Just ASK:
Building an Architecture for Extensible Self-Service Spoken Language Understanding

Anjishnu Kumar
Amazon.com
anjikum@amazon.com

Arpit Gupta
Amazon.com
arpgup@amazon.com

Julian Chan
Amazon.com
julichan@amazon.com

Sam Tucker
Amazon.com
samtuck@amazon.com

Bjorn Hoffmeister
Amazon.com
bjornh@amazon.com

Markus Dreyer
Amazon.com
mddreyer@amazon.com

Stanislav Peshterliev
Amazon.com
stanislp@amazon.com

Ankur Gandhe
Amazon.com
aggandhe@amazon.com

Denis Filiminov
Amazon.com
denf@amazon.com

Ariya Rastrow
Amazon.com
arastrow@amazon.com

Christian Monson
Amazon.com
cmonson@amazon.com

Agnika Kumar
Amazon.com
agnika@amazon.com

Abstract

This paper presents the design of the machine learning architecture that underlies the Alexa Skills Kit (ASK) a large scale Spoken Language Understanding (SLU) Software Development Kit (SDK) that enables developers to extend the capabilities of Amazon's virtual assistant, Alexa. At Amazon, the infrastructure powers over 25,000 skills deployed through the ASK, as well as AWS's Amazon Lex SLU Service. The ASK emphasizes flexibility, predictability and a rapid iteration cycle for third party developers. It imposes inductive biases that allow it to learn robust SLU models from extremely small and sparse datasets and, in doing so, removes significant barriers to entry for software developers and dialogue systems researchers.

1 Introduction

Amazon's *Alexa* is a popular digital assistant that was not designed around a smart phone, but rather as a service for *ambient computing* [1] in the home. Due to the intuitiveness of the *Voice User Interface (VUI)*, the demand from software engineering teams for voice controlled features far outstripped the pace at which the language experts designing Alexa's SLU system could accommodate them. Removing this dependency posed several challenges, the first was the need to build a self service SLU architecture that could transform natural language queries into API calls, while ensuring that the architecture played well with Alexa's existing user experience. This paper describes the design and creation of this service.

The Alexa Skills Kit (ASK) was initially designed to empower internal Amazon developers to prototype new features independently of Alexa's core Automatic Speech Recognition (ASR) and Natural Language Understanding (NLU) systems, and then extended to give third-party developers the same

1

capabilities. In order to make Alexa a popular service for ambient computing, it was important to enable external developers to build sophisticated experiences for Alexa similar to the common operating systems of the smartphone era, Android and iOS.

An Alexa *skill* is an SLU subsystem that has two customized components corresponding to the SLU stack - ASR and NLU. When the user speaks to a particular skill, the Alexa service handles the conversion of speech into text, performs intent classification, and slot-filling according to a schema defined by the skill developer. Besides writing the skill definition, the developer is also responsible for creating a web service, which interacts with JSON requests sent by Alexa. Given the structured request from Alexa, the developer's web service can return text to be synthesized by Alexa's text-to-speech engine, an audio stream to be played back, with an optional graphical representation to be displayed in case the Alexa endpoint supports the visual modality. Figure 1 illustrates the interaction flow of a skill. A common practice for Alexa skill developers is to use a *serverless* endpoint such as AWS's Lambda product [2].
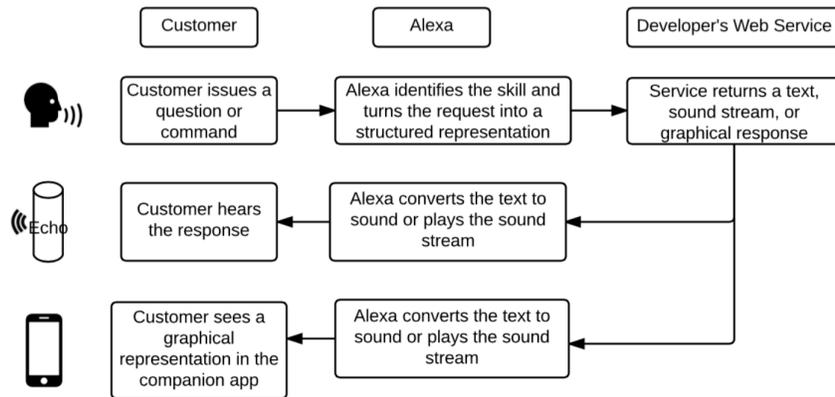

Figure 1: Interaction flow of an ASK skill

As of November 2017, there are over 25,000 Alexa Skills that have been built and deployed to customers via ASK. The SLU architecture described in this paper is also at the foundation of the Amazon Lex SLU service [3] and applications such as *Amazon Connect*, a hybrid human/AI customer service product which uses Amazon Lex to power virtual agents. In the year following the launch of the ASK, competing SDKs following similar design principles have also been launched by Google and Microsoft, namely *Actions on Google* and the *Cortana Skills Kit*.

ASK allows researchers to experiment on conversation and dialogue systems without the additional overhead of maintaining their own ASR and NLU systems [4, 5, 6].

2 Related Work

Prior toolkits allowed users to build ASR and NLU models individually. For example, Kaldi [7] and HTK [8] are popular toolkits for speech recognition. Stanford's Core NLP [9] offers a suite of NLP libraries. These toolkits allow a lot of flexibility to the developer. For example, CoreNLP gives complete independence in selecting the libraries to use in the language understanding task. Similarly, Kaldi offers pre-built recipes which can be used as is or modified according to need of developer. However, this flexibility also poses challenges for the developer who is not well versed with Speech and NLP literature and ML methodology. These toolkits did not provide speech and language understanding capabilities in an open and self-service manner. At the other end of the spectrum, standards such as VoiceXML have existed since the early 2000s. They support clearly defined interfaces for software developers, but have supported only rigid command structures for end users. The need to develop a set of portable tools to address domain specificity with small datasets has long been considered a bottleneck for the large scale deployment of spoken language technology [10].

In parallel to our work, SpeakToIt (now DialogFlow) launched Api.ai and Microsoft launched LUIS.ai [11], both of which provided self-service SDKs to third party developers to voice enable

their applications in isolation. In our work, we attempt to close the gap by offering ASR and NLU models that work together out of the box with limited training samples and do not require expertise in either field, as well as the capability to rapidly deploy these systems directly to a large and growing audience by extending a widely available virtual assistant.

3 Design Considerations

As an SDK for a voice assistant, speech recognition and language understanding technologies are the key value addition that Alexa can offer compared to existing frameworks. Since such a framework had never been built for a large user base before, we had to look beyond the software design tenets that were used to build largely monolithic high performance SLU systems in the past. Successful modern software development frameworks such as Amazon Web Services, the Python programming language, and Linux, have large developer communities and could provide relevant design tenets. We describe some of these tenets below.

The system needs to offer modular and flexible building blocks. To enable maximum flexibility, we chose to allow developers to specify commands they would want to support rather than limiting them to a set of commands or intents designed by Amazon. We chose to implement decoupled, modular subsystems could be updated independently. We believe that decoupled systems are commonly studied in software engineering, but remain an underexplored area in existing machine learning research. In recent advances in deep learning research, there is a trend towards training complex end-to-end (E2E) models directly from data, [12, 13, 14, 15]. These models offer improved performance and are sometimes easier to maintain by reducing the number of components as compared to decomposed or cascaded architectures. However, joint modeling and end-to-end modeling can also introduce dependencies between constituent systems, and make it harder to deploy improvements independently [16].

A third party must not be allowed to degrade the first party user experience. The danger of customer experience degradation was mitigated by sandboxing skills and allowing them to be used only once *enabled*, either explicitly or implicitly. This design choice resulted in rapid adoption by developers since they were no longer in direct competition with each other or with the first party system, but made skill access more difficult for a customer by impairing the naturalness of the interaction.

The discoverable surface area for voice is limited, so the framework must prevent cybersquatting. To prevent early developers from taking up valuable voice real-estate, a decision was made to allow overlapping skill names, and allowing developers to choose any name of their choosing as long as it does not reflect a brand protected by trademark or copyright which is not owned by them. Namespace disambiguation would be performed by the user at enablement time. Other frameworks chose to *elect winners*, by having a more rigorous vetting process and by awarding valuable real estate to chosen vendors, which results in a more consistent user experience early on, but may limit innovation in the long run.

The framework should allow fast iteration to support a rapid VUI development cycle. Since a skill developer did not have any usage data while developing a skill, enabling a rapid iteration cycle was important to enable them to quickly address issues. This requirement meant that SLU models needed to be trained in minutes, not hours, and created a tradeoff with the demand for highly accurate models.

The framework must remain relevant in the face of rapid advances in machine learning. Since the system used machine learning extensively, the state of the art models implementing its underlying ASR and NLU technologies were changing rapidly. The API contract thus had to be independent of the our initial model choice. We chose a lightweight shallow semantic parsing formalism comprised just of *intents* and *slots*. This formalism is analogous to an API call, the *intent* representing the function and *slots* representing parameters to that function. The choice of this simple interface for communicating between the language understanding system and software developers meant that the underlying systems could be updated to use arbitrarily complex models and representations, which could then be compiled back into the API call representation. Since model architectures become outdated quickly, it was necessary to build up a repertoire of techniques that treat model choices as black boxes, in order to enable the rapid deployment of new model architectures without having cascading effects on other functionality.

Since a developer could only provide a limited number of samples, any model trained on these utterances was unlikely to be of high quality. It thus became important to build components to leverage transfer learning and low resource learning techniques while remaining primarily model agnostic.

This can be done in several ways. Firstly, *data-based transfer*, that is, the curation of data resources that can be used generically. Secondly, *representation-based* transfer, the creation of a label space that encapsulates prior knowledge about how different entities relate to each other. Thirdly, *feature-based* transfer such as in [17, 18] or aided by unsupervised representation learning techniques [19, 20]. Investments in these transfer learning strategies are likely to remain relevant even when the state of the art model architecture changes. Once gains from from generic techniques start to stagnate, it is imperative to invest in *model-based* transfer learning strategies that exploit the specific characteristics of the machine-learning models being used [21, 22, 23, 24]. In order to develop and deploy these strategies in a fast changing research landscape, it was critical to develop an infrastructure designed for the rapid deployment of research, we discuss these tradeoffs in subsequent sections.

4 Customer Experience

Customers primarily interact with skills primarily in 3 different ways. *Modal* interactions are explicit invocations where the customer first invokes a skill or service e.g. *"Open Twitter"* then issues a command *"Search for trending tweets"*. *One-shot* invocation targets a skill or service and issues a command simulatenously, for example, *"What is trending on twitter"*.

Both the modal and one-shot invocation modalities are supported by a combination of deterministic systems and statistical shallow parsing models. The one-shot modality is only available for the set of skills a customer has *enabled* previously, in order to prevent unintended collisions. The explicit launch functionality, for example *"Open the Twitter skill"* attempts to disambiguate skills and do an implicit enablement if there is no ambiguity.

A recently launched third modality is *skill suggestion*. This allows for customers to be suggested a skill and routed to it when they issue a command that cannot be serviced by Alexa's first party systems but can likely be handled by a skill. This can be done by performing statistical matching between user utterances and relevant skills by using techniques derived from information retrieval or using semantic matching performed by deep neural networks [25].

Once a skill is invoked, a skill context is established, effectively sandboxing the interaction and preventing a customer from accessing any of Alexa's native capabilities until the skill exits, either gracefully after task completion or because the customer ceases to interact with the device.

5 BlueFlow: A Flexible Model Building Infrastructure

Our modeling infrastructure is developed using *BlueFlow*, a Python framework intended to accelerate the pace at which ML projects could be brought from research to production by using a shared codebase. We developed an internal design paradigm called *CLI-to-Server*. The paradigm ensures that every API call defined by BlueFlow is also executable locally via a programmatically generated *Command Line Interface (CLI)* in order to ensure that it is possible to reproduce the production stack during experimentation. Using autogenerated CLIs for the APIs adds a extra layer of abstraction to the codebase but helps mitigate a common machine learning anti-pattern in which production services and research tooling are implemented separately and go out of sync after a period of time.

BlueFlow is designed to enforce a clean separation between operational concerns and system logic using the constructs of *artifacts, components, activities and recipes*. In a manner similar to some open source deep learning libraries [26], BlueFlow uses python code constructs as a *declarative language* to define a *symbolic computational graph* for data flow management. This computational graph is a *directed acyclic graph (DAG)* and can be serialized, then optimized and executed by a compatible *executor* locally or in a distributed manner. Refer to Appendix A for details on the BlueFlow architecture and its syntax.

BlueFlow runs model building in production, but can also run it ad-hoc on a research cluster for conducting experiments. Having both research and production services use the same system allows us to quickly deploy new modeling techniques to production without spending significant time on productizing throwaway research code. The use of Python allows for rapid experimentation and a concise codebase, with an option to optimize bottlenecks in by using C++ bindings. However Python's type system increases reliance on unit test coverage. Along with the BlueFlow task execution framework, we extensively leverage technologies developed by AWS. All static artifacts needed for a model build are stored in Amazon's Simple Storage Service (S3) [27] and all artifacts that are loaded at runtime are stored in Amazon's highly performant key value store DynamoDB [28].

6 Skill Definition and Query Representation

To create a new skill for Alexa, a developer begins by defining an *interaction model*, which includes defining an intent schema, slot types, providing sample utterances corresponding to a simple grammar, and an invocation phrase.

A wide variety of utilities exist to help developers define interaction model for an Alexa Skill, including but not limited to Skill Builder, a rich web interface with hints and completions and a testing console; AWS's Lex UI, which has an 'export-as-skill' option; and programmatic skill management via the command-line using the Alexa Skill Management API - which also enables third party developers to start building skill development tooling.

ASK gives developers full freedom when defining a voice experience; however, a skill developer cannot realistically be expected to provide large and representative data samples that reflect real world usage. While this can be addressed in part by using transfer learning techniques, we also directly expose concepts from Alexa's internal domains in the form of *builtin intents* and *builtin slot types* for the developer to pick and choose from.
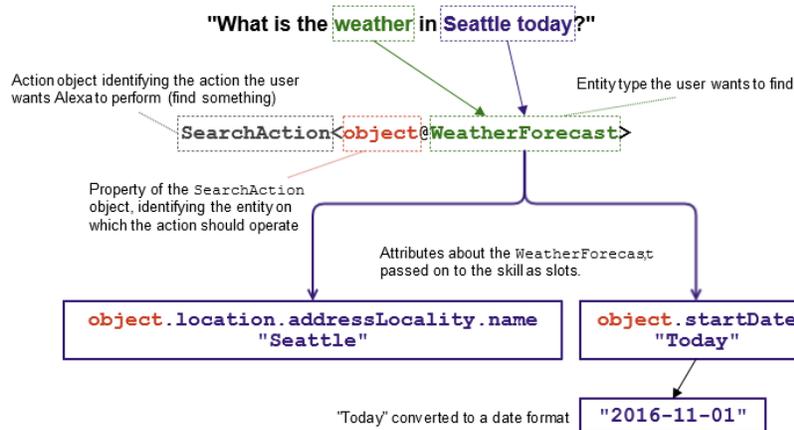

Figure 2: An utterance represented in AlexaMRL

6.1 The Alexa Meaning Representation Language

In this section we briefly describe the *Alexa Meaning Representation Language (AlexaMRL)* [29], which is necessary to understand builtin intents. The AlexaMRL is a decomposable semantic parsing formalism for defining spoken language commands, which allows for concepts to be efficiently reused across different SLU domains.

AlexaMRL is composed of *Actions*, *Entities*, and *Properties*. Actions are a higher order abstraction than the *intents* that are typically used in natural language tasks, and can be viewed as a templating system for intents. An Entity from the Alexa Ontology is analogous to a slot from an ASK developer's perspective. Properties are completely transparent to a skill developer, but under the hood, they tie Actions and Entities together by making certain Entity Types compatible with certain Ac-

5

tions. Entities possess Properties similar to object attributes in Object Oriented Programming. For example, *LocalBusiness* is an Entity Type. It has Properties such as *business hours, address, phone number* etc. Actions require Properties, i.e. they require for Entities to possess certain Properties in order to be compatible, a *CallAction* cannot be completed unless it is operating on an Entity with a *Callable* Property.

An intent represents an action the user wants to take. This could be searching or information, or playing a media object. For example, a *FindPlaceIntent* intent would internally route to an API call for a location search engine. The reason an Action is more abstract than an NLU intent is because an intent needs to be aware of the surface forms (string representations) of the Entities it operates on, but an Action does not. Instead, an Action operates on abstract *interfaces* that fulfill certain criteria. For example, An *AddAction* requires an object Entity which can be added and a target Entity which can be added to. This is specified using the required Properties of the Action. AddAction has a required property *targetCollection* which identifies the type of list to add to and an *object* Property that marks the type of the object to be added.

6.2 Exposing Query Representation to Skills

Thus developers have an option to *compile* builtin intents by filling an Action template with compatible Entity Types, which allows reuse data from Alexa's ontology via AlexaMRL. The skills that use compiled intents use shared deterministic models and efficiently reuse internal data resources for their stochastic models. We will discuss statistical modeling in greater detail in Section 7.

Developers that choose to design their own intents with Alexa's builtin slot types, instead of Action templates, do not benefit from AlexaMRL's ability to automatically derive semantic roles, and cannot fully reuse shared deterministic models. In this case, only the shared data for an Entity Type can be reused, and the semantic roles of the Entities in an utterance must be derived from developer provided samples.

7 Statistical Modelling

In order for Alexa to turn user requests into structured representations, we need to build models based on the developer's definition for both the ASR and NLU systems. The first step to this process is to efficiently represent the grammars provided by the developer.

7.1 Weighted Finite State Transducers

Weighted Finite-State Transducers (wFST) provide an easy way to represent data under a weighted grammar. A path through the FST encodes an input string (or sequence) into output string. We generate FSTs for both ASR and NLU. ASR uses a skill specific FST to decode utterances into sentences defined by the developer, whereas NLU uses an FST to recognize intent and slot values. Both can be generated from the developer-provided sample utterances, intents, and slots. Most of the FST-generation code is shared between the ASR and NLU. Fig. 3 shows an FST that recognizes the *GetHoroscope* intent along with its *Date* slot from an utterance.

This representation is powerful because we can impose arbitrary distributional priors on the grammar. We infer a distribution over intents and slots from the sample utterances. As the data in sample utterances is often imbalanced, we follow the *principle of maximum entropy* to impose uniform priors over intents and then slots on the grammar. This is a configurable switch that can be turned off when the wFST is generated from the usage pattern of a skill.

The weighted FST representation can be used directly by common sequence models designed to work with text such as CRFs [30] and LSTMs [31] by sampling utterances according to the distributional priors and feeding them into the respective models as training data via a data recombination technique similar to [32]. One can also train directly on the lattice itself [33].

7.2 Automatic Speech Recognition

The ASR system uses weighted finite-state transducers (WFST) to represent the Hidden Markov Model (HMM) transducer (H), phone context-dependency transducer (C), pronunciation lexicon
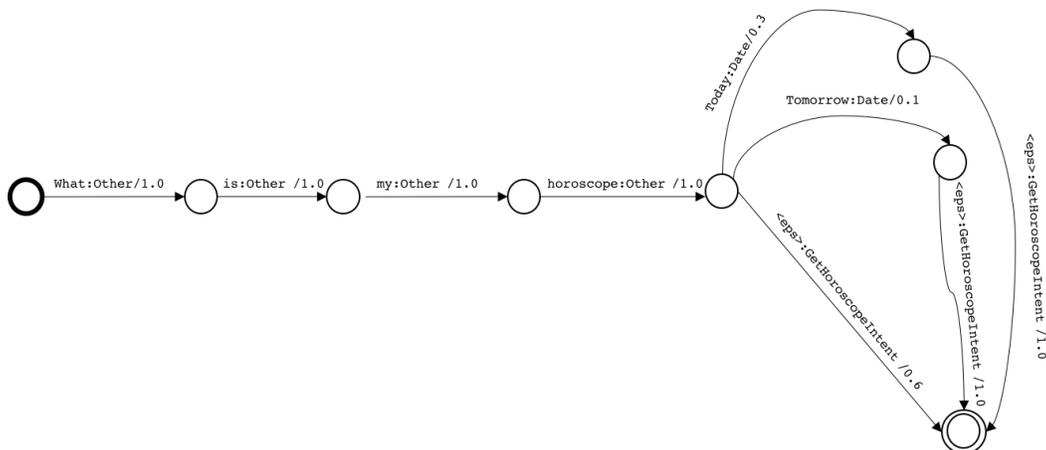
Figure 3: wFST states are represented by circles and marked with their unique number. The input label *i*, the output label *o*, and weight *w* of a transition are marked on the arcs.

(L), and word-level grammar (G). These FSTs are composed to form an end-to-end recognition transducer. We refer the reader to Mohri et al. [34] for details on these transducers and their role in ASR. The goal of the ASR recipe is to generate a word-level grammar (G) that guides the system to recognize utterances directed to the skill [35]. The (G) decoder for ASK is a hybrid decoder that uses a skill-specific grammar as well as a main n-gram based Statistical Language Model (SLM) that shares data with other skills.

Continuous improvements are key to any machine learning product. In addition to regularly ingesting human-transcribed skill-directed utterances to the training corpus of the main SLM, we also ingest ASR recognition results directly as a form of semi-supervised training [36]. In order to reduce the risk of reinforcing recognition error, we employ weighted language model training [37] where each semi-supervised sentence is weighted according to its confidence value given by the ASR system. These semi-supervised learning techniques ameliorate ASR errors due to distributional misallocations during the initial build phase.

7.3 Natural Language Understanding

In this section we describe the Language Understanding component of the Alexa Skills Kit. Given an utterance, our NLU system aims to recognize its *intent* and relevant parameters as *slots*, a task known as shallow semantic parsing. The NLU system is divided into deterministic and stochastic subsystems.

The deterministic NLU subsystem uses FST data representation to compile sample utterances provided by developer into a recognizer model. This recognizer guarantees coverage on all utterances specified by a skill developer while designing her grammar, allowing for predictable experience for both the developer and the customer.

The stochastic system uses BlueFlow to lend flexibility to the choice of model. We have built individual algorithmic components which implement a linear chain CRF [38], and a maximum entropy classifier [39], an updated pipeline using LSTMs for each task [40] and joint neural models for entity and intent prediction [41, 42]. This allowed us to experiment with different configurations as long as it conforms to the API of receiving a sentence and returning a semantic frame. In fig 4 we show a traditional NLU system which performs entity recognition followed by intent classification and finally slot resolution.

7.3.1 Knowledge Injection

The performance of statistical models can be greatly improved by using features derived from Knowledge-Base (KB) lookups, but it is infeasible to perform expensive queries at runtime. By using efficient data structures to encode sketches of relevant portions of a knowledge base, we can
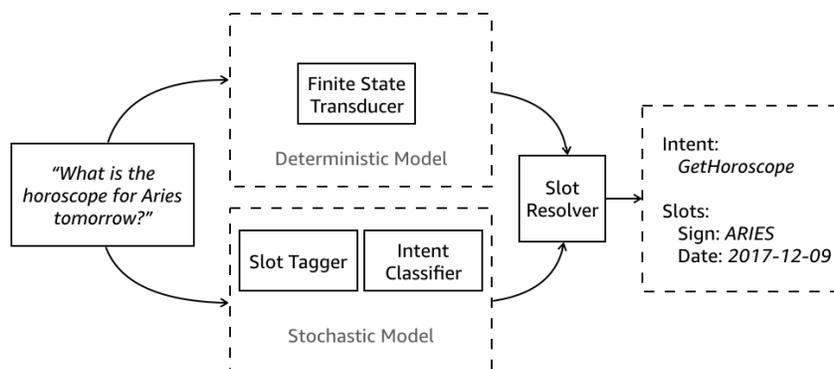
7

Figure 4: Overview of the hybrid NLU pipeline.

make them available to statistical models as feature extractors. This is done by encoding onto-logically derived word-clusters as bloom filters during training time [43, 44], including those from custom types defined by developers. These features can be extremely useful for training models in a low data regime since they encourage feature generalization to classes or categories, with a small probability of false positives. This can massively increase the effective vocabulary of a small model without blowing up its feature-space, and has added benefit of enabling class coverage improvements to deploy asynchronously without needing to retrain statistical models.

Training with KB features can lead to language feature under-training, the models learn to rely too heavily on KB features to distinguish entities. This can be addressed by introducing a *knowledge dropout* regularization parameter to prevent the model from overfitting [45].

7.3.2 Model Optimization for Deployment

Skill models are stored in DynamoDB [28] and loaded at runtime, and network calls account for the bulk of added latency. We use standard techniques like feature hashing [46], weight quantization, [47] and sparsity constraints via modified elastic net regularization [48] to ensure that our models are small enough to be stored cheaply and can be transferred over network calls quickly with no statistically significant loss in accuracy.

8 Dialogue Subroutines

ASK supports the specification of dialogue subroutines for common tasks, a unified dialogue model automates simple procedural dialogue capabilities, such as slot elicitation (e.g., *User: Get me a cab – Alexa: Where do you want to go?*) confirmation questions (e.g., *User: Portland – Alexa: You want to go to Portland, Oregon, right?*) and other easily specified mechanisms. These procedural subroutines can either be invoked by developers to issue a *dialogue act directive* during the course of interaction, or defined as part of the interaction model. Although this system fulfills many usecases, we recognize the limitations of having a simple dialogue system.

9 Conclusions and Future Work

We described the ASK SLU service which allows third party developers to expand the capability of the Alexa Voice Service. Our system abstracts away the intricacies of ASR and NLU, and provides developers with an interface based on structured request data. The ASK service now hosts over 25,000 consumer facing SLU subsystems that improve over time, with hundreds of new Skills being added every week. This implies that we have made significant progress towards our goal of building a fully extensible SLU architecture. However challenges still remain in building a seamless user experience, and to enable the creation of useful and engaging agents. In order to seed research in this area, Amazon has introduced the Alexa Prize, a 2.5 million dollar university competition to advance conversational AI through voice. Alexa prize bots were launched to customers as a skill using ASK, and early results have shown promise [6].

References

[1] Fariba Sadri, "Ambient intelligence: A survey," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, pp. 36, 2011.

[2] Amazon Web Services, "AWS Serverless Multi-Tier Architectures," *AWS Whitepapers*, 2017.

[3] Amazon Web Services, "Amazon Lex Developer Guide," *AWS Whitepapers*, 2017.

[4] Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Cetintemel, and Tim Kraska, "Making the case for query-by-voice with echoquery," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 2129–2132.

[5] Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Çetintemel, "Voice-based data exploration: Chatting with your database," *Proceedings of the 2017 workshop on Search-Oriented Conversational AI*, 2017.

[6] Iulian V Serban, Chinnadhurai Sankar, Mathieu Germain, Saizheng Zhang, Zhouhan Lin, Sandeep Subramanian, Taesup Kim, Michael Pieper, Sarath Chandar, Nan Rosemary Ke, et al., "A deep reinforcement learning chatbot," *arXiv preprint arXiv:1709.02349*, 2017.

[7] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al., "The kaldi speech recognition toolkit," in *IEEE 2011 workshop on automatic speech recognition and understanding*. IEEE Signal Processing Society, 2011, number EPFL-CONF-192584.

[8] Steve J Young, *The HTK hidden Markov model toolkit: Design and philosophy*, University of Cambridge, Department of Engineering, 1993.

[9] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky, "The stanford corenlp natural language processing toolkit.," 2014.

[10] Victor W Zue and James R Glass, "Conversational interfaces: Advances and challenges," *Proceedings of the IEEE*, vol. 88, no. 8, pp. 1166–1180, 2000.

[11] Jason D Williams and Eslam Kamal, "Fast and easy language understanding for dialog systems with microsoft language understanding intelligent service (LUIS)," 2015.

[12] Jason D. Williams, Kavosh Asadi, and Geoffrey Zweig, "Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning," in *ACL*, 2017.

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[14] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[15] Iulian V Serban, Alessandro Sordoni, Yoshua Bengio, Aaron Courville, and Joelle Pineau, "Building end-to-end dialogue systems using generative hierarchical neural network models," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[16] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young, "Machine learning: The high interest credit card of technical debt," in *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[17] Hal Daumé III, "Frustratingly easy domain adaptation," *ACL 2007*, p. 256, 2007.

[18] Young-Bum Kim, Karl Stratos, Ruhi Sarikaya, and Minwoo Jeong, "New transfer learning techniques for disparate label sets," in *ACL*, 2015.

[19] Jeffrey Pennington, Richard Socher, and Christopher D Manning, "Glove: Global vectors for word representation," in *In EMNLP*. Citeseer, 2014.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[21] Ankur Bapna, Gokhan Tur, Dilek Hakkani-Tur, and Larry Heck, "Towards zero shot frame semantic parsing for domain scaling," in *Interspeech 2017*, 2017.

[22] Young-Bum Kim, Karl Stratos, and Ruhi Sarikaya, "Frustratingly easy neural domain adaptation," in *COLING*, 2016.

[23] Jan Trmal, Jan Zelinka, and Luděk Müller, "Adaptation of a feedforward artificial neural network using a linear transform," in *International Conference on Text, Speech and Dialogue*. Springer, 2010, pp. 423–430.

[24] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[25] Anjishnu Kumar, Pavankumar Reddy Muddireddy, Markus Dreyer, and Bjorn Hoffmeister, "Zero-shot learning across heterogeneous overlapping domains," in *Proc. Interspeech 2017*, 2017, pp. 2914–2918.

[26] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[27] Amazon Web Services, "AWS Storage Services Overview," *AWS Whitepapers*, 2015.

[28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*. 2007, vol. 41, pp. 205–220, ACM.

[29] Xing Fan, Emilio Monti, Lambert Mathias, and Markus Dreyer, "Transfer learning for neural semantic parsing," in *Proceedings of the 2nd Workshop on Representation Learning for NLP*, Vancouver, Canada, August 2017, pp. 48–56, Association for Computational Linguistics.

[30] John Lafferty, Andrew McCallum, and Fernando CN Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the 18th International Conference on Machine Learning*, 2001, vol. 951, pp. 282–289.

[31] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer, "Neural architectures for named entity recognition," in *Proceedings of NAACL-HLT*, 2016, pp. 260–270.

[32] Robin Jia and Percy Liang, "Data recombination for neural semantic parsing," *Proceedings of ACL*, 2016.

[33] Faisal Ladhak, Ankur Gandhe, Markus Dreyer, Lambert Mathias, Ariya Rastrow, and Björn Hoffmeister, "LatticeRnn: Recurrent neural networks over lattices," *Interspeech 2016*, pp. 695–699, 2016.

[34] Mehryar Mohri, Fernando Pereira, and Michael Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech & Language*, vol. 16, no. 1, pp. 69–88, 2002.

[35] Petar Aleksic, Mohammadreza Ghodsi, Assaf Michaely, Cyril Allauzen, Keith Hall, Brian Roark, David Rybach, and Pedro Moreno, "Bringing contextual information to google speech recognition," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[36] Thomas Drugman, Janne Pylkkönen, and Reinhard Kneser, "Active and semi-supervised learning in asr: Benefits on the acoustic and language models.," in *INTERSPEECH*, 2016, pp. 2318–2322.

[37] Hui Zhang and David Chiang, "Kneser-ney smoothing on expected counts," in *ACL*, 2014.

[38] J. Lafferty, A. McCallum, and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proc. 18th International Conf. on Machine Learning*, 2001, pp. 282–289.

[39] A. L Berger, V. J.D Pietra, and S. A.D Pietra, "A maximum entropy approach to natural language processing," *Computational Linguistics*, vol. 22, no. 1, pp. 71, 1996.

[40] Kaisheng Yao, Baolin Peng, Yu Zhang, Dong Yu, Geoffrey Zweig, and Yangyang Shi, "Spoken language understanding using long short-term memory neural networks," in *Spoken Language Technology Workshop (SLT), 2014 IEEE*. IEEE, 2014, pp. 189–194.

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

[42] Dilek Hakkani T ur, G okhan T ur, Asli Celikyilmaz, Yun Nung Chen, Jianfeng Gao, Li Deng, and Ye Yi Wang, "Multi domain joint semantic frame parsing using bi directional rnn lstm.," in *INTERSPEECH*, 2016, p. 715 719.

[43] Xiaohu Liu, Ruhi Sarikaya, Liang Zhao, Yong Ni, and Yi-Cheng Pan, "Personalized natural language understanding.," 2016.

[44] Jan A. Botha, Emily Pitler, Ji Ma, Anton Bakalov, Alex Salcianu, David Weiss, Ryan Mcdonald, and Slav Petrov, "Natural language processing with small feed-forward networks," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark, 2017, p. 28692875, Supplementary material: http://aclweb.org/anthology/attachments/D/D17/D17-1308.Attachment.zip.

[45] Eunsuk Yang, Young-Bum Kim, Ruhi Sarikaya, and Yu-Seop Kim, "Drop-out conditional random fields for twitter with huge mined gazetteer," in *Proceedings of NAACL-HLT*, 2016, pp. 282–288.

[46] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 2009, pp. 1113–1120.

[47] Armand Joulin, Edouard Grave, and Piotr Bojanowski Tomas Mikolov, "Bag of tricks for efficient text classification," *EACL 2017*, p. 427, 2017.

[48] Hui Zou and Trevor Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

Appendix A   BlueFlow Architecture

Architecture. The fundamental units of BlueFlow are *components*. Component are wrapped in *activities*. Activities are chained together into *recipes*. Recipes are executed by an *executor*.

Components. A component is a function that solves a defined task. This can be simple (convert from format A to format B) or complex (train a classifier). In either case, a component is the smallest unit that other scientists would expect to reuse. Components are normal Python code and can be used independently of BlueFlow.

Activities. Activities are lightweight wrappers around components, responsible for fetching data from *artifacts* and then running a component. Artifacts are lazy file like objects that support a uniform interface for reading and writing data from heterogeneous data sources such as local files, DynamoDB [28] and S3 [27]. Activities are expressed as regular Python, but they specify input and output via an annotation syntax.

Below we show a simple activity to train a Maximum Entropy Classifier.

```
@Activity(inputs=('features_artifact'),
              outputs=('model_artifact'))
def train_classifier(features_artifact, model_artifact):
  model_file = components.train_maxent(features_artifact.fetch())
  model_artifact.put(model_file)
```

Recipes. A recipe is a chained together set of activities. Recipe code is no longer normal Python, but rather uses Python language constructs to define a symbolic graph of data flowing through Activities as Artifacts, and can be serialized as a Directed Acyclic Graph (DAG). Then, we can execute the independent paths of the DAG in parallel either on the same machine or on a cluster.

Below we show a recipe for training a Maximum Entropy Classifier.

```
@Recipe
def build_ic_model(data_file, executor):
    data_artifact = executor.new_artifact(data_file)
    features_artifact = executor.new_artifact()
    model_artifact = DynamoDBArtifact('models/classifier')
    extract_features(data_artifact, features_artifact)
    train_classifier(features_artifact, model_artifact)
```

Executors. An executor abstracts the details of a recipe execution. It is responsible for vending appropriate artifacts and for taking the appropriate actions when executing specific activities. Thus, a *local* executor vends local files as intermediate artifacts and runs recipe's activities on the same machine, whereas a *remote* executor vends S3 files as intermediate artifacts and runs recipe's activities on a cluster.

The executor is also responsible for performing recipe optimizations, via improved scheduling, IO or smart object caching. The local executor can be used for low latency model builds such as the ASK model builds. For offline experiments with higher throughput requirements we use the remote executor to run jobs on a cluster. Switching from a naive local executor to a multithreaded local executor results in model build speed increases of about 50 percent.

Other than recipe execution, executors add important features for production code such as logging, metrics, and retries, that would otherwise result in significant boilerplate. Every Recipe is automatically converted into an equivalent command line tool for local execution and reproduction.

Appendix B   Intent Schema and Custom Slot Types

The *intent schema* specifies the intents supported by a skill and the expected slots for an intent.

An *intent* is the intended action for an utterance. Developers may define their own intents or use the Amazon provided built-in intents. An intent may accept a *slot* as an argument to represent an entity along with it's semantic role in the sentence. The data type of the slot is captured by the *slot type*, which corresponds to entity types from the ontology.

Developers may use the Amazon provided built-in slot types or define their own *custom slot types*. Each custom slot type requires a list of representative values.

Listing 1: Sample Intent Schema for a Horoscope skill from the ASK Documentation

```
1   {
2     "intents": [
3       {
4         "intent": "GetHoroscope",
5         "slots": [
6           {
7             "name": "Sign",
8             "type": "ZODIAC_SIGNS"
9           },
10          {
11            "name": "Date",
12            "type": "AMAZON.DATE"
13          }
14        ]
15      }
16    ]
17  }
```

Sample utterances form the training data used for building spoken language understanding models. Each sample is labelled with an intent followed by an utterance corresponding to the intent.

Listing 2: Sample Utterances for a Horoscope skill

```
1   GetHoroscope what is the horoscope for {Sign}
2   GetHoroscope what will the horoscope for {Sign} be on {Date}
3   GetHoroscope get me my horoscope
4   GetHoroscope {Sign}
```