

COMP4906 HONOURS

THESIS: SKYDOT

FINAL REPORT

Lexi Brown - 100956208

TABLE OF CONTENTS

Summary	3
1 Introduction	6
1.1 Problem	6
1.2 Motivation	7
1.3 Project Goals.....	7
1.4 Project Objectives	8
1.5 Possible Features	8
1.6 Technology and Equipment Requirements	9
2 Background	10
2.1 Architecture.....	10
2.1.1 Microservices In Skydot.....	12
2.1.2 Layered.....	16
2.2 Technologies.....	17
2.2.1 Azure (AKS).....	17
2.2.2 Kubernetes.....	18
2.2.3 Docker.....	20
2.3 Alternatives.....	22
2.3.1 Alternative Technologies	22
2.3.2 Competition.....	23
3 Solution	27
3.1 Client.....	27
3.2 Skydot.....	27
3.2.1 Micro-Apps	28
3.2.2 Micro-Services.....	32
3.2.3 Host Gateway.....	34
3.3 Backend	35
3.4 DevOps.....	38
3.4.1 Dashboard.....	38
3.4.2 Logging.....	39
3.4.3 Monitoring.....	39

4 Results/Validation.....	41
4.1 Metrics	41
4.1.1 Memory Usage.....	41
4.1.2 CPU Usage	44
4.1.3 Latency	46
4.2 Cost.....	47
4.3 Development	52
4.3.1 Agile	52
4.3.2 Collaboration	53
4.3.4 Production	55
4.4 Comparison.....	56
5 Conclusion	58
5.1 Future Work	59
5.1.1 Multi-Cloud.....	59
5.1.2 Improvements	60
Additional Figures.....	67
Glossary	69
References	70

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:

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

1.1 Problem – An outline of the problems Skydot will be addressing.

1.2 Motivation – What has inspired and motivated me to build Skydot.

1.3 Project Goals – A list of all the goals Skydot aims to accomplish.

1.4 Project Objectives – Objectives set by this project to be completed in the implementation of Skydot.

1.5 Possible Features – A list of possible features that may be completed in the implementation of Skydot.

1.6 Technology and Equipment Requirements – A list of technology and equipment required to complete the project.

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

2.1 Architecture – A simple outline to the architecture used by Skydot. This includes microservice and layered architecture styles.

2.2 Technologies – A simple introduction to technologies utilized by Skydot.

2.3 Alternatives – An overview of Skydots design choices in comparison to alternative technologies that could have been used.

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

3.1 Client – An overview on the web and mobile client solutions.

3.2 Skydot – An in-depth look at how Skydot was built and the capabilities of the solution.

3.3 Back-end – An outline of how the back-end was put together to represent older and newer technologies.

3.4 DevOps – An overview of all completed features that pertain to development operations.

4. 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.

- 4.1 Metrics** – An detailed section on all metric measurements taken from analyzing Skydot.
- 4.2 Cost** – An in-depth look at how Skydot counters the cost of cloud and on-premise development.
- 4.3 Development** – An outline of how Skydot supports the development process.
- 4.4 Comparison** – A brief look at the metrics of alternative solutions in comparison to Skydot.
- 5. Conclusion** – A summary of all the aspects of the project.
 - 5.1 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 back-end services, databases and other resources via a REST translation layer. This way back-end 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 and Web applications.

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

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

1.4 Project Objectives

- i. Set up a Kubernetes application container that wraps the entire project and is used for deployments.
- ii. Utilize docker to wrap micro-apps and micro-services for deployment within Kubernetes.
- iii. Establish an API gateway in Kubernetes through which micro-apps can register and client applications can send requests.
- iv. Establish a service gateway through Kubernetes that allows micro-apps to make REST requests to micro-services.
- v. Establish a host gateway that provides a REST API for micro-services to access back-end services. The back-end must consist of REST and SOAP services.
- vi. Build an authentication database server to generate and keep tokens for client application requests.
- vii. Build back-end services for authentication, account information, transactions and bill payments. At least one WSDL service must be provided.
- viii. Provide micro-services in Java, Python, JavaScript and C++. Micro-services coverage: Authentication, Account summary and details, Transfers, Bill payment.
- ix. Build front-end application in Kotlin for Android and in ReactJS for web to display services.

1.5 Possible Features

- i. DevOps: Dashboard, health checking, logging, monitoring, continuous integration and continuous delivery (CICD).
- ii. Populate a string database with error and warning messages (en_CA and fr_CA).
- iii. Build an iPhone and/or tablet application.
- iv. Integrate oAuth2 and LDAP authentication.

- v. Use Swagger for design and documentation.

1.6 Technology and Equipment Requirements

- Microsoft Azure
- Android device(s)
- ReactJS
- Android Studio
- Kubernetes
- Docker
- Minikube
- Postman
- SoapUI
- GitHub

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.

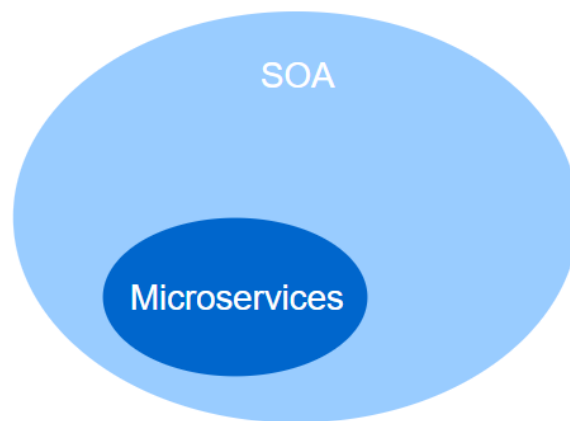


Figure 1 - Microservices vs. SOA

The alternative to microservice architecture is monolithic, n-tier and SOA (Figure 2 – Monolithic vs. SOA vs. Microservices). These traditional models, and the inherent 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 than 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.

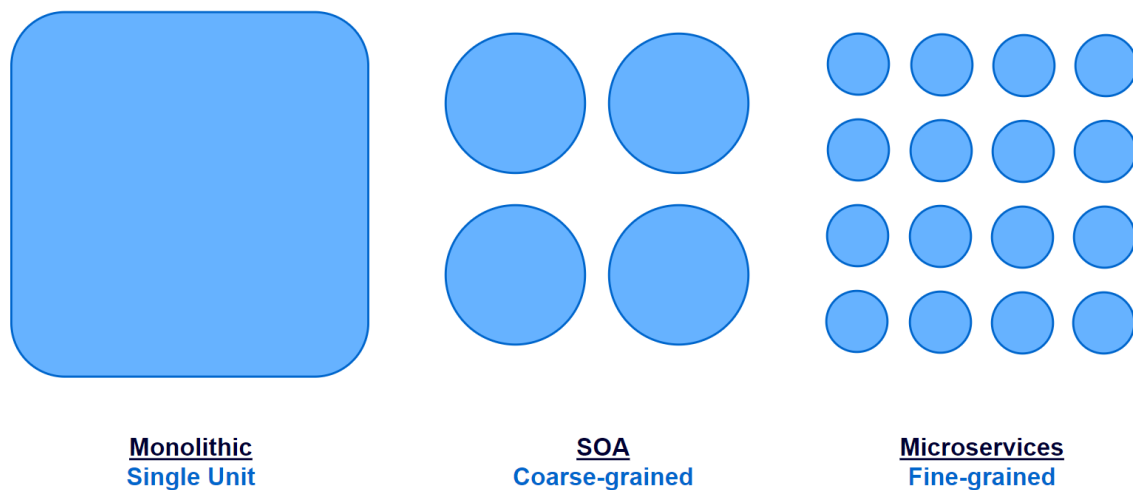


Figure 2 – Monolithic vs. SOA vs. Microservices

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 arise 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 Microservices In Skydot

Skydot utilizes a microservice architecture style (Figure 3 - Skydots Microservice Architecture). 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.

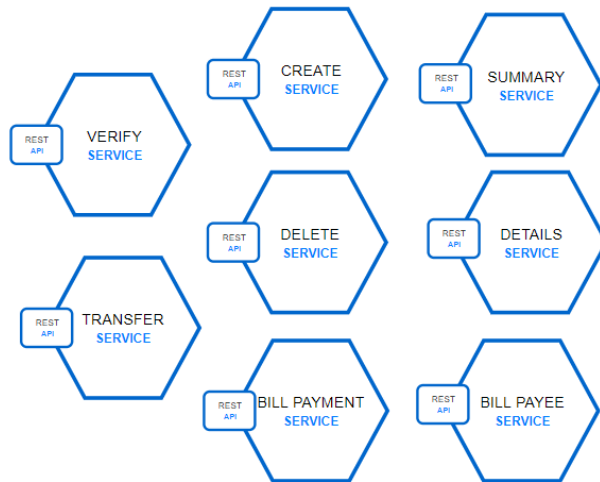


Figure 3 - Skydots Microservice Architecture

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

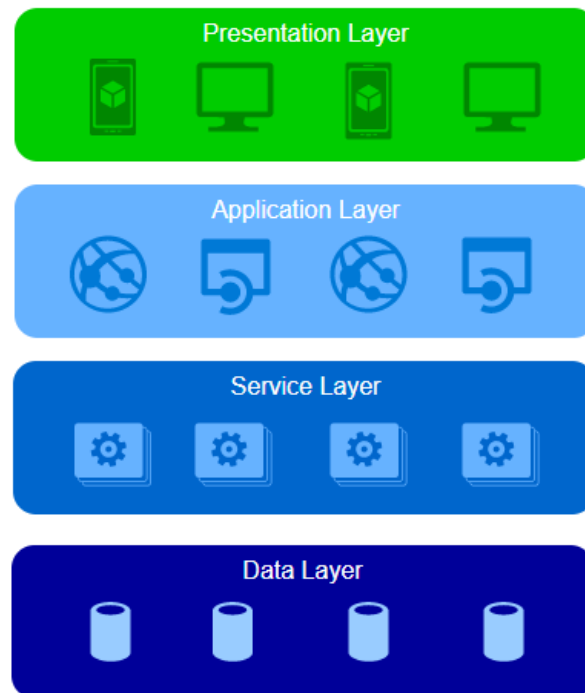


Figure 4 - Diagram of Skydots Layered Architecture

The entirety of Skydot is structured in a strict 4 tier layered style (Figure 4 - 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 remove the complexity of implementing, installing, maintaining and securing Kubernetes in Azure. To start up AKS in Azure requires the following simple flow (Figure 5 - Azure AKS Start Up Flow),

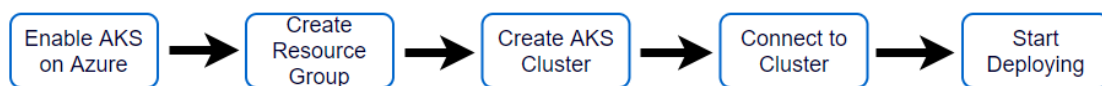


Figure 5 - Azure AKS Start Up Flow

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 (Figure 6 - Azure AKS Resources) so all cost metrics are using Microsoft Azures' pricing.

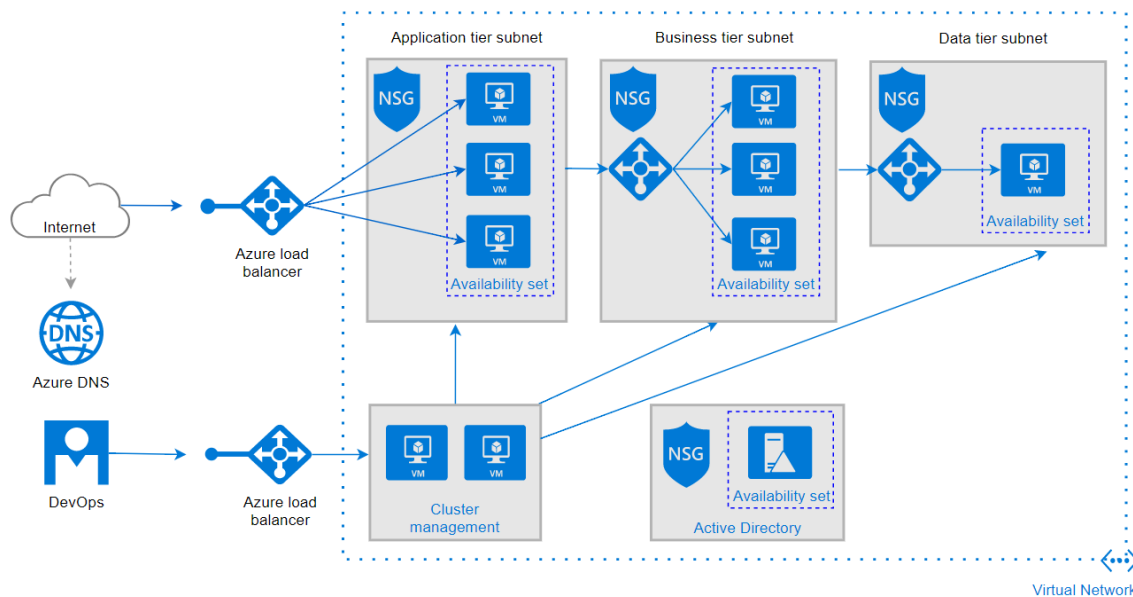


Figure 6 - Azure AKS Resources

It is important to note that Skydots design does not require Azure specifically. A different cloud provider could be used. However, for this display of Skydots capabilities, Microsoft Azure was the chosen provider.

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 and it 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 7 - 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, and the resources for applications within themselves.

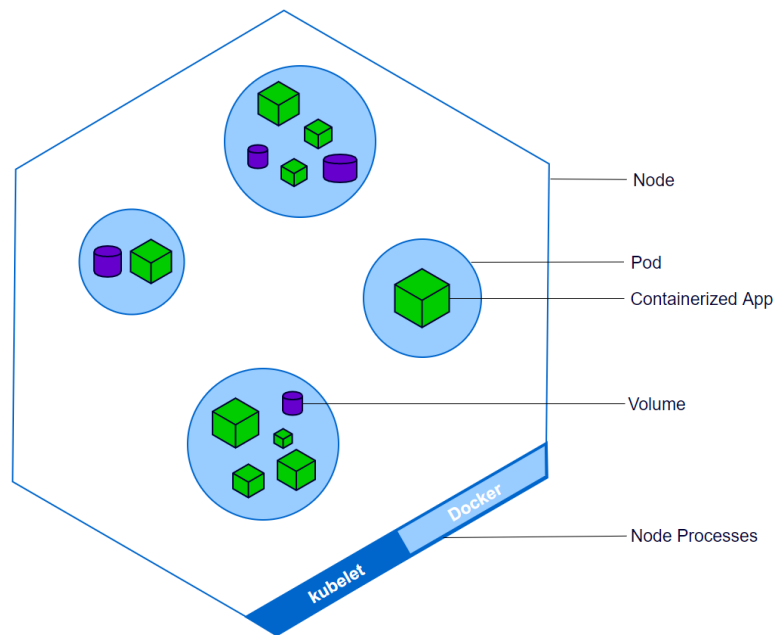


Figure 7 - Kubernetes Node

Within 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 port on the node it's within. Each pod can be scaled when managed by a deployment, which is specified in YAML of that deployment. Though, 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 Skydots 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 8 - Skydot In Kubernetes).

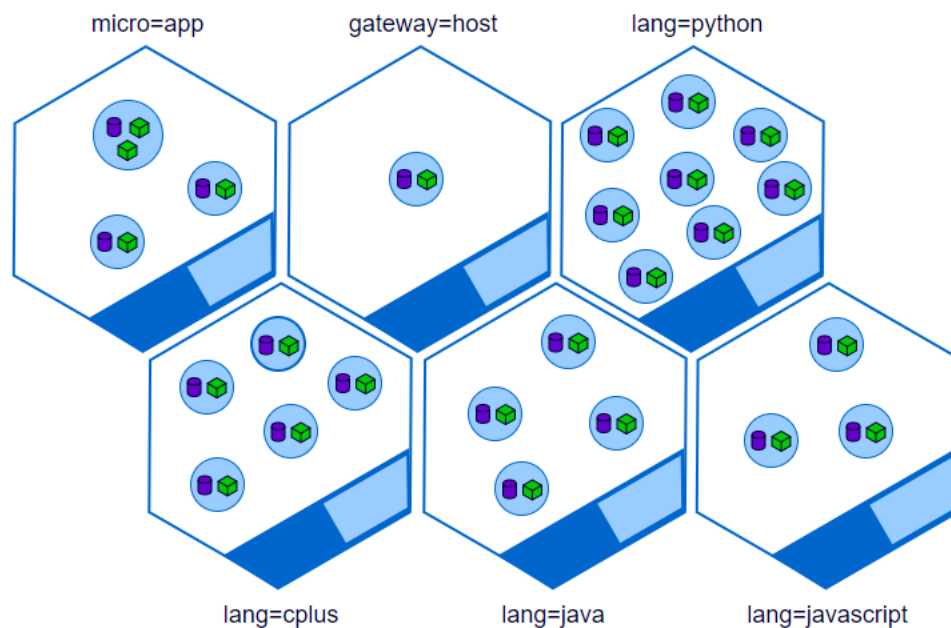


Figure 8 - Skydot In Kubernetes

2.2.3 Docker

Docker is a tool designed to make it easier 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 complete 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 environments 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 needed 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 (Figure 9 - Docker Process) 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.

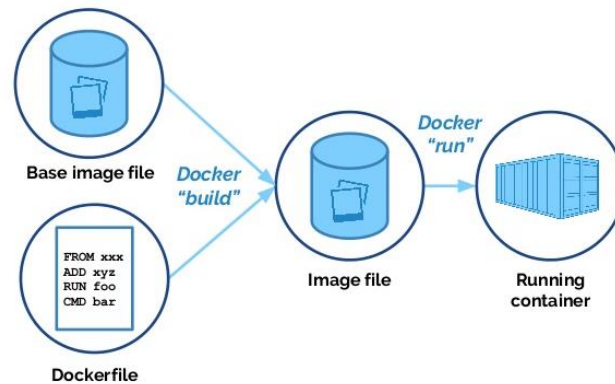


Figure 9 - Docker Process

In terms of Skydot, applications are built and dockerized locally. Then those images are pushed to Skydots 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

2.3.1 Alternative Technologies

There is a lot of technology Skydot could've utilized in this project. In Skydots demonstration build, it is using Kubernetes within Microsoft Azure. However, Skydot is not tied to one specific technology. It just provides a template as to how to set up an excellent system in the cloud. Although, the same set up could be put in on-premise technologies.

Here are some of the alternative technologies that a person implementing Skydot could use (Table 1 - Alternative Technologies).

Cloud/IaaS/PaaS	Containerization	Repository Management	SCM
Amazon Web Service (AWS)	Docker Swarm	Docker Hub	GitHub
Azure	rkt	npm	BitBucket
Google Cloud Platform (GCP)	Mesos	Artifactory	Gitlab
Rackspace	Kubernetes	Nexus	Subversion
OpenStack	Nomad	GitHub	Mercurial
Heroku	DC/OS	CoreOS	ISPW
OpenShift	Diego	pulp	Helix

Table 1 - Alternative Technologies

The two key technologies are the Cloud/IaaS/PaaS provider and the containerization orchestration platform. These are the foundation as everything else just builds on top of them. From there, there are too many additions and enhancements to be listed in this report. However, the main point is, Skydot allows the freedom of choosing the technologies a company wants to work with.

2.3.2 Competition

Although the chosen container orchestration system was Kubernetes running in Azure, there are other approaches to automation, service discovery and containerization. One approach is the use of an off-the-shelf platform-as-a-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 10 - 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.

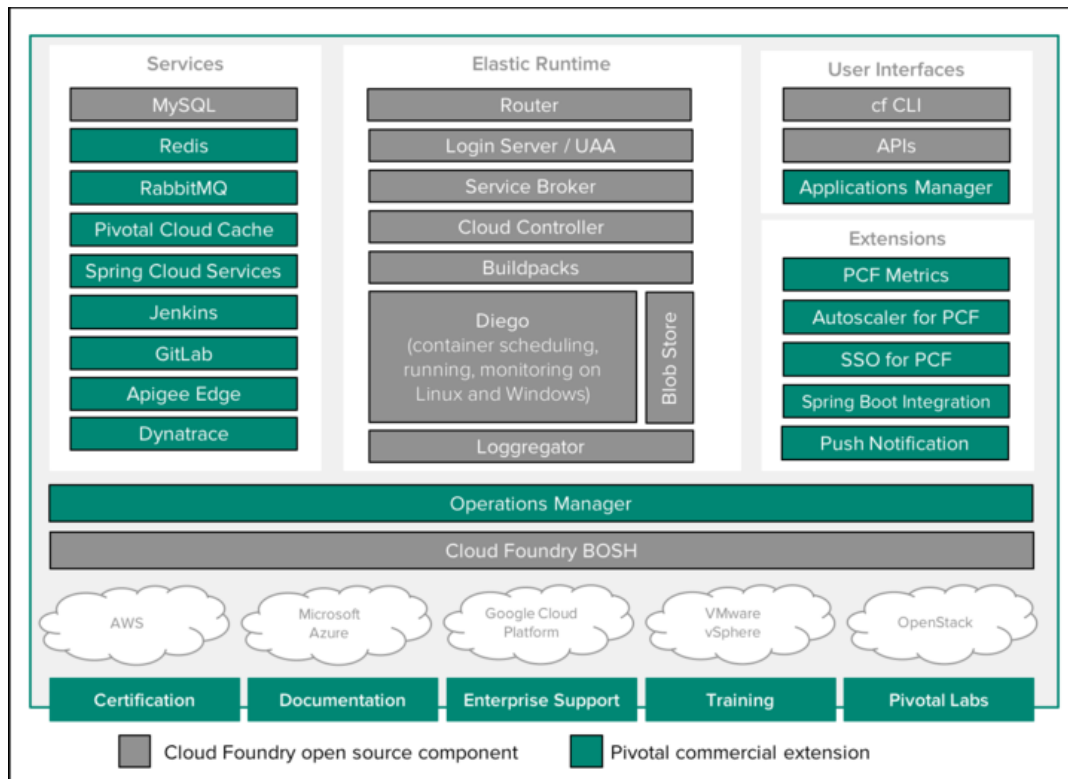


Figure 10 - Pivotal Cloud Foundry architecture – open source and enterprise

One of the main differences between a system like Cloud Foundry and Skydot is that with Cloud Foundry you can provide the information you know, and the platform will imply the rest. Which doesn't mean you can't be specific but that it allows you to not worry about how it runs your code in the cloud. While with Skydot, you provide exactly how the system should run your code in the cloud and it will not make assumptions or change anything without your say-so.

So, in no means is a system like Cloud Foundry a bad investment. It provides a lot more handholding but at a steeper price, and you lose a lot of flexibility in the technologies you can use but you gain maintenance support. These are some of the pros and cons of choosing an out-of-the-box tool. Whereas for Skydot, it is essentially a make-your-own PaaS which utilizes a clustering solution by employing container orchestration framework. Both solutions do the same thing, the difference is how much control you give up.

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.

Cloud-Native Application Service Models			
On-Premises	Infrastructure (as a service)	Platform (as a service)	Software (as a service)
Applications	Applications	Applications	Applications
Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware
O/S	O/S	O/S	O/S
Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking

■ You manage ■ Other manages

Figure 11 - Cloud-Native Service Models Comparison

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 11 - 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 Skydots 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 sections support the micro-apps from within Skydots private virtual network. Although all three layers can access the internet, if need be, only the host gateway can access private back-end 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 12 - 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 back-end services but differ in 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 verify if a token is valid by requesting verification from the auth micro-app.

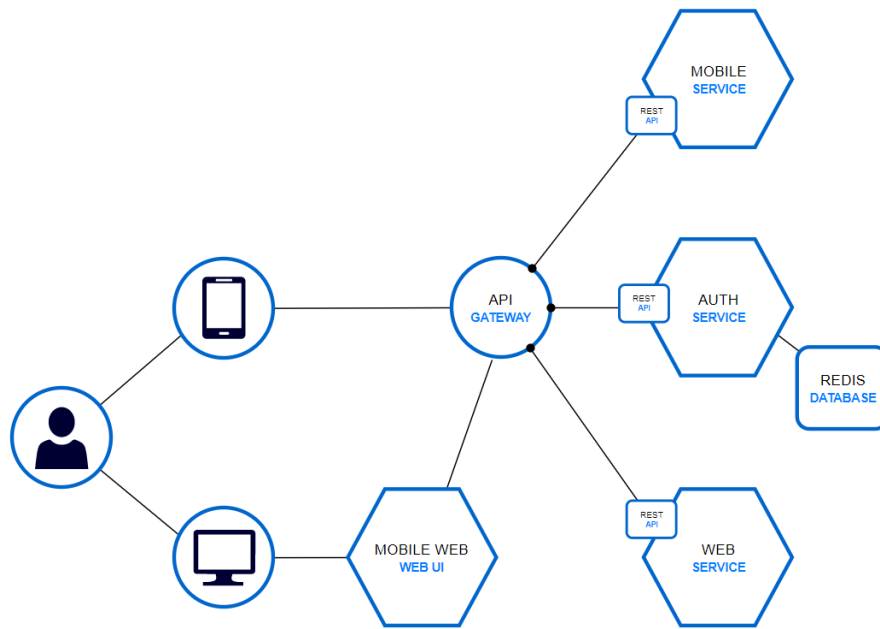


Figure 12 - Skydots Micro-apps

Since the micro-apps are running within the same cluster, they 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-apps (i.e., cluster IP, number of pods, node location, etc.) to make a request to them. If the auth micro-app validates the token, the user is valid and request can go through, provided that 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 2 - 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. They also 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 (Figure 13 - Timeout Flow). This ensures that all login tokens are handled by one application and that every client logs in the same way.

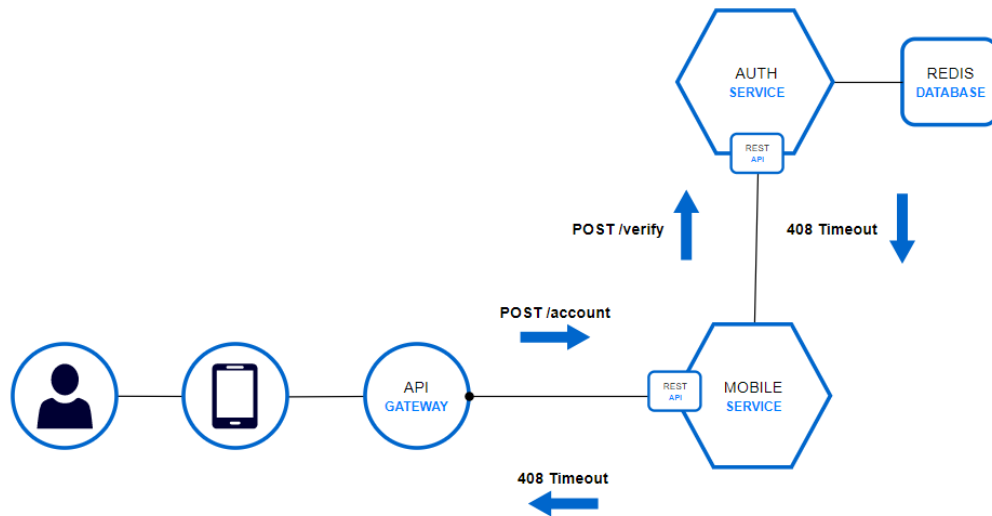


Figure 13 - Timeout Flow

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 (Table 3 - Mobile Micro-App Endpoints).

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 3 - 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 (Table 4 - Web Micro-App Endpoints).

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 4 - 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 DNSs 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

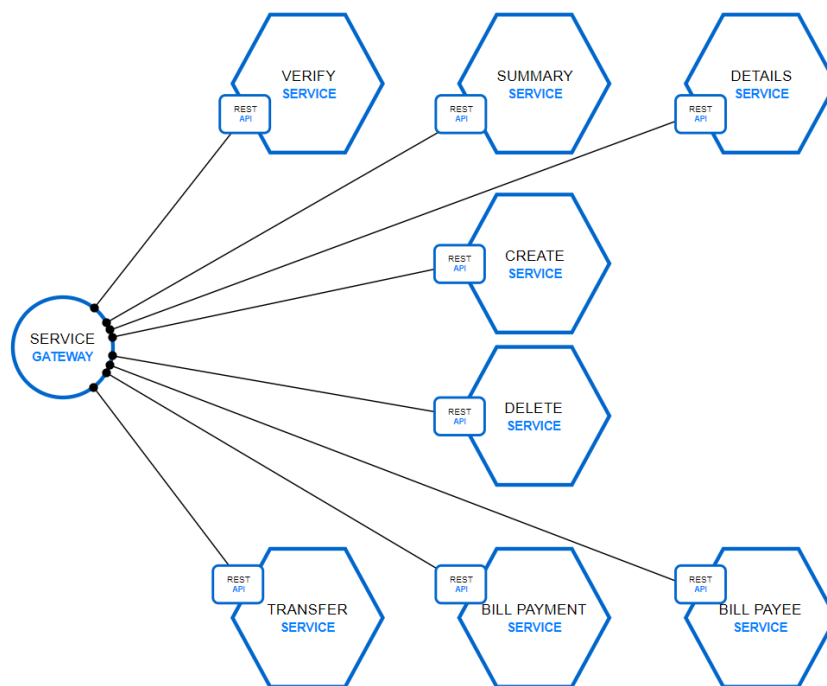


Figure 14 - Skydots Micro-Services

The micro-services are within Skydots service layer (Figure 14 - 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 5 - Banking Services):

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

Table 5 - 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. Thus, 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 6 - 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 6 - 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 15 - Skydots Host Gateway). It fronts back-end 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).

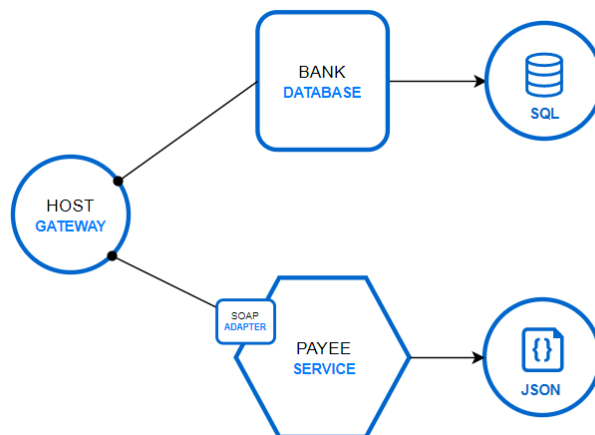
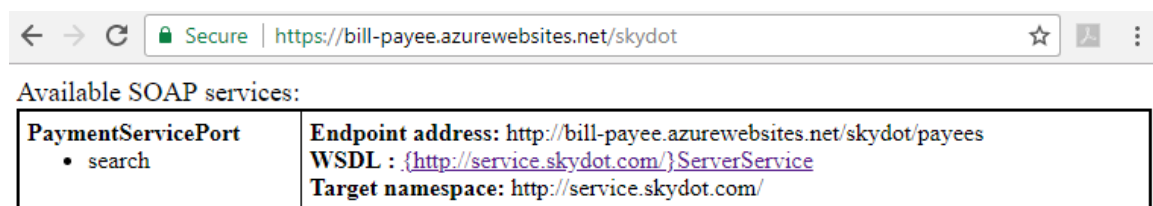


Figure 15 - Skydots Host Gateway

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.

3.3 Backend

As mentioned above in the host gateway section, there are two parts to the back-end. 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 16 - SOAP Service):



Available SOAP services:	
PaymentServicePort <ul style="list-style-type: none">• search	Endpoint address: http://bill-payee.azurewebsites.net/skydot/payees WSDL : http://service.skydot.com/ ServerService Target namespace: http://service.skydot.com/

Figure 16 - SOAP Service

And performing a search on the service returns an XML data response (Figure 17 - SOAP Service Response WSDL):

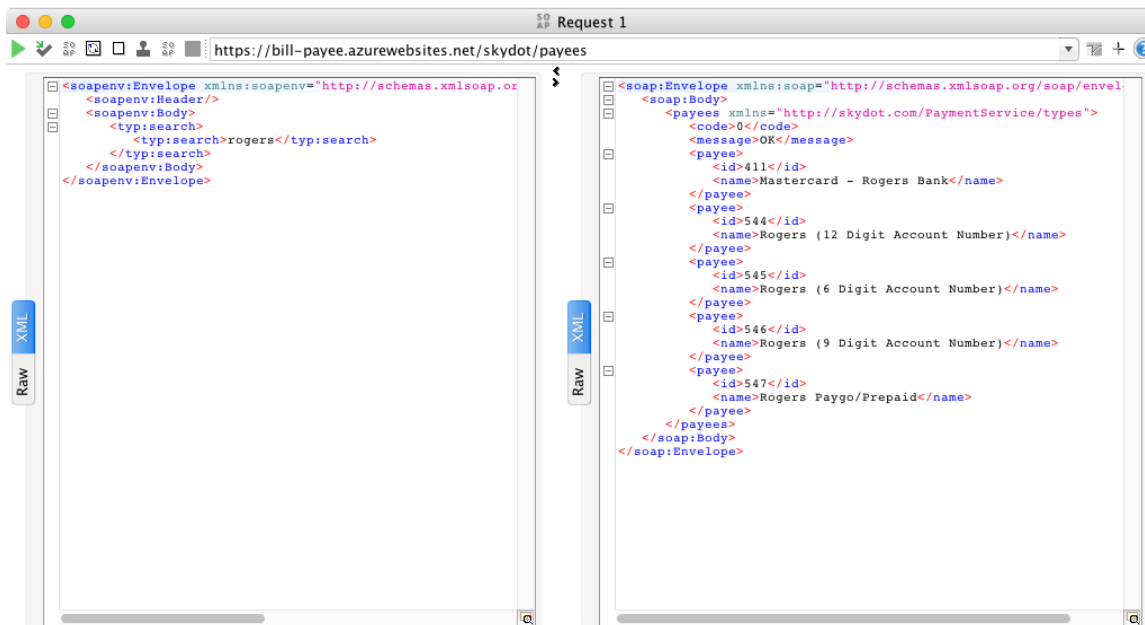


Figure 17 - SOAP Service Response WSDL

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 18 - SOAP Service Response JSON):

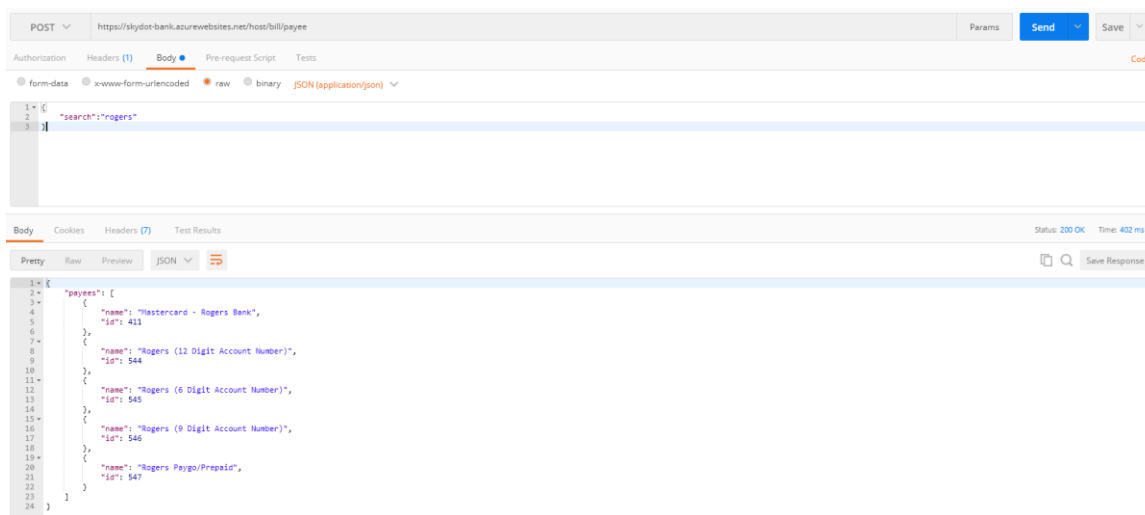


Figure 18 - SOAP Service Response JSON

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 is made with all information relevant to the transaction (i.e., date, amount, account id).

The entity relationship of the database is shown in Figure 19 - Database Entity Relationship. A profile must have at least one account, but an account doesn't have to have any transactions.

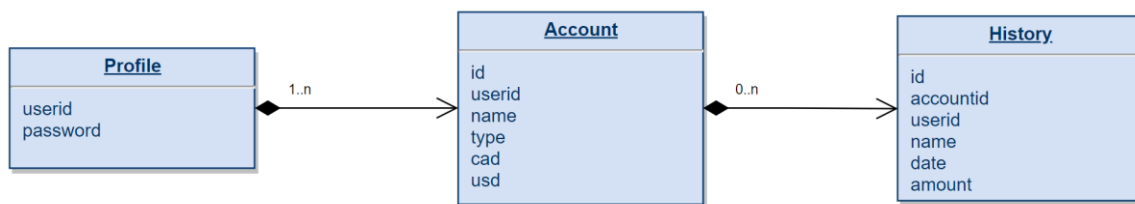


Figure 19 - Database Entity Relationship

3.4 DevOps

3.4.1 Dashboard

Kubernetes offers a web-based user interface (Figure 20 - 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 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.

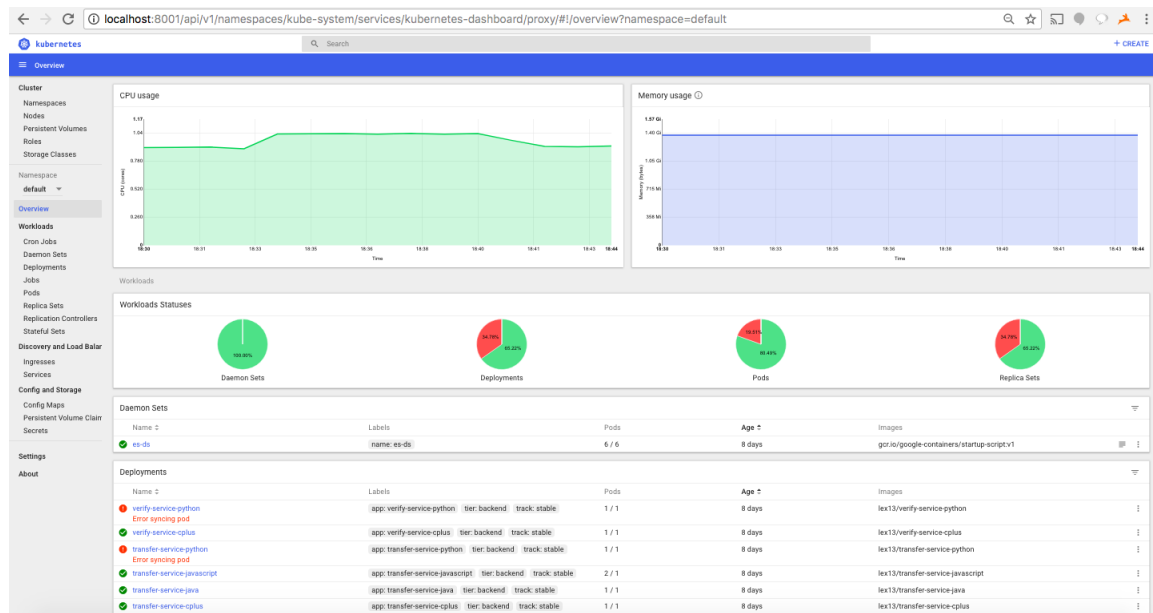


Figure 20 - Kubernetes Dashboard

3.4.2 Logging

The logging that was used is EFK style logging. EFK logging (Figure 21 - 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 21 - 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 systems applied to Skydot: InfluxDB and Prometheus (Figure 22 - 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 22 - 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

4.1 Metrics

Metrics in the initial research was taken from memory intense tests (Figure 34 - Memory Usage Comparison), from runtime speed tests (Figure 35 - Runtime Speed Comparison) and from CPU usage tests (Figure 36 - CPU Consumption on Different Operating System). The initial research done for this report led to believe that Java, and JavaScript, would always perform poorly in comparison to other languages in these categories. However, testing and analyzing Java in Skydots environment proved some of the research wrong. What the initial research lacked was consideration of a dockerized Java image environment as apposed to a full JVM environment. A docker environment is smaller and more resource efficient than a JVM running on a full virtual or physical machine. This also led to the consideration of bare-bones Java applications versus Java applications with an additional framework. The languages that were focused on were C++, Python, JavaScript and Java but after these considerations Spring, a Java framework, was added to the analysis for comparison. Spring is framework that provides a comprehensive programming and configuration model for modern Java-based enterprise applications on any kind of deployment platform. It is very popular in todays industry as it has a lot of desired functionality.

4.1.1 Memory Usage

Memory usages in cloud development is very important to monitor as it greatly contributes to the cost of maintaining the software. Thus, low memory consumption languages are more beneficial in the cloud as they use less resources. Initial research predicted Java would use a lot of memory both initially and when preforming tasks while other languages would perform a lot more efficiently.

Research Predictions		Actual Results
Language	Memory Usage	
Java	High	High
Spring (Java)	High	Highest
C++	Low	Low
Python	Low	Medium
JavaScript	Medium	Medium

Table 7 - Memory Usage Research vs. Actual

These research predictions proved mostly correct (Table 7 - Memory Usage Research vs. Actual). Although Java did perform poorly compared to C++, Python and JavaScript, Spring clearly performed the worst.

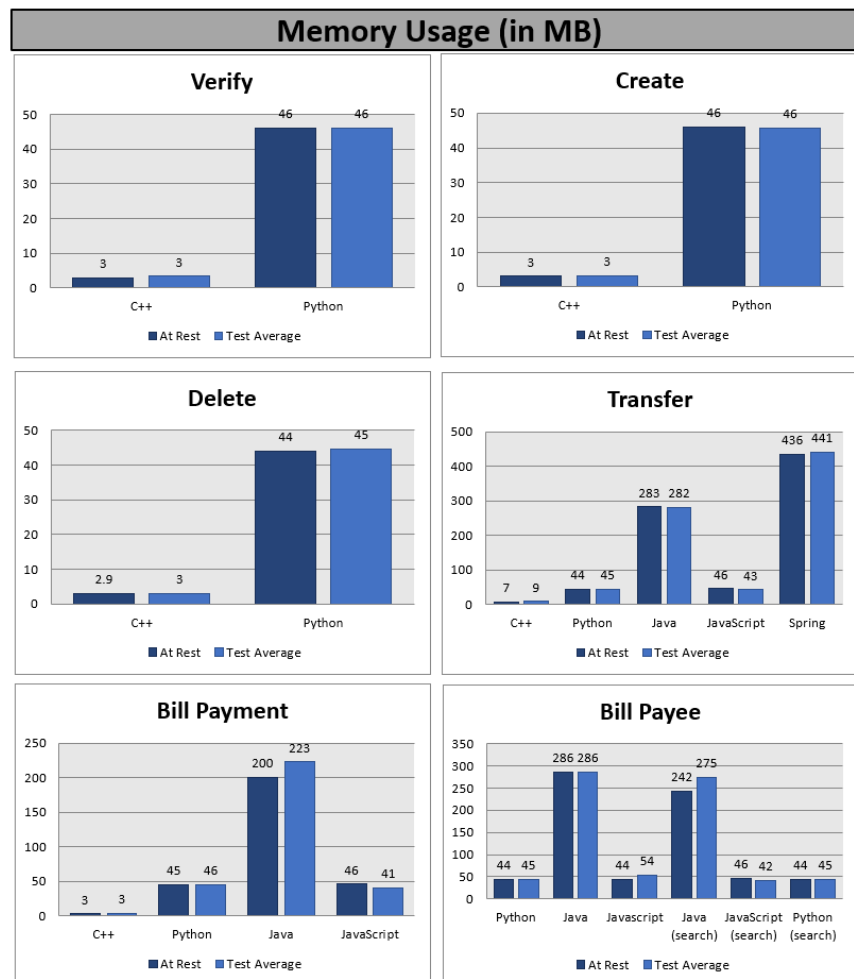


Figure 23 - Micro-Service Memory Usage (in MB)

When looking at all services memory usages (Figure 23 - Micro-Service Memory Usage (in MB)), C++ is undoubtedly the most efficient, Python and JavaScript are both fairly efficient, and Java and Spring are the least efficient. Although this numbers are small (in MB), you must consider what happens when the services are scaled. Consider the situation where 1 million clients access a service at the exact same time. If, for example, every 10,000 clients the service is scaled up and adds one additional deployment, that means the service will be scaled 100 times. Let's apply this situation to the transfer service, as it covers all languages (Table 8 - Micro-Services Scaled Memory Usage).

Language	At Rest	One instance running	One hundred instances running
C++	7 MB	9 MB	900 MB
Python	44 MB	45 MB	4.5 GB
JavaScript	46 MB	43 MB	4.3 GB
Java	283 MB	282 MB	28.2 GB
Spring	436 MB	441 MB	44.1 GB

Table 8 - Micro-Services Scaled Memory Usage

Once services begin to be scaled the memory usage is more apparent. Scaled Spring is 49 times more expensive than C++ and this is only considering one service. There is a total of eight services currently deployed in Skydot. Imagine each service developed in Spring scaled one hundred times, the total memory usage would be 352.8 GB. Whereas, having each service developed in C++ scaled one hundred times would only be 7.2GB. Therefore, it would be much more efficient to scale eight different services in C++ one hundred times than to scale just one service in Spring one hundred times.

It's important to remember that the C++, Python, JavaScript, Java and Spring Transfer micro-services are all performing the same task and are built to be as efficient as possible. However, there is a clear divide in memory efficiency between the languages. This is not ideal for cloud development as the cloud host being utilized, in this case Microsoft Azure,

charges based on how much memory is being used. It can be noted that Java and Spring have many features and can do a lot more compared to other languages, however, for simplistic microservices, using Java and Spring would burn through project funding and cloud recourses.

4.1.2 CPU Usage

CPU usages in cloud developer is very important as it determines how many cores are needed on a VM to run the application. Thus, the more CPU usage per application, the more cores needed to maintain that application. Initial research predicted that Java would be a high CPU usage language while other languages would be a low CPU usage languages.

Research Predictions		Actual Results
Language	CPU Usage	
Java	High	Low
Spring (Java)	High	Low
C++	Low	Low
Python	Low	Low
JavaScript	Low	Low

Table 9 - CPU Usage Research vs. Actual

These research predictions proved false for Java and JavaScript (Table 9 - CPU Usage Research vs. Actual). It seems all languages were efficient with CPU usage.

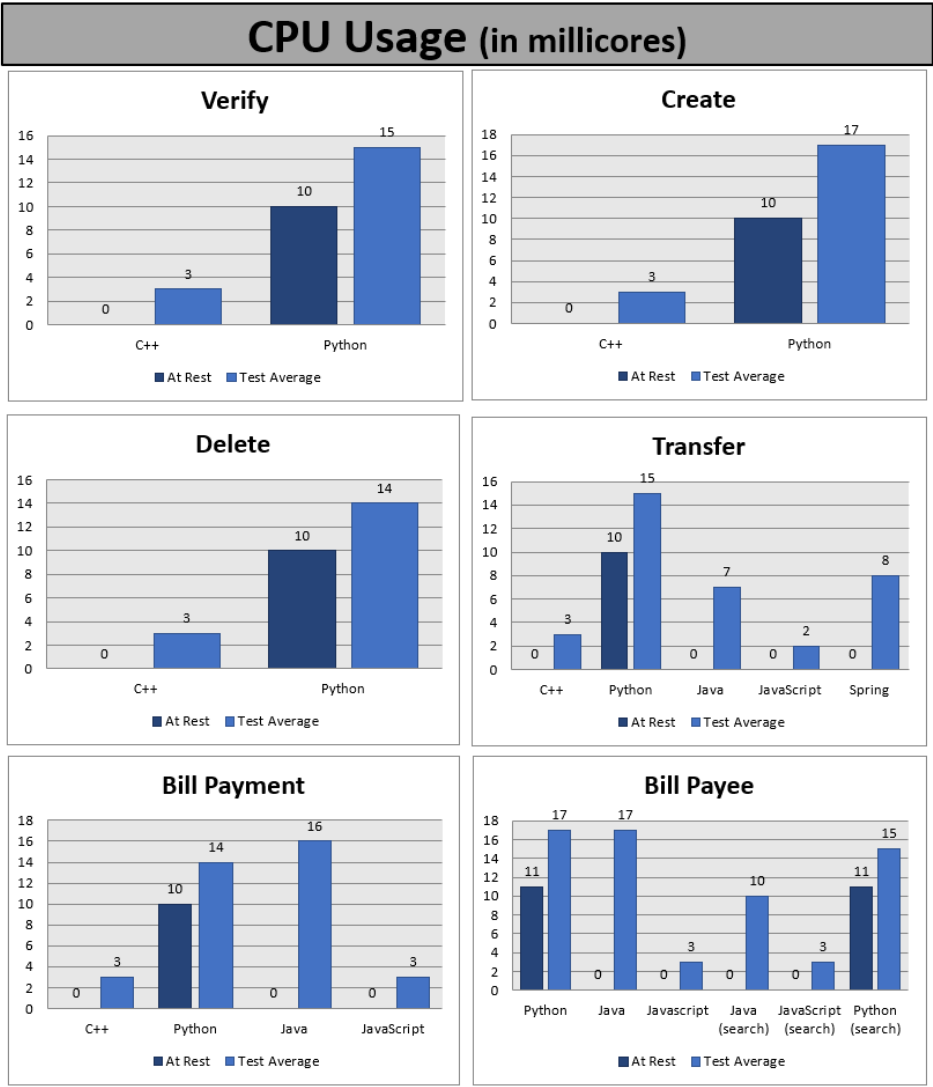


Figure 24 - Micro-Service CPU Usage (in millicores)

When looking at all services CPU usages (Figure 24 - Micro-Service CPU Usage (in millicores)Figure 25 - Micro-Service Latency (in ms)), all usages are low. Kubernetes has a metric called Millicores that is used to measure CPU usage. CPU is specified in units of cores and one CPU core is split into 1000 units. Thus, 100 millicores translates to a tenth (10%) of a single core. The highest millicore usage between all services and all languages is 17, which is 1.7% of a core. Since these numbers are so low, the range being 0.3% to 1.7% CPU usage, it would not be reasonable to state that one outperforms another. Therefore, all languages can be said to have handled CPU usage well.

4.1.3 Latency

Latency is always an important metric to monitor as it greatly contributes to user satisfaction. Slow services are less desired especially in a microservice architecture since the micro-service performs a single task. The task can be complex however the service should do one thing and do that one thing well. Initial research predicted that Java and JavaScript would perform slower than C++ and Python, however that was not the case.

Research Predictions		Actual Results
Language	Runtime Speed	
Java	Slow	Fast
Spring (Java)	Slow	Slowest
C++	Fast	Fast
Python	Medium	Medium
JavaScript	Slow	Fast

Table 10 - Runtime Speed Research vs. Actual

These research predictions proved false for both Java and JavaScript (Table 10 - Runtime Speed Research vs. Actual). Both languages were high performing and even competed with C++ in terms of speed. Although Java did perform well, Spring clearly performed the worst with an average 2 second delay.

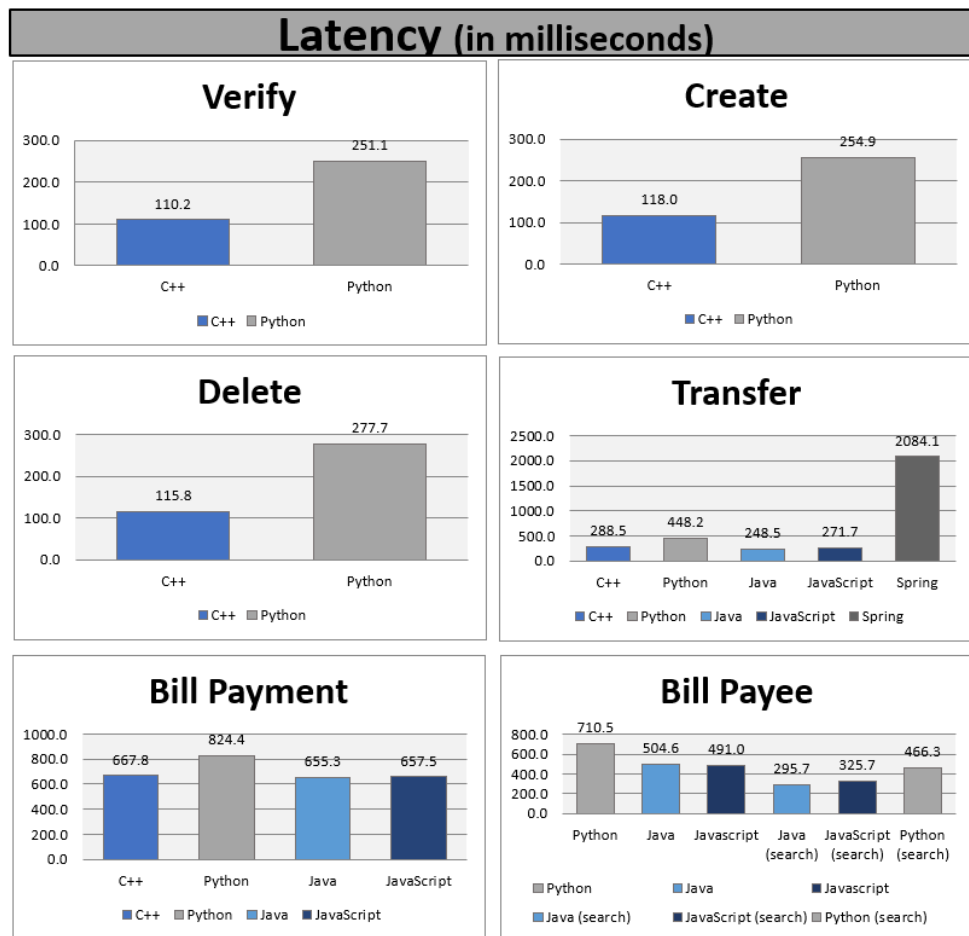


Figure 25 - Micro-Service Latency (in ms)

When looking at all services latencies (Figure 25 - Micro-Service Latency (in ms)), all languages performed well except Spring. Low latency is important for a positive client experience and Spring is clearly struggling in this type of environment.

4.2 Cost

Cost is an extremely important aspect of this project. It has been mentioned many times in this report how the way resources are handled in the cloud affects how much running the system will cost. The more resources an application uses in a cloud environment, the costlier running it becomes. Using languages and frameworks that allow less memory usage

consumption, for example, opens more virtual machine options. As this project is currently deployed in Microsoft Azure, all pricing numbers will be using Azures billing costs.

In Azure, there are many different virtual machine (VM) instances that vary in cost depending on how many resources are allocated to that VM. Each instance has a number of cores, an amount of RAM, in GB, and an amount of temporary storage, in GB. The combination of these three recourses determines the cost per hour of running the virtual machine. In the following example, the number of cores will be set at 16 and the amount of temporary storage will be set at 800 GB (Table 11 - Azure Node Instances Cost). The resource that is being focused on is RAM, as in how much memory an application will use on the VM.

Node Instances	Cores	RAM (GB)	Temporary Storage (GB)	Virtual Machines	Cost/hour (USD)	Cost/Month (USD)	Cost/Year (USD)
D14	16	112	800	100	\$1.542	\$112,566.00	\$1,350,792.00
D5 v2	16	56	800	100	\$1.170	\$85,410.00	\$1,024,920.00
							\$325,872.00

Table 11 - Azure Node Instances Cost

When a less memory consuming language is used, you will know that less RAM is required to run it. A language like C++ doesn't use a lot of memory when it's running and therefore will not need a lot of RAM while a language like Java, which consumes much more than C++, will require more RAM. Looking at the chart above, choosing the VM, D5 v2, with less RAM saves \$325,872.00 per year, if 100 virtual machines were purchased. The only difference between the two instances is the RAM and going with a language like C++ allows you to choose the lower RAM of the two. Therefore, the difference in savings is centralized around the language that is chosen.

We can also consider the cost of working in a cloud environment versus on-premise technologies. Using Microsoft Azures Total Cost of Ownership (TCO) Calculator, we can estimate how much it would cost to run 100 physical servers on-premise versus 100 virtual

machines in the cloud, both over a three-year time span. First, let's compare the costs of running 100 D14 virtual machines in Azure to 100 physical servers on-premise with similar specifications (Table 12 - On-Premises vs. D14 Azure Costs). Note that there are many assumptions that the calculator makes including: electricity costs (\$0.10 per kWh), hourly rate of IT administrator (\$50/hr), cost per GB for storage (\$2/GB) and much more. The underlying calculations in this assessment tool have been reviewed and tested by Nucleus Research for accuracy and transparency.

On-Premises		Azure
2 processor		D14 Standard
10 Core		16 Core
128 GB RAM		112 GB RAM
Linux		Linux
Local Disk/SAN-HDD (RAID 10 config)		Page Blob Storage – LSR (RAID 10 config)
Computation Costs		
Hardware	\$2,260,000	\$1,880,800
Software	\$0	
Electricity	\$171,424	
Total	\$2,431,424	
Data Center Costs		
	\$470,646	\$0
Networking Costs		
	\$678,036	\$0
Storage Cost		
	\$2,662	\$921
IT Labor Costs		
	\$78,398	\$58,581
Total Costs		
	\$3,661,166	\$1,940,302

Table 12 - On-Premises vs. D14 Azure Costs

Savings over 3 years in each category is shown in Table 13 - On-Premises vs. D14 Azure Savings and the cost over 3 years in shown graphically in Figure 26 - Total On-premises vs D14 Azure Cost Over Time.

Estimated cost savings over 3 years by category	
Compute	\$550,624
Data Center	\$470,646
Networking	\$678,036
Storage	\$1,741
IT Labor	\$19,818
Estimated savings are \$ 1,720,865 (47%) over 3 years with Microsoft Azure	

Table 13 - On-Premises vs. D14 Azure Savings

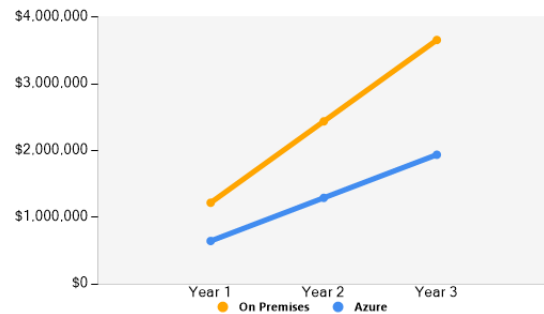


Figure 26 - Total On-premises vs D14 Azure Cost Over Time

Now, lets compare the costs of running 100 D5v2 virtual machines in Azure to 100 physical servers on-premise with similar specifications (Table 14 - On-Premises vs. D5v2 Azure Costs).

On-Premises		Azure
2 processor		D5v2 Standard
8 Core		16 Core
64 GB RAM		56 GB RAM
Linux		Linux
Local Disk/SAN-HDD (RAID 10 config)		Page Blob Storage – LSR (RAID 10 config)
Computation Costs		
Hardware	\$2,256,800	\$1,013,700
Software	\$0	
Electricity	\$179,308	
Total	\$2,436,108	
Data Center Costs		
\$470,646		\$0

Networking Costs	
\$677,076	\$0
Storage Cost	
\$2,662	\$921
IT Labor Costs	
\$78,398	\$58,581
Total Costs	
\$3,664,890	\$1,073,202

Table 14 - On-Premises vs. D5v2 Azure Costs

Savings over 3 years in each category is shown in Table 15 - On-Premises vs. D5v2 Azure

Savings Table 13 - On-Premises vs. D14 Azure Savings

and the cost over 3 years is shown graphically in Figure 27 - Total On-premises vs D5v2 Azure Cost Over Time.

Estimated cost savings over 3 years by category	
Compute	\$1,422,408
Data Center	\$470,646
Networking	\$677,076
Storage	\$1,741
IT Labor	\$19,818
Estimated savings are \$ 2,591,689 (71%) over 3 years with Microsoft Azure	

Table 15 - On-Premises vs. D5v2 Azure Savings

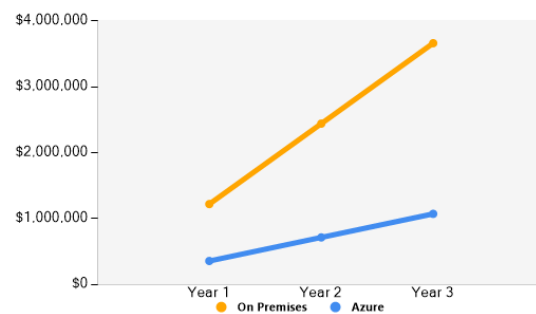


Figure 27 - Total On-premises vs D5v2 Azure Cost Over Time

This shows that it's more cost efficient to go with a cloud provider, in this case Microsoft Azure, than an on-premise solution. In conclusion, switching from an on-premise solution to a cloud solution and choosing a language that allows the use of less expensive virtual machines all contributes to lowering the costs of maintaining your software.

4.3 Development

4.3.1 Agile

Agile development and Skydot easily go hand in hand. In the today's industry, agile is gaining a lot of popularity as the how-to in software delivery over the older methodology Waterfall. Agile is a time boxed, iterative approach to software delivery that builds software incrementally from the start of the project, instead of trying to deliver it all at once near the end. It works by breaking project down into little bits of user functionality called user stories, prioritizing them, and then continuously delivering them in short usually two-week cycles (as an example) called iterations. This breakdown of functionality is key to microservices. You would organize your user stories to each type of functionality a user can do (i.e., Account functionality, Bill payment functionality, etc.). Each micro-service takes a section of user stories that pertain to a single type of functionality, unless it can be further divided, and that micro-service will handle it. Meanwhile, another group of user stories would be applied to different micro-services. You then add on to each micro-service, which are relatively small since they only pertain to one set of functionalities, incrementally. Each developer or development team doesn't have to worry about impeding or conflicting with each other as the micro-services are independent from one another. One team or developer could write their service in Go and the other team or developer could write their service in C++ and nothing conflicts. Even with user stories that enforce interaction between micro-services that are written in different languages, there won't have to be language conversion stories.

Another aspect of agile is testing and the ability to test each user story. Since each micro-service covers a cohesive set of functionalities and is independent of other technologies outside of the module, you can have modular testing and easily test each user story. The most important aspect of the agile process to remember is that the word 'agile'

should not be treated as a noun, but rather as an adjective – your project should be agile and not slowed down by the agile process.

Testing

The flexibility of Skydot allows for different ways of testing inside and outside of the micro-service. In agile, each user story should be able to be tested, all positive and negative paths. All micro-services can be unit tested easily since each one is essentially a single unit and is independent of other technologies outside of it. Since each service is a black-box container, testing doesn't have to pertain to the language the service is developed in. It is only required to send requests to the service and verify the responses. As for testing within the service, there are many testing frameworks that can be utilized and a lot of them cover multiple languages. Some of the testing tools include: Selenium, JUnit, TestNG, Cucumber, Karma, Qunit and more.

Being able to test services individually speeds up deployment time as a fully tested micro-service can be pushed to production right away without being delayed by other micro-services that haven't been fully tested or have failed tests. In a monolithic system, if some tests fail for one service then the entire system must be held back until the issue is fixed.

4.3.2 Collaboration

Skydot promotes collaboration between teams in multiple ways. One way that it does this is by having common micro-services. Common micro-services allow micro-apps to employ the services they require without the need of producing duplicate code. For example, the Account Summary service is used by both the mobile and web micro-apps (Figure 28 - Common Account Summary Service). The Account Summary service does all the business logic that pertains to an account summarization, and the micro-apps only need to write transforming code that formats, localizes and trims the response from the micro-

service to meet the needs of the micro-apps client. Typically, the mobile and web teams in a company would develop their own separate servers with the business logic for account summarization but Skydots methodology removes that step and increases code integrity with common access points.

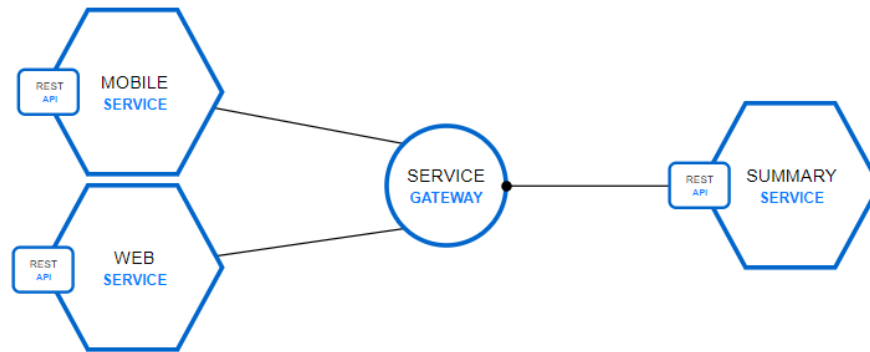


Figure 28 - Common Account Summary Service

This collaboration starts to shape full stack developers, or at least gives developers an understanding of the complete stack. A full stack developer works on the server side of the software development but also fluently speaks the front-end languages. This kind of developer can assist in the development of both the front and back ends whereas front-end and back-end developers split those responsibilities. Typically, in a development team there are front-end teams and back-end teams. These teams only interact in terms of how the client (front-end) and the server (back-end) interact. However, front-end developers typically don't assist in back-end developer and vice versa. With Skydots flexibility in development language, front-end developers can assist or become back-end developers since any developer can build a micro-service. They can utilize languages that they are most familiar with, thus eliminating the learning curve of using other back-end technologies. Having developers be flexible in where they can contribute in the development cycle is always beneficial as those developers become more valuable to the team and for future projects.

4.3.4 Production

Skydot also greatly benefits the production cycle of software by limiting the impact of change. Updates and additions become less involved and risky.

When updating an on-premise system, there can be a lot of circuit breaking involved. While one server goes down for updating, the others must be handle the updating servers' client traffic. Once the server has finished updating, it is restarted, and the same updating process starts on a different server until all servers are up to date. This process is usually manual, and users can experience latency problems while the system is update, especially when there aren't many on-premise servers. The less servers, the more strained a server becomes having to handle the traffic of another. In Skydot, deployments are updated automatically and without affecting other services. When, for example, the Transfer deployment needs to be updated, the process is as simple as setting the image within the container to the updated image on Docker Hub. The following command is an example of how to update the image,

```
kubectrl set image deployment/<app> container=<repository>/<app>:<version>
```

Once the image has been set, Kubernetes automatically does a rolling update. This means, if there are five transfer pods currently in service, each one-by-one get terminated while a new pod instance, with the updated image, gets spun up to replace it. This process continues until all pods are replaced. With this, the responsibility of circuit breaking and updating is handled by the orchestration software which is less prone to errors or mistakes. Also, since Skydot is a microservice architecture, when the Transfer service get updated, none of the other micro-services are affected by it. Those other service are independent and will continue to service clients without being hindered by the update.

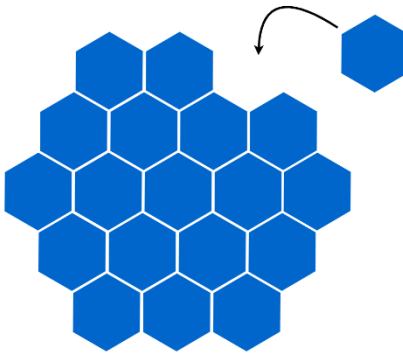


Figure 29 - Honeycomb Structure

Furthermore, Skydot limits the impact of change that additions make (Figure 29 - Honeycomb Structure). In an on-premise, monolithic environment when a new service needs to be added to the software, the entire system must pause to accommodate the addition, similar to when the software is being updated. Also, the addition requires all aspects of the software to be retested as there is a possibility that the add-on has affected another part of the system. Once the new software is thoroughly tested then the same monolithic circuit breaking pattern is followed. In comparison, when a new service, or application, is added to Skydot it doesn't affect other services, or applications, as they are all independent of each other. This way other services aren't delayed and don't need to be regression tested when the addition is made. And if the new services unexpectedly crashes or fails that issue doesn't propagate throughout the entire system.

4.4 Comparison

One technology that can be compare to Skydot is Pivotal Cloud Foundry (PCF). PCF is an open source PaaS and the enterprise version of Cloud Foundry. PCF can be deployed on top of any cloud providers like Azure, AWS and Google Cloud Platform. It is a cloud-native platform for deploying and operating applications. A lot of PCF utilizes Spring and, as shown in previous sections, Spring is a very heavy resource.

From working with PCF, and even stated on their website, a Java application needs one GB to run in PCFs environment. The primary reason is because Java applications are deployed with the app server that is bundled with it during the buildpack deploy process. This means no matter how efficient your code is, one GB will be allocated for the service, best case scenario. Just looking at the heaviest memory usage currently deployed in Skydot (Figure 23 - Micro-Service Memory Usage (in MB)), which is the Transfer service running in Spring, the memory footprint is half of a GB (Table 16 - Skydot vs. PCF Java Memory Usage).

Language	Skydot (Transfer service)	Pivotal Cloud Foundry
Java	~300 MB	1 GB
Spring	~450 MB	1 GB

Table 16 - Skydot vs. PCF Java Memory Usage

Although, Java, and even more so Spring, is memory heavy, at least in Skydots environment the developer can control the memory usage of the application. And thus, control how much running the application costs. As stated in the section on alternative solutions (2.3.2 Competition), in no means is a system like Pivotal Cloud Foundry a bad investment. It provides a lot more handholding and you gain maintenance support but at a steeper price, and you lose a lot of flexibility in the technologies you can use. These are the pros and cons of choosing this tool and both solutions, PCF and Skydot, do the same thing, but the difference is how much control you give up.

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 have been 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 effective 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, though research and validation showed that the latter option yielded more benefits without hindering performance and development, both individual and as a team. Skydot was shown to be beneficial to the development process, both in teams and in production. And it has displayed aspects that overshadowed alternative technologies and solutions.

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 flexibility of Skydot, it would be possible to utilize both Microsoft Azure and Amazon Web Services (AWS) (Figure 30 - Possible future Skydot usage with AWS).

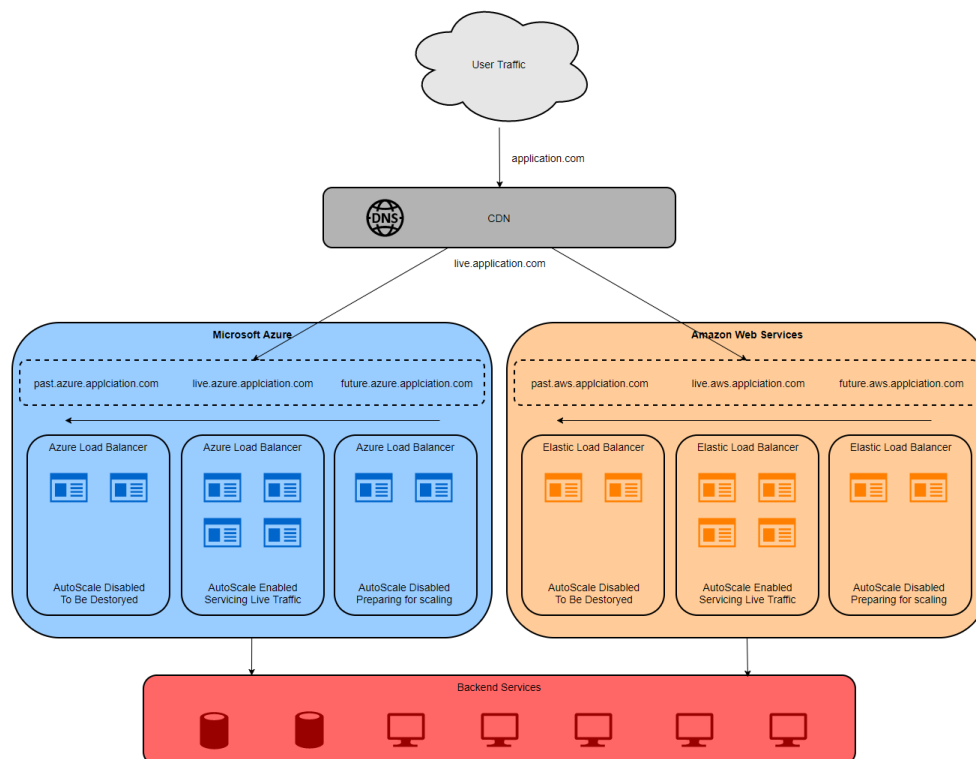


Figure 30 - Possible future Skydot usage with AWS

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 17 - CICD Technologies).















Issue Tracking	Version Control	Continuous Integration	Continuous Delivery	Artifact Repository
 JIRA	 Bitbucket	 Bamboo		 amazon S3 web services™
 slack	 mercurial	 TC	 Octopus Deploy	 JFrog Artifactory
 servicenow	 GitHub Enterprise	 Jenkins	 urban {code}	 Sonatype Nexus

Table 17 - CICD Technologies

In a CICD workflow, there is a lot of verification and validation of code before it's allowed to go into production (Figure 31 - CICD Workflow). Developers can use local or virtual workstations to complete backlogged development tasks. However, before the code is committed to the repository, it is built and verified through the CI pipeline. The branch verification process runs unit tests, performs static code analysis for code quality, UI-specific tests, and calculates test coverage. If the tests pass, that build is deployed to the development/test environment to run automated tests as well as integration tests. Any issues that are discovered are added to the backlog. From there, pull requests to the master repository must pass through the master CI pipeline, and then can be reviewed and approved by the project architect or technical lead. The master verification process runs all the same tests as in the branch but includes security scanning and other long-running automated integration tests. Once these tests pass, the build gets stored in an artifact repository for release management. That artifact can be deployed to cloned environments for each stage of the release process (i.e., UAT, Load testing, Staging). However, testing in either the staging or UAT environment is vital as approved builds which have been tested in one of these environments using production data can be deployed to production automatically or manually; either as a DNS switch from staging, or UAT, for quick rollback, or through the same deployment process as testing. The entire process ensures proper integration and deployment within a system. There are many versions of the process that can be used, this is just one potential implementation.

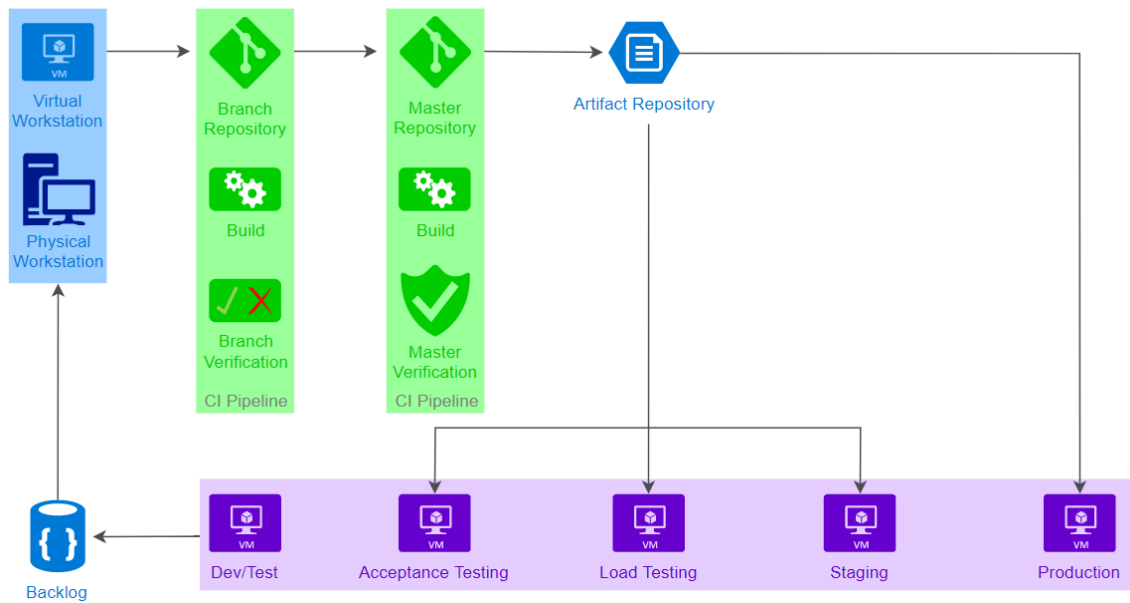


Figure 31 - CICD Workflow

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 32 - Error Message Flow),

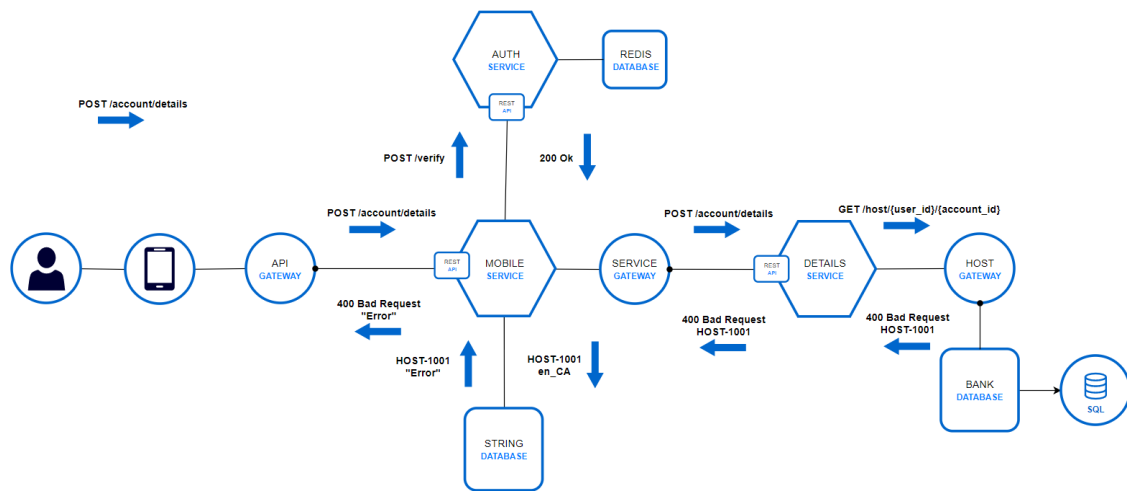


Figure 32 - Error Message Flow

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 logic to perform 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 strings. 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 Skydots 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 33 - OAuth 2.0 Protocol Flow). OAuth 2.0 is the industry standard protocol for authentication.

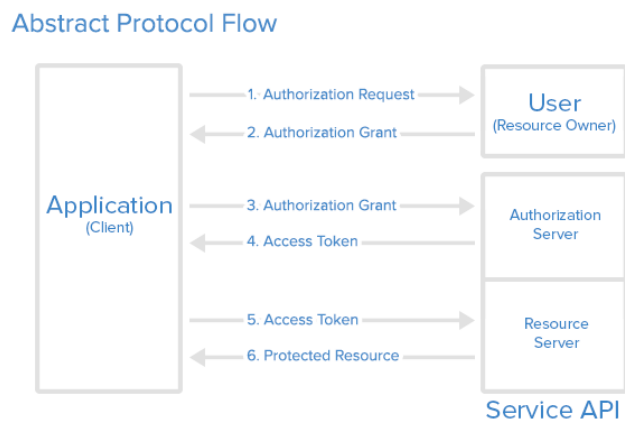


Figure 33 - OAuth 2.0 Protocol Flow

It works similar to Skydots 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

Memory usage															
Line size Kb	C (gcc)	C++ (G++)	Perl5	Python	Python3	Ruby	Lua	tcl	PHP	Javascript (sm)	Javascript (V8)	Java (gcj)	Java (OpenJDK)	Java (Sun)	
0	1,668	2,932	4,776	5,352	10,328	11,040	2,416	1,236	36,752	7,720	39,272	49,156	72,4832	658,560	
256	1,928	3,444	5,052	6,384	13,404	9,620	3,960	13,696	38,040	50,664	47,236	68,320	725,852	661,056	
512	2,184	3,956	5,308	5,876	16,476	11,672	5,404	14,720	39,064	29,672	47,636	76,200	725,852	661,056	
768	2,440	3,956	5,564	7,676	19,548	7,328	6,428	18,052	40,088	16,872	49,404	84,392	725,852	661,056	
1024	2,696	4,980	5,820	6,388	14,420	12,704	7,820	14,716	41,112	53,224	46,540	92,584	725,852	661,056	
1280	2,952	4,980	6,076	9,212	15,444	8,604	6,104	15,228	42,136	44,520	47,044	110,072	725,852	661,056	
1536	3,208	4,980	6,332	6,900	16,468	11,164	10,572	18,816	43,160	21,480	50,124	118,264	725,852	662,080	
1792	3,464	4,980	6,588	7,156	17,492	8,856	11,812	16,252	44,184	38,376	51,916	126,976	725,852	662,080	
2048	3,720	7,028	6,844	11,516	18,516	13,724	10,908	16,764	45,208	51,176	47,540	126,976	725,852	662,080	
2304	3,976	7,028	7,100	7,668	19,540	12,700	6,644	17,276	46,232	38,376	46,252	161,824	725,852	662,080	
2560	4,232	7,028	7,356	7,924	20,564	11,160	15,592	22,912	41,876	41,960	44,452	161,824	725,852	662,080	
2816	4,488	7,028	7,612	8,180	21,588	14,748	16,848	18,300	42,388	79,336	50,612	161,824	725,852	662,080	
3072	4,744	7,028	7,868	8,436	22,612	15,772	15,716	18,812	49,304	73,704	51,636	161,824	725,852	662,080	
3328	5,000	7,028	8,124	8,692	23,636	16,796	19,492	19,324	50,328	39,400	55,996	170,536	725,852	662,080	
3584	5,256	7,028	8,380	12,536	24,660	17,820	17,072	19,840	43,924	27,624	46,500	170,536	725,852	662,080	
3840	5,512	7,028	8,636	9,204	25,684	18,844	23,276	20,348	44,436	29,160	58,556	170,536	725,852	662,080	
4096	5,768	11,124	8,892	9,460	26,708	15,768	20,200	20,860	44,948	96,232	59,836	170,536	725,852	662,080	

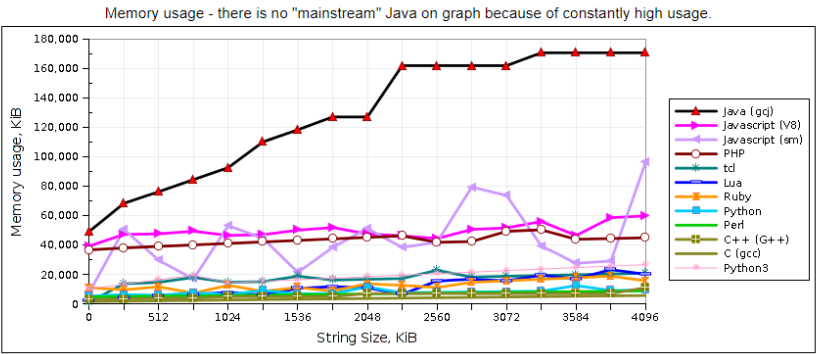


Figure 34 - Memory Usage Comparison

This table shows number of seconds taken to complete every testing stage.

Line size Kb	Perl5	PHP	Ruby	Python	C++ (g++)	C (gcc)	Javascript (V8)	Javascript (sm)	Python3	tcl	Lua	Java (openJDK)	Java (Sun)	Java (gcj)
256	2	6	7	7	7	2	3	30	17	33	49	39	38	451
512	7	23	29	32	26	8	21	131	81	141	203	162	157	1783
768	16	54	75	78	60	19	51	300	201	324	480	381	371	3937
1024	27	96	141	144	107	34	91	535	373	583	886	711	696	6952
1280	43	153	225	232	167	53	144	842	598	921	1423	1161	1145	10744
1536	62	227	328	342	242	76	208	1220	877	1334	2090	1751	1739	15372
1792	84	318	452	476	329	104	283	1672	1211	1823	2886	2489	2478	20819
2048	109	424	597	634	431	136	370	2203	1598	2387	3856	3370	3358	27132
2304	139	549	758	815	546	173	469	2799	2039	3030	4963	4453	4448	34302
2560	171	691	941	1019	675	214	578	3463	2533	3753	6198	5710	5719	42330
2816	206	849	1143	1248	817	259	700	4198	3070	4553	7568	7146	7186	51118
3072	245	1022	1366	1497	972	309	834	4997	3659	5422	9084	8852	8983	60779
3328	288	1211	1607	1771	1142	363	979	5875	4300	6378	10759	10784	10916	71275
3584	334	1414	1869	2064	1324	423	1136	6825	4992	7409	12594	12696	12867	82619
3840	384	1634	2150	2381	1522	487	1304	7848	5729	8503	14564	14861	15053	94686
4096	437	1869	2455	2720	1731	555	1484	8928	6534	9680	16674	17262	17426	107887

This table has the same results in more human-readable format (h:m:s)

Line size Kib	Perl5	PHP	Ruby	Python	C++ (g++)	C (gcc)	Javascript (V8)	Javascript (sm)	Python3	tcl	Lua	Java (openJDK)	Java (Sun)	Java (gcj)
256	0:00:02	0:00:06	0:00:07	0:00:07	0:00:07	0:00:02	0:00:03	0:00:30	0:00:17	0:00:33	0:00:49	0:00:39	0:00:38	0:07:31
512	0:00:07	0:00:23	0:00:29	0:00:32	0:00:26	0:00:08	0:00:21	0:02:11	0:01:21	0:02:21	0:03:23	0:02:42	0:02:37	0:29:43
768	0:00:16	0:00:54	0:01:15	0:01:18	0:01:00	0:00:19	0:00:51	0:05:00	0:03:21	0:05:24	0:08:00	0:06:21	0:06:11	1:05:37
1024	0:00:27	0:01:36	0:02:21	0:02:24	0:01:47	0:00:34	0:01:31	0:08:55	0:06:13	0:09:43	0:14:46	0:11:51	0:11:36	1:55:52
1280	0:00:43	0:02:33	0:03:45	0:03:52	0:02:47	0:00:53	0:02:24	0:14:02	0:09:58	0:15:21	0:23:43	0:19:21	0:19:05	2:59:04
1536	0:01:02	0:03:47	0:05:28	0:05:42	0:04:02	0:01:16	0:03:28	0:20:20	0:14:37	0:22:14	0:34:50	0:29:11	0:28:59	4:16:12
1792	0:01:24	0:05:18	0:07:32	0:07:56	0:05:29	0:01:44	0:04:43	0:27:52	0:20:11	0:30:23	0:48:06	0:41:29	0:41:18	5:46:59
2048	0:01:49	0:07:04	0:09:57	0:10:34	0:07:11	0:02:16	0:06:10	0:36:43	0:26:38	0:39:47	1:04:16	0:56:10	0:55:58	7:32:12
2304	0:02:19	0:09:09	0:12:38	0:13:35	0:09:06	0:02:53	0:07:49	0:46:39	0:33:59	0:50:30	1:22:43	1:14:13	1:14:08	9:31:42
2560	0:02:51	0:11:31	0:15:41	0:16:59	0:11:15	0:03:34	0:09:38	0:57:43	0:42:13	1:02:33	1:43:18	1:35:10	1:35:19	11:45:30
2816	0:03:26	0:14:09	0:19:03	0:20:48	0:13:37	0:04:19	0:11:40	1:09:58	0:51:10	1:15:53	2:06:08	1:59:06	1:59:46	14:11:58
3072	0:04:05	0:17:02	0:22:46	0:24:57	0:16:12	0:05:09	0:13:54	1:23:17	1:00:59	1:30:22	2:31:24	2:27:32	2:29:43	16:52:59
3328	0:04:48	0:20:11	0:26:47	0:29:31	0:19:02	0:06:03	0:16:19	1:37:55	1:11:40	1:46:18	2:59:19	2:59:44	3:01:56	19:47:55
3584	0:05:34	0:23:34	0:31:09	0:34:24	0:22:04	0:07:03	0:18:56	1:53:45	1:23:12	2:03:29	3:29:54	3:31:36	3:34:27	22:56:59
3840	0:06:24	0:27:14	0:35:50	0:39:41	0:25:22	0:08:07	0:21:44	2:10:48	1:35:29	2:21:43	4:02:44	4:07:41	4:10:53	26:18:06
4096	0:07:17	0:31:09	0:40:55	0:45:20	0:28:51	0:09:15	0:24:44	2:28:48	1:48:54	2:41:20	4:37:54	4:47:42	4:50:26	29:58:07

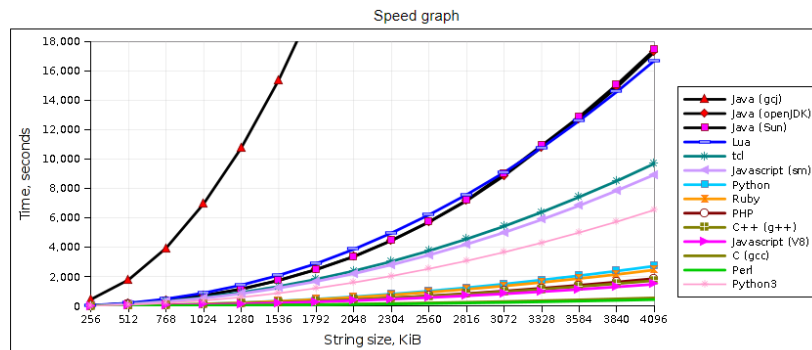


Figure 35 - Runtime Speed Comparison

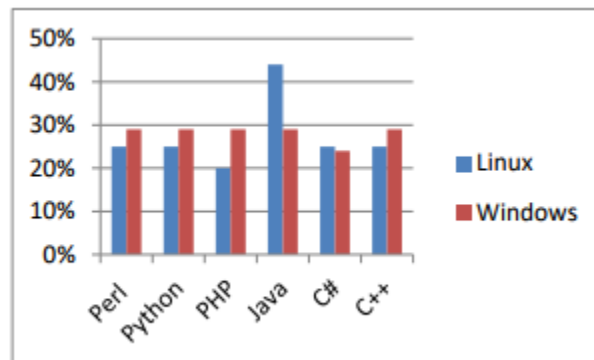


Figure 36 - CPU Consumption on Different Operating System

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

IaaS – Infrastructure as a Service

IOT – Internet of Things

OAS – OpenAPI Specification

OSS – Open Source Stack

PaaS – Platform as a Service

PCF – Pivotal Cloud Foundry

SaaS – Software as a Service

SCM – Software Configuration Management

SOA – Service-Oriented Architecture

UAT – User Acceptance Testing

UWP – Universal Windows Platform

VM – Virtual Machine

REFERENCES

- [1] T. Soroker, "Pivotal Cloud Foundry vs Kubernetes: Choosing The Right Cloud-Native Application Deployment Platform," 6 December 2017. [Online]. Available: <https://blog.takipi.com/pivotal-cloud-foundry-vs-kubernetes-choosing-the-right-cloud-native-application-deployment-platform/>. [Accessed December 2017].
- [2] "Perl, Python, Ruby, PHP, C, C++, Lua, tcl, javascript and Java comparison," 8 March 2011. [Online]. Available: <https://raid6.com.au/~onlyjob/posts/arena/#faq>. [Accessed October 2017].
- [3] "ReactJS vs Angular Comparison: Which is better?," 16 December 2016. [Online]. Available: <https://da-14.com/blog/reactjs-vs-angular-comparison-which-better>. [Accessed October 2017].
- [4] S. Greif, "Front-end Frameworks," 2016. [Online]. Available: <http://stateofjs.com/2016/frontend/>. [Accessed October 2017].
- [5] HotFrameworks, "JavaScript," 2017. [Online]. Available: <https://hotframeworks.com/languages/javascript>. [Accessed October 2017].
- [6] JetBrains, "Kotlin," [Online]. Available: <https://kotlinlang.org/>. [Accessed October 2017].
- [7] SimilarTech, "Top JavaScript Technologies," [Online]. Available: <https://www.similartech.com/categories/javascript>. [Accessed October 2017].
- [8] GitHub, "Front-end JavaScript frameworks," 2017. [Online]. Available: <https://github.com/collections/front-end-javascript-frameworks>. [Accessed October 2017].
- [9] Wappalyzer, "JavaScript Frameworks," [Online]. Available: <https://www.wappalyzer.com/categories/javascript-frameworks>. [Accessed October 2017].
- [10] W3Techs, "Usage of JavaScript libraries for websites," [Online]. Available: https://w3techs.com/technologies/overview/javascript_library/all. [Accessed October 2017].
- [11] BuiltWith, "JavaScript Usage Statistics," 2017. [Online]. Available: <https://trends.builtwith.com/javascript>. [Accessed October 2017].
- [12] Wikipedia, "Programming languages used in most popular websites," 2017. [Online]. Available:

https://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites. [Accessed September 2017].

- [13] J. Neuhaus, "Angular vs. React vs. Vue: A 2017 comparison," 28 August 2017. [Online]. Available: <https://medium.com/unicorn-supplies/angular-vs-react-vs-vue-a-2017-comparison-c5c52d620176>. [Accessed October 2017].
- [14] E. Korotya, "5 Best JavaScript Frameworks in 2017," 19 January 2017. [Online]. Available: <https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282>. [Accessed October 2017].
- [15] N. Kharchenko, "Vue.js and React.js – a Quick Comparison," 16 November 2017. [Online]. Available: <https://scotch.io/bar-talk/vuejs-and-reactjs-a-quick-comparison>. [Accessed November 2017].
- [16] R. T. & T. Android Articles, "Kotlin vs Java, What is the difference?," 16 October 2017. [Online]. Available: <http://androiddeveloper.galileo.edu/2017/10/16/kotlin-vs-java-what-is-the-difference/>. [Accessed November 2017].
- [17] D. Sato, "CanaryRelease," 25 June 2014. [Online]. Available: <https://martinfowler.com/bliki/CanaryRelease.html>. [Accessed November 2017].
- [18] N. Dhandala, "Docker Swarm vs Kubernetes," 6 June 2017. [Online]. Available: <https://blog.cloudboost.io/docker-swarm-vs-kubernetes-c796e630ca87>. [Accessed November 2017].
- [19] Netflix, "Netflix Open Source Software Center," [Online]. Available: <https://netflix.github.io/>. [Accessed September 2017].
- [20] C. Richardson, "Microservice Architecture," 2017. [Online]. Available: <http://microservices.io/>. [Accessed September 2017].
- [21] J. Rasmusson, "What is Agile?," [Online]. Available: <http://www.agilenutshell.com/>. [Accessed November 2017].
- [22] T. Conforto, "[Chicken-users] Hello World execution time," 13 March 2011. [Online]. Available: <https://lists.nongnu.org/archive/html/chicken-users/2011-03/msg00070.html>. [Accessed October 2017].
- [23] I. Zahariev, "C++ vs. Python vs. Perl vs. PHP performance benchmark (2016)," 9 February 2016. [Online]. Available: <https://blog.famzah.net/2016/02/09/cpp-vs-python-vs-perl-vs-php-performance-benchmark-2016/>. [Accessed October 2017].

- [24] B. Peabody, "Server-side I/O Performance: Node vs. PHP vs. Java vs. Go," May 2017. [Online]. Available: <https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go>. [Accessed October 2017].
- [25] A. M. Abdo, "Comparing Common Programming Languages to Parse Big XML File in Terms of Executing Time, Memory Usage, CPU Consumption and Line Number on Two Platforms," September 2016. [Online]. Available: <https://eujournal.org/index.php/esj/article/viewFile/8056/7762>. [Accessed October 2017].
- [26] B. Aruoba, "A Comparison of Programming Languages in Economics," 5 August 2014. [Online]. Available: http://economics.sas.upenn.edu/~jesusfv/comparison_languages.pdf. [Accessed October 2017].
- [27] Platform9, "Kubernetes vs Docker Swarm," 22 June 2017. [Online]. Available: <https://platform9.com/blog/kubernetes-docker-swarm-compared/>. [Accessed September 2017].
- [28] Anita, "What are Microservices?," 31 October 2016. [Online]. Available: <https://www.weave.works/blog/what-are-microservices/>. [Accessed December 2017].
- [29] Wikipedia, "Unix philosophy," [Online]. Available: https://en.wikipedia.org/wiki/Unix_philosophy. [Accessed December 2017].
- [30] hgraca, "Onion Architecture," 17 September 2017. [Online]. Available: <https://herbertograca.com/2017/09/21/onion-architecture/>. [Accessed December 2017].
- [31] Docker, "Overview of Docker Hub," Docker, [Online]. Available: <https://docs.docker.com/docker-hub/>. [Accessed April 2018].
- [32] T. Ugurlu, "Walkthrough and Benefits of Managed Kubernetes in Azure (Azure Container Service, AKS)," 9 December 2017. [Online]. Available: <https://medium.com/ingeniouslysimple/walkthrough-and-benefits-of-managed-kubernetes-in-azure-azure-container-service-aks-45168667920c>. [Accessed April 2018].
- [33] Janakiram, "KUBERNETES: AN OVERVIEW," 7 November 2016. [Online]. Available: <https://thenewstack.io/kubernetes-an-overview/>. [Accessed April 2018].
- [34] C. Richardson, "Introduction to Microservices," Nginx, 19 May 2015. [Online]. Available: <https://www.nginx.com/blog/introduction-to-microservices/>. [Accessed April 2018].

- [35] E. Forbes, "Debunking the Java Performance Myth," 22 October 2017. [Online]. Available: <https://blog.cloudboost.io/debunking-the-java-performance-myth-29b842955a24>. [Accessed March 2018].
- [36] "Microservices in Java—A Second Look," 18 October 2017. [Online]. Available: <https://hackernoon.com/microservices-in-java-a-second-look-460ba3909c44>. [Accessed March 2018].
- [37] "Kubernetes Millicores," 14 December 2016. [Online]. Available: <http://www.noqcks.io/notes/2016/12/14/kubernetes-understanding-millicores/>. [Accessed April 2018].
- [38] Spring, "Spring Framework," [Online]. Available: <https://projects.spring.io/spring-framework/>. [Accessed April 2018].
- [39] M. Azure, "Pricing Calculator," [Online]. Available: <https://azure.microsoft.com/en-us/pricing/calculator/>. [Accessed February 2018].
- [40] M. Azure, "Total Cost of Ownership (TCO) Calculator," [Online]. Available: <https://www.tco.microsoft.com/Home/Calculator?correlationId=1612358e-ad00-44fb-8f9d-466bca7fdf85>. [Accessed April 2018].
- [41] T. Soroker, "Pivotal Cloud Foundry vs Kubernetes: Choosing The Right Cloud-Native Application Deployment Platform," 6 December 2017. [Online]. Available: <https://blog.takipi.com/pivotal-cloud-foundry-vs-kubernetes-choosing-the-right-cloud-native-application-deployment-platform/>. [Accessed April 2018].
- [42] T. Point, "OAuth 2.0 - Overview," [Online]. Available: https://www.tutorialspoint.com/oauth2.0/oauth2.0_overview.htm. [Accessed April 2018].
- [43] Xebialabs, "Periodic Table of DevOps Tools," [Online]. Available: <https://xebialabs.com/periodic-table-of-devops-tools/>. [Accessed December 2017].
- [44] A. Omelianenko and S. Zabigailo, "kubernetes + travis-ci," 3 August 2017. [Online]. Available: <https://www.softserveinc.com/en-us/blogs/kubernetes-travis-ci/>. [Accessed April 2018].
- [45] A. Alliance, "Agile 101," [Online]. Available: <https://www.agilealliance.org/agile101/>. [Accessed April 2018].
- [46] B. Nice, "Front-End vs Back-End vs Full Stack Development," 12 May 2017. [Online]. Available: <https://medium.com/level-up-web/front-end-vs-back-end-vs-full-stack-development-78267f545121>. [Accessed April 2018].

- [47] P. Software, "Pivotal Web Services," [Online]. Available: <https://run.pivotal.io/pricing/>. [Accessed April 2018].
- [48] A. W. Services, "Pivotal Cloud Foundry on AWS," [Online]. Available: <https://aws.amazon.com/quickstart/architecture/pivotal-cloud-foundry/>. [Accessed April 2018].
- [49] C. N. C. Foundation, "Sustaining and Integrating Open Source Technologies," [Online]. Available: <https://www.cncf.io/>. [Accessed April 2018].
- [50] Pivotal, "Pivotal Cloud Foundry," [Online]. Available: <https://pivotal.io/platform>. [Accessed April 2018].