University Chatbot: A Journey into Cloud-Native Development

Harrison Kahl, Brian Montecinos-Velazquez, and Dominic Spampinato

***Abstract*—The University Chatbot is a cloud-based project executed over CloudLab infrastructure. The project displays the utility of cloud-based development with concepts such as containerization and Continuous Integration/Continuous Delivery. The front-end is a simple chat interface allowing users to ask about campus-related events. The backend involves a deep-learning chatbot application that can process user input and query a database for information webscraped from university-related sites. Development spanned two versions of a framework for cloud-native development. Reflections on how each framework affected the development process are proviced.**

I. Introduction

The University Chatbot [1] will provide users an interface to make specific queries about campus-related events through a natural language chat app. The motivation behind the project is to highlight the utility of cloud-native development for complex, full-stack applications. Development relies on cloud computing concepts such as containerization, container orchestration, and Continuous Integration/Continuous Delivery (CI/CD). These concepts are implemented through open-source tools such as Docker, Kubernetes, Jenkins, and Helm.

The Chatbot is comprised of four components: with the front-end being a Web User Interface(WebUI) and the back-end being a Chatbot, Webscraper, and Database. Development includes the use of languages and open source tools, such as Python, JavaScript, MySQL, and Rasa. The project is executed over infrastructure provided by the academic cloud, CloudLab. Given the technical complexities of the application's work flow, a cloud-native development framework was followed [3]. Development of the project spanned two stages of this framework, providing insight into how different models affect the development process. The initial framework allowed for each application component to be containerized in Docker and orchestrated across a Kubernetes cluster through CI/CD pipelines in Jenkins. However, this framework had several drawbacks, including the lack of consideration for security practices and the need manual configuration of services. To address these concerns, the framework was improved with the inclusion of features, such as ingress control, access control, and automated configuration through Helm.

The remainder of this paper is organized as follows. Section II presents a review of cloud security topics and the framework on which the Chatbot was built. Section III describes the state of the project prior to the updated framework. Section IV discusses how the updated framework affected the project's development process. Section VI provides a reflection on the project.

Kahl, Montecinos-Velazquez, and Spampinato were with the Department of Computer Science, West Chester University of Pennsylvania, Pennsylvania, 19383 USA e-mail: {hk869465,bm935325,ds946528}@wcupa.edu.

Manuscript received December 16/ 2022.

II. Literature Survey

The University Chatbot is based in a Kubernetes framework for learning cloud native Development [3]. This framework is designed to be deployed on CloudLab, an academic cloud. Funded by the National Science Foundation in 2014, CloudLab was built to provide researchers with a robust cloud-based environment for next generation computing research [4]. With CloudLab, users can program computing infrastructures, startup commands, and topology using Python. This *profile* is used to generate a resource description document, from which the actual cloud-based experiment is deployed. Development spanned over two versions of this framework: an initial manual configuration stage and a security enhanced Helm-based deployment stage.

A. Manual Configuration

The initial framework encompassed a simple setup to launch a Cloudlab experiment with installation scripts for the desired infrastructure. While the deployment of a Kubernetes cluster and Jenkins automation server could be automated through Bash scripts, configuring the infrastructure to meet the needs of a project required a manual step at the start of each CloudLab experiment. Since Cloudlab experiment are typically only active for a 16-hour period, reconfiguring the Jenknins server each time became a significant hurdle for development. Further, this framework neglected to consider cloud security best practices.

B. Security Enhanced Helm-based Deployment

The current framework introduces several features to address security concerns and leverages Helm [2] to automate the configuration of infrastructure. With Helm, configuration parameters can be stored in YAML files to be reused for each CloudLab experiment. Further, experiment-dependent parameters can be dynamically substituted in configuration files through Bash scripts. This allows for the deployment of the Jenkins server to be automated with the start of each experiment, requiring no manual configuration. Beyond optimizing the deployment of infrastructure, several security features were introduced, such as a private Docker registry, ingress control, and access control.

A private Docker registry is deployed within the Kubernetes cluster, keeping custom Docker images secure within the cluster. A certificate manager is deployed alongside the registry, which allows for the use of self-signed certificates to authorize access from cluster resources. An ingress controller allows for external traffic to be routed to cluster services. Further, TLS certificates can be applied to enable secure HTTPS traffic. The Kubernetes cluster is additionally secured through the implentation of RBAC authorization. Access to cluster resources is strictly reserved for roles defined by Kubernetes. The Jenkins server is granted access to cluster resources through a service account to enable the use of CI/CD pipelines.

C. AWS Access Control

To enhance the team's understanding of RBAC, a survey of industry-standard implementations was carried out. Amazon Web Services (AWS) uses a RBAC model that associates permissions to Identity and Access Management (IAM) policies, which can then be connected to an identity or resource. AWS has 6 different types of policies to leverage; the 2 most common are identity-based policies and resource-based policies, which are used in combinations with AWS session policies. [5]

1) Identity-based Policies: Identity-based policies are composed of permission files associated with an identity. An identity represents a user. These policies control the actions that a user is authorized to execute, which can be limited to specific resources and conditions. [5]

2) Resource-based Policies: Where as identity-based policies connect permissions to the users, resource-based policies connect the permissions to the cloud resources. Resource-based policies gives AWS consumers the ability to enable cross-account access to specific resources [5]. Furthermore, when combined with an identity-based policy, the policies act similar to multi-factor authentication. For example, consider a real-world scenario where an AWS Lambda is trying to access an S3 bucket. The AWS Lambda must have an identity-based policy associated that gives it permission to access an S3 bucket. If configured under the concept of ”least privileged access”, then the lambda would have to be appended to the S3 bucket's resource-based policy before gaining access.

3) Session Policies: Session policies are parameterized policies specific to an ephemeral session. Permissions are attached to a session based upon the intersection of session policies and identity-based policies. Intersecting these two policies prevents users from gaining unauthorized access. Additional session permissions can also come a resource-based policy.

There are three main ways session policies can be utilized to create effective security:

1) A resource-based policy specifies a user as a principal owner. Then, the additional permissions granted within the resource policy are appended to the identity based policy prior to the creation of the session. The configuration described is depicted in Figure 1.

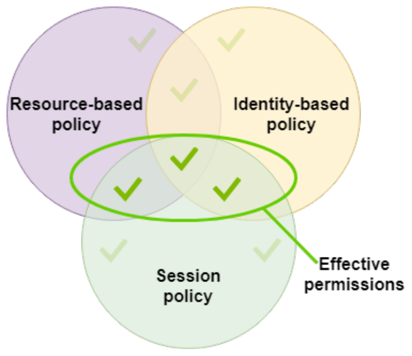

Fig. 1. Resource based policy with user as principal [5]

2) Instead of specifying a user as principal, a resource could designate a session as the principal. This strategy would result in a session with all of the resource-based permissions combined with the intersection session policy permissions and the identity-based policy permissions, as seen in Figure 2. [5]

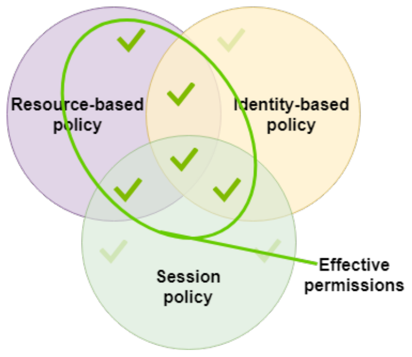

Fig. 2. Resource based policy with session as principal [5]

3) Lastly, a permission boundary can limit the number of permissions for a user that can be applied when creating the session. This limits the session permissions to the intersection of the session policy, identity-based policy and the permissions boundary. It is important to note that the permissions granted by a resource policy with the session as a principal are not restrained by the permission boundary. [5]

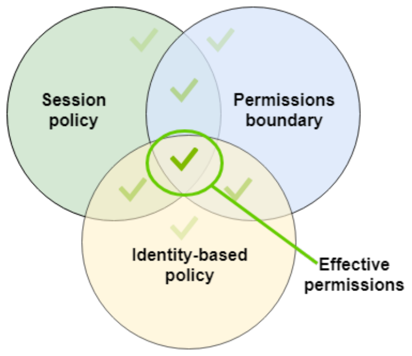

Fig. 3. permission boundary of a session policy [5]

III. Pre-Authorization

The Chatbot is comprised of four components: the front-end being a Web User Interface (WebUI) and the back-end being a

Chatbot, Webscraper, and Database. As outlined in Figure 4, a typical process flow involves a user making a query in natural language through the WebUI. This query is forwarded to the Chatbot's Natural Language Understanding(NLU) server for processing. Once the Chatbot has processed the intent of the user input, it queries the Database through an actions server and returns a reply to the WebUI. The Webscraper executes independently to populate the Database with documents collected from university-related websites.

After the first phase of development with the initial framework, the project was incomplete. The backend was functional as a proof of concept for the project but the frontend WebUI had not been developed to a functional state. Each component was successfully containerized and configured for automated deployment into the Kubernetes cluster through a CI/CD pipeline in Jenkins. However, the pipeline implementation lacked automated integration and required a manual build step. The following sections provide further technical details about each component.

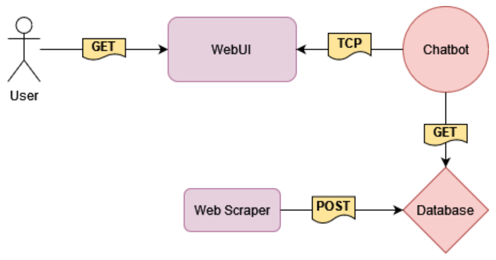

Fig. 4. Chatbot flow diagram

Front-end

The WebUI is meant to be a web-based application mimicking standard instant messaging applications available on the market. Through this chat interface, users will be able to make specific, university-related queries and receive accurate responses from the Chatbot. Development is based in JavaScript frameworks, such as React.js and Socket.IO. The WebUI was deployed as an Nginx service within a container made accessible externally through a NodePort protocol. However, the development of a Socket.IO application to interface with the Chatbot was unsuccessful.

Back-end

The Chatbot is a standard deep learning application developed within the Rasa framework for chatbots. Rasa allows for the creation of sophisticated chatbots by providing tools to expedite the development process. Within Rasa, chatbots are composed of two servers: an NLU server for processing user input and an actions server for carrying out request from the NLU server. The NLU server is configured to interface with the WebUI. The actions server can query the Database and return documents to the NLU server. Each server is containerized and deployed within a multi-container pod. The Chatbot was able to handle a base case of user input and make simple queries to the Database. Development beyond this base case proved to be difficult and slow.

The WebScraper is a Python script that collects data from university-related sites and populates the Database. It makes use of dependencies for web scraping and MySQL database management, including Beautiful Soup and SqlAlchemy. The script is containerized and deployed as a Kubernetes job.

The Database stores university-related data to be accessed by the Chatbot. The deployment pulls directly from the official MySQL Docker image and requires a persistent volume for storage within the Kubernetes cluster.

Discussion

This initial phase of development was considerably slow due to the hurdle of configuring the infrastructure required for cloud-native development. A particular shortcoming was the manual setup of the Jenkins automation server required at the start of each CloudLab experiment. This tedious process dissuaded the development team from taking advantage of the benefits of CI/CD pipelines earlier in the development process, as evidenced by the lack of an automated build step in the pipeline implementation. As such, the Chatbot was not able to reach a fully functional state at the end of this initial phase. Another shortcoming in the original project was the lack of consideration for cloud security practices. Security deficiencies included hard-coded configuration files and lack of ingress control or access control for cluster resources.

IV. Post-Authorization

Within the new framework, the development process of the Chatbot [1] has improved significantly. The new framework's use of Helm to automate the deployment the Jenkins server made the concept of CI/CD more attainable. The project now benefits from full CI/CD pipeline integration, making the continued development of the Chatbot more efficient. Component files are now dynamically configured through parameter substitution by CI/CD pipelines rather than hard-coded values. This makes the project more secure and portable across CloudLab infrastructure without manual configuration. Particularly, development of the Chatbot can be expedited without the need to manually retrain and test the NLU models throughout the process.

The inclusion of infrastructure to address security deficiencies has enriched the development process to be in line with security standards. Rather than through an unrestricted NodePort protocol, external traffic to the WebUI is now TLS-secured through the deployment of an ingress controller. The inclusion of a private Docker Registry removes the need to maintain public Docker repositories through DockerHub accounts. This ensures critical data within custom-built images remains secure within the cluster.

With the benefit of this new framework, several improvements to the project can be made. The initial vision for the WebUI as a chat application is under development and soon to be realized. The Chatbot is being improved to handle increasingly complex user input. For example, can apply entity extraction to make more specific queries to the Database. The Webscraper is being adapted into a service that can collect new data based on parameters defined by the user through the Chatbot. Access to the Database can be secured through parameter substitution and secret managers.

V. Discussion

Exploring the rich field of cloud computing can be a daunting task. With the aid of a solid framework for cloud computing education [3], working on a cloud-native project can be an effective way to learn about cloud concepts. Development of the University Chatbot has been an enriching experience for the team. Even at an unfinished state, working on the project has enabled the team to explore concepts ranging from the low-level architecture of Docker containers to highly-abstracted Kubernetes clusters. The team has gained a well rounded understanding of core cloud computing concepts and a strong foundation to continue learning. Further, the consideration for security aspects elevates this framework by enabling students to comprehend real-world best practices. Key topics covered by this framework, namely containerization, CI/CD, and Access Control, have yet to be widely integrated into academic curriculum. As such, the integration of such frameworks into university programs can be invaluable towards bridging the skill gap of students to match the needs of the industry.

VI. Conclusion

Development of the Chatbot is ongoing. Each component can be improved upon significantly to provide greater utility to users. As development continues, the team can continue to highlight drawbacks of the framework for cloud-native development and provide feedback for ways to improve it. Some potential ideas include:

- Standardizing a secrets manager to store critical project values
- Improving the sophistication of Jenkins CI/CD pipelines through the use of plugins for build artifacts and parameter substitution
- Introducing options to explore serverless resources

References

[1] Chatbot. https://github.com/bmv0161/csc603-project.git/, 2022.

[2] Andrew Block and Austin Dewey. *Managing Kubernetes Resources Using Helm.* Packt Publishing, 2022.

[3] Harrison Kahl, Brian Montecinos-Velazquez, Linh-B. Ngo, Rackeem Reid, Austin Reppert, Danielle Rivas, Dominic Spampinato, and Hudson Zhong. A kubernetes framework for learning cloud native development. *; login:: the Consortium for Computing Sciences in Colleges*, 2022.

[4] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[5] Amazon Web Services. Policies and permissions in iam. https://docs.aws.amazon.com/IAM/latest/UserGuide/access$_policies.html$.