Hey, Lumi! Using Natural Language for Intent-Based Network Management

Arthur S. Jacobs, Ricardo J. Pfitscher, and Rafael H. Ribeiro, *Federal University of Rio Grande do Sul* (*UFRGS*); Ronaldo A. Ferreira, *UFMS;* Lisandro Z. Granville, *Federal University of Rio Grande do Sul* (*UFRGS*); Walter Willinger, *NIKSUN, Inc.;* Sanjay G. Rao, *Purdue University*

https://www.usenix.org/conference/atc21/presentation/jacobs

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

Hey, Lumi! Using Natural Language for Intent-Based Network Management

Arthur S. Jacobs *UFRGS*
Ricardo J. Pfitscher *UFRGS*
Rafael H. Ribeiro *UFRGS*
Ronaldo A. Ferreira *UFMS*
Lisandro Z. Granville *UFRGS*
Walter Willinger *NIKSUN, Inc.*
Sanjay G. Rao *Purdue University*

Abstract

In this work, we ask: what would it take for, say, a campus network operator to tell the network, using natural language, to *"Inspect traffic for the dorm"*? How could the network instantly and correctly translate the request into low-level configuration commands and deploy them in the network to accomplish the job it was "asked" to do? We answer these questions by presenting the design and implementation of LUMI, a new system that *(i)* allows operators to express intents in natural language, *(ii)* uses machine learning and operator feedback to ensure that the translated intents conform with the operator's goals, and *(iii)* compiles and deploys them correctly in the network. As part of LUMI, we rely on an abstraction layer between natural language intents and network configuration commands referred to as *Nile* (Network Intent LanguagE). We evaluate LUMI using synthetic and real campus network policies and show that LUMI extracts entities with high precision and compiles intents in a few milliseconds. We also report on a user study where 88.5% of participants state they would rather use LUMI exclusively or in conjunction with configuration commands.

1 Introduction

Deploying policies in modern enterprise networks poses significant challenges for today's network operators. Since policies typically describe high-level goals or business intents, the operators must perform the complex and error-prone job of breaking each policy down into low-level tasks and deploying them in the physical or virtual devices of interest across the entire network. Recently, *intent-based networking (IBN)* has been proposed to solve this problem by allowing operators to specify high-level policies that express how the network should behave (*e.g.,* defining goals for quality of service, security, and performance) without having to worry about how the network is programmed to achieve the desired goals [17]. Ideally, IBN should enable an operator to simply tell the network to, for example, *"Inspect traffic for the dorm"*, with the network instantly and correctly breaking down such an intent into configurations and deploying them in the network.

In its current form, IBN has not yet delivered on its promise of fast, automated, and reliable policy deployment. One of the main reasons for this shortcoming is that, while network policies are generally documented in natural language, we cannot currently use them as input to intent-based management systems. Despite growing interest from some of the largest tech companies [7, 47, 62] and service providers [32, 38, 52], only a few research efforts [4, 13] have exploited the use of natural language to interact with the network, but they lack support for IBN or other crucial features (*e.g.,* operator confirmation and feedback). However, expressing intents directly in natural language has numerous benefits when it comes to network policy deployment. For one, it avoids the many pitfalls of traditional policy deployment approaches, such as being forced to learn new network programming languages and vendor-specific Command-Line Interfaces (CLI), or introducing human errors while manually breaking down policies into configuration commands. At the same time, its appeal also derives from the fact that it allows operators to express the same intent using different phrasings. However, the flexibility makes it challenging to generate configurations, which must capture operator intent in an unambiguous and accurate manner.

In this paper, we contribute to the ongoing IBN efforts by describing the design and implementation of LUMI, a new system that enables an operator "to talk to the network", focusing on campus networks as a use case. That is, LUMI takes as input an operator's intent expressed in natural language, correctly translates these natural language utterances into configuration commands, and deploys the latter in the network to carry out the operator's intent. We designed LUMI in a modular fashion, with the different modules performing, in order: information extraction, intent assembly, intent confirmation, and intent compilation and deployment. Our modular design allows for easy plug-and-play, where existing modules can be replaced with alternative solutions, or new modules can be included. As a result, LUMI's architecture is extensible and evolvable and can easily accommodate further improvements or enhancements.

In addressing the various challenges above, we make the following contributions:

Information extraction and confirmation. We build on existing machine learning algorithms for *Named Entity Recognition (NER)* [30] to extract and label entities from the operator's natural language utterances. In particular, we implement NER using a chatbot-like interface with multi-platform support (§3) and augment the existing algorithm so that LUMI can learn from operator-provided feedback (§5).

Intent assembly and compilation. We introduce the *Network Intent Language* (Nile), use it as an abstraction layer between natural language intents and network configuration commands for LUMI, and illustrate its ability to account for features critical to network management such as rate-limiting or usage quotas (§4). We also show how LUMI enables compilations of *Nile* intents to existing network programming languages (§6), such as *Merlin* [57] .

Evaluation. We evaluate (§8) LUMI's accuracy in information extraction, investigate LUMI's ability to learn from operator-provided feedback and measure both the compilation and deployment times in a standard campus topology [5]. Using our own datasets consisting of synthesized intents as well as real-world intents derived from network policies published by 50 different campus networks in the US, we show that LUMI can extract entities with high precision, learn from the feedback provided by the operator, and compile and deploy intents in less than a second.

User study. In addition to an in-depth evaluation of LUMI, we also report our main findings of a small-scale user study, with 26 subjects (§9). The study was performed to get feedback from subjects on the perceived value of using natural language for network management with LUMI and soliciting user feedback during the intent confirmation stage.

Prototype. We implemented our prototype of LUMI using a combination of tools and libraries (*e.g.,* Google Dialogflow [28], *Scikit-learn* library [49]). The full implementation as well as all datasets used in our evaluation are available on the project's website [39].

Together, the results of our evaluation and user study show that LUMI is a promising step towards realizing the vision of IBN of achieving fast, automated, and reliable policy deployment. By allowing operators to express intents in natural language, LUMI makes it possible for operators to simply talk to their network and tell it what to do, thus simplifying the jobs of network operators (*i.e.,* deploying policies) and also saving them time. While promising, developing LUMI into a full-fledged production-ready system poses new and interesting challenges on the interface of networking and NLP, which we detail in §10.

2 Lumi in a Nutshell

Figure 1 illustrates the high-level goal of LUMI with the intent example *"Hey, Lumi! Inspect traffic for the dorm"* and shows the breakdown of the workflow by which LUMI accomplishes

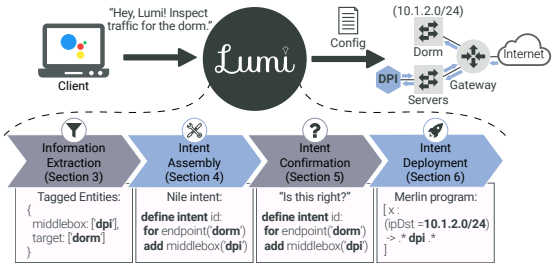


Figure 1: The four modules of LUMI.

the stated objective. Below, we provide a brief overview of the four key components that define this workflow (*i.e.,* the LUMI pipeline) and refer to the subsequent sections for a more in-depth description and design choices of each of these modules.

First, for the Information Extraction module (described in Section 3), we rely on machine learning to extract and label entities from the operator utterances and implement them using a chatbot-like conversational interface. The extracted entities form the input of the Intent Assembly module (described in Section 4), where they are used to compose a *Nile* network intent. *Nile* closely resembles natural language, acts as an abstraction layer, and reduces the need for operators to learn new policy languages for different types of networks. Then, as part of the Intent Confirmation module (described in Section 5), the output of the Intent Assembly module (*i.e.,* a syntactically-correct *Nile* intent) is presented to the network operator, and their feedback is solicited. If the feedback is negative, the system and the operator iterate until confirmation, with the system continuously learning from the operator's feedback to improve the accuracy of information labeling over time. Finally, once the system receives confirmation from the operator, the confirmed *Nile* intent is forwarded to the Intent Deployment module. Described in Section 6, this module's main task is to compile *Nile* intents into network configuration commands expressed in *Merlin* and deploy them in the network. In Section 6, we also explain why we picked *Merlin* as a target language over other alternatives.

3 Information Extraction

The main building blocks for LUMI's Information Extraction module are a chatbot interface as the entry point into our system and the use of Named Entity Recognition (NER) [30] to extract and label entities from the operators' natural language intents. Given the popularity of personal assistants, such as Google Assistant, Amazon's Alexa or Apple's Siri, our goal in providing a natural language interface for LUMI goes beyond facilitating the lives of traditional network operators and seeks to also empower users with little-to-no knowledge of how to control their networks (*e.g.,* home users).

Even in the case of the traditional user base of network operators, providing a natural language interface to interact with the system benefits teams composed of operators with

different levels of expertise or experience. This type of interface is particularly relevant in campus or enterprise networks with small groups and in developing countries where network teams are often understaffed and lack technical expertise. In short, while deploying a policy as simple as redirecting specific traffic for inspection can be a daunting task for an inexperienced operator, nothing is intimidating about expressing that same policy in natural language and letting the system worry about its deployment, possibly across multiple devices in the network.

Solving the NER problem typically involves applying machine learning (for extracting named entities in unstructured text) in conjunction with using a probabilistic graphical model (for labeling the identified entities with their types). Even though, in theory, NER is largely believed to be a solved problem [43], in practice, to ensure that NER achieves its purpose with acceptable accuracy, some challenges remain, including careful "entity engineering" (*i.e.,* selecting entities appropriate for the problem at hand) and a general lack of tagged or labeled training data.

Below, we first discuss the entity engineering problem and propose a practical solution in the form of a hierarchically-structured set of entities. Next, we describe in more detail the different steps that the NER process performs to extract named entities from a natural language intent such as *"Inspect traffic for the dorm"* and label them with their types. Finally, to deal with the problem caused by a lack of labeled training data, we describe our approach that improves upon commonly-used NER implementations by incorporating user feedback to enable LUMI to learn over time.

Table 1: Hierarchical set of entities defined in Lumi.

Type	Entity Class
Common	@middlebox, @location, @group, @traffic, @protocol, @service, @qos_constraint, @qos_metric, @qos_unit, @qos_value, @date, @datetime, @hour
Composite	@origin, @destination, @target @start, @end
Immutable	@operation, @entity

3.1 Entity Selection

To ensure that NER performs well in practice, a critical aspect of specifying the underlying model is entity engineering; that is, defining which and how many entities to label in domain application-specific but otherwise unstructured text. On the one hand, since our goal with LUMI is to allow operators to change network configurations and manipulate the network's traffic, the set of selected entities will affect which operations are supported by LUMI. As discussed in more detail in Section 4, LUMI-supported actions are dictated almost entirely by what intent constructions *Nile* supports. At the same time, we would like to consider a generic enough set of entities to enable users to express their intents freely, independent of *Nile*. Moreover, the selected set of entities should also allow for easy expansion (*e.g.,* through user feedback; see Section 3.4 below) while ensuring that newly added entities neither introduce ambiguities nor result in unforeseen entries that might break the training model.

On the other hand, the selected set of entities directly influences the trained model's accuracy, especially if the entities have been chosen poorly. Therefore, it is crucial to pick a set of entities that is at the same time rich enough to solve the task at hand and concise enough to avoid ambiguities that might hamper the learning process. For instance, one common source of uncertainty in network intents is highlighted with the two examples *"Block traffic from YouTube"* and *"Block traffic from the dorms"*. Here, the word 'from' appears in both intents but is used for two different purposes. While in the first example, it specifies a service, in the second example, it defines a location. If we choose to tag both entities (*i.e.,* "YouTube" and "dorms") with the same entity type (*e.g.,* "location") to avoid ambiguities, we lose information on what is being tagged (*i.e.,* service vs. location). However, if we simply use different entities for both cases (*e.g.,* "service" and "location"), we generate an ambiguity (*e.g.,* very similar phrasings produce entirely different results) that causes the accuracy of the NER model to decrease as similar example intents are encountered (*e.g., "Block traffic from Twitter"*).

With these design requirements in mind, we defined the set of LUMI entities hierarchically and organized them into three different categories: common, composite, and immutable entities (see Table 1). Here, common entities form the bottom of the hierarchy, comprise raw textual values, and largely determine what LUMI can understand. For instance, the textual values in the common entity class @*middlebox* are network functions such as firewalls, packet inspection, and traffic shaping. The hierarchy's intermediate level consists of composite entities. The entities in this class do not have any inherent nouns, verbs, or even synonyms associated with them; they only establish a relationship between common entities through the aggregation of prepositions. For instance, the composite entity class @*origin* consists of composite values such as "from @*location*" and "from @*service*". Composite entities help avoid the ambiguity problem mentioned earlier. Finally, immutable entities make up the top of the hierarchy and form the core of LUMI. In particular, while the entity class @*operation* expresses the operations that *Nile* supports, the entity class @*entity* consists of a list of LUMI-supported common entities.

3.2 Entity Encoding: Bi-LSTMs

Figure 2 shows the overall NER architecture that we use in LUMI and illustrates the critical intuition behind NER; that is, finding named entities in unstructured text (*e.g.,* "Inspect traffic for the dorm" in Step 1 in Figure 2) and labeling them

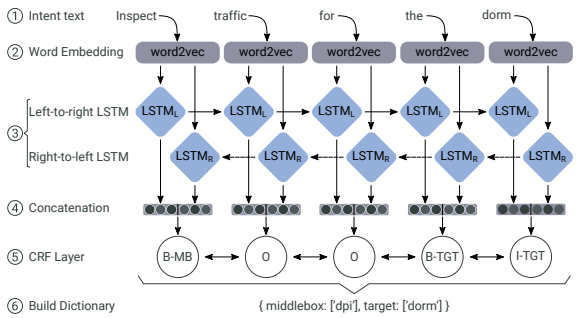


Figure 2: The NER architecture with Bi-LSTM (see Section 3.2) and CRF (see Section 3.3). The entity tags are abbreviated as MB for middlebox and TGT for target.

with their types (*e.g.,* Step 6 in Figure 2). With respect to the machine learning part of the NER problem, standard approaches leverage Recurrent Neural Networks (RNN) [34]; *i.e.,* a family of neural networks that processes arrays of data sequentially, where the output of one element is carried over as input to the next element in the sequence. However, RNNs typically require numerical inputs and not words. Therefore, when applied in the context of our text processing problem, each word of an intent that the operator expresses in natural language has to be first encoded into a numerical vector before an RNN can process it. Rather than using one-hot encoding [30], a simple encoding scheme that represents each encoded word as a vector of 0's and 1's, we rely in Step 2 (see Figure 2) on a more powerful approach that uses one-hot encoded vectors of words as input to a word embedding algorithm known as *word2vec* [44]. The *word2vec* algorithm uses a pre-trained mini-neural network that learns the vocabulary of a given language (English, in our case) and outputs a vector of decimal numbers for each word. As a neural network, *word2vec* can provide similar vector representations for synonyms, so that words with similar meanings (*e.g.,* 'dormitories' and 'housing') have similar embedded vectors, which in turn allows an RNN to process them similarly.

Before processing the output of *word2vec*, we need to specify the type of RNN model in our architecture. The Long Short-Term Memory (LSTM) model has been a popular choice for text processing [25] as it can capture and carry over dependencies between words in a phrase or intent (*e.g.,* to identify multi-word expressions). It also creates a context-full representation of each processed word as an encoded array of numerical values. However, to further enhance each word's context, in our LUMI design of the information extraction stage, we rely in Step 3 on an enhanced version of LSTM, the so-called Bi-LSTM model [16, 36]. The Bi-LSTM approach yields the best results for the English language in a majority of evaluated cases [63]. In a Bi-LSTM, a phrase is evaluated by two LSTMs simultaneously, from left-to-right and from right-to-left, and the outputs of the two LSTMs are then concatenated to produce a single output layer at a given position, as shown in Step 4.

3.3 Entity Labeling: CRFs

The labeling part of the NER problem consists of using the context-full encoded vectors of words to represent the "observed" variables for a type of probabilistic graphical models known as Conditional Random Fields (CRFs) [33], as shown in Step 5. CRFs are a widely-used technique for labeling data, especially sequential data arising in natural language processing. Their aim is to determine the conditional distribution $P(Y|X)$, where $X = \{x_1, x_2, ..., x_n\}$ represents a sequence of observations (*i.e.,* encoded vectors of words in a sentence) and $Y = \{y_1, y_2, ..., y_m\}$ represents the "hidden" or unknown variable (*i.e.,* NER tags or labels) that needs to be inferred given the observations. CRFs admit efficient algorithms for learning the conditional distributions from some corpus of training data (*i.e.,* model training), computing the probability of a given label sequence Y given observations X (*i.e.,* decoding), and determining the most likely label sequence Y given X (*i.e.,* inference).

The technical aspects of the specific CRF model we use in this work are described in detail in LUMI's website [39] and show the flexibility afforded by CRF models to account for domain-specific aspects of labeling sequential data (*e.g.,* accounting for the likelihood of one entity label being succeeded by another). However, irrespective of the specific CRF model used, the method outputs as the most likely tag for a given word the one with the highest probability among all tags. Specifically, at the end of Step 5, the result of the NER algorithm is provided in the form of named entities with IOB tagging [30]. In IOB tagging, each named entity tag is marked with an indicator that specifies if the word is the beginning (B) of an entity type or inside (I) of an entity type. If a word does not match any entity, then the algorithm outputs an (O) indicator (for "outside"). Note that by using IOB tagging, it is possible to distinguish if two side-by-side words represent a single entity or two completely different entities. For instance, without IOB-tagging, it would be impossible to tag an operation like "rate limiting" as one single named entity. Finally, we parse the IOB-tagged words resulting from the NER model, build a dictionary with the identified entities, and output it in Step 6 as the final result of LUMI's Information Extraction module.

3.4 NER and Learning

The described NER process is an instance of a supervised learning algorithm. It uses a corpus of training data in the form of input-output examples where input is an intent (*i.e.,* phrase with named entities defined in LUMI and other words or non-entities), and an output is the list of named entities with IOB tagging (*i.e.,* correct entity tags or labels). The primary training step consists of both adapting the weights of the Bi-LSTM model to extract the desired X vector and re-calculating the conditional distribution $P(X|Y)$ to infer the correct NER tags Y and may have to be repeated until convergence (*i.e.,* Steps 3-5).

Note that retraining can be done each time the existing corpus of training data is augmented with new key-value pairs or with existing entities used in a novel context (*i.e.,* requiring a new tag). A basic mechanism for obtaining such new key-value pairs or for benefiting from the use of existing entities in a novel context is to engage users of LUMI and entice their feedback in real-time, especially if this feedback is negative and points to missing or incorrect pieces of information in the existing training data. By enticing and incorporating such user-provided feedback as part of the Intent Confirmation stage (see Section 5 for more details), LUMI's design leverages readily available user expertise as a critical resource for constantly augmenting and updating its existing training data set with new and correctly-labeled training examples that are otherwise difficult to generate or obtain. After each new set of key-value pairs is obtained through user feedback, LUMI immediately augments the training corpus and retrains the NER model from scratch.

In light of the hierarchical structure of our LUMI-specific entities set, with user-provided feedback, we aim to discover and learn any newly-encountered common entities. Moreover, as the data set of common entities is augmented with new training instances, the accuracy of identifying composite entities also improves. With respect to the immutable entities, since the entity class *@operation* dictates what operations LUMI supports and understands, these operations cannot be learned from user-provided feedback. However, given the limited set of LUMI-supported operations, a relatively small training dataset should suffice to cover most natural language utterances that express these operations. As for the entity class *@entity*, being composed of a list of LUMI-supported common entities, it ensures that user-provided feedback can be correctly labeled.

4 Intent Assembly

Using a chatbot interface with NER capabilities as the front end of LUMI solves only part of the network intent refinement and deployment problem. For example, if a network operator asks a chatbot *"Please add a firewall for the backend."*, the extraction result could be the following entities: *{middleboxes: 'firewall'}*, *{target: 'backend'}*. Clearly, these two key-value pairs do not translate immediately to network configuration commands. Assembling the extracted entities into a structured and well-defined intent that can be interpreted and checked for correctness by the operator before being deployed in the network calls for introducing an abstraction layer between natural language intents and network configuration demands.

To achieve its intended purpose as part of LUMI, this abstraction layer has to satisfy three key requirements. First, the operations supported by the abstraction layer's grammar have to specify what entities to extract from natural language intents. Instead of trying to parse and piece together every possible network configuration from user utterances, we require that a predefined set of operations supported by the abstraction layer's grammar guide the information extraction process. As a result, when processing the input text corresponding to a network intent expressed by the operator in natural language, we only have to look for entities that have a matching operation in the abstraction layer's grammar, as those are the only ones that the system can translate into network configurations and subsequently act upon.

A second requirement for this abstraction layer is to allow for easy confirmation of the extracted intents by the operators by having a high level of legibility. We note that requiring confirmation does not burden the operators in the same manner that requiring them to express their network intents directly in the abstraction layer's language would. For instance, it is well-known that a person who can understand a written sentence cannot necessarily create it from scratch (*i.e.,* lacking knowledge of the necessary grammar).

Finally, the abstraction layer is also required to allow different network back-ends as targets given that a wide range of candidate network programming languages [6, 11, 21, 31, 46, 54, 57] exist, and none of them seem to have been favored more by operators or industry yet. Using an abstraction layer on top of any existing languages allows us to decouple LUMI from the underlying technology, so we can easily change it if a better option arises in the future.

4.1 *Nile*: Network Intent Language

To satisfy the above requirements, in our design of the Intent Assembly module (*i.e.,* stage two of the LUMI pipeline), we rely on the Network Intent Language (*Nile*) to serve as our abstraction layer language. In a previous work [29], we proposed an initial version of *Nile* that provided minimal operation support for intent definition and focused primarily on service-chaining capabilities. Here, we extend this original version to cover crucial features for network management in real-world environments (e.g., usage quotas and rate-limiting). Closely resembling natural language, the extended version of *Nile* has a high level of legibility, reduces the need for operators to learn new policy languages for different types of networks, and supports different configuration commands in heterogeneous network environments. Operationally, we designed this module to ingest as input the output of the information extraction module (*i.e.,* entities extracted from the operator's utterances), assemble this unstructured extracted information into syntactically-correct *Nile* intents, and then output them.

Table 2: Overview of *Nile*-supported operations.

Operation	Function	Required	Policy Type
from/to	endpoint	Yes	All
for	group/endpoint/service/traffic	Yes	All
allow/block	traffic/service/protocol	No	ACL
set/unset	quota/bandwidth	No	QoS
add/remove	middlebox	No	Service Chaining
start/end	hour/date/datetime	No	Temporal

Table 2 shows the main operations supported by our extended version of *Nile*, and the full grammar of *Nile* is made available in LUMI's website [39]. Some of the operations have opposites (*e.g.,* allow/block) to undo previously deployed intents and enable capturing incremental behaviors stated by the operators. Some operations in a *Nile* intent are mandatory, such as **from/to** or **for**. More specifically, an operator cannot write an intent in *Nile* without stating a clear target (using **for**) or an origin and a destination (using **from/to**) when the direction of the specified traffic flow matters.

To enforce the correct syntax of the assembled intent, we leverage the *Nile* grammar to guarantee that the intent contains the required information for correct deployment. If the grammar enforcement fails due to the lack of information, the system prompts the operator via the chatbot interface to provide the missing information. Assume, for example, that the operator's original intent stated *"Please add a firewall."*, without providing the target of the intent. Since specifying a target is required according to the *Nile* grammar, the module will not attempt to construct a Nile program but will instead interact with the operator to obtain the missing information.

4.2 *Nile* Intents: An Example

With *Nile*, we can express complex intents intuitively. For example, an input like "*Add firewall and intrusion detection from the gateway to the backend for client B with at least 100mbps of bandwidth, and allow HTTPS only*" is translated to the *Nile* intent shown in Listing 1. The **group** function is used as a high-level abstraction for clients, groups of IP addresses, VLANs, or any other aggregating capacity in low-level network configurations. Note that the IDs provided by the operator must be resolved during the compilation process, as they represent information specific to each network. This feature of the language enhances its flexibility for designing intents and serving as an abstraction layer. The example illustrates how *Nile* provides a high-level abstraction for structured intents and suggests that the grammar for *Nile* is expressive enough to represent many real-world network intents.

```
define intent qosIntent:
  from   endpoint('gateway')
  to     endpoint('database')
  for    group('B')
  add    middlebox('firewall'), middlebox('ids')
  set    bandwidth('min', '100', 'mbps')
  allow  traffic('https')
```

Listing 1: *Nile* intent example.

5 Intent Confirmation (Feedback)

The major challenge with using natural language to operate networks (*e.g.,* as part of IBN) is that the method is inherently ambiguous. There are many different forms in which an operator can express the same intent in different network environments in natural language. Despite recent advances, natural language processing is prone to producing false positives or false negatives, resulting in incorrect entity tagging, leading to deploying incorrect network configuration commands. However, to be of practical use, network configurations are required to be unquestionably unambiguous and correct-by-construction.

One approach to address this challenge is to create an extensive dataset with training phrases and entities. However, it is unrealistic to expect that such a dataset will cover every possible English language (or any other language for that matter) example of phrase construction with domain-specific entities. Operators are free to use terms or expressions that the system has never encountered in the past. However, without proper safeguards, any such new phrase will likely result in misconfigurations of the network. An alternative approach to implementing a reliable intent deployment process is through "learning from the operator." Here, the basic idea is to leverage the operators' existing knowledge by requesting their feedback on the information extracted from natural language intents. Then, in the case of negative feedback, it is critical to engage with the operators to identify missing or incorrect pieces, include this such acquired new knowledge as additional phrases or entities in the original training dataset, and retrain the original learning model.

Our solution to deal with the ambiguity inherent in using natural language to express network intents follows the learning approach and leverages the chatbot interface implemented as part of our first module. In particular, in designing the intent confirmation module for realizing stage three of the four-stage LUMI pipeline, we require the output of the intent assembly module (*i.e.,* syntactically-correct *Nile* intents) to be confirmed by the operator. When presented with assembled intents that result in false positives or negatives, the operator is asked to provide feedback that we subsequently use to augment the original training dataset with new labeled data; that is, LUMI is capable of learning new constructs over time, gradually reducing the chances of making mistakes. While this interaction may slow down initial intent deployments until LUMI adapts to the operator's usage domain, it is essential to guarantee reliable intent refinement and deployment.

Extracting pertinent information from user feedback also requires identifying specific entities in the user text, similarly to extracting entities from an input network intent. To this end, LUMI uses the same NER model for both tasks, relying primarily on the immutable *@entity* for extracting which entity class is being augmented and what value is being added. Relying on the same NER model also requires us to train the model to identify and extract entities in the received user feedback. However, since we limit LUMI's interactions with the user to answering simple questions, processing user feedback does not require a large set of training samples.

To reduce an operator's need for technical knowledge during this intent confirmation stage, we opted for supporting feedback interactions that induce the operator to provide the information that LUMI needs (*i.e.,* offering suggestions and asking complementary questions). Also, to provide operators

with more flexibility and not insist that they have to use specific keywords, this module compares the entities and values provided by the operators with synonyms already in the training dataset. Figure 3 illustrates a case where, due to a lack of training, LUMI misses an entity in the input text, and the confirmation mechanism lets operators easily catch this mistake and provide the necessary feedback for LUMI to learn. Also, note that our design of this module is conservative in the sense that operator feedback is requested for each assembled intent, irrespective of the level of confidence that LUMI has concerning its accuracy.

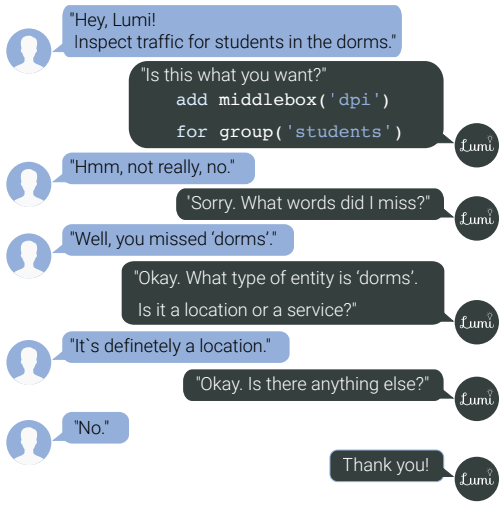


Figure 3: LUMI's feedback mechanism in action.

6 Intent Deployment

The fourth and last stage of LUMI compiles the operator-confirmed *Nile* intents into code that can be deployed on the appropriate network devices and executes the original network intent expressed by the operator in natural language. Fortunately, the abstraction layer provided by *Nile* enables compilations to a number of different existing network configurations, including other policy abstractions languages such as *Merlin* [57], *OpenConfig* [48], *Janus* [2], *PGA* [51], and *Kinetic* [31].

For our design of LUMI's Intent Deployment module, we chose to compile structured *Nile* intents into *Merlin* programs. We picked *Merlin* over other alternative frameworks because of its good fit with *Nile*, the network features it supports, its performance, and the availability of source code. Also, given that none of the existing network programming languages natively supports all features proposed in *Nile*, we opted for the one that had the largest intersection of supported operations. In the process, we mapped each *Nile* feature to a corresponding *Merlin* feature.

Resolving logical handles. Logical handles in *Nile* intents are decoupled from low-level IP addresses, VLAN IDs and IP prefixes, which LUMI now resolves (e.g., `dorm` →`10.1.2.0/24`) using information provided during the bootstrap process. ACLs rules are resolved similarly. Once LUMI produces *Merlin* programs with resolved identifiers (*i.e.,* VLAN IDs, IPs and prefixes), compilation to corresponding OpenFlow rules is handled by *Merlin*.

Temporal constraints and QoS. As *Merlin* does not support temporal policies, LUMI stores every confirmed *Nile* intent so that it can install or remove device configurations according to times and dates defined by the operator. We achieve quota restrictions (not natively supported by *Merlin*) by relaying all traffic marked with a quota requirement to a traffic-shaping middlebox, taking advantage of *Merlin*'s support for middlebox chaining. Other QoS policies, such as rate-limiting, are already supported in *Merlin*

Middlebox chaining. LUMI focuses on middlebox chaining, i.e., correctly relaying traffic specified in the intents through a specified middlebox. Since the actual configuration of each middlebox is currently done outside of LUMI, LUMI can handle chaining policies associated with *any* middlebox type, virtual or physical, compilation for which is straightforward since *Merlin* natively supports middlebox chaining.

7 Implementation

We implemented LUMI's prototype using different tools and libraries for each module of the LUMI-pipeline. For the chatbot interface and NER, we used Google Dialogflow [28] because of its conversational interface and multi-platform integration. However, these stages can be easily implemented with other open-source NLP frameworks (*e.g.,* SpaCy [26] and Stanford CoreNLP [41]) or machine learning toolkits such as Scikit-learn [49] and Keras [18]. We exported the Dialogflow implementation of LUMI as JSON files and uploaded them to GitHub [39], so other researchers can build on our work and reproduce our results.

We implemented the WebHook service that Dialogflow calls with the extracted entities in Python. This service builds the *Nile* intents and interacts with the chatbot interface if necessary. We implemented the compilation of *Nile* programs into *Merlin* programs and the deployment of the resulting *Merlin* program as a separate Python RestAPI service that the previous WebHook service calls after the operator confirms that the intent is correct. We developed this module as a separate service so that it can be deployed on a local server with access to the network controller. All the other modules can be deployed on a cloud server. The full implementation of LUMI, comprising over 5,451 lines of Python code and 1,079 lines of JavaScript and HTML, a working demo, and all datasets used in the course of this work are publicly available [39].

8 Evaluation

In this section, we first evaluate the accuracy of our Information Extraction module to show that LUMI extracts entities with high precision and learns from operator-provided feedback. We then show that LUMI can quickly compile *Nile* intents into Merlin programs and deploy them. To assess the accuracy of the Information Extraction module, we use the standard metrics Precision, Recall, and F1-score. For the

evaluation of the Intent Deployment stage, we measure the compilation time for translating *Nile* intents into Merlin statements and their deployment time.

8.1 Information Extraction

Evaluating systems like LUMI is challenging because of (i) a general lack of publicly available datasets that are suitable for this problem space (several operators we contacted in industry and academia gave proprietary reasons for not sharing such data), and (ii) difficulties in generating synthetic datasets that reflect the inherent ambiguities of real-world Natural Language Intents (NLIs).

To deal with this problem, we created two hand-annotated datasets for information extraction. The dataset *alpha* is "semi-realistic" in the sense that it is hand-crafted, consisting of 150 examples of network intents that we generated by emulating an actual operator giving commands to LUMI. In contrast, the *campi* dataset consists of real-world intents we obtained by crawling the websites of 50 US universities, manually parsing the publicly available documents that contained policies for operating their campus networks, and finally extracting one-phrase intents from the encountered public policies. From those 50 universities, we were able to extract a total of 50 different network intents. While some universities did not yield any intents, most universities published network policies related to usage quotas, rate-limiting, and ACL, and we were able to express all of them as *Nile* intents. We manually tagged the entities in each of these 200 intents to train and validate our information extraction model.

We used both datasets, separately and combined, to evaluate our NER model, with a 75%-25% training-testing random split. The small size of each dataset precludes us from performing conventional cross-validation. Table 3 shows the results for the *alpha* dataset, for the *campi* dataset, and for a combination of the two and illustrates the high accuracy of LUMI's information extraction module. Given the way we created the training examples for the *alpha* dataset, the excellent performance in terms of Precision, Recall, and F1-score is reassuring but not surprising. In creating the intent examples, we paid special attention to extracting all the entities defined in LUMI (see Section 3.1) and also creating multiple intents for each entity class.

Table 3: Information extraction evaluation using the *alpha* and *campi* dataset.

Dataset	# of Entries	Precision	Recall	F1
alpha	150	0.996	**0.987**	**0.991**
campi	50	**1**	0.979	0.989
alpha + *campi*	200	0.992	0.969	0.980

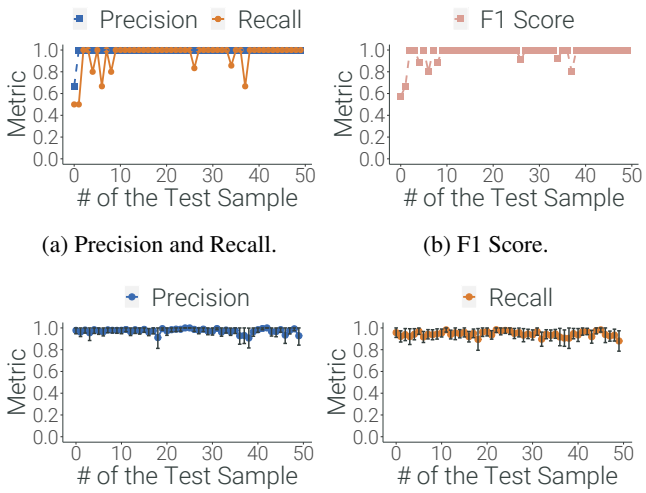
At the same time, despite the largely unstructured nature and smaller number of intent examples in the *campi* dataset, the results for that dataset confirm the above observation. Even though the example intents in this case were not designed with the NER model in mind, LUMI's performance remains excellent and is essentially insensitive to the differences in how the intent examples were generated. We attribute this success of LUMI at the information extraction stage to both continued advances in using machine learning for natural language processing and the fact that the complete set of LUMI-defined entities is relatively small and at the same time sufficiently expressive.

8.2 Intent Confirmation and Feedback

To evaluate the impact of operator-provided feedback on LUMI's ability to learn, we first trained our NER model using 75% of the combined *alpha* and *campi* datasets (*i.e.,* a total of 150 training examples) and then used the remaining 25% of examples (*i.e.,* a total of 50 test entries) as new intents that we presented LUMI in random order. We fed each of the 50 test intents into the NER model for information extraction and evaluated the Precision and Recall on a case-by-case basis. If a new intent generated False Positives or False Negatives, we inserted the intent into NER's existing training dataset, alongside the pre-tagged entities, mimicking the operator's feedback given during the Intent Confirmation stage.

The results for this experiment (Precision, Recall and F1-score) are shown in Figures 4a and 4b. Since each point in the plots represents the Precision/Recall for one specific test sample rather than for the global model, the depicted values fluctuate as test samples are checked one after another. As can be seen, while Precision quickly reaches and then stays at 1.0, the Recall metric dips each time the model encounters an entity or construct it has not yet seen (*i.e.,* resulting in a False Negative). However, as more of these examples become part of NER's training data through the feedback mechanism (and because of re-training of the model after each new example is added), these dips become less frequent. Out of the 50 test intents, only eight resulted in a dip in Recall; that is, were used as feedback. Note, however, that the cases where the model does not identify an entity are precisely the situations where feedback is most informative and enables the model to learn for the benefit of the users.

To assess how often a user has to rely on the feedback mechanism, we repeated our experiment 30 times, each time with a different 75-25 traininig-testing split. The resulting mean values for Precision and Recall are shown in Figures 4c and 4d, with corresponding 95% confidence intervals. As expected, over the 30 repetitions, both Precision and Recall remain close to 0.99, with very small fluctuations. And just as in the previous experiment, whenever there is a significant variation in Precision or Recall (*i.e.,* large confidence intervals), we added that particular intent example to NER's latest training dataset and retrained the model. We attribute the fact that about 20% of the test examples were used as feedback to the small size of our training dataset, but argue that having only this many feedbacks is a positive outcome.



(a) Precision and Recall. (b) F1 Score.

(c) Precision. (d) Recall.

Figure 4: Feedback impact on information extraction.

8.3 Intent Deployment

To evaluate the deployment capabilities of LUMI, we compile and deploy five categories of *Nile* intents, with an increasing level of complexity: middlebox chaining, ACL, QoS, temporal, and intents with mixed operations. The last category of intents mixes *Nile* operations with distinct goals, which we use to evaluate the deployment of more complex intents. We generated a dataset with 30 intents per category, totaling 150 different intents, and measured the mean compilation and deployment time of each category. We ran this experiment on a generic campus network, with approximately 180 network elements. We relied on the Mininet [35] emulator to perform the experiments. The results are given in Table 4. While deployment time necessarily depends on the network environment, in our setting, we consistently measured sub-second combined compilation and deployment times.

Table 4: Compilation and deployment time for five categories of *Nile* intents.

Intent Type	Compilation Time (ms)	Deployment Time (ms)
Middlebox chaining	4.402	110
ACL	3.115	112
QoS	3.113	136
Temporal	4.504	111
Mixed	4.621	1030

9 User Study

To evaluate LUMI's ability to work in a real-world environment rather than with curated datasets of intents, we designed and carried out a small-scale user study. Specifically, we wanted to assess three critical aspects of our system: *(i)* How well does the information extraction process work with actual humans describing their intents in different forms and phrasings? *(ii)* How often is it necessary for operators to provide feedback for LUMI while using the system? and *(iii)* Compared to existing alternative methods, what is the perceived value of a system like LUMI that leverages natural language for real-time network management?

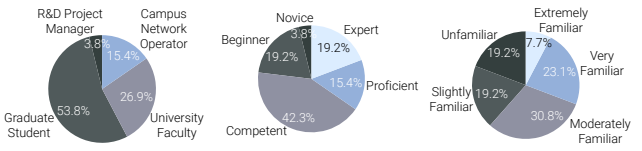
In this section, we describe the experiments we conducted, the participants' profiles, and the obtained results. We set up the user study as an online experiment that users could access and participate in anonymously. To select participants for the user study from different technical backgrounds and still keep their anonymity, we distributed a link to the online user study in mailing lists of both networking research groups and campus network operators. According to the guidelines of our affiliated institution, due to the fully anonymous nature of the experiment, no IRB approval was required to conduct this study, so this work does not raise any ethical issues.

9.1 Methodology

Participating users were asked to fill out a pre-questionnaire (*e.g.,* level of expertise, degree of familiarity with policy deployment, and use of chatbot-like interfaces) and then take on the role of a campus network operator by performing five specific network management tasks using LUMI. Based on the information these users provided in their pre-questionnaire, we had participants from three different continents: the Americas (88.5%), Europe (7.7%), and Asia (3.8%).

Each of the tasks required the user to enforce a specific type of network policy: *(i)* reroute traffic from the internet to an inspection middlebox; *(ii)* limit the bandwidth of guest users using torrent applications; *(iii)* establish a data usage quota for students in the university dorms; *(iv)* block a specific website for students in the labs, and *(v)* add a daily temporal bandwidth throttle to the server racks from 4pm to 7pm. Every interaction users had with LUMI was logged to a database for post-analysis.

After finishing the tasks, users were asked to complete a post-questionnaire (*e.g.,* number of completed tasks, the perceived value of LUMI, and comments on its usability). The complete set of management tasks presented to the users and all the results are available on the LUMI's website. Out of the 30 participants, four did not complete the online questionnaires and were excluded from the study, leaving a total of 26 subjects. Figure 5 shows a breakdown of the user profiles by type of job, level of experience with network management, and familiarity with chatbot-like interfaces.



(a) Job. (b) Expertise. (c) Familiarity.

Figure 5: Subjects profiles.

9.2 Information Extraction and Feedback

To assess the accuracy of our Information Extraction module, we use the number of tasks each participant concluded. For

completing any given task, a specific set of labeled entities was required to build the correct *Nile* intent. Hence, each user's number of completed tasks reflects how accurately LUMI identified the entities in the input texts. The results in the left part of Figure 6 show that most users completed either 5/5 or 4/5 tasks. Some examples of successful intents for each task can be found on LUMI's website [39]. An analysis of the users' interactions with the system revealed that LUMI had trouble understanding temporal behavior (*e.g.,* "from 4pm to 7pm"), likely due to a lack of such training examples. This issue prevented some users from completing all five tasks. One user could not complete any task, reportedly because of an outage in the cloud provider infrastructure used to host LUMI.

(a) Tasks completed by users. (b) User Intents by status.

Figure 6: LUMI information extraction and feedback.

To evaluate the value of LUMI's feedback mechanism as part of the Intent Confirmation module, we considered all intents that the 26 users generated and checked how many were confirmed and how many were rejected. If an intent was rejected, the user could provide feedback to correct it, thus improving LUMI. Such a corrected intent could then once again be accepted or rejected by the user. The right-hand side of Figure 6 gives the breakdown of the intents and shows that, most of the time, LUMI correctly interpreted the users' intents; in the few times feedback was needed, LUMI was able to correct and learn more often than not. This result is encouraging given the somewhat limited amount of data with which LUMI was trained.

On further analysis of the interactions that users had with LUMI, we observed that the feedback mechanism worked as expected in cases where the participants used a different or unusual phrasing of their intents. For instance, one user expressed the intent for task 3 in an unstructured manner, as "*student quota dorms 10GB week*", in which LUMI was not able to recognize the word "*dorms*" as a *@location* entity. However, the user then provided feedback that was successful in correcting the original *Nile* intent.

One concrete example where the feedback was unsuccessful happened in task 3, with a user that typed the intent "*Lumi, limit students to 10GB maximum download per week in dorms*". Lumi was only trained to recognize phrases of the form "10GB per week", and the additional text in between resulted in Lumi being unable to recognize the user's phrase. When asked what information LUMI had missed, the user provided feedback indicating that "10gb/wk" was an entity class and "Gb per week" was the value, instead of labeling "Gb per week" as a *@qos_unit* entity. We note that Lumi is an initial prototype, and such cases can be avoided in the future by improving the clarity of suggestions LUMI makes to the user, and by including sanity checks on user inputs.

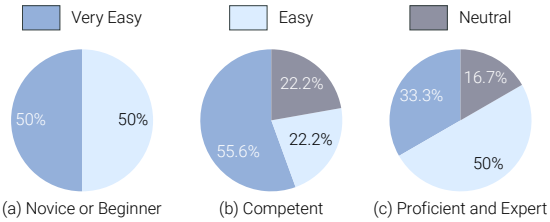
9.3 Users Reactions and Usability

In the post-questionnaire, we asked the users to comment on LUMI's usability and overall merit by answering three questions: (i) How easy was it to use LUMI to configure the network? (ii) Compared to traditional policy configuration, how much better or worse was using LUMI instead? and (iii) Would they rather use LUMI to enforce policies or conventional network configuration commands. Figures 7, 8 and 9 summarize the users' responses, broken down by expertise in network management. The results show that the participants' overall reaction to LUMI was very positive, with most of them stating that they would either use LUMI exclusively or, depending on the tasks, in conjunction with configuration commands. Note that the expert users who identified themselves as campus network operators all had a positive reaction to LUMI. Overall, among all different levels of expertise, 88.5% of participants stated they would rather use LUMI exclusively or in conjunction with configuration commands.

We also asked participants to provide insights they had that could help us improve the implementation and design of LUMI. One important feedback we received was the lack of support for querying the network state. For example, one participant stated:

> "*Many network management tasks are about monitoring something or figuring out what's happening, not just installing a new policy...*"

While LUMI was not designed with this goal in mind, we do not foresee any major NLP challenge to incorporate such features into it, as the entity set could be extended to cover this use case. Acquiring the state information from the network requires further investigation, but Lumi's modular design makes it simpler to plug in a new module or an existing one [13] to query the network devices. This would enable LUMI to "understand" changes made through other tools, as LUMI will likely co-exist with different management applications in real deployments. Overall, the feedback received from participants was positive and highlighted the promise and value of LUMI.



(a) Novice or Beginner (b) Competent (c) Proficient and Expert

Figure 7: User reaction to LUMI's usability, by expertise on network management.

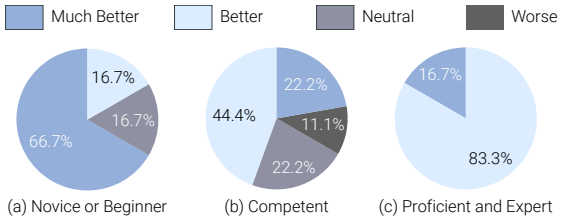


Figure 8: User reaction to LUMI's when compared to traditional network configuration, by expertise on network management.

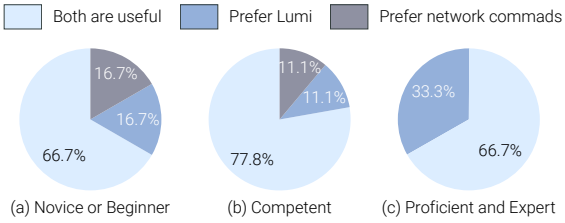


Figure 9: User reaction to LUMI's when compared to traditional network configuration, by expertise on networking.

10 Ongoing Work and Open Problems

While we consider LUMI to be a promising and necessary step towards fully realizing IBN, several challenges remain and are part of our ongoing work.

10.1 Ambiguities in natural language policies

NLP policies have the potential for ambiguities. In exploring this issue further, we extracted pairs of intents from our *campi* dataset and generated 213 pairs in all. Furthermore, we adapted existing NLP efforts on contradiction detection in general text [24, 37, 40, 42, 53, 55, 60] by developing a Random Forest Classifier trained to classify pairs of network intents based on contradiction indicators (features) found between the two input intents. The classifier flagged 9 cases with potential contradictions.

Manual inspection of all 213 intent pairs indicated that most of these cases were benign. They typically corresponded to cases where universities expressed policies that depended on a user's total traffic usage over different time periods (e.g., a 10 GB weekly limit vs a 5 GB daily limit). In these cases, the two policies could be applied in any order with no negative consequences. One interesting case, however, was a campus that expressed two different policies at different locations on their website. The first policy indicated H323 video conferencing was allowed by the University firewall, while the second indicated MSN audio and video communications were not allowed. This is a case where the relative precedence between the two policies impacts their joint effect.

Part of our ongoing work is to combine LUMI with formal methods to help detect ambiguities that are potentially of concern. Specifically, translating NLP policies into LUMI intents enables the use of automated methods that can check whether the impact of applying two policies is sensitive to their relative ordering, but also provides opportunities to use methods for detecting policy conflicts [2, 51].

10.2 Deploying Lumi in a production network

The initial design of LUMI was aimed at solving management problems that arise in a Campus network environment. We have been engaged in discussions with operators of our campus network regarding validating LUMI in production. Below, we discuss some of the issues raised by the operators and outline challenges and potential solutions.

Co-existing with current technologies. Most campus networks consist of legacy network equipment from multiple vendors with vendor-specific configuration interfaces. Getting LUMI "production-ready" requires developing a *Nile* compiler that can accommodate this diversity in legacy devices. Since the *Nile* abstraction offers isolation from the system's interface and minimizes the need for changes in the early stages of the LUMI pipeline, it is well-suited for OpenConfig [48], a vendor-neutral model for network management that is supported by an increasing number of devices.

Extending LUMI for other use-cases. While we have focused on Campus networks, deploying LUMI in other environments may require extending its feature set. For example, unlike the Campus networks we have access to, multi-tenant data-center networks may use VXLANs or NVGRE as solutions to scalably share network infrastructure between tenants. However, LUMI is easily extended to support such features, and we illustrate the four generic steps required for such extension with the VXLAN example. In a first step, one must decide the abstraction level in which LUMI should handle natural language text. Consistent with LUMI's design philosophy, for VXLAN support, a high-level intent could be *"Block incoming traffic for tenant A"*, where *"tenant A"* refers to a specific VXLAN Network Identifier (VNI). Next, LUMI's training dataset has to be augmented appropriately, the *Nile* language must be extended with new keywords (e.g., a `tenant('A')` operator for the VXLAN example), and the *Nile* compiler has to be instrumented to handle the new set of configurations. In the case of VXLAN, similarly to VLANs, tenants names must be mapped to VNIs. Lastly, since OpenFlow switches support VXLANs, all that is needed is to extend *Merlin* to allow matching traffic based on VNI for each tenant.

Creating sandbox environments. Migrating LUMI to production requires that operators have trust in LUMI. Based on our discussions with operators, a mirrored sandbox environment [3] could be a good starting point.

10.3 How to verify if Lumi is correct?

We discuss potential sources of errors in each stage of the LUMI pipeline and possible solution approaches.

Translating human language to *Nile* intents. Information extraction from natural language is inherently prone to errors. LUMI alleviates this problem by asking operators for

feedback and using their responses to check if the extracted information was correct. Over the longer term, we plan to leverage ongoing ML research efforts that focus on making ML models more robust and secure so they can be deployed in security- and safety-critical settings such as production network environments [9, 27].

Compiling *Nile* intents. Recent works on formal network verification [1, 10, 50] provide sub-second verification of waypointing, reachability, and isolation properties and are well suited for verifying configurations generated by LUMI-compiled intents. LUMI supports time-constrained intent deployment as well as QoS features. Despite recent work on verifying such properties (e.g., [31, 58]), more advances may be needed in the area, including the possible adaption of existing verification techniques to a LUMI-specific setting.

Post-deployment behavior monitoring. Ultimately, we envision the LUMI pipeline shown in Figure 1 to include a monitoring module for verifying that the deployed configurations respect the intents produced by the refinement process and achieve the objective(s) that the operator expressed (in natural language) in the first place. By monitoring both the traffic and configurations of specific devices affected by a deployed intent, such a module would allow operators to query at any time if the deployed intent produced the desired network behavior [22], thereby improving the operators' trust in relying on LUMI. However, deciding which traffic, devices, and properties to monitor will require instrumenting networks with an unprecedented level of control that is currently only possible by leveraging the latest programmable data plane technologies [8, 23]. At the same time, the development of such a module can be viewed as a first step towards realizing the vision of self-driving networks [12, 19, 45].

11 Related Work

Natural language in networking. Very few prior works use natural language to interact with the network. In [13], the authors present *Net2Text*, a system that allows network operators to query network-wide forwarding behaviors using natural language, but it does not allow operators to configure the network. Alsudais *et al.* [4] proposes using natural language to deploy network intents. It uses the Stanford CoreNLP Parts-of-Speech (POS) Tagger [61] to parse and structure the input text but does not cover NLP aspects relevant to LUMI such as intent confirmation for user feedback and learning over time.

Network programming languages. Recent works on IBN feature several intent languages, frameworks, and compilers to efficiently deploy intents in network devices and middleboxes [2, 6, 21, 31, 51, 54, 57, 59]. At the same time, Cocoon [54] introduces a framework to guarantee the correctness of SDN programs that resembles our approach, but it uses first-order logic. With LUMI, we examine the use of machine learning to convert natural language intents into lower-level configurations without the need for using a specific programming language.

Natural language processing. Information extraction has been addressed with different methods and techniques, including *(i)* only CRF models [20]; *(ii)* character-level embeddings instead of whole words [34], and *(iii)* rule-based approaches [15]. Yet, recent studies [63] show the benefits of the approach considered as part of LUMI. To our knowledge, the problem of using natural language for management tasks has not received much attention in the networking domain.

IBN. INSpIRE [56] focuses on intents related to security middleboxes and uses a refinement process to determine which middleboxes should compose a service chain to fulfill an intent. Cheminod et al. [14] propose an automatic process for refining, deploying, and enforcing ACL policies that use set notation for policy specification. PGA [51] applies a graph-based abstraction to compose high-level policies and deploy them in SDN networks. Janus [2] extends PGA to support policies with QoS requirements, mobility, and temporal dynamics. However, these proposals differ from LUMI as they are not concerned with using natural language or obtaining feedback from the network operator.

12 Conclusions

In this paper, we propose LUMI, a novel end-to-end intent refinement and deployment system that allows operators to express their intents in natural language and then check and confirm the intents before deploying them in the network. By demonstrating that LUMI can successfully deal with a wide range of network policies, this paper represents a promising step towards realizing the vision of intent-based networking with natural language. Still, much work remains. For example, while our design choices for LUMI's different modules resulted in a working prototype, other features might be necessary for a production-ready version of the system. However, LUMI's modular design can readily accommodate such improvements. Also, since LUMI in its current form is mainly intended for use in campus networks, supporting other environments (*e.g.,* home or enterprise networks) will most likely require that the set of *Nile* operations and functions (and in turn the set of LUMI entities) be judiciously extended.

Acknowledgments

We thank our shepherd Costin Raiciu and the anonymous reviewers for their valuable feedback. We thank Jennifer Rexford, Hyojoon Kim, Shir Landau-Feibish, and Ross Teixeira for their feedback on earlier drafts of this paper. We also thank the anonymous participants of our user study for their contributions and insights. This work was supported in part by the Brazilian National Research and Educational Network (RNP), the Brazilian Federal Agency for Support and Evaluation of Graduate Education (CAPES), the Brazilian National Council for Scientific and Technological Development (CNPq) procs. 423275/2016-0, 312392/2017-6, 142089/2018-4, INCT InterSCity, and FAPESP procs. 2018/23085-5, 2020/05183-0, and 2015/24494-8, and the US National Science Foundation Grant FMitF 1837023.

References

[1] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast and General Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, Santa Clara, CA, February 2020. USENIX Association.

[2] A. Abhashkumar, J. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu. Supporting Diverse Dynamic Intent-based Policies Using Janus. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 296–309, New York, NY, USA, 2017. ACM.

[3] R. Alimi, Y. Wang, and Y. R. Yang. Shadow Configuration as a Network Management Primitive. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '08, page 111–122, New York, NY, USA, 2008. ACM.

[4] A. Alsudais and E. Keller. Hey network, can you understand me? In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 193–198, May 2017.

[5] A. Amokrane, R. Langar, R. Boutaba, and G. Pujolle. Flow-Based Management For Energy Efficient Campus Networks. *IEEE Transactions on Network and Service Management*, 12(4):565–579, December 2015.

[6] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.*, 49(1):113–126, January 2014.

[7] J. Apostolopoulos. Improving Networks with Artificial Intelligence, Oct 2020. https://blogs.cisco.com/networking/improving-networks-with-ai.

[8] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic In-Band Network Telemetry. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. ACM.

[9] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[10] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '17, pages 155–168, New York, NY, USA, 2017. ACM.

[11] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '16, pages 328–341, New York, NY, USA, 2016. ACM.

[12] M. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. Carpenter, S. Jiang, and L. Ciavaglia. Autonomic Networking: Definitions and Design Goals. Rfc 7575, RFC Editor, June 2015.

[13] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev. Net2Text: Query-Guided Summarization of Network Forwarding Behaviors. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 609–623, Renton, WA, 2018. USENIX Association.

[14] M. Cheminod, L. Durante, L. Seno, F. Valenza, and A. Valenzano. A comprehensive approach to the automatic refinement and verification of access control policies. *Computers & Security*, 80:186–199, 2019.

[15] L. Chiticariu, M. Danilevsky, Y. Li, F. Reiss, and H. Zhu. SystemT: Declarative Text Understanding for Enterprise. In *Proceedings of the 2018 Conference of the North American Chapter of the ACL: Human Language Technologies, Volume 3*, pages 76–83. ACL, 2018.

[16] J. Chiu and E. Nichols. Named Entity Recognition with Bidirectional LSTM-CNNs. *Transactions of the ACL*, 4:357–370, 2016.

[17] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. Intent-Based Networking - Concepts and Definitions. Internet-Draft draft-irtf-nmrg-ibn-concepts-definitions-00, Internet Engineering Task Force, December 2019. Work in Progress.

[18] F. Chollet et. al. Keras, 2015. https://github.com/fchollet/keras.

[19] N. Feamster and J. Rexford. Why (and How) Networks Should Run Themselves. In *Proceedings of the Applied Networking Research Workshop*, ANRW '18, page 20, New York, NY, USA, 2018. ACM.

[20] J. R. Finkel, T. Grenager, and C. Manning. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. In *Proceedings of the 43rd Annual Meeting on ACL*, Acl '05, pages 363–370, Stroudsburg, PA, USA, 2005. ACL.

[21] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. *SIGPLAN Not.*, 46(9):279–291, September 2011.

[22] N. Foster, N. McKeown, J. Rexford, G. Parulkar, L. Peterson, and O. Sunay. Using Deep Programmability to Put Network Owners in Control. *SIGCOMM CCR*, 50(4):82–88, October 2020.

[23] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. ACM.

[24] S. M. Harabagiu, A. Hickl, and V. F. Lacatusu. Negation, Contrast and Contradiction in Text Processing. In *AAAI*, 2006.

[25] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[26] M. Honnibal and I. Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *To appear*, 2017.

[27] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 3–29, Cham, 2017. Springer International Publishing.

[28] Google Inc. Dialogflow, March 2018. https://dialogflow.com/.

[29] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville. Refining Network Intents for Self-Driving Networks. In *Proceedings of the Annual Conference of the ACM SIGCOMM Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, pages 15–21, New York, NY, USA, 2018. ACM.

[30] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2019.

[31] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable Dynamic Network Control. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 59–72, Berkeley, CA, USA, 2015. USENIX Association.

[32] B. Koley. The Zero Touch Network, 2016. Keynote Speech at the 12th International Conference on Network and Service Management. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45687.pdf.

[33] J. D. Lafferty., A. McCallum, and F. C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the 18th International Conference on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[34] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer. Neural Architectures for Named Entity Recognition. In *Proceedings of the 2016 Conference of the North American Chapter of the ACL: Human Language Technologies*, pages 260–270, San Diego, California, June 2016. ACL.

[35] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[36] N. Limsopatham and N. Collier. Bidirectional LSTM for Named Entity Recognition in Twitter Messages. In *Proceedings of the 2nd Workshop on Noisy User-generated Text, NUTCOLING 2016, Osaka, Japan, December 11, 2016*, pages 145–152, 2016.

[37] V. Lingam, S. Bhuria, M. Nair, D. Gurpreetsingh, A. Goyal, and A. Sureka. Deep learning for conflicting statements detection in text. *PeerJ Preprints*, 6:e26589v1, March 2018.

[38] H. H. Liu. The Practice of Network Verification in Alibaba's Global WAN, May 2021. https://netverify.fun/the-practice-of-network-verification\-in-alibaba-global-wan/.

[39] Lumi. Lumi supplemental material, January 2020. https://lumichatbot.github.io/.

[40] B. MacCartney, T. Grenager, M. C. de Marneffe, D. Cer, and C. D. Manning. Learning to Recognize Features of Valid Textual Entailments. In *Proceedings of the Main Conference of the North American Chapter of the ACL*, NAACL '06, pages 41–48, Stroudsburg, PA, USA, 2006. ACL.

[41] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL System Demonstrations*, pages 55–60, 2014.

[42] M. C. De Marneffe, A. N. Rafferty, and C. D. Manning. Finding contradictions in text. In *Proceedings of the Conference of the ACL*, ACL '08, pages 1039–1047, 2008. cited By 108.

[43] M. Marrero, J. Urbano, S. Sánchez-Cuadrado, J. Morato, and J. M. Gómez-Berbís. Named Entity Recognition: Fallacies, Challenges and Opportunities. *Computer Standards & Interfaces*, 35(5):482–489, 2013.

[44] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. *CoRR*, abs/1301.3781, 2013.

[45] J. Mogul. Unsafe at Any Speed? Self-Driving Networks without Self-Crashing Networks. In *Keynote Speech at the ACM SIGCOMM Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, New York, NY, USA, 2018. ACM.

[46] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, Lombard, IL, 2013. USENIX Association.

[47] Juniper Networks. What Is Intent-Based Networking? https://www.juniper.net/us/en/products-services/what-is/intent-based-networking/.

[48] OpenConfig. OpenConfig, January 2016. http://www.openconfig.net/.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. http://scikit-learn.org.

[50] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, Santa Clara, CA, February 2020. USENIX Association.

[51] C. Prakash, J. Lee, Y. Turner, J. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '15, pages 29–42, New York, NY, USA, 2015. ACM.

[52] Microsoft Research. Hyperscale cloud reliability and the art of organic collaboration, Nov 2018. https://www.microsoft.com/en-us/research/blog/hyperscale-cloud-reliability-and-the\-art-of-organic-collaboration.

[53] A. Ritter, D. Downey, S. Soderland, and O. Etzioni. It's a Contradiction–no, It's Not: A Case Study Using Functional Relations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Emnlp '08, pages 11–20, Stroudsburg, PA, USA, 2008. ACL.

[54] L. Ryzhyk, N. Bjørner, M. Canini, J. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese. Correct by Construction Networks Using Stepwise Refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 683–698, Boston, MA, 2017. USENIX Association.

[55] F. Sarafraz. *Finding conflicting statements in the biomedical literature*. PhD thesis, University of Manchester, UK, 2012.

[56] E. J. Scheid, C. C. Machado, M. F. Franco, R. L. dos Santos, R. P. Pfitscher, A. E. Schaeffer-Filho, and L. Z. Granville. INSpIRE: Integrated NFV-based Intent Refinement Environment. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 186–194, May 2017.

[57] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Managing Network Resources. *IEEE/ACM Transactions on Networking*, 26(5):2188–2201, October 2018.

[58] Y. E. Sung, C. Lund, M. Lyn, S. G. Rao, and S. Sen. Modeling and Understanding End-to-End Class of Service Policies in Operational Networks. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '09, page 219–230, New York, NY, USA, 2009. ACM.

[59] Y. E. Sung, X. Tie, S. H. Y. Wong, and H. Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '16, pages 426–439, New York, NY, USA, 2016. ACM.

[60] N. S. Tawfik and M. R. Spruit. Automated Contradiction Detection in Biomedical Literature. In Petra Perner, editor, *Machine Learning and Data Mining in Pattern Recognition*, pages 138–148, Cham, 2018. Springer International Publishing.

[61] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the ACL*, pages 252–259, 2003.

[62] VMware. Intent-based Networking, May 2021. https://www.vmware.com/topics/glossary/content/intent-based-networking.

[63] V. Yadav and S. Bethar. A Survey on Recent Advances in Named Entity Recognition from Deep Learning model. In *Proceedings of the 27th International Conference on Computational Linguistic*, pages 2145–215. ACL, 2018.