Dynamic Natural Language User Interfaces Using Microservices

Riyadh Mahmood, Arth Joshi, Adwait Lele, Jay Pennington
{riyadh.mahmood,arth.h.joshi,adwait.lele,jay.m.pennington}@aero.org
The Aerospace Corporation
Chantilly, Virginia, USA

ABSTRACT

Microservices have become a popular way to build and deploy software systems as they allow engineers to break down systems into smaller components that can be deployed and maintained independently. Interactions with these systems typically employ a Graphical User Interface (GUI) that weaves and orchestrates the services together to perform use cases. However, each time the requirements change or the underlying services change, the logic in the GUI must be updated, increasing maintenance costs. There is a need for the orchestration logic to be more flexible. We present an automated approach to build user interfaces using natural language that can augment or replace these types of graphical interfaces. Our approach creates the user interface by extracting *intents* from requirement statements and satisfying them by *dynamically orchestrating* the underlying microservices. We are able to handle requirement changes as well as adding, removing, or updating services on the fly without additional development.

CCS CONCEPTS

• **Human-centered computing** → **Natural language interfaces**; • **Software and its engineering** → *Software prototyping*.

ACM Reference Format:
Riyadh Mahmood, Arth Joshi, Adwait Lele, Jay Pennington. 2020. Dynamic Natural Language User Interfaces Using Microservices. In *IUI '20 Workshops, March 17, 2020, Cagliari, Italy.* ACM, New York, NY, USA, 11 pages.

1 INTRODUCTION

Applications have traditionally been built as a monolith where a Graphical User Interface (GUI) is tightly coupled with the underlying business logic. The needs of the users change over time leading to changes in the GUI or the business logic. System changes are also needed to fix defects that are uncovered in production. In a monolithic system, if either the GUI or the business logic changes, the entire system has to be updated, repackaged, and redeployed. This pattern increases development, testing, and deployment time. It also reduces user satisfaction as there are times when the system must be patched or is unavailable due to maintenance.

To mitigate these challenges, there has been a paradigm shift to the DevOps [9] software engineering model. A common DevOps practice is to adopt a microservices architecture [27] where the system is decomposed into smaller, simpler components (services) that can be developed and deployed independently. These services work together to perform system tasks and use cases. This architecture allows different teams to work on different services, make updates, and deploy code while reducing coordination of large changes across the system. It also allows the GUI to be decoupled from the business logic with each underlying microservice fulfilling a portion of one or more use cases. The orchestration logic in the GUI determines which services to call and the invocation sequence of the calls.

However, in the microservices architecture, the GUI is still not flexible because the service orchestration code has to be updated each time the requirements change, the underlying service interfaces change, or the services are retired. To make the user interface more flexible, emerging designs use natural language to augment or replace the GUI. Examples of these types of interfaces include Amazon Alexa [3], Apple Siri [7], Google Assistant [19], and Microsoft Cortana [26]. These conversational interfaces, dubbed smart assistants [32], fulfill various tasks (e.g. ordering a pizza). Each task is supported by one or more capabilities of these smart assistants. The capabilities are generally built manually (e.g. using Alexa Skills Kit [2]) in a flowchart-like manner. The flow and execution of the skills is rigid and require manual changes to handle varying user interactions and service bindings.

In this paper, we combine the loose coupling strength of the microservices architecture with the flexibility of natural language interfaces to build a dynamic user interface. Our approach takes a set of requirements (e.g. *the system shall support ordering a pizza*) and a set of available microservices to dynamically generate a natural language user interface on the fly. Using the requirements, we automatically extract the user intents, then extrapolate phrases and utterances from the requirements and map them to the intents to handle varying user interactions. We create models from the available microservices using their OpenAPI specifications [30] to derive service orchestration flows that satisfy the intents. This allows for the orchestration logic to be dynamically built depending on service capability and availability. If the requirements change or the underlying microservices change, our approach is able to automatically regenerate the interface. We built a prototype showing that this approach is feasible.

The remainder of this paper is organized as follows. Section 2 provides background information and an overview of related works. Section 3 describes the research challenge and the motivation for our work using a simple example. Section 4 provides the details of our approach, while Section 5 describes the prototype we built and our preliminary results. The paper concludes with a discussion of limitations and future work in Section 6.

2 BACKGROUND AND RELATED WORK

DevOps [9] is a combination of *development* and *operations* software engineering practices. Under a DevOps model, development and operations teams work together across the entire application lifecycle. A common practice in DevOps is to adopt a microservices architecture [27] where the system is decomposed into smaller, simpler components that can be developed and deployed independently. There are many benefits of using the microservices paradigm including increased autonomy, better fault isolation, continuous delivery, scalability, and reusability [13]. The OpenAPI specification [30] is the de facto standard that describes microservices. It facilitates machine-to-machine communication over RESTful [15] application programming interfaces (APIs).

Smart assistants [3, 7, 19, 26, 32] are an emerging type of user interface that use natural language to fulfill user intents through conversation. Many tools are available that help power these smart assistants. Amazon Web Services (AWS) offers Lex [5], Comprehend [4], and Polly [6]. Lex is a service for building conversational bots that can understand natural language in both text and audio. It performs speech-to-text conversion and intent management using the same deep learning [18] algorithms that power Alexa [3]. Comprehend is a natural language processing (NLP) service that can understand text and extract key phrases, places, events, entities, and overall sentiment. Polly is a text-to-speech service that synthesizes audio from text input.

Building chatbots using serverless computing has been investigated [35] with IBM OpenWhisk [20]. However, microservice orchestration and the OpenAPI specification were not used. One way to orchestrate multiple disconnected services is through a graph framework that makes connections between RESTful APIs using a meta-model to describe their properties [1]. In another approach [16], dependency relationships between elements are captured and elements are combined to build form-like user interfaces, but this is limited to manipulation of spreadsheet cells and formulas. Our approach is more generalized and independent of the type of user interface. Previously, mobile apps have been designed to combine app functions by running multiple apps in the background [22]; our approach is more responsive because it calls API endpoints independently without waiting on all services.

Slot filling is a common technique used to derive parameters used by chatbots and smart assistants. One approach to slot filling is using conditional random fields (CRFs) [25]. Other techniques use recurrent neural networks [23, 25, 36] and word confusion networks (WCNs) [33] in addition to CRFs. When attempts at slot filling and general dialog are unsuccessful, admitting failure, showing a list of capabilities with examples, and providing a witty cover-up can be helpful, as shown by a study conducted on first-time chatbot users [21]. More robust automatic speech recognition (ASR) systems can be built with noise feedback loops and labeled word bins [33].

Paraphrase generation is used to exemplify the different types of natural language inputs when invoking an intent. Ganitkevitch et al. have developed a Paraphrase Database that contains over 200 million paraphrase pairs generated using bilingual pivoting [17]. In this technique, phrases are translated from a source language to a pivot language and then translated back to the source language, creating a new phrase in the source language. Marton et al. have proposed an improved statistical machine translation (SMT) method which uses monolingually-derived paraphrases, not relying on bitexts and therefore having larger amounts of training data [24]. Additionally, Monte-Carlo sampling [12] and statistical paraphrase generation [37] have been developed as alternatives to SMT, each with their own strengths and weaknesses. Class diagrams have also been used to generate paraphrases based on abstract syntax [10]. Campgna et al. have proposed formalizing virtual assistant capabilities using a Virtual Assistant Programming Language [11]. They use crowdsourced paraphrases and data augmentation, along with the synthesized data, to train a semantic parser and lessen manual efforts. Iris [14] presents a way to support complex tasks by combining commands through nested conversations and applying them in the data science domain. The Iris team created a domain-specific language that transforms Python functions into combinable automata and regulates their combinations through a type system. Our approach is more generalized and can be applied in any domain that uses the OpenAPI specification, but there may be opportunities to combine the techniques.

3 RESEARCH CHALLENGE

We use a simple pizza ordering application to illustrate the research challenge and help the reader understand the motivation of our work. In this application, the user interacts with a GUI to place pizza orders by filling out a form with information about the pizza, including the size and toppings. Then the user fills out another form with payment information. Finally, the application verifies the payment information with a third-party interface and submits the pizza order.

In a traditional architecture, shown in Figure 1a, the GUI is tightly coupled with the underlying business logic. The logic for gathering the user inputs, verifying the payment information, and placing the pizza order is packaged into a monolithic system. A change to any part of the application involves development and redeployment of the entire application. For example, if the third-party payment interface is discontinued, the application must be updated to handle that change. Since the user interface is coupled together, the entire application must be redeployed.

A microservices architecture can overcome this issue by decomposing the system into multiple small components so they can be developed and deployed independently as shown in Figure 1b. The pizza ordering application is divided into three components: the GUI, the payment processing microservice, and the pizza order placement microservice. With this approach, if the third-party payment interface changes, only the payment processing microservice needs to be updated, and the remaining application stays unchanged. However, the orchestration logic to first call the payment service and then the ordering service is hardcoded in the GUI. The workflow logic to satisfy the user intent to order a pizza is not flexible or dynamic. For example, if a new payment processing microservice were to be used, the GUI logic must be updated to leverage that service and the GUI will have to be redeployed.

Smart assistants belong to an emerging class of user interfaces that are more flexible. However, the underlying logic to fulfill user intents is still built at design time. Supported intents, as well as their input parameters, must be developed manually. Deviations from the scripted interaction flows are not well supported. There is also no standard way to fulfill the intents; it is left up to the designer. Our approach combines the loosely coupled microservices architecture with the flexibility of the natural language interface to overcome these challenges. We are able to automatically generate the user interface on the fly by processing OpenAPI specifications for microservices and mapping them to user intents that are derived from simple requirement statements. We can automatically derive the intents, supporting invocation phrases, and parameter slots to dynamically orchestrate the services.

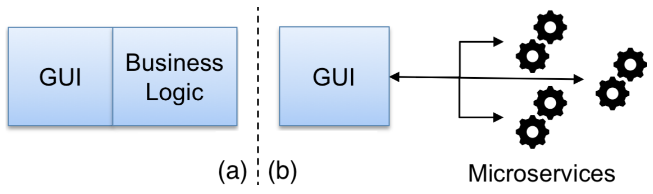


Figure 1: (a) Traditional architecture (b) Microservices architecture

4 APPROACH

The overall approach is shown in Figure 2. To generate the natural language user interface, we take a set of microservices, defined using the OpenAPI specification [30], and a set of requirement statements as inputs. The microservices are a set of endpoints that are available for use in the system. They can either be curated, discovered via crawling, or obtained from a service registry. The requirements are a set of statements written in natural language commonly found in many system requirement specifications (e.g. *the system shall support ordering a pizza*).

As shown in Figure 2, we analyze each microservice by parsing its OpenAPI specification to understand its structure, semantics, and behavior. This information is captured into two models, namely the *Services Model* (SM) and the *Invocation Graph Model* (IGM). The SM captures *syntactic* and *semantic* information regarding each endpoint in a given microservice. Syntactic information includes the operations each microservice supports (e.g. *F1*), the parameters needed to invoke each of the operations (e.g. *size*), the order and type of each parameter (e.g. *string*), whether the parameter is required, and the output of each operation (e.g. *number*). The semantic information of each endpoint is captured by analyzing specification metadata (e.g. *cost of pizza based on options*) as well as the text in the endpoint name itself. After the SM is generated, we develop a set of graphs for each of the endpoints within and across the microservices that can be connected together and capture it in the IGM.

We use two simple rules to infer if two endpoint operations are to be connected. Connect them only if: 1) their syntactic signatures are compatible, and 2) they are semantically compatible. These rules are satisfied by using information from the SM. The syntactic compatibility is very similar to method or function signatures used in programming languages and determines if the operations *can be connected* (e.g. if function F1 returns a number and function F2 takes in a number parameter, then *F2(F1)* is syntactically compatible). The semantic compatibility is based on text analysis to determine if the operations *should be connected* based on the similarity of their input and output (e.g. if F1 returns a *dollar amount* and F2 takes in a *dollar amount*, then they are semantically compatible). The result is a set of graphs, or *orchestration* *plans*, that describe the possible invocation sequence of the microservices. We optimize the plans based on service availability, latency, and other quality of service attributes. The optimization also takes into account the quality of the user interaction from previous executions. Each plan is scored based on these criteria.

After the models are created, we take the requirement statements and apply key phrase detection to understand the *intent* of each statement. This process involves implementing terminology extraction, which ranks n-gram [34] candidates within the text. This ranking is converted to a relevancy score using tf-idf [31], a frequency statistic that shows the importance of words in a given corpus. This is used to determine the overall intent of each requirement statement. For example, given a set of requirements that start with *the system shall support x*, the *x* in each statement would be given more weight and interpreted as the intent. This method is used to be able to support legacy systems where the *the system shall support x* format is commonly used. It can also be applied in cases where standardized formats such as *the system shall support x* are not available and we need to extract intent(s) from non-standard textual descriptions (e.g. use case description or user story text). Additionally, for simpler systems, the intents can be supplied as a list of of words or phrases directly without having to perform semantic parsing.

To handle user interactions, each intent is mapped to one or more orchestration plans in the IGM by inferring the semantics of the intent as well as the semantics of the nodes of the orchestration plan. As seen in Figure 2, the set of intents can each be mapped to a set of invocation sequences (e.g. *F1 → F2 or F9 → F10 → F11 as two separate orchestration plans*). In the pizza ordering example, there may be two orchestration plans in the IGM that involve calling different payment services for pizza ordering. Semantically they are very close and are related to pizza ordering, but beneath the surface they invoke different services. The pizza ordering intent can be connected to both plans, but the plan with the higher score will be executed at runtime.

Once an intent is mapped, several invocation phrases are created using natural language generation [10, 12, 17, 24, 37] to capture the various ways the intent can be invoked. The seed phrases are derived using the requirement statements and the intents. We use the SM to incorporate the parameters needed for a particular intent and corresponding invocation phrases. This helps generate possible conversation flows within the system, as each intent knows what parameters it needs from the user for each orchestration plan. In the pizza ordering example, the specification may list an enumerated type for pizza size (e.g. *small, medium, large*) that defines the valid values for the *size* parameter. It may also mark required parameters that the user must provide to continue the

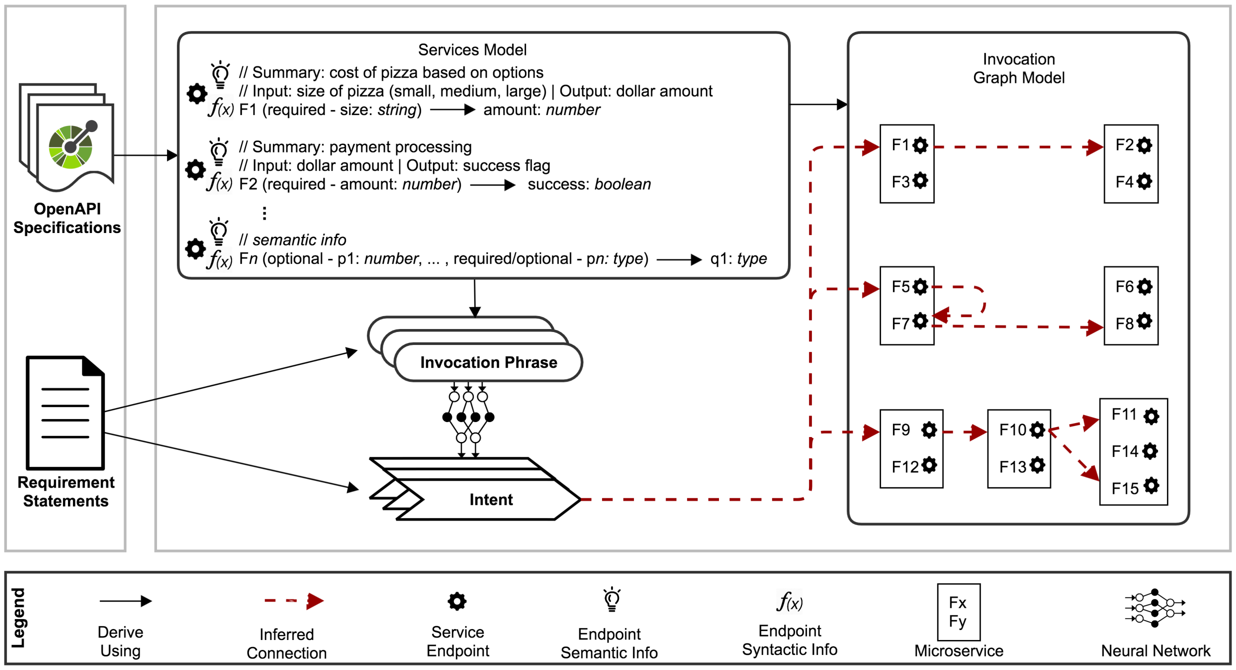

Services Model
// Summary: cost of pizza based on options
// Input: size of pizza (small, medium, large) | Output: dollar amount
f(x) F1 (required - size: string) → amount: number
// Summary: payment processing
// Input: dollar amount | Output: success flag
f(x) F2 (required - amount: number) → success: boolean
// semantic info
f(x) Fn (optional - p1: number, ... , required/optional - pn: type) → q1: type
OpenAPI Specifications
Requirement Statements
Invocation Phrase
Intent
Invocation Graph Model
F1
F3
F2
F4
F5
F7
F6
F8
F9
F12
F10
F13
F11
F14
F15
Legend
Derive Using
Inferred Connection
Service Endpoint
Endpoint Semantic Info
Endpoint Syntactic Info
Fx Fy
Microservice
Neural Network

Figure 2: Approach Overview

conversation. The invocation phrases are used as a training set for a neural network model that maps user utterances to intents (e.g. *I would like a pizza* vs. *I would like a large pizza*, where *large* is the size parameter). This allows the interface to handle varying user queries by resolving to the correct intent and automatically filling in given parameters when possible. We intentionally underfit our model to allow wide-ranging user interactions.

The next sections describe our approach in further detail.

Services Model

The Services Model (SM) provides an abstraction on the most pertinent details of the service specification. It helps us understand what the endpoints do, what parameters they require, and how they may be able to call each other.

To build the SM, we parse the specification for each endpoint in each microservice. Figure 3a shows a partial OpenAPI specification for the pizza ordering microservice. We parse the specification based on the version information, in this case 3.0.0 (line 01). We also extract metadata information such as the title (line 03). For each endpoint (e.g. order-pizza, line 05), we extract the summary information as well as the inputs and output for the endpoint.

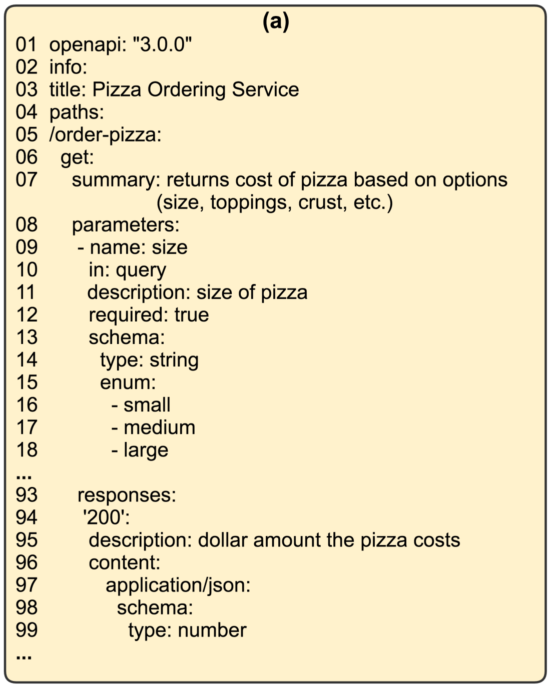
In this example, we see that the input parameters (line 08) include a size parameter (line 09), as well as the size parameter's description (line 11), type (line 14), and list of enumerated values (lines 15-18). We also see that this parameter is required (line 12). The response (line 93) of this endpoint tells us the description (line 95) and the return type (line 99).

Using this information, we can create the model for this endpoint as shown in Figure 3b. The semantic portion of the model captures the meaning of this endpoint described by the name of the endpoint, the summary of the endpoint, and the description of the parameters. This can be as simple as capturing a list of keywords. The syntactic portion of the model represents the endpoint as a function with input parameters and output along with their data types. This information is used to connect endpoints together as described in the Invocation Graph Model section below.

One thing to note is that there may be multiple function signatures listed due to the optional parameters. Since parameters other than *size* (*toppings, crust,* etc. not shown in the partial specification) are not required, it is possible for the user to optionally supply these parameters.

Invocation Graph Model

The Invocation Graph Model (IGM) informs us how the service endpoints are connected together. The service endpoints can invoke other endpoints in the same service or in other services. The invocations are captured as a graph where the



(a)
```
01  openapi: "3.0.0"
02  info:
03  title: Pizza Ordering Service
04  paths:
05  /order-pizza:
06    get:
07      summary: returns cost of pizza based on options
                   (size, toppings, crust, etc.)
08      parameters:
09       - name: size
10         in: query
11         description: size of pizza
12         required: true
13         schema:
14           type: string
15           enum:
16             - small
17             - medium
18             - large
...
93       responses:
94        '200':
95         description: dollar amount the pizza costs
96         content:
97            application/json:
98              schema:
99                type: number
...
```

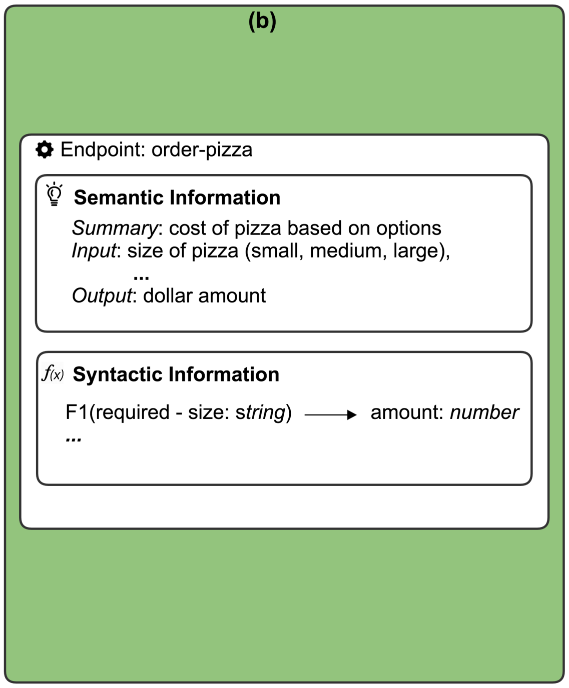

(b)
Endpoint: order-pizza
Semantic Information
Summary: cost of pizza based on options
Input: size of pizza (small, medium, large),
...
Output: dollar amount
f(x) Syntactic Information
F1(required - size: string) → amount: number
...

Figure 3: (a) Partial OpenAPI specification for pizza ordering service, (b) Partial Services Model based on the specification

nodes are the endpoints and the edges are the calls. As previously mentioned, these invocation paths are also known as orchestration plans and are used to fulfill user intents.

To construct the IGM, we use a similar approach described by Atlidakis et al. [8]. However, we connect the endpoints based on their semantic information as well as their syntactic information from the SM. Figure 4a shows two partial endpoints in the SM. One endpoint is from the pizza ordering microservice, and another is from the payment processing microservice. The points of interest are marked by ① and ②. Given this information we would like to see if they should be connected. To infer if two endpoint operations are to be connected, we use two rules described earlier: connect them only if their syntactic signatures are compatible and if they are semantically compatible. We see that the output semantic information for the order-pizza endpoint is very similar to the input semantic information of the payment-processing endpoint ①. Additionally, the syntactic signatures are also compatible; one outputs a number and the other takes a number as the input ②. Based on this information, we connect these two endpoints.

By connecting endpoints based on these two rules, we derive the IGM as shown in Figure 4c. This partial model shows the two endpoints we connected for pizza payment processing F1 → F2, and the connection is marked using ① and ②. Figure 4c also shows several other notional endpoints (F3 to F15)

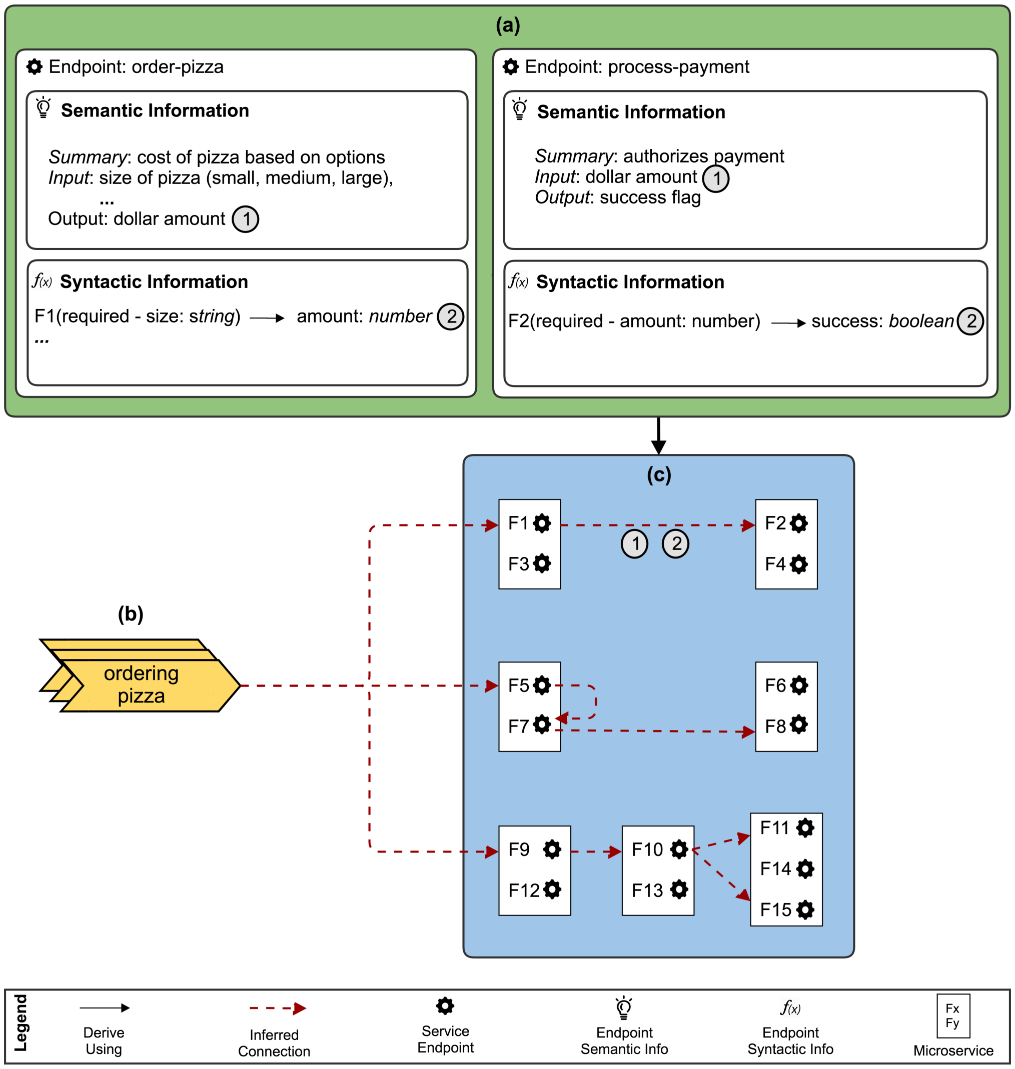


Figure 4: (a) Partial Services Model for the pizza ordering service and payment processing service, (b) Intents to be mapped to one or more invocation graphs, (c) Partial Invocation Graph Model showing inferred connections

that may be connected in the same manner. Together these represent the possible invocation sequences in the system. It should be noted that it is possible to have a single node in the graph if we inferred that a service does not need to call any other services to fulfill its objectives. It is also possible for a service to invoke other endpoints within itself (e.g. F5 $\rightarrow$ F7) or not require other services at all.

Now that the invocation flows have been inferred, we map the intents to one or more invocation flows in the IGM. The intents are connected to the first node of the invocation sequences as seen in Figure 4b. We can infer the pizza ordering intent can be connected to F1 based on the semantic information of F1 (i.e. summary field) compared to the intent itself. All inferences made are probabilistic in nature and it is possible to have incorrect mappings. However, we improve this over iterations based on the satisfaction of the user interactions.

Essentially, when an intent is triggered by the user, the system executes a path in the IGM. The user is prompted by the system to supply all required parameters at each node to continue on to the next node. Therefore, it is easier for the user to be able to directly supply some of the parameters, both required and optional ones, when invoking an intent. To support this, we generate a set of invocation phrases as described in the next section.

Invocation Phrase Generation

To handle the various ways the user can invoke intents, we extrapolate invocation phrases based on the requirements and information from the SM. Figure 5c shows the requirement statement for ordering pizza. From this requirement, we derive the intent as shown in Figure 5d. To do this, we take the requirement statements and apply key phrase detection to understand the *intent* of each statement. As previously mentioned, this process involves implementing terminology extraction, which ranks n-gram candidates within the text. This ranking is converted to a relevancy score using tf-idf [31]. In the simplest form, the least common n-gram can be used as the intent (e.g. ordering pizza).

We know from the IGM that this intent is mapped to the order-pizza endpoint, and the SM tells us the semantic and syntactic information regarding the endpoint including parameter information. Figure 5a shows a subset of the parameters for the pizza ordering endpoint, namely the enumerated values. Recall the enumerated data type for the size parameter from the sample specification listed in Figure 3a. We take the requirement statement from Figure 5c and the parameter information from Figure 5a to generate invocation phrases and place various parameters in the phrases. This can be achieved via paraphrase generation techniques [10, 12, 17, 24, 37]. We take these generated invocation phrases in Figure 5b and reduce them back to the intent in Figure 5d by training a neural network. This helps the users interact with the system in various ways for each supported intent.

It is possible for nonsensical invocation phrases to be generated (e.g. *can you order pizza to large in Pete's Pizza?*), but this is acceptable because it just means the user will not interact with the system in this manner. We intentionally generate many invocation phrases for each intent to have sufficient coverage. Currently, it is a limitation that we only support parameters with a known domain (i.e. enumerated types) as it is not feasible to consider all possible strings. However, if the domain was not specified or the parameters are not present in an invocation phrase, the system will prompt the user for

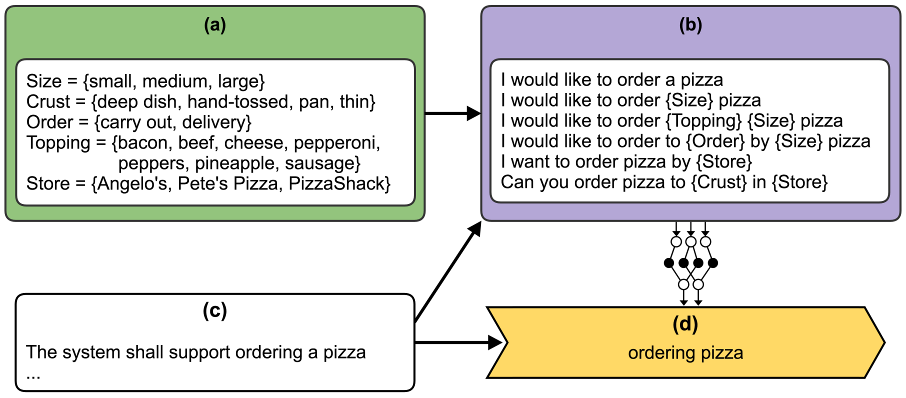

(a)
Size = {small, medium, large}
Crust = {deep dish, hand-tossed, pan, thin}
Order = {carry out, delivery}
Topping = {bacon, beef, cheese, pepperoni, peppers, pineapple, sausage}
Store = {Angelo's, Pete's Pizza, PizzaShack}
(b)
I would like to order a pizza
I would like to order {Size} pizza
I would like to order {Topping} {Size} pizza
I would like to order to {Order} by {Size} pizza
I want to order pizza by {Store}
Can you order pizza to {Crust} in {Store}
(c)
The system shall support ordering a pizza
...
(d)
ordering pizza

Figure 5: (a) Sample parameters for the pizza ordering endpoint from the Services Model, (b) Extrapolated invocation phrases for the pizza ordering intent generated using the requirement statement and parameters, (c) Subset of the requirements statements, (d) Pizza ordering intent derived from the requirement statement; extrapolated invocation phrases for this intent are trained to map to this intent

the required parameters when executing the endpoints. This mitigates the limitation, but the user interactions are less fluid.

Orchestration Plan Ranking

Each orchestration plan – or invocation sequence path in the IGM – is used to fulfill a user intent. Since the plans are inferred automatically and each intent can be mapped to multiple plans, we use two strategies to rank the plans and improve the quality of the user interactions. These strategies are used together to select the best plan for execution. If an intent can be fulfilled by multiple plans, then the plan with the highest score is always selected for execution followed by lower ranked plans.

First, we monitor health information for each service such as CPU, memory, I/O, and network use as well as the current load and latency on that service. This information is refreshed in near real time and used to assign a quality of service score to the services. The quality of each plan is calculated by factoring in the quality of service score for every service within the plan. Healthier and more responsive plans are ranked higher.

Second, we also keep the execution history and user interactions for all intents. By analyzing user interactions, we can identify patterns of successful or satisfactory orchestration plans over time. This is similar to detecting spam e-mail; the more users that mark an e-mail as spam, the higher the likelihood that the e-mail is indeed spam. Similarly, if many users are frustrated with an interaction, it is likely that the plan is not very good at fulfilling the intent. For example, if we observe that many users are individually repeating variations of the same query, it is likely that their intent is not being fulfilled and the orchestration plan supporting that query should be replaced. Plans that satisfy intents for large number of users without issues are ranked higher.

5 PROTOTYPE AND EVALUATION

We built a prototype that implements our approach as shown in Figure 6. Five microservices were used in the prototype: weather, jokes, stocks, top songs, and pizza ordering. The weather service provides the weather for a given location. The joke service provides jokes for a given topic. The stock service provides stock prices for ticker symbols. The song service provides names of the current top songs. Finally, the pizza service simulates ordering pizzas. We created a web application to facilitate the user interactions, manage requirements, and manage service specifications. We used five requirement statements to describe the desired functionality.

As seen in Figure 6, to begin interactions, the user either speaks into a microphone or types a query. The recorded audio is passed to Amazon Lex to convert from speech to text. In the case of the user typing in the query, this step is skipped since the text is already available. The text input is then passed to Amazon Comprehend for analysis. Next, a custom-developed component, called the Service Invocator, executes an orchestration plan based on the intent that matches the invocation phrase. The result of the execution is then passed back to the web GUI as text and audio. To generate audio, we use the Amazon Polly text-to-speech service.

The prototype parses the OpenAPI specification for each microservice and creates the SM. Then it creates the IGM based on the signatures and semantic metadata for each endpoint. We use Comprehend to perform key phrase detection to generate n-gram frequency statistics from the requirement statements. The least common unigrams and bigrams serve as the intents. We map the intents to the graphs in the IGM to create the potential interaction flows. The mapping is done by key phrase similarity matching between the intents and the endpoint metadata. For this prototype, most graphs in the IGM only contain a single node, but the pizza ordering flow contains multiple nodes. This is expected because the intent to check the weather only requires invoking a single microservice, whereas ordering a pizza requires invoking multiple microservices. Ranking the orchestration plans is straightforward in the prototype as we have a very small number of graphs.

After the set of intents is determined and mapped to the IGM, we use a neural network model in Amazon Lex and train it with sample invocation phrases to correctly map user utterances to intents. We employ a simple strategy to generate these sample training invocation phrases. We use the requirement statements, intents, keywords, and entities, then combine them with invocation words (e.g. *what, how, where, want, can, will*) and common verbs to form sentences that are

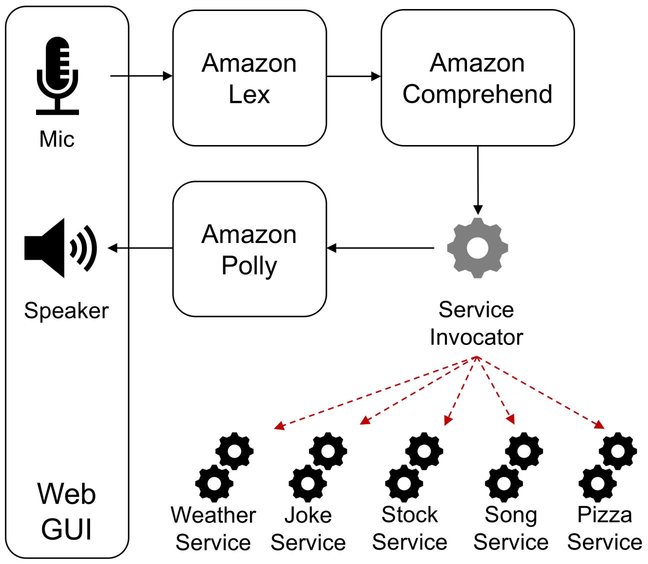
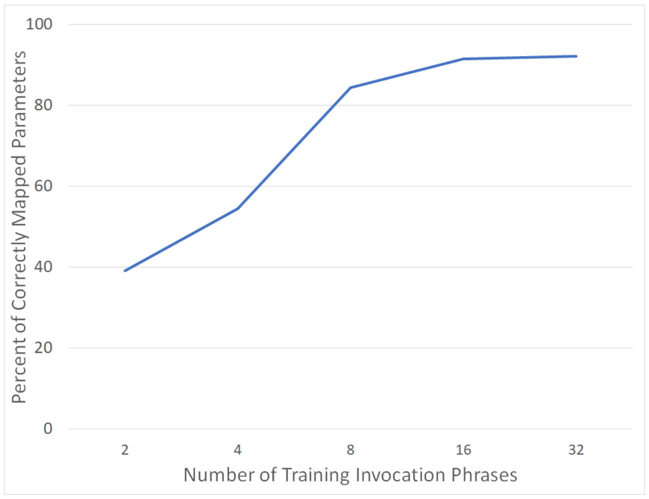
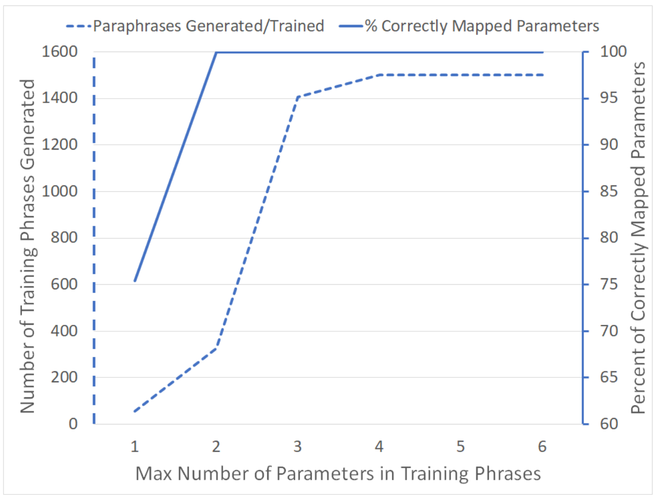


Figure 6: Prototype System



(a)



(b)

Figure 7: (a) Parameter mapping, (b) Maximum parameters in invocation phrase training

in the first person perspective (e.g. *I want to order a pizza*). While generating training invocation phrases, we have to properly incorporate the SM parameters. Figure 7a shows how well training phrases mapped parameters for the pizza order intent as a function of the number of training phrases used. For this training, 500 phrases were generated with parameter names derived from the SM. From these 500 phrases, we set aside separate groups of 2, 4, 8, 16, and 32 randomly-selected phrases (for a total of 62) to be used as training data. We then created five different bots, each trained with one of these training sets. The remaining 438 phrases from the original 500 were used as test data. Each bot was tested using these same 438 test phrases and the percent of parameters that correctly resolved was calculated. Figure 7a shows that there is initially a significant increase in the parameter resolution as the number of training phrases increases, but then it begins to plateau. Additionally, to determine a rough threshold on the number of training phrase parameters needed to build a bot that could reliably map invocation phrases with potentially unlimited parameters, we created multiple bots with different numbers of training phrase parameters. Figure 7b shows this testing. For example, there was a bot trained with phrases that each had a maximum of 2 parameters. It was trained with 327 phrases (created from permutations of the parameters, common verbs, and invocation words) and when tested against invocation phrases with at least 2 parameters (and up to 6), it correctly mapped 100% of the parameters. As seen in Figure 7b, training a model using phrases with only to 2 parameters was sufficient to correctly map invocation phrases with up to 6 parameters.

It took 58 seconds to automatically generate a pizza ordering Lex chatbot with 1500 (the Lex limit at the time of experimentation) sample invocation phrases. It took a trained user 7.5 minutes (772% longer) to manually build the same chatbot with 10 sample phrases. Of that time, the user spent over 2 minutes creating the invocation phrases, so we can project that it would take nearly 6 hours (37,000% longer) to manually build the same 1500-phrase bot. Although it can be argued that using phrases created by a trained user would be of better quality (Figure 5b shows the system generating some non-sensical phrases), this may not necessarily be the case since automatic generation provides a much higher quantity of phrases, which allows for a more varied set of user utterances. The quantity and diversity of phrases created automatically would be sufficient as the goal is to provide varied coverage for the intents. Also, manually building, updating, or maintaining the interface supporting all of the intents would take much longer. From this we see that not only is it possible to regenerate the user interface without additional development effort, but it also takes much less time.

Lastly, to evaluate the flexibility of the user interface, we removed some of the requirement statements and then regenerated the interface. We verified that the intents supporting these requirement statements were no longer handled by the system. Similarly, we removed some of the available underlying microservices while leaving the original set of requirement statements and rebuilt the interface to verify that the system no longer handled the intents affected by these microservices. This shows that our approach is dynamic and allows us to add, remove, and update services on the fly without additional development.

6 CONCLUDING REMARKS AND FUTURE WORK

We have presented a novel way to automatically generate natural language interfaces using microservices. Our approach shows that user intents can be understood from simple requirement statements and satisfied by underlying services. By exploiting the OpenAPI specification and NLP, we are able to dynamically orchestrate microservices and fulfill user intents. Even though the initial prototyping efforts using a handful of microservices were fruitful, challenges remain with more complex conversational interactions (e.g. account for contextual factors such as sentiment, personality, emotion, dialogue state). We plan on exploring ways to integrate adaptation algorithms to tackle these types of personalization and conversation modeling issues. Our requirements statements are also simple and do not capture more nuanced capability descriptions (e.g. use cases and user stories). In the future, we plan to address these challenges, improve our heuristics to scale, and deploy the system in an enterprise environment. We plan to explore ways to utilize centralized logging and apply machine learning techniques to improve our orchestration plan ranking. We also plan to explore GraphQL [28] as an alternative to OpenAPI, and explore synergy with service meshes such as Istio [29].

GraphQL is an up-and-coming API standard that provides a more efficient, flexible, and robust alternative to the widely used OpenAPI standard. Unlike RESTful APIs, GraphQL exposes only a single endpoint that provides the ability for callers to specify which data to return as well as mutators to update the data. By using GraphQL, we may be able to simplify our model generation as there is only one endpoint, and all entities and mutators are predefined.

A service mesh is a dedicated layer on top of microservices to connect, manage, and secure the services. As the number of services grows, so does the complexity of their connectivity and management. This infrastructure layer routes requests and optimizes traffic flow between microservices. However, the management of the mesh itself is a laborious task and it may be possible to extend our technique to build a mesh automatically or annotate it to fulfill user intents.

REFERENCES

[1] Iosif Alvertis, Michael Petychakis, Fenareti Lampathaki, Dimitrios Askounis, and Timotheos Kastrinogiannis. 2014. A community-based, graph API framework to integrate and orchestrate cloud-based services. In *Computer Systems and Applications (AICCSA), 2014 IEEE/ACS 11th International Conference on*. IEEE, 485–492.

[2] Amazon.com, Inc. 2020. Alexa Skills. https://developer.amazon.com/alexa-skills-kit.

[3] Amazon.com, Inc. 2020. Amazon Alexa. https://developer.amazon.com/alexa.

[4] Amazon.com, Inc. 2020. Amazon Comprehend. https://aws.amazon.com/comprehend/.

[5] Amazon.com, Inc. 2020. Amazon Lex. https://aws.amazon.com/lex/.

[6] Amazon.com, Inc. 2020. Amazon Polly. https://aws.amazon.com/polly/.

[7] Apple Inc. 2020. Apple Siri. https://www.apple.com/ios/siri/.

[8] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 748–758.

[9] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

[10] Håkan Burden and Rogardt Heldal. 2011. Natural language generation from class diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*. ACM, 8.

[11] Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S Lam. 2019. Genie: a generator of natural language semantic parsers for virtual assistant commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 394–410.

[12] Jonathan Chevelu, Thomas Lavergne, Yves Lepage, and Thierry Moudenc. 2009. Introduction of a new paraphrase generation tool based on Monte-Carlo sampling. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*. Association for Computational Linguistics, 249–252.

[13] Ekaterina Novoseltseva. 2017. Benefits of Microservices Architecture Implementation. https://dzone.com/articles/benefits-amp-examples-of-microservices-architectur.

[14] Ethan Fast, Binbin Chen, Julia Mendelsohn, Jonathan Bassen, and Michael S Bernstein. 2018. Iris: A conversational agent for complex tasks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 473.

[15] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine.

[16] Jun Fujima, Aran Lunzer, Kasper Hornbæk, and Yuzuru Tanaka. 2004. Clip, connect, clone: combining application elements to build custom interfaces for information access. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*. ACM, 175–184.

[17] Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. 2013. PPDB: The paraphrase database. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 758–764.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[19] Google LLC. 2020. Google Assistant. https://assistant.google.com/.

[20] International Business Machines Corporation. 2019. IBM OpenWhisk. https://developer.ibm.com/code/open/projects/openwhisk/.

[21] Mohit Jain, Pratyush Kumar, Ramachandra Kota, and Shwetak N Patel. 2018. Evaluating and informing the design of chatbots. In *Proceedings of the 2018 Designing Interactive Systems Conference*. ACM, 895–906.

[22] Donghwi Kim, Sooyoung Park, Jihoon Ko, Steven Y Ko, and Sung-Ju Lee. 2019. X-Droid: A Quick and Easy Android Prototyping Framework with a Single-App Illusion. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM, 95–108.

[23] Bing Liu and Ian Lane. 2016. Attention-based recurrent neural network models for joint intent detection and slot filling. *arXiv preprint arXiv:1609.01454* (2016).

[24] Yuval Marton, Chris Callison-Burch, and Philip Resnik. 2009. Improved statistical machine translation using monolingually-derived paraphrases. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*. Association for Computational Linguistics, 381–390.

[25] Grégoire Mesnil, Xiaodong He, Li Deng, and Yoshua Bengio. 2013. Investigation of recurrent-neural-network architectures and learning

methods for spoken language understanding. In *Interspeech*. 3771–3775.

[26] Microsoft Corporation. 2020. Microsoft Cortana. https://www.microsoft.com/en-us/cortana.

[27] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems*. "O'Reilly Media, Inc.".

[28] Open Source Contributors. 2020. GraphQL. https://graphql.org/.

[29] Open Source Contributors. 2020. Istio. https://github.com/istio/istio.

[30] Open Source Contributors. 2020. OpenAPI Specifications. https://github.com/OAI/OpenAPI-Specification.

[31] Karen Spärck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation* 28, 1 (1972), 11–21.

[32] Techopedia. 2020. Intelligent Virtual Assistant. https://www.techopedia.com/definition/31383/intelligent-virtual-assistant.

[33] Gökhan Tür, Anoop Deoras, and Dilek Hakkani-Tür. 2013. Semantic parsing using word confusion networks with conditional random fields. In *INTERSPEECH*. 2579–2583.

[34] Wikipedia. 2020. N-gram. https://en.wikipedia.org/wiki/N-gram.

[35] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. 2016. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. ACM, 5.

[36] Kaisheng Yao, Baolin Peng, Geoffrey Zweig, Dong Yu, Xiaolong Li, and Feng Gao. 2014. Recurrent conditional random field for language understanding. In *Acoustics, Speech, and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 4077–4081.

[37] Shiqi Zhao, Xiang Lan, Ting Liu, and Sheng Li. 2009. Application-driven statistical paraphrase generation. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2*. Association for Computational Linguistics, 834–842.