Generating Chat Bots from Web API Specifications

Mandana Vaziri
IBM Research, USA
mvaziri@us.ibm.com

Louis Mandel
IBM Research, USA
lmandel@us.ibm.com

Avraham Shinnar
IBM Research, USA
shinnar@us.ibm.com

Jérôme Siméon
IBM Research, USA
simeon@us.ibm.com

Martin Hirzel
IBM Research, USA
hirzel@us.ibm.com

Abstract

Companies want to offer chat bots to their customers and employees which can answer questions, enable self-service, and showcase their products and services. Implementing and maintaining chat bots by hand costs time and money. Companies typically have web APIs for their services, which are often documented with an API specification. This paper presents a compiler that takes a web API specification written in Swagger and automatically generates a chat bot that helps the user make API calls. The generated bot is self-documenting, using descriptions from the API specification to answer help requests. Unfortunately, Swagger specifications are not always good enough to generate high-quality chat bots. This paper addresses this problem via a novel in-dialogue curation approach: the power user can improve the generated chat bot by interacting with it. The result is then saved back as an API specification. This paper reports on the design and implementation of the chat bot compiler, the in-dialogue curation, and working case studies.

CCS Concepts • **Software and its engineering → Domain specific languages**;

Keywords Conversational agents, cloud, REST, compilers

ACM Reference Format:
Mandana Vaziri, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Martin Hirzel. 2017. Generating Chat Bots from Web API Specifications. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 14 pages.
https://doi.org/10.1145/3133850.3133864

Onward!'17, October 25–27, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5530-8/17/10…$15.00
https://doi.org/10.1145/3133850.3133864

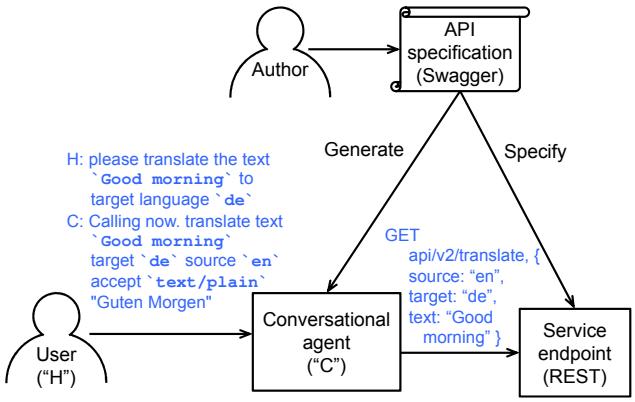

Figure 1. Calling an API with SwaggerBot.

1 Introduction

Companies increasingly rely on chat bots to offer support and services to their customers and employees. Chat bots, or conversational agents, communicate with users via natural-language dialogue. Thanks to recent technological advances, chat bots are starting to see wide-spread adoption [13]. They can be accessed through a web page, a phone, or a messaging system. They are programmed to answer commonly asked questions, help navigate a web page more effectively, or fill out online forms.

Like graphical applications, chat bots offered by companies usually accomplish their work by calling web Application Programming Interfaces (APIs). These APIs offer customers and employees access to the resources stored in the company's databases and the actions implemented in the company's systems. Today, the dominant approach for providing web APIs is REST (REpresentational State Transfer) [21]. In REST, the service provider hosts resources, and the provider and consumer interact by transferring representations (typically JSON: JavaScript Object Notation) of the state of the resources.

Unfortunately, chat bots are difficult to build. Like any piece of software, it takes time to get them right. But unlike other software, chat bots depend upon NLU (natural language understanding), which is imperfect. Since it is hard to

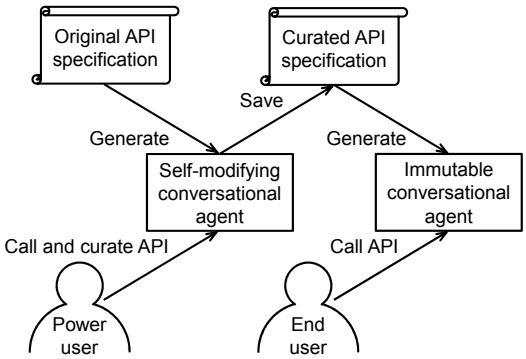

Figure 2. Curating an API with SwaggerBot.

anticipate what the common NLU mistakes will be, it pays to deploy an initial bot quickly and gather experience with it in the field to improve it [25]. Chat bot development is further complicated by requiring both software development and machine learning skills. And finally, since companies have only recently started to broadly embrace chat bots, there is a lack of programming models for non-experts to develop chat bots [9].

This paper presents a compiler that takes a Swagger OpenAPI specification [20] and automatically generates a chat bot that helps the end user call the corresponding web API. Many web APIs already have specifications written in the Swagger format [20]. Swagger is popular as a source language for generating a variety of artifacts including API documentation, client SDKs (software development kits), server stubs, and tests, but has not been previously used for generating chat bots. The first contribution of this paper is to use Swagger as a source language for generating chat bots that enable end users to call a web API via natural-language dialogue.

Figure 1 shows the overall approach presented in this paper, including a simple but working example (more full-fledged examples come later in the paper). The user ("H" for human) converses with the chat bot ("C" for conversational agent) by asking to translate some text. In this example, the user is already aware of the necessary parameters and knows to back-quote them. The chat bot fills in missing parameters (source via call-chaining and accept via defaults, Section 3.3), then makes a REST call via the HTTP GET method to the translator service, and returns the result to the user. The vision behind our approach is to enable end users to call web APIs without learning them first. Later sections will show dialogues that use reflection to let the end user discover available actions and their parameters. Our chat bot compiler enables developers to reuse existing API specifications to quickly boot-strap a full working bot.

The quality of the dialogues between a generated bot and the end user hinges on the quality of the Swagger it is generated from. Unfortunately, not all Swagger specifications are high-quality, or specific to a user's needs, making the resulting chat bot harder to use than necessary. Swagger specifications may be incomplete by omitting things like descriptions or types for parameters. They may specify some constraints informally in descriptions rather than formally in types. And they may require more details than should be exposed to end users. Swagger specifications may also be too general for a user's needs. For example, for a Translation API, a user may only want to translate from French to English, and not any other languages. In those cases, the Swagger specification needs to be improved or specialized to obtain a better chat bot.

To reduce the need for manually editing Swagger specifications, our generated bots support in-dialogue curation. In programming-languages terms, if the focus for the end user was on calling existing functions, the power user can also define new functions by aliasing and partial application. Specifically, generated bots allow adding new actions that represent useful shortcuts and usage scenarios for the end user, while interacting with the bot in natural language, and without specific knowledge of REST APIs or the format of Swagger specifications.

Figure 2 illustrates this in-dialogue curation process. The first step is to generate a conversational agent from an API specification and to have the power user call and curate the API via dialogue. Saving the results yields a curated API specification. The second step is to generate another conversational agent and have the end user call the API via dialogue, as shown earlier in Figure 1. Besides requiring less coding skills, in-dialogue curation offers fluidity, where the power user can seamlessly move back and forth between calling and curating the web API. This approach, supporting both end users and power users within a single tool, has been successful in other contexts, such as spreadsheet tools. Furthermore, it helps us focus on improving the conversational interface, since that yields the dual benefit of improving both the calling experience and the curation experience.

This paper makes the following contributions.

- A chat bot compiler from web API specifications, generating self-documenting chat bots for calling APIs.
- In-dialogue curation of generated chat bots using natural language.
- A prototype implementation for the compiler and a realization in the Slack messaging system.
- Case studies on a variety of API specifications.

Overall, this paper presents contributions to programming models for both web APIs and chat bots.

```
1  {swagger: "2.0",
2   info: { version: "2.0.0", title: "Language Translator" },
3   basePath: "/language-translator/api",
4   paths: {
5    "/v2/identify": {
6     get: {
7      summary: "Identifies the language of the input text",
8      parameters: [
9       { name: "text", in: "query", required: true,
10        type: "string",
11        description: "Input text in UTF-8 format." },
12      { name: "accept", in: "header", required: false,
13        type: "string", default: "text/plain",
14        enum: ["application/json", "text/plain" ],
15        description: "Format of the return values."}],
16     responses: {
17      200: {
18       description: "Success",
19       schema: { $ref: "#/definitions/identifiedLanguages" }},
20      400: ..., 500: ... } },
21    post: ...},
22   "/v2/translate": ...,
23   "/v2/identifiable_languages": ...,
24   "/v2/models": ...,
25   "/v2/models/{model_id}": ... },
26  definitions: {
27   identifiedLanguages: {
28    required: [ "languages" ],
29    properties: {
30     languages: {
31      type: "array", description: ...,
32      items: { $ref: "#/definitions/identifiedLanguage" }}}},
33   identifiedLanguage: {
34    required: [ "language", "confidence" ],
35    properties: {
36     language: { type: "string", description: ... },
37     confidence: { type: "number", description: ... }}},
38   ... }}
```

Figure 3. Swagger for Translator web API [10].

2 Example

This section gives a concrete example, using transcripts of dialogues with a bot generated by our compiler. The example is based on a Language Translator web API, which can translate between different natural languages, but also supports additional features such as identifying a language [10].

The input to the bot compiler is a web API specification in Swagger. Figure 3 shows the Swagger for the Translator service. The generated bot (which we call SwaggerBot) uses the

```
1  C: Welcome to the `Language Translator`.
2     SwaggerBot is here to help you.
3     What would you like to do? To begin name an action or
4     resource.
5  H: help me please
6  C: You may work with the following resources and actions:
7     - identifiable_languages
8     - identify
9     - models
10    - models model_id
11    - translate
12    What would you like to do? To begin name an action or
13    resource.
14 H: identify please
15 C: You chose: identify.
16    What operation would you like to perform?
17 H: I have no idea, pleas help
18 C: Available operations are:
19    (GET) Identifies the language of the input text,
20    (POST) Identifies the language of the input text
21 H: get then
22 C: What is the value of the parameter text?
23 H: La vie est belle
24 C: identify text `La vie est belle`
25    OK, call now?
26 H: yes
27 C: OK, making call now.
28    identify text `La vie est belle` accept `text/plain`
29    "fr"
```

Figure 4. Example dialogue calling the Translator web API.

Swagger as guidance both for talking to the user (in natural language) and for calling the web API (in HTTP). Lines 1–3 specify the Swagger version, API version and title, and base path. Lines 4–25 specify paths relative to the base path for resources (nouns, e.g., identifiable languages) and actions (verbs, e.g., identify). For brevity, we show details for only one of the actions, addressed by the path "/v2/identify". This path supports two operations, get and post, corresponding to the HTTP methods of the same names. Each operation has a summary, parameters, and responses, which can have both human-readable descriptions and schemas. In the example, the schemas involve references ($ref) to type definitions provided in Lines 26–38.

Figure 4 shows a transcript of a dialogue between the generated SwaggerBot conversational agent ("C") and a human end user ("H") for calling the Translator web API. We generated the chat bot directly from the Swagger in Figure 3. On Line 4, the end user asks for help; note that rather than having to learn a computer language to know how to ask

```
1  H: identify the text `La vie est belle`
2  C: Calling now.
3      identify text `La vie est belle` accept `text/plain`
4      "fr"
5  H: translate the text `La vie est belle` from source `fr`
6      to target `en`
7  C: Calling now.
8      translate text `La vie est belle` source `fr` target `en`
9      accept `text/plain`
10     "Life is Beautiful"
11 H: name this action
12 C: What would you like to name this action?
13 H: anglicize
14 C: Thanks, action anglicize created.
15 H: anglicize
16 C: Calling now.
17     translate text `La vie est belle` source `fr` target `en`
18     "Life is Beautiful"
19 H: anglicize with something else for text such as `Le chat dort`
20 C: Calling now.
21     translate text `Le chat dort` source `fr` target `en`
22     "The cat sleeps"
23 H: I would like to add a synonym for an action.
24 C: On which action or resource would you like to add a
25     synonym?
26 H: On the action translate.
27 C: What synonym would you like to add for translate?
28 H: convert
29 C: Ok, adding convert as a synonym for translate.
```

Figure 5. Example dialogue curating the Translator API.

for help, they can just use natural language. SwaggerBot is configured to understand the help intent via a machine learning classifier trained from examples; Line 14 shows a different utterance that also gets classified into the help intent. Lines 5–11 show help text generated from the Swagger paths. On Line 12, the end user picks the action identify. Subsequently, the chat bot collects additional information it needs to call identify. After the human has provided the operation (get) and the text parameter (La vie est belle), the chat bot calls the web API and displays the result ("fr" for French). Along the way, the bot also displays the action and its parameters concisely using back-quotes. This has two effects. First, confirmation to establish common ground with the user [5]; and second, educating the user how to make this call more easily.

Figure 5 shows a transcript of a dialogue with a power user for calling and curating the Translator web API. On Line 1, the power user directly calls the identify action, compressing the "H" utterances from Figure 4 into a single utterance. On Line 5, the power user directly calls the translate action, again providing all necessary information in a single utterance. On Lines 11–14, the power user curates the API by creating a new action, anglicize, that encapsulates the preceding call.

On Lines 15–18, the power user tests anglicize, fluidly moving back from curating to calling the web API. With this gesture, the power user has created a new action in the API called anglicize, which is serviced by the translate action with specific parameters. It is a shortcut for translating a specific text from French to English. On Lines 19–22, the power user calls anglicize but provides a different text parameter, leading to a different result. This showcases that the additional action can now be used with variations in parameters, and simplifies the task of making API calls. Finally, on Lines 23–28, the power user adds a synonym for translate, enriching the natural language understander involved in recognizing this action.

Now that we have seen a SwaggerBot in action, we will look at how it is generated and how it works.

3 End-User Dialogue for API Calls

As mentioned before, a generated SwaggerBot conversational agent serves two personas: it enables the end user to call a web API, and it enables the power user to call and curate a web API. This section focuses on API calls by the end user, leaving API curation to the next section. Figure 6 shows the runtime architecture. The centerpiece is the dialogue controller, which guides the conversation for calling web APIs. Like in most chat bots [19], the controller receives inputs from the user via an NLU (natural language understander) component, and sends outputs to the user via an NLG (natural language generator) component. SwaggerBot agents are built on the WCS (Watson Conversation Service) platform for conversational agents [11].

In WCS, NLU consists of an entity extractor and an intent classifier, which a SwaggerBot customizes for the Swagger at hand. In addition, a SwaggerBot adds a direct call parser as another NLU component not usually found in WCS or other platforms. Section 3.1 elaborates further on the NLU component. In WCS, the controller can be driven from an FSM (finite-state machine) dialogue specification. Section 3.2 elaborates on the mapping from Swagger to the controller FSM, and on additional state that the controller maintains. The output of the controller consists of natural-language prompts for the human and HTTP calls to service endpoints. Section 3.3 describes the NLG and actuator components that implement these outputs.

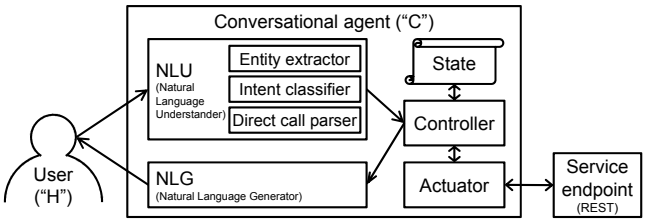

Figure 6. SwaggerBot runtime architecture.

3.1 Input: NLU

The NLU component in Figure 6 turns natural-language utterances from the user into symbols for the dialogue controller. Most modern chat bot platforms recognize two kinds of symbols: entities and intents.

An *entity* represents a term or object in a user utterance, and WCS marks entities with the @-sigil [11]. For example, in Figure 4 Line 18, the word get belongs to the @get entity. The entity extractor in the NLU is implemented by a pattern and may return multiple entities for a single user utterance, one for each matching term. Some entities are common to all SwaggerBot agents independently of the concrete Swagger specification at hand, for instance, @get and @post. In addition, some entities are generated from parameters found in the Swagger specification, for instance, @text, @source, and @target.

An *intent* represents a purpose or goal, something a user wants to do, and WCS marks intents with the #-sigil [11]. For example, in Figure 4 Line 4, 'help me please' belongs to the #help intent. The intent classifier in the NLU returns the intent with the highest confidence for a given utterance, or a special #irrelevant intent if nothing has high confidence. The intent classifier is implemented with supervised machine learning, where the training data consists of ⟨example, intent⟩ pairs. The intent classifier works best when there are many examples for each intent, examples for different intents are not similar, and the examples are representative of actual user utterances. For instance, training examples for the #help intent might include 'help', 'What are the options?', 'What can I do?', and 'what is possible'. There are some intents common to all SwaggerBot agents independently of the concrete Swagger specification at hand, for instance, #help, #yes, and #no. In addition, there are intents generated from paths found in the Swagger specification, for instance, #identify and #translate.

While the basic NLU functionality of entities and intents suffices for many chat bots, it turns out to be too limiting for obtaining good conversations for calling a web API. One problem is that some parameters have free-flow values that cannot be easily matched or classified against a pre-defined entity or intent. Furthermore, some inputs should be hidden from the entity extractor and the intent classifier altogether. For example, the text parameter to the identify action can contain arbitrary words that should not trigger their own entities or intent. Therefore, when SwaggerBot prompts for such a parameter, it treats the entire next human utterance as one value, as shown in Figure 4 Line 20.

While this solves the problem, it unfortunately requires a separate turn for each piece of information, leading to a prolonged dialogue. Therefore, we introduced a quoting feature. We settled on backquotes (`...`), because they are familiar to users of the Slack messaging platform for rendering verbatim text. SwaggerBots can be deployed on Slack, and also need quotes to signal verbatim text. In addition, we introduced a convention by which a parameter name entity in the utterance followed by quoted text sets the parameter to the quoted value. This convention makes it possible to render an API call in a single utterance, and is implemented by the direct call parser. For example, in Figure 5 Line 1, 'identify the text `La vie est belle`' calls the identify action, setting the text to 'La vie est belle', and defaults to using the HTTP GET method.

3.2 Controller and State

The controller component in Figure 6 maintains state and turns symbols from the NLU into instructions to the NLG and the actuator. The most common low-level formalism for specifying dialogue flow in chat bots is FSMs [19]. The WCS programming model supports FSMs, among other features [11]. The current implementation of the SwaggerBot compiler uses FSMs as its code-generation target (the source being Swagger, of course). Figure 7 depicts an excerpt of the FSM that the SwaggerBot compiler generates from the Translator Swagger in Figure 3. The following text explains the notation and discusses the dialogue flow it specifies.

Each rounded rectangle in Figure 7 represents a state. There is one special start state marked with an incoming arrow that does not originate from any other state. There are several final states, marked with double borders. There is an implicit top-level loop from final states back to the start state. Directed edges between states are transitions and their labels are predicates. State labels have the form stateName / action, but most state names are omitted for brevity. Since actions reside on states, not on transitions, the FSM is a Moore machine, not a Mealy machine. We chose Moore machines as the formalism because the WCS programming model supports Moore machines.

Transition predicates are based on symbols from the NLU, i.e., entities and intents, marked with @ and #, respectively. Some transitions are marked with the empty-word symbol ε, indicating that the chat bot does not wait for user input before taking the transition. Finally, some transitions are

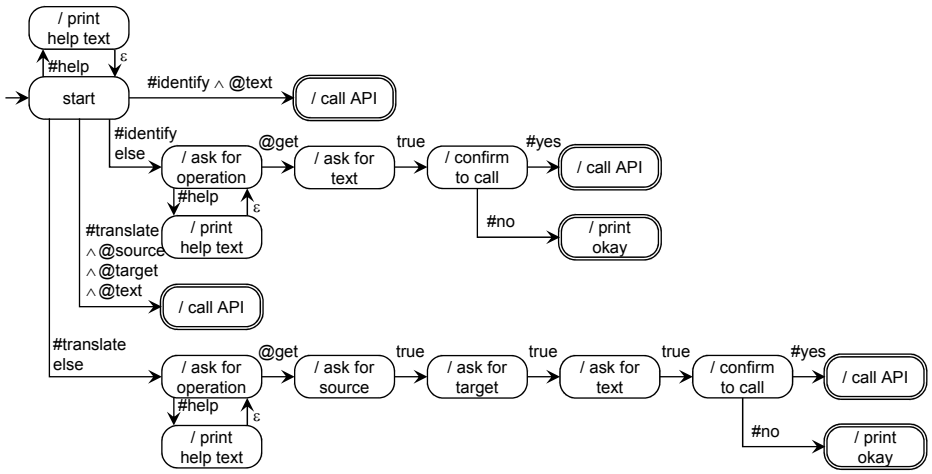

Figure 7. Excerpt of SwaggerBot state machine for Translator web API.

marked with the true predicate, indicating that the chat bot collects a user input, treating the entire utterance as one value without the transition depending on that value. Actions in FSM states are instructions to the NLG and the actuator. For instance, the print help text action is an instruction to the NLG, and the call API action is an instruction to the actuator. For brevity, Figure 7 does not spell out the details of the actions, but they are context specific. The context for help text is the current FSM state, and context for an API call consists of the current FSM state as well as the operation and parameters collected in states leading up to it.

The dialogue flow in Figure 7 shows five transitions from the start state.

- From start, if #help, the bot prints the list of known resources and actions, generated from the paths in Swagger. SwaggerBot implements a heuristic to elide common path prefixes such as "/v2/" in the Translator Swagger because they provide no useful information and cause unnecessary confusion to end users. After displaying the help text, the chat bot returns to the start state without collecting a user input, indicated by the ε-transition.
- From start, if #identify ∧ @text, the direct call parser has provided all the necessary information to call the "/v2/identify" path with the required text parameter. If no HTTP method is specified, the method defaults to GET if the path supports that. SwaggerBot implements a heuristic to not ask for optional parameters that have a default value, such as the accept parameter in this case. Furthermore, the accept parameter implements content negotiation, a feature general to REST and not specific to the Translator API. Content negotiation is an implementation technicality better hidden from the non-technical end user.
- From start, if #identify but the previous transition did not fire, the chat bot has a chain of nodes collecting the operation and the parameters for calling the "/v2/identify" path. In this context, the help text lists operations for the given path. The figure only shows the FSM states for @get, eliding those for @post. This part of the FSM also contains an example of a true condition, because the text parameter can be any string and should thus not be subjected to NLU.
- From start, if #translate ∧ @source ∧ @target ∧ @text, the direct call parser has provided all the necessary information to call the "/v2/translate" path.
- From start, if #translate but the previous transition did not fire, the chat bot has a chain of nodes collecting the operation and the parameters for calling the "/v2/translate" path.

The state in Figure 6 thus consists of the FSM state plus partial information collected by the current part of the dialogue flow towards the goal of making an API call.

3.3 Output: NLG and Actuator

The NLG components in Figure 6 turn instructions from the dialogue controller into natural-language responses to the user and HTTP calls to the REST service endpoint. To encapsulate the controller with a clean interface, our implementation reifies these instruction as JSON objects.

Most chat bots use a very simple NLG [19], and SwaggerBot is no exception. For the most part, the SwaggerBot NLG consists of literal hard-coded strings, some of which are generated from the Swagger at compile time, for example, the list of paths in Figure 4 Lines 6–10. In some cases, the NLG

```
1  "/v2/translate": {
2   get: {
3    summary: "Translates the input text from the source ...",
4    parameters: [ ...,
5     { name: "source", in: "query", required: false,
6       type: "string",
7       description: "Used in combination with target as an ...",
8       "x−sb−callchaining": {
9          base: "/language−translator/api"
10         path: "/v2/identify",
11         method: "GET",
12         params: [
13            { name: "text", in: "query", value: "$.text" } ] } },
14    ... ], ... }, ... }
```

Figure 8. Annotation on the source parameter of a translate call to chain it to an identify call in the same web API.

also uses string interpolation at runtime, for example, for the confirmation 'Thanks, action anglicize created' in Figure 5 Line 14, where the action name, anglicize, is interpolated into a string template.

The actuator is in charge of making the HTTP calls to the web API. Our prototype implementation of SwaggerBot is written in Java and uses a simple HTTP client to execute a synchronous call to the service endpoint. Currently, the actuator is deployed as a local application on the end user's device. The NLU and most of the controller, on the other hand, are deployed as a cloud-hosted application in the Watson Conversation Service.

Finally, the actuator is in charge of filling in default values for missing parameters, when possible. This comes in two flavors. First, Swagger files sometimes contain a constant default value, such as `default: "text/plain"` in Figure 3 Line 13. Second, we invented a Swagger extension to indicate a non-constant default value, which calls other web API paths to fill in a parameter value. In Figure 1, the accept parameter is set by a constant default, whereas the source parameter is set by using our extension, explained below.

Figure 8 shows the corresponding annotations for the source language parameter of the translate call in the Translator web API. It indicates that if the source parameter is not provided by the user, a default can be obtained by calling identify with the input text. The value of the appropriate parameter is obtained from the current context if it is present. The call chaining annotation is read as part of the Swagger file and passed to the actuator. It contains all the information necessary to make a sub-call to another part of the API.

This section described the facet of a generated SwaggerBot that is concerned with enabling end users to call web APIs. Since this facet does not use self-modifying curation, for the most part, it is about compilation from an API specification to a dialogue specification. Before generated SwaggerBots, this translation had to be done by hand, involving developers and subject matter experts. The contribution of this section is to recognize that it can be automated; show what the automation looks like; and describe heuristics leading to a more usable chat bot.

4 Power-User Dialogue for API Curation

Generating useful chat bots for web APIs hinges on the quality of the Swagger specifications. These could be incomplete in many ways. First, there may be missing or insufficient summaries and descriptions, which we use to generate help sentences. There could be missing default values, which could have been used by generated bots to streamline calling. The Swagger specification may also simply be out-of-date with respect to the actual service it represents. For some of these issues, such as adding a description, editing the Swagger specification is simple enough. Others, such as creating shortcuts, require deeper knowledge of REST APIs and the Swagger format. We help with this task by providing in-dialogue curation of chat bots. Currently, our generated chat bots offer two features for curation: creating new actions (Section 4.1) and creating synonyms for existing actions (Section 4.2).

4.1 Creating New Actions

Creating a new action helps the end user quickly access an API and make calls without the full knowledge of all the parameters that must be provided. Part of the difficulty in making API calls is knowing exactly what values to supply for each parameter. By creating shortcuts in the form of new actions, the power user simplifies this task for the end user and specializes the API.

To create a new action, the power user makes an utterance that matches the intent to 'name this action', which refers to the very last executed API call. The bot prompts the user for a name, then updates its internal representation of the Swagger specification with a new path having this name, with the corresponding HTTP method, and with parameters having the default values of the last executed call. It then regenerates the low-level dialogue specification from the modified Swagger. Specifically, the name of the action becomes a new NLU intent for WCS, and in addition, the FSM for WCS is extended with new states corresponding to the new action. Therefore, the creation of an action causes two events:

- The Swagger specification gets updated in-memory with a new path (which could be saved to disk).

```
{swagger: "2.0",
 basePath: "/tone-analyzer/api",
 info: {
  version: "3.0.0", title: "Tone Analyzer API",
  description: "Detect three types of tones from written text:
                emotions, social tendencies, and style. ..." },
 paths: {
  "/v3/tone": {
   get: {
    summary: "GET Analyze tone",
    description: "Analyzes the tone of a piece of text.",
    parameters: [
     { name: "version", in: "query", required: true,
       type: "string", default: "2016-05-19",
       description: "a date that identifies the API version" },
     { name: "text", in: "query", required: true,
       type: "string",
       description: "Content to be analyzed." },
     { name: "tones", in: "query", required: false,
       type: "string",
       description: "Filter the results by a specific tone." },
     { name: "sentences", in: "query", required: false,
       type: "boolean",
       description: "Filter your response to remove the
                     sentence level analysis." }],
    responses: {
     200: {
      description: "successful operation",
      schema: { "$ref": "#/definitions/ToneAnalysis" }},
     400: ... }},
    post: ... }},
 definitions: {
  ToneAnalysis: {
   description: "The results of performing tone analysis.",
   required: [ "document_tone" ],
   properties: {
    document_tone: {
     type: "array", description: ...,
     items: { "$ref": "#/definitions/ElementTone" }},
    sentences_tone: {
     type: "array", description: ...,
     items: { "$ref": "#/definitions/SentenceAnalysis" }}}},
  ElementTone: ...,
  ... }}
```

Figure 9. Swagger for Tone Analyzer web API.

```
C: What would you like to do? To begin name an action or
   resource.
H: What's the tone of the text `I am very happy`?
C: Calling now.
   tone text `I am very happy` version `2016-05-19`
   { "document_tone": {
       "tone_categories": [
        {
         "tones": [
           { "score": 0.0, "tone_id": "anger",
             "tone_name": "Anger" },
           { "score": 0.0, "tone_id": "disgust",
             "tone_name": "Disgust" },
           { "score": 0.0, "tone_id": "fear",
             "tone_name": "Fear" },
           { "score": 0.9, "tone_id": "joy",
             "tone_name": "Joy" },
           { "score": 0.0, "tone_id": "sadness",
             "tone_name": "Sadness" } ],
         "category_id": "emotion_tone",
         "category_name": "Emotion Tone"
        }, {
         "tones": [
           { "score": 0.0, "tone_id": "analytical",
             "tone_name": "Analytical" },
           { "score": 0.97759, "tone_id": "confident",
             "tone_name": "Confident" },
           { "score": 0.0, "tone_id": "tentative",
             "tone_name": "Tentative" } ],
         "category_id": "language_tone",
         "category_name": "Language Tone"
        }, {
         "tones": [
           { "score": 0.096859, "tone_id": "openness",
             "tone_name": "Openness" },
           { "score": 0.264058, "tone_id": "conscientiousness",
             "tone_name": "Conscientiousness" },
           { "score": 0.472657, "tone_id": "extraversion",
             "tone_name": "Extraversion" },
           { "score": 0.61522, "tone_id": "agreeableness",
             "tone_name": "Agreeableness" },
           { "score": 0.104851, "tone_id": "emotional_range",
             "tone_name": "Emotional Range" } ],
         "category_id": "social_tone",
         "category_name": "Social Tone"
        } ] } }
```

Figure 10. Example dialogue with Tone Analyzer web API.

- The chat bot self-modifies and changes itself by making a REST call to the WCS service to upload the new NLU intent and controller FSM.

Future mentions of the newly created action cause it to execute and perform a call with all the supplied default values. The end user also has the possibility of supplying variations for the stored parameter values by simply mentioning them when uttering the name of the new action.

4.2 Creating Synonyms for Existing Actions

When the compiler generates a bot from an API specification, all paths become intents with a single example, consisting of the name of the path. This provides little flexibility for the end user who must employ something very close to that name in order to invoke the corresponding action. Like most machine-learning algorithms, the intent classifier that detects classes for human utterances works best if it is trained from many examples. To improve the quality of the bot, one curation activity consists of adding synonyms for existing actions, which serve as additional training data for the classifier. This improves the accuracy of the NLU and thus helps the user employ a larger vocabulary when referring to an action.

Since the NLU is hosted in WCS, adding synonyms is implemented by using the WCS REST API. WCS supports dynamically adding examples for intents. Therefore the chat bot self-modifies again. Both in the case of adding new actions and synonyms, WCS has to retrain its classifiers, which may take a minute or two. Therefore, the effects of these gestures may not be immediately visible in the bot.

These curation features are an example of how to do live-programming for chat bots using conversational gestures.

5 Case Studies

This section presents case studies for two more web APIs beyond the Translator API examples in Section 2. This demonstrates that the initial implementation of SwaggerBot works for more than just one Swagger. This section also showcases integration with the Slack messaging platform [23], which inspired the backquote syntax.

The **Tone Analyzer** analyzes text documents to detect three things: emotions (e.g. anger, joy, sadness), style (e.g. confidence, tentativeness), and social tendencies (e.g., extraversion, agreeableness). Figure 9 shows an excerpt of the web API specification. It offers only a single path, "/v3/tone", with four parameters version, text, tones, and sentences. Three of the four parameters are either optional or have a default value, leaving only the text parameter as mandatory information to be specified for each call. Figure 10 shows the transcript of a SwaggerBot dialogue for calling the Tone Analyzer. The human simply makes a direct call, setting the text to `` `I am very happy` ``.

```
8  "/v3/tone": {
9    get: {
10     "x-sb-datashaping": "$..tones[?(@.score > 0.6)]",
11     ... } }
```

Figure 11. Enhanced fragment of Tone Analyzer web API.

```
2  H: What's the tone of the text `I am very happy`?
3  C: Calling now.
4      tone text `I am very happy` version `2016-05-19`
5    [ { "score": 0.9, "tone_id": "joy",
6          "tone_name": "Joy" },
7      { "score": 0.97759, "tone_id": "confident",
8          "tone_name": "Confident" },
9      { "score": 0.61522, "tone_id": "agreeableness",
10         "tone_name": "Agreeableness" } ]
```

Figure 12. Dialogue with enhanced Tone Analyzer web API

The result of the call is what one would expect given the input text, but is a large JSON structure that is better suited for processing by a computer than for a conversation. To address this issue, SwaggerBot allows the API designer to annotate calls in the Swagger file with JsonPath [6] expressions for formatting the JSON response. Figure 11 presents a fragment that can be added to the Swagger from Figure 9. The x-sb-datashaping JsonPath annotation on Line 10 returns only tone elements whose score exceeds a 0.6 threshold.

This results in the conversation presented in Figure 12, a marked improvement from Figure 10. It makes it easier for the user to see that the analyzer has determined that the dominant emotion is joy, the style is confident, and the strongest social tendency found in this text is agreeableness.

The **Visual Recognition** API analyzes image files to identify three things: scenes, objects, and faces. Figure 13 shows an excerpt of the web API specification. The figure hones in on only one of the paths, "/v3/detect_faces". Unlike in the other two Swaggers earlier in the paper, the parameters are declared by reference instead of inline, because other paths share some of the same parameters. There are three parameters: api_key (a hard-to-guess string of digits and numbers used to authenticate); url (an address of an image file); and version (an API version given as a string). The version has a default value.

Figure 14 shows the transcript of a SwaggerBot dialogue for calling the Visual Recognition API. It is a screenshot of SwaggerBot running integrated into the Slack messaging platform [23]. Slack is often used for persistent multi-party

```
{swagger: "2.0",
 info: {
  title: "Visual Recognition", version: "3.0",
  description: "Uses deep learning algorithms to identify scenes,
                objects, and faces." },
  basePath: "/visual-recognition/api",
  paths: {
   "/v3/detect_faces": {
    get: {
     summary: "Detect faces in an image",
     parameters: [
      { "$ref": "#/parameters/ApiKeyParam" },
      { "$ref": "#/parameters/SingleURLParam" },
      { "$ref": "#/parameters/DateVersionParam" }],
     responses: {
      200: {
       description: "success",
       schema: { "$ref": "#/definitions/Faces" }},
      400: {
       description: "Invalid request",
       schema: { "$ref": "#/definitions/ErrorTopLevel" }}}},
    post: ... },
   ... },
  parameters: {
   ApiKeyParam: {
    name: "api_key", in: "query", required: false, type: "string",
    description: "API Key used to authenticate." },
   SingleURLParam: {
    name: "url", in: "query", required: true, type: "string",
    description: "URL of an image (.jpg, .png). Redirects are followed, so you
                  can use shortened URLs. The resolved URL is returned ..." },
   DateVersionParam: {
    name: "version", in: "query", required: false, type: "string",
    default: "2016-05-20", pattern: "^\\d{4}-\\d{2}-\\d{2}$",
    description: "The release date of the version of the API you want to use.
                  Specify dates in YYYY-MM-DD format." },
   ... },
  definitions: {
   Faces: {
    type: "object",
    required: [ "images_processed", "images", "warnings" ],
    properties: {
     images_processed: { "$ref": "#/definitions/ImagesProcessedOutParam" },
     images: {
      type: "array",
      items: { "$ref": "#/definitions/FacesTopLevelSingle" } },
     warnings: {
      type: "array",
      description: "Omitted if there are no warnings.",
      items: { "$ref": "#/definitions/WarningInfo" }}}},
   ... }}
```

Figure 13. Swagger for Visual Recognition web API.

Figure 14. Example dialogue with Visual Recognition web API.

chat by collaborating teams. In this case, there are two participants, SwaggerBot and a human end user called **louis**. The dialogue showcases a step-by-step call, where the end user asks for help and the chat bot prompts for parameters one by one. The end user copy-and-pastes the API key, which they obtained separately; we redacted ours from the figure. One Slack feature is that when a user pastes a URL, it displays a preview of its destination. In this case, the URL is a picture of a historical personality, and Slack shows an excerpt of the surrounding Wikipedia article. SwaggerBot does not prompt for the version parameter, because the Swagger specifies a default for it. If a user wanted to set it explicitly, they would need to do so via a direct call.

Before making the call, as usual, SwaggerBot echoes the concise version for grounding and teaching. This showcases how backquotes are rendered in Slack markdown. The quoted text shows up in a typewriter font with a different color scheme than normal text. In the end, SwaggerBot makes the call, and the Visual Recognition API reports back its guess for the age, gender, and identity of the historical personality. These guesses turn out to be accurate.

Besides the formatting niceties, there are other good reasons for integrating bots that call web APIs into Slack. Since employees at a company often chat with each other via Slack, putting the chat bot there reduces the need to context-switch. In the calling case, it provides a persistent record of what happened for accountability. It is also an easy way to keep each other informed, for instance, when the web API returns some kind of status report. In the curation case, it fosters collaboration between multiple power users improving the same chat bot. And finally, it increases the learning opportunities, where one user's successful calls are there for other users to emulate.

Between the Translator example from Section 2 and the Tone Analyzer and Visual Recognition examples from this section, this paper showcases three SwaggerBot chat bots generated from three different API specifications.

6 Related Work

The two main ideas in this paper are (i) generating a chat bot from a web API specification and (ii) improving a chat bot with in-dialogue curation. To the best of our knowledge, both of these ideas are novel. This section reviews previous work related to either of these contributions.

6.1 Chat Bots and Web APIs

VoiceXML is a standard that was designed to play the same role for conversations that HTML plays for textual content [16]. The vision was that of a conversational web, where providers serve up VoiceXML; consumers interact via voice browsers; and VoiceXML can be hyper-linked. Compared to SwaggerBot, one draw-back of VoiceXML is that it requires providers to write new chat bots in that language. In contrast, SwaggerBot uses pre-written API specifications in Swagger to boot-strap a chat bot.

The idea of generating artifacts from Swagger is of course not new. To the contrary, Swagger is designed as a source language from which to generate assorted different artifacts [20]. What is new is generating a chat bot from it. SwaggerBot thus fits right into the Swagger ecosystem as another complementary code-generation target. The most closely related among other targets is the Swagger UI, which not only visually renders documentation, but also lets developers interact with a web API in a sandbox. However, that interaction does not use natural-language conversation, does not target end users, and is not intended for production use.

There is a substantial body of literature on NLIDB (natural language interfaces to databases) [1]. Just like SwaggerBot is generated from Swagger, NLIDB agents are generated from database schemas. Both enable a conversation to drive an action, consisting of a web API call for SwaggerBot or a database query for NLIDB. NLIDB work that employs multi-turn conversation has only emerged recently and is still an active research area [17].

IFTTT (if this then that) enables end users to call web APIs in a simple and convenient way [12]. Participants in the IFTTT ecosystem must wrap their end-points into either triggers or actions. Once those are created, the end user can combine them using recipes. In contrast, SwaggerBot users need not wrap their end-points in actions; instead, the actions are compiled into the chat bot by using an API specification as the source language. Also, unlike SwaggerBot, IFTTT does not offer a natural-language chat interface.

6.2 Self-Documenting and Auto-Curating Systems

Many programming languages come with an interactive REPL (read-eval-print loop). The first language with a REPL was probably LISP, and LISP inventor McCarthy credits Deutsch for implementing "the first interactive LISP on the PDP-1 computer in 1963" [18]. Chat bots are similar to REPLs in that both are a linear dialogue between human and computer, but chat bots use human language whereas REPLs use computer language. That said, both benefit greatly from being interactive and self-documenting.

ASK is a system allowing users to interact with a database via "ASK English" [24]. It is similar to a chat bot in that it offers a conversational interface. Like SwaggerBot, ASK supports some forms of curation; for instance, it lets power users introduce new synonyms. Unlike SwaggerBot, the curation does not use natural language; for instance, the following utterance introduces a synonym: 'definition:tub:old ship'.

Spreadsheets serve both end users and power users in the same tool [22], offering a fluid back-and-forth between simple tasks like entering data in tables and more advanced tasks such as creating formulas. Only about half the spreadsheets in the Enron corpus contain any formula and could have, at least in principle, been created by non-technical users [8]. SwaggerBot takes inspiration from how spreadsheets serve both end users and power users, but uses a natural-language conversation about resources and actions instead of tables of rows and columns.

Interactive learning is an approach for improving intent classifiers [25]. Given logs of unlabeled example utterances, it prompts the subject-matter expert to label the utterance most likely to improve overall classifier accuracy. Like SwaggerBot, this improves the chat bot, but unlike SwaggerBot, it does not do so from within the chat bot. Furthermore, in interactive learning, the initiative for curation lies with the computer, whereas in SwaggerBot, curation is driven by the initiative of the power user.

6.3 Programming with Natural Language

In a recent paper, we outlined a vision for using grammars to specify chat bots [9]. Our paper surveyed common flow patterns for chat bots and concluded that for many outcome-driven patterns, the outcome consists of information gathered from a user. This led to the idea that when a chat bot needs to gather non-trivial structured information from a user, they can elicit this information through a multi-turn dialogue driven by a grammar. In contrast, SwaggerBot does not propose a new programming language, but rather, retargets an existing specification format, namely, Swagger.

NLyze is an Excel feature that, given a natural-language sentence, synthesizes a formula for computing the value or format of a cell or column [7]. The synthesis operates at the granularity of one sentence, unlike the back-and-forth of a multi-turn dialogue. A spreadsheet user can get around that limitation by breaking down multi-step transforms into separate formulas synthesized from a different sentence. Unlike SwaggerBot, NLyze does not target web APIs and is not generated from a web API specification.

Similarly to NLyze, Kate et al. transform natural to formal languages [14]. They parse a natural-language sentence with a natural-language grammar, and then transform the resulting parse tree to the target formal language. As with NLyze, the focus is on one sentence; the technique does not target web APIs; and it does not use web API specifications.

A CNL (controlled natural language) is a constructed language that is based on a natural language [15]. For example, there are CNLs designed to specify data models or event-processing rules [2]. In a way, the direct call syntax in SwaggerBot fits the definition of a CNL, albeit a particularly simple one. In contrast to other CNLs, SwaggerBot aims at being completely self-documenting. The user can learn everything they need to know about SwaggerBot, including how to make direct calls, by interacting with it.

Overall, we could not find any prior work on either chat bots from web API specifications or chat bot curation using natural-language dialogue.

7 Future Work

While the SwaggerBot compiler succeeds at its main objective to generate a chat bot for calling a web API, we would like it to generate a better chat bot than it currently does. This section outlines several directions for improvements.

One problem is that many web APIs are not designed for end users but designed to be called from code written by developers. Our generated chat bots still expose low-level details such as HTTP methods, content negotiation, verbose JSON outputs, or the need to make multiple calls to answer one request. We have an expanding set of features to help with this issue: curation to hide spurious low-level details; data shaping to extract the most relevant information out of JSON outputs; and call chaining to simplify multiple calls. These features could ultimately add up to a marriage of Prolog-style backward chaining with Swagger.

Our current implementation offers two disjoint ways to call an API: step-by-step guidance vs. direct calls. This dichotomy reflects the coherence-flexibility dilemma [9] of chat bots: step-by-step guidance ensures coherent information but feels rigid and inflexible to the user, whereas a direct call is more concise but gives little guidance to get coherent parameters. We are currently working on unifying the two based on frames, a well-known chat bot concept for slot-filling dialogues [4] which is also supported by WCS [11]. While this changes our compiler substantially, the main idea, of using Swagger as the source language, remains.

Another problem is the lack of NLU training examples which are necessary for the intent classifier. In general, a classifier trained from more examples has higher accuracy. Since a Swagger file does not contain example utterances, we currently use the name of the action as the single training example of the intent. This is not enough to train a good classifier. While SwaggerBot allows the power user to add synonyms via in-bot curation, that does not help with the out-of-the-box experience. We are actively exploring various machine-learning techniques to obtain the best NLU using only the information commonly available in a Swagger file.

Our current prototype requires manual deployment and operations. While parts of SwaggerBot are hosted in WCS as a cloud service, other parts are not. That means the provider of a generated SwaggerBot must manually install it, manage

its lifecycle, and tackle operational concerns such as availability and scaling. We are working on making SwaggerBot fully cloud-hosted. Since the controller component can be viewed as a simple transducer from symbols to instructions, and since the actuator component amounts to calling cloud functions, we chose OpenWhisk for the cloud-hosted implementation [3]. Ultimately, we want providers to get from a Swagger file to a deployed chat bot in seconds.

At this point, SwaggerBot only supports scalar parameters for calls to web APIs. In the examples in this paper, all parameters are scalar values such as strings, dates, URLs, or API keys. However, for other web APIs, the parameters may also be composite objects and arrays. We are planning to address this issue with nested frames, following ideas for information-gathering dialogues that we outlined in another paper [9]. This will enable SwaggerBot to conduct longer and deeper dialogues before making a call.

Another problem is that many web APIs require authentication. This may be simply because the web API charges its users money for every call, or because calls to the web API may return sensitive information or have irrevocable effects. In the Visual Recognition case study in Figure 14, the user had to copy-and-paste an API key. We are planning to provide a more general sign-in solution that is secure but also natural and convenient. Solving this at the SwaggerBot level reduces the time-to-value for the chat bot provider.

Finally, SwaggerBot currently handles only one Swagger file at a time. Extending the prototype to handle a set of multiple web APIs, each with their own Swagger file, is relatively straightforward through the use of sub-dialogues. But it raises the issue of conflicts of action names between different APIs. A more challenging extension is to be able to search and load new Swagger files dynamically. In order to help the user to find the right API, we are considering to use API Harmony [26].

8 Conclusions

This paper describes a compiler from web API specifications (written in Swagger) to chat bots for calling those web APIs. That means that a company that has a Swagger specification for the services it offers its customers or employees can immediately obtain an initial natural-language chat bot for them. Doing so enables the company to jump right into improving the chat bot, which tends to be a continuous feedback-driven process. The generated bot is self-documenting, so that users who do not know how to use the bot or the web API can find out how to do that by interacting with the bot. Besides the compiler, this paper also presents in-dialogue curation for improving the chat bot. The curation addresses the problem that the API specification may not be high-quality, either in general or when it comes to generating chat bots from it. Currently, the curation features include adding new actions with default parameter values, as well as adding synonyms to make the natural-language understanding more robust.

This paper includes examples of generated bots for three web APIs that offer language translation, tone analysis, and visual recognition, respectively. The examples all work and show-case both the self-documenting facilities and the in-dialogue curation. Ultimately, our goal is to democratize the creation of chat bots; to make sophisticated APIs easy to call via chat bots; and to make chat bots delightful to use.

References

[1] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases – an introduction. *Natural Language Engineering* 1, 1 (1995), 29–81.

[2] Matthew Arnold, David Grove, Benjamin Herta, Michael Hind, Martin Hirzel, Arun Iyengar, Louis Mandel, V.A. Saraswat, Avraham Shinnar, Jérôme Siméon, Mikio Takeuchi, Olivier Tardieu, and Wei Zhang. 2016. META: Middleware for Events, Transactions, and Analytics. *IBM R&D* 60, 2–3 (2016), 15:1–15:10.

[3] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*.

[4] Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry Thompson, and Terry Winograd. 1977. GUS, a frame-driven dialog system. *Artificial Intelligence* 8, 2 (1977), 155–173.

[5] Herbert H. Clark and Susan E. Brennan. 1991. Grounding in communication. *Perspectives on socially shared cognition* 13 (1991), 127–149.

[6] Stefan Goessner. 2017. JsonPath. http://goessner.net/articles/JsonPath (Retrieved August 2017).

[7] Sumit Gulwani and Mark Marron. 2014. NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation. In *International Conference on Management of Data (SIGMOD)*. 803–814.

[8] Felienne Hermans and Emerson Murphy-Hill. 2015. Enron's Spreadsheets and Related Emails: A Dataset and Analysis. In *International Conference on Software Engineering (ICSE)*, Vol. 2. 7–16.

[9] Martin Hirzel, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Mandana Vaziri. 2017. I Can Parse You: Grammars for Dialogs. In *Summit oN Advances in Programming Languages (SNAPL)*. 6:1–6:15.

[10] IBM. 2015. Watson Language Translator Service. https://www.ibm.com/watson/developercloud/language-translator.html (Retrieved August 2017).

[11] IBM. 2016. Watson Conversation Service. https://www.ibm.com/watson/developercloud/conversation.html (Retrieved August 2017).

[12] IFTTT. 2011. If This Then That. https://ifttt.com/ (Retrieved August 2017).

[13] Ron Kaplan. 2013. Beyond the GUI: It's Time for a Conversational User Interface. *Wired* (2013). https://www.wired.com/2013/03/conversational-user-interface/

[14] Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to transform natural to formal languages. In *Conference on Artificial Intelligence (AAAI)*. 1062–1068.

[15] Tobias Kuhn. 2014. A Survey and Classification of Controlled Natural Languages. *Computational Linguistics* 40, 1 (2014), 121–170.

[16] Bruce Lucas. 2000. VoiceXML for Web-based Distributed Conversational Applications. *Communications of the ACM (CACM)* 43, 9 (2000),

53–57.

[17] Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. 2016. Making the Case for Query-by-Voice with EchoQuery. In *Demonstration Paper, International Conference on Management of Data (SIGMOD-Demo)*. 2129–2132.

[18] John McCarthy. 1981. History of LISP. In *History of Programming Languages (HOPL)*. 173–185.

[19] Michael F. McTear. 2002. Spoken dialogue technology: Enabling the conversational interface. *ACM Computing Surveys (CSUR)* 34, 1 (2002), 90–169.

[20] OpenAPI Initiative. 2014. OpenAPI Specification. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md (Retrieved August 2017).

[21] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. 2016. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In *International Conference on Web Engineering (ICWE)*. 21–39.

[22] Jorma Sajaniemi and Jari Pekkanen. 1988. An Empirical Analysis of Spreadsheet Calculation. *Software – Practice and Experience (SP&E)* 18, 6 (June 1988), 583–596.

[23] Slack. 2013. Messaging Platform. https://slack.com/ (Retrieved August 2017).

[24] Bozena H. Thompson and Frederick B. Thompson. 1983. Introducing ASK, A Simple Knowledgeable System. In *Conference on Applied Natural Language Processing (ANLP)*. 17–24.

[25] Jason D. Williams, Nobal B. Niraula, Pradeep Dasigi, Aparna Lakshmiratan, Carlos Garcia Jurado Suarez, Mouni Reddy, and Geoff Zweig. 2015. Rapidly Scaling Dialog Systems with Interactive Learning. In *International Workshop on Spoken Dialog Systems (IWSDS)*. 1–13.

[26] Erik Wittern, Vinod Muthusamy, Jim Laredo, Maja Vukovic, Aleksander Slominski, Shriram Rajagopalan, Hani Jamjoom, and Arjun Natarajan. 2016. API Harmony: Graph-based search and selection of APIs in the cloud. *IBM Journal of Research and Development* 60, 2-3 (2016), 12–1.