Building a Chatbot with Serverless Computing

Mengting Yan*
Department of Computer Science
University of Illinois, Urbana-Champaign
myan9@illinois.edu

Paul Castro, Perry Cheng, Vatche Ishakian
IBM Watson Research Center
Cambridge MA
{castrop, perry, vishaki}@us.ibm.com

ABSTRACT

Chatbots are emerging as the newest platform used by millions of consumers worldwide due in part to the commoditization of natural language services, which provide provide developers with many building blocks to create chatbots inexpensively. However, it is still difficult to build and deploy chatbots. Developers need to handle the coordination of the cognitive services to build the chatbot interface, integrate the chatbot with external services, and worry about extensibility, scalability, and maintenance. In this work, we present the architecture and prototype of a chatbot using a serverless platform, where developers compose stateless functions together to perform useful actions. We describe our serverless architecture based on function sequences, and how we used these functions to coordinate the cognitive microservices in the Watson Developer Cloud to allow the chatbot to interact with external services. The serverless model improves the extensibility of our chatbot, which currently supports 6 abilities: location based weather reports, jokes, date, reminders, and a simple music tutor.

CCS Concepts

• Software and its engineering →Software System Structures • Human-Centered Computing→Interaction Paradigms.

Keywords

Serverless, FaaS, bots, cloud computing.

1. INTRODUCTION

Cloud providers are offering a growing number of cognitive services, such as machine learning, translation, and analytics, as building blocks for developing AI applications. One such application is a "chatbot," which uses a conversational interface to interact with the user. Chatbots have been around since the 1960s [1], but recently there has been a rapid increase in the number of chatbots due in part to a wide market adoption of mobile and "smart" devices. Chatbots are now embedded in popular messaging programs [2] and also appear as stand-alone services like Amazon Alexa, Microsoft's Cortana, and Apple Siri.

The IBM Watson Developer Cloud (WDC) offers cloud-based "cognitive" services of composable AI building blocks that developers can use to build applications like chatbots [3]. WDC provides speech-to-text (STT) and text-to-speech (TTS), tone analysis, language classification (NLC) and translation, as well as conversation modeling [4]. WDC provides SDKs for multiple platforms that developers can use to quickly put these services into their applications. Instead of offering one monolithic voice service, WDC allows the developer to compose these services as building blocks. Developers can mix-and-match voice services from different vendors according to their needs. For example, a mobile app developer can combine the STT service offered by the iOS platform with Watson NLC by writing custom "glue" code that coordinates the two services.

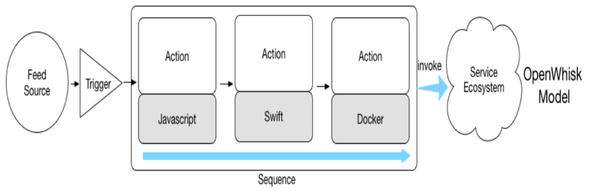

Figure 1. OpenWhisk Programming Model

Despite the commoditization of cognitive services, it is still difficult to build a chabot. Developers have to handle the coordination of the cognitive services to build the chatbot interface, integrate the chatbot with external services, and worry about extensibility, scalability, maintenance, and resource costs to run the chatbot.

Serverless [1] has recently emerged as an alternative way of creating backend applications. Serverless does not require dedicated infrastructure. Major cloud vendors such as Amazon Web Services, Google, Microsoft, and IBM have created versions of serverless. Serverless lets the developer deploy "functions" (serverless is also referred to as Functions-as-a-Service) into a shared platform that is maintained by the vendor. The functions are typically standard code snippets in popular languages that execute in a stateless fashion. It is the vendor's responsibility to keep the infrastructure running smoothly and scale resources to satisfy function executions due to changes in user demand. Serverless life-cycle costs are typically lower than costs for dedicated infrastructure as serverless vendors do not charge for idle time.

In the serverless programming model, developers deploy code snippets as "functions" into the cloud. Functions are stateless and each invocation is independent of previous runs. Functions can be invoked directly or triggered by events. Functions are not meant to be long running; hence, this encourages a software model where an application is broken up into several functions containing a small amount of logic. Non-trivial applications like .chatbots will utilize many functions chained together.

* Work done while author was an intern at IBM Watson Research Center in Cambridge, MA

MOTA '16, December 12-16 2016, Trento, Italy
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4669-6/16/12…$15.00
DOI: http://dx.doi.org/10.1145/3007203.3007217

Serverless functions can be used coordinate the microservices used by the chatbot. While serverless functions are not specifically designed for mashups, the programming model provides data flow and event-based abstractions usable for mashup architectures. In addtion, serverless hides most operational concerns from the developer and provides a scalable rutime for creating chatbot solutions.

In this work, we investigate the use of the serverless programming model as a mashup framework to modularize domain-specific processing for the chatbot, as well as a way to create a mechanism for extending the chatbot's capabilities. We are specifically interested in creating serverless chatbots that interact with a set of diverse commodity services publically available on the Internet like a weather service and stateful reminders. We present an architecture as well as a protoype implementation of our chatbot that uses WDC services as our AI building blocks and the IBM OpenWhisk serverless computing service [5]. Our architecture supports incremental extension of a chatbot's basic set of capabilities -- possibly written using different programming languages by other organizations in the company -- and is generic enough to support different user interaction medium (e.g. audio, text). We plan to release the code as open source.

Our approach is similar to the AWS Alexa Skill SDK [9], which uses the AWS Lambda serverless platform coupled with the Amazon Voice Service to create chatbot apps. However, the Alexa SDK is tied to a single vendor and uses a more monolithic voice service. Our approach is more amenable for introducing custom voice models and intent processing. Although we used OpenWhisk as our serverless platform, we believe our work can be generalized across other serverless programming models that use stateless actions, e.g. AWS Lambda [1].

The remainder of this paper is organized as follows. In Section 2 we provide an overview of the OpenWhisk serverless programming model and how it pertains to our chatbot. In Section 3 we provide the blueprints for a serverless chatbot, protoype implementation, and discuss initial performance results. In Section 4 we describe open research challenges. Finally, we present related work and conclusions in Sections 5 and 6, respectively.

2. OPENWHISK

OpenWhisk (c.f. Figure 1) is a serverless platform from IBM that is freely available in open source [5]. Developers can write functions using a variety of languages including Javascript, Swift, Python, Java, or an embedded binary in a Docker container. Interaction with OpenWhisk is through a REST API.

Figure 1 shows the basic programming model of Openwhisk. The OpenWhisk model is based on three primitives:

Action: a stateless function that executes arbitrary code. Actions can be invoked asynchronously, in which the invoker --caller-- does not expect a response, or synchronously where the invoker expects a response as a result of the action execution.

Trigger: a class of events that come from a variety of sources.

Rule: a mapping from a trigger to an action.

In Figure 1, three actions written in different languages are chained together into a *sequence* where the output of one action is piped to the input of another action. Actions can be grouped together into *packages* to create a service eco-system that can be invoked from other actions. For example, the sequence in the

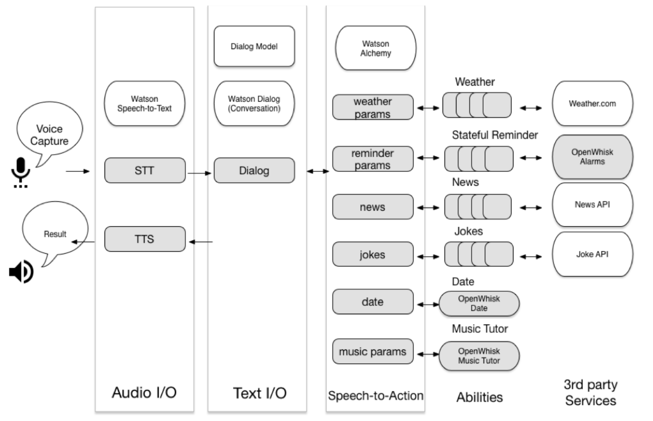

Figure 2. Chatbot Architecture

figure could invoke a package containing actions that interact with a NoSQL database.

Actions can be invoked directly a REST API, or executed based on a trigger. In Figure 1, there is an event feed which is connected to a trigger. A trigger fires when the event occurs. Triggers accept parameters which are passed down to actions mapped to the trigger using rules.

Below is an example HelloWorld action written in Javascript:

```
function main(args) {
   Console.log("Hello world")
}
```

The developer can install this function into OpenWhisk and it will be executed as a cloud-native component. OpenWhisk will automatically manage resources needed to support this action and trigger invocations. It will also dynamically scale in response to workload changes.

3. CHATBOT

In this section, we present the basic architecture and prototype implementation of our chatbot. As noted earlier, serverless computing is an ideal platform for building extensible chatbots due to its differentiating characteristics. First, it provides a way to create cloud-native "glue" code for composable AI building blocks. It frees the developer from the burden of managing the scalability due to fluctuation in user demand. It allows for a normalized scripting environment that can be packaged and distributed. Finally, it also provides a critical missing piece: a means for composable AI services to interact with other, non-conversational services to perform interesting tasks for users.

3.1 Architecture

Figure 2 shows the architecture of our serverless chatbot. The basic architecture consists of four levels of serverless actions and a microservice endpoint.

The first level of processing is Audio I/O, which converts user audio input to text, and vice versa. This part of our architecture is optional and only used if text is not available for input.

The second level of processing is Text I/O. This level is responsible for routing the user input to the correct set of serverless actions for domain-specific processing. In this stage, one can utilize any publically available conversation services to help route the request, as well as "box" in the user to provide feedback if the user input cannot be processed.

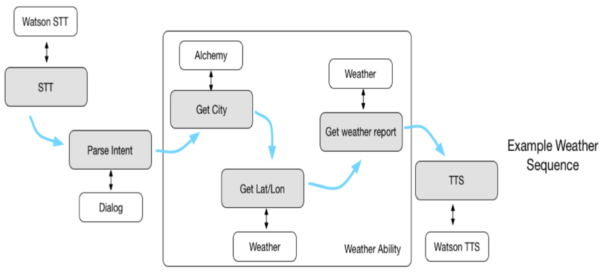

Figure 3. Weather flow example

The third level of processing is for domain specific chatbot "abilities." Each ability has to do at least two things. First, it must convert the raw user input into a parameterized function call. We created actions to interact with the Watson Alchemy service to help us extract the necessary parameters. Clearly, parameter extraction is done differently for each ability. Secondly, it takes the parameters and invokes one or more actions to handle the imput. Handler actions may call out to external services to complete their processing. Depending on the ability, our chatbot returned results as text and/or audio output.

To deploy the chatbot, the developer needs accounts for each of the Watson services we used. OpenWhisk allows the binding of parameter values at deploy time, so the developer only has to set the value of authentication tokens once. Since our architecture consists only of serverless actions, it is possible to interact with our chatbot through any of the three layers : 1) audio for any spoken interaction; 2) text for textual input; 3) direct invocation of an ability using the serverless framework API.

3.2 Prototype

We implemented six abilities that span different application paradigms: (1) News: an action that calls IBM Watson news service to get the top three news article titles for the day, (2) Jokes: An action that REST calls a joke service, (3) Date: an action that returns the current date (4) Weather, (5) Music Tutor: a customized ability that allows the user to ask questions of the type "How many flats are in B flat major?", and (6) an Alarm Service.

In Figure 3, we show the weather sequence used in our chatbot as an example. The shaded blocks are OpenWhisk actions, while the white blocks represent endpoints used by the actions. In the figure, we draw a box to separate common routing code from domain-specific ability code. Extending the chatbot with a new ability means following the pattern shown in Figure 3. We describe details of this pattern in the next section.

To initiate the conversation flow, we use a 3rd party audio application to capture speech, encode the recording into a Base64 encoded string, and uses the *STT* action to convert the audio data to text. The *STT* action uses the Watson Speech-to-Text service. The *Parse Intent* action is responsible for interacting with the Watson Dialog Service (WDS) with the goal of determining which ability needs to be invoked from the audio. We note that we also used the newer Watson Conversation service but will focus on the Dialog service in this description.

WDS provides a comprehensive, robust platform for managing conversations. Developers script models of conversations as they would happen in the real world in an XML formatted file, and upload it to the dialog application by using either the watson-developer-cloud Node.js module or the web based WSD tool.

The dialog XML file encodes a dialog model that consists of several sections that outline default behaviors such as greeting and failover when the conversation is not understandable. The main section of the XML file consists of the input tags which allows a developer to specify templates of user questions. In our example, the WDS parses the intent and maps it to a grammar that correspond to an ability. For, example, the phrase "What is the weather in *?" is in the grammar for the weather ability. WDS provides several regex wildcards to help simplify the process and allow the specification of generic question patterns. For example, the asterisk replaces a word or set of words in the position it occupies. The dollar sign ($) specifies that the phrase after it must appear exactly as it is written, but any words can precede it or follow it.

In WDS, an *output* tag contains the corresponding answers to a user's specific question. However, as noted in the architecture section, the result of *Parse Intent* is not intended for human consumption. Instead, we need the output to invoke an OpenWhisk ability. To enable this, *Parse Intent* expected certain keywords in the response from Dialog that indicate which ability is required.

The basic structure of the Weather ability consists of an OpenWhisk sequence of three actions. The *Get City* action REST calls IBM Watson's Entity extraction service to detect the city in question. The *Get Lat/Lon* action uses the weather service to identify the longitude and latitude points for the detected city. The *Get Weather Report* action uses the weather service to get a weather report from the longitude and latitude points specified. The response from the weather service is a human readable weather report. Once the appropriate weather report is fetched we use *TTS* action to generate a Base64 encoded speech string, the string is stored as an audio file locally and played to the user.

Our focus was on creating an extensible chatbot architecture using serverless and this we did not optimize for performance. For a real deployment, the latency to respond should be be on the order of 1-2 seconds, while our current, un-optimized implementation could be more than double that. While we believe we can reduce this latency just by improving our serverless implementation, our chatbot relies on many outside endpoints, each with their own service guarantees and performance latencies. Our weather example relies on outside services for authentication, speech processing, conversation modeling, etc. which are outside our control.

4. RESEARCH CHALLENGES

While the idea of composing serverless functions is quite appealing, it lacks certain aspects that would hinder its long term wide adoption and smoother composition of services. We broadly classify these aspects into four major categories: Programming & Debugging, Performance, Monitoring, and Security.

4.1 Programing & Debugging

Developing individual functions in serverless computing is relatively straightforward, but composing a series of functions together to create a certain flow is a difficult task to accomplish, particularly if different composition flows depend on one another. There is currently a lack of tooling for allowing developers to more easily debug why their actions failed. Logs from cloud-based function invocations need to be reported back to the developer and provide detailed stack traces. Also, it may be useful to develop actions in emulated runtimes first, which could provide access to debug tools like breakpoints.

The serverless composition will likely be dependent on many endpoints – when an error occurs, we require better ways to report exactly what happened; for example, for client-side programming we can get a stack trace when something fails. The equivalent of a stack trace for serverless microservice compositions is currently not available.

In addition to tooling, it may be beneficial to understand what types of programming abstractions are necessary to model serverless applications. Today, serverless platforms like AWS Lambda allow functions to communicate with each other over a shared messaging channel, while OpenWhisk provides a *sequences* abstraction that creates a single action by chaining other actions together. As this technology evolves, it is useful to look at work on workflows and ECA styles of programming to see what is useful for microservice composition using serverless.

4.2 Performance

Serverless can makes developing services easier, but providing QoS guarantees is hard. The serverless platform itself must have some guarantees of scalability and availability; but this is of little use if the application relies on outside services for authentication, data persistence, etc. which are outside the control of the serverless platform.

To provide certain QoS guarantees for scalability, availability, latency, etc., the serverless platform needs to communicate the required QoS to the underlying, outside components. Enforcement must be done across functions and APIs, and the careful measurement of such services, through self- reporting or by an evaluation system, is essential.

4.3 Monitoring

Serverless abstracts away the notion of a server from the developer, and pushes operational concerns to the providers of the serverless runtime. Despite this, it will be important for developers to monitor the deployment of their solutions since the execution of their functions is directly mapped to a cost model that charges for execution time (typically) and not for idle time. The monitoring infrastructure will be the responsibility of the serverless vendor and have to scale to potentially billions of installed functions.

4.4 Security

Understanding the attack surface of a serverless composition is an on-going issue. Clearly, the serverless runtime may inadvertently have security flaws that can be exploited by an attacker. Many serverless platforms rely on a container service like Docker to create application siloes; it is important to ensure that functions cannot break out of this silo and gain access to unauthorized resources.

As is true for any composition, it is important to understand potential sources of information leakage and ensure that protected information remains safe. Developers may create compositions that accidentally exposes information that should be private to one component of the composition.

Token management is also an issue. Developers must deal with: 1) tokens needed to access microservice APIs 2) tokens needed to access 3rd party services on behalf of end users 3) tokens users need to access the chatbot. We should investigate ways to reduce the development burden when dealing a collection of authentication and authorization mechanisms in a mashup.

5. Related Work

There are many examples of chatbots with work spanning 60 years [6]. We explore a completely serverless chatbot that is not tied to a specific vendor. Our chatbot is extensible through mashups using OpenWhisk to composition logic. The first serverless platform was AWS Lambda [12]. OpenWhisk and OpenLambda are open source serverless platforms [11]. Serverless is comparable to mashup frameworks that allows developers to deploy application logic to compose microservices [7][8][10], but is less formal than web service coordination by providing a generic (and sometimes polyglot) programming model. Our chatbot is closest to the Alexa Skill Set SDK [9], except we are not tied to a specific vendor. Our architecture could be implemented using other serverless platforms, though refactoring the code is non-trivial given the differences in programming and deployment models.

6. CONCLUSION

In this work, we presented a generic architecture for a chatbot framework built on top of Serverless computing platform. Our architecture is extensible, inherently scalable, and supports different user interactions mediums.

Our on-going research work is pursued along the following dimensions. First, we plan to fully automate extending our prototype with new abilities, which require automatic redeployment of XML dialog models, and perform additional performance measurement to detect the bottlenecks and identify consolidation points.

7. REFERENCES

[1] Weizenbaum, Joseph. "ELIZA—a computer program for the study of natural language communication between man and machine." *Communications of the ACM* 9.1 (1966): 36-45.

[2] Facebook Messenger Platform, https://messengerplatform.fb.com

[3] AskTanmay, https://github.com/tanmayb123/AskTanmay.

[4] Watson Developer Cloud, https://www.ibm.com/watson/developercloud/

[5] OpenWhisk, https://github.com/openwhisk/openwhisk

[6] Shawar, Bayan Abu, and Eric Atwell. "Chatbots: are they really useful?." *LDV Forum*. Vol. 22. No. 1. 2007.

[7] Yu, Jin, et al. "Understanding mashup development." *IEEE Internet computing* 12.5 (2008): 44-52.

[8] Tai, Stefan, Rania Khalaf, and Thomas Mikalsen. "Composition of coordinated web services." *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer Berlin Heidelberg, 2004.

[9] Alexa Skills Set SDK, https://developer.amazon.com/appsandservices/solutions/alexa/alexa-skills-kit.

[10] Baldini, I., Castro, P., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., ... & Suter, P. (2016, May). Cloud-native, event-based programming for mobile applications. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems* (pp. 287-288). ACM.

[11] Hendrickson, Scott, et al. "Serverless Computation with OpenLambda." *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. 2016.

[12] AWS Lambda, https://aws.amazon.com/lambda/

[13] Obie Fernandez, Serverless: Patterns of Modern Application Design Using Microservices (Amazon Web Services Edition), in preparation, https://leanpub.com/serverless