical answer reasoning engine – which given rules and online datastreams can produce explainable leads – and its integration in our architecture. Our engine is based on the theory originally presented in [4], so our work also serves as the first validation of its usefulness.
- A performance evaluation of our lead generation component (the reasoner).

Structure of the paper. Section 2 presents relevant related work and background on the reasoning theory that we use. Section 3 describes our explainable AI engine. Section 4 is dedicated to the system's architecture. Section 5 provides our experimental evaluation. Finally, section 6 concludes with future work.

2 Related Work

AI and Journalism. The use of AI in journalism is seeing increased focus. One of the perspectives relevant to this work is automated news generation. In this realm, the authors in [17] designed 'News robot', a system that automatically generates live events or news of the 2018 Winter Olympic Games. This system generates six news types by joining general and individualised content with a combination of text, image, and sound. In another work [10], an automatic news generation solution with semantically meaningful content was proposed by integrating audio, video, and text information in the context of broadcast news. While in [12] the authors presented an automatic news generation system that is largely language and domain independent.

Another perspective is trending topic detection, where programs try to distinguish trending topics in a wealth of news sources. In that context, the authors in [1] compared six methods used for topic detection relevant to major events on Twitter. Moreover, the work in [13] proposed 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. While the authors in [2] 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.

There is also a lot of research dedicated to predicting trends, mostly using machine learning techniques. For instance, the authors of [5] tackled trending topic prediction as a classification problem. They used online Twitter data to detect features that are distinguished as trending/non-trending hashtags, and developed classifiers using these features. Furthermore, the work in [18] proposed a solution that extracts keywords from an article and then predicts its popularity based on these keywords. They compared their approach to other popular ones based on the BERT model and text embeddings. A connected problem is that of predicting future user interests, which the authors of [20] explored in the context of microblogging services and unobserved topics. Specifically, they built topic profiles for users based on discrete time intervals, and then transferred user interests to the Wikipedia category structure.

The most relevant work to lead generation is the one proposed by the authors in [6]. They designed, developed, and evaluated a news discovery tool, called 'Lead Locator', which supplements the reporting of national politics by suggesting possibly noteworthy locations to write a story about. They analysed a national voter file using data mining to rank counties with respect to their possible newsworthiness to reporters. Then, 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. In a similar way, the authors in [19] developed 'CityBeat', a system that finds potential news events. It collects geo-tagged information in real-time from social media, finds important stories, and makes an editorial choice on whether these events are newsworthy.

Microservices and Jolie. Microservices are cohesive and independently-executable software applications that interact by message passing. Their origins and reasons for diffusion are surveyed in [7], along with open challenges and future directions.

Jolie is a service-oriented programming language that provides native linguistic constructs for the programming of microservices [15]. Its abstractions have been validated both in terms of industrial productivity [9], development of security strategies [16], and engineering: Jolie's structures resemble the architectural metamodels found in tools for Model-Driven Engineering of microservices based on Domain-Driven Design [8]. We mention a few relevant aspects. First, Jolie comes with an algebraic language for composing communication actions, which 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). We use the 'all separate' option in our description, but adopters are free to change this decision.

Hypothetical Query Answering. The explainable AI component of μXL implements the theory presented in [4], which allows for producing hypothetical answers (answers that depend on the occurrence of future events). We dedicate the rest of this section to the necessary background on this theory.

The theory in [4] is based on *Temporal Datalog* [3], which is a negation-free variant of Datalog where predicates include temporal attributes. Temporal

Datalog has two types of terms: *object* and *time*. An object term is either an object (constant) or an object variable. A time 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 $P(t_1, \ldots, t_n, \tau)$, where P is a name of a predicate with $n \in \mathbb{N}$ object terms, t_1 to t_n are all object terms, and τ is a time term. Intuitively the semantics of $P(t_1, \ldots, t_n, \tau)$ is defined such that predicate P holds for terms t_1 to t_n at time τ.

Programs are sets of rules of the form $\alpha \leftarrow \alpha_1 \wedge \ldots \wedge \alpha_n$ with α, α_1, $\ldots$, α_n atoms. The *head* of the rule is α, and the *body* of the rule is $\alpha_1 \wedge \ldots \wedge \alpha_n$. 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 *datastream* is a set of *dataslices*, one for each natural number. Each dataslice consists of a finite number of atoms with extensional predicates, each with the dataslice's index as their temporal parameter. A *query* is a list of atoms, and an *answer* to a query is a substitution that makes the query valid.

Given a Temporal Datalog program and a datastream, we can compute *hypothetical answers* 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 [4] is a modification of SLD-resolution [11] from logic programming. It maintains a list of hypothetical answers that are updated as new dataslices are produced.

To provide explainability of how hypothetical answers are deduced, they are associated to a set of atoms called evidence. These atoms are the past atoms from the datastream that have been used in deducing the answer. As new dataslices arrive, new hypothetical answers are generated; atoms in hypotheses are moved to evidence if they appear in the dataslice; and hypothetical answers whose hypothesis include atoms with the current time that do not appear are discarded.

3 HARP: Hypothetical Answer Reasoning Program

The microservice implementing the AI of this architecture is called Hypothetical Answer Reasoning Program (HARP). HARP contains an implementation of the reasoning framework of [4] to perform lead deduction from a set of rules 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 [15].

3.1 Specification of Rules

We illustrate the specification of rules by using streams of data that originate from Google Trends. The implementation requires that variables start with an

[2] Rules have to be stratified, i.e., they cannot have circular dependencies [4].

uppercase letter and 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 rules cover three arguments for why a topic should be considered a lead.

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 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 two topics are certain leads and if both topics closely relates to a third topic, then the third topic is a lead derived from other leads:[3]

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

We differentiate between certain leads and derived leads to avoid derived leads being used to derive other derived leads. This would make a topic a lead solely because related topics twice or more removed are trending.
The final rules denote that both certain leads and derived leads are leads.

[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.

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

3.2 User-Defined Predicates (UDPs)

Predicates such as `Less` and `AllDiff` are not practical to specify as rules, but rather algorithmically. Our implementation allows for specifying such *User-Defined Predicates* (UDP). An atom whose predicate is a UDP is called a *User-Defined Atom* (UDA). UDAs in 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()`.

3.3 HARP as a Microservice

HARP allows our implementation of [4] 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 [4] 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 the reasoner to update 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 5.

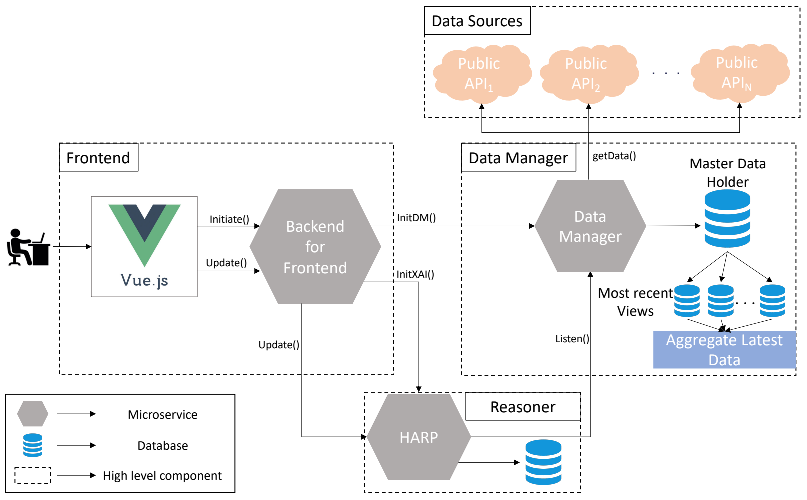


Fig. 1. Architecture overview.

4 Architecture

The overall architecture as shown in Figure 1 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 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.

4.1 Application of Patterns for API Design

The proposed architecture can be analysed wrt the following categories of patterns for API design, as described in [21]:

- 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. In more detail, the Data Manager integrates with:

- The Backend for Frontend, which makes `initDM()` requests of this type to the Data Manager:

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

  A JSON payload example is as follows:

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

  The corresponding response is of the following format:

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

- Public APIs, like Google Trends, with messages of the following format:

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

  An example of JSON payload is the following:

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

  The corresponding 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 */ }
```

  An example of 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, which 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 request example is the following:

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

As far as it 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 utilises two endpoints with one Endpoint Role. Both endpoints are *Processing Resources*, 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. In more detail, HARP integrates with the:

– Backend for Frontend, which 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 }
```

An example of JSON payload is shown below:

```
{ "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 is of the following format:

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

The Backend for Frontend can also make `update()` requests to HARP:

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

An example of JSON payload is the following:

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

Furthermore, HARP gives the following response:

```
type UpdateResponse { answers*: answerElement,
  hypotheticalAnswers*: HypotheticalAnswerElement }
type ActualAnswerElement { answer: string, evidence*: string }
type HypotheticalAnswerElement {
  answer: string, hypothesis*: string, evidence*: string }
```

An example of JSON payload is as follows:

```
{"answers": [
   {"answer": "Lead(detroit lions,dk,1669299502000)",
    "evidence": [
        "DailyTrend(detroit lions,dk,1669299502000)",
        "Popularity(detroit lions,dk,8K,1669299502000)",
        "Popularity(detroit lions,dk,9k,1669303102000)",
        "Popularity(detroit lions,dk,10K,1669306702000),"
        "8K<9k","9k<10k"]},
   {"answer": "Lead(thanksgiving parade,dk,1669299502000)",
    "evidence": [
        "DailyTrend(thanksgiving parade,dk,1669299502000)"
        "Popularity(thanksgiving parade,35K,1669299502000)",
        "Popularity(thanksgiving parade,47K,1669303102000)",
        "Popularity(thanksgiving parade,50K,1669306702000)"
        "35K<47K","47K<50K"]}],
 "hypotheticalAnswers": [
   {"answer": "Lead(detroit lions,dk,1669303102000)",
    "hypothesis": ["10k<Pop2",
        "Popularity(detroit lions,dk,Pop2,1669311402000)"],
    "evidence": [
        "Popularity(detroit lions,dk,9K,1669303102000)",
        "Popularity(detroit lions,dk,10K,1669306702000)",
        "9k<10k"]},
   {"answer": "Lead(thanksgiving parade,dk,1669303102000)",
    "hypothesis": ["50k<Pop2"
        "Popularity(thanksgiving parade,dk,Pop2,1669311402000)"],
    "evidence": ["47k<50k",
        "DailyTrend(thanksgiving parade,dk,1669303102000)",
        "Popularity(thanksgiving parade,dk,47K,1669303102000)",
        "Popularity(thanksgiving parade,dk,50K,1669306702000)"]}] }
```

– Data Manager, which HARP makes `listen()` requests to of the form:

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

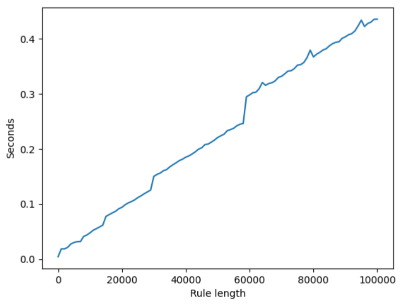
and receives a response like the following:

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

A JSON payload example is given below:

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

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



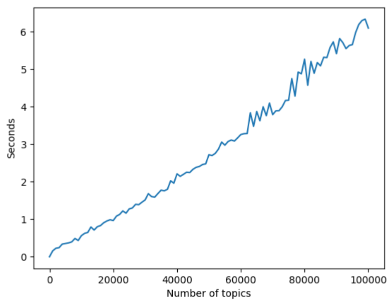


Fig. 2. Left: preprocessing time for a single rule with a body of 0 to 100 000 atoms, at intervals of 1 000, averaged over 10 runs. Right: updating time for a datastream with atoms of the form `DailyTrend(Topic, dk, t)` and `Popularity(Topic, dk, t, t)` for 0 to 100 000 different values of `Topic`, at intervals of 1 000, averaged over 10 runs.

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 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 Fig. 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 paper.) 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 (Fig. 2, left side). The general case is known to be exponential [11].

The time for updating the set of hypothetical answers depends on two factors: the amount of data in the given dataslice and the current number of hypothetical answers. The latter depends on the relevance of the previous data. If data from the datastream matches with atoms in the hypothesis of a hypothetical answer, then more hypothetical answers might be created. The worst-case scenario is that every data matches with atoms in all hypotheses. The execution time of this update has been evaluated for this case, with data size ranging from 0 to

100 000 atoms, with execution time peaking at around 6 seconds (Fig. 2, right side). Increasing the data size to 1 000 000 or larger exceeds the available memory limits of our testing setup, due to the program's current need to have the entire dataslice in memory before updating the hypothetical answers.

Overall, this preliminary 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 the next section.

6 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. These needs oriented us towards the adoption of recent theories and tools, in particular API Patterns [21], the Jolie programming language [15], and hypothetical reasoning over data streams [4]. Thus, our work also serves as a practical validation of these methods.

In this work we have focused on the architectural and technical aspects of μXL. In the future we would like to evaluate the usefulness of μXL for journalists 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. Regarding our reasoner, extending HARP such that it could process dataslices in chunks could be interesting for processing large amounts of data with small amounts of RAM. 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, as described in [4]. 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.

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)
2. Chen, Y., Amiri, H., Li, Z., Chua, T.: Emerging topic detection for organizations from microblogs. In: Procs. SIGIR. pp. 43–52. ACM (2013)

3. Chomicki, J., Imielinski, T.: Temporal deductive databases and infinite objects. In: Prods. SIGMOD. pp. 61–73. ACM (1988)
4. Cruz-Filipe, L., Nunes, I., Gaspar, G.: Hypothetical answers to continuous queries over data streams. In: Procs. AAAI. pp. 2798–2805 (2020)
5. Das, A., Roy, M., Dutta, S., Ghosh, S., Das, A.K.: Predicting trends in the Twitter social network: A machine learning approach. In: Swarm, Evolutionary, and Memetic Computing. Lecture Notes in Computer Science, vol. 8947, pp. 570–581. Springer (2014)
6. Diakopoulos, N., Dong, M., Bronner, L.: Generating location-based news leads for national politics reporting. In: Proc Computational + Journalism Symposium (2020)
7. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer (2017)
8. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S.: Jolie and LEMMA: model-driven engineering and programming languages meet on microservices. In: Procs. COORDINATION. Lecture Notes in Computer Science, vol. 12717, pp. 276–284. Springer (2021)
9. Guidi, C., Maschio, B.: A jolie based platform for speeding-up the digitalization of system integration processes. In: Procs. Microservices (2019)
10. 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 (1999)
11. Kowalski, R.A.: Predicate logic as programming language. In: Procs. IFIP. pp. 569–574. North-Holland (1974)
12. Leppänen, L., Munezero, M., Granroth-Wilding, M., Toivonen, H.: Data-driven news generation for automated journalism. In: Procs. INLG. pp. 188–197. Association for Computational Linguistics (2017)
13. Mathioudakis, M., Koudas, N.: TwitterMonitor: trend detection over the Twitter stream. In: Procs. SIGMOD. pp. 1155–1158. ACM (2010)
14. Montesi, F.: Process-aware web programming with jolie. Sci. Comput. Program. **130**, 69–96 (2016)
15. Montesi, F., Guidi, C., Zavattaro, G.: Service-oriented programming with Jolie. In: Web Services Foundations, pp. 81–107. Springer (2014)
16. Montesi, F., Weber, J.: From the decorator pattern to circuit breakers in microservices. In: Procs. ACM SAC. pp. 1733–1735. ACM (2018)
17. 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: Procs. CHI. pp. 1–13. ACM (2020)
18. Pugachev, A., Voronov, A., Makarov, I.: Prediction of news popularity via keywords extraction and trends tracking. In: Recent Trends in Analysis of Images, Social Networks and Texts. Communications in Computer and Information Science, vol. 1357, pp. 37–51. Springer (2020)
19. Schwartz, R., Naaman, M., Teodoro, R.: Editorial algorithms: Using social media to discover and report local news. In: Procs. ICWSM. pp. 407–415 (2015)
20. Zarrinkalam, F., Fani, H., Bagheri, E., Kahani, M.: Predicting users' future interests on Twitter. In: Advances in Information Retrieval. Lecture Notes in Computer Science, vol. 10193, pp. 464–476. Springer (2017)
21. 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 (2022)