COMP4906 HONOURS

THESIS: SKYDOT

PROPOSAL

Lexi Brown - 100956208

TABLE OF CONTENTS

OUTLINE

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 proposal, 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 Proposal Objectives** – Objectives set by this proposal 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. **Investigation of Alternative Solutions** – An analysis of Skydots design choices in comparison to alternative technologies that could have been used and to technologies that inhibit problems Skydots features will aim to solve.
    - **2.1 Architecture** – A simple introduction to microservices and a comparison to other architecture styles.
    - **2.2 Technologies** – A simple introduction to technologies utilized by Skydot and a comparison between them.
3. **Analysis/Results of Research** – An in-depth report of all research and findings involved in choosing the right technologies for Skydot. And an analysis of key components that will show off Skydots abilities.
    - **3.1 Kubernetes vs Docker Swarm** – A comparison between two similar technologies, Kubernetes and Docket Swarm, for the chose of application container.
    - **3.2 Microservices Architecture** – An in-depth look at microservices and how they are deployed in creating and maintaining applications.
    - **3.3 Layered Architecture** – An analysis of the benefits of the layered architecture.
    - **3.4 Micro App and Service Languages** – An analysis of languages that will reflect the benefits of Skydot.

3.5 Client Side – Research gone into choosing client-side technologies that properly mimic the current industry.

3.6 Development – How Skydot contributes to application development techniques.

4. **Solution** – A summary of how Skydot is expected to complete all the project goals and how that solution will be measured and verified. Also, an outline of the plan of action for the next few months of implementation.
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 backend services, databases and other resources via a REST translation layer. This way backend services won't have to change to adopt new technologies and new services won't have to accommodate for old technologies. The project will be presented as a mobile banking service providing data for Android, Web and, if time permits, iPhone mobile application.

1.1 Problem

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

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

1.2 Motivation

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

1.3 Project Goals

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 Proposal 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. Optionally establish a host gateway that provides a REST API for micro-services to access backend services. The backend consists of REST and SOAP services.

vi. Build an authentication database server to generate and keep tokens for client application requests.

vii. Build backend services for authentication, account information, currency conversion and bill payments. At least one WSDL service must be provided.

viii. Provide micro-services in Java, Python 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 development (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

2 INVESTIGATION OF ALTERNATIVE SOLUTIONS

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.

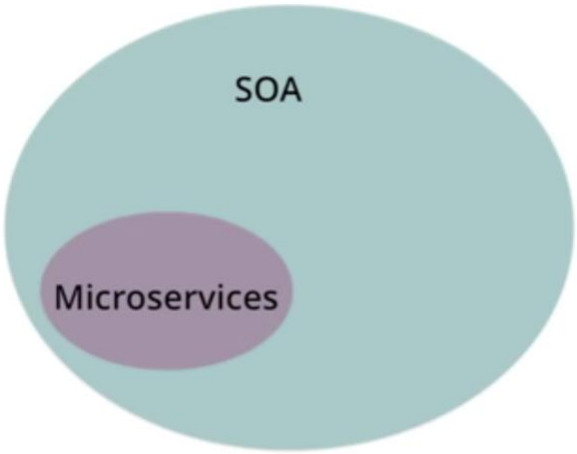

Figure 1 - Microservices vs. SOA

The alternative to microservice architecture is monolithic (Figure 2 - Microservices vs. Monolithic), n-tier and SOA. These traditional models and the inherit disadvantages of substantially less iteration, high maintenance cost, and associated organization has led to the 'Micro' trend. Skydots reference architecture is based on the notion of micro-services and micro-apps. In microservices, services can operate and be deployed independently of

other services. This way it is easier to deploy new versions of services frequently or scale a service independently. The main difference between SOA and microservices lies in the size and scope. Microservices are independently deployed and significantly smaller then what SOA tends to be, allowing for a more focused decoupling. SOA tends to be large deployments of closely coupled services. The micro theme encourages the separation and break down of code into manageable chunks, while still allowing interaction via a REST interface. Other technologies like WebSockets and gRPC could also be utilized.

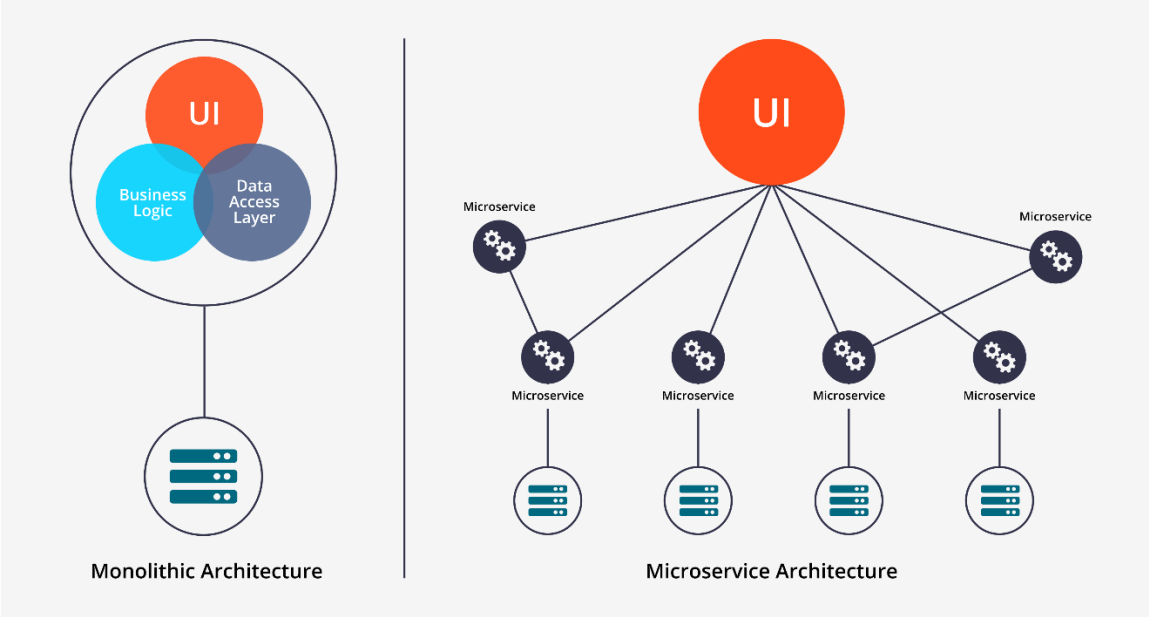

Figure 2 - Microservices vs. Monolithic

However, there are some drawbacks to microservices in comparison to the alternatives. As Fred Brooks wrote in The Mythical Man-Month, there are no silver bullets. One drawback microservices features is the emphasis on service size. While small services are preferable, they are a means to an end, not the primary goal. The goal of microservices is to sufficiently decompose an application to facilitate agile development and deployment but there are still other aspects to the application that need handling that microservices do not cover. An example of this would be service organization, which is where API gateways

and service discovery comes in. Problems arises from the complexity of this distributed system. An API gateway needs to be chosen and implemented as an inter-process communication mechanism that also handles partial failure when requests may be unavailable or slow. Although this isn't extremely complex, the solution is much simpler in a monolithic application. There are more drawbacks to microservices such as dealing with partitioned databases, managing testing, implementing changes that span multiple services and deploying each service. However, Skydot tackles all these problems within microservices and the remaining sections go into detail on the technologies used to do so.

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

2.2 Technologies

2.2.1 API Gateways

An API gateway is a single-entry point for all clients. It handles the passing of requests to services. API gateways can expose a different API for each client so that the provided API is best suited to the client requirements. This insulates the clients from how the application is partitioned into microservices and from the problem of determining the locations of each service.

When I researched which technology to use for Skydots API gateway to tackle microservice issues such as service organization and complexity of a microservice distributed systems, I created a list of mandatory and optional API requirements that would help narrow down the search (Table 1 - API Gateway Requirements).

API Requirements	Mandatory	Impact
Load Balancing	Y	Handles client downtime on any application update
Zero downtime deployment	Y	
Programmatic API registration	Y	
Authentication/Authorization	Y	No common transparent security mechanism
Supported types: JWT, OAuth2	Y	Supports standards
Key Expiry	Y	Access revocation. Handles zombie clients
Log pumps	Y	Provides real-time troubleshooting and metrics
Real-time analytics and monitoring	Y	
Micro app and service version handling	Y	Be able to support older versions of applications
REST API	Y	
Programmatic API registration	Y	Zero downtime deployments
API Gateway dashboard	N	Allows for easier understanding with visual representation of service topology
API Management portal	N	
Auto-generated developer documentation from Swagger API	N	Provides documentation coverage
Auto-generated client templates and test from Swagger API	N	Provides template for developers and reduces development setup time
Swagger support	Y	Wide market support for tooling. Alternatives, like RAML, have a limited pool of industry knowledge
Orchestration, Mediation and Transformation	Y	

Routing based on URI, port or headers	Y	Provides A/B testing, canary deployment[1] and load balancing
Request and response header manipulation	Y	Allows conversion of security keys into real customer objects
SOAP translation	N	Wouldn't have to create systems own SOAP-to-REST converter
Local and Cloud deployment	Y	Allows for local testing and development. Deals with network latency and reliability
Stateless and horizontally scalable	Y	Allow for single point failure

Table 1 - API Gateway Requirements

This are the technologies I researched as possible API gateways. I compared their functionalities to the table above, Table 1 - API Gateway Requirements.

Technology	Pros	Cons
Tyk **https://tyk.io/**	• Community edition is free • Setup and use is simple • Has many technologies such as, JWT, blacklisting, whitelisting, etc • Open Source • Supports Swagger	• Not well documented • Not widely used

[1] **Canary [deployment]** is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody. [17]

MuleSoft **https://www.mulesoft.com/**	• Includes open source components • Offers excellent ESB • Many connectors for legacy systems • Allow for local testing	• Expensive • API management exclusively tied to the ESB • Analytics are not real-time and unavailable in private cloud applications • No zero-downtime API deployment • Minimal support for 3rd party protocols like OAuth • API Modeling is tied to RAML • Does not support Swagger
Sentinet **http://www.nevatech.com/**	• Microsoft compatible • Clean, simple, easy-to-use dashboard • Supports Swagger • Well documented	• Only runs on Windows • Uses Silverlight which isn't supported in Chrome • Setting up parameters for REST to SOAP requests is done manually • No caching out of the box
Kubernetes **https://kubernetes.io/**	• Supports Swagger • Gateway is provided by the	• Difficult to set up outside of a cloud provider (Effects local development/testing) • Steep learning curve

	application container • Service registration between identical applications on different ports • Well documented with tutorials • Integrated with docker to support zero downtime deployments • Provides private and public network access to APIs • Auto-scaling, auto-placement, auto-restart, auto-replication • Open Source	• Incompatible with Docket CLI and Compose tools • YAML definitions are unique. Would have to be rewritten if there was a platform switch

Table 2 - API Gateway Technologies Researched Pro/Con

API Gateway Chosen: **Kubernetes**

Although I've chosen to utilize an API gateway, Kubernetes, there is another approach to automation, service discovery and containerization. That is the use of an off-

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

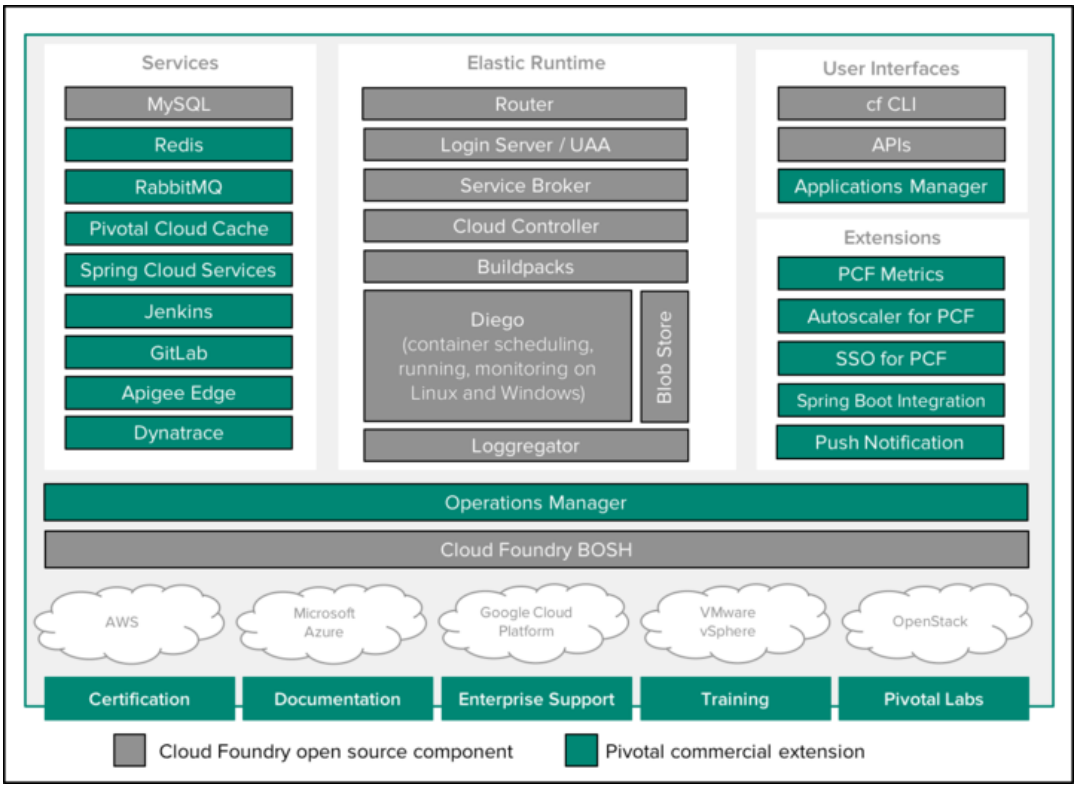

Figure 3 - Pivotal Cloud Foundry architecture – open source and enterprise

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

What is a PaaS?

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

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

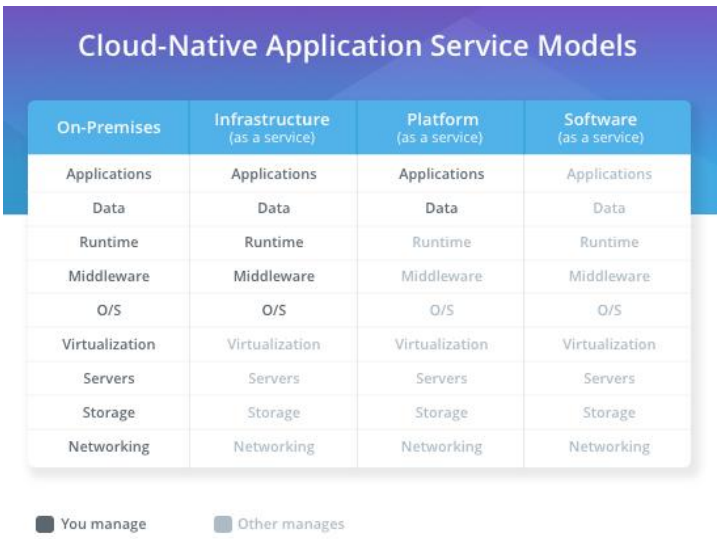

Figure 4 - Cloud-Native Service Models Comparison

3 ANALYSIS/RESULTS OF RESEARCH

3.1 Kubernetes vs Docker Swarm

When researching what technologies to rely on for Skydot, I had to look at which application container I would use. There are many cloud compatible application containers such as Kubernetes, Docker Swarm, AWS ECS and Apache Mesos. Unfortunately, AWS ECS is for AWS. Mesos was a good contestant, however, based on popularity and mindshare, I focused on Kubernetes and Docker Swarm.

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 for Kubernetes is shown below:

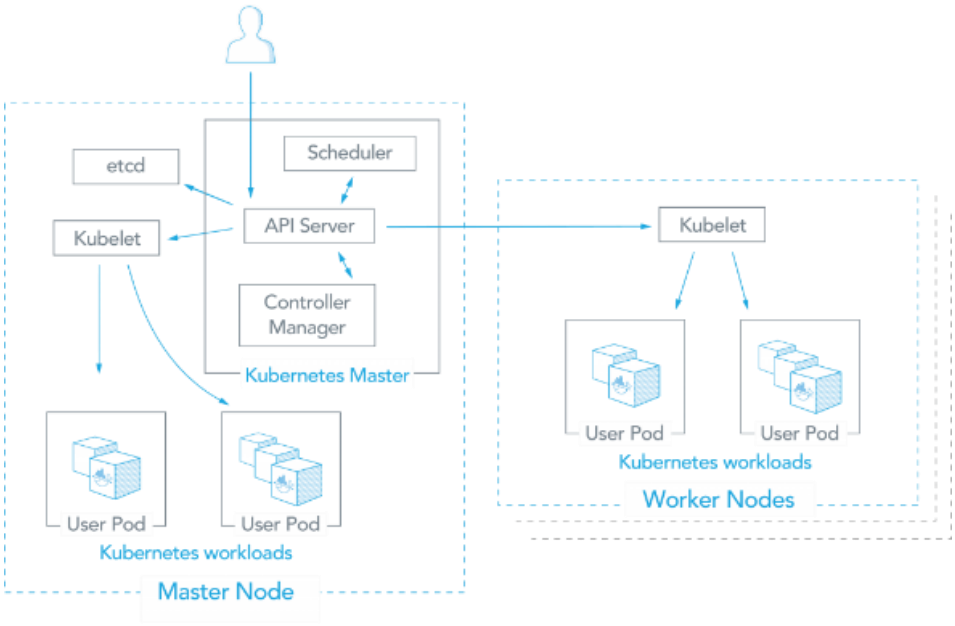

Figure 5 - Kubernetes topology

Docker Swarm is a clustering and scheduling tool for docker containers. Docker Swarm allow developers to deploy, establish and manage clusters of Docker nodes in a

single virtual system container in a mode called 'Swarm Mode'. A Swarm cluster consists of a docker engine deployed on multiple nodes. Manager nodes perform orchestration and cluster management. Worker nodes receive and execute tasks from the manager nodes. A service, which can be specified declaratively, consists of tasks that can be run on Swarm nodes. Services can be replicated to run on multiple nodes. The architecture for Docker Swarm is shown below:

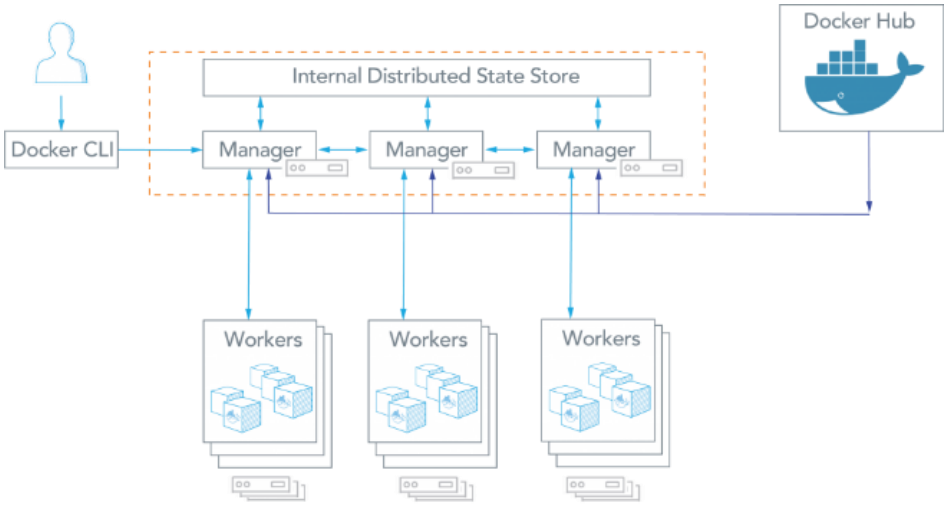

Figure 6 – Docker Swarm topologies

This tables compares the features of Kubernetes and Docker Swarm.

Kubernetes	Docker Swarm
Application Definition	
Applications can be deployed using a combination of pods, deployments, and services (or "microservices"). A pod is a group of co-located containers and is the atomic unit of a deployment. A deployment can have replicas across multiple nodes. A service is the "external face" of container workloads	Applications can be deployed as services (or "microservices") in a Swarm cluster. Multi-container applications can be specified using YAML files. Docker Compose can deploy the app. Tasks (an instance of a service running on a node) can be distributed across datacenters using labels. Multiple placement preferences

and integrates with DNS to round-robin incoming requests.	can be used to distribute tasks further, for example, to a rack in a datacenter.
Application Scalability Constructs	
Each application tier is defined as a pod and can be scaled when managed by a deployment, which is specified in YAML. The scaling can be manual or automated. Pods can be used to run vertically integrated application stacks such as LAMP (Apache, MySQL, PHP) or ELK/Elastic (Elasticsearch, Logstash, Kibana), co-located and co-administered apps such as content management systems and apps for backups, checkpoint, compression, rotation, snapshotting.	Services can be scaled using Docker Compose YAML templates. Services can be global or replicated. Global services run on all nodes, replicated services run replicas (tasks) of the services across nodes. For example, A MySQL service with 3 replicas will run on a maximum of 3 nodes. Tasks can be scaled up or down, and deployed in parallel or in sequence.
High Availability	
Deployments allow pods to be distributed among nodes to provide high availability, thereby tolerating application failures. Load-balanced services detect unhealthy pods and remove them. High availability of Kubernetes is supported. Multiple master nodes and worker nodes can be load balanced for requests from kubectl and clients. etcd can be clustered and API Servers can be replicated.	Services can be replicated among Swarm nodes. Swarm managers are responsible for the entire cluster and manage the resources of worker nodes. Managers use ingress load balancing to expose services externally. Swarm managers use Raft Consensus algorithm to ensure that they have consistent state information. An odd number of managers is recommended, and most managers must be available for a functioning Swarm cluster (2 of 3, 3 of 5, etc.).
Load Balancing	
Pods are exposed through a service, which can be used as a load-balancer within the cluster.	Swarm mode has a DNS component that can be used to distribute incoming requests to a service name. Services

	can run on ports specified by the user or can be assigned automatically.
Auto-scaling for the Application	
Auto-scaling using a simple number-of-pods target is defined declaratively using deployments. CPU-utilization-per-pod target is available.	Not directly available. For each service, you can declare the number of tasks you want to run. When you manually scale up or down, the Swarm manager automatically adapts by adding or removing tasks.
Rolling Application Upgrades and Rollback	
The deployment controller supports both “rolling-update” and “recreate” strategies. Rolling updates can specify maximum number of pods unavailable or maximum number running during the process.	At rollout time, you can apply rolling updates to services. The Swarm manager lets you control the delay between service deployment to different sets of nodes, thereby updating only 1 task at a time.
Health Checks	
Two kinds of health checks: liveness (is app responsive) and readiness (is app responsive, but busy preparing and not yet able to serve) Out-of-the-box K8S provides a basic logging mechanism to pull aggregate logs for a set of containers that make up a pod.	Docker swarm health checks are limited to services. If a container backing the service does not come up (running state), a new container is kicked off.
Networking	
The networking model is a flat network, enabling all pods to communicate with one another. Network policies specify how pods communicate with each other. The flat network is typically implemented as an overlay.	Node joining a Docker Swarm cluster creates an overlay network for services that span all the hosts in the Swarm and a host only Docker bridge network for containers. By default, nodes in the Swarm cluster encrypt overlay control and management traffic between

	themselves. Users can choose to encrypt container data traffic when creating an overlay network by themselves.
Service Discovery	
Services can be found using environment variables or DNS. Kubelet adds a set of environment variables when a pod is run. Kubelet supports simple {SVCNAME_SERVICE_HOST} AND {SVCNAME_SERVICE_PORT} variables, as well as Docker links compatible variables. DNS Server is available as an addon. For each Kubernetes Service, the DNS Server creates a set of DNS records. If DNS is enabled in the entire cluster, pods will be able to use Service names that automatically resolve.	Swarm Manager node assigns each service a unique DNS name and load balances running containers. Requests to services are load balanced to the individual containers via the DNS server embedded in the Swarm. Docker Swarm comes with multiple discovery endpoints: • Docker Hub as a hosted discovery service is intended to be used for dev/test. Not recommended for production. • A static file or list of nodes can be used as a discovery backend. The file must be stored on a host that is accessible from the Swarm Manager. You can also provide a node list as an option when you start Swarm.
Performance and scalability	
With the release of 1.6, Kubernetes scales to 5000-node clusters. Kubernetes scalability is benchmarked against the following Service Level Objectives (SLOs): • API responsiveness: 99% of all API calls return in less than 1s.	Docker Swarm has been scaled and performance tested up to 30,000 containers and 1,000 nodes with 1 Swarm manager.

• Pod startup time: 99% of pods and their containers (with pre-pulled images) start within 5s.	

Table 3 - Kubernetes vs Docker Swarm Comparison

Kubernetes	Docker Swarm
Pros	
• Based on extensive experience running Linux containers at Google. Deployed at scale more often among organizations. • Backed by enterprise offerings from both Google (GKE) and RedHat (OpenShift) • Can overcome constraints of Docker and Docker API • Auto scaling based on factors such as CPU utilization • Largest community among container orchestration tools. Over 50,000 commits and 1200 contributors	• Deployment is simpler and Swarm mode is included in Docker Engine • Integrates with Docker Compose and Docker CLI – native Docker tools. Many of the Docker CLI commands will work with Swarm. Easier learning curve.
Cons	
• Do-it-yourself installation can be complex, but flexible. Deployment tools include kubeadm, kops, kargo, and others • Uses a separate set of tools for management, including kubectl CLI	• Does not have as much experience with production deployments at scale • Limited to the Docker API's capabilities • Services can be scaled manually • Smaller community and project. Over 3,000 commits and 160 contributors

Common Features
• Open source projects. Anyone can contribute using the Go programming language • Various storage options • Networking features such as load balancing and DNS • Logging and Monitoring add-ons. These external tools include Elasticsearch/Kibana (ELK), sysdig, cAdvisor, Heapster/Grafana/InfluxDB

Table 4 - Pros and Cons of Kubernetes and Docker Swarm

Based on Table 3 - Kubernetes vs Docker Swarm Comparison and Table 4 - Pros and Cons of Kubernetes and Docker Swarm, and implementing both containers locally, I went with **Kubernetes**. Also, a large contributing facture was the mindshare (Figure 7 - Kubernetes vs. Docker Swarm Mindshare) between the two technologies; Kubernetes is the more popular choice and is increasingly being embedded into third-party solutions.

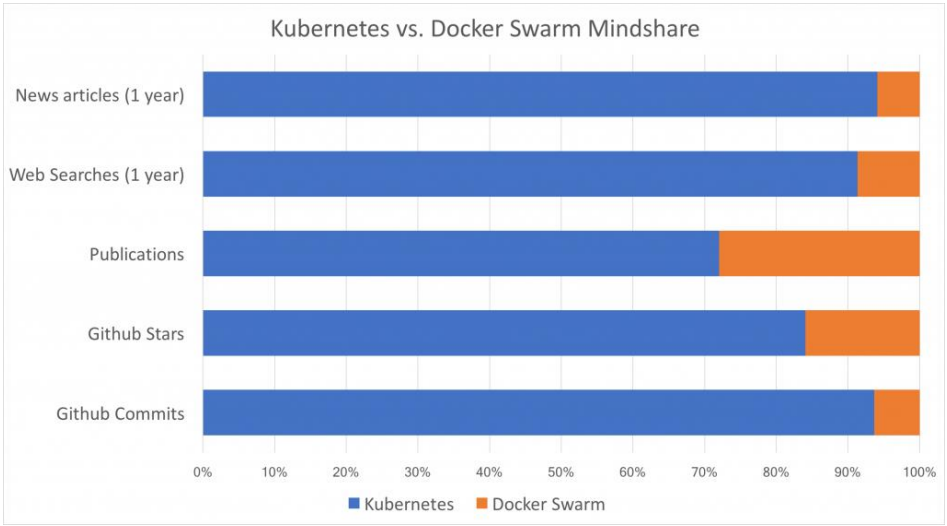

Figure 7 - Kubernetes vs. Docker Swarm Mindshare

3.2 Microservice Architecture

Skydot utilizes a microservice architecture style. The idea of microservices, in terms of Skydot, is a decomposition of a once monolithic application into self-sufficient modules that perform one function extremely well. This type of structure is very appealing; modules can

scale to meet client popularity surge and wane, infrastructure can scale automatically to match the true cost of running these modules; and most importantly, developers and operators can work together as a single DevOps team, working independently of other teams. These teams would deliver modules with customer focused value.

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

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

I considered utilizing the Netflix stack for the platform because it was a mature offering that was production-proven. However, I quickly realized that it would not fit into the

modernized platform I am designing because I wanted three key things from a microservices perspective: polyglot development, multi-occupancy, and platform independence.

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

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

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

smaller the footprint, the more applications that can be run on the same resources (CPU, memory, and storage).

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

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

3.3 Layered Architecture

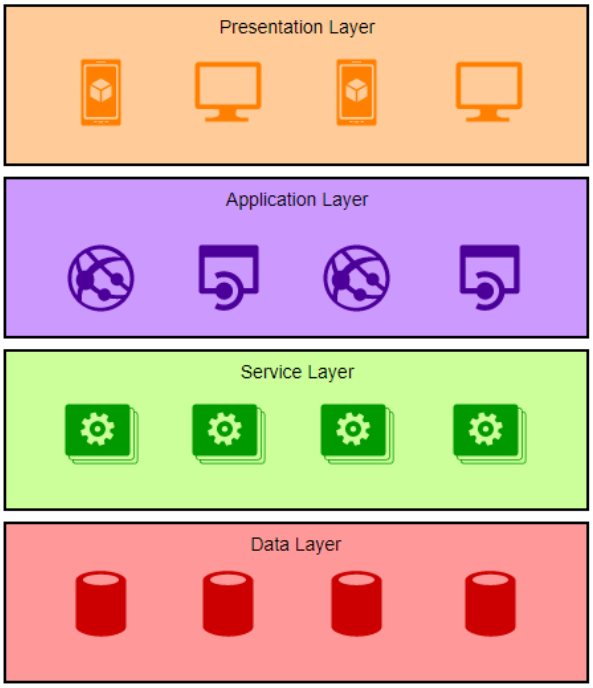

Figure 8 – Diagram of Skydots layered architecture

The entirety of Skydot is structured in a strict 4 tier layered style (Figure 8 – 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.

3.4 Micro App and Service Languages

For the micro apps and services, I wanted to show off how you can use any language you want to or any language you think is best suited for your application within Skydot. The current industry standard is Java and therefore a lot of companies are utilizing it while believing it is the best language to use. So, I researched and analysed the runtime and memory usage of various languages, including Java, to come up with a list of languages to use within Skydots micro apps and services.

From my research of both runtime speed (Figure 12 - Runtime Speed Comparison) and memory usage (Figure 14 – Memory Usage Comparison) it was clear that Java was the slowest and most memory consuming language. When the relative speed of Java was compared to other languages (Figure 13 - Runtime Speed Comparison Relative to Perl5), languages like C++ were ten times faster than the fastest version of Java (OpenJDK) and over sixty times faster than the slowest version of Java (GCJ). This was also reflected in the memory usage of Java. When the memory usage of Java was compared to other languages

(Figure 15 - Memory Usage Comparison Relative to Perl5), languages like C++ were twenty times more memory efficient than the least memory efficient version Java (GCJ) and over a hundred times more memory efficient than the least memory efficient version of Java (OpenJDK). Even programs as small as Hello World runs significantly slower in a Java environment (Figure 16 - Hello World Execution Time Comparison). Also, since Skydot will be utilizing virtual machines, different types of operating systems (OS) can be mimicked. Therefore, I considered run times of languages on Windows and Linux environments. There were three categories of comparison: runtime speed (Figure 17 - Language Execution Time on Linux and Windows), memory usage (Figure 18 - Language Memory Usage on Linux and Windows), and CPU consumption (Figure 19 - Language CPU Time Consumption on Linux and Windows). The overall statistics of runtime speed, memory usage and CPU consumption are features in Figure 20 - Overall Statistics of Memory Usage, Execution Time and CPU Consumption. As expected, Java ran better on Windows than Linux. However, Linux is a lighter OS than Windows, and Java almost always preformed the worst on the Linux OS. The only category that Java did not preform the worst in was runtime but in contrast, memory usage and CPU time consumption were very high. This is not ideal for cloud development as the cloud host being utilized, in this case Microsoft Azure, charges based on how much space you use and CPU usage. It can be noted that Java has many features and can do a lot more compared to other languages, however, for simplistic microservices, using Java would burn through project funding.

If the same application can be written in Python, with improvements on speed and memory usage, and cost less to host in the cloud than Java, why not use it? C and C++ were consistently in the top when it comes to efficiency with memory and computational speed and these are languages that can do more than Java. Here, it can also be noted that there may be more Java programmers than C and Python programmer hence why no one turns to these languages. However, now consider JavaScript. Any application that has a web front

end is extremely likely to be using some sort of JavaScript library, with few exceptions to this. This means there are many JavaScript programmers available. Although JavaScript isn't the best in speed or memory usage, it still beats out Java.

So, to show how important language selection is I've chosen four languages to use for the micro apps and services in Skydot. They will demonstrate how memory heavy and CPU intensive languages lose out in cloud development and how light and quick languages are key in cloud development. These languages are:

Java – Representing the high memory usage, slower language

Python – Representing the low memory usage, medium speed language

C++ – Representing the low memory usage, faster language

JavaScript – Representing the medium memory usage, medium speed language

These are the four main languages I will be utilizing. Additional languages that I may utilize given time are Go, C and PHP.

Language	Memory Usage	Runtime Speed
Java	High	Slow
C++	Low	Fast
Python	Low	Medium
JavaScript	Medium	Slow
Additional Languages		
Go	Low	Fast
C	Low	Fast
PHP	High	Medium

Table 5 - Language Memory and Runtime Chart

3.5 Client Side

Client-side technologies are not the most important aspect of this project. However, I feel it is necessary to consider what companies are utilizing on the client facing end to

properly show Skydots flexibility and to meet the current needs of the industry. Android and web will be fronting Skydots' services so I researched what frameworks are most commonly picked for each.

3.5.1 Android

To save development and research time, and because most Android OS is written in Java, I only looked at native solutions. Also, when using an Android language, more reliable code can be written. This means apps will have less bugs and crashes, and therefore a better user experience. The native language solutions are Java or Kotlin. Java is the official language for Android development whereas Kotlin is the new, up-and-coming Android development solution. For Skydot, I've chosen to use Kotlin as the language for the Android client app. Syntax wise Kotlin is completely interoperable with Java but adds its own flares and functionality. Kotlin can also work side by side with Java so one can call into Java from Kotlin and vice versa. Comparatively, for Android development, Kotlin does the same things as Java just easier and faster to develop, and there isn't a huge learning curve from Java to Kotlin. This makes Kotlin the new language to look to for Android development and Google is helping push this narrative. Kotlin offers:

- Full compatibility with JDK 6, ensuring Kotlin can be deployed on older Android devices
- Runtimes as fast as Java because of its similar bytecode structure
- Support for inline functions, code using lambdas often runs faster than the same code written in Java
- Allows use of all existing Android libraries, including annotations
- A very compact runtime library
- Efficient incremental compilation

3.5.2 Web

There many technologies used in web development: Angular, ReactJS, jQuery, Ember, Vue, Meteor, etc. However, the most stand out point is that JavaScript dominates the front-end language market. All the most popular websites use it and there really isn't a strong contender for second place. So, when choosing a framework for fronting Skydots services, I focused on three things: ease of use, usage in industry and support. These helped weed out frameworks like jQuery which is widely used but is older technology and not frequency used in new applications. The top frameworks that fit all three of those categories were: Angular, Ember, ReactJS, Vue and Meteor (Figure 9 - JavaScript Library Information Chart).

	AngularJS	Angular 2	ReactJS	Vue.js	Ember.js	Meteor.js
Definition	MVW framework	MVC framework	JavaScript library	MVC framework	MVC framework	JavaScript app platform
1st Release	2009	2016	2013	2014	2011	2012
Homepage	angularjs.org	angular.io	reactjs.net	vuejs.org	emberjs.com	www.meteor.com
# Contributors on GitHub	1,562	392	912	62	636	328
GitHub Star Rating	54,402	19,832	57,878	39,933	17,420	36,496

Figure 9 - JavaScript Library Information Chart

Meteor and Ember worked out to be the least talked about. Rarely was there research on JavaScript frameworks where both Meteor and Ember were included, it was one or the other (and sometimes Backbone). For Vue, those who used it gave it high praise and it was usually included in comparisons because of its simplicity, combination of Angular and React

capabilities, easy learning curve and high satisfaction amongst developers. However, it is too small and simple, and doesn't have enough backing to be used for Skydot.

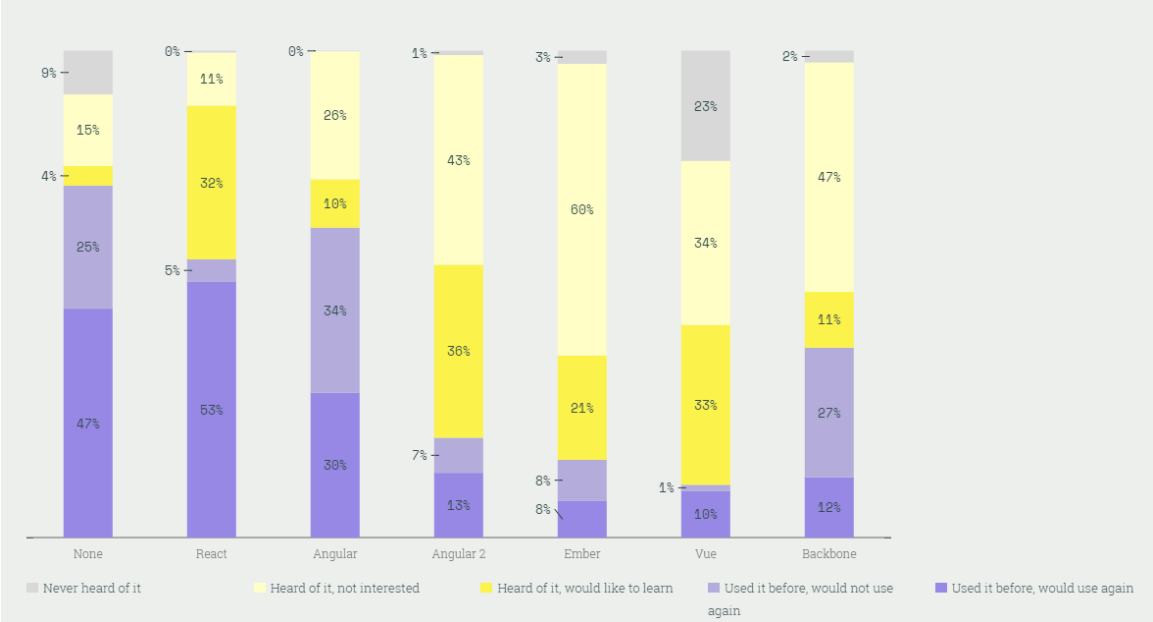

Figure 10 - JavaScript Library Interest Comparison

That left Angular versus ReactJS. Both are highly used and praised technologies and either would be fit enough to show off Skydots capabilities and represent the latest technology usage in the current industry. So really it came down to personal preference and what I needed to create. I spoke with some front-end development coworkers and did a few Angular and ReactJS tutorials and decided to go with ReactJS. As a non-front-end developer, I found that ReactJS had an easier learning curve and simpler set up than Angular.

3.6 Development

Agile development and Skydot easily go hand in hand. In the todays 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 the 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 a different micro-service. 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. 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.

4 SOLUTION

The plan of action for the next few months is to completely set up Skydot in Azure Cloud and analyze the costs and time saved by using the solution. Firstly, I will have to use Kubernetes as the application container that wraps the entire project and helps deploy Skydot. Through Kubernetes, an API gateway will be established so that micro-apps can register with it and client applications will be able to be send requests to these micro-apps. This will provide a common point of access for the client applications. These micro-apps will use service discovery to communicate with the service gateway through REST calls. The service gateway will be created to facade all the micro-services. These micro-services and micro-apps will be written in Java, C++, Python and JavaScript. Java will be used to represent the memory heavy and CPU intense language, while C++ will be used to represent the low memory usage, faster language. Python and JavaScript will represent the middle ground of memory usage and execution speed. These four languages will be used to demonstrate how choosing the right language will minimize the costs of cloud development.

After setting up all the services, a backend database system will be created offering both JSON and SOAP (WSDL) formatted data. The backend database will be accessed through a host gateway that the micro-services will talk to. To demonstrate end-to-end communication, Skydot will be retrofitted with a banking application. An android and web application will be built as the front, customer-facing end. There will be three micro-apps: a mobile micro-app that the android application will use, a web micro-app that the web application will use and an authentication micro-app that both clients will use. The services provided to these micro-apps will be: Authentication, Account summary and details, Transfers and Bill payment. Those services will be represented by micro-services. Then the backend database will be populated with multiple accounts and payment data.

This set up will show off all Skydots features and to meet the goals outlined for the project. The micro-apps and micro-services will be overloaded with requests for analyzing up and down scaling, memory usage, execution time and CPU consumption. Service application size will be recorded to analyze decreases in development time, this would include aspects like line size and efficient library usage. I will also compare the costs of running Skydot in the cloud as opposed to purchasing multiple on premise servers. And finally, I will analysis how many people would be needed to maintain a project running in Skydot. All these points will contribute to analyzing my solutions to the problems I've outline in 1.1 Problem.

5 CONCLUSION

In conclusion, the goal of Skydot is not just to use a “Micro Architecture” within the cloud. It’s a culture and an end-to-end process. Using all technologies and designs I’ve chosen for Skydot, all project goals can be met. However, even with these decisions, Skydot is decoupled enough that integrating a new technology (i.e., adding Cassandra) or shifting to a different technology (i.e., Kubernetes to Docker Swarm) would not bring down the whole system and require extensive conversion time. 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

After implementing Skydot within Microsoft Azure, future considerations would be integrating a multi-cloud design. With CICD and the flexibly of Skydot, it would be possible to utilized both Microsoft Azure and Amazon Web Services (AWS) (Figure 11 - Possible future Skydot usage with AWS).

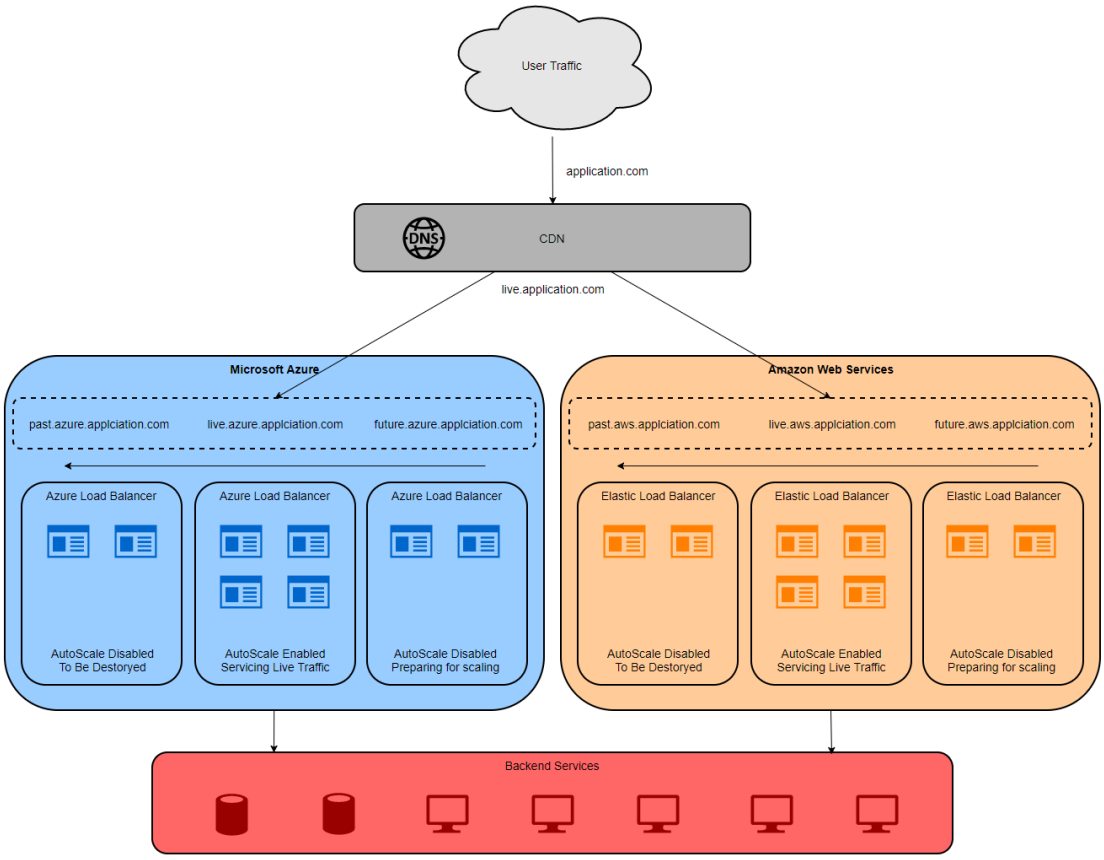

Figure 11 - Possible future Skydot usage with AWS

This way I 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.

ADDITIONAL FIGURES

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

Speed graph

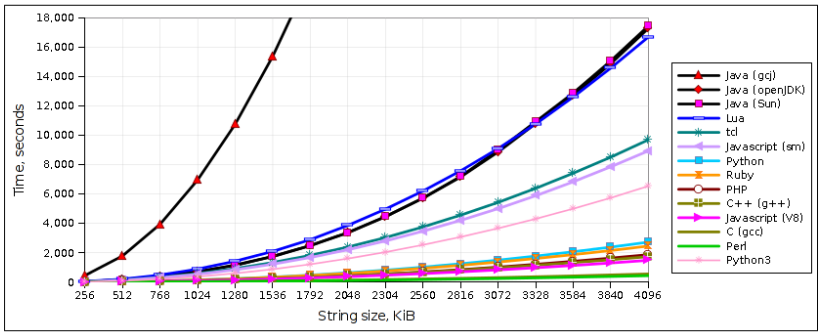

Figure 12 - Runtime Speed Comparison

Relative speed: Perl5 (fastest) taken as 1.

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	1	3.00	3.50	3.50	3.50	1.00	1.50	15.00	8.50	16.50	24.50	19.50	19.00	225.50
512	1	3.29	4.14	4.57	3.71	1.14	3.00	18.71	11.57	20.14	29.00	23.14	22.43	254.71
768	1	3.38	4.69	4.88	3.75	1.19	3.19	18.75	12.56	20.25	30.00	23.81	23.19	246.06
1024	1	3.56	5.22	5.33	3.96	1.26	3.37	19.81	13.81	21.59	32.81	26.33	25.78	257.48
1280	1	3.56	5.23	5.40	3.88	1.23	3.35	19.58	13.91	21.42	33.09	27.00	26.63	249.86
1536	1	3.66	5.29	5.52	3.90	1.23	3.35	19.68	14.15	21.52	33.71	28.24	28.05	247.94
1792	1	3.79	5.38	5.67	3.92	1.24	3.37	19.90	14.42	21.70	34.36	29.63	29.50	247.85
2048	1	3.89	5.48	5.82	3.95	1.25	3.39	20.21	14.66	21.90	35.38	30.92	30.81	248.92
2304	1	3.95	5.45	5.86	3.93	1.24	3.37	20.14	14.67	21.80	35.71	32.04	32.00	246.78
2560	1	4.04	5.50	5.96	3.95	1.25	3.38	20.25	14.81	21.95	36.25	33.39	33.44	247.54
2816	1	4.12	5.55	6.06	3.97	1.26	3.40	20.38	14.90	22.10	36.74	34.69	34.88	248.15
3072	1	4.17	5.58	6.11	3.97	1.26	3.40	20.40	14.93	22.13	37.08	36.13	36.67	248.08
3328	1	4.20	5.58	6.15	3.97	1.26	3.40	20.40	14.93	22.15	37.36	37.44	37.90	247.48
3584	1	4.23	5.60	6.18	3.96	1.27	3.40	20.43	14.95	22.18	37.71	38.01	38.52	247.36
3840	1	4.26	5.60	6.20	3.96	1.27	3.40	20.44	14.92	22.14	37.93	38.70	39.20	246.58
4096	1	4.28	5.62	6.22	3.96	1.27	3.40	20.43	14.95	22.15	38.16	39.50	39.88	246.88
Average:	1	3.84	5.21	5.59	3.89	1.23	3.23	19.66	13.92	21.35	34.36	31.16	31.12	247.32

Figure 13 - Runtime Speed Comparison Relative to Perl5

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

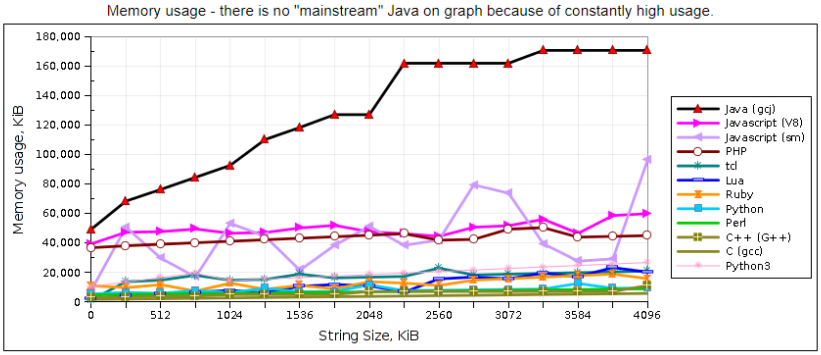

Figure 14 – Memory Usage Comparison

Memory usage in Perls + average

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	0.35	0.61	1	1.12	2.16	2.31	0.51	0.23	7.70	1.62	8.22	10.29	151.77	137.89
256	0.38	0.68	1	1.26	2.65	1.90	0.78	2.15	7.53	10.03	9.35	13.52	143.68	130.85
512	0.41	0.75	1	1.11	3.10	2.20	1.02	2.51	7.36	5.59	8.97	14.36	136.75	124.54
768	0.44	0.71	1	1.38	3.51	1.32	1.16	2.35	7.20	3.03	8.88	15.17	130.46	118.81
1024	0.46	0.86	1	1.10	2.48	2.18	1.34	2.30	7.06	9.15	8.00	15.91	124.72	113.58
1280	0.49	0.82	1	1.52	2.54	1.42	1.00	1.65	6.93	7.33	7.74	18.12	119.46	108.80
1536	0.51	0.79	1	1.09	2.60	1.76	1.67	2.73	6.82	3.39	7.92	18.68	114.63	104.56
1792	0.53	0.76	1	1.09	2.66	1.34	1.79	2.27	6.71	5.83	7.88	19.27	110.18	100.50
2048	0.54	1.03	1	1.68	2.71	2.01	1.59	1.46	6.61	7.48	6.95	18.55	106.06	96.74
2304	0.56	0.99	1	1.08	2.75	1.79	0.94	2.25	6.51	5.41	6.51	22.79	102.23	93.25
2560	0.58	0.96	1	1.08	2.80	1.52	2.12	2.89	5.69	5.70	6.04	22.00	98.67	90.01
2816	0.59	0.92	1	1.07	2.84	1.94	2.21	2.24	5.57	10.42	6.65	21.26	95.36	86.98
3072	0.60	0.89	1	1.07	2.87	2.00	2.00	2.23	6.27	9.37	6.56	20.57	92.25	84.15
3328	0.62	0.87	1	1.07	2.91	2.07	2.40	2.22	6.19	4.85	6.89	20.99	89.35	81.50
3584	0.63	0.84	1	1.50	2.94	2.13	2.04	1.58	5.24	3.30	5.55	20.35	86.62	79.01
3840	0.64	0.81	1	1.07	2.97	2.18	2.70	2.21	5.15	3.38	6.78	19.75	84.05	76.67
4096	0.65	1.25	1	1.06	3.00	1.77	2.27	2.21	5.05	10.82	6.73	19.18	81.63	74.46
Average:	0.53	0.85	1	1.20	2.79	1.87	1.62	2.09	6.45	6.28	7.39	18.28	109.87	100.13

Figure 15 - Memory Usage Comparison Relative to Perl5

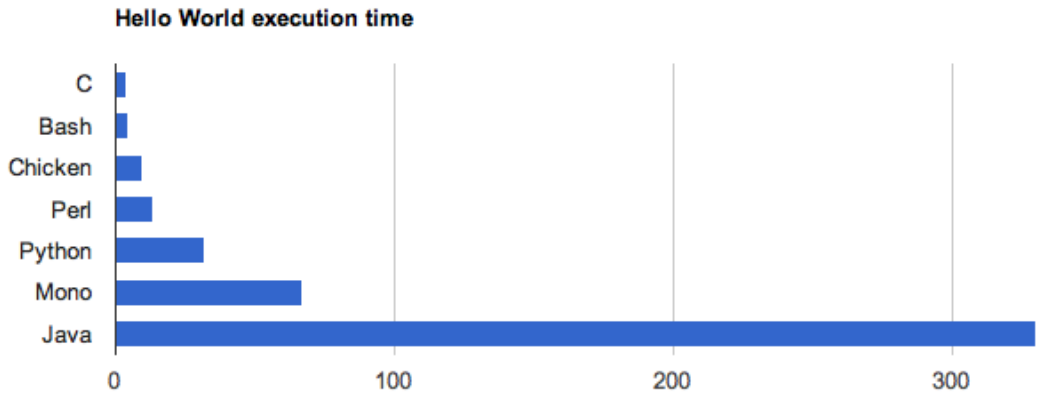

Figure 16 - Hello World Execution Time Comparison

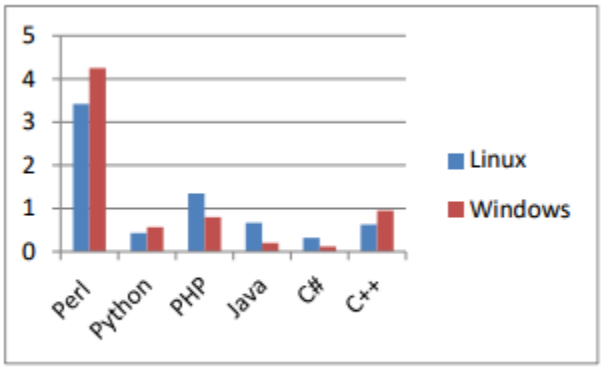

Figure 1: Execution time for each programming language on both platforms. It shows that C# is the faster one while Perl was the slowest one on both operating systems.

Figure 17 - Language Execution Time on Linux and Windows

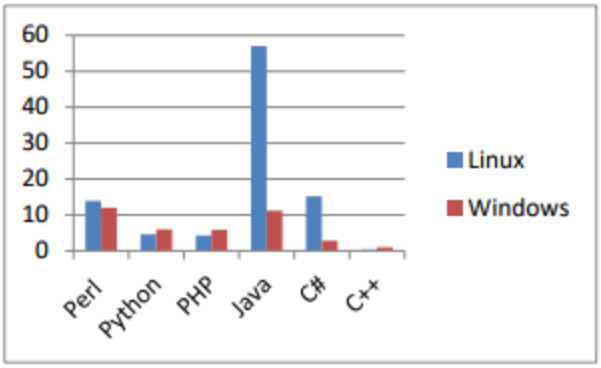

Figure 2: It shows the memory usage for each language on both platforms in Megabytes.

Figure 18 - Language Memory Usage on Linux and Windows

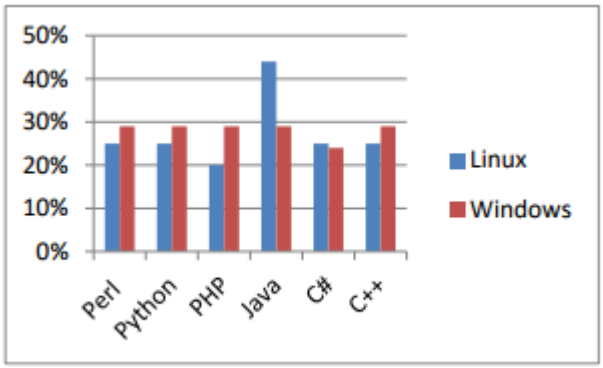

Figure 3: It displays CPU time consumption of each language on both operating systems.

Figure 19 - Language CPU Time Consumption on Linux and Windows

Time			Memory in MB			CPU			Time in Min			Line number		
Pro.Lang	Linux	Windows	Pro.Lang	Linux	Windows	Pro.Lang	Linux	Windows	Pro.Lang	Linux	Windows	Pro.Lang	Linux	Windows
Perl	3.42 min	4.25 min	Perl	13.9	12	Perl	25%	29%	Perl	3.42	4.25	Perl	46	46
Python	26 sec	34 sec	Python	4.6	6	Python	25%	29%	Python	0.43	0.57	Python	46	46
PHP	1.35 min	0.8 min	PHP	4.3	5.9	PHP	20%	29%	PHP	1.35	0.8	PHP	66	66
Java	40 sec	12.5 sec	Java	57	11.2	Java	44%	29%	Java	0.67	0.2	Java	108	108
C#	19 sec	7 sec	C#	15.2	2.8	C#	25%	24%	C#	0.32	0.12	C#	56	56
C++	38 sec	57 sec	C++	0.37	1	C++	25%	29%	C++	0.63	0.95	C++	109	109

Table 3: The comparison table shows the performance of each examined language for each issue on both platforms

Figure 20 - Overall Statistics of Memory Usage, Execution Time and CPU Consumption

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].

COMP4906 HONOURS THESIS: SKYDOT

FINAL REPORT OUTLINE

X

Alexandra Brown
Student

X

Dwight Deugo
Supervisor