|  |
| --- |
| COMP4906 HONOURS THESIS: SKYDOT |
| Final Report |

|  |
| --- |
| Lexi Brown - 100956208 |

Table of Contents

[Summary 3](#_Toc512293065)

[1 Introduction 6](#_Toc512293066)

[1.1 Problem 6](#_Toc512293067)

[1.2 Motivation 7](#_Toc512293068)

[1.3 Project Goals 7](#_Toc512293069)

[1.4 Proposal Objectives 8](#_Toc512293070)

[1.5 Possible Features 8](#_Toc512293071)

[1.6 Technology and Equipment Requirements 9](#_Toc512293072)

[2 Background 10](#_Toc512293073)

[2.1 Architecture 10](#_Toc512293074)

[2.1.1 Microservice 12](#_Toc512293075)

[2.1.2 Layered 15](#_Toc512293076)

[2.2 Technologies 16](#_Toc512293077)

[2.2.1 Azure (AKS) 16](#_Toc512293078)

[2.2.2 Kubernetes 17](#_Toc512293079)

[2.2.3 Docker 19](#_Toc512293080)

[2.3 Alternatives 21](#_Toc512293081)

[3 Solution 24](#_Toc512293082)

[3.1 Client 24](#_Toc512293083)

[3.2 Skydot 24](#_Toc512293084)

[3.2.1 Micro-Apps 25](#_Toc512293085)

[3.2.2 Micro-Services 28](#_Toc512293086)

[3.2.3 Host Gateway 30](#_Toc512293087)

[3.3 Backend 32](#_Toc512293088)

[3.4 DevOps 34](#_Toc512293089)

[3.4.1 Dashboard 34](#_Toc512293090)

[3.4.2 Logging 35](#_Toc512293091)

[3.4.3 Monitoring 36](#_Toc512293092)

[4 Results/Validation 37](#_Toc512293093)

[Project Goals 37](#_Toc512293094)

[5 Conclusion 38](#_Toc512293095)

[5.1 Future Work 39](#_Toc512293096)

[5.1.1 Multi-Cloud 39](#_Toc512293097)

[5.1.2 Improvements 40](#_Toc512293098)

[Additional Figures 46](#_Toc512293099)

[Glossary 46](#_Toc512293100)

[References 48](#_Toc512293101)

# Summary

The rise of microservices has been a remarkable advancement in application development and deployment. With microservices, an application is developed or refactored into individual services that have the capability to communicate with one another through a common template, for instance APIs. Each service is self-contained, manages its own data storage and can be updated independently of other services. Moving to a microservice-based approach makes application development faster and easier to manage; requiring fewer people to develop and maintain the system. A system designed as a collection of microservices is easier to run on multiple servers and, in the case of this project, multiple cloud environments with load balancing. This allows for better handling of demand spikes and of slower increases in demand over time while reducing downtime caused by hardware or software problems.

Microservices are a critical part of the paradigm shift occurring in the way applications are being built. Agile development techniques, the transition from on-premise to cloud, DevOps culture, continuous integration and continuous deployment (CICD), and containerization of applications all work alongside microservices to revolutionize application development and delivery.

In this report, I cover information relevant to implementing microservices through Skydot and the work that went into finding the best way to model Skydot if it was to be implemented in industry. The sections within this report are:

**Introduction** – A simple and clear introduction to Skydot.

1. **Problem** – An outline of the problems Skydot will be addressing.
2. **Motivation** – What has inspired and motivated me to build Skydot.
3. **Project Goals** – A list of all the goals Skydot aims to accomplish.
4. **Proposal Objectives** – Objectives set by this proposal to be completed in the implementation of Skydot.
5. **Possible Features** – A list of possible features that may be completed in the implementation of Skydot.
6. **Technology and Equipment Requirements** – A list of technology and equipment required to complete the project.

**Background** – A summary of all the background research completed in preparation of building Skydot.

1. **Architecture** – A simple outline to the architecture used by Skydot. This includes microservice and layered architecture styles.
2. **Technologies** – A simple introduction to technologies utilized by Skydot.

**Solution** – A summary of how the whole project was built from top to bottom.

1. **Client** – An overview on the web and mobile client solutions.
2. **Skydot** – An in-depth look at how Skydot was built and the capabilities of the solution.
3. **Backend** – An outline of how the backend was put together to represent older and newer technologies.
4. **DevOps** – An overview of all completed features that pertain to development operations.

**Results/Validation** – An in-depth report of all results and findings involved in analyzing Skydot. And an analysis of key components that will show off Skydots abilities.

**Conclusion** – A summary of all the aspects of the project.

1. **Future Work** – An analysis of how Skydot could be improved in the future.

Every section pertains to the construction of Skydot and meeting the goals outlined by the project. I hope you find every section worthwhile and inspirational to your own utilization of Skydot and cloud-based microservice architecture.

# 1 Introduction

Skydot is a cloud-based architecture that allows companies to minimize the cost of cloud and on-premise services and provides an environment where developers can utilize any language that best suits their needs and/or skills. This is achieved by utilizing a universal REST API and auto scaling services and apps. Skydot also fronts backend services, databases and other resources via a REST translation layer. This way backend services won’t have to change to adopt new technologies and new services won’t have to accommodate for old technologies. The project will be presented as a mobile banking service providing data for Android, Web and, if time permits, iPhone mobile application.

## 1.1 Problem

Skydot will be addressing the high cost of maintaining on-premise technology and the hidden transition costs to cloud based services while maximizing the productivity of software development. These problems encompass the following issues in today’s industry:

* On-premise technology costs are very high
* Maintenance costs increase as hardware gets older and therefore must be replaced
* Many people are needed to manage the infrastructure of on-premise technologies
* Disaster recovery sites are needed to reduce risk and are costly to maintain
* Cloud resources can be expensive if not handled efficiently
* Incorporating new technologies while maintaining old software frameworks can become unmanageable and can cause licensing and compatible issues
* Lack of collaboration between teams causes duplication of code and effort, and risks reduction of data integrity

## 1.2 Motivation

The inspiration for this project was ignited at a company I worked with previously who wanted to move to cloud based services. The industry was, and still is, moving in the direction of cloud-based technologies since it can be cost effective, and forward-thinking companies want to stay ahead on the latest technologies. In the end, the company decided upon out-of-the-box software that does much of what I’ve outlined for this project but is more limiting and costly. I believe there is a cheaper, more efficient and more inclusive way of utilizing cloud services. Many frameworks cost from thousands to millions, depending on the needs of the company purchasing the product, and only provide a limited amount of compatible languages and frameworks that developers can use.

## 1.3 Project Goals

1. Decrease the number of people needed to maintain software and the cost of maintaining that software.
2. Increase code integrity and decrease code development and integration time between teams.
3. Counter long, multi-step manual deployment with simple, quick, autonomous cloud deployment.
4. Handle cloud resources as efficiently as possible.
5. Provide a common point of access for client applications.
6. Create a layered, microservice framework that separate client applications from common services.
7. Provide a common point of access to host services and data.
8. Utilize Skydot as the server side of a mobile banking application to present the capabilities of the project.

## 1.4 Proposal Objectives

1. Set up a Kubernetes application container that wraps the entire project and is used for deployments.
2. Utilize docker to wrap micro-apps and micro-services for deployment within Kubernetes
3. Establish an API gateway in Kubernetes through which micro-apps can register and client applications can send requests.
4. Establish a service gateway through Kubernetes that allows micro-apps to make REST requests to micro-services.
5. Optionally establish a host gateway that provides a REST API for micro-services to access backend services. The backend consists of REST and SOAP services.
6. Build an authentication database server to generate and keep tokens for client application requests.
7. Build backend services for authentication, account information, currency conversion and bill payments. At least one WSDL service must be provided.
8. Provide micro-services in Java, Python and C++. Micro-services coverage: Authentication, Account summary and details, Transfers, Bill payment.
9. Build front end application in Kotlin for Android and in ReactJS for web to display services.

## 1.5 Possible Features

1. DevOps: Dashboard, health checking, logging, monitoring, continuous integration and continuous delivery (CICD).
2. Populate a string database with error and warning messages (en\_CA and fr\_CA).
3. Build an iPhone and/or tablet application.
4. Integrate oAuth2 and LDAP authentication.
5. Use Swagger for design and documentation.

## 1.6 Technology and Equipment Requirements

* Microsoft Azure
* Android device(s)
* ReactJS
* Android Studio
* Kubernetes
* Docker
* Minikube

# 2 Background

## 2.1 Architecture

Microservices are currently gaining a lot of attention online in articles, blogs and on social media but also in industry within conference presentations and workshops. However, some skeptics in the software community dismiss microservices as nothing new and claim it is just a rebranding of service-oriented architecture (SOA). However, this is not the case. Microservice architecture has significant benefits, especially when it comes to forwarding agile development and delivering complex enterprise applications. Although it may inherit from SOA (Figure 1 - Microservices vs. SOA), it also solves many problems SOA has and has its own benefits.

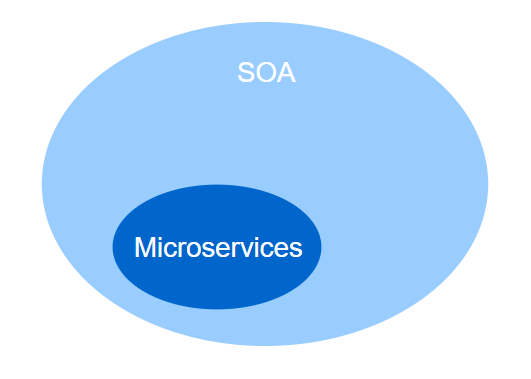


Figure - Microservices vs. SOA

The alternative to microservice architecture is monolithic (Figure 2 – Monolithic vs. SOA vs. Microservices), n-tier and SOA. These traditional models and the inherit disadvantages of substantially less iteration, high maintenance cost, and associated organization has led to the ‘Micro’ trend. Skydots reference architecture is based on the notion of micro-services and micro-apps. In microservices, services can operate and be deployed independently of other services. This way it is easier to deploy new versions of services frequently or scale a service independently. The main difference between SOA and microservices lies in the size and scope. Microservices are independently deployed and significantly smaller then what SOA tends to be, allowing for a more focused decoupling. SOA tends to be large deployments of closely coupled services. The micro theme encourages the separation and break down of code into manageable chunks, while still allowing interaction via a REST interface. Other technologies like WebSockets and gRPC could also be utilized.

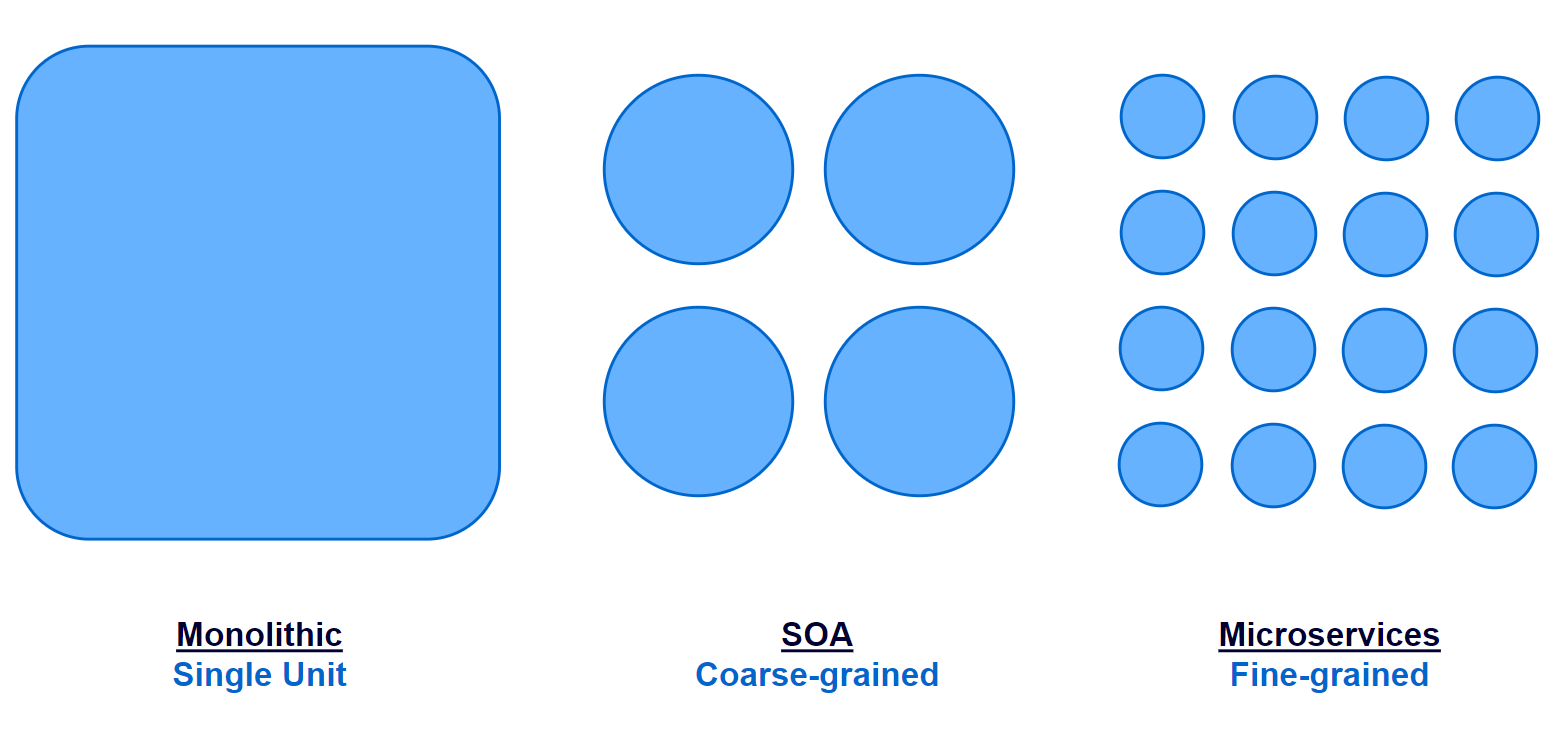


Figure – Monolithic vs. SOA vs. Microservices

However, there are some drawbacks to microservices in comparison to the alternatives. As Fred Brooks wrote in The Mythical Man-Month, there are no silver bullets. One drawback microservices features is the emphasis on service size. While small services are preferable, they are a means to an end, not the primary goal. The goal of microservices is to sufficiently decompose an application to facilitate agile development and deployment but there are still other aspects to the application that need handling that microservices do not cover. An example of this would be service organization, which is where API gateways and service discovery comes in. Problems arises from the complexity of this distributed system. An API gateway needs to be chosen and implemented as an inter-process communication mechanism that also handles partial failure when requests may be unavailable or slow. Although this isn’t extremely complex, the solution is much simpler in a monolithic application. There are more drawbacks to microservices such as dealing with partitioned databases, managing testing, implementing changes that span multiple services and deploying each service. However, Skydot tackles all these problems within microservices and the remaining sections go into detail on the technologies used to do so.

In summary, microservice architecture has both benefits and flaws. However, building complex applications is inherently difficult and architectures such as monolithic and SOA only outperform microservices in a simple, lightweight application. The better choice for complex, evolving applications is microservices, despite some drawbacks and implementation challenges. Though, Skydot aims to alleviate those drawbacks and challenges.

### 2.1.1 Microservice

Skydot utilizes a microservice architecture style. The idea of microservices, in terms of Skydot, is a decomposition of a once monolithic application into self-sufficient modules that perform one function extremely well. This type of structure is very appealing; modules can scale to meet client popularity surge and wane, infrastructure can scale automatically to match the true cost of running these modules; and most importantly, developers and operators can work together as a single DevOps team, working independently of other teams. These teams would deliver modules with customer focused value.

It's reasonable to say that Netflix pioneered the microservices and DevOps approach at a global scale when it had to reinvent itself to meet surging demand for streaming video content. Netflix open-sourced many of its innovations to allow other companies to learn and adapt, and their extensive use of the cloud suddenly made AWS, GCP and Microsoft Azure appealing to both start-ups and enterprises alike. Applications no longer had to be architected around aging infrastructure or obscure billing models. Microservice-based applications and dynamic cloud infrastructure became a template that companies could use to solution these problems.

Netflix's open source stack (OSS) solutions were revolutionary, and they found homes in many commercial Platform-as-a-Service (PaaS) offerings, however they were designed before Docker and the container paradigm exploded. Services running the Netflix OSS were designed to run on entire virtual machines (VMs) which sometimes meant that the VMs were underutilized. The OSS libraries had to solve highly scalable distributed problems such as service discovery and communication, load-balancing, and circuit breaking. They were designed to be integrated tightly into the service code which could only be written in Java. Ports to other languages were handled by the enthusiastic community to varying degrees of success, but they could not solve a fundamental problem: if something foundational needed to be modernized, such as service-to-service communication, every single service had to be updated to use the new technology stack.

I considered utilizing the Netflix stack for the platform because it was a mature offering that was production-proven. However, I quickly realized that it would not fit into the modernized platform I am designing because I wanted three key things from a microservices perspective: polyglot development, multi-occupancy, and platform independence.

**Polyglot development**, or writing applications in multiple languages, was an important Skydot design consideration. Companies have many talented developers who are strong in Java, Objective-C, Swift, and JavaScript, but have few developers who are strong in multiple languages. A full-stack developer can work on both the customer-facing frontend and the mission critical backend. For example, if a development team want an iOS developer to build a microservice, Skydot must be able to support the same familiar language and tools from end to end. Conversely, if a development team decides to rewrite their service in another language for better performance metrics or easier maintainability, there should be nothing to stand in their way.

**Multi-occupancy** is the concept of running multiple applications on the same server or virtual machine, but contained and isolated from each other. This allows for better utilization the infrastructure. This is shown in Skydot with Kubernetes and Docker, Docker for containerization and Kubernetes for orchestration and isolation while running within the same machine.

**Platform independence** in the context of Skydot really means to abstract and decouple an application from its dependencies as much as possible to achieve true modularity; those dependencies can be internal such as relying on prescriptive frameworks like Spring, or external such as a leverageable cloud provider. Containerization technologies like Docker have reduced complexity, but some care still must be taken when choosing libraries and languages; a simple framework choice can reduce a service footprint from 1024MB down to 256MB and choosing another language can reduce that down to as small as 16MB. The smaller the footprint, the more applications that can be run on the same resources (CPU, memory, and storage).

However, containerization hasn't solved the issue where polyglot services need to be able to communicate with each other in a standardized yet modular way, and you don't want to bake load-balancing and service discovery into the code for fear of all the technical debt (and lots of regression testing) when a change in the current accepted solution occurs. This is where external load-balancing and discovery solutions such as Tyk, Apigee and Linkerd come into play. Multiple instances of a service can be running anywhere in a datacentre (or even across multiple datacentres) but appear as a single fixed end-point, which means that the application doesn’t need to rely on heavy duty libraries and leverage simple language platform calls to communicate with other services. By separating the goal-oriented code from the glue code as much as possible, you can iterate and adapt to the pace of technological change.

Microservices often are marketed as a silver bullet to solve all our problems, but they really are designed to solve a specific problem: scaling an organisation quickly enough to meet its customers' needs. Without careful planning and solid architectural design, one could deconstruct what was once a large, impenetrable, monolithic problem into discrete, distributed, and possibly unmanageable problems. This is one of the possibilities Skydot can prevent.

### 2.1.2 Layered

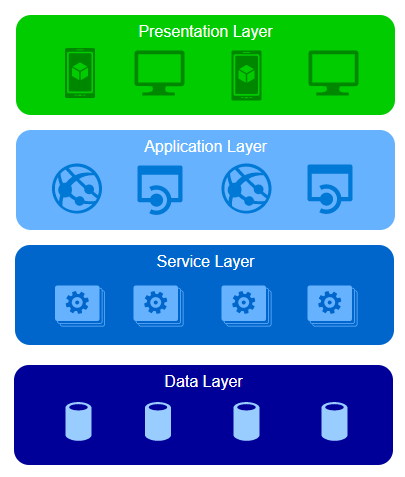


Figure - Diagram of Skydots Layered Architecture

The entirety of Skydot is structured in a strict 4 tier layered style (Figure 3 - Diagram of Skydots Layered Architecture). This layering allows for most of the code to be shared and ensures common services are not duplicated. The reason for making the layering strict is for filtering. Consider that the bottom tier (the data layer) holds all data in the rawest form, this could be JSON, XML, WSDL, various types of SQL, etc., and has an extensive collection of details on each of its stored data objects. Depending upon which type of client you are within the presentation layer you want to receive information on a specific data object, but you only need a portion of that raw data and some of that raw data is accompanied by business logic that you, the client, are not aware of. This is where the application and service layers come into play. The service layer provides the bulk of the business logic shared by all micro-apps within the application layer. All logic within this layer is generic and not tailored to any specific application. For example, formatting would not happen within this layer. That information is then passed to the application layer where the data is filtered even more and structured in a way the client in the presentation layer wants and understands.

This structure of data communication is essential to meeting my goals of decreasing the amount of developers needed to maintain software, decreasing code development time between teams and increasing code integrity. If there is a common service, like pulling account details for a user, there is no need for teams developing separate apps to write their own account detail retrieval service. This service would sit in the service layer and each team would have their own micro-app within the application that only deals with tailoring the information for their application, ensuring separation of customize functionality while preventing duplication of the code base.

## 2.2 Technologies

### 2.2.1 Azure (AKS)

Azure Container Service (AKS) is a managed Kubernetes container orchestration service in Azure. It helps removing the complexity of implementing, installing, maintaining and securing Kubernetes in Azure. To start up AKS is Azure requires the following simple flow (Figure 4 - Azure AKS Start Up Flow),

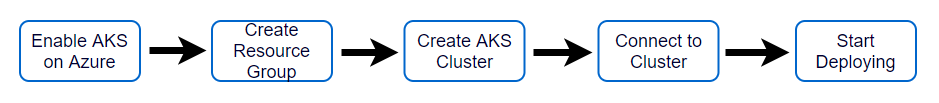


Figure - Azure AKS Start Up Flow

The environment Skydot is working in for this project is Azure so the simplest solution to use is AKS. It also reduces the operational overhead of managing a Kubernetes cluster by offloading much of that responsibly to Azure. Resources like virtual machines, virtual networks and load balancers are created and maintained by Azure so all cost metrics are using Microsoft Azure’s pricing.

### 2.2.2 Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It was built by Google based on their experience running containers in production using an internal cluster management system called Borg. The architecture of Kubernetes has a flexible, loosely-coupled mechanism for service discovery. Kubernetes utilizes clusters for its distributed computing platform. A cluster consists of at least one master node and can have several computing nodes. The master node is responsible for providing the application program interface, kubectl, scheduling deployments and managing the entire cluster. Each node (Figure 5 - Kubernetes Node) in the cluster has a container runtime, in the case of this project Docker is being used but an alternative is rkt. The node also has additional components for logging, monitoring, service discovery and many optional add-ons. Nodes are very important to the cluster as they manage the private and public networking in the cluster and the resources for applications.

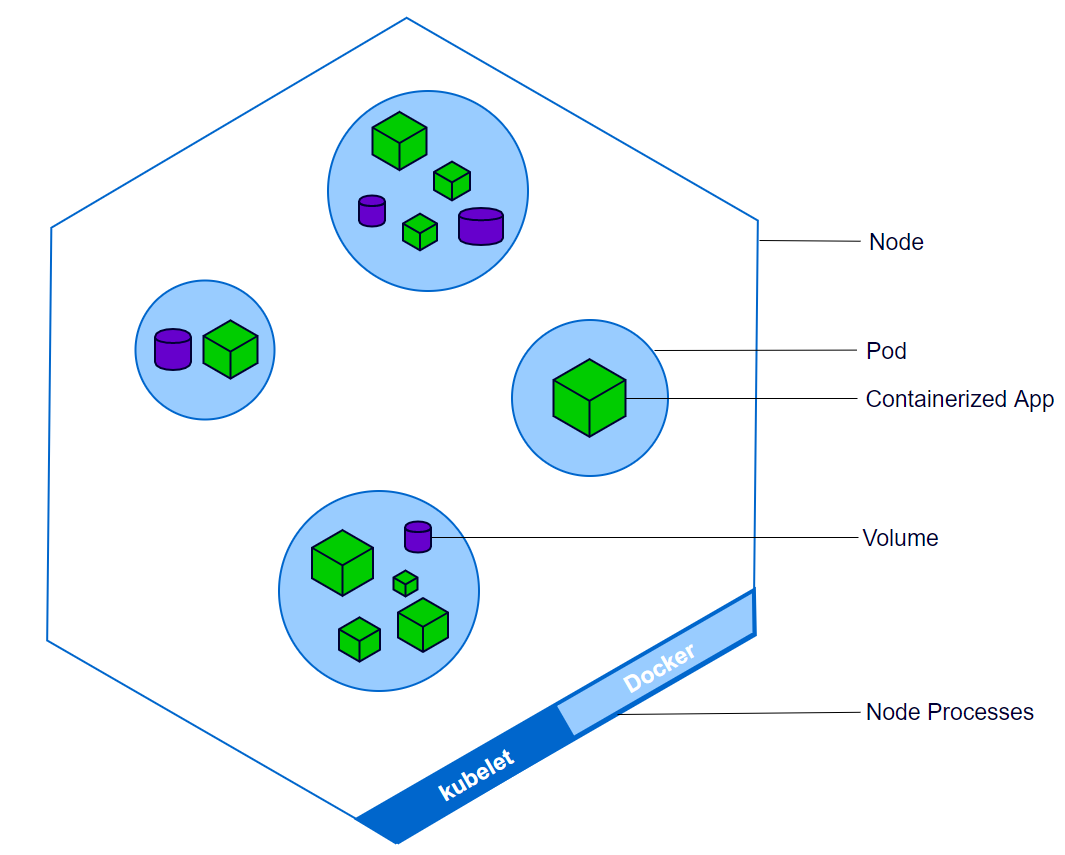


Figure - Kubernetes Node

With a node is a collection of pods, a pod can contain one or more containerized applications as well as storage. All containers in a pod share the same context and resources. Pods can be exposed publicly and privately by services. Privately exposed pods can only be accessed by other pods in the cluster while publicly exposed pods can be accessed by the public through a post on the node it’s within. Each pod can be scaled when managed by a deployment, which is specified in YAML. Scaling can be manual or automated. Deployments allow pods to be disturbed among nodes to provide high availability, thereby tolerating application failures. Replica sets, and load-balanced services can help detect unhealthy pods, remove them and scale up replacements. These resources can also support both “rolling-update” and “recreate” strategies. Rolling updates can specify a maximum number of pods unavailable or a maximum number running during the update process. There are many different types of resources in Kubernetes that can be utilized such as daemons sets, replication controllers, configurations map and much more.

Within Skydot’s cluster there are six nodes. Four of the nodes are language based for micro-services; there is a Python, Cplus, Java and JavaScript node and each only has micro-services with the matching language. There is one node that just runs the host gateway. And the last node holds all the micro-apps. Below is an overview of the cluster (Figure 6 - Skydot In Kubernetes).

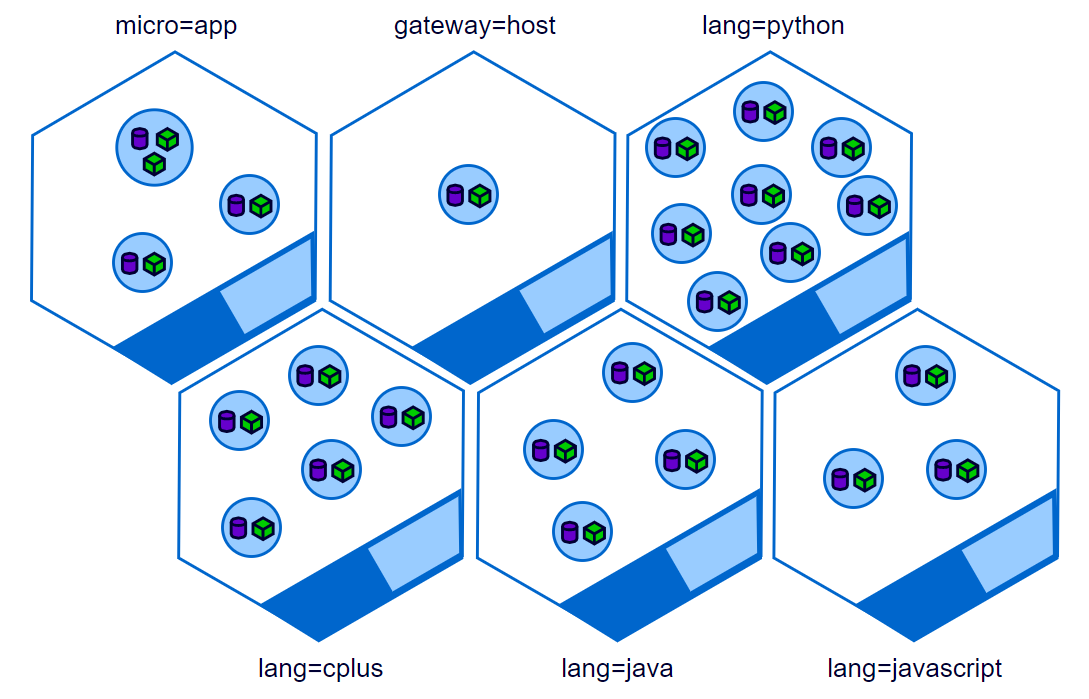


Figure - Skydot In Kubernetes

### 2.2.3 Docker

Docker is a tool designed to make it easer to create, deploy and run applications by using containers. These containers allow developers to package an application in the environment it needs with the resources it needs, such as libraries and dependencies, and deliver it as an all-in-one package. This containerization allows applications to thrive on any Linux machine regardless of custom settings on that machine. This ensures that a container can run on a developer machine and a production machine without issue.

The dockerized containers are called images. Images are similar to virtual machines except they aren’t a whole virtual operating system. Docker allows images to use the same Linux kernel as the system they’re running on and only requires images be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the image.

To build a docker image, one would need the application they want to dockerize and a dockerfile specifying how to compile and run the program. Here is an example of the dockerfile needed to containerize a Python application,

FROM python:alpine

ADD . /code

WORKDIR /code

RUN pip install -r requirements.txt

CMD ["python", "app.py"]

The first line specifies the environment the application runs in and the docker version of the environment. The Alpine version of things is a popular version and usually is a lighter image. The second and third lines move the code into a working directory in the container and the fourth line installs all the required libraries need for the Python application, as specified in a requirements text file. Finally, the last line depicts how to run this application. The dockerfile paired with the command,

docker build -t <python-app> .

Builds the image to the developers’ local repository. This image can then be run locally or be pushed to a repository on Docker Hub. Docker Hub is a cloud-based registry services which allows you to link to code repositories, build your images and test them, stores manually pushed images, and links to Docker Cloud so you can deploy images to your hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

In terms of Skydot, applications are built and dockerized locally. Then those images are pushed to Skydot’s Docker Hub repository. Once the images are public, deployments in Kubernetes can pull and run the images in pods just by adding the following line to the YAML definitions that configure each deployment container,

image: <repository>/<image>:<version>

## 2.3 Alternatives

Although I’ve chosen to utilize an API gateway, Kubernetes, there is another approach to automation, service discovery and containerization. That is the use of an off-the-shelf platform-as-as-service (PaaS) (see sub-section, ‘What is a PaaS?’) such as Cloud Foundry. A PaaS provides developers with an easy way to deploy and manage their microservices. It facades the procuring and configuring of IT resources and can ensure compliance with best practices and company policies. However, if one goes with an off-the-shelf technology, they end up in a proprietary system that is only so flexible. Off-the-shelf PaaS technologies can also cost a lot of money and time as products like Cloud Foundry are open source but are only free up to a certain point and may require time to fit to your application or system. Companies also will require a better version than the open source one (enterprise version) and support for it. As shown in Figure 3 - Pivotal Cloud Foundry architecture – open source and enterprise, there are many things listed in the commercial extension that would be desirable to a company. So, there will be a cost for the upgraded version, which scales higher and higher depending on the size of the company using the product, and a cost for the support provided.

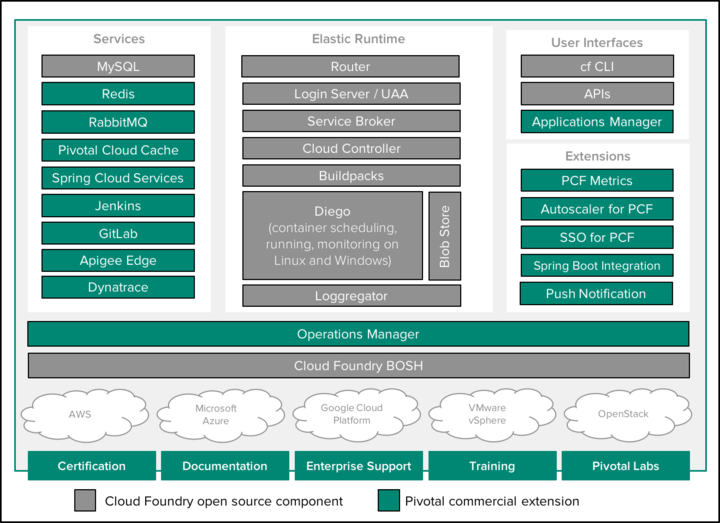


Figure - Pivotal Cloud Foundry architecture – open source and enterprise

Essentially, Skydot is a make-your-own PaaS which utilizes a clustering solution. It does this by employing container technologies such as Kubernetes and Docker Swarm which are described later in this report.

#### What is a PaaS?

There are three levels of cloud-service abstractions: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS category, gives users the basic infrastructure needed to build and deploy an application. PaaS products offer a higher level of abstraction, so the user won’t be exposed to the O/S, middleware or runtime and needs only to concern themselves with the application and data. And lastly, SaaS products are applications built and hosted by a third-party platform and made available to users via the internet.

A PaaS is a platform upon which developers can build and deploy applications. These products offer a higher level of abstraction than we get from IaaS products meaning that, beyond networking, storage and servers, the application’s O/S, middleware and runtime are all managed by the PaaS. [1] These cloud-service abstractions are graphically represented in Figure 4 - Cloud-Native Service Models Comparison.

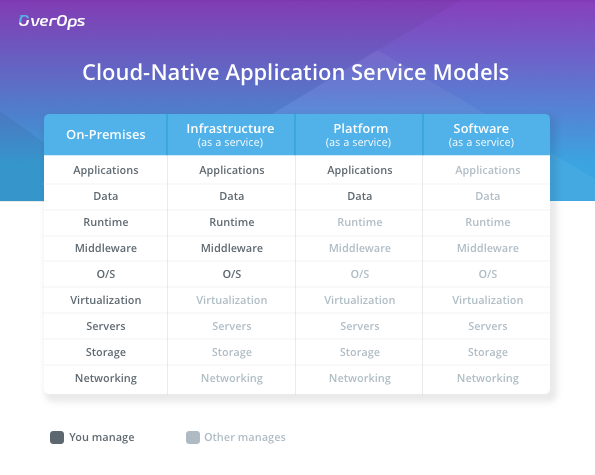


Figure - Cloud-Native Service Models Comparison

# 3 Solution

## 3.1 Client

There are two client applications that were made for this project: an Android application and a web application. The Android application was built in both Java and Kotlin while the web application was built in JavaScript. The significance behind building two client applications is to show how two projects using the same services would work within Skydot’s architecture. Each client application has a micro-app built specifically for it. The mobile micro-app services the Android application. It returns data differently than the web micro-app, which services the web application, because a mobile device can’t display as much information on one page as a web application in a browser can. For example, on a bill payee call, which would return a list of bill payees one can use for a bill payment, the Android application provides search functionality for a payee since displaying the entire list of payees takes up too much screen space and would be tedious to scroll through. Whereas the web application can display the entire list, while also providing search functionality, because there is more screen real estate to work with. So, in terms of each micro-app, the mobile micro-app would provide a bill payee search endpoint and the web micro-app would provide a bill payee search and bill payee get all endpoint.

## 3.2 Skydot

Skydot is comprised of three sections: the micro-apps, the micro-services and the host gateway. These three sections are essential to the project as they maintain access control, business logic and data translation. The micro-apps layer is the only one that is publicly accessible while the host gateway and micro-services support the micro-apps from within Skydot’s private virtual network. Although all three layers can access the internet, if need be, only the host gateway can access private backend data services and storages that provide access to secure client information. The following sub-sections further outline each layer.

### 3.2.1 Micro-Apps

The micro-apps are the application layer of Skydots architecture (Figure 9 - Skydots Micro-apps). They sit between the presentation layer and the service layer. Each micro-app is made by a team in service to usually one client application within the presentation layer. In terms of this project, this is represented with the web and mobile micro-apps. The mobile micro-app tailors to the needs of mobile client applications while the web micro-app works with web applications. Both micro-apps require the same backend services but differ is things like request parameters, return types, amount of information returned and development language.

The exception to this template is the authentication micro-app. The authentication (auth) micro-app is not specific to any client. The auth micro-app controls access to all services through token validation; it acts as the central hub for authentication. All clients must login through the auth micro-app to receive a token that will allow them to access other micro-apps. A request to the mobile or web micro-app will be rejected if there isn’t a token in the request. All micro-apps must then verify if the token is valid by requesting verification from the authentication micro-app.

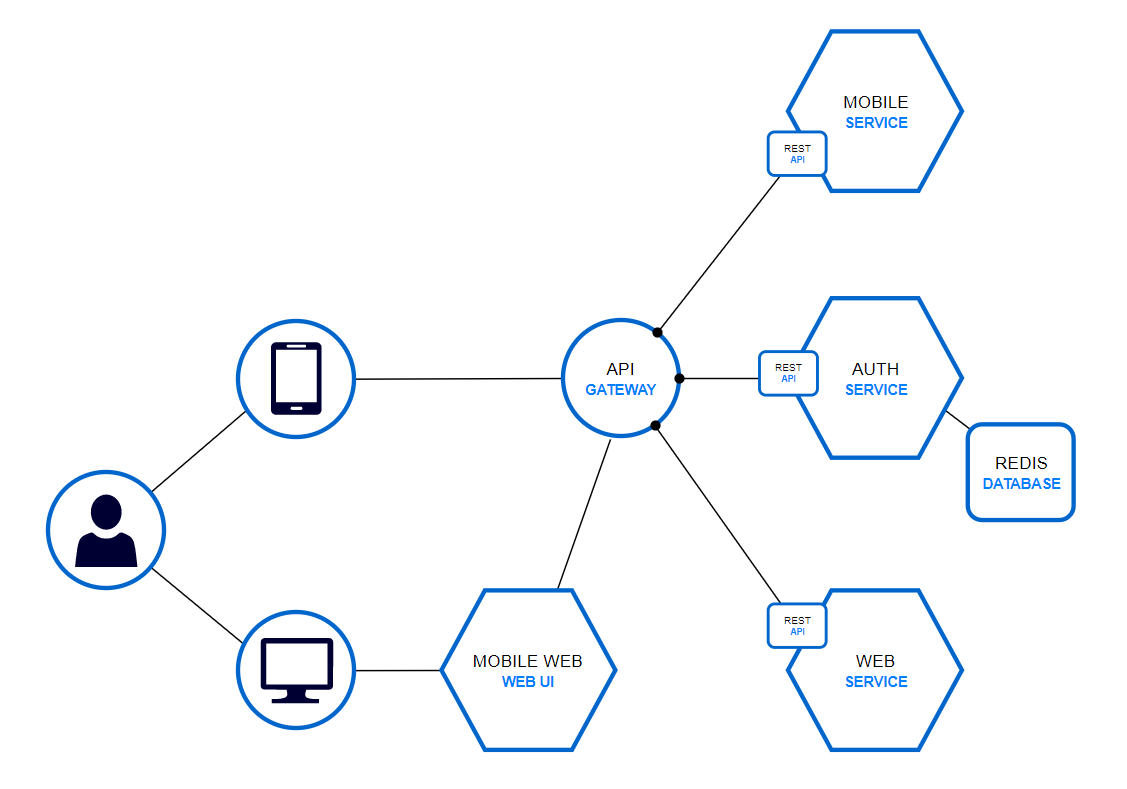


Figure - Skydots Micro-apps

Since the micro-apps are running within the same cluster, the micro-apps can use service discovery to quickly access the auth micro-app without having to know the auth micro-apps internal cluster IP address or having to make a request out of the cluster to the auth micro-apps external IP or DNS. All micro-apps can access the auth micro-app at ‘auth-app’, which is the service name I specified in the YAML definition. This ensures that each micro-app doesn’t need to know the specifics of other micro-app (i.e., cluster IP, number of pods, node location, etc) to make a request to it. If the auth micro-app validates the token the user is valid, and request can go through, provided the any other required parameters are present.

|  |  |  |  |
| --- | --- | --- | --- |
| Micro-App | Service Name | Language | Used By |
| Authentication | auth-app | Python | All clients and micro-apps |
| Mobile | mobile-app | Java, Kotlin | Mobile client |
| Web | web-app | JavaScript | Web client |

Table - Micro-apps

**Authentication Micro-App –** This micro-app handles everything to do with authentication. It is developed in python and utilizes a redis cache to store logged in users. There are four endpoints that the app handles: /auth/login, /auth/logout, /auth/verify and /auth/user. These endpoints allow clients to login, and receive a session token, and logout. And they allow micro-apps to verify client session tokens and retrieve the user id that is encrypted into the token. No micro-app, other than the auth micro-app, knows how to encrypt or decrypt session tokens. This ensures that all login tokens are handled by one application and that every client logs in the same way.

**Mobile Micro-App –** This micro-app handles all service requests a mobile client can make except logging in. Here is a list of all the endpoints it provides and what micro-service each endpoint goes to in default mode.

|  |  |  |
| --- | --- | --- |
| Endpoint | Method | Micro-Service |
| /account | GET | None |
| /account | POST | account-summary-service-python |
| /account/details | POST | account-details-service-python |
| /bill | GET | None |
| /bill | POST | bill-payment-service-javascript |
| /bill/payee | POST | bill-payee-service-javascript |
| /bill/payee/search | POST | bill-payee-service-javascript |
| /transfer | GET | None |
| /transfer | POST | transfer-service-java |
| /user | GET | None |
| /user/create | POST | create-service-cplus |
| /user/delete | POST | delete-service-cplus |

Table - Mobile Micro-App Endpoints

**Web Micro-App –** This micro-app handles all service requests a web client can make except logging in. Here is a list of all the endpoints it provides and what micro-service each endpoint goes to in default mode.

|  |  |  |
| --- | --- | --- |
| Endpoint | Method | Micro-Service |
| /account | GET | None |
| /account | POST | account-summary-service-python |
| /account/details | POST | account-details-service-python |
| /bill | GET | None |
| /bill | POST | bill-payment-service-cplus |
| /bill/payee | POST | bill-payee-service-java |
| /bill/payee/search | POST | bill-payee-service-java |
| /transfer | GET | None |
| /transfer | POST | transfer-service- javascript |
| /user | GET | None |
| /user/create | POST | create-service- python |
| /user/delete | POST | delete-service- python |

Table - Web Micro-App Endpoints

Each micro-app has its own external IP and DNS so that, even if the micro-apps have the same endpoints, each request will go to the correct micro-app. The DNS are as follows:

**Auth** – auth-skydot.<azure-dns>.com

**Mobile** – mobile-skydot.<azure-dns>.com

**Web** - web-skydot.<azure-dns>.com

As the banking application part of this project is just for demonstration purposes, I have used the default DNS Azure provides for public IP addresses. However, it is possible to use ones own custom domain if they own that domain.

### 3.2.2 Micro-Services

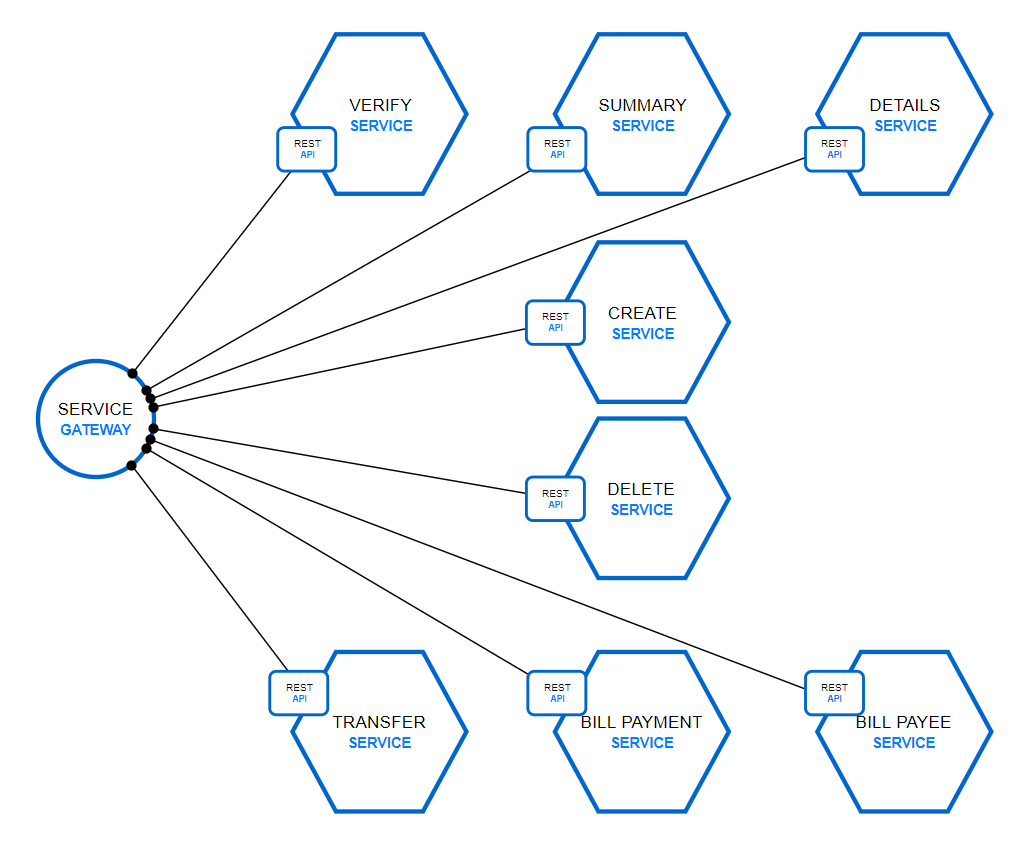


Figure 10 - Skydots Micro-Services

The micro-services are within Skydots service layer (Figure 10 - Skydots Micro-Services). This layer is between the application and data layers. Micro-services are common services used by micro-apps in the application layer. They typically provide only one service. In this project I have five categories of services and a total of eight services between them. A list of categorized services follows (Table 4 - Banking Services):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Authentication | User | Account | Transfers | Bills |
| - verify | - create  - delete | - account summary  - account details | - transfer | - bill payment  - bill payee |

Table - Banking Services

However, there are total of nineteen micro-services deployed in Skydot. This is because micro-services can be developed in almost any language. If the micro-service development language can provide a REST api then it can be used. So, I have set up duplicate services in different languages. This does change service discovery a bit since you cannot deploy multiple services with the same name. Therefore, I have a simple naming format for micro-services which is as follows:

<service name>-service-<language>

This format depicts the service name needed in service discovery. All micro-services are private and can only be accessed within the clusters’ private virtual network. Although each micro-service has a cluster IP, that IP is dynamic and will change when a new pod of that service is created. So, the service name in my specified format is used, which is defined in the YAML for each service. Below is a list of all micro-services (Table 5 - Micro-Services):

|  |  |  |  |
| --- | --- | --- | --- |
| Service | Service Name | Endpoint(s) | Languages |
| Authentication | verify | /auth | C++\*, Python |
| Account Summary | account-summary | /account/summary | Python |
| Account Details | account-details | /account/details | Python |
| User Create | create | /create | C++, Python |
| User Delete | delete | /delete | C++, Python |
| Transfers | transfer | /transfer | C++, Java, JavaScript, Python, Spring |
| Bills | bill-payment | /bill | C++, Java, JavaScript, Python |
| Bills | bill-payee | /bill/payee  /bill/payee/search | Java, JavaScript, Python |

Table - Micro-Services

*\*When formatting service name for a C++ service, the word ‘cplus’ is used instead. Ex, create-service-cplus*

### 3.2.3 Host Gateway

The host gateway is part of the bottom most layer to Skydots layered architecture (Figure 11 - Skydots Host Gateway). It fronts backend legacy services and databases. These legacy endpoints could return data in any kind of format. For this project there are two return types: JSON and XML (WSDL).

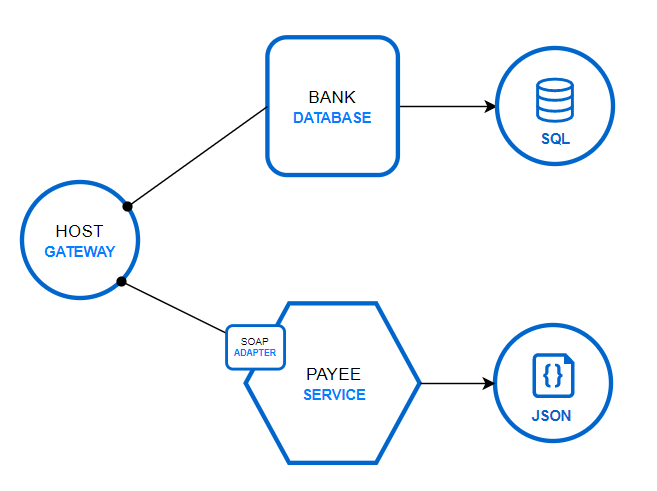


Figure - Skydots Host Gateway

There are two parts to the host gateway: the SQL part and the WSDL part. The bank database holds all bank account data, this includes profile data, account data and transaction history. The payee service is a SOAP service that provides a list of bill payees that the user can make bill payments to. The payee service represents legacy systems that use older technologies like, in this case, WSDL. The bank database utilizes a SQLServer and represents newer technologies. The host gateway transforms data returned from both into a format suitable to return to the micro-services. For this project, JSON is the data type used within the cluster, so the host gateway converts the data from the payee service, XML, to JSON and just acts as a passthrough for the returned data from the SQLServer. This is no business logic done within the host gateway.

In the original design, the host gateway would be located within the cluster so that Kubernetes could provide scaling for it. Unfortunately, due to the network I am testing on, outbound requests on port 1433 are blacklisted which is the port SQLServer runs on. Without access to port 1433 I cannot test Skydot locally. The work around for this is to have the host gateway running outside of the cluster. Due to this, there may be a slight increase in latency.

## 3.3 Backend

As mentioned above in the host gateway section, there are two parts to the backend. There is a SQL database that contains all bank related information and a payee service that returns a list of available bill payees. The SQLServer represents technologies currently used in industry while the payee service is a SOAP service and represents legacy technologies. These legacy systems utilize out-of-date technologies but are usually too big or too expensive to upgrade or convert to a newer solution. This flaw makes it so that the legacy system must be used instead of implementing something new. The payee SOAP service is available at the following endpoint (Figure 12 - SOAP Service):

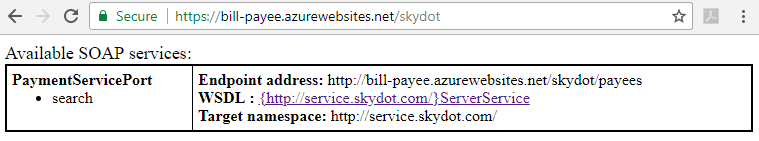


Figure - SOAP Service

And preforming a search on the service returns a response like this (Figure 13 - SOAP Service Response WSDL):

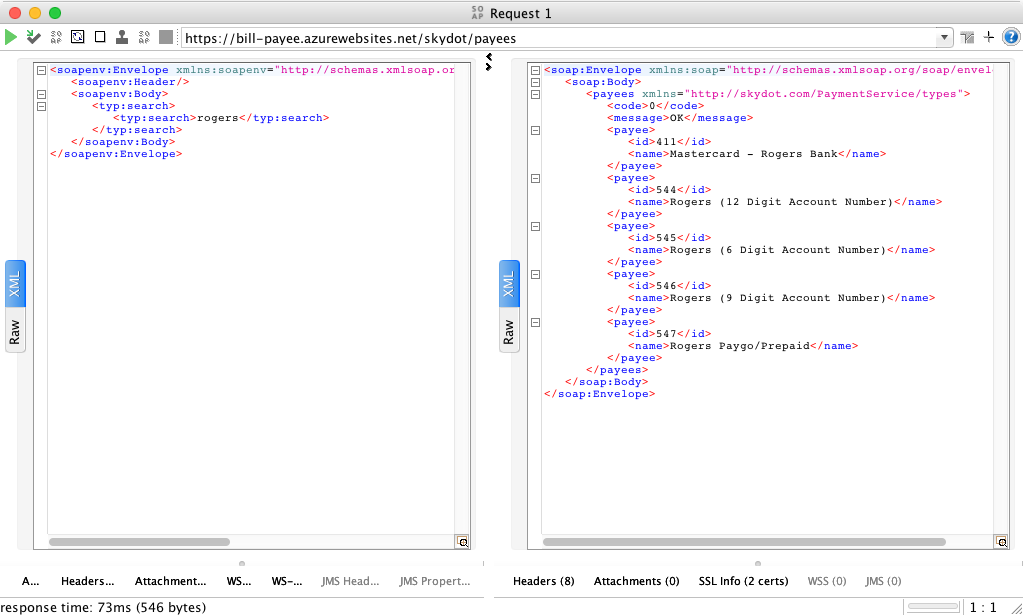


Figure - SOAP Service Response WSDL

This is the type of response (on the right) that is parsed by the host gateway into JSON data. The following shows the host gateway handling the same search request but now JSON is returned (Figure 14 - SOAP Service Response JSON):

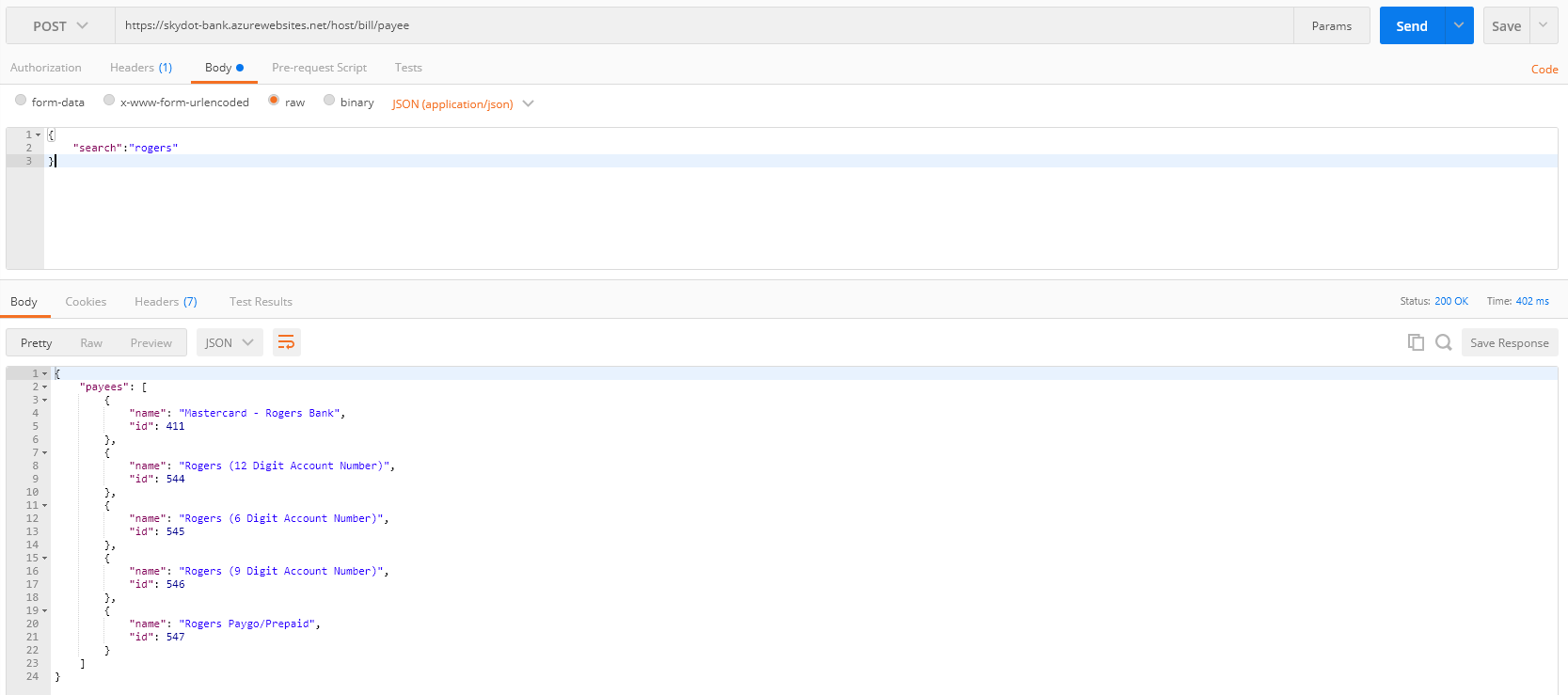


Figure - SOAP Service Response JSON

As for the banking SQLServer, there are three data tables that hold banking information. The first is the Profile table. This table holds a users’ user id and password which are used to verify a user when they log in. The user id is used in all three tables to connect each account and transaction to the appropriate user.

The second table is the Account table. This table contains data pertaining to each account a user has. Each account has a unique id, a name, a type and an amount in both Canadian and American currency. There are three types of bank accounts: banking (e.g., debit), borrowing (e.g., credit) and investing (i.e., investments but not brokerage). With different types of accounts business rules can be enforced by the micro-services. Rules like investing account can’t transfer to other accounts or banking accounts can’t pay more than its available funds.

And lastly, the third table is the History table. This table contains transaction history for each account. When a transfer or bill payment is made, a record to make when all information relevant to the transaction (i.e., date, amount, account id).

The entity relationship of the database is as follows (Figure 15 - Database Entity Relationship). A profile must have at least one account, but an account doesn’t have to have any transactions.

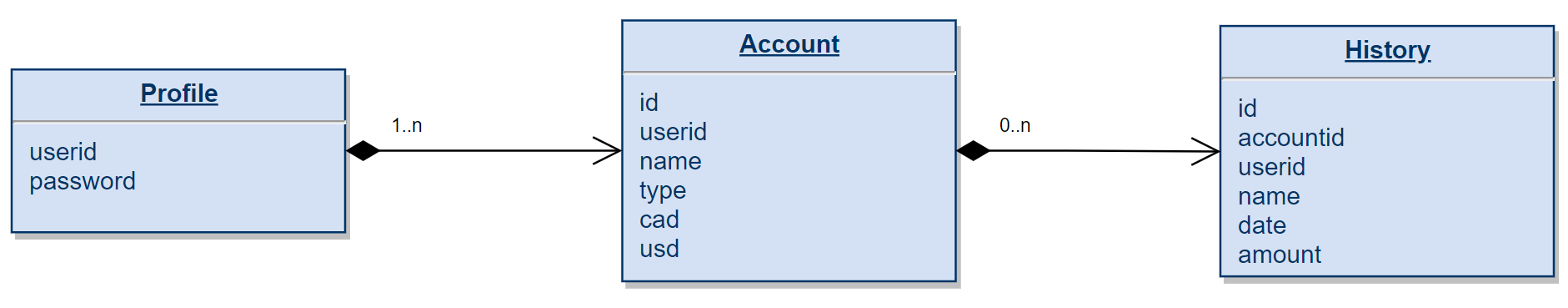


Figure - Database Entity Relationship

## 3.4 DevOps

### 3.4.1 Dashboard

Kubernetes offers a web-based user interface (Figure 16 - Kubernetes Dashboard) that delivers an overview of the cluster Skydot is running on. However, not only does the dashboard give an overview of the applications running on the cluster, it also provides the ability to deploy containerized applications, troubleshoot containerized applications and manage cluster resources. The dashboard can be used to create, or modify, all kinds of resources within Kubernetes such as Deployments, Pods, DaemonSets, Services and much more. It also provides an API for scaling, updating and restarting pods. Logs and YAML definitions can be viewed in the dashboard and the Kubernetes dashboard even provides CPU and Memory usage graphs for each pod.

An alternative dashboard was not needed as the out-of-the-box dashboard was a suitable for this project.

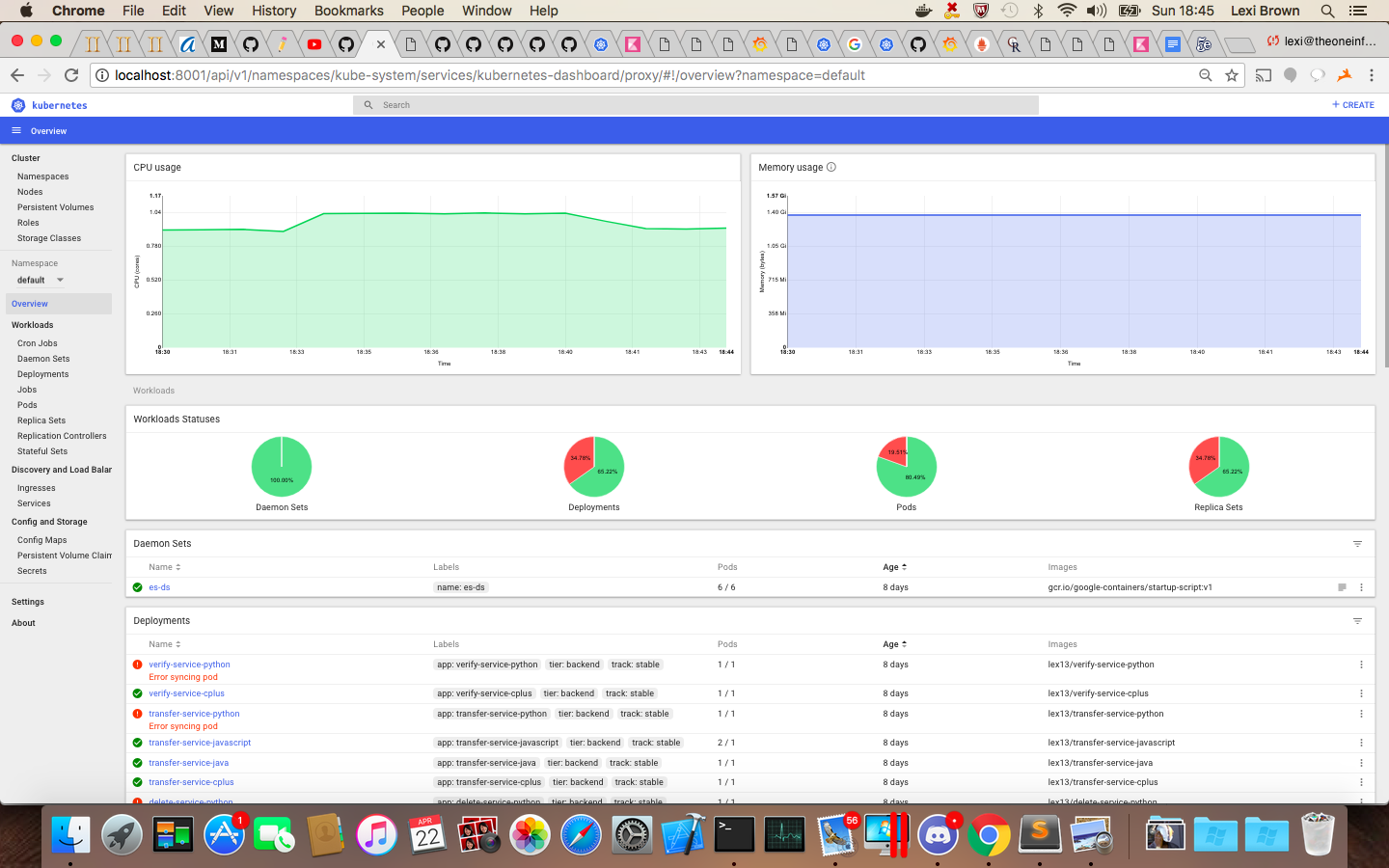


Figure - Kubernetes Dashboard

### 3.4.2 Logging

The logging that was used is EFK style logging. EFK logging (Figure 17 - EFK Logging Libraries) is the usage of Elasticsearch, Fluentd and Kibana to organize and display logs. Within Kubernetes, each pod logs information in its own way, some just stream to stdout while others generate log files. Fluentd is used to find each pods logs and parse, filter and enrich them. It then allows Elasticsearch to take those compiled logs and index and store them for quick and reliable searching. From there, Kibana uses Elasticsearch as a data source for its web-based dashboard. The dashboard provides searching capabilities, through Elasticsearch, and graphical representations of the logs.

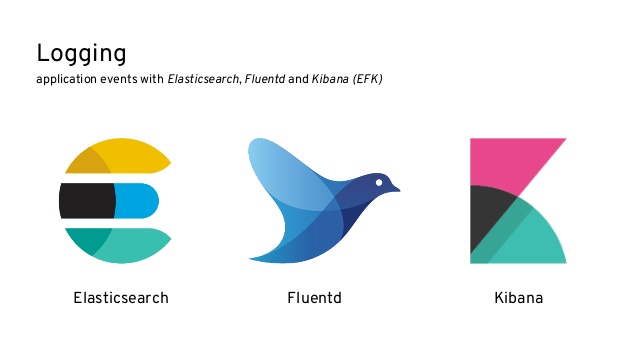


Figure - EFK Logging Libraries

An alternative to EFK is ELK, which utilizes Logstash instead of Fluentd. However, there isn’t much of a difference between the two as Fluentd was inspired by Logstash

### 3.4.3 Monitoring

There were two monitoring system applied to Skydot: InfluxDB and Prometheus (Figure 18 - Monitoring Libraries). Both provide monitoring for Kubernetes pods and nodes, and both utilize Grafana. Grafana has a web-based dashboard that offers many kinds of graphs to show metrics.

Figure - Monitoring Libraries

InfluxDB was used first as it gives basic metrics readings and then Prometheus was added on later as it gives more detailed information on metrics. This is because InfluxDB is a push-based system whereas Prometheus is a pull-based system. A push-based system requires the application tracking metrics, in this case Heapster, to actively push data into the monitoring system. While a pull-based based system fetches the metrics values from Heapster periodically. The centralized control of how polling is done in Prometheus makes it easier to adjust configurations and can act as a synthetic health check monitor. Although Prometheus delivered more out of the box, both were utilized, and all metrics collected in this report were either monitored through InfluxDB or Prometheus.

# 4 Results/Validation

TODO

## Project Goals

1. Decrease the number of people needed to maintain software and the cost of maintaining that software.
2. Increase code integrity and decrease code development and integration time between teams.
3. Counter long, multi-step manual deployment with simple, quick, autonomous cloud deployment.
4. Handle cloud resources as efficiently as possible.
5. Provide a common point of access for client applications.
6. Create a layered, microservice framework that separate client applications from common services.
7. Provide a common point of access to host services and data.
8. Utilize Skydot as the server side of a mobile banking application to present the capabilities of the project.

# 5 Conclusion

In conclusion, the goal of Skydot is not just to use a “Micro Architecture” within the cloud. It’s a culture and an end-to-end process. Using all technologies and designs I’ve chosen for Skydot, all project goals can be met. However, even with these decisions, Skydot is decoupled enough that integrating a new technology (i.e., adding Cassandra) or shifting to a different technology (i.e., Kubernetes to Docker Swarm) would not bring down the whole system and require extensive conversion time. Testing the system proved that any language or framework can be utilized in Skydot however there are some options that are more cost and performance efficient than others. Developing in a cloud environment requires one to be as efficient with resources as possible to prevent investing too much money into the project. It was seen that Spring, for example, was a heavy resource choice and would negatively contribute to cost of maintaining the project. Alternative solutions like Python and JavaScript proved to be more cost efficient and performed well in the cloud environment. Although both solutions can be utilized in Skydot, research and validation showed that the latter option yielded more benefits without hindering performance and development, both individual and as a team.

Skydot is a template that any company will be able to use, understand and customize for their needs. When used properly, one should be able to see the improvements Skydot brings to their application within cost, maintainability, developer management, productivity and adaptability/flexibility.

## 5.1 Future Work

### 5.1.1 Multi-Cloud

After implementing Skydot within Microsoft Azure, future considerations would be integrating a multi-cloud design. With CICD and the flexibly of Skydot, it would be possible to utilized both Microsoft Azure and Amazon Web Services (AWS) (Figure 19 - Possible future Skydot usage with AWS).

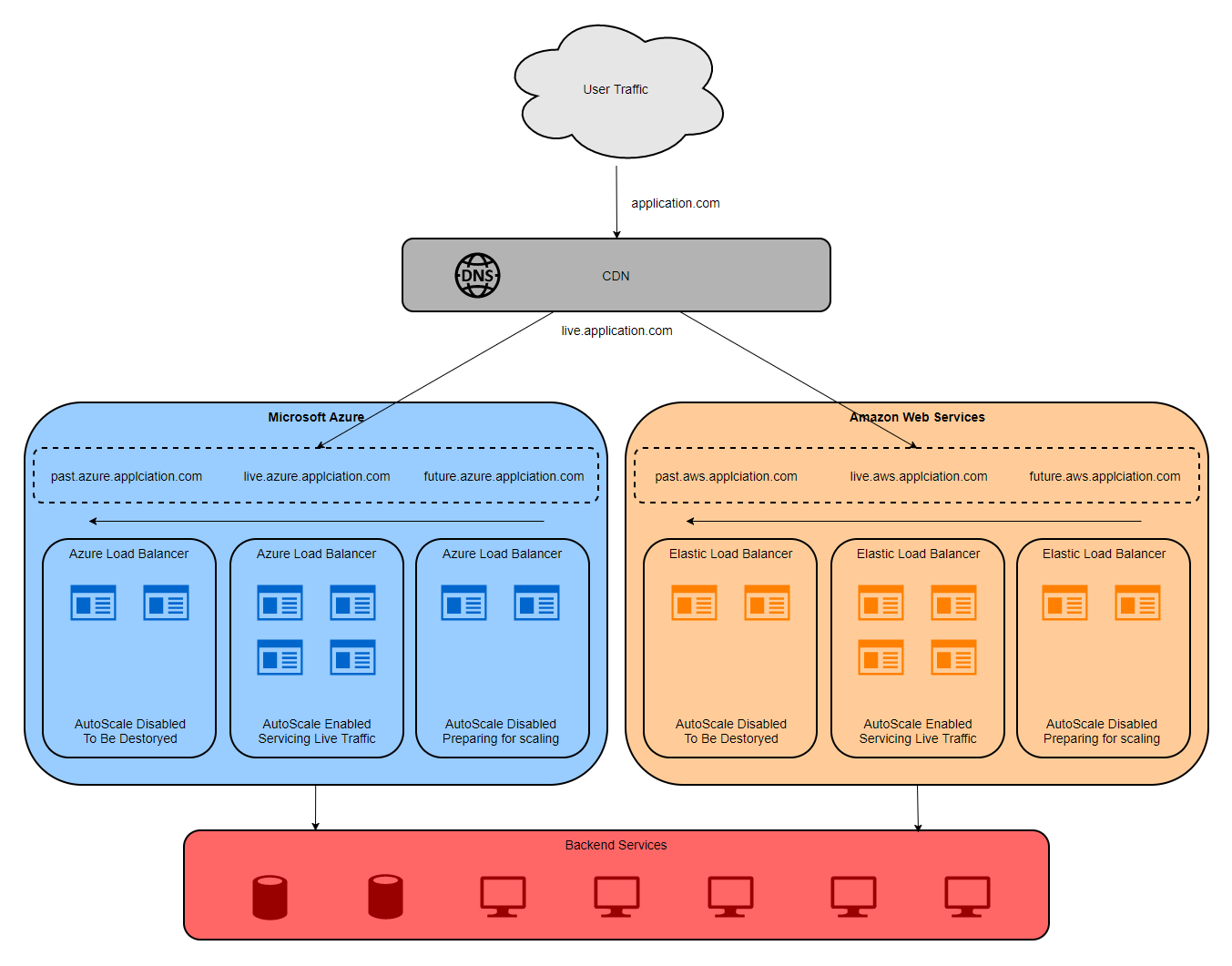


Figure - Possible future Skydot usage with AWS

This way one could compare the cost and usability of Azure and AWS, and show off Skydots portability, while also presenting a possible multi-cloud implementation. With an added layer between the client applications and the cloud services, a content delivery network (CDN) could be added to handle the load balancing between Azure and AWS. Additions would be a DNS cycle on both cloud services where the CDN looks for the live DNS on the cloud hosts to direct traffic to and the cloud hosts also have a past and future version of Skydot behind different DNSs. The past and future DNSs would help compare and test old and new versions of micro-app and micro-services in both environments as it is possible Azure and AWS could handle Skydot differently.

### 5.1.2 Improvements

Some more future improvements one could make to Skydot are to implement a CICD system, integrate a string database, build alternative client applications, add commonly used authentication services and utilized Swagger definitions. These are all things that could not be tackled in the timeframe of this project as the primary goals were more important to focus on. However, most of these would be required in a full, enterprise implementation of Skydot.

#### CICD

Continuous integration (CI) is a process where developers and testers collaboratively validate new code and continuous delivery (CD) is a process where releasable artifacts are continuously being created. So together CICD is a process of continuous development, testing and delivery of code. There are several different technologies that could be used in Skydot for this like Jenkins or Bamboo. Though to implement a full and well-done CICD process there are several technologies that should be put in place: an artifact repository (for CD), issue tracking and version control. Here is a chart of the possible choices (Table 6 - CICD Technologies).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Issue Tracking | Version Control | Continuous Integration | Continuous Delivery | Artifact Repository |
| Image result for jira | Image result for bitbucket atlassian | Image result for bamboo cicd | | Related image |
| Image result for jira | Image result for bitbucket atlassian | Image result for teamcity | Image result for octopus deployment | Image result for artifactory |
| Image result for jira | Image result for github enterprise | Image result for jenkins | Image result for urbancode | Related image |

Table - CICD Technologies

#### String Database

An important aspect of a server is the handling of errors and messages. As micro-services do not do any formatting or localization, there needs to be a way for them to return messages that all clients can understand. A good way of doing this is having a database of string key-value pairs. The key represents the error or warning or message that has occurred. The key can be sent back from the micro-apps, the micro-services, the host gateway or the backend. Then, once the key reaches back to a micro-app, the micro-app, which knows the localization of the client (i.e., English, French, Spanish, etc), can query the string database for the appropriate message to return to the client. Here is an example error message flow (Figure 20 - Error Message Flow),

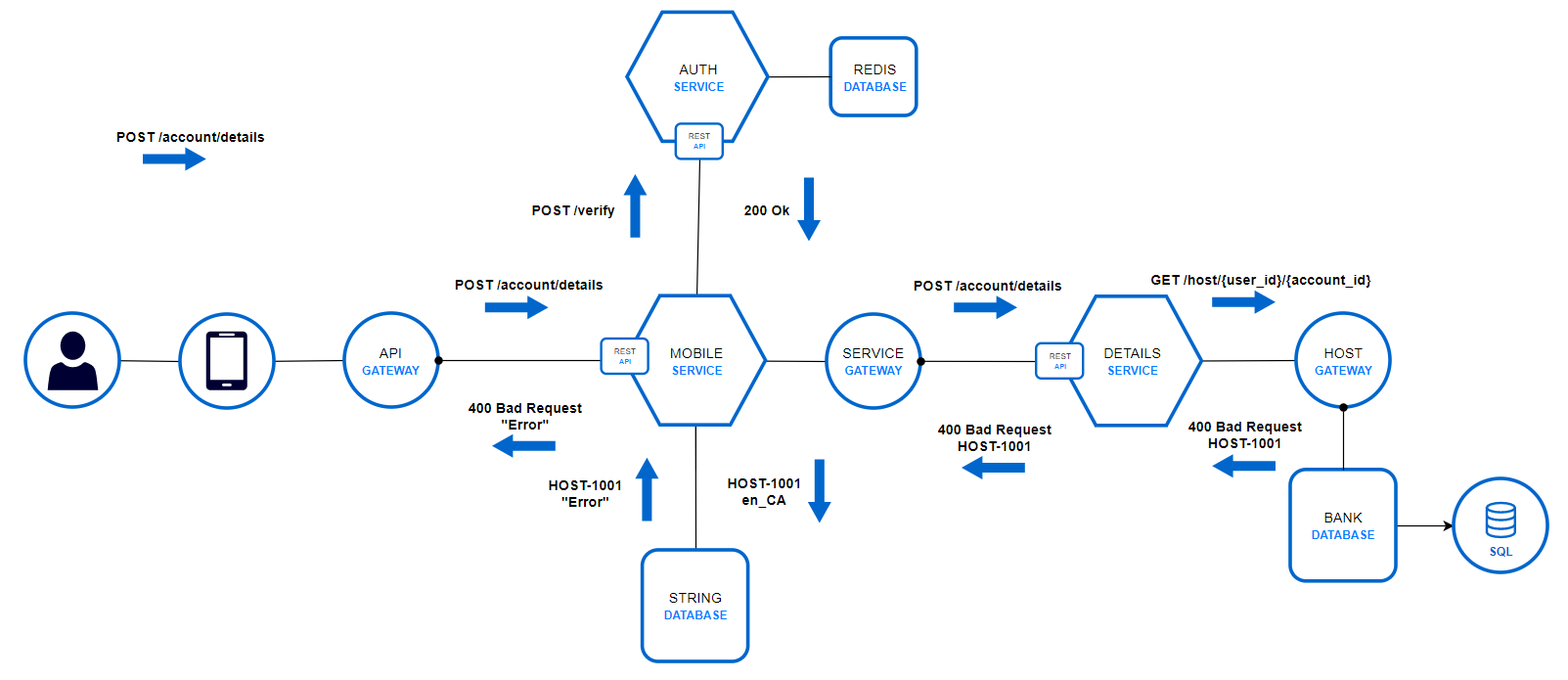


Figure - Error Message Flow

In this scenario, the user tries to request details on an account that does not exist. After the requests token is validated, and the request passes through the Details service and host gateway, the backend returns that such an account does not exist for this user and the error key sent back is HOST-1001. The Details micro-service just passes the message through to the micro-apps as it has no business login to preform on such a response. However, the mobile micro-app now must retrieve the message associated with that error key. To do this, it must send to the string database the error key and the localization key for the user, in this case en\_CA (English [Canada]). The database would then return the message associated with the key HOST-1001 in English, which in this example is “Error”, and the micro-app would return that error message to the user. The same applies for warnings or information messages.

This setup would ensure that all messages shown to the client match regardless of the clients’ platform. So, producing an error on the Android client yields the same error message as producing that error on a web client. Also, this format saves storage space as each micro-app won’t have to have its own storage of those string. Instead a common, scalable access point will provide for all micro-apps.

#### Client Applications

Adding more client applications would just show off the flexibility of Skydot and highlight how simple it is to add more micro-apps. Building an iPhone, tablet, UWP or any IOT client is as easy as just adding a new micro-app. Plus that addition does not affect any other micro-app. Each client can have its own way of wanting data to be returned to it. For example, perhaps the iPhone client application requires the return type of responses to be XML. The iPhone micro-app would convert response to XML specifically for this client even though Skydot’s internal communication is all JSON.

#### OpenAPI (Swagger)

The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases.

Using OpenAPI gives developers a template to work with, both when they build a service or when they want to call another service. When they are building a service, the specifications outline exactly what the service should return so there is no ambiguity involved or returning whatever the developer wants to. When they are utilizing a service, that may or may not be completed, there is no uncertainty to what that service will return in any scenario as the specification outlines the possibilities.

The Spring Transfers service is one service that utilizes OpenAPI specifications. Here is how the service is defined,

swagger: '2.0'

info:

title: Skydot Transfer

description: Money transfer

version: 1.0.0

host: localhost:8080

schemes:

- https

- http

basePath: /

paths:

/transfer:

post:

consumes:

- application/json

produces:

- application/json

description:

Transfers money.

parameters:

- in: body

name: TransferRequest

description: Transfer request data

schema:

$ref: '#/definitions/TransferRequest'

responses:

'200':

description: Transfer response data

schema:

$ref: '#/definitions/TransferResponse'

'400':

description: Bad request

schema:

$ref: '#/definitions/TransferResponse'

Following this would be the definitions of the request and response object: TransferRequest and TransferResponse. The definition states the type of object they are and the properties of the object with their types. Therefore, in theory, one could represent the entirety of Skydot services as a collection of OpenAPI specifications as the language and libraries used within the services is not relevant to the definitions and duplicate services written different languages would have the exact same RESTful specifications.

#### Authentication

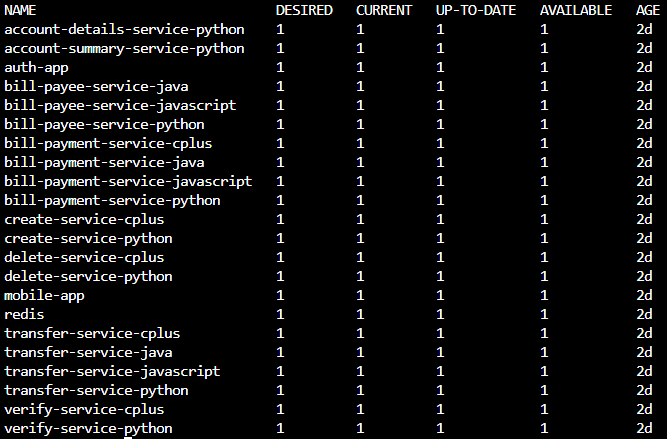
Currently, Skydot is only using simple token encryption and decryption for authentication. However, a more secure and mainstream authentication solution would be to use oAuth 2.0 (Figure 21 - 0Auth 2.0 Protocol Flow). OAuth 2.0 is the industry standard protocol for authentication.



Figure - 0Auth 2.0 Protocol Flow

It works similar to Skydot’s current method of authentication where the client must request authorization from an authentication server before they have access to other server resources. However, this method is more secure as it allows sharing of data without having to release personal information. So, in an enterprise version of Skydot, this is a more acceptable means of authentication.

# Additional Figures



# Glossary

AKS – Azure Container Services

AWS – Amazon Web Services

CDN – Content Delivery Network

CICD – Continuous Integration and Continuous Delivery

EFK – Elasticsearch Fluentd Kibana

ELK – Elasticsearch Logstash Kibana

GCP – Google Cloud Platform

IOT – Internet of Things

OAS – OpenAPI Specification

OSS – Open Source Stack

PaaS – Platform-as-a-Service

SOA – Service-Oriented Architecture

UWP – Universal Windows Platform

VM – Virtual Machine

# References