

Electronic Thesis and Dissertation Repository

8-24-2021 10:30 AM

WesternAccelerator : Rapid Development of Microservices

Haoran Wei, *The University of Western Ontario*

Supervisor: Nazim H. Madhavji, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in
Computer Science

© Haoran Wei 2021

Follow this and additional works at: <https://ir.lib.uwo.ca/etd>



Part of the [Software Engineering Commons](#), and the [Systems Architecture Commons](#)

Recommended Citation

Wei, Haoran, "WesternAccelerator : Rapid Development of Microservices" (2021). *Electronic Thesis and Dissertation Repository*. 8117.

<https://ir.lib.uwo.ca/etd/8117>

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlsadmin@uwo.ca.

Abstract

Context & Motivation/problem: In the context that cloud platforms are widely adopted, Microservice Architecture (MSA) has quickly become the new paradigm for modern software development due to its great modularity, scalability, and resiliency, which fits well in the cloud environment. However, to embrace the benefits of MSA, organizations must overcome the challenges of adopting new methodologies and processes to deal with the extra development complexities that microservices created, e.g., establishing interface-based communication between distributed services and managing the configurations and locations of services. Consequently, creating a microservice-based application is relatively complex and effortful. **Research Question:** How to create a tool to automate the development of microservice-based applications? **Principal Ideas:** In this research, we created WesternAccelerator, a tool to generate extensible microservice-based skeleton applications. This tool embodies the design patterns for building an MSA and automates time-consuming development tasks. **Contribution:** Currently, numerous tools are available to assist in developing microservices, but most of them serve a particular purpose, such as configuration management or service discovery. Our solution is an improvement by merging technologies from different problem domains and provides the service in an automated manner. **Conclusion:** By empirical validation, we conclude that the tool we have created aids in building microservice-based applications relatively quickly and effortlessly.

Keywords

Microservice Architecture, Microservices Design Patterns, Microservices Development, Microservices Configuration Management, Service Discovery.

Summary for Lay Audience

Microservices architecture embodies elements and principles for structuring and developing a collection of software services as cloud-based applications. Microservices allow a large application to be divided into smaller independent parts, with each part having its own responsibility. Microservice architecture is a recent paradigm for modern software development. However, building microservices is challenging because it is complex to manage many small and distributed services regarding their configurations, network locations, and health status. This thesis proposes WesternAccelerator, a tool that simplifies the development of microservices through its capability of generating runnable microservice-based skeleton applications pre-fabricated with solutions for configuring, discovering, securing, and tracing services.

Acknowledgments

I am thankful to Professor Madhavji for research and technical guidance throughout my M.Sc. degree program. Also I am grateful to Mr. Steinbacher of IBM Canada for helping me to understand the problem context, solution opportunities, possibilities, and scenarios, and documents from the cloud project environment. I am particularly thankful to the Dept. of Computer Science for access to the computing facilities, and to the department, IBM Canada, and NSERC for research support.

Table of Contents

Abstract	ii
Summary for Lay Audience	iii
Acknowledgments	iv
Table of Contents	v
List of Tables	vii
List of Figures	viii
Glossary of Abbreviations	ix
Glossary of Terms	x
Chapter 1	1
1 Introduction	1
Chapter 2	4
2 Related Work	4
2.1 Microservices Design Patterns	4
2.1.1 Externalized Configuration Patterns	4
2.1.2 Service Discovery Patterns	5
2.1.3 API Gateway Patterns	6
2.1.4 Security Patterns	7
2.1.5 Load Balancing Patterns	7
2.1.6 Distributed Tracing Patterns	7
2.2 Commercial and Open-source Tools	8
2.3 Analysis	11
Chapter 3	13
3 Structure and Dynamics of the Generated Application	13
3.1 Architecture of the Generated Application	13

3.2 Run-time Dynamics of the Generated Application	16
Chapter 4	18
4 Implementation of the Generated Application	18
4.1 Implementation of the Infrastructure Components.....	18
4.2 Implementation of the Service.....	20
Chapter 5	21
5 Design of the WesternAccelerator Tool	21
5.1 Use Case Description of WesternAccelerator	21
5.2 Architectural Design of WesternAccelerator	22
5.3 Run-time Dynamics of WesternAccelerator	23
Chapter 6	26
6 Implementation of the WesternAccelerator Tool	26
Chapter 7	31
7 Validation of WesternAccelerator	31
Chapter 8	34
8 Discussion.....	34
8.1 Comparison of WesternAccelerator with related tools	34
8.2 Advantages and Limitations of WesternAccelerator.....	37
Chapter 9	39
9 Conclusion and Future work	39
References.....	40
Curriculum Vitae	44

List of Tables

Table 1 : Tools supporting microservices development	8
Table 2 : Tools integrated in the infrastructure components	19
Table 3 : Use case description of WesternAccelerator	21
Table 4 : Comparison of WesternAccelerator with IBM Accelerators, JHipster, and MAGMA.....	36

List of Figures

Figure 1 : High-level design of the generated application	14
Figure 2 : Component diagram of the generated application.....	15
Figure 3 : Sequence diagram of the generated application	17
Figure 4 : Architectural design of WesternAccelerator	23
Figure 5 : Activity diagram of WesternAccelerator	24
Figure 6 : Infrastructure Components Repository on GitHub	26
Figure 7 : Main class of the Service Discovery component	27
Figure 8 : Configuration of the Service Discovery component	28
Figure 9 : Configuration of the Gateway component	29
Figure 10 : Service Archetypes Repository on GitHub	29
Figure 11 : Scripts to generate a sample service from the archetype.....	30
Figure 12 : Feedback received from executing the WesternAccelerator	31
Figure 13 : Eureka Service Dashboard	32

Glossary of Abbreviations

MSA	Microservice Architecture
API	Application Programming Interface
BFF	Backends for Frontends
PEP	Policy Enforcement Point
HTTP	Hypertext Transfer Protocol
JWT	JSON Web Tokens
CNCF	Cloud Native Computing Foundation
REST	Representational State Transfer

Glossary of Terms

Microservice Architecture	An architectural style that structures an application as a collection of loosely coupled, independently deployable, and highly maintainable services
Service Discovery	Service Discovery is the process of locating microservices on a network.
API Gateway	API gateway is a service component that sits between a client and a collection of backend services acting as a reverse proxy to accept all application programming interface (API) calls, aggregate the various services required to fulfill them, and return the appropriate result.
Authentication	Authentication is the act of recognizing the identity of a computer system user.
Authorization	Authorization is the process of giving computer system users rights/privileges to access a specific resource or function.
Cross-cutting Concerns	Cross cutting concerns are technical requirements that are applicable throughout the application and it affects the entire application.
Policy Enforcement Point	Policy Enforcement Point is the point where a policy decision is used to grant or deny access to a protected resource.
Representational State Transfer	Representational State Transfer is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web

Chapter 1

1 Introduction

The emergence of cloud computing with its unique characteristics, such as on-demand self-service, rapid elasticity, and broad network access, demands a change in software architectural styles that take full advantage of the benefits provided by the cloud. The traditional monolithic architecture, a single-tiered software application in which all services are composed into a single code base, has become suboptimal (Leymann et al., 2017). For example, the monolithic architecture is hard to scale and does not contain fault isolation. In contrast, Microservice Architecture (MSA), as an architectural style that structures an application as a collection of loosely coupled, independently deployable, and highly maintainable services (Richardson, 2018), has emerged as a prime candidate for boosting the return for cloud adoption (Wu, 2017).

MSA has the following advantages compared with the monolithic architecture: high resilience, a single failure does not affect the whole system; the scaling process is more accessible because only the services that need actual scaling are scaled, contrary to a monolithic application requiring to be scaled as a whole unit; ease of deployment, each service can be deployed independently without affecting other services (Al-Debagy & Martinek, 2018). Moreover, in other research, the performance of the two types of architecture is compared. The results revealed that microservice-based applications outperforms monolithic applications with less response time when handling a large number of requests (Singh & Peddoju, 2019).

Although microservices solve certain problems, it is not a panacea. The recently released survey findings of microservices adoption among software engineers and systems architects reveal that 77% of respondents have adopted microservices. However, 56% of respondents cite complexity in one form or another ("Increased complexity" and "Complexity of managing many services") as the biggest impediment to microservices adoption (Loukides, 2020). While alleviating many of the issues inherent to monolith

applications, microservices also produce extra complexities, including but not limited to the following:

1. It is challenging to manage many services regarding their configurations, network locations, and health status.
2. Communication between the client and services is complex since services are independent, requests traveling between different modules need to be carefully routed.
3. Cross-cutting concerns in a distributed architecture, such as load balancing, authentication/authorization, log aggregation, and distributed tracing.

To mitigate this situation, we have created a tool (called WesternAccelerator) to automate the development of microservices by generating runnable and extensible microservice-based skeleton applications. The generated application embodies pre-fabricated infrastructure components that can be used to solve the microservices-specific challenges, including configuring, discovering, securing, routing, and tracing microservices. It also comprises editable microservices for application developers to implement their business features. With these capabilities, application developers can focus on their business logic rather than MSA design, infrastructure setup, and microservices configuration because their starting point is a feature-packed microservice-based application.

Research Contributions

The contribution of this thesis is the WesternAccelerator tool which, as described above, generates microservice-based skeleton applications (Chapter 5). As an integral part of the tool design, we analyzed a number of tools and made decisions for the best configuration of the skeleton application generated by the tool. The WesternAccelerator is comparatively better than the alternatives as discussed in Chapter 8.

Thesis Structure

The rest of the thesis is organized as follows: Chapter 2 summarizes the related works in microservices design patterns and reviews the available tools supporting microservice development. Chapter 3 presents the architecture design of the application generated by WesternAccelerator. The detailed implementation of the generated application is described in Chapter 4. We further present WesternAccelerator and explicate the design and implementation of WesternAccelerator in Chapter 5 and Chapter 6. Chapter 7 demonstrates the usage of our tool. Chapter 8 compares WesternAccelerator with related tools and discusses the advantages and limitations of WesternAccelerator. Finally, chapter 9 concludes our work and elaborates on future research directions.

Chapter 2

2 Related Work

Before developing a microservice-based application, it is essential to identify MSA-specific challenges and learn the common design patterns to solve them with reusable solutions. In this chapter, we review common microservices design patterns and available tools supporting microservices development.

2.1 Microservices Design Patterns

This section reviews the microservices design patterns, the formalized best practices to solve microservices-specific challenges. The patterns discussed in this section are the following: Externalized configuration patterns, the patterns for microservices configuration management; Service discovery patterns, the patterns for locating microservices in a network; API gateway patterns, the patterns for aggregating services and routing client requests; Security patterns, the patterns for service endpoints and resources protection; Load Balancing Patterns, the patterns for distributing workloads among services; Distributed tracing patterns, the patterns for transaction monitoring.

2.1.1 Externalized Configuration Patterns

Christian (2017) discussed the importance of configuration externalization. Completely separating the configurations from microservices and put them in a centralized repository allows developers to modify service configurations without searching for the configuration files from many code repositories. Moreover, with externalized configuration, we can enable a service to run in multiple environments without a single modification on it. A general solution is to separate the service configuration information from the physical deployment into a few repositories. This configuration information should be passed as environment variables to the starting service or retrieved from a centralized repository through a REST-based service when the service starts through a service interface by a REST-based request (Carnell, 2021).

2.1.2 Service Discovery Patterns

Services usually need to communicate with each other. In a monolithic application, services invoke one another through language-level methods or procedure calls. However, microservices typically run in containers, where the number and location of service instances are dynamic. Consequently, we must implement a mechanism that enables the service clients to discover a dynamically changing set of temporary service instances.

Richardson (2018) summarized the two common ways to implement service discovery:

1. **Application-level service discovery:** The service instances register their network locations with the service registry component. A service client invokes a service by first querying the service registry to find network locations of available service instances. It then sends a request to one of those instances.
2. **Infrastructure-provided service discovery:** The deployment platform gives each service a virtual IP address, and a DNS name that resolves to the address. A service client makes a request to the DNS name, and the deployment platform automatically routes the request to one of the available service instances. As a result, service registration, service discovery, and request routing are entirely handled by the deployment platform.

One benefit of application-level service discovery is that it handles the scenario when services are deployed on multiple platforms. For example, some of the services are deployed on Kubernetes, and the others are running in a virtual machine. Application-level service discovery works across both domains, whereas Kubernetes-based service discovery only works within Kubernetes. In this research, we focus on application-level service discovery.

Montesi and Weber (2016) reviewed the two most common design patterns for application-level service discovery. In the client-side service discovery pattern, service instances register their network locations with the service registry. The service client

invokes a service by first querying the service registry to obtain a list of service instances. It then uses a load-balancing algorithm, such as round-robin, to select a service instance.

In the alternative server-side service discovery pattern, when the client requests a service, the request is handled by a router that runs at a durable location. The router queries the service registry, which might be built inside the router, and forwards the request to an available service instance.

2.1.3 API Gateway Patterns

In MSA, the client apps usually need to call a chain of microservices. If the interaction mode between the client and microservices is direct access, the client then needs to manage a list of microservice endpoints. Furthermore, it is the client's responsibility to keep pace with the evolution of these microservices, e.g., update the list of service endpoints immediately when there is a change.

Therefore, having an intermediate component serving as a facade (gateway) to provide an abstraction over internal complexity and change can be convenient for the service clients.

Torre et al. (2020) described two API gateway design patterns: the single custom API gateway pattern and the backends for front-ends (BFF) gateway pattern. In the single custom API gateway pattern, all the client apps connect to a single API gateway exposed as an endpoint in front of the microservices. The gateway acts as a reverse proxy, routing requests from clients to the endpoints of the internal microservices. In such a way, the client apps will not be affected by any Uniform Resource Identifier (URI) change when the microservices are evolved or refactored. Another functionality the gateway can offer is request aggregation. The client only needs to send a single request to the gateway when it requires information from multiple microservices. The gateway will then gather information from the target microservices, aggregate the results, and send everything back. The essential advantage of this design is to reduce traffic load from the client app, which is especially beneficial when the requests are made from a remote location.

The BFF gateway pattern describes an approach that multiple gateways are created to serve different client types, such as one for website and one for mobile clients, to prevent the gateway from being bloated and overloaded.

2.1.4 Security Patterns

The most commonly used design pattern to implement authentication and authorization for microservices was introduced by Richardson (2018). In the proposed pattern, an authentication server is created and works in coordination with the API gateway. Clients trying to access the internal services must first log in to the system by posting its credentials to the gateway. The gateway validates the credentials with the authentication server and returns an access token containing the identity and role of the user in an encrypted format. The client then provides the access token in subsequent requests. Requests with valid access tokens will be forwarded to the target microservice by the gateway. Eventually, the target service verifies the token's signature and extracts information about the user, including their identity and roles. Only authenticated users who have the proper authorization can access the service.

2.1.5 Load Balancing Patterns

Carnell (2017) discussed several approaches to enable load balancing in the API gateway pattern. The simplest and most common way is to build a load balancer inside the service discovery agent, making it able to load balance the service instances it discovers. Then integrate the API gateway with the service discovery agent. Thus, the gateway can query the service discovery to get the already load-balanced instances.

2.1.6 Distributed Tracing Patterns

Tracking and logging user actions across multiple microservices are critical in a distributed architecture. It is challenging to implement these capabilities in each service consistently. However, since the gateway is the sole entry of the system, we can apply certain mechanisms to make it works as a single Policy Enforcement Point (PEP). One approach is to insert a unique correlation id to each external request that passes through the gateway. The correlation id will be forwarded to all services that are involved in

handling the request. Each service is responsible for managing the propagation of the correlation id to outbound service calls and logging the information about the request and operations performed when handling it (Carnell & Sánchez, 2021). A centralized logging service is needed to aggregate logs from each service instance. Otherwise, it is impossible for the system operator to collect this information from dozens of microservices (Richardson 2020).

2.2 Commercial and Open-source Tools

Various open-source and commercial tools are available to assist in building microservices, most of which serve a particular purpose such as configuration management, service discovery, service gateway, and distributed tracing. etcd is a light-weighted command-line driven tool for key-value management, usually used as the data backbone for Kubernetes and other distributed platforms (IBM Cloud Education, 2019). ZooKeeper is a high-performance service for maintaining configuration information, naming and grouping services (Apache, 2021). Consul provides a full-featured control plane with service discovery, configuration, and segmentation functionality (HashiCorp, n.d.). Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system (Spring, n.d.-b). Eureka (Netflix, 2012) is a service discovery tool that offers dynamic service status refresh. Other tools such as Spring Cloud Gateway (Spring, n.d.-b) and Zuul (Netflix, 2013) are used for building the API gateway. NGINX Plus (NGINX, Inc., 2020) serves a similar purpose, but it is not designed to work in conjunction with service discovery. Distributed tracing tools, such as Zipkin (Zipkin, n.d.) and Jaeger (Uber Technologies, n.d.) are almost identical in the features provided. They only differ in how the components are packaged. A general summary of these tools is presented in Table 1.

Table 1: Tools supporting microservices development

Configuration Management & Service Discovery Tools		
Tool	Developer	Description

etcd	CNCF	Open source distributed key-value store used to hold and manage the critical information that distributed systems need to keep running. Used for service discovery and key-value management.
ZooKeeper	Apache	A distributed key-value store typically used for distributed configuration management, can be used as the basis to implement service discovery. Mostly a common-purpose distributed key/value store used for service-discovery in conjunction.
Consul	Hashicorp	Provides a large set of features, including service discovery, integrated health checking, and distributed configuration.
Spring Cloud Config	Spring	Designed specifically for externalizing configuration in a distributed system.
Eureka	Netflix	A service discovery tool, the architecture is primarily client/server, with clients mainly using embedded SDK to register and discover services.

API Gateway Tools

Tool	Developer	Description
Spring Cloud Gateway	Spring	Used for building the API Gateway on top of Spring WebFlux, it provides cross-cutting concerns, such as security, monitoring/metrics, and resiliency.
Zuul	Netflix	An application gateway that provides capabilities for dynamic routing, monitoring, resiliency, and security.
NGINX Plus	NGINX	Used to authenticates API calls, routes requests to

appropriate backends, applies rate limits.

Distributed Tracing Tools

Tool	Developer	Description
Zipkin	Twitter	A <i>distributed tracing system</i> helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.
Jaeger	Uber	A distributed tracing system used for distributed context propagation, transaction monitoring, root cause analysis, etc.

Even though numerous tools are available to build a microservice application, it is never easy for developers to quickly acquire proficiency in these technologies. Furthermore, each of these tools is supposed to be used individually, serving different purposes. It is still the developer's responsibility to determine the appropriate tool to use in different situations and let them work in conjunction with minimum friction.

To mitigate this issue, IBM Cloud created a web-based tool called IBM Accelerators (Harris & Ziemann, 2020) that can automatically generate the required source code repositories in Git (Git, 2021) with scaffolded microservices, ready for deployment. The generated microservices are restricted to be deployed on OpenShift. JHipster (Dubois et al., 2013) is another development platform to generate, develop, and deploy microservice architectures. JHipster provisions both frontend and backend infrastructures in an automated manner. MAGMA (Wizenty et al., 2017) is a build management tool that relies on Maven Archetype mechanism (van Zyl, 2009). It can create microservices foundations based on predefined service templates.

2.3 Analysis

It takes a robust MSA and coordinated use of various tools to create a microservice-based application. In the entire development process, application developers usually need to overcome the following challenges:

1. Design an MSA that embodies solutions for the cross-cutting concerns in microservices such as configuration management, service discovery, authentication/authorization, etc.
2. Determine the proper tools to use to assist in the implementation of the designed architecture.
3. Implement the architecture components, establish connections between them, and set up the run-time configuration details for each service.

The three tools mentioned above (IBM Accelerators, JHipster, MAGMA) can automate these development tasks to a certain extent, but they have some drawbacks. The IBM Accelerators (Harris & Ziemann, 2020) relies on the cloud platform (OpenShift), which means the offered microservices features are platform-provided, not pre-fabricated in the generated application. Therefore, the generated application can only be deployed on OpenShift to realize the offered benefits such as service discovery and service health check.

JHipster (Dubois et al., 2013) is another tool for automating microservices development tasks. It builds and provides the architecture components in an abstracted and encapsulated manner. Consequently, these infrastructure components are easy to use but are difficult to extend or modify. For example, it encapsulates both service discovery and configuration management in a single infrastructure component and exposes the functionalities with self-defined interfaces, which is difficult to be modified since the underlying logic is not accessible.

MAGMA (Wizenty et al., 2017) gives the users complete control over the generated infrastructure components, but it provides limited features. For example, there is no

support for configuration management and distributed tracing. Moreover, some of the technologies used in MAGMA, such as Zuul and Ribbon (Netflix, 2014), are outdated.

In such a context, we aim to design an MSA that embodies the design solutions for all the challenges (discussed in Section 2.1) and automate the development tasks (implementing the architecture components, establishing connections between them, and setting up the runtime configuration details for microservices) to generate an easily extensible microservices skeleton application.

Chapter 3

3 Structure and Dynamics of the Generated Application

As discussed in Section 2.3, in order to develop a solid microservice-based application, we first need to design an MSA that embodies solutions for service discovery, configuration management, security, etc. The application generated by our tool should be based on such an architecture. Therefore, in this chapter, we define the structure and the run-time dynamics of the microservice-based application that our tool generates. The proposed MSA is our first research result.

3.1 Architecture of the Generated Application

This section defines the architecture of the generated application by first introducing the high-level design of the generated application and then explicating the application components with a component diagram. Figure 1 shows the general design of the generated application. Five infrastructure components (the red modules in Figure 1) are provided for use by the application developer. Each component has functionalities that are accessible to the developer through the component's interfaces. These components can work across different deployment platforms and even development frameworks because their functionalities are exposed through REST-based interfaces. For example, a microservice written in Python, deployed in a virtual machine, and a microservice written in Java, deployed in a container, can both access our components without distinction.

A number of microservices such as the SampleService (Figure 1) can also be generated on user demand. These generated microservices are service templates for application developers to fill in their own business logic. They are pre-configured to use the functions offered by the infrastructure components, e.g., registered to the service discovery and connected to the configuration service in advance. Furthermore, the generated services are embedded with load balancers similar to the Gateway. In actual operation, services usually need to invoke each other, which means a service can be a client of other services. However, there's no gateway sitting in between internal services,

therefore an embedded load balancer is needed to distribute the outbound traffic of the internal services.

With these capabilities, application developers no longer need to worry about the architectural solutions for the microservices-specific concerns and can focus on their business logic rather than infrastructure setup or application configurations because their starting point is a feature-packed runnable application.

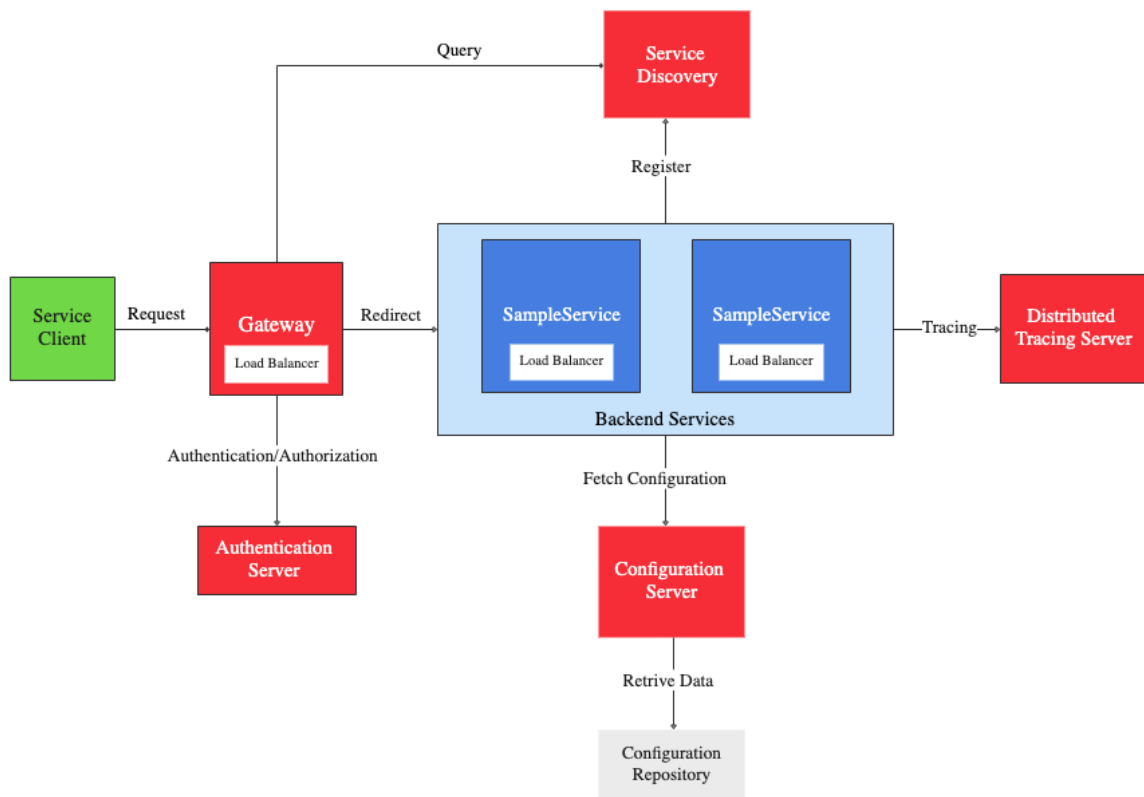


Figure 1: High-level design of the generated application

The structure of the generated application is illustrated in Figure 2, which describes the application components, their interfaces, and their dependencies. The detailed design of each infrastructure component is explicated below:

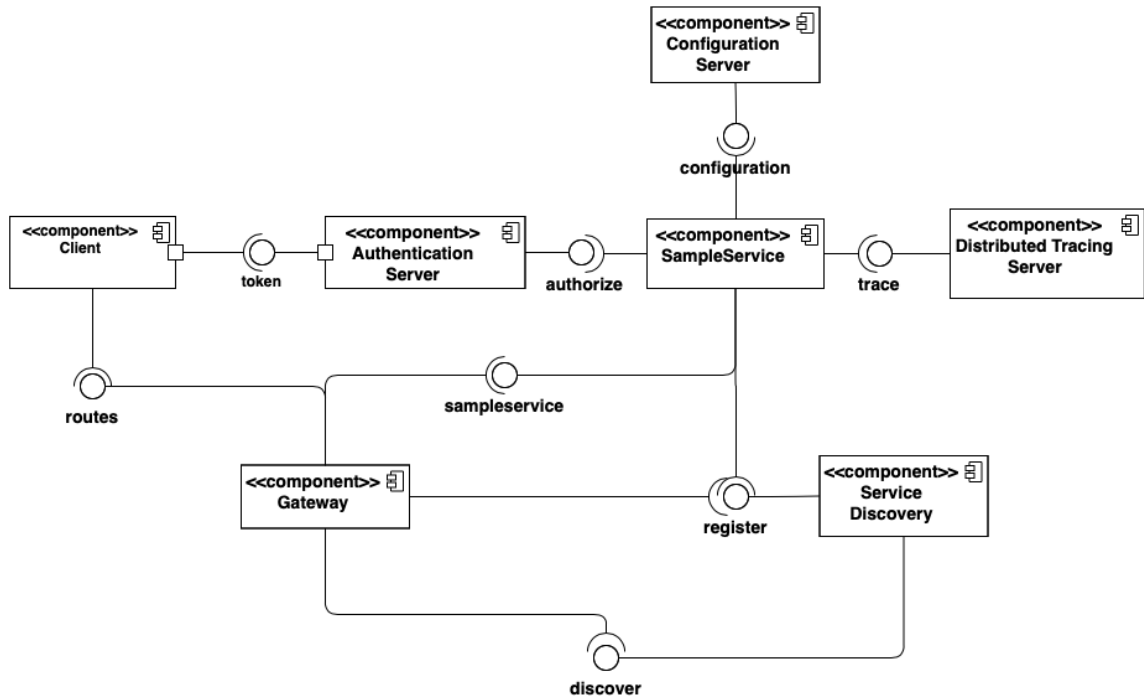


Figure 2: Component diagram of the generated application

Configuration Server: The configuration server utilizes the common distributed key-value store to manage the environment-specific information and application configuration information for each microservice. These pieces of information are persistently saved in an external repository connected to the configuration server. The Configuration Server provides an HTTP-based configuration interface that can be used by the SampleService for retrieving the configuration files.

Service Discovery: Service instances register themselves to the service discovery through the register interface (HTTP endpoint) provided by it. The service discovery agent monitors the health of each service instance registered with it and removes any failing service instances from its routing table by pinging these instances periodically. The Gateway uses the discover interface provided by the service discovery to discover the registered services.

Authentication Server: The authentication server acts as an identity provider (IdP) that creates, maintains, and manages the clients' identity information and authenticates them

by checking their credentials and issuing access tokens. The access token is an encrypted string representing the identity and role of the client that can be validated back to the authentication server. With such a scheme, the service clients can be authenticated and authorized by each microservice without having to present their credentials repeatedly to each microservice processing their request. It provides an HTTP-based interface for the client to authenticate themselves and get an access token and an authorization interface for the internal services to verify the token and authorize client operations.

Gateway: The gateway is the sole entry point to our entire architecture that sits between the client and the backend services. It acts as a reverse proxy to accept all client requests, aggregate the various services required to fulfill them and return the appropriate result. A load balancer is built into the gateway to load balance the incoming requests. It provides a routes HTTP endpoint accepting client requests and invokes the interfaces provided by the internal services (SampleService) to forward the requests.

Distributed Tracing Server: The distributed tracing server instruments incoming requests with correlation IDs by adding filters and interacting with other components such as the Gateway to let the generated correlation IDs pass through to all the service calls. It then utilizes a data visualization tool to show the flow of a transaction across multiple services. Every internal service (SampleService) should send the trace information to the distributed tracing server through the trace interface provided by the distributed tracing server.

3.2 Run-time Dynamics of the Generated Application

This section introduces how the generated application handles the run-time flows of information and how are the client requests fulfilled. The sequence of messages passed between the application components is shown in Figure 3.

During the startup of the services (SampleService), they fetch configuration from the Configuration Server and register themselves to the Service Discovery. To access these services, the service client first needs to authenticate with the Authentication Server to get an access token. Once the token is obtained, the client makes the subsequent request

to the Gateway carrying the access token. The Gateway then communicates with the service discovery to retrieve the locations of the target service instances and uses the built-in load balancer to select an instance and redirects the request to the specific instance. Once the request arrives at the target service, it validates the access token against the authentication server to see if the user has permission to continue the process. Once validated, the service processes the request and sends the results back to the client as an HTTP response.

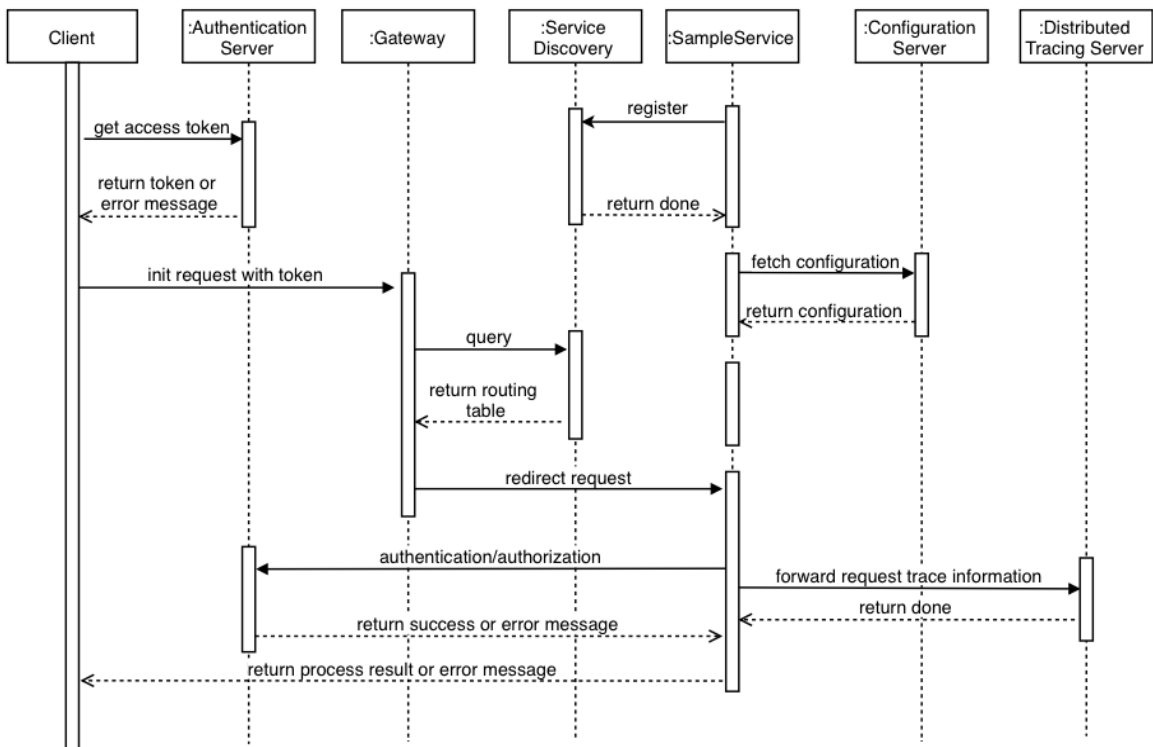


Figure 3: Sequence diagram of the generated application

Chapter 4

4 Implementation of the Generated Application

This chapter elaborates on implementing the microservice-based application in correspondence to the design presented in Chapter 3. This implementation consists of the decisions to select certain third-party tools to build the infrastructure components that are provided to the application developers. Therefore, this is our second research result.

The development framework used to build such an application is Spring Boot, the most widely used Java framework for creating Microservices (Spring, n.d.-a). How each infrastructure component and microservice is implemented is explained in more detail in the following sections.

4.1 Implementation of the Infrastructure Components

Third-party tools we depend on to build the infrastructure components are exhibited in Table 2. These tools are picked from the tools we outlined in Chapter 2.2. Eureka is used for implementing service discovery. Spring Cloud technologies including, Spring Cloud Config, Spring Cloud Gateway, Spring Cloud Load Balancer, and Spring Cloud Security are integrated into corresponding infrastructure components for configuration management, service gateway, load balancing, and authentication/authorization. Zipkin and Spring Cloud Sleuth is used for distributed tracing.

The Service Discovery is implemented by integrating Netflix Eureka in it. Netflix Eureka is an application-level service discovery solution because it provides a REST-based interface for microservices to register themselves with it. The main benefit of Netflix Eureka is that it offers dynamic client update and service health check out of the box. It monitors the microservices registered with it by periodically pinging them and refreshing its routing table dynamically (add newly registered service instances and remove failing service instances). All these benefits can be achieved with minimum setup costs.

The Configuration Server is implemented with the Spring Cloud Config. With Spring Cloud Config, the configuration data of the microservices can be completely separated

from the application code and stored in either a remote repository or a shared filesystem and injected into the microservices automatically at their startup. More importantly, since the Spring Cloud Config is designed explicitly for Spring Boot projects, it is easy to set up and use.

The Gateway is built on top of the Spring Cloud Gateway. Spring Cloud Gateway offers built-in filters to inspect and act on the requests and responses coming through the Gateway. It also provides built-in predicates, which allow us to check if the requests fulfill a set of given conditions before executing or processing them. Additionally, the Spring Cloud Load Balancer (Spring, n.d.-b) can be easily integrated into Spring Cloud Gateway to achieve load balancing since they are both developed by Spring Cloud. Alternative tools such as Zuul is in maintenance mode and deprecated (Gibb, 2018), and NGINX Plus is not an application-level solution for the API gateway.

The Authentication Server is built by integrating the Spring Security OAuth2 plugin (Spring, n.d.-c), a commonly used security solution that follows the token-based security protocols, OAuth2 and JSON Web Tokens (JWT). Spring Security OAuth2 plugin offers the function to register service clients with the Authentication Server and generate and issue JWT tokens to the clients. With Spring Security OAuth2, we can also easily integrate the services we want to protect with the Authentication Server.

The Distributed Tracing Server is implemented using Zipkin because it offers collection, storage, and visualization of tracing information in one process, and Spring Cloud Sleuth is used to instrument client requests with trace IDs that are used to track the transaction flow of the requests.

Table 2: Tools integrated in the infrastructure components

Infrastructure Components	Tools
Service Discovery	Netflix Eureka
Configuration Server	Spring Cloud Config

Gateway	Spring Cloud Gateway + Spring Cloud Load Balancer
Authentication Server	Spring Security OAuth2
Distributed Tracing Server	Zipkin + Spring Cloud Sleuth

4.2 Implementation of the Service

The generated service (SampleService in Figure 1) is a REST-based web service built using Spring Boot and Maven (Porter, n.d.) with embedded load balancers implemented with Spring Cloud Load Balancer. It is registered with the Service Discovery, connected to the Configuration Server, and pointed to the Authentication Server and Distributed Tracing Server by either setting the Spring Annotations or configuring the Application Properties (Spring n.d.-a).

Chapter 5

5 Design of the WesternAccelerator Tool

To generate an application described in Chapter 4 quickly and allow application developers to modify or extend the generated application easily, we built the tool WesternAccelerator. This is another of our research results.

5.1 Use Case Description of WesternAccelerator

We identify the use case of WesternAccelerator as generating runnable microservice-based skeleton applications that contain all necessary infrastructure components and editable service templates (as described in Chapter 3). Table 3 shows the detailed use case description depicting the standard operations of the tool.

Table 3: Use case description of WesternAccelerator

Use Case Name	Generate a microservice-based application
Use Case Description	To generate a microservice-based skeleton application that can further be used as the foundation to create a complex microservice-based system.
Actors	Microservice developers
Trigger	User executes the command to generate an application.
Pre-conditions	WesternAccelerator is available. Git, Java, and Maven are installed.
Flow of Events	<ol style="list-style-type: none"> 1. System generates the infrastructure components. 2. System prompts the user to enter service metadata. 3. System generates the services. 4. System returns the generated application.

Post-conditions	The generated application saved in the user's local filesystem.
-----------------	---

5.2 Architectural Design of WesternAccelerator

There are two function modules in WesternAccelerator, the Infrastructure Component Supplier and the Service Generator as shown in Figure 4. The two modules are explained in detail below.

The Infrastructure Component Supplier provides ready-to-use infrastructure components (the five infrastructure components discussed in Chapter 3.1.1) to application developers. It fetches these components from the Infrastructure Components Repository and returns them to the application developers on their demand. As the developer and maintainer of WesternAccelerator, we build and set up the infrastructure components in advance and prefill them into the Infrastructure Components Repository. Thus, the application developer can use the retrieved components directly without further configuration, and they can easily modify these components since they get the complete code.

The Service Generator generates customizable microservices that are pre-connected to the infrastructure components. It achieves such function by building the same kind of services from the service archetype fetched from the Service Archetype Repository.

A service archetype is an abstraction of the same kind of services. It standardizes the initial setups, including the configuration of common dependencies, infrastructure setup for communication with other components, and initial security configuration for authentication/authorization. The archetype can be instantiated into a concrete parameterized microservice. We build such an archetype adhering to the SampleService we defined in Section 3.1 and put it into the Service Archetypes Repository for use.

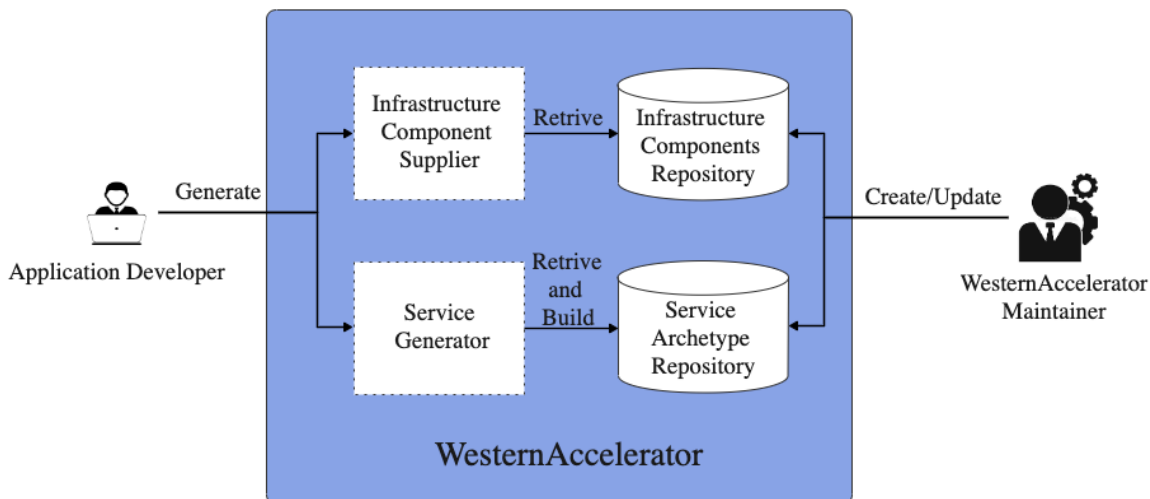


Figure 4: Architectural design of WesternAccelerator

5.3 Run-time Dynamics of WesternAccelerator

This sections discusses the the run-time behaviors of the tool, with a focus on the workflows of actions performed by the system. Figure 5 shows the activity diagram of the system.

In WesternAccelerator, a complete generation process is divided into two concurrent processes. The infrastructure components and services are supposed to be generated separately. The source code of the infrastructure components is fetched from the Infrastructure Component Repository and returned to the user. All the five infrastructure components are returned by default for the completeness of a microservice architecture.

The process for generating services requires the user to enter the service metadata, such as the group name, project name, and version number, to build a customized microservice. As it should be, such a process can be executed repeatedly to generate multiple services, because a microservice-based application typically contains many microservices.

Finally, the infrastructure components and services will be placed in the same directory of the local file system.

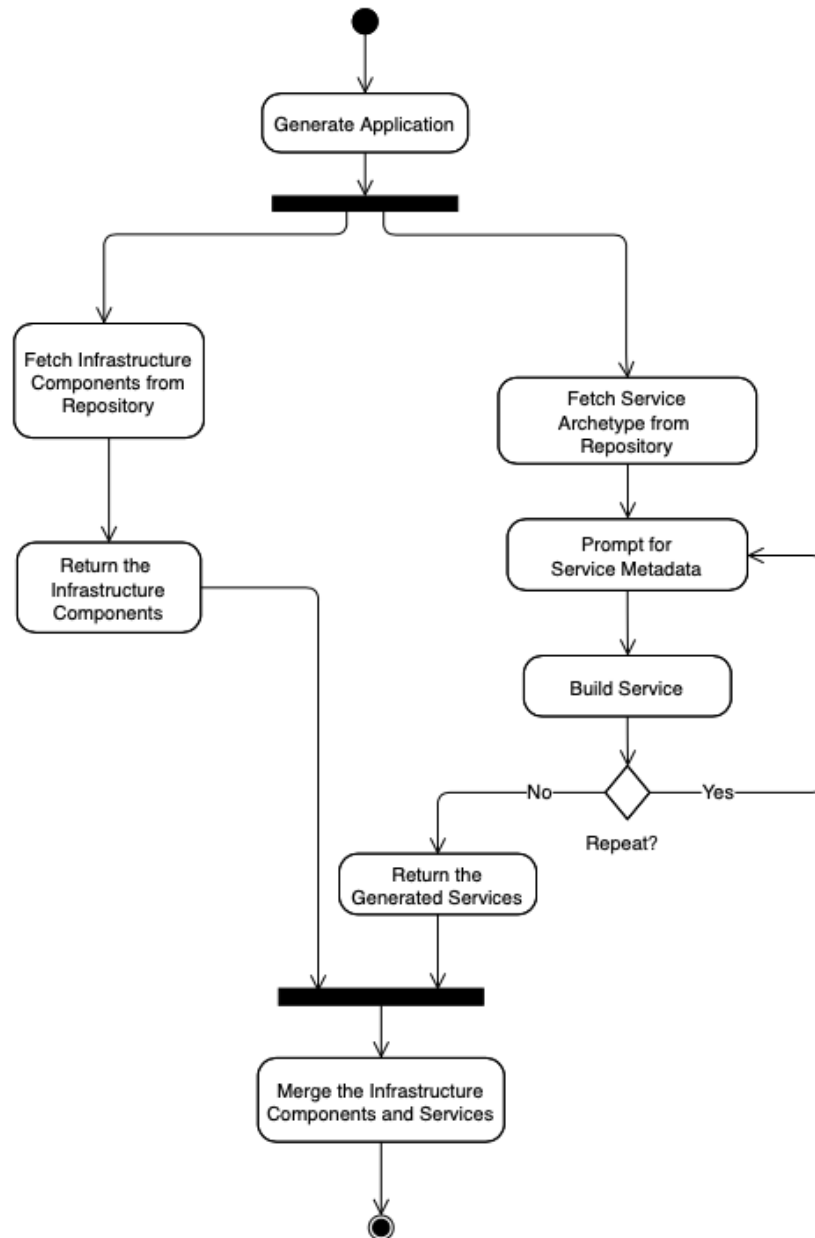


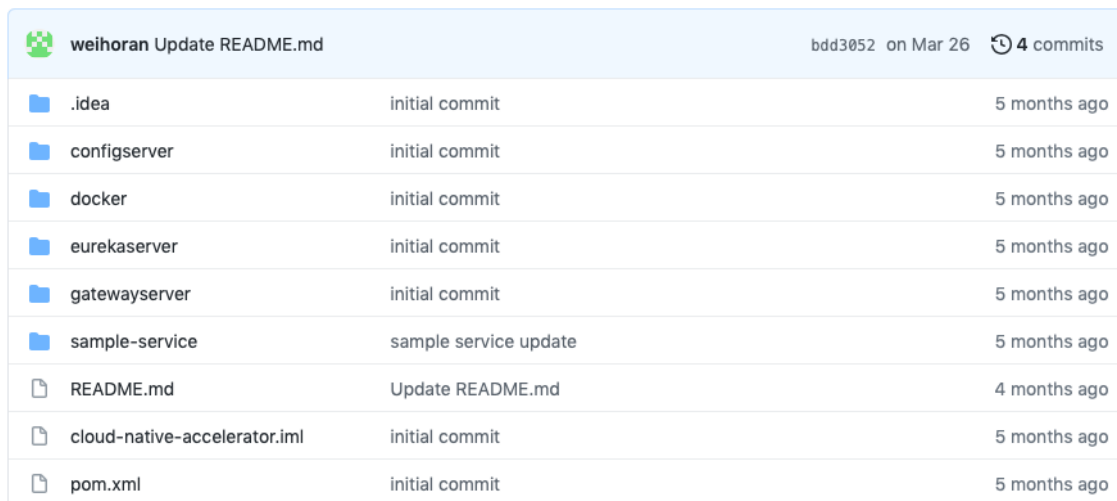
Figure 5: Activity diagram of WesternAccelerator

Chapter 6

6 Implementation of the WesternAccelerator Tool

WesternAccelerator is command-line driven and developed with shell scripts. The tools and technologies that each part of the WesternAccelerator relies on are introduced below.

The Infrastructure Components Repository is created with Git and hosted on GitHub. We use GitHub to manage these repositories because it is the most advanced version control and source code management platform (GitHub, 2007). The structure of the Infrastructure Components Repository is shown in Figure 6. The Infrastructure Component Supplier uses Git commands (Git, 2021), e.g., `git clone`, to fetch the source code of the infrastructure components from Github. It then returns the source code to the application developers on their demand.



weihoran Update README.md		bdd3052 on Mar 26	🕒 4 commits
📁 .idea	initial commit		5 months ago
📁 configserver	initial commit		5 months ago
📁 docker	initial commit		5 months ago
📁 eurekaserver	initial commit		5 months ago
📁 gatewayserver	initial commit		5 months ago
📁 sample-service	sample service update		5 months ago
📄 README.md	Update README.md		4 months ago
📄 cloud-native-accelerator.iml	initial commit		5 months ago
📄 pom.xml	initial commit		5 months ago

Figure 6: Infrastructure Components Repository on GitHub

The returned infrastructure components can be used directly without further setup or configuration. However, our users (application developers) get the complete code and configuration of these components, and they can extend or modify them if needed.

These components are individually runnable servers whose capabilities are accessible through HTTP endpoints. Figure 7 shows the main class of the generated Service

Discovery component (Figure 1) as an example. The "@EnableEurekaServer" annotation denotes that the Eureka tool is integrated into the component. And thanks to the power of the Spring Boot framework, the service discovery capabilities are automatically included with the implementation of the HTTP endpoints abstracted out.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class ServiceDiscoveryApplication {

    public static void main(String [] args) {
        SpringApplication.run(ServiceDiscoveryApplication.class, args);
    }

}
```

Figure 7: Main class of the Service Discovery component

We format the configuration data of the infrastructure components using YAML Ain't Markup Language (YAML). Figure 8 shows the detailed configuration of the Service Discovery component. The default listening port of the component is set to 8070. The parameter "defaultZone" in Figure 8 provides the service URL to access the Service Discovery component. Our users can add extra configurations to change the behavior of the component, such as setting the time to wait before the server takes requests with the parameter "waitTimeInMsWhenSyncEmpty".

```
spring:
  application:
    name: ServiceDiscovery
  boot:
    admin:
      context-path: /admin
server:
  port: 8070
eureka:
  instance:
    hostname: ServiceDiscovery
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
server:
  waitTimeInMsWhenSyncEmpty: 5
```

Figure 8: Configuration of the Service Discovery component

According to the design of the generated application discussed in Chapter 3, the infrastructure components and services are pre-connected to form a runnable skeleton application. We achieve this by adding extra configurations to establish dependencies between components. Figure 9 shows how the Gateway is connected to the Service Discovery as an example. First, the Gateway is registered with the Service Discovery by specifying "registerWithEureka: true", as shown in Figure 9. And "discovery.locator.enabled: true" enables the Gateway to create routes and redirect client requests based on services registered with the Service Discovery.

```

server :
  port: 8072

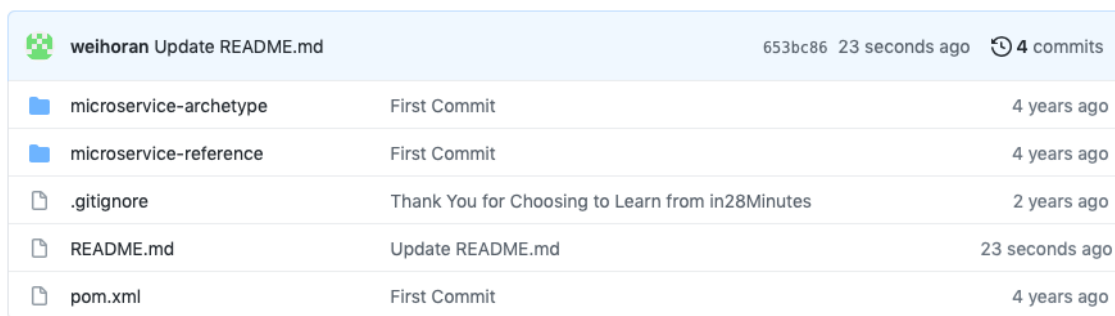
eureka :
  instance :
    preferIpAddress: true
  client :
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl :
      defaultZone: http://eureka-server:8070/eureka/

spring :
  cloud :
    gateway :
      discovery.locator :
        enabled: true
        lowerCaseServiceId: true

```

Figure 9: Configuration of the Gateway component

The Service Archetype Repository is also created with Git and hosted on GitHub, where the source code of the archetype we used to generate microservices are saved, as shown in Figure 10.



Commit Message	Timestamp
653bc86 Update README.md	23 seconds ago
microservice-archetype	First Commit
microservice-reference	First Commit
.gitignore	Thank You for Choosing to Learn from in28Minutes
README.md	Update README.md
pom.xml	First Commit

Figure 10: Service Archetypes Repository on GitHub

We build the service archetype with Maven Archetype (van Zyl, 2009). Maven Archetype is a project templating toolkit that allows us to build archetypes and generate concrete projects from the archetypes. It provides a variety of functions to allow us to set the properties of a project during generation, configure which resources will be copied into the generated project, and generate projects with multiple modules. The Service

Generator retrieves the archetypes from Github with Git commands and builds them into runnable microservices with the project generating capability provided by Maven Archetype. Figure 11 shows a key portion of the scripts to generate a sample service from the service archetype. The "mvn archetype :generate" is the central command to create a service project. The parameters such as DgroupId, DartifactId, and Dmicroservice-name in Figure 11, are supposed to be replaced with the user's inputs.

```
mvn install archetype:update-local-catalog
mvn archetype:crawl
mvn archetype:generate -DarchetypeCatalog=local
  mvn archetype:generate \
    -DarchetypeGroupId=com.archetypes \
    -DarchetypeArtifactId=archetype-sample-service \
    -DarchetypeVersion=0.0.1-SNAPSHOT \
    -DgroupId=com.example \
    -DartifactId=sample-service \
    -Dversion=0.0.1-SNAPSHOT \
    -Dmicroservice-name=SampleService
```

Figure 11: Scripts to generate a sample service from the archetype

Chapter 7

7 Validation of WesternAccelerator

We validate WesternAccelerator by using it to generate an application basing on the use case description discussed in Section 5.1 and test whether each part of the generated application works as expected.

Figure 12 shows the feedback received from executing the WesternAccelerator to generate an application. We can conclude that the generated application is successfully downloaded (i.e., receiving objects: 100% in Figure 12) to the local filesystem.

```
Haorans-MacBook-Air:desktop horan$ ./WesternAccelerator.sh infrastructures
Cloning into 'WesternAccelerator'...
remote: Enumerating objects: 493, done.
remote: Counting objects: 100% (493/493), done.
remote: Compressing objects: 100% (131/131), done.
remote: Total 493 (delta 337), reused 485 (delta 332), pack-reused 0
Receiving objects: 100% (493/493), 130.55 KiB | 395.00 KiB/s, done.
Resolving deltas: 100% (337/337), done.
```

Figure 12: Feedback received from executing the WesternAccelerator

We then build and run the generated application to verify that the components are successfully configured and connected with each other. We do this by checking the application monitor dashboard provided by Eureka. The dashboard displayed in Figure 13 reveals that the service discovery component accepted Gateway and the generated sample service (i.e., GATEWAY-SERVER, SAMPLE-SERVICE are under the Instances currently registered with Eureka).

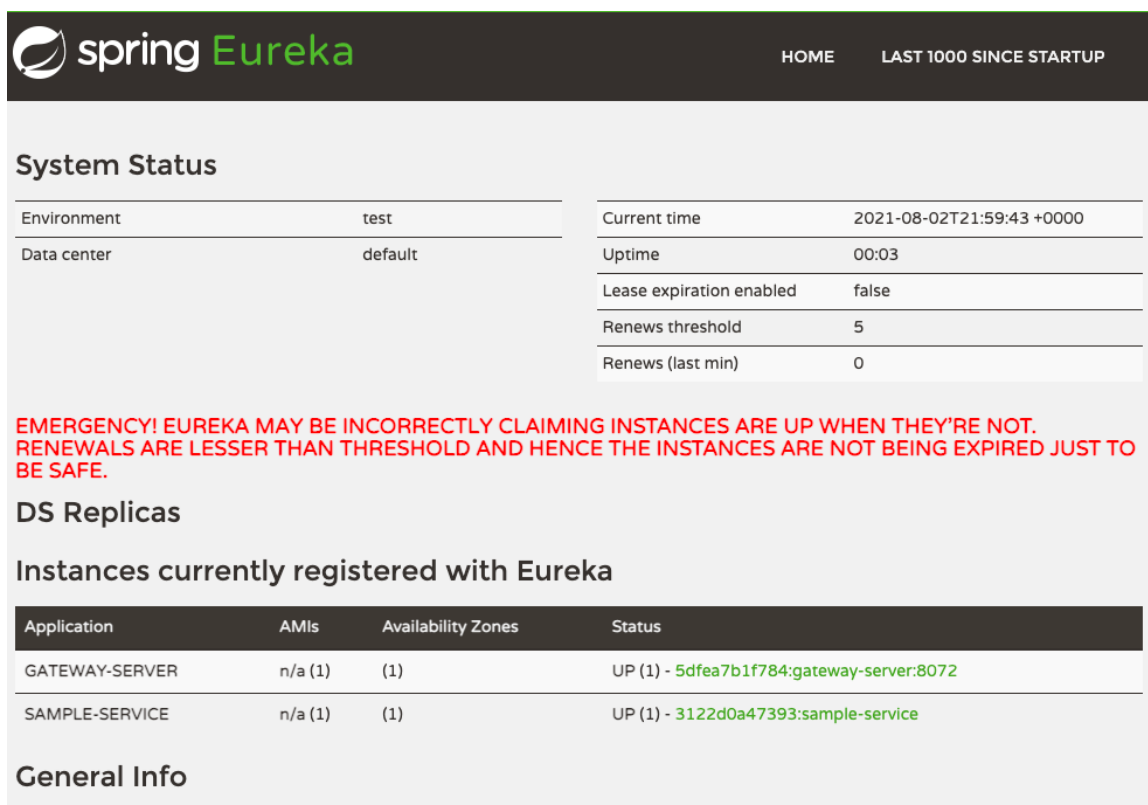


Figure 13: Eureka Service Dashboard

Based on the above two preliminary test cases and results, we can conclude that the WesternAccelerator tool is operational and generates the infrastructure components and services (see Figure 1) through a simple command-line invocation of the tool.

However, our test for the Service Generator is preliminary. We have only verified that the Gateway, the Service Discovery, and the sample service are runnable and connected.

However, a more detailed test should be conducted to check whether the generated sample service and all the five generated infrastructure components (presented in Figure 1) are correctly configured and connected. Such a detailed test would include:

1. Create a sizable and complex end-user application from the generated skeleton application by filling in some simulative business logic.
2. Invoke the internal service endpoints through the Gateway to validate its routing capabilities.

3. Perform a set of client requests with correct and illegal user credentials to validate the Authentication Server.
4. Update the configuration of each application component from the Configuration Server and observe whether the components can capture such change by re-running each component.
5. Check the Zipkin dashboard provided by the Distributed Tracing Service to confirm that the trace information is successfully collected.

This detailed test is outside the scope of this thesis because of the lack of time.

Chapter 8

8 Discussion

This chapter compares WesternAccelerator with competitive tools in Section 8.1 and discusses the advantages and limitations of WesternAccelerator in Section 8.2.

8.1 Comparison of WesternAccelerator with related tools

In this section, we compare WesternAccelerator with the IBM Accelerators, JHipster, and MAGMA. Table 4 shows an overview of the differences between the four tools, focusing on how the microservices features are implemented. WesternAccelerator is compared with each of the other tools in more detail in the following paragraphs.

WesternAccelerator vs. IBM Accelerators

WesternAccelerator and IBM Accelerators are both used for generating runnable microservice-based applications. The main difference between them is that WesternAccelerator is an application-level solution, whereas IBM Accelerator is more of a DevOps Solution (it automates both the development and deployment of microservices). WesternAccelerator implements the microservices required features within the scope of application code. Contrastingly, the MSA features offered by IBM Accelerators depend on the cloud platform, which is OpenShift (Red Hat, 2011). Even though the microservices generated by the IBM Accelerators can be deployed on OpenShift automatically, such automation comes at a cost because the features offered by IBM Accelerators, such as service discovery and service health check, can only be realized when deployed on OpenShift. Compared with the IBM Accelerators, our tool enables more flexibility in deployment by allowing application developers to choose their preferred cloud platforms to deploy the generated application. Our solution should work across different cloud platforms because we implement the microservices features as components at the application level. These infrastructure components are platform-independent since they communicate with each other through HTTP-based interfaces.

Besides automation of deployment, IBM Accelerators as a commercial tool offers a powerful graphical user interface where users can create microservices and establish relations between them by dragging and connecting components on a canvas. Such functionality is beyond the scope of our work.

WesternAccelerator vs. MAGMA

MAGMA is an application-level solution for generating pre-configured, runnable microservices. However, it does not provide enough infrastructure components to address several critical cross-cutting concerns in microservices, such as configuration management, distributed tracing, and log aggregation. Moreover, compared with WesternAccelerator, some of the technologies used in MAGMA are outdated. The Spring Cloud Netflix tools that MAGMA relies on to build the infrastructure components, including Zuul, Ribbon, and Hystrix (Spring. (n.d.-b)), are in maintenance mode and deprecated. There will not be any new features added to these tools, and the Spring Cloud team will perform only some bug fixes and fix security issues. Since MAGMA and WesternAccelerator are both heavily dependent on the Spring Cloud, it becomes important to use the latest Spring Cloud technologies.

WesternAccelerator vs. JHipster

JHipster is the most feature-rich tool available for developing microservices. Besides providing application-level infrastructure components and generating runnable microservices, it also supports the modeling of service entities through a visual web form that can help implement business features.

However, similar to MAGMA, some tools JHipster uses to implement the infrastructure components such as Zuul for the API gateway and Ribbon for load balancing are deprecated. Additionally, the infrastructure components provided by JHipster are hard to be extended or modified because it abstracts the underlying implementation of the provisioned infrastructures. For example, it combines service discovery and configuration management in a single component and exposes the functionalities with self-defined interfaces. Consequently, a full-fledged application generated by JHipster might be hard

to extend with other non-generated components. It takes extra effort for application developers to change the configuration or code of a non-generated component to connect it to the generated application. In contrast, the WesternAccelerator generated components can be easily extended or modified owing to the fact that they are developed strictly following the standard approaches demonstrated in the Spring Cloud reference documentation (Spring, 2020), which means any project developed with Spring Boot can integrate with our generated application smoothly. However, a detailed test on extending each tool's generated application and comparing the overall time and effort is yet to be conducted.

Table 4: Comparison of WesternAccelerator with IBM Accelerators, JHipster, and MAGMA

Features	Tools			
	WesternAccelerator	IBM Accelerators	JHipster	MAGMA
API Gateway	Spring Cloud Gateway	/	Netflix Zuul	Netflix Zuul
Load Balancing	Spring Cloud Load Balancer	OpenShift	Netflix Ribbon	Netflix Ribbon
Authentication/ Authorization	Spring Security OAuth2	/	JHipster UAA Server	Spring Security OAuth2
Service Discovery	Netflix Eureka	OpenShift	Netflix Eureka	Netflix Eureka
Configuration Management	Spring Cloud Config	/	Spring Cloud Config	/
Distributed	Zipkin	OpenShift	JHipster	/

Tracing			Console	
Log Aggregation	/	OpenShift	JHipster Console	/
Service Modeling		IBM Accelerators UI	JHipster UML	/
Easy to extend	+	-	-	+
Easy to use	+	0	0	0
capabilities	0	-	+	-

Legend:

+: the tool positively supports the quality attribute.

0: the tool's support for the quality attribute is neither weak nor strong.

-: the tool's support for the quality attribute is weak.

/: the tool doesn't support the feature.

8.2 Advantages and Limitations of WesternAccelerator

WesternAccelerator has several advantages over the other tools we discussed in the previous section. First of all, compared with the related tools, WesternAccelerator is easier to use because to generate an application with the tool, users only need to execute a single command. The other tools either require multiple commands or multiple actions in the graphical interface to generate an application. And as discussed in Section 8.1, the application generated by WesternAccelerator is easier to extend. Most of the features offered by IBM Accelerators depend on OpenShift, whereas the other three tools provide application-level implementations of the microservices features (we have discussed the benefits of application-level solutions in Section 6.1.). Compared with JHipster and MAGMA that also implement the microservices features at the application level, the main advantage of WesternAccelerator is that it uses the latest technologies to implement these features. JHipster and MAGMA both use Zuul for the API gateway and Ribbon for

load balancing. As we have mentioned in Section 6.1, these tools are deprecated by the Spring Cloud team. WesternAccelerator uses Spring Cloud Gateway and Spring Cloud Load Balancer, which are the successors of Zuul and Ribbon.

Despite the advantages mentioned above, our tool has some limitations. For example, it does not support aggregating, storing, and visualizing the log data for the microservices. Also, it lacks the capability of establishing dependencies between generated microservices, e.g., enabling application developers to configure the hierarchy of the microservices before generating them. Additionally, it does not help application developers implement business features, for example, generating entity classes in the microservices from class diagrams.

Chapter 9

9 Conclusion and Future work

In this thesis, we discuss the characteristics of microservices and review the common microservices design patterns. These design patterns are the theoretical baselines for our proposed MSA shown in Chapter 3. We further presented WesternAccelerator in Chapter 5, a tool to generate microservice-based skeleton applications comprising functional infrastructure components and runnable service templates (see Section 3.1 for the definitions of both these), in accordance with our proposed MSA. The generated infrastructure components are application-level solutions for configuring, discovering, securing, and tracing services. The service templates are user-customized microservice starter projects where business logic can be added. The thesis has shown that with these capabilities, application developers can focus on their business logic rather than architecture design, infrastructure setup, application configuration, and service templates. This benefit is derived from a generation capability of the WesternAccelerator (see Chapter 6 for the validation of WesternAccelerator).

As future work, we first need to conduct a more thorough test on the tool, as discussed in Chapter 7. And we plan to expand the WesternAccelerator functionalities in two directions. First, extend the infrastructure components to support more MSA features such as log aggregation and visualization. Second, according to Cloud Native Computing Foundation (CNCF) (2020), the use of containers in production has increased by 300% since 2016. Containers can further improve the scalability and availability of microservices. So we plan to integrate support for containerizing microservices and automating the deployment process of the container-based microservices.

References

- Al-Debagy, O., & Martinek, P. (2018, November). A Comparative Review of Microservices and Monolithic Architectures. *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. <https://doi.org/10.1109/cinti.2018.8928192>
- Apache. (2009, August 26). Maven – *Introduction to Archetypes*. <https://maven.apache.org/guides/introduction/introduction-to-archetypes.html>
- Apache. (2021, March 27). *ZooKeeper: Because Coordinating Distributed Systems is a Zoo*. Zookeeper.Apache. <https://zookeeper.apache.org/doc/r3.7.0/index.html>
- Carnell, J. (2017). *Spring Microservices in Action* (1st ed.). Manning Publications.
- Carnell, J., & Sánchez, H. I. (2021). *Spring Microservices in Action, Second Edition* (2nd ed.). Manning Publications.
- Cloud Native Computing Foundation (CNCF). (2020). *CNCF Survey 2020*. CNCF. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf
- Dubois, J., Sasidharan, D., & Grimaud, P. (2013). *Doing microservices with JHipster*. JHipster. <https://www.jhipster.tech/microservices-architecture/>
- Gibb, S. (2018, December 12). Spring Cloud Greenwich.RC1 available now. Spring. <https://spring.io/blog/2018/12/12/spring-cloud-greenwich-rc1-available-now>
- Git (2.32.0). (2021). [Distributed version control system]. Git. <https://git-scm.com>
- GitHub. (2007). GitHub: Where the world builds software. <https://github.com/>
- Harris, D., & Ziemann, N. (2020, June 12). *Introduction to accelerators for cloud-native solutions*. IBM Developer.

<https://developer.ibm.com/articles/introduction-to-accelerators-for-cloud-native-solutions/>

HashiCorp. (n.d.). *Introduction to Consul*. Consul.Io. Retrieved July 6, 2021, from <https://www.consul.io/docs/intro>

IBM Cloud Education. (2019, December 18). *etcd*. IBM Cloud. <https://www.ibm.com/cloud/learn/etcd>

Leymann, F., Breitenbücher, U., Wagner, S., & Wettinger, J. (2017). Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation. *Cloud Computing and Services Science*, 16–40. https://doi.org/10.1007/978-3-319-62594-2_2

Loukides, M. S. S. (2020, July 15). *Microservices Adoption in 2020*. O’Reilly Media. <https://www.oreilly.com/radar/microservices-adoption-in-2020/>

Netflix. (2012, September 4). *Netflix Shares Cloud Load Balancing And Failover Tool: Eureka!* Medium. <https://netflixtechblog.com/netflix-shares-cloud-load-balancing-and-failover-tool-eureka-c10647ef95e5>

Netflix. (2013, June 12). *Announcing Zuul: Edge Service in the Cloud - Netflix TechBlog*. Medium. <https://netflixtechblog.com/announcing-zuul-edge-service-in-the-cloud-ab3af5be08ee>

NGINX, Inc. (2020, November 2). *Plus Feature: API Gateway*. NGINX. <https://www.nginx.com/products/nginx/api-gateway/>

Porter, B. (n.d.). Maven – Welcome to Apache Maven. Apache. Retrieved August 30, 2021, from <https://maven.apache.org>

Red Hat. (2011). Red Hat OpenShift, the open hybrid cloud platform built on Kubernetes. OpenShift. <https://www.openshift.com>

- Richardson, C. (2018). *Microservices Patterns: With examples in Java* (1st ed.). Manning Publications.
- Richardson, C. (2020). *A pattern language for microservices*. Microservices.Io.
<https://microservices.io/patterns/>
- Singh, V., & Peddoju, S. K. (2017, May). Container-based microservice architecture for cloud applications. *2017 International Conference on Computing, Communication and Automation (ICCCA)*.
<https://doi.org/10.1109/ccaa.2017.8229914>
- Sorgalla, J., Wizenty, P., Rademacher, F., Sachweh, S., & Zündorf, A. (2018, September). AjiL. *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. <https://doi.org/10.1145/3241403.3241406>
- Spring. (n.d.-a). *Spring Boot*. Retrieved July 13, 2021, from <https://spring.io/projects/spring-boot>
- Spring. (n.d.-b). *Spring Cloud*. Retrieved July 13, 2021, from <https://spring.io/projects/spring-cloud>
- Spring. (n.d.-c). *Spring Boot and OAuth2*. Retrieved August 2, 2021, from <https://spring.io/guides/tutorials/spring-boot-oauth2/>
- Spring. (2020). *Spring Cloud Reference Documentation*.
<https://docs.spring.io/spring-cloud/docs/current/reference/html/>
- Torre, C., Wagner, B., & Rousos, M. (2020). *.NET Microservices: Architecture for Containerized .NET Applications* (5.0 ed.) [E-book]. Microsoft Developer Division, .NET and Visual Studio product teams.
- Uber Technologies. (n.d.). *Jaeger: open source, end-to-end distributed tracing*. Jaegertracing. Retrieved July 18, 2021, from <https://www.jaegertracing.io>

- van Zyl, J. (2009, August 26). Maven – Introduction to Archetypes. Maven.
<https://maven.apache.org/guides/introduction/introduction-to-archetypes.html>
- Wizenty, P., Sorgalla, J., Rademacher, F., & Sachweh, S. (2017, September). MAGMA. *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*. <https://doi.org/10.1145/3129790.3129821>
- Wu, A. (2017). *Taking the Cloud-Native Approach with Microservices*. Cloud Google.
<https://cloud.google.com/files/Cloud-native-approach-with-microservices.pdf>
- Zipkin. (n.d.). *OpenZipkin · A distributed tracing system*. Retrieved July 18, 2021, from <https://zipkin.io>

Curriculum Vitae

Name: Haoran Wei

Post-secondary Education and Degrees: The Chinese University of Hong Kong
Hong Kong, China
2014-2018 B.A.

The University of Western Ontario
London, Ontario, Canada
2019-2021 M.A.

Related Work Experience: Teaching Assistant
The University of Western Ontario
2019-2021