Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

DEV-OPS

By
Sean Skelton

April 28, 2025

Minor Dissertation

**Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.**

B.Sc. (Hons) in Software Development

Contents

List of Figures

1 Introduction

What is DevOps? DevOps, while it has no clear definition, is often thought of as [1]: "DevOps is a set of procedures which combines the process of Development and Operations." This methodology aims to create a seamless integration between deployment and operations teams, ultimately fostering a continuous product life cycle that does not break at any point. DevOps is an extension of the Agile Methodology, with a focus on continuous integration, continuous delivery, and automation. Through the use of various tools, DevOps enhances collaboration and automates processes such as building, testing, deployment, and monitoring to streamline software development.

In the context of this project, we can define a successful implementation based on [2], [3], and [4] using the following principles:

- Implementation of Microservice Architecture
- Continuous Integration
- Continuous Deployment
- Automation
- Leveraging IaC (Infrastructure as Code) tooling
- Monitoring

The goal of this project is to apply these practices associated with DevOps to build a web-based messaging and authentication application, designed to showcase these DevOps practices in action. The project will adopt a microservice architecture, leverage Git workflows, Terraform, and Docker as IaC tools for automating continuous integration and continuous deployment workflows. Additionally, AWS will be used for hosting alongside AWS CloudWatch for logging and monitoring. This application will serve as a practical example of how DevOps practices can streamline the software development life-cycle and improve each stage of a project, from feature conception to the full deployment of the application.

2 Methodology

How I began my research approach for this project was by reviewing three papers: one of which was a literature review and case study on six organizations [5], the second was a case study on one company [6], and the last one was a literature review [7]. The reasoning behind the selection of these papers was to cast a wide net in identifying if there is variance between countries and companies in what defines DevOps practices. The combination of these three papers covers the Netherlands, United Kingdom, USA, New Zealand, Germany, India, China, Brazil, Australia, and Finland. The first paper focuses on six companies [5]. The second is a case study on an individual company [6], and the third provides a global perspective [7].

A problem identified in all of these papers is that there currently exists no standardization of what constitutes DevOps—it has not been quantified, and definitions and practices vary from company to company. While I will not attempt to define what DevOps is or is not, I will use this specific combination of papers to provide a basis for identifying the characteristics that separate DevOps from other methodologies and to establish a foundation for identifying the commonalities present in all bodies of research.

The first paper explores how six organizations implemented DevOps and their respective outcomes [5]. It evaluated existing research in the form of a systematic literature review and interviews with six organizations, whereas the second paper opted for a case study approach, looking at a single organization's migration to DevOps and its respective consequences [6]. The third paper was a systematic literature review exploring what DevOps is on a larger, general scale [7]. I believe the strengths in the combined approaches of case studies, interviews, and literature reviews across the papers create a good bridge between the theoretical aspects of DevOps and real-world implementations. The drawback of this approach is that although a large geographical area is covered, seven organizations may be too small a sample size to guarantee an accurate representation of DevOps.

The three pillars of DevOps, as defined in the first paper [5], are cultural philosophy, process integration, and collaborative practice. The first concept, cultural philosophy, emphasizes the importance of shifting organizational mindsets to foster trust, shared responsibility, and alignment between traditionally siloed teams. This cultural approach focuses on creating a unified vision where development and operations work as one, breaking away from rigid, hierarchical structures.

The second concept, process integration, highlights the seamless unification of workflows across the software development lifecycle. By integrating processes like coding, testing, and deployment, the focus is on minimizing inefficiencies and ensuring smoother transitions between stages, enabling faster and more reliable releases.

The final concept, collaborative practice, ties these principles together through actionable methods such as continuous integration, infrastructure as code, and automated testing. This practice not only encourages teams to work together effectively but also ensures that collaboration is embedded into the daily activities of delivering software, reducing friction and fostering innovation. Together, these pillars present a holistic approach to DevOps that prioritizes both cultural transformation and practical execution.

The three pillars of DevOps, as defined in the second paper [6], are tooling and automation achieved through the use of microservice architecture, cloud deployments (AWS, Azure, etc.), IaC tooling like AWS CloudFormation and Terraform, and automation through CI/CD pipelines. All of these are used in conjunction to both speed up the development process, decrease time spent between testing, development, staging, and production environments, and automate this process so one does not have to manually manage pushing changes between the different environments.

The second pillar, cultural collaboration, is more of a company mindset. It is about breaking down barriers between teams through shared goals and ownership of the full development process rather than a more traditional separated approach where teams "ping pong" the application back and forth. This traditional approach was argued to create tension and friction, not collaboration.

The last pillar is the embedded operations model, which unifies these concepts. This involves the use of the tools described above, in conjunction with cultural collaboration, to remove the need for separate teams and unify them under the banner of DevOps. This eliminates the need for separate operations, QA, and development teams, reducing time spent passing code changes between different groups and alleviating friction by fostering shared goals.

The three pillars of DevOps, as defined in the third paper [7], are cultural philosophy, process integration, and collaborative practice. Cultural philosophy emphasizes shifting organizational mindsets to foster trust, shared responsibility, and alignment between traditionally siloed teams, creating a unified vision where development and operations collaborate seamlessly.

Process integration focuses on unifying workflows across the software development lifecycle, such as coding, testing, and deployment, to minimize inefficiencies and enable faster, more reliable releases.

Collaborative practice ties these principles together through actionable methods like continuous integration, infrastructure as code, and automated testing, embedding collaboration into daily activities and fostering innovation. Together, these pillars offer a holistic approach to DevOps, balancing cultural transformation and practical execution.

The three papers define the pillars of DevOps with overlapping themes but distinct emphases. The first paper [5] presents a balanced view, focusing equally on cultural philosophy, process integration, and collaborative practice, highlighting both organizational mindset shifts and practical execution through actionable methods like CI and IaC.

The second paper [6] leans heavily on tooling and automation, emphasizing microservices, cloud deployments, and CI/CD pipelines as enablers of speed and efficiency, while also discussing cultural collaboration and the embedded operations model to unify teams and eliminate silos.

The third paper [7] closely mirrors the first, focusing on cultural transformation, workflow unification, and collaboration but is more concise, omitting detailed examples of tools or organizational models. While the first and third papers emphasize a blend of culture and execution, the second paper prioritizes technical tooling and its integration with cultural collaboration for efficiency and team unification.

As this project was an individual undertaking, I decided to look at DevOps primarily through the lens of automation and tooling. As the sole developer of the project, I did not have to worry about blending teams—I already had the responsibility of developing, deploying, and monitoring. Therefore, I recognized tooling and automation as the most optimal way to achieve this.

These tools are leveraged so that if someone were onboarded to this project at a later date, there would be an established way to develop, deploy, and test the application, allowing them to replicate the application in their own work environment.

All of these papers made many references to different tools. Paper 1 [5] referenced:

- **Microservice Architecture**: A design approach for building applications as a suite of small, independent services that communicate with each other.
- **Cloud Deployments**: Platforms like AWS and Azure for scaling and streamlining infrastructure.
- **IaC Tools**: Examples include Terraform (for infrastructure provisioning and management) and AWS CloudFormation (for setting up AWS resources).
- **CI/CD Pipelines**: For automating and streamlining code changes across environments.

Paper 2 [6] referenced:

- **Terraform** and **AWS CloudFormation**.
- **CI/CD Tools**: Examples include TeamCity (for building and testing code changes) and Octopus Deploy (for deployment automation).

- **Monitoring Tools**: Examples include Datadog (for monitoring infrastructure and logs) and New Relic (for application performance).

Paper 3 [7] referenced:

- **CI/CD Tools**: Examples include Jenkins, GitLab CI, and Travis CI.
- **IaC Tools**: Examples include Ansible, Terraform, and Chef.
- **Version Control Systems**: Examples include Git (and platforms like GitHub, GitLab, and Bitbucket).
- **Monitoring and Logging Tools**: Examples include Prometheus, Grafana, and Splunk.
- **Containerization Tools**: Examples include Docker and Kubernetes.
- **Automated Testing Tools**: Examples include Selenium and JUnit.
- **Collaboration Platforms**: Examples include Slack and Jira.

Through these papers, I gathered tools to help me lay out how I was going to implement DevOps into my project. Using Git as my version control system gave me access to Git Actions to build my CI/CD pipeline. Since I selected Vue as my framework, I opted for Vitest instead of JUnit to allow me to perform both unit tests and component tests.

Docker was used to containerize my frontend and backend, AWS was used to host my containers and database, and AWS EKS (its Kubernetes implementation) was used to orchestrate containers. Terraform was used to automate deployment. This provided a framework to fully develop, test, build, deploy, and automate the process, which I believe meets the criteria for making this a working example of DevOps in practice.

Once my automation and tooling were completed, I began to architect my application. I knew it was going to have an authentication system that I wanted to build myself. I decided to use a JSON Web Token-based authentication system since it is better suited to a microservice architecture.

For user registration, I decided the user would need a password to secure the account, a username for identification, and an email for a point of contact. This determined the structure of my submission form: I would have at least a user table with an ID, username, email, and password. Correspondingly, the login form would take the username to identify who is trying to log in and the password to verify their identity.

For the chat aspect, as discussed in the technology review, many chat applications (e.g., Twilio and Sendbird) use Web Sockets for real-time communication.

Web Sockets were chosen to manage the chat aspect of my application, as this approach avoided issues with free tiers, usage rates, and documentation.

During development, I did not initially have a CI/CD pipeline set up, as my focus was on building the application. I settled on a feature branch approach in my version control system, Git. Each aspect of the application (login, register, chat) was broken into corresponding branches for the frontend and backend, with each being merged into the main branch upon completion. Once the application was finished, I built the application.

The testing methodology was straightforward. It involved unit tests that tested the happy paths for all the components in the frontend. Each test began with a mocked action, with the exception of the WebSocket test, which started with a mocked WebSocket. A real store was then created with the mocked actions, mutations, and necessary states. The component to be tested was mounted with the state onto the application, and a fake view was created with the function to be tested. The evaluation of this function was then checked, and where applicable, the state or component was verified for the relevant updates.

3 Technology Review

4 Front End

Vue.js is a front-end JavaScript framework. I selected this framework because it is a fast, lightweight solution that is well-suited for single-page applications, offering routing support to navigate and render UI components, local state contained within components, and support for global state management with Vuex [8]. Since this project's emphasis is neither DevOps nor the application logic itself, Vue.js was a favorable choice over both React and Angular [9] [10]. Although React and Angular have larger ecosystems, they are generally considered more verbose and complex compared to Vue.js, requiring more boilerplate code to get the application up and running.

Axios is an extremely popular framework for asynchronous HTTP communication between the backend and the server. It operates using promises, which either resolve to a response object containing the HTTP response, or, if the request fails, return an Axios error object explaining the reason for failure. As I was implementing an auth API, I knew I would need to send and receive HTTP requests, and I did not want my application to cease functioning if the requests failed. application up and running.

I considered using the native JavaScript Fetch API, which is also a promise-based HTTP API (https://medium.com/@johnnyJK/axios-vs-fetch-api-selecting-the-right-tool-for-http-requests-ecb14e39e285). However, it quickly became clear to me that although Fetch offers more customization and the benefit of not needing to install dependencies, it requires more lines of code to achieve the same functionality. Fetch does not automatically catch errors, and while Axios automatically transforms the server response into a response object, with Fetch this must be done manually. Additionally, Axios uses only one promise, whereas Fetch often requires two. This led me to conclude that Axios was more in line with what this project required.
.
I knew I needed a way to show/hide endpoints based on if a user was logged in or not and to block access to those endpoints until a user is logged. Given id worked out in theory my password encryption. I was thinking of when a user logs in I could have global vue store vuex have a boolen for when a user is or is not logged in and pass this boolean with http requests to the server. Upon a bit of experimentation I realized this was not a very secure approach as someone could just past this boolean into a http request.

I thought about requiring the username and encrypted password to be sent with every request, but this would produce unnecessary overhead I wasn't to sure how id then logout and felt like there was probably a simpler solution. https://supertokens.com/blog/token-based-authentication-vs-session-based-authentication I came across a session based approach which would involve a login session that would be created in the backend from a successful login in the frontend and this session propagated back. Something mentioned in the article was how this is beneficial for a monolithic architecture and simpler setup, While Ive been favouring a simpler approach so far to get onto the DevOps tooling, Managing global state in the backend is not something I know alot about at this point and is something that might introduce alot more complexity into my application when I begin splting the architecture.

The article then discussed. A token based approach and how this is better suited to a microserice architecture. https://zuplo.com/blog/2025/01/03/top-7-api-authentication-methods-compared this was further supported in another article I came across. Both of these articles talked about the fact that tokens are self contained requring no data. They can be generated in the backend upon login request and sent back, this token id imagine through its generation method can be re-verified on the backend to access protected routes and used to change states between logged in, logged out and to Log out. While I'm not completely sure of the implementation yes it seems to strike a balance of security whilst not being as complex as using API keys or or 0Auth2.0 header.

5 Backend

I selected Node.Js for my my backend as Id felt that implementing the auth system, and websockets and fully deploying this application had good balance of complexity in architecutre and simplicity in implementation. In staying with this I feet keeping the same language for the backend and fronted reduces the burdern of juggling two languages and promoting code resuse.
I settled on the idea of using [11] Bycrypt to hash user passwords on the Backend. I needed some mechanism to safely store passwords. If the password is not encrypted, if someone had managed to to the database they would have the credentials necessary to hijack someones account. This achieves security through hashing the password with a salt which is random value added to the password before hashing and sorting this hash in the database. How I intended to use this is When someone registers bycrpyt will add this random value to the password in

the backend and hash it. Store the password in the relevent part of the database. When a login attempt is made, bycrypt extracts the this random value (the salt) from the password stored in the database, adds it to the password in the login attempt, rehashes it and checks if the hashed value matches what is in the database. If the username and password match the user will successfully login. Id seen that Auth0 use this under the hood https://auth0.com/ as they are prominent in the secure authorization industry aswell as it being a straight forward to use I figured it fit aligned with my application.

Socket.IO is a web socket library and will be used in both the backend and frontend. Once I have my authorization built. I want to have protecteds page for viewing chatrooms and for users to message back and forth. There are APIs such as Twilio, Tencent Cloud, and Sendbird https://sendbird.com/pricing/chat that support point-to-point messaging, they are locked behind paywalls , https://login.twilio.com/u/signup offer free tiers that are bare-bones or missing features or free trials that expire. Given that WebSockets form the backbone of most real-time communication systems, I opted to use the official WebSockets API and created my own messaging service that, while lacking aesthetics, has the core functionality of messaging systems. I feel like ive swapped out enough complexity in place of simplicity that I can afford this change.

When it comes to databases, the most popular there are relational and non-relational (NoSQL) databases. Given the fact I knew I was going to have to register, log in, create chatrooms, and send messages, it made sense to me that one user would register/log in but would need to have multiple chatrooms and send multiple messages, already showing that the data would be relational. MySQL was chosen due to it being the database I had the most exposure to. aligned with my application.

As node.js is not type safe and SQL being a stricter database, I wanted to ensure I could programatically generate and interaect with the database and respective tables to avoid manual creation of resources and so I could migrate this over to a production environment when the tie came My previous experience with object orientated porgramming java exposed to be object relational mappers for databases which I felt fit this criteria. Additionally I wanted something that would give me the model methods that I could invoke so I did not have to declare them which let me to Sequelize. Which is ORM (Object-Relational Mapper) that met these criteria. It didnt take me long to find and as its getting later into the semester I still working on technology selection I want to speed up this process to begin development.

I settled on AWS for hosting and deploying my application. While Its a total behemoth, it seems well document it has Kubernetes implementation and it has tools to automate deployments. It seems to be cheaper then [12] and far more user-friendly than [13] azure web services. It wont be till later point that Im fully deploying my application so this may change but for now its planned.

GitHub was chosen as the source control. I initially started off with GitLab due to just wanting to try something new and it has pheonmonal integrated tooling for DevOps. With a working pipeline out of the box. I want to stress that is entirely opinated, but while it was alot of powerful tools, its UI is clunky, slow and was diffiuclt for me to navigate and get a grasp on. While I can acknowlege it may have been a better suited suited for this project, I felt would have been alot of spent getting proficient in this tool that may have come at cost of some of the deliverables.

GitHub Actions I decided to use to create and manage the CI/CD pipeline for this application. Given the fact I was using GitHub, it made sense to integrate a tool from within the Git ecosystem to manage my pipeline.

Express is one of the most, if not the most, popular frameworks for server-end development in js. It simplifies server setup by provides a myriad of methods for web servers including but not limited to routing, configuring cors options. These things are paramount for most back-ends, in from previous experience I knew Id need the server setup functions and routing at the very least.

As each Application requires testing and it will part of the final pipeline. Given the frontend is in Vue I thought id use vi-test as it was made by the createer of vue for component tests and unit test as its designed to work with vue components. I want to have some end-end testing in this application but I don't know enough about that process to make a descion yet so I will revist this when I have backend and frontend working and communcation.

`https://www.docker.com/` Given that I was opting for a microservice architecture for my system and would have to deploy my application I figured Docker was the natural choice. `https://srivastavayushmaan1347.medium.com/how-companies-are-leveraging` Countless companies seem to use Docker for their microservice architecture. Images are like a set of instructions to create an instance of a piece of an application. Containers are what execute thse resocues to create an instace of that piece of an application. `https://www.freecodecamp.org/news/how-docker-containers-work/`

I know a lot of this technology review focused on the implementation of my application rather then the DevOps tooling, I feel more time is needed to research the technologies that exist on AWS, that I need something that is production ready to for a pipeline that automating this process will depending on having gone through all of this manually to gather a better understanding and apply at a later stage in my project.

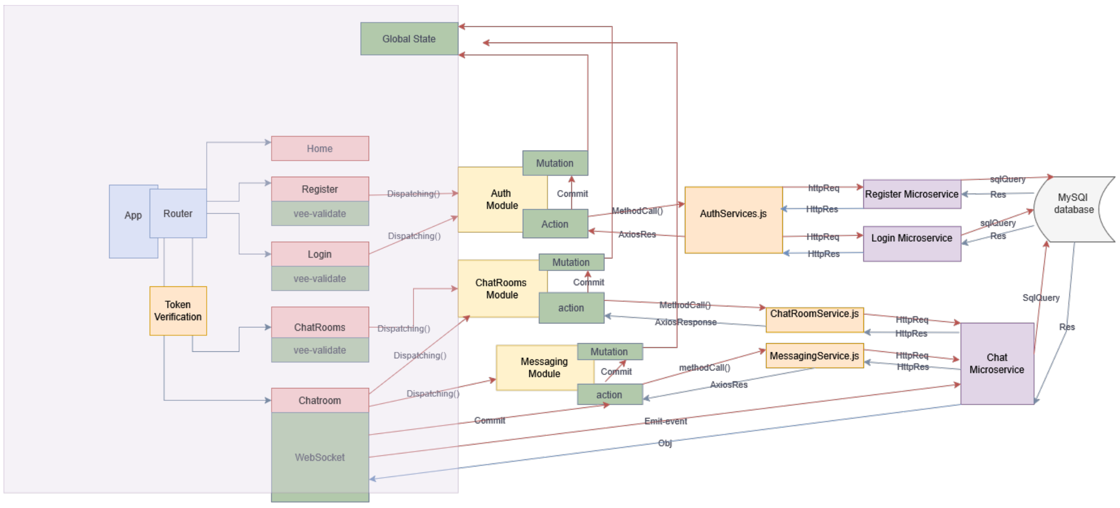

Figure 1: Front-end Architecture

6 Design

Just before I talk about the design, I want to explain Vue's answer to managing the global state of the application. For component lifecycles such as `onCreate`, or user-defined methods such as my login that is triggered on the submission of the form, `this.$store.dispatch/module/action` is a way of calling an entry in a list of actions that, for most of this application, are just promises returned by an Axios HTTP call. Based on this, it will commit `actionSuccess`/`actionFailure` mutations that change the UI via global state. For instance, deleting a chatroom will remove it from the view, and logging in as someone else will change the user data in local storage, etc.

What is Dispatch? [14] Dispatch, in the context of Vue, is a defined way of interacting with the Vuex store in Vue.js. It maps to a corresponding key-value pair in actions. The key is a string that is the name of an action, and its value is the function being called.

What is a Mutation? A mutation is what the result of an action does. This mutation is usually based on a success or failure mechanism that will impact something the user sees or change how the user interacts with something.

This provides an overview of the front-end architecture. We are going to go from left to right, going through the application row by row. We have the root of the application called `App`, which contains the routing. The router is Vue.js's way of rendering components based on links the user clicks. The chatrooms and navigating to an individual chatroom are not visible to the user until they are logged in, and a token check is performed if they try to access the URLs. It has a logout link, which is not a component but a method, which will be touched on when we talk about global state.

```
<template>
    <div id="app">
        <h1>Welcome</h1>
        <div id="nav">
            <!-- If the user is not logged in these are the routes they can access
            <template v-if="!isLoggedIn">
                <router-link to="/">Home</router-link> |
                <router-link to="/register">Register</router-link> |
                <router-link to="/login">Login</router-link> |
            </template>
            <!-- If the user is logged in these are the routes they can access -->
            <template v-if="isLoggedIn">
                <router-link to="/chatRoom">Chatrooms</router-link>
                <a href="#" @click.prevent="logout">Logout</a>
            </template>
        </div>
        <router-view />
    </div>
</template>
```

This is the skeleton of the router, which maps our components to their corresponding paths that are turned into clickable links. You can see the protected paths, which are checking for the JSON web token.

```
const routes = [
  { path: "/", name: "Home", component: Home },
  { path: "/register", name: "Register", component: Register },
  { path: "/login", name: "Login", component: Login },
  { path: "/chatRoom", name: "Chatroom", component: ChatRoom, meta: { requiresAuth
  { path: '/chatroom/:id', name: 'ChatRoomView', component: ChatRoomView, meta: { r
]
```

When you navigate to the Register component, you will be met with a form that has three visible inputs where you enter a username, email, and password.

All three have attributes: a label to identify them, fields for the input, and an error message which we get from `vee-validate`. There is also a register button, which, when clicked, will trigger the register method that dispatches the call to the store. [15] This was implemented as a combination of `vee-validate` and Yup, when researching how to enforce required fields for user input parameters.

```
<Form @submit="Register" :validation-schema="schema">
    <div>
        <label for="username">Username</label>
        <Field name="username" type="text" />
        <ErrorMessage name="username" />

        <label for="email">Email</label>
        <Field name="email" type="email" />
        <ErrorMessage name="email" />

        <label for="password">Password</label>
        <Field name="password" type="password" />
        <ErrorMessage name="password" />

        <button :disabled="loading">
            <span v-show="loading"></span>
            Register
        </button>
    </div>
</Form>
```

The form validation is done through the Yup package, which allows you to define the rules for the respective fields, such as "cannot be empty," "emails must have @", "password must be more than 10 characters," etc. Based on those rules, it also produces the relevant error messages that are defined. `Vee-validate` provides the attributes, and a schema is defined and built with Yup, which `vee-validate` uses to validate the form.

```
data(){
  return {
    schema: yup.object().shape({
      username: yup.string().required('Username is required'),
      password: yup.string().required('Password is required'),
      email: yup.string().required('Email is required')
    }),
  }
}
```

The other part of this is the local state.

```
message: "",
successful: false,
loading: false,
```

The `loading` is set to true upon submitting the register form. This disables the button until the `successful` value is updated. If `successful` is true, you will see a "User successfully registered" message. If `successful` is false, the error message is displayed. In both cases, the button becomes available again to register a user.

```
Register(user){
  console.log('Register method triggered with:', user)
  this.message = ""
  this.successful = false
  this.loading = true

  this.$store.dispatch("auth/register", user).then(
    (data) => {
      console.log("Dispatching register action with:", user)
      this.message = data.message
      this.successful = true
      this.loading = false
    },
    (error) => {
      console.log("Dispatching error", user)
      this.message = (error.response && error.response.data && error.response.data
      error.message ||
      error.toString()
      this.successful = false
      this.loading = false
    }
  )
}
```

The login format follows a very similar architecture, with the only difference being that a JSON web token is sent back from the web server. This token now allows access to the protected routes and updates the global state so each component, when loaded, will identify who is logged in through the web token.

Once logged in and on the chatroom page, there are methods in this component that dispatch actions to the store. These actions can pass no parameters, a room object, or a room ID. These actions call service functions that return promises. These promises either resolve to an Axios error message (indicating a request failure, but not necessarily an HTTP error like 404) or return the Axios response object. The response object will either contain relevant HTTP status codes or messages to be logged in the console or displayed to the user, as well as data to trigger mutations in the store. These mutations modify the global state of the application to:

- Render all the chatrooms.
- Create, join, leave, or delete chatrooms.
- Allow users to join chatrooms in the database.

Lastly, there is the chatroom page. If a user is a member and joins the page, the WebSocket connection emits a connection event with the user's token to the WebSocket server. Then, a dispatch is made with the `roomId` to the store, which routes the request to the messaging service. This service forwards the request to the backend, which sends back the room name and room description to the controller. The controller sends it to the store, which updates the state of the chat by adding the room name and description. It emits a `joined room` event to the WebSocket server.

The front-end is then listening for connection and disconnection events. There is also a delete message event where the user can delete messages they sent. This will commit a mutation straight away, removing them from the store. A `receive message` event checks whether the incoming message is unique before adding it to the store to avoid duplicate messages being displayed to the user.

The architecture for the front-end was based on the clean architecture idea: [16]. The entire point of this architecture is the separation of concerns, so that each layer can be tested independently and changes in one layer will not affect the others. This was achieved through the usage of the model-view-viewmodel (MVVM) architecture:[?].. Each component of the application had a corresponding UI (e.g., `register.vue`, which was the UI), the model (business logic, which is done via `auth.service` for sending HTTP requests), and the view-model (`auth.module.js`, which acts as the bridge between the two). The UI (view) sends a registered user via Vuex (view-model) to the service. The service returns a response to Vuex (view-model), which then affects changes in the UI (view).

With each layer contained, it made troubleshooting the UI easier when unexpected behavior was experienced. It also made testing the components easier.

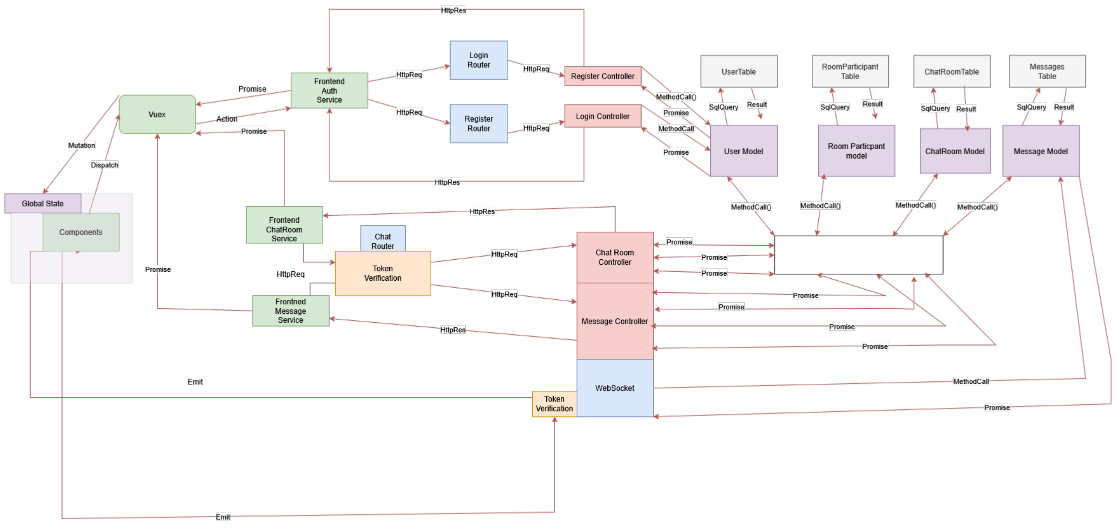

Figure 2: Backend Architecture

This approach implies that changes can be made to one part of the application without affecting the others. For example, the backend ports might change, but only the service needs to know. We would just need to make changes in the auth module. If the UI needs to be modified, we can do that in its view component. This is the benefit of this architectural pattern.

[17] Microservices architecture was used in the backend to provide a scalable application. By splitting the backend into three separate services that run independently of each other, it means that the chat service might go down, but new users can still register. Conversely, the register service might go down, but existing users can still log in and chat. It promotes the single responsibility principle: [18]. Historically, backends could have comprised mostly replicated monoliths (reference needed). Rather than having these massive servers, each one could be broken down into exactly what is pertinent and what is not. This architectural approach is good but also introduces more complexity from a design perspective, where you now have to build additional deployment steps into the pipeline. When deploying this application, it adds complexity to routing and costs but provides a more scalable and decoupled application.

The register microservice receives an HTTP POST request from the front-end. This request is comprised of a user object with a username, email, and password. The router invokes a verify register method that checks if the user's username or password is already in the database. This is then routed to the controller, which calls the create function on the model. The model deconstructs the user object into a username, email, and password, hashes the password, and converts this into a MySQL insert query. This populates the database with a new user and sends back a message notifying whether it succeeded or failed.

For the login microservice, it receives an HTTP POST request from the front-end comprised of a user object with a username and password. The login router routes the request to the login controller. The controller separates the username from the login request and calls the `findOne` function on the User table, converting it into an SQL query via Sequelize to see if the user exists. If they do, it moves on to comparing the hashed passwords (as described in the technology review). If successful, it creates a response object comprised of HTTP status 200, a JSON web token, and the user's email and username. This is sent back to the login service in the front-end.

The chat microservice is comprised of two aspects: one for chatrooms and the other for sending messages in the respective chatrooms. The routes for this are all protected, which means that a user must be logged in and have a JSON token that was created at login and sent with the request. This token is validated by checking its signature to verify if it was signed with the secret stored in the backend.

There are routes for `GET`, `POST`, and `DELETE` requests that all match the create, join, navigate, and delete requests from the chatroom service in the front-end. These all map to the controller methods. The first parses the room name from the request body in the create chatroom request and ensures it is not empty. The room name is passed to the `findOne` function, which Sequelize converts into a MySQL query to check if the chatrooms table already exists. If it does, an HTTP status code 400 is sent back alongside an error message. If it doesn't, a chatroom object is created with the parameters from the request. This chatroom is passed as a parameter to the create function in the model, which adds the chatroom to the database. It then calls the creation function on the `roomParticipant` model, passing the `userId` and `roomID` as parameters to add the user to the room participants for that room. Then, a success HTTP response is sent back to the controller, which sends it back to the chat service in the front-end. This updates the state with the new room and the creator as a participant.

on and on the chatroom page. Theres methods this compontent that dispatch actions to the store that passes no parameters, passes a room object, and a room ID. These actions call service functions that return promises that either resolve to an axios error message which is the request failing to send not an error like 404 etc. The axios response object will either contain the relevant http status codes,messages to be logged in the console or displyed to the user and a response object containing the data to trigger mutations in the store to modify the global state of the application to 1. Render all the chatrooms, To create, Join, Leave and delete chatrooms. all the chat Rooms in the database, Allow a user to join a chatroom

The second allows a user to join a chatroom. It gets the room ID from the URL request parameters and the user ID from the request. It uses to call the ChatRoomModel,RoomParticpant model with the function findOne passing the roomId to the chatRoom and to check if the table exists see if the user is/is not a member of the chatroom. If they are, an error message is propagated back controller that sends it back to the chat service. frontend; if not, the user is added as a member of the chatroom and a success message is propagated back to the frontend. The third API endpoint calls the `findAll` method on the ChatRooms model and returns all the chatrooms in the database to the frontend.

The fourth endpoint finds an individual chatroom in the database. It parses the chatroom ID from the request body and calls the `findOne` function on the ChatRooms model, which builds an SQL query to find the individual chatroom.

The fifth endpoint deletes a user from a chatroom. It parses out the room ID and user ID from the request parameters and calls the `destroy` method on the RoomParticipant model, passing in the room ID and user ID as parameters to the destroy method. This builds the query to remove the user. If the user is not a member, a message is propagated back to the controller that sends it to the chat service. If it succeeds, the success response is sent back to the controller, which forwards it to the messaging service.

The sixth endpoint deletes a chatroom from the database. It parses the room ID from the request parameters and the user ID from the request. It queries the database with the room ID to find it and checks the user ID to verify if they created it. If they were the creator, the chatroom is removed from the database; otherwise, an error message is propagated back.

For the chatroom page, when a user who is a member joins, the WebSocket connection emits a connection event with the user's token to the WebSocket server. Then a dispatch is made with `roomId` to the store, which routes the request to the messaging service. This service routes the request to the backend, which sends back the room name and room description to the controller. The controller forwards this to the store, which updates the chat state by adding the room name and description. It emits a "joined room" event to the WebSocket server. The frontend listens for connection and disconnection events. There is also a delete message event where users can delete messages they sent, which commits a mutation immediately, removing them from the store. The receive message event checks that incoming messages are unique before adding them to the store to avoid duplicate messages in the user view.

The routes for these operations are all protected in the same way as the chatrooms.

There are routes for GET, POST, and DELETE requests that match their respective controller methods. The first route creates new messages from the request body in the database if the WebSocket fails. If successful, it adds the message to the chatroom and propagates that message back to the frontend.

The second route returns all messages in a specific chatroom. It gets the room ID from the request parameters and the user ID from the request. It checks the room participant table to verify if the user is a member of the chatroom. If not, an error message is propagated back to the frontend. If they are a participant, the system retrieves all messages for that chatroom from the database, including user details, and returns a success message to the frontend.

The third route deletes a message from the database. It parses the message ID from the request parameters and the user ID from the request. It queries the database with the message ID to find it and checks if the user created the message. If they did, the message is removed from the database and a success response is sent back. If not, an error message is propagated to the frontend.

The WebSocket authenticates each user who connects to a room, listens for send message events, calls the message model with the parameters from the message, creates a record in the database, emits a received event message with metadata, and listens for delete message events which call the model's destroy method with the message parameters and returns a delete message event to the view.

The CI/CD pipeline for the frontend runs unit tests, builds the Docker image from the Dockerfile, and pushes the image to the ECR repository.

Terraform scripts then create the relevant scripts to deploy the application to the EKS cluster.

The CI/CD pipeline for the backend builds Docker images from the Dockerfiles for each microservice and pushes the images to the respective ECR repositories.

Terraform then creates the relevant scripts to deploy the microservices to the EKS cluster.

This covers the design, but next I will discuss the corresponding Terraform scripts and examine the GitHub workflows file.

The mainEKS.tf creates the EKS cluster. It specifies the region on AWS where the resources will be created, stores the cluster name locally, and gets information about the existing VPC the cluster will be created on.

A VPC [19] is a private, secure network specifically for your AWS resources to operate on.

All the public subnets for the VPC are then found; the subnets are sections of the VPC responsible for internet access to the various parts of AWS services.

Terraform has a module that abstracts the creation of the EKS cluster. It uses this module alongside the cluster name, VPC, and subnets, as well as creating node groups. These node groups run pods that host and execute the Docker image in your ECR repo.

A role is created on AWS that manages the load balancer.

Then a tag is added to the subnets as a way for the Kubernetes cluster and load balancer to identify and integrate with each other.

For the frontend Terraform file, a connection is created to the Kubernetes cluster. It gets information about the cluster, creates the deployment file that sets up an identifier, allocates resources to run it, and adds a health check to ensure the website is accessible once deployment completes. Then a service file of type ClusterIP is created for the frontend, which enables communication with the microservices in the cluster.

There is a horizontal scaling mechanism so that if the website experiences higher traffic than usual, copies of the website are created for other users to access.

A load balancer is then created and configured to make the website publicly accessible and allow the frontend to communicate with the backend via HTTP requests. As part of this configuration, sticky sessions are added. When you have two or more instances of the same service, if traffic is re-routed to another instance, this is acceptable for HTTP requests as they are short-lived, but with WebSockets that require a continuous connection, such re-routing would break the connection. Sticky sessions ensure that traffic for a WebSocket connection is always routed to the same instance.

Paths are then added that correlate to the endpoints of the different microservices. This architectural approach is good practice, as using ClusterIPs internally and the load balancer with defined routes for public access means that when the application is redeployed, the URL might be different but the changes are handled automatically internally.

The Terraform files all follow the same format for the backend, defining the region where resources are created, establishing a connection to the EKS cluster, gathering information about the existing cluster, and creating the deployments which label the deployment to identify it, pull its image from the ECR repo, specify the port to run it on, and allocate computational resources.

Then a service file configures them as type ClusterIP. External load balancers are unnecessary for these services as only resources within the cluster need to access them.

The Terraform script for the database specifies which subnets it will be deployed on.

The security group for the database places it on the same VPC as the EKS cluster and defines an ingress that allows MySQL requests from the services within the cluster to the database and allows the database to send responses back to anywhere.

A parameter group is defined that determines the database engine to be used, the character encoding standard, and a maximum database connection limit.

Then it defines the blueprint for creating the database and establishes the role and policy to allow AWS to manage the database instance.

The workflows for both services follow the same format, with the difference being that the test job was not present in the backend pipeline.

There are four jobs in the frontend pipeline and three in the backend pipeline. Each job first clones the repository into the runner environment. The unit test job installs Node.js v20, installs the necessary dependencies, and runs the Vue unit and component tests.

Once that completes, the build job starts, obtaining the credentials to log in to AWS (stored in GitHub secrets for security). It installs the necessary actions to build and push Docker files, locates the Dockerfile in the directory, and pushes it to its respective ECR repository.

The deploy job then runs, obtaining AWS credentials, installing Terraform, initializing it, deleting AWS resources associated with the frontend deployment, and executing terraform apply, which redeploys the frontend application.

This process is replicated for the backend, with these steps repeated for each microservice but without the test job.

What is Terraform? Terraform [20] is a tool that lets you "define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share." It is another result of the IaC movement. What this means in the context of this application is that, rather than having to navigate the AWS console and manually create a Kubernetes cluster, manually configure this cluster, then manually configure roles, users, install plugins, and manually configure routing for public access as well as routing for internal communication, this can be quite an error-prone process. Configurations aside, once the information is input, the actual generation of the cluster and other components can take upwards of 20 minutes [21].

So the Terraform scripts essentially provide a set of instructions for deploying the frontend and microservices for the application. The EKS cluster and database were already created, with the EKS cluster configured to use two node groups unless traffic spikes. The Terraform scripts for the backend automate the creation of the pods via deployment scripts. My pipeline executes these scripts, pushing a new image to the ECR repository, which pulls the new Docker image configuration into the deployment script. The nodes provide the computational resources, and the deployment scripts create the pods that spin up a Docker container. The replicas for each backend service are set to two. The pods and replicas are configured so that EKS tries to maintain a copy of each node group. Note this is not guaranteed every time without special configuration. If I deploy this tomorrow, it could result in two chat services and two register services in one pod, etc. The service scripts created by Terraform for each backend component define ClusterIPs which allow discovery through resources inside the cluster but not outside the cluster.

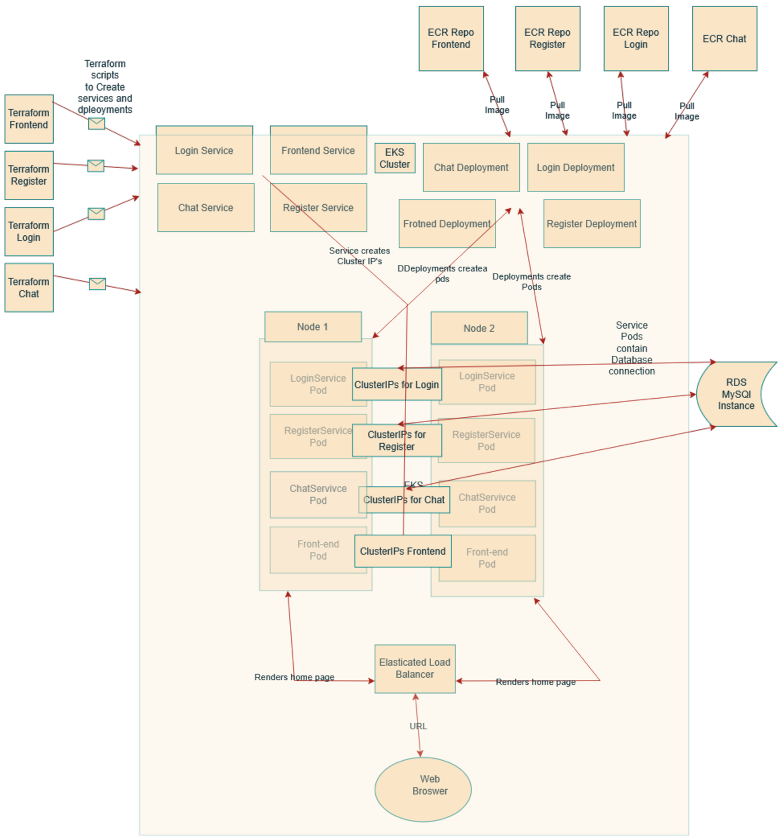

Figure 3: Deployment Architecture

The frontend deployment is similar in terms of replicas, with a deployment and service script, but it also defines a horizontal pod scaler which will add up to four copies of the frontend under high traffic to ensure consistent state for all users. It uses a load balancer of type ingress to provide the frontend with a publicly accessible HTTP IP. This is how users interact with the application. It includes routing that uses relative paths and ClusterIPs to connect to microservices that use their own cluster IPs to route traffic to and from the MySQL instance back to the frontend.

7 System Evaluation

My evaluation is based on my implementation of Microservice Architecture, Continuous Integration, Continuous Deployment, Automation, Usage of IaC (Infrastructure as Code), and monitoring.

The microservice architecture was partially delivered on, as each backend service had no dependency on the others and all were running on different ports, meaning that any one service could fail while the others would remain operational. However, the coupling with the database configuration, which will be discussed below, unfortunately eliminates some of the benefits of microservices.

Ideally, I would have split the frontend login, register, and chat into separate micro frontends with separate deployments, and each service having its own database. I will defend my decision to just break up the backend, with one frontend and one database. I acknowledge that having everything split into its own service is architecturally the better solution, but this comes with increased complexity and cost. The general consensus is to not migrate to something like this until the application actually requires it.

The continuous integration was not completely delivered on. The parts that were delivered included frequent code integration through continuous pushes to both feature and the main branch on Git, and automated builds through the pipeline, which, if for any reason failed, gave instant feedback. While automated testing was part of the frontend, it unfortunately did not make it into the backend, instead relying on manual testing through running the Docker containers, Postman requests, and an HTML page for the WebSocket. The testing for the frontend was not very verbose, only testing the happy paths for each component and not the sad paths or edge cases. I wouldn't say I achieved total success in continuous integration for the frontend, but I would say I had all the key components with gaps in automated testing. In the backend, I was unfortunately missing a key component, which was automated testing, only partially delivering on this metric.

The continuous deployment aspect of this application was fully delivered on both the frontend and backend, albeit with room for improvement. When changes are pushed to the repository, the pipeline triggers with a downtime of approximately 2 minutes before both the frontend and backend are fully live with no manual intervention.

Full automation is implicit in the frontend, as all this is done through the CI/CD pipeline, which takes care of the testing, building, and deployment of the application, and was partially achieved in the backend due to the absence of automated tests.

The leveraging of IaC was achieved through writing Terraform scripts [20] that created my EKS cluster, RDS database, deployed my frontend and backend, and writing Docker files to package my frontend and backend. The combination of

Docker and Terraform allowed me to automate my creation and configuration for all my AWS resources, as well as manage the state of the deployment.

Monitoring came in two forms: with CloudWatch [22], which is integrated with AWS services and allows me to check logs for any of the services on AWS, such as my cluster, node groups, pods, deployments, database instance, etc. The second way monitoring is achieved is if my pipeline fails or breaks, there is an automated email sent to my address.

While I feel I delivered on most of my objectives, once I began preparing my code for deployment and migrating from my local development environment to a production environment, I realized that there were decisions I made in the architecture of the application whose consequences only became apparent in this migration.

For one, when I initially built the backend, I'd started it as a monolith and included a directory called `config` that contained everything for my local database connection. When I began breaking this up into microservices, I kept this directory and just imported it into the respective microservices. When it came to the deployment of the application, I realized that in order for these services to work, I had to package the config file with all the different services, which created tight coupling and code duplication, which is resource inefficient. In retrospect, I should have leveraged some form of centralized configuration management, such as AWS Secrets Manager [23], which would mean my microservices could have fetched these details via an AWS API and would have eliminated the tight coupling and code duplication.

Another change I would have made was when I initially began building the backend, I focused on writing my own servers. I wanted to implement everything from scratch both as a learning process and to have full customization of the backend.

When I was experimenting with AWS, I came across AWS Lambdas [24], which are stateless serverless functions hosted on AWS. While the chat service has state, such as tracking users in a room, maintaining message history and a WebSocket connection, the login and register services both essentially had one responsibility: receiving a register or login request and returning a response.

Theoretically, I could have extracted the controller logic for both services and created Lambda functions for both, configured an API Gateway that would take the request, pass it to the Lambda function that would perform the necessary operation on the database, and return the response to the API Gateway, which would then send the response to the frontend.

By that point, I had invested a large amount of time and effort into my project, and making this kind of change halfway through was a risky gamble that may not have paid off. If I hadn't focused exclusively on creating and deploying my own servers, and had used AWS Lambdas for my register and login service earlier, it would have simplified much of my development and deployment on the backend, and allowed more time to work on the other parts of this project.

8 Conclusion

My deliverables for this project were as follows:

-Implementation of Microservice Architecture
-Continuous Integration
-Continuous Deployment
-Automation
-Leveraging IaC (Infrastructure as Code) tooling
-Monitoring

In evaluating my system, I found that I successfully implemented a microservice architecture, albeit with inefficiencies. I had to package a configuration file with all the microservices.

I created continuous integration and continuous deployment pipelines for both the front-end and backend. However, these pipelines had some inefficiencies. For instance, I had to delete services and deployments instead of simply applying changes. Additionally, the backend lacked automated tests. Instead, I relied on testing endpoints with Postman, testing Web Sockets using a bare-bones HTML page, and manually validating controllers by interacting with the backend through the front end console, logging requests, responses, and Web-socket events.

Automation was largely achieved in the building and deployment phases of this application. I successfully leveraged Terraform as the IaC tool to accomplish these tasks. I opted for Terraform because AWS does not support the pausing of resources, which comes with costs that Terraform is able to offset considerably.

I believe where I fell short in my project was task prioritization. I sank a lot of time into trying to manually provision AWS resources very early in my development. While this did pay off when it came to automating my cloud infrastructure, as it gave me an understanding of the process, I spent a lot of time troubleshooting issues before I had a full application.

I was troubleshooting cluster creation, node group creation, and pod communication, then manually creating service and deployment files for my front-end to make it publicly accessible. A lot of these problems stemmed from a lack of experience. The cluster creation and node group creation issues arose because I opted for creating the associated roles and policies myself, rather than letting AWS handle them. The internal communication problems were due to the version of the Elastic Kubernetes Service no longer including the plugins kube-proxy, AWS-CNI and AWS-DNS, all of which are essential for internal communication. I only figured this out when I SSHed into one of the pods and encountered a "no CNI configured" error.

I managed to successfully deploy my frontend and database, but the backend was up and not accessible. At this stage, costs were starting to accumulate, so I had to dismantle my infrastructure. I then rebuilt my application, getting it into a production-ready state, only to encounter the same problems again. This led me to discover Terraform, and I began writing scripts to manage the building and deployment of my application and the AWS services. Using Terraform, I eventually got back to where I was, and not having to manually create and delete resources saved a lot of time. However, I still couldn't connect to anything.

The first breakthrough came when I separated frontend creation from Kubernetes deployment and realized that my Kubernetes instance had a TLS flag, meaning only HTTPS requests could go through the cluster. This required additional configuration. After removing the TLS requirement and instead requiring AWS authorization, I was able to connect to the cluster.

For the database, I found discrepancies in my configuration. I had configured it to allow traffic from a CIDR block, which defines the range of IPs that Kubernetes pods run on. However, the CNI was assigning the pods IP addresses based on the subnets of the network the cluster was on. After changing the database to allow connections from the pods' IP addresses, I was able to spin up a pod and connect to the database. From there, I successfully deployed my microservices and frontend, and after some tweaking, I had my application fully deployed.

Another area where I fell short was end-to-end testing with Cypress and unit tests for the Vuex store. I spent a large amount of time trying to get Cypress working locally. I was attempting to test the conditional rendering based on successful or failed messages from the backend, but Cypress often got stuck, failing to pick up new content on the screen. I don't think it integrates very well with the Vue ecosystem. I spent even more time trying to get Cypress to work on the pipeline, not realizing that I didn't actually have to do anything beyond installing the artifact, as it would execute the tests automatically.

The Vuex store was another pain point. I spent a lot of time on this with very few results until the last week or so. I had no experience with testing global state for applications and was not aware that, with Vuex, you cannot simply call something like mockStore = vue.mockStore. You have to create an instance of the store in your test, populate it with relevant state, actions, and mutations, and only then can you use it in a test. I feel this is rather complex and laborious for testing something so crucial to applications.

9 Appendix

GithubProject

References

[1] Mayank Gokarna and Raju Singh. Devops a historical review and future works, 12 2020.

[2] Atlassian. Devops, 2025. Accessed: 2025-03-19.

[3] Amazon Web Services. What is devops?, 2025. Accessed: 2025-03-19.

[4] GitLab. Devops topics, 2025. Accessed: 2025-03-19.

[5] F. Erich, C. Amrit, and M. Daneva. A qualitative study of devops usage in practice. *Journal of Software: Evolution and Process*, 2017. Accessed: 2025-04-28.

[6] L.E. Lwakatare, P. Kuvaja, and M. Oivo. Relationship of devops to agile, lean and continuous deployment. *PROFES 2016: Product-Focused Software Process Improvement*, 2016. Accessed: 2025-04-28.

[7] F. Erich, C. Amrit, and M. Daneva. Challenges of devops adoption - a systematic literature study. *Journal of Systems and Software*, 199, 2023. Accessed: 2025-04-28.

[8] Peter Pšenák and Matus Tibensky. The usage of vue js framework for web application creation. *Mesterséges intelligencia*, 2:61–72, 01 2020.

[9] DDI Dev. Pros and cons of reactjs web app development, 2025. Accessed: 2025-03-19.

[10] AltexSoft. The good and the bad of angular development, 2025.

[11] bcrypt contributors. bcrypt - npm. `https://www.npmjs.com/package/bcrypt`, 2024. Accessed: 2025-04-08.

[12] Kinsta. Google cloud vs aws: Which is better for your business?, 2023. Accessed: 2025-03-19.

[13] Softteco. Aws vs azure: Which is better?, 2023. Accessed: 2025-03-19.

[14] Javokhirbek Khaydarov. Dispatch and mutation in vuex. `https://javokhirbekkhaydarov.medium.com/dispatch-and-mutation-in-vuex-2714c6920ed3`, 2023. Accessed: 2025-04-28.

[15] Logaretm. Handling forms - veevalidate. `https://vee-validate.logaretm.com/v4/guide/components/handling-forms/`, 2023. Accessed: 2025-04-28.

[16] Robert C. Martin. The clean architecture. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html, 2012. Accessed: 2025-04-28.

[17] Google Cloud. What is microservices architecture? https://cloud.google.com/learn/what-is-microservices-architecture, 2024. Accessed: 2025-04-28.

[18] Stackify. Solid principles: The five principles of object-oriented programming. https://stackify.com/solid-design-principles/, 2025. Accessed: 2025-04-28.

[19] Amazon Web Services. What is amazon vpc? https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html, 2025. Accessed: 2025-04-28.

[20] HashiCorp. Terraform: Infrastructure as code. https://developer.hashicorp.com/terraform/intro, 2025. Accessed: 2025-04-28.

[21] ReleaseWorks Academy. How to create an eks cluster with node groups using eksctl. https://tutorials.releaseworksacademy.com/learn/how-to-create-an-eks-cluster-with-node-groups-using-eksctl.html, 2025. Accessed: 2025-04-28.

[22] Amazon Web Services. Amazon cloudwatch. https://aws.amazon.com/cloudwatch/, 2025. Accessed: 2025-04-28.

[23] Amazon Web Services. Aws secrets manager. https://aws.amazon.com/secrets-manager/, 2025. Accessed: 2025-04-28.

[24] Amazon Web Services. Aws lambda. https://aws.amazon.com/lambda/, 2025. Accessed: 2025-04-28.