

# Scootr Studio:

Serverless on Wheels



By Braden Hitchcock

## **Final Master's Project Report**

Submitted to

Oregon State University

In partial fulfillment of  
the requirements for the degree of  
Master of Science in Computer Science

Presented 17 March 2020  
Commencement March 2020

# Abstract

Significant increases in the amount of data being streamed, collected, and processed have resulted in widespread adoption of the use of microservices to build scalable software applications. Unfortunately, current tools and frameworks are often insufficient at providing a simple, unified experience for the design, development, and deployment of microservices. They also have the tendency to be overly-complicated, resource intensive, and vendor-specific. This Master's Project Report introduces Scootr Studio, the first in a class of next-generation integrated development environments (IDEs) for microservice-based applications. Scootr Studio unifies the design, development, and deployment of microservice-based applications through the use of a small set of abstractions. These abstractions create an Event-Driven Application Architecture Model (EDAAM) that minimizes hosting-provider dependence. A laboratory study showed that software developers are able to build microservice-based applications 4.4 times faster using Scootr Studio than their existing IDE of choice. These users' experiences also illustrate how the use of Scootr Studio eliminates common errors encountered during deployment of microservice-based applications. Participants in the laboratory study also completed a usability survey, where Scootr Studio was given favorable ratings for its learnability, memorability, efficiency, and error rate reduction. These promising results serve as a starting point for creating tools that provide a simpler, more holistic development experience for agile software development teams looking to properly utilize microservices in their systems.

# 1 - Introduction

Microservices are mainstream [NGINX, 2016][Ford, 2018]. Significant increases in the amount of data being streamed, collected, and processed have resulted in a greater need for building scalable software applications that provide high-availability and consistency. In a recent survey conducted by Red Hat, a world leader in enterprise open source solutions, 69 percent of respondents say that they use microservices to re-architect their existing applications as much as they use them to build their brand new products [Red Hat, 2018]. This widespread adoption of microservices for providing services to customers has sparked the creation of numerous tools and frameworks aimed at easing the process of microservice-based application development.

Unfortunately, 99 percent of individuals adopting microservices report that there are still significant challenges to their use [Dimensional Research, 2018]. What's more, microservices are inherently difficult to implement, with 87 percent of Red Hat survey respondents indicating that they are cobbling together multiple technologies for their microservice-based applications [Red Hat, 2018]. The complexities of interdependent moving parts and the technical expertise required to deploy and manage the supporting infrastructure have left many software developers scratching their heads on where to even start with building microservice-based applications.

Traditionally, Software Architects would design a system and pass the design off to Software Developers for implementation. The deployment of the finished product was then handled by IT Administrators. However, the adoption of agile software development methodologies has collapsed barriers between teams within small and mid-sized organizations, placing more of a responsibility for controlling the entire lifecycle of an application squarely on

the shoulders of a smaller application engineering team [Abrahamsson et al., 2002]. Despite this added responsibility, existing tools have not provided a sufficiently holistic solution to simplify the process of designing, developing, and deploying microservice-based applications for agile software teams. Furthermore, no representation of physical resources as an architectural model that abstracts the components of a microservice-based application while reducing vendor-specific dependencies exists. This results in a high entry-barrier into the world of microservices that discourages their use in an agile development environment, forcing smaller teams to use inadequate technologies to support their business logic.

This Master's Report presents Scootr Studio, the first in a class of next-generation integrated development environments (IDEs) that aims to simplify the design, development, and deployment of microservice-based applications. The system is designed specifically with less-experienced cloud-based application software developers in mind. It is intended to (1) enable these users to more efficiently develop microservice-based applications, (2) reduce the number of errors associated with their deployment, and (3) increase the overall usability of the cloud for software developers. Scootr Studio is based on a combination of principles from Model-Driven Engineering (MDE), Domain-Driven Design (DDD), and Event Storming for building highly scalable and maintainable software [Schmidt, 2006][Evans, 2003][Brandolini, 2019]. These principles result in a small set of abstractions that create an Event-Driven Application Architecture Model (EDAAM) that represents microservice-based systems in a way that minimizes vendor-specific dependencies. Users build EDAAMs from a model-driven IDE. An EDAAM is then sent to backend services and fed into provider-specific drivers that manage the deployment of the application and associated resources to the cloud.

A laboratory study involving 7 students from the Oregon State University Software Innovation Track was used to evaluate the efficiency of Scootr Studio over alternative methods. Evaluation participants then completed a survey to assess the overall usability of the system compared to alternatives. The combined laboratory results and the usability survey show that the highly usable nature of Scootr Studio successfully increases the efficiency of developing microservice-based applications while eliminating common deployment errors.

The remainder of this document is organized as follows. Section 2 presents related work and existing solutions in the areas of microservice design, development, and deployment respectively. Section 3 contains greater implementation details about Scootr Studio. Section 4 reports the methodologies and results from the laboratory study and usability survey used to evaluate Scootr Studio. Finally, Section 5 draws conclusions and summarizes opportunities for future work.

## 2 - Related Work

Industrial solutions to the development of microservices are quickly increasing in quantity and quality [Baldini et al, 2017]. As more individuals adopt microservices as their architecture of choice, companies and open-source organizations alike are developing a myriad of tools, frameworks, languages, and services to increase the ease and efficiency of designing, developing, and deploying their services. While these solutions alleviate many of the pains associated with developing distributed, cloud-based systems, they still require significant amounts of background knowledge, complicate application development, and introduce vendor lock-in.

## 2.1 - Microservice Design Techniques

Good software starts with good design, and the design of a microservice-based system is no exception. Traditionally, “PowerPoint” software architects have discussed the design of software systems and then handed off the design to software developers to implement [Rehman et al., 2018]. In recent years, the adoption of Agile Development processes has shattered the barriers between software architects and software developers, often merging the two together in a closely-knit team. Nonetheless, the distinct role of an architect versus developer introduces a communication failure point and increases the potential for new bugs to enter the system [Rehman et al., 2018].

A number of tools can assist with the creation of designs and help architects effectively communicate the system specifications to the developers. LucidChart and Draw.io are popular online options that provide dozens of shapes and a drag-drop canvas with connectors. Both also have cloud-storage integration and multi-user sharing, making it easy for teams to collaborate on designs [Lucidchart website, 2020][Draw.io website, 2020]. However, the representations are static and cannot be translated to code without the use of additional tooling such as Visual Paradigm [Visual Paradigm website, 2020]. Making changes in the architecture then requires revising the implementation, testing it, and potentially making further changes. New tools could facilitate the transition from design to development of microservice-based application development by more seamlessly translating architecture to code [Spillner, 2017].

## 2.2 - Microservice Development Processes

In order to create a scaffold for the microservice that they can subsequently fill in with application-specific details, developers can begin coding microservices by focusing on parts of the application not directly associated with the business logic. This scaffolding can include components such as authentication/authorization, message brokers, IPC, logging, monitoring, failure detection, and security. Many of these components can be replicated across multiple systems without significant changes to their code, which means that continually implementing this code for each microservice amounts to reinventing the wheel over and over again. Implementing this “boilerplate” is not as trivial as simply copy-pasting from an old application due to the fact that the names and APIs of microservices differ from application to application.

Therefore, many languages, libraries, frameworks, and tools have sprung up to help alleviate the pains associated with repetitive aspects of implementing microservices. For example, Micro (written for the Go programming language) aims to provide much of the boilerplating required for microservice development [Micro website, 2020]. It even features pluggable components that allow developers to customize their technology stack. A similar tool is the Ballerina programming language, which is written specifically for microservice development. Programs written in Ballerina compile to JVM byte-code and help take care of distributed communication, recovery, and coordination [Ballerina website, 2020]. Ballerina even introduces network-level abstractions in the language to facilitate the use of networks in the system.

While both of these options heavily reduce the amount of boilerplate code that development teams must write to use microservices, they still require developers deploy the microservices on their own. Moreover, such tool suites tend to work only on one specific platform (such as Ballerina on JVM), which means that developers are locked in to particular runtime platforms (such as Google Cloud, which natively supports JVM runtimes, versus Azure, which requires time-consuming initial configuration and ongoing management).

A more recently, widely-adopted process involves the use of “serverless” functions [Baldini et al., 2017]. Serverless development, as the name implies, involves developing an application with a strict focus on code that implements the application business logic and without much regard to how that code is eventually run. The hardware and supporting infrastructure is not considered. In this sense, it focuses on the *what* rather than the *how* of microservices. This process significantly reduces the amount of time and money it takes to develop microservice-based applications [Adzic et al., 2017]. However, each hosting provider has their own way of providing a serverless service, increasing the risk of vendor lock-in [Adzic et al., 2017].

## 2.3 - Microservice Deployment Tools

Another challenging aspect of developing microservices is infrastructure provisioning. In recent years, hosting providers such as Amazon Web Services (AWS), Microsoft Azure Cloud, Google Cloud Platform, and others have emerged to bear the burden of IT management for corporations and projects world-wide.



Currently, the most widely used infrastructure provider is AWS, which owns approximately 50% of the market share of cloud infrastructure and service providers [Su, 2019]. The usage of AWS over self-hosting creates significant cost reductions. In fact, using the AWS TCO calculator, a system configuration of 10 compute VMs and 3 database VMs, each with 4 cores and 32 GB of RAM, in addition to a total storage capacity of 100 TB can cost up to 52% less on AWS than in a self-hosted model [AWS TCO website, 2020].

All providers also allow fine-grained tuning of the system resources through web dashboards, but many of these low-level details are often very similar across different applications, meaning that developers unnecessarily learn the nuances of specific hosting providers while trying to accomplish the same bootstrapping of the infrastructure to support their applications. Some of these nuances can be removed by using containerization technology such as Docker to create isolated execution environments that are the same during development as they are in production [Docker website, 2020]. Containers can even be orchestrated and managed using a service such as Kubernetes, which can help coordinate and configure multiple services in the presence of failure [Kubernetes website, 2020]. Many hosting providers offer Kubernetes as a service, making it somewhat easier to set up a cluster and have pods running without manual intervention. However, proper use of tools such as Kubernetes require copious amounts of training study, and practice, increasing the cost of starting with or migrating to a microservice-based architecture.

Almost all hosting providers also offer platform services that simplify much of the low-level configuration needed for deploying applications. These Platform as a Service (PaaS) providers simplify the deployment process by attempting to help the developer only worry about

their code and a few configuration options. However, crucially, there currently does not exist a PaaS provider that offers a platform-independent implementation for microservice deployment. Therefore, using these deployment tools inherently creates vendor lock-in.

Some declarative frameworks exist to help ease the process of microservice-based systems using specific cloud providers and help to illustrate the lock-in issues discussed above. An example of this is Stackery, a tool that helps teams to quickly build and manage serverless infrastructure [Stackery website, 2020]. As a type of serverless acceleration software, Stackery utilizes a visual approach to representing the configuration of serverless applications on AWS. Under the hood, Stackery uses the AWS Serverless Application Model (SAM) to represent serverless applications. By using a visual approach, Stackery is able to provide a high-level view of the system and merge the design and deployment stages of a serverless application. It even comes with a Visual Studio Code extension, allowing the coding of the microservice to be included in the development process. Stackery works well as an initial attempt to merge the design, development, and deployment of microservice-based applications. However, it is built only for AWS and utilizes AWS SAM, meaning a developer has to learn how AWS SAM works on top of learning how Stackery works, creating a steeper learning curve when onboarding new developers. It also lacks additional abstractions on top of AWS SAM, meaning it has an extremely narrow infrastructure application context. And, as mentioned above, it is only for AWS: once a developer team has committed to using Stackery, they will find migrating to another platform extremely difficult.

Another tool called Architect sports an internal domain specific language (DSL) that can be used to represent the configuration of a microservice-based system. The framework interprets

the DSL and provisions the proper resources on AWS [Architect website, 2020]. The use of a DSL is one step towards simplifying the deployment process of microservices; however, similar to the Stackery model, the DSL is built around the AWS serverless ecosystem, which results in vendor lock-in and a requirement for understanding certain details about how AWS serverless works.

Still another popular tool is the Serverless Framework (also simply known as Serverless) [Serverless website, 2020]. Serverless features a small set of abstractions for configuring serverless functions inside of a YAML file. It then interprets the YAML configuration and deploys the appropriate resources to the hosting providing of choice. These configurations make deploying simple serverless applications extremely simple and efficient. Serverless also provides tooling and plugins for local development and configuration transformation to help simplify the process of developing for specific cloud providers. Most of the configuration is vendor-agnostic; however, if the function being deployed depends on any additional resources (such as a database or message queue), then those resources must be included in the YAML file using the infrastructure templating syntax specified by the target hosting provider. For example, using AWS as the provider requires the use of CloudFormation templates inside the Serverless YAML file. This implies that more knowledge about specific providers is required in order to build a system of significant value, which hinders the vendor-agnostic feature of the framework.

In addition to vendor-specificity, serverless also only supports serverless functions, which by their nature are short-lived, narrowly scoped processes. If the application calls for a zero downtime component, then the Serverless Framework cannot guarantee those requirements will

be met without additional knowledge of design-patterns for accomplish such a system setup using serverless functions.

## 3 - Solution

Scootr Studio is the first system to effectively combine the design, development, and deployment of cloud applications into one visual tool. The system's ability to enable efficient development of microservices lies in its architectural abstractions. These abstractions create an architectural model of the desired system called the Event-Driven Application Architecture Model (EDAAM). As shown in Figure 1, users create a description of the EDAAM using the model-based IDE user interface. During deployment, the IDE sends the description of the EDAAM to a collection of backend services called the Studio Services via the Studio Services' API Controller. The Controller queues the request for later processing by the Deployment Processor. Once the Deployment Processor transforms this description into a valid EDAAM, it feeds the model into a driver that allocates and configures the required infrastructure on the user's selected hosting provider. Progress of the deployment is communicated back to the IDE via an event stream for the user to view.

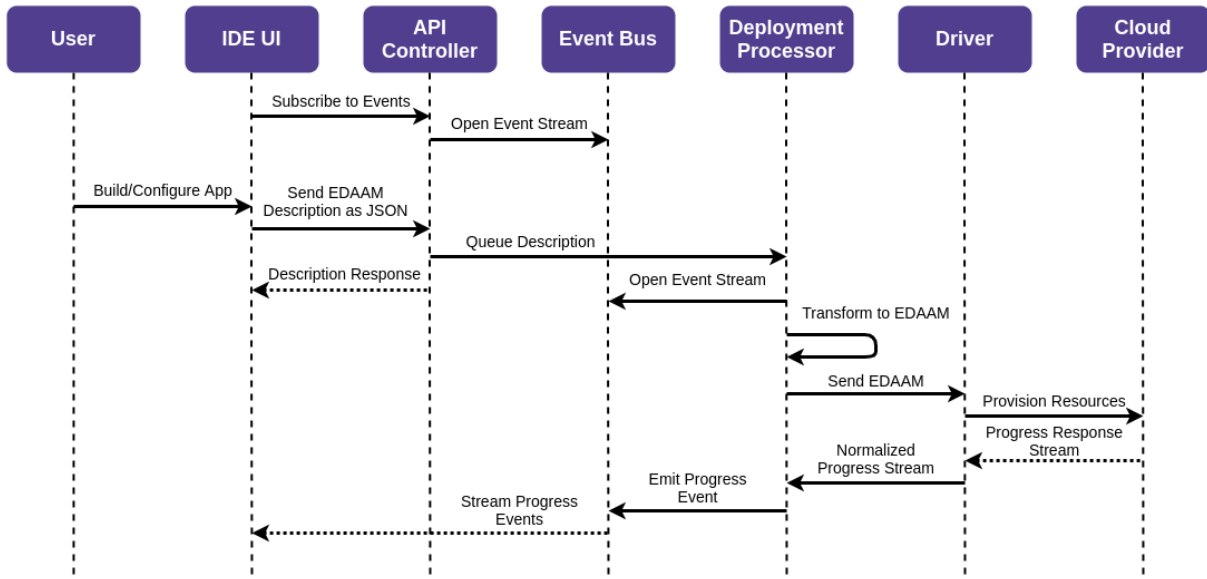


Figure 1: Sequence of events as a user interacts with Scootr Studio.

### 3.1 - A Model-Based Approach

The need to improve the process of building microservices has led to research about how to support Model-Driven Engineering (MDE) within development of distributed systems [Steinegger et al., 2017]. MDE consists of two primary components: (1) a domain-specific modeling language, and (2) transformation engines and generators that can analyze the model and synthesize various types of artifacts (e.g. code, binaries, infrastructure resources, etc.) [Schmidt, 2006]. Using these two components, MDE has the potential to reduce system complexity and the number of errors that occur post-deployment due to robust model validation and artifact generation. An additional benefit of using MDE to describe complex systems is that the use of visual elements that relate directly to a specific domain not only helps flatten learning curves, but also invites a broader range of subject matter experts to the table to help ensure that the software meet their requirements [Schmidt, 2006].

MDE fits hand-in-hand with the use of Domain-Driven Design (DDD), an approach to software design that divides a system into components that share similar functionality, known as domains [Evans, 2003]. The *core domain* is the domain that holds the primary business logic of the system that gives the system its competitive advantage. All domains, including the core domain, can be represented using model-based abstractions that are then fed into a generator of some kind to produce the application. Such an approach facilitates the unification of the design, development, and deployment of microservice applications.

## 3.2 - Architectural Abstractions

Scootr Studio is built on a small set of logical abstractions that represent collections of physical cloud resources. The abstraction boundaries are largely influenced by the process of Event Storming, a companion to DDD for determining the design of complex software systems [Brandolini, 2019]. Event Storming prompts the developer to think of events as the core component driving system design. Scootr Studio builds on this idea by breaking up events and the resources that handle them into six separate abstractions: Application, Event (subclassed with External and Internal), Compute, Trigger, Storage, and Reference. These abstractions allow us to represent microservice-based systems in a manner that significantly reduces the amount of hosting provider-specific configuration required to run the application.

For example, assume we are creating a simple e-commerce system as shown in Figure 2. Our application will have two External Events exposed as a RESTful HTTP endpoints: one for getting a user's orders (GetOrders), and another for allowing a user to order another item (OrderItem). The functional requirement for the Compute resource that handles the GetOrders

event (GetOrdersHandler) is trivial: simply read the user’s orders from a Storage resource (OrderStorage) and return a list of past and current orders. For the OrderItem event, a separate Compute resource (OrderItemHandler) first needs to add the order to OrderStorage. Then OrderItemHandler will trigger shipping of the item. It does this by emitting an Internal Event (ItemOrdered) on an internal message broker. A separate Compute resource (ItemOrderedHandler) listens for this event, executing some business logic involving a separate Storage resource containing shipping information for orders (ShippingStorage).

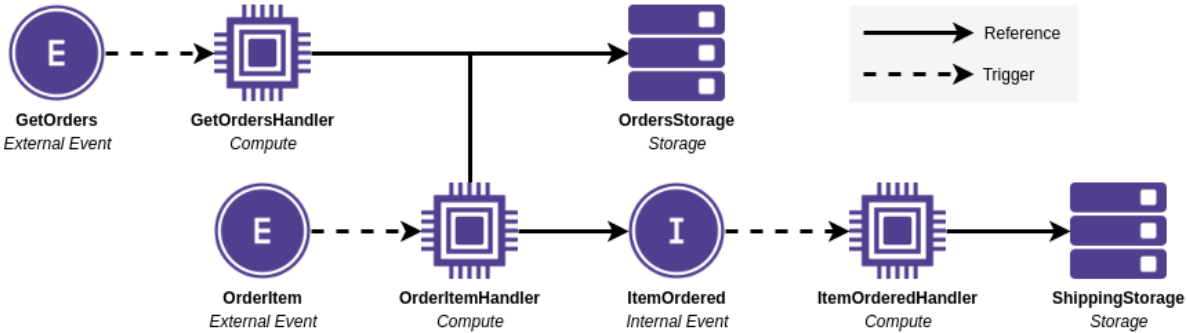


Figure 2: An e-commerce system represented using the Event-Driven Application Architecture Model.

### 3.2.1 - Application

An Application represents the collection of architectural components that compose a microservice-based application. It contains metadata about the system and determines the provider that will host the infrastructure used for the application, as well as the region to which the application will be deployed. The entire system represented in Figure 2 is considered the Application.

### 3.2.2 - Event

The Event abstraction logically represents actual events that trigger actions within the application. These events can be sourced from outside the application itself (classed as External Events) or from resources within the configuration (classed as Internal Events). External events are the entry point for the application, whereas Internal Events are communication channels between different resources in the application. This subclassing of the Event abstraction facilitates the creation of event-driven systems, which are easier to maintain and enhance than their more tightly-coupled counterparts [Michelson, 2011].

External events primarily represent HTTP requests on RESTful API endpoints. Internal events can be events emitted over an internal message broker. Users configure the Event's type and any additional type-specific configuration, such as HTTP path and method or the broker used for internal events. The Event abstraction's configuration is vendor-agnostic up to the point that hosting-provider specific methods are not used as External Event sources or as the broker for Internal Events. For example, if the application uses Amazon SNS as the internal event broker, then that configuration would not be valid on another hosting provider such as Microsoft Azure or Google Cloud Platform.

Our e-commerce example contains two External Events (GetOrders and OrderItem) and one Internal Event (ItemOrdered).

### 3.2.3 - Compute

The Compute resource executes actions within the system in response to events. The key configuration values supplied by the user for Compute resources include the code used to handle



the event and the runtime environment that code will be executed in. Realistically, the Compute resource can represent any kind of processing infrastructure (virtual machine instances, Docker containers, Kubernetes pods, serverless functions, etc.). Currently, the implementation of the abstraction supports configuration fields that allow a driver to easily create a serverless function (see section 3.6). Additional supported physical Compute resource configuration fields (such as containers) are under active development. Similar to Events, the Compute resource is provider-agnostic up to the point that the code does not contain references to hosting provider-specific SDKs or APIs.

Our e-commerce example has three Compute resource instances: (1) `GetOrdersHandler`, (2) `OrderItemHandler`, and (3) `ItemOrderedHandler`.

### 3.2.4 - Trigger

Triggers are connections from an Event to a Compute resource. As the name implies, they create the configuration that will tell the hosting-provider to “trigger” the Compute resource action whenever the Event is emitted. Triggers are considered a trivial abstraction in that they do not require any additional configuration outside of the source resource (an Event) and the target resource (a Compute).

In our example e-commerce application, there are three triggers represented by dashed lines: (1) from `GetOrders` to `GetOrdersHandler`, (2) from `OrderItem` to `OrderItemHandler`, and (3) from `ItemOrdered` to `ItemOrderedHandler`.

### 3.2.5 - Storage

Storage resources represent locations where data can be persisted by Compute resources. They exist independent of the Compute resources that use them, increasing the compositionality of the system. There are currently two main types of Storage resources: Key-Value and Relational. Each type allows the user to configure a database engine that is used to persist the data fed to the resource by the Compute resources in the application. Any additional required configuration for the Storage resource depends on the type. For instance, Key-Value Storage resources only require the database collection name, primary key field name and primary key field data type. On the other hand, a Relational Storage resource requires the user to provide the entire schema for the database table or tables the driver will ultimately create.

The e-commerce example at the beginning of this section contains two Storage resources: (1) OrderStorage and (2) ShippingStorage.

### 3.2.6 - Reference

References are the connections from a Compute resource to Storage resources and Events. References are considered a different type of connection from Triggers because they give Compute resources access to other resources in the application. As such, this access needs to be secured. References can be configured to allow create, read, update, or delete actions on the target resource, ensuring the architecture can conform with the principle of least-privileged access [Ma et al., 2011].

In our e-commerce example, there are three references represented by solid lines: (1) from GetOrdersHandler to OrderStorage, which would have *read* permission on the

OrderStorage resource; (2) from OrderItemHandler to OrderStorage, which would have *create* permission on the OrderStorage resource; and (3) from OrderItemHandler to the ItemOrdered event, which corresponds to the use of an internal message broker and has *create* permission for that resource.

### 3.3 - User Interface

From a user's perspective, Scootr Studio is a single-page web application IDE user interface (IDE UI) built using JavaScript and the React Framework. Its goal is to merge the design, development, and deployment of microservices into a single view. To accomplish this, it features a drag-and-drop canvas on the left (design) and a details pane for configuration on the right (development). Deployment is managed by the toolbar above the drag-and-drop canvas.

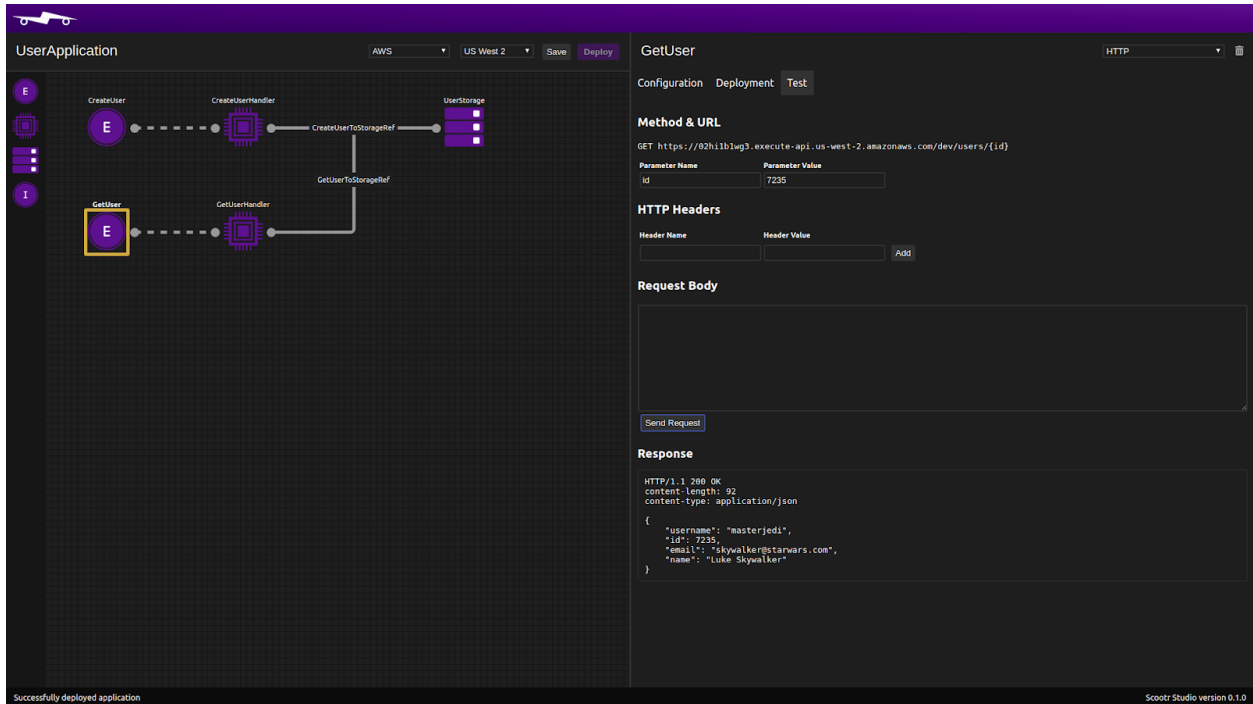


Figure 3: Screenshot of a completed application using the Scootr Studio user interface.

Much of the web application is built entirely with React, including state management. However, some third-party library integrations for blueprint canvas and code editing were developed as custom React hooks for use within the system.

### 3.3.1 - State Management

State for the application is handled exclusively by React hooks and the Context API. Both hooks and context are React concepts used for state management by many existing libraries. Using native features of the framework as opposed to any one of these libraries helps reduce dependencies and minimize the attack surface of the application, which is important when building software that creates software and provisions infrastructure [Zhang et al., 2015]. It also

allows the code to be more easily written in a functional style, which increases the modularity of the system and reduces complexity [Bosch et al., 2010][Bitman, 1997].

Scootr Studio uses three separate contexts to manage state: the Application Context, the Status Context, and the Workspace Context. The Application context manages general application configuration (i.e. for the Application abstraction in the EDAAM). The Status Context manages the state of notifications and status bar messages for the entire application. The Workspace Context manages the remainder of the state as controlled by the drag-drop canvas and the configuration pane, including the rest of the EDAAM description configuration and validation. The motivation behind using separate contexts boils down to performance. React triggers a re-render for all components that subscribe to a specific state context whenever that context changes. If all the state existed in single context, the entire application would re-render each time state changed. This can get expensive, so to reduce memory overhead and give more of the CPU back to the computer, Scootr Studio splits the state into subsets that reduce magnitude of each re-render.

Updating state in the Application and Status Contexts is trivial, as it is a simple replacement of the current state object with a new state object containing the most up-to-date values. However, the Workspace Context requires a little more work to update, wherein the following algorithm is used: (1) create a new resource or connection state object (depending on what type of resource has changed value), (2) merge the new values in with the old resource state, (3) validate all the properties of the new resource or connection state, (4) create a new Workspace state object and copy the previous state values into it, (5) merge the new resource or connection state into the new Workspace state object, (6) validate the names and fields for all

resources and connections that are required to be unique across the EDAAM, and (7) update the selected resource or connection reference (if it is not null).

The steps involving validation are the most expensive in the algorithm, mainly because they require looping over all of the fields of a resource and then over all of the resources in the state tree. If a resource has  $f$  fields, and there are  $r$  resources and  $c$  connections, then updating state in the Workspace Context has a runtime complexity of  $O(f + r + c)$ , which simplifies to linear complexity. In reality, most EDAAMs are relatively small, and each resource has only a few fields, so the added time complexity due to validation doesn't manifest itself as lag during a re-render. This allows front-end validation to provide a better user experience to the developer before deployment, eliminating the need to deploy an invalid configuration and wait for the server to respond with the errors.

The final step in the algorithm is critical for preventing memory leaks and stale references, particularly when the resource or connection being updated in the state tree is currently selected. If the selected resource or connection is not updated, then the reference in the state tree will be to the object in the previous state, which can lead to state-corruption and potential unresponsiveness of the application.

For the purposes of assessing the innovative nature of Scootr Studio, state is currently persisted across browser sessions through the use of local storage, removing the need for a database in the backend to store user and application information. Future iterations of Scootr Studio will produce lockfiles to be committed with source code as a means for storing application configuration and state, leaving everything in the control of the developer.

### 3.3.2 - Blueprint Canvas

The blueprint canvas provides a visually appealing way of constructing a valid EDAAM. Under the hood, it's implemented using the HTML 5 Draggable API and jsPlumb, a library for visually connecting HTML DOM elements [Mozilla website, 2020][jsPlumb website, 2020]. Custom React hooks are used to implement drag-and-drop, whereas a single instance of jsPlumb is used to implement visual connectivity between resources on the canvas. React integration for jsPlumb had a difficult API, so a newer library (react-plumb) was developed in parallel to work being done on Scootr Studio. react-plumb uses React hooks for interfacing a React application with an instance of jsPlumb efficiently. The use of this library enables the creation of Triggers and References for the EDAAM from Scootr Studio.

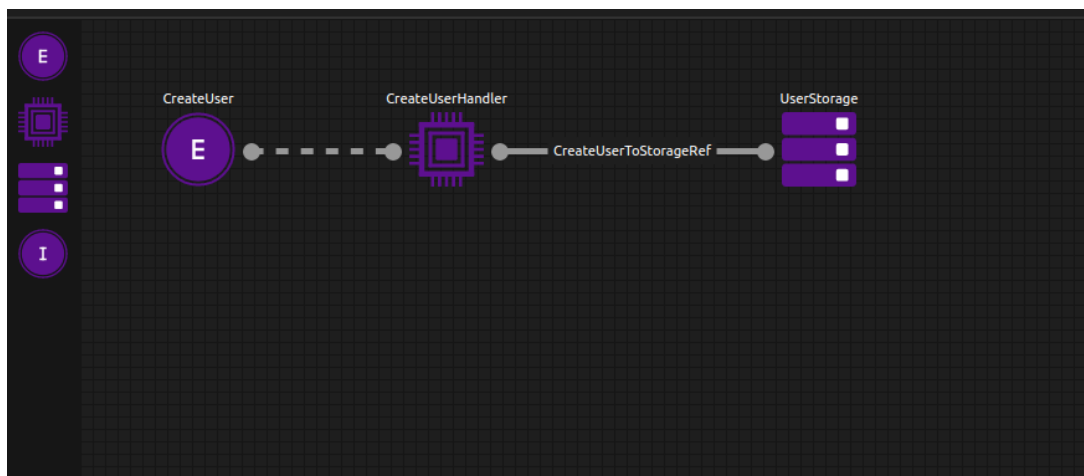
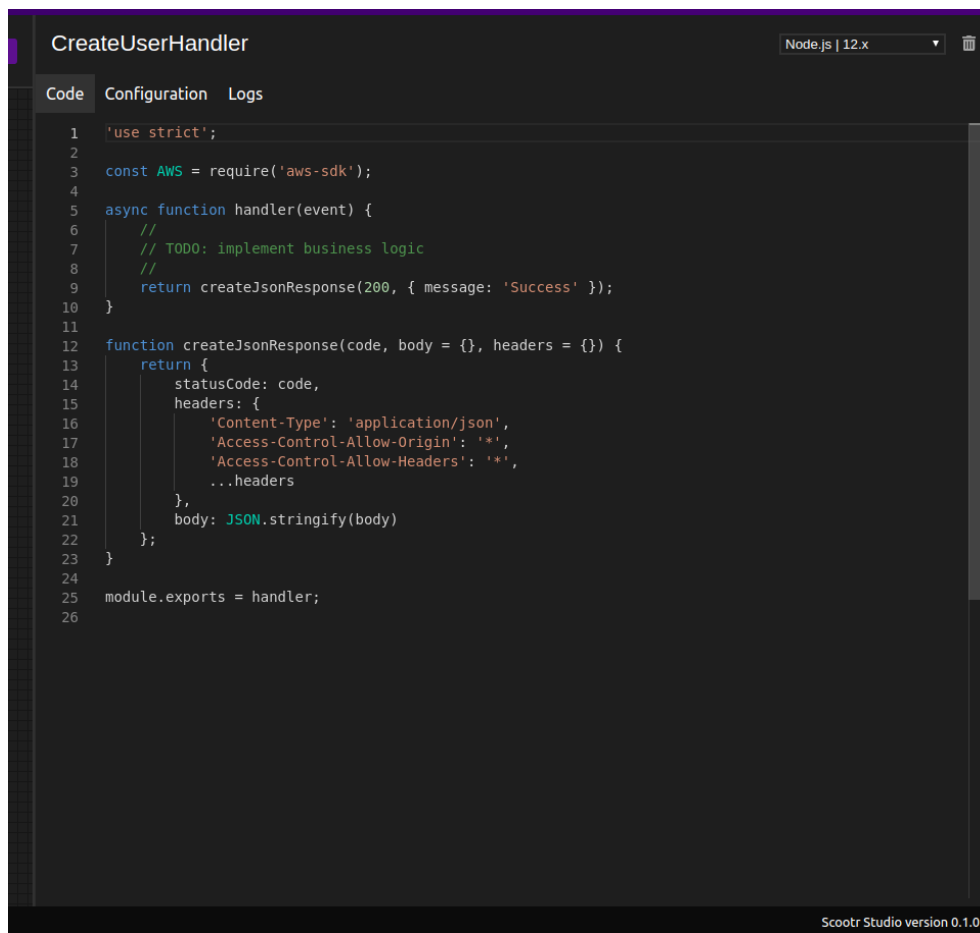


Figure 4: Screenshot of the blueprint canvas where model objects are dropped and connected.

### 3.3.3 - Code Editor

The code editor on the Compute resource details pane utilizes the Monaco Code Editor by Microsoft [Microsoft Monaco website, 2020]. This is the same editor engine used in the

popular text editor Visual Studio Code. Scootr Studio uses the Singleton Pattern to improve the performance of having multiple editor views for different Compute resources [Gamma et al., 1994]. There is a single, global instance of a monaco editor for the entire application. Each Compute resource has its own view model that is loaded when the Compute resource is selected.



```
1 'use strict';
2
3 const AWS = require('aws-sdk');
4
5 async function handler(event) {
6   //
7   // TODO: implement business logic
8   //
9   return createJsonResponse(200, { message: 'Success' });
10 }
11
12 function createJsonResponse(code, body = {}, headers = {}) {
13   return {
14     statusCode: code,
15     headers: {
16       'Content-Type': 'application/json',
17       'Access-Control-Allow-Origin': '*',
18       'Access-Control-Allow-Headers': '*',
19       ...headers
20     },
21     body: JSON.stringify(body)
22   };
23 }
24
25 module.exports = handler;
26
```

Figure 5: Screenshot of the code editor for Compute resources in the right panel of Scootr Studio

### 3.3.4 - Testing and Monitoring

Scootr Studio provides testing for External Events and log monitoring for Compute resources. This allows users to troubleshoot issues in their code without needing to access the hosting provider dashboards. It also allows faster debugging in the absence of a full-featured



local development environment that mimics the hosting provider of choice, which is not available in the current version of Scootr Studio.

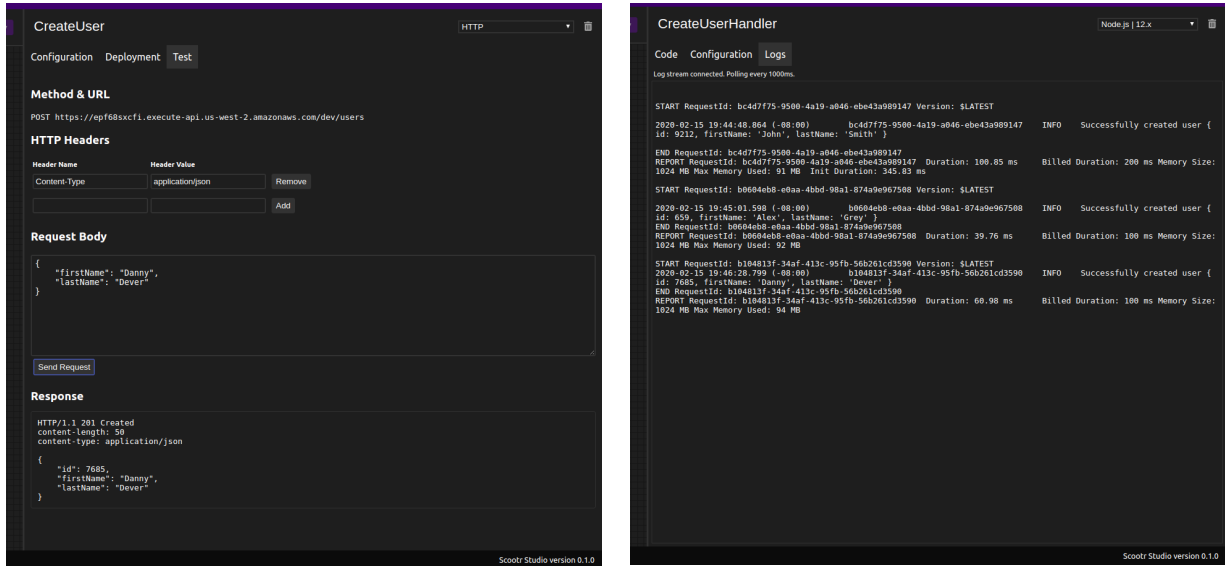


Figure 6: Screenshots of the test pane (left) for an Event and the log pane (right) for a Compute resource.

### 3.4 - Studio Services

Once a user has successfully built an EDAAM description and clicked on the “Deploy” button in the toolbar, the IDE UI sends the description to a collection of backend services called the Studio Services. These services (shown in Figure 7) communicate with each other over IPC and an event stream to collectively transform the EDAAM description into an actual EDAAM capable of being processed by a driver. The EDAAM is subsequently deployed by the driver to the user’s selected hosting provider.

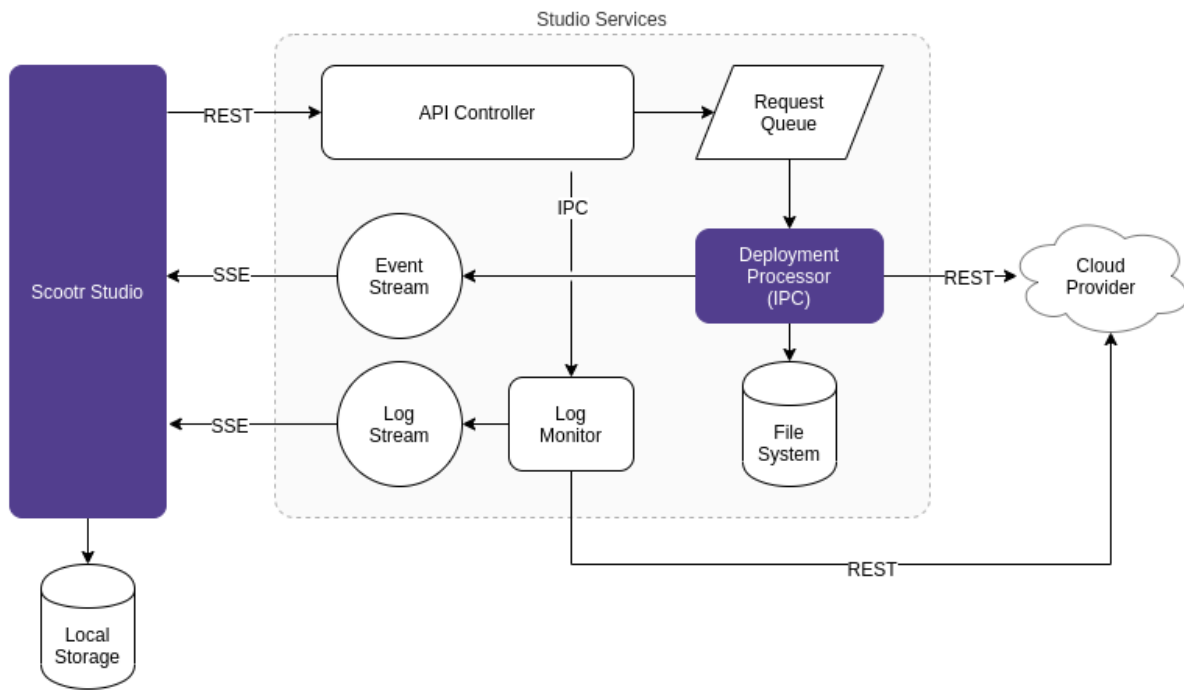


Figure 7: System design and components of Studio Services

All requests made by Scootr Studio to the Studio Services go through the API Controller, which acts as the gateway through which Scootr Studio interacts with the backend services. The Studio Services are stateless, so they use encrypted and signed session tokens (stored as cookies in the IDE UI) in order to keep track of the client they are communicating with.

The API Controller communicates with other services to fulfill requests made by Scootr Studio. These other services, such as the Log Monitor and Deployment Processor, relay information and progress through IPC back to endpoints initialized by the API Controller. These endpoints transmit the information to Scootr Studio using separate event streams. These streams are implemented using Server Sent Events (SSE), a unidirectional event stream over HTTP from server to client [Mozilla website, 2019]. Studio Services use SSE instead of web sockets because

Scootr Studio is not making real-time requests to the Studio Services, only occasional ones. This means that the bidirectional stream offered by the WebSocket API is not necessary, and so we reduce complexity and maintain a clean interface by not using it.

A key component of the Studio Services is the Deployment Processor, which uses the Scootr runtime library to process a request containing an EDAAM description (see section 3.5). This process runs outside of the main API Controller event loop, increasing the concurrency capabilities of the Studio Services. The Deployment Processor is also in charge of spawning the Serverless Framework process that will ultimately handle deployment. Communication between the two processes is handled over standard input and output, with output from the Serverless Framework process being parsed and cleanly formatted for status reporting inside of the Studio Services. The system also utilizes logging to capture it's behavior and provide a reference for troubleshooting issues. These logs are emitted as JSON-formatted event packets on the Studio Services standard output pipe, allowing other processes to determine how to handle and display the logs.

In order for users to view the logs for the applications they are building, the log monitor captures log events and streams them back to the client once a subscribing connection has been established. Logs are polled once every second after a connection has been established and sent to the client using SSE.

## 3.5 - Scootr Runtime Library

The key architectural abstractions are implemented as a runtime library called Scootr. The library provides a chainable API that simplifies the representation of a system's architecture

as code. This API is currently supported in Node.js. Each object in the API allows for complex configuration by using the Builder Pattern for constructing instances of objects with multiple properties (some required and others optional) [Gamma et al., 1994].

The API hides the creation of Triggers and References behind the creation of Compute and Storage resources that use Events. This increases the expressiveness and chainability of the API by creating the representation of an application architecture using a more declarative style of programming [Lloyd, 1994].

When the Deployment Processor receives an EDAAM description, it uses the Scootr runtime library to transform the description into a valid EDAAM capable of being processed by a driver. Figure 8 shows the result of transforming our EDAAM description from the IDE UI for our e-commerce application into a valid EDAAM using the Scootr runtime library.

```

1  const { application, compute, storage, http, topic,
2     types, actions } = require('scootr');
3  const { driver, enums } = require('scootr-aws');
4
5  const orderStorage = storage('OrderStorage', types.KeyValueStorage)
6     .engine(enums.Storage.DynamoDB)
7     .collection('Orders')
8     .key('oid')
9     .keytype(enums.Storage.Number);
10
11 const shippingStorage = storage('ShippingStorage', types.KeyValueStorage)
12     .engine(enums.Storage.DynamoDB)
13     .collection('ShippingInfo')
14     .key('iid')
15     .keytype(enums.Storage.Number);
16
17 const getOrdersEvent = http('GetOrders').method('GET').path('/orders');
18
19 const orderItemEvent = http('OrderItem').method('POST').path('/orders');
20
21 const itemOrderedEvent = topic('ItemOrdered')
22     .broker(enums.Brokers.SNS)
23     .name('item-ordered');
24
25 application('ECommerceExample')
26     .with(
27         compute('GetOrdersHandler')
28             .runtime(enums.Runtimes.Node_12_x)
29             .code(/* Event-handler code here */)
30             .on(getOrdersEvent)
31             .use(orderStorage, actions.Read, 'GetOrdersStorageRef')
32     )
33     .with(
34         compute('OrderItemHandler')
35             .runtime(enums.Runtimes.Node_12_x)
36             .code(/* Event-handler code here */)
37             .on(orderItemEvent)
38             .use(orderStorage, actions.Create, 'OrderItemStorageRef')
39             .use(itemOrderedEvent, actions.Create, 'ItemsOrderedRef')
40     )
41     .with(
42         compute('ItemOrderedHandler')
43             .runtime(enums.Runtimes.Node_12_x)
44             .code(/* Event-handler code here */)
45             .on(itemOrderedEvent)
46             .use(shippingStorage, actions.All, 'ShippingStorageRef')
47     )
48     .deploy(driver, enums.Regions.UsWest2);

```

Figure 8: The example e-commerce application's EDAAM in Node.js using the Scootr library

## 3.6 - Drivers

The Scootr runtime library exposes a function called “deploy” that begins the process of application deployment. The first argument the deploy function accepts is the driver used to deploy the configuration. The second argument is the region to which that application should be deployed. When the “deploy” function is called, the Scootr runtime library systematically feeds the EDAAM into the supplied driver (see Figure 9). The runtime library first provides all of the Events in the system, followed by the Compute and Storage resources. Finally, Triggers and References are provided to the driver.

The order of abstraction delivery to the driver is important. As the core abstraction and the source of action in applications built using Scootr, Events must be provisioned first. This is especially important because Compute resources depend on events, and so the events must be configured properly before they can be used. Compute resources must come before Storage resources for this same reason. Finally, all the connections between the resources in the system come after resource provisioning.

While drivers are primarily responsible for application configuration deployment, they also are responsible for generating the boilerplate code used to successfully run the microservice-based application on the target hosting provider. Delegating the creation of boilerplate code to the driver helps relieve an additional burden on users of the system and encourages reusable code across multiple applications.

Drivers are pluggable, which enable the same EDAAM to be deployed to multiple hosting providers with minimal changes to configuration. This means that, given suitable drivers,

a programmer could create a single program with Scootr Studio, then build and deploy it on multiple platforms (e.g., AWS and Azure). The current version of Scootr requires that certain configuration be changed depending on the driver used. For example, if one driver deploys to Azure, a Key-Value Storage resource configuration that uses AWS DynamoDB as its database engine would not be valid and would need to be changed. In this way, EDAAMs are guaranteed to be provider-agnostic up until the point of utilizing services offered by a specific provider.

The driver used in evaluations for Scootr Studio was the AWS driver (see section 4). This driver uses the Serverless Framework (see Section 2.3) to deploy resources to AWS. The driver constructs a workspace directory and the source-code root to add all of the necessary Compute configurations and dependencies, then builds the YAML configuration file used by Serverless to provision infrastructure on AWS. The advantage to implementing the AWS driver using an existing framework is the reliability that comes with an professionally backed, open-source product, along with the agility to focus on developing better abstractions and a cleaner user interface to increase the usability of the EDAAM.

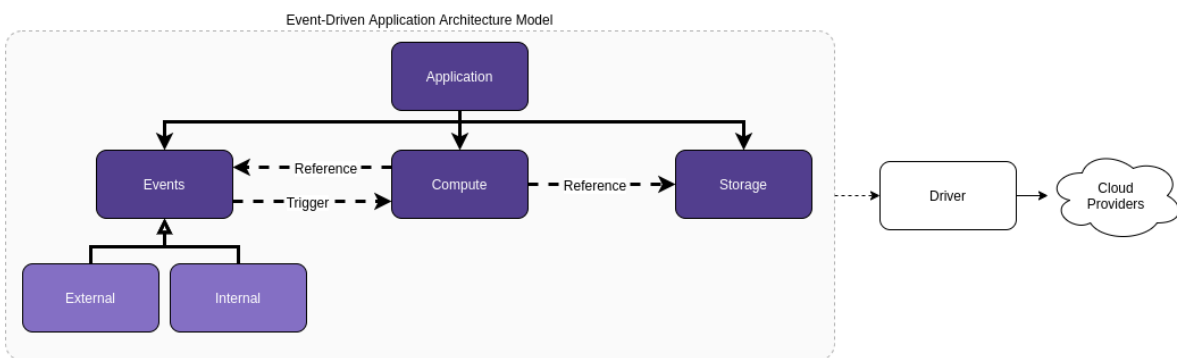


Figure 9: The EDAAM is fed into a hosting-provider-specific driver that manages the deployment of resources to the cloud.

Another advantage of using Serverless as the framework for the driver is the ability to rollback on failures. Serverless for AWS uses CloudFormation, an AWS service that allows the declarative creation and maintenance of infrastructure, to provision resources [Amazon Web Services website, 2020]. CloudFormation allows failed deployments to be rolled back to the most recently successfully deployment. These failures are reported to the driver so that it can report back to the Scootr library on the reason for failure.

Due to insufficient time to create more drivers, the only available driver for the Scootr runtime library is the AWS driver. Although the current lack of drivers for platforms other than AWS prevents Scootr Studio from deploying users' code onto other platforms (e.g., Azure), the abstractions noted above will make it straightforward to add support for additional platforms. In particular, supporting a new platform only requires implementing a new Driver for that platform, including support for mapping events and storage references to appropriate platform-specific APIs, as well as support for deploying executables to the cloud. Implementing the Driver for AWS took approximately 2 weeks of work. As it was the first to be implemented, it is likely that implementing a Driver for another platform would take even less effort.

## 4 - Evaluation

Evaluations consisted of two parts: a laboratory study followed by a usability survey. The laboratory study aimed to analyze the efficiency and advantages of using Scootr Studio over other methods of microservice-based application development. The usability survey gave participants the opportunity to assess Scootr Studio on the aspects of learnability, memorability, efficiency, and error rates.



## 4.1 - Laboratory Study

The laboratory study investigated how quickly participants were able to create a RESTful API using Scootr Studio as opposed to their preferred IDE. The goal was to explore how much the unification of the design, development, and deployment stages of microservice-based application development reduces the amount of time it takes to build these systems.

The study specifically focused on the following two questions: (1) How much more quickly could individuals build microservice-based applications using Scootr Studio? (2) How many fewer errors would participants report while building a microservice-based application using Scootr Studio?

The skills required to complete the evaluation task, in addition to the length of each evaluation (for several hours), resulted in a lower number of participants than would typically be expected for statistically significant results. As such, the goal was not to uncover statistically significant differences between Scootr Studio and other tools. Instead, the evaluations looked at whether developers were at least several times faster with Scootr Studio than existing IDEs. This is because (from the perspective of increasing adoption of Scootr Studio among software developers) such differences play a more important role than small differences that would only be significant in a large sample.

### 4.1.1 - Methodology

*Participants:* Students from the Software Innovation (SWI) Track at Oregon State University (N=7) were recruited via email. The SWI Track is home to students who have

industry experience developing software applications. As such, many of the students have professional experience with cloud-based application development, making them viable candidates for the study.

*Procedure:* Scootr Studio was installed on four Windows 10 machines in the Oregon State University EUSES Laboratory. The required libraries and processes were started in the background prior to the participants' arrival, along with the necessary API keys required for interacting with isolated AWS accounts for each machine. Participants were invited to attend one of three sessions during which the evaluation would take place. Food and refreshments were provided throughout the duration of the evaluation.

Participants were asked to build the RESTful API for a simple todo-list application. The API had four endpoints: (1) creating an item, (2) getting a list of items, (3) updating a single item, and (4) deleting a single item. They were provided with in-depth descriptions of the functionality of each endpoint, as well as example HTTP request/response pairs mimicking successful calls to the API. They were instructed that the items must adhere to the format specified in the evaluation task, be persisted in a database, and be publically accessible via their API (i.e. no "localhost" services). Once participants had completed the task, their implementations were tested using scripts powered by Postman, a RESTful API development tool [Postman website, 2020].

Participants were asked to complete the evaluation task in two ways: (1) using Scootr Studio, for which they were given a time limit of one hour, and (2) using whatever method they preferred, for which they were given a time limit of four hours. For this comparison method, participants were free to use any language, framework, database, and hosting provider they

desired, along with any online resources they could find. Following a counterbalanced study format, approximately half of the participants (N=3) completed the evaluation using Scootr Studio first, followed by their preferred method. The other half (N=4) used their preferred method first, followed by Scootr Studio.

When completing the evaluation task using Scootr Studio, participants were shown a five-minute demonstration of the system. They were then asked to complete a 20 minute tutorial that walked them through the process of building a simple web application with two RESTful API endpoints. This allowed them to familiarize themselves with Scootr Studio before attempting to accomplish the evaluation task. This was not considered an unfair advantage, as participants already had prior experience developing RESTful APIs using the tools they would ultimately use for their method of choice. It also allowed a more accurate assessment of the learnability of Scootr Studio using the post-evaluation usability survey (see section 4.2).

*Data acquisition:* Participants were timed using a stopwatch during the use of Scootr Studio (after the tutorial), as well as during the use of their preferred method.

Whenever a participant asked a question or reported an error, the question or error was recorded so as to assess the types of questions and the frequency of errors reported by participants during their attempts to complete the evaluation task.

*Analysis:* The time it took for participants to complete the task was converted to seconds and rounded up to the nearest whole second. Results were then computed and converted to decimal representations of minutes. The speedup of using Scootr Studio over using the participant's preferred method was then calculated by dividing the time it took to complete the task using the method of choice by the time it took to complete the task using Scootr Studio.

All questions and errors reported by were systematically separated into three categories based on their topic: design, development, and deployment. The design category contained questions and errors that had to do with the overall understanding and architectural representation of the application before any coding took place. Development issues were those involving the actual writing of code to implement the business logic required to complete the task. Issues classified as deployment issues were, as the name implies, those that involved the use of a hosting provider and the attempts to deploy the locally tested and built application to the cloud. The frequency of each type of question and error was compared in each category to create an additional aspect of efficiency achieved by using Scootr Studio—that is, minimization of errors.

#### 4.1.2 - Results

*(1) Average speedup when using Scootr Studio:* Only 3 participants were able to complete the task using their chosen method within the 4 hour time limit. On average, they took 151.71 minutes. For the remaining 4 participants, 4 hours was used in the analysis as their nominal time to complete the application with their preferred methods, even though they didn't actually finish; thus, the speedup reported below with Scootr Studio is actually a conservative estimate. With this caveat, the average completion time for all participants using their preferred method was 202.16 minutes.

In contrast, all of the participants were able to complete the task using Scootr Studio. The average completion time for using Scootr Studio was 45.68 minutes.

Therefore, on average, using Scootr Studio allowed the development of a microservice-based application to happen 4.4 times faster than when using alternate methods. As explained above, this is a conservative estimate. That said, even if only the 3 participants who successfully completed the task using their preferred method are considered, the use of Scootr Studio still results in the completion of the task an average of 4.1 times faster than the use of their preferred method.

(2) *Reductions in error reporting*: Table 1 presents the resulting error classification totals and averages for the 7 participants in the evaluations. All deployment errors were eliminated by the use of Scootr Studio. There was 1 design error reported when using Scootr Studio, but upon further discussion with the participant it was discovered that this was intentional and ultimately due to a difference of philosophy about how RESTful APIs should be built and what best-practices were for developing APIs, as opposed to an actual error.

Table 1: Total and averages for the three types of errors reported by final evaluation participants during the completion of the evaluation task.

|                    | Chosen Method |         | Scootr Studio |         |
|--------------------|---------------|---------|---------------|---------|
|                    | Total         | Average | Total         | Average |
| <i>Design</i>      | 2             | 0.29    | 1             | 0.14    |
| <i>Development</i> | 7             | 1       | 29            | 4.14    |
| <i>Deployment</i>  | 8             | 1.33    | 0             | 0       |

Interestingly, although Scootr Studio eliminated *deployment* errors, participants did experience more development errors with the unfamiliar Scootr Studio as opposed to their alternate method. Upon further investigation, it was discovered that this was due to the

participants' lack of experience with some of the AWS APIs they used in order to accomplish the task using Scootr Studio (AWS Lambda integration and DynamoDB SDKs). The nature of Compute resource configuration allows the user to supply the code that will be run in response to an event. Such an approach allows greater flexibility in what types of systems can be built using Scootr Studio; however, it also introduces a point of hosting provider dependence if the user decides to use services offered exclusively by a single hosting provider (such as DynamoDB). As will be seen in the results of the usability survey in section 4.2, many of the participants expressed a desire for more supporting tools for development of the application business logic to increase the usability of Scootr Studio and ease the burden on developers.

## 4.2 - Usability Survey

A usability survey with three main sections was developed and used to capture information about the participants and their opinions regarding Scootr Studio: (1) participant background information, (2) a series of agree/disagree statements, and (3) general ratings and feedback. The second section was further split into four areas: (a) learnability, or how quickly and easily a user can figure out how to use the system; (b) memorability, or how easy it is to remember how to use the system without extra effort; (c) efficiency, or how quickly users can accomplish tasks with the system; and (d) error rates, or how successfully users can accomplish tasks with the system.

### 4.2.1 - Methodology

Upon completion of the laboratory study), participants were asked to complete a short survey following the outline above. The survey focused on learnability, memorability, efficiency, and error rates as key indicators of the system’s usability. These questions, located in the second section of the survey, were 5-option agree/disagree Likert scale response questions as outlined in Table 2.

Table 2: Agree/Disagree statements included in the usability survey

|   |
|---|
| <b>Learnability</b>   |
| Learning to use Scootr Studio took less time than learning my preferred method of microservice development.                           |
| Learning to use Scootr Studio required learning fewer concepts than my preferred method of microservice development.                  |
| <b>Memorability</b>   |
| Scootr Studio makes it easier to visualize my system as a whole than with my preferred method.  |
| Scootr Studio naturally prompts me towards building microservices better than my preferred method.                                    |
| <b>Efficiency</b>   |
| Scootr Studio makes it more efficient to develop microservices than my preferred method.  |
| Scootr Studio makes it more efficient to deploy microservices than my preferred method.   |
| <b>Error Rates</b>  |
| I make fewer errors developing microservices with Scootr Studio than I do using my preferred method.                                  |
| It is easier to tell when the configuration of my microservices is not correct using Scootr Studio as opposed to my preferred method. |

Additional questions were included in the first section of the survey to determine how long the participants had been developing cloud based applications and what tools and resources they used for their preferred method during the evaluation. The final section gave participants the opportunity to rate the overall usability and efficiency of the system on a scale from 1-10. It also allowed them to provide feedback on what parts of the system they wished to see improved and what features they wanted to see added.

#### 4.2.2 - Results

*(1) Participant background information:* A large proportion of participants had less than 1 year of experience with developing cloud-based applications, as outlined in Figure 10. Overall, these participants performed approximately as well as participants who had more 2 or more years of experience, indicating that Scootr Studio is effective at flattening learning curves and reducing overhead when building microservice-based applications. No participants had only between 1 and 2 years of experience.



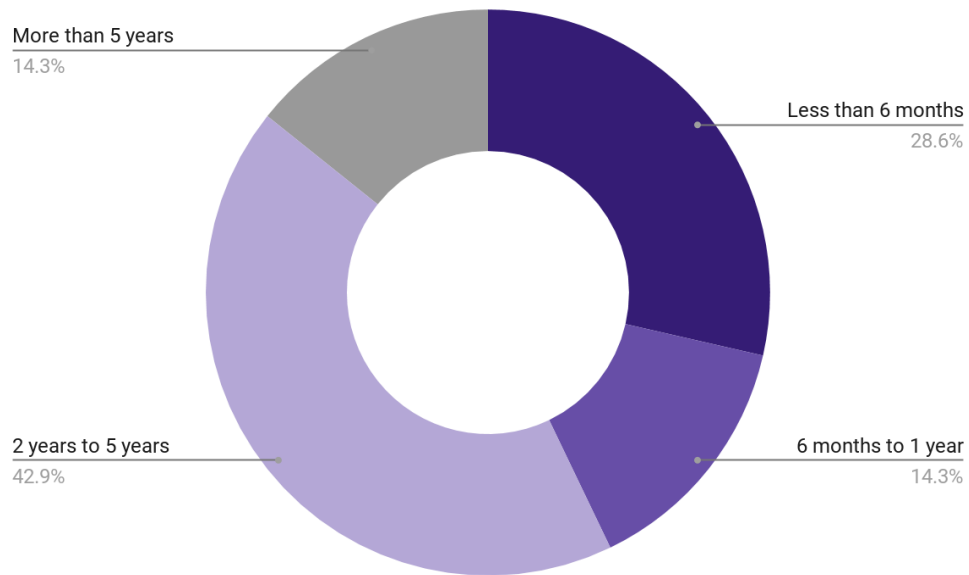


Figure 10: Distribution of cloud-application development experience among participants

(2) *Agree/disagree statements:* In general, participants strongly agreed that Scootr Studio was an improvement in the areas of learnability, memorability, efficiency, and error rates when compared to their method of choice. As indicated in Table 3, almost every response was positive (above neutral, meaning that participants selected “Agree” or “Strongly Agree”). With the exception of one neutral participant, all participants strongly agreed that Scootr Studio significantly reduced the number of errors encountered when developing microservice-based applications.

Table 3: Participant responses to agree/disagree statements about Scootr Studio’s usability  
 (Strongly Agree = 3, Agree = 2, Disagree = 1, Strongly Disagree = 0)

|                     | <b>Average Score</b><br>(out of 3) | <b>Responses Above</b><br><b><i>Neutral</i></b> |
|---------------------|------------------------------------|---|
| <i>Learnability</i> | 2.86                               | 13 of 14  |
| <i>Memorability</i> | 2.69                               | 13 of 14  |
| <i>Efficiency</i>   | 2.77                               | 13 of 14  |
| <i>Error Rates</i>  | 3                                  | 13 of 14  |

(3) *Ratings and Feedback*: Overall, participants gave an average rating of 9.29 out of 10 for both Scootr Studio’s usability and efficiency. This serves as a testament to the ability for Scootr Studio to not only increase the speed and ease of microservice-based application development, but also to its ability to simplify the usage of cloud-providers and flatten the entry barrier for developing applications for the cloud. The summary of responses to the ratings can be seen in Figures 11 and 12.

### Scootr Studio Usability Rating

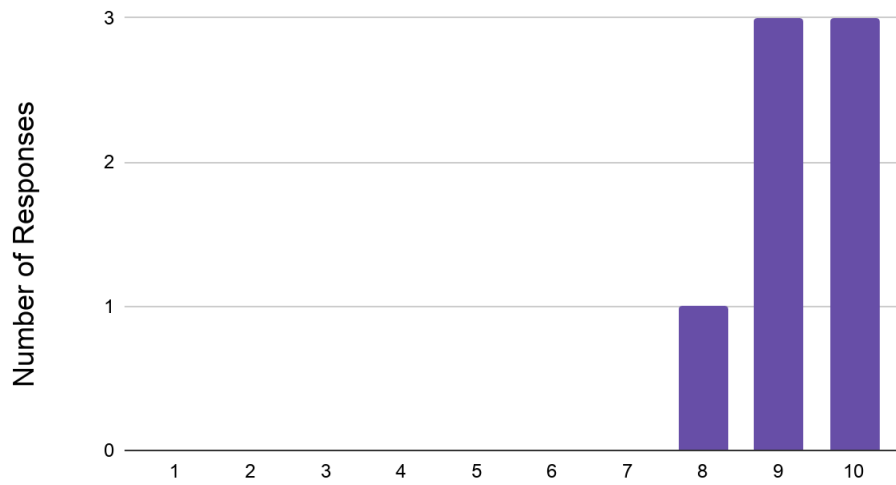


Figure 11: Overall usability ratings for Scootr Studio.

### Scootr Studio Efficiency Rating

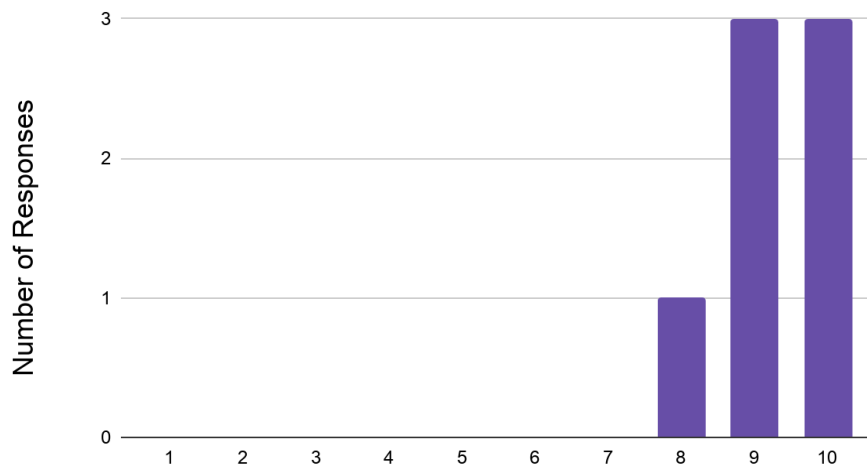


Figure 12: Overall efficiency ratings for Scootr Studio.

The most common feature request for future iterations of Scootr Studio was the addition of a local testing environment. Although the deployment process was simplified significantly by Scootr Studio, participants wished they could have tested their code locally first before deploying

to increase the speed of development and mitigate many of the reported development errors. The most common improvement suggested for an existing feature was the addition of better linting and auto-completion to the code editor. This was also desired in an attempt to reduce the sharp rise in development errors due to the current lack of a sufficiently powerful development environment.

## 5 - Conclusion

Scootr Studio is a next-generation IDE and the first of its class that successfully unifies the design, development, and deployment of microservice-based applications. It enables developers to build microservices faster than alternate methods while eliminating common deployment errors. Scootr Studio accomplishes this by utilizing principles from Domain-Driven Design, Event Storming, and Model-Driven Engineering to create a hosting-provider agnostic Event-Driven Application Architecture Model (EDAAM) that provides helpful abstractions for increasing the speed and agility of application development [Evans, 2003][Brandolini, 2019][Schmidt, 2006].

The system performed successfully during evaluation. Laboratory tests showed a favorable increase in microservice-based application development speed, with users successfully using Scootr Studio to build a microservice-oriented RESTful API 4.4 times faster than with alternate methods. They also showed valuable reductions in design and deployment errors while developing the application. Although errors encountered while writing the code for the API increased when using Scootr Studio, it was discovered that this was primarily due to a lack of experience with using the Amazon Web Services APIs required when using the AWS driver with

the system. This metric also indicates that users are able to spend more time developing, testing, and successfully debugging their system using Scootr Studio instead of wasting valuable time deploying their infrastructure.

The usability survey indicated that, overall, users find Scootr Studio to be highly usable in all four areas of learnability, memorability, efficiency, and error rates. Furthermore, users strongly agree that Scootr Studio reduces the number of errors encountered when building and deploying microservice-based applications. Users rated both the overall usability and the overall efficiency of Scootr Studio as 9.29 out of 10. They suggested that features such as a local development environment and better code support (linting, autocompletion, etc.) would further increase the usability of the system.

Overall, these results indicate that Scootr Studio could serve as a valuable means of easing the burden of microservice development for software engineers. Flattening the learning curve has the potential to increase the adoption of microservices, especially among smaller teams that lack the technical expertise and experience necessary to build high-quality microservice-based systems. The use of the EDAAM also tremendously simplifies the representation of microservices and encourages software developers to think about their system from a high level before implementing any business logic, further increasing the quality of resulting systems.

The results also indicate the Scootr Studio provides a viable basis for expansion into a full-featured integrated development environment (IDE). This will require the addition of new features.

First, the lack of a supported local development environment makes testing without deployment impossible in the current version of Scootr Studio. This is less than ideal in a production scenario, as requiring a full deployment before any testing wastes development time and consumes unnecessary deployment resources. This feature was also requested by several participants in the system's evaluations. Such a feature could help Scootr Studio better conform with best practices, reduce the overall cost of developing an application, and further increase the speed of application development by minimizing the number of deployments.

Second, the current limitations of the code editor make development of the business logic more complicated for software developers who are not familiar with driver APIs or Scootr Studio implementation details. Since Scootr Studio uses the Monaco code editor from Microsoft (the same editor used in the popular extendable text editor Visual Studio Code), the system has the potential to handle additional linting, autocompletion, extensions, and general coding support if the editor is configured to do so during initialization. Such feature additions would further increase development speed by helping Scootr Studio have more of a native IDE feel, prompting users towards valid code that represents their business logic.

Third, the present lack of Continuous Integration/Continuous Deployment (CI/CD) integration also makes additional automation of the development process challenging. Users would have to utilize the hosting-providers built-in CI/CD services or provision their own resources to set up their preferred method of automating their deployments and testing. One way Scootr Studio can help bridge the gap is through integration with Git source code repository providers such as GitHub. Allowing the code and configuration created by Scootr Studio to be

available as version controlled files owned by the developer provides an easy point of integration for triggering builds and running tests on code produced by the system.

Finally, Scootr Studio is targeted primarily at the development of backend technologies. It does not focus on nor have a representation of user interfaces required to use the system being developed, such as web, mobile, and desktop applications. While the details of front-end development is outside the scope of designing, developing, and deploying the architectural representation of microservice-based application, this crucial part of providing software as a service could be provided by another system that integrates with Scootr Studio. Such a system could utilize a similar model-driven user interface for building the layout of front-ends with a built-in code editor allowing the business-logic of each component to be fully customized by the developer.

In conclusion, Scootr Studio is an innovation focused on simplifying the process of microservice-based application development and enabling software developers with various levels of experience to more quickly build their applications and serve their customers. It successfully increases the speed of application development while eliminating errors encountered during the deployment. The addition of new features has the potential to flatten the learning curve of the system and reduce the burden of microservice development on software engineers and architects by a greater degree. The integration of Scootr Studio with additional tools for CI/CD and front-end development has the potential to lead to a revolution in traditional software development methods and the introduction of a new era of agile cloud software development.

## References

1. Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. (2002) “Agile software development methods: Review and Analysis”. *VTT Publications 478*.
2. Adzic, G. and Chatley, R. (2017) “Serverless Computing: Economic and Architectural Impact”. *11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*
3. Amazon Web Services (2020) AWS CloudFormation - Infrastructure as Code & AWS Resource Provisioning. visited 15 Feb 2020. <https://aws.amazon.com/cloudformation/>
4. Amazon Web Services (2020) TCO Calculator. visited 1 Dec 2019. <https://awstcocalculator.com/>
5. Architect (2020) Architect serverless framework. visited 10 Feb 2020. <https://arc.codes/>
6. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Mitchell, N., Muthusmay, V., Rabbah, R., and Slominski, A. (2017) “Serverless Computing: Current Trends and Open Problems”. *Research Advances in Cloud Computing*, 1-20
7. Ballerina. 2020. Let’s Learn Ballerina!. visited 12 Feb 2020. <https://ballerina.io/v1-1/learn/>
8. Bitman, W.R. (1997) “Balancing Software Composition and Inheritance to Improve Reusability, Cost, and Error Rate”. *John Hopkins APL Technical Digest*, 18(4), 485-500
9. Bosch, J., Bosch-Sijtsema, P. (2010) “From integration to composition: On the impact of software product lines, global development and ecosystems”. *Journal of Systems and Software*, 83(1), 67-76



10. Brandolini, A. (2019) *Event Storming*. LeanPub
11. Dimensional Research (2018) “Global Microservices Trends Report”. LightStep.  
<https://go.lightstep.com/global-microservices-trends-report-2018.html>
12. Docker (2020) Docker Documentation. visited 12 Feb 2020. <https://docs.docker.com>
13. Draw.io (2020) Online Diagramming. visited 12 Feb 2020. <https://about.draw.io/>
14. Evans, E. (2003) *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.
15. Ford, N. (2018) *The State of Microservices Maturity: Survey Results*. O’Reilly Media.
16. Gamma, E., Helm R., Johnson, R., Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
17. jsPlumb (2020) jsPlumb Community Edition Documentation. visited 15 Feb 2020.  
<http://jsplumb.github.io/jsplumb/home.html>
18. Kubernetes (2020) Kubernetes Documentation. visited 12 Feb 2020.  
<https://kubernetes.io/docs/home/>
19. Lloyd, J.W. (1994) “Practical Advantages of Declarative Programming”. *Joint Conference on Declarative Programming*
20. Lucidchart (2020) Online Diagramming Software and Visual Solution. visited 12 Feb 2020. <https://www.lucidchart.com/pages/>
21. Ma, X., Li, R., Lu, Z., Lu J., Dong, M. (2011) “Specifying and enforcing the principle of least privilege in role-based access control”. *Concurrency and Computation: Practice and Experience* 32(12), 1313-1331. Wiley Online Library

22. Michelson, B. (2011) “Event-Driven Architecture Overview”. Elemental Links Research, Patricia Seybold Group Research Service
23. Micro (2020) Docs | Micro. visited 12 Feb 2020. <https://micro.mu/docs/index.html>
24. Microsoft Monaco (2020) Monaco Editor. visited 15 Feb 2020. <https://microsoft.github.io/monaco-editor/>
25. Mozilla (2019) Server Sent Events - Web APIs | MDN. visited 15 Feb 2020. [https://developer.mozilla.org/en-US/docs/Web/API/Server-sent\\_events](https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events)
26. Mozilla (2020) HTML Drag and Drop API - Web APIs | MDN. visited 15 Feb 2020. [https://developer.mozilla.org/en-US/docs/Web/API/HTML\\_Drag\\_and\\_Drop\\_API](https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API)
27. NGINX (2016) “The Future of Application Development and Delivery is Now: Containers and Microservices Are Hitting the Mainstream”. NGINX Application Development Survey. <https://www.nginx.com/resources/library/app-dev-survey/>
28. Postman (2020) Postman | The Collaboration Platform for API Development. visited 14 Feb 2020. <https://postman.com>
29. Red Hat (2018) “The State of Microservices”. Red Hat Blog. <https://www.redhat.com/en/blog/state-microservices>
30. Rehman, I., Mirakhorli, M., Nagappan, M., Uulu, A.A., and Thornton, M. (2018) “Roles and impacts of hands-on software architects in five industrial case studies”. *40th International Conference on Software Engineering (ICSE '18)*, 117-127
31. Schmidt, D.C. (2006) “Model-Driven Engineering”. *IEEE Computer Magazine* 39(2), 25-31. IEEE Computer Society

32. Serverless (2020) Serverless - The Serverless Application Framework. visited 12 Feb 2020. <https://serverless.com/>
33. Spillner, J. (2017) “Practical Tooling for Serverless Computing”. *10th International Conference on Utility and Cloud Computing (UCC’17)*, 5–8
34. Stackery (2020) Introduction to Stackery - Stackery Documentation. visited 12 Feb 2020. <https://docs.stackery.io/docs/using-stackery/introduction/>
35. Steinegger, R., Giessler P., Hippchen B., and Abeck S. (2017) “Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications”. SOFTENG: The Third International Conference on Advances and Trends in Software Engineering
36. Su, J. (2019) “Amazon Owns Nearly Half of the Public-Cloud Infrastructure Market Worth Over \$32 Billion: Report”. *Forbes*  
<https://www.forbes.com/sites/jeanbaptiste/2019/08/02/amazon-owns-nearly-half-of-the-public-cloud-infrastructure-market-worth-over-32-billion-report/#5d76fad729e0>
37. Visual Paradigm (2020) UML/Code Generation Software. visited 12 Feb 2020. <https://www.visual-paradigm.com/features/code-engineering-tools/>
38. Zhang S., Zhang X., Ou X., Chen L., Edwards N., Jin J. (2015) “Assessing Attack Surface with Component-Based Package Dependency”. *Network and System Security (NSS ‘15)*. *Lecture Notes in Computer Science*, 9408(), 405-417