An A/B Testing Platform for Fastly's Compute@Edge Platform

A Major Qualifying Project Report submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

Authors:

Samuel Gehly, Computer Science

Lauren Hatfield, Robotics Engineering and Computer Science

Caitlin Jung, Computer Science

Advisor:

Professor Mark Claypool, Computer Science and IMGD Professor

Sponsors:

Patrick Hamann, Principal Software Engineer at Fastly

Salman Saghafi, Principal Software Engineer at Fastly

March 18, 2021

Abstract

A/B testing has become a common practice that companies use to continually improve and optimize existing products and services due to its ability to evaluate design ideas quickly, precisely, and cheaply. There are multiple A/B testing frameworks and platforms available for companies to use, yet of all the existing ones, currently none use Rust, which is the primary supported language with Fastly's Compute@Edge (C@E) platform, the serverless execution environment for this project. We designed and developed an end-to-end A/B testing platform compatible with Fastly's C@E serverless service offering. The three components of this platform are a Rust Crate (Framework), an API written in Go, and a user interface (UI) written in JavaScript using React and Next.js. We thoroughly unit tested these components and system tested the overall platform with an example A/B test, which demonstrated that the platform is a successful minimum viable product. By taking a novel serverless approach to A/B testing and utilizing the maturity and benefits of the three languages, Fastly can contribute to the business experimentation and serverless communities.

Acknowledgments

First, our team would like to thank our sponsor, Fastly, for giving us this amazing opportunity to contribute, learn and work with Fastly’s serverless execution environment, Compute@Edge. Specifically, we want to thank our awesome and incredible Fastly mentors, Patrick Hamann and Salman Saghafi, for their guidance and support throughout this project. We also want to thank Peter Bourgon, Patrick McManus, Jim Rainville, katelyn martin, and all the other fantastic Fastly employees that helped us along the way. Lastly, we would like to thank our MQP advisor, Professor Mark Claypool, for organizing this project and for all his guidance throughout this project.

Table of Contents

List of Figures

1. Introduction

A/B testing has become a common practice that companies use to continually improve and optimize existing products and services. From choosing color schemes to surveying potential marketing campaigns, companies utilize A/B testing to meet different goals. A/B testing is a process of showing two variants of some element (a button, a web page, a title) to different website visitors at the same time and comparing which variant drives more conversions, a metric that a business might want to track (user signups, for example) [1]. A collection of related variants is an experiment. With this method, companies can collect large amounts of conversion data on websites as well as run experiments which evaluate ideas and design choices quickly, precisely, and cheaply.

Due to the numerous benefits of experimentation, there are multiple A/B testing frameworks and platforms available for companies to use. For the purposes of this document, platforms are usually end-to-end solutions that include a user-interface, while frameworks are usually code-exclusive. There exists a wide array of open-source testing frameworks, such as PlanOut [2] and Optimizely [3], that support more complex experimental designs while considering and satisfying the constraints of deployed Internet services, such as scalability and performance. These frameworks are written in a broad range of programming languages and execution environments. There are implementations of this in JavaScript, Objective-C, Java, C#, PHP, Python, Go, and other popular languages.

Of all the existing frameworks that conduct A/B testing, currently none use Rust. Rust is a systems programming language designed for performance, reliability, and productivity. Rust is compatible with WebAssembly (WASM), a binary code format that provides native system performance with little overhead, while also providing support for web APIs [4]. Furthermore, Rust is the primary supported language with Fastly’s Compute@Edge (C@E) platform, the serverless execution environment for this project [5]. Serverless is a cloud computing model where the cloud provider allocates backend resources as needed, that is, the server scales up and down based on how many requests the endpoint receives and is priced by usage [6].

Recently, there has been a shift towards edge computing, a paradigm which moves code closer to users for improved response times [7]. In addition, rapid prototyping has become more important to startups who may want to write and deploy code quickly. In light of both of these, companies are looking to create and update A/B testing frameworks and platforms, namely by utilizing the cloud. This provides rapid deployment capabilities and the ability for developers to quickly iterate on experiments [8]. Another way companies achieve rapid prototyping is through the use of hosted third-party solutions that do not require server integration.

In this project, we developed an open-source, end-to-end A/B testing platform, called GOAT-AB, which runs on C@E. Companies can use this platform to experiment with which variants a user interacts with. The platform has three primary components: the Rust Crate/Framework, Representational State Transfer (REST) API, and user interface (UI). When combined, these components provide a solution for A/B testing at the edge. Moreover, the platform is flexible to give the developer control over how they configure and utilize the platform.

There are three varieties of people that are likely to interact with this platform. In the context of this paper, developers are the people who utilize the Rust Crate. They are people who have experience with programming in Rust. Experiment designers are people who use the UI. They may not have much, if any, programming experience. The end-users are the people who visit a website and get bucketed into a variant.

Overall, this platform can serve as the foundation for future experimentation at Fastly's C@E platform and can give businesses the opportunity to improve conversions. For example, a news company could determine which headline of an article gets more clicks, which could then lead that company to improve its headlines to get more viewership for future articles. By combining the maturity and benefits of Rust and deploying it to a new environment to take a novel approach to A/B testing, Fastly can contribute to the serverless and business experimentation communities.

The rest of the paper is structured as follows. Chapter 2 discusses background information and related work about cloud computing, A/B testing and the software development tools we used. Chapter 3 explains the detailed steps and system architecture of the project. Chapter 4 describes the design of the project and the evaluation procedures and methods to fully test the platform. This chapter also showcases our platform in depth along with a demonstration. Chapter 5 summarizes our project conclusions, and Chapter 6 identifies potential future steps to improve or expand upon our work.

2. Background

Our project relies on many modern technologies to provide businesses with an A/B testing platform. This section first details these technologies, beginning with cloud computing services, the kind of service on which GOAT-AB runs. The section then describes cloud services at Fastly, the company at which our platform runs, and continues with an explanation of A/B testing, the functionality of the platform. The third part of the section outlines the languages utilized in this project to help with understanding the project software and technologies. Finally, the last section discusses the necessary software development tools and methodologies for the project's workflow.

2.1. Cloud Computing Services

Cloud computing services are computing services provided through the Internet [9]. Companies typically use these services to increase computing power without the hassle of maintaining their own hardware and networking. This simplifies companies' operations as it moves more of their infrastructure to what is called the "cloud," where large companies like Amazon and Google manage these servers and resources [10]. Companies typically provide these services using pay-as-you-go pricing, however the metrics used to calculate usage depend on the service. This cost-effective approach incentivizes many businesses to utilize their services. Three common variations of cloud services are Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Functions as a Service (FaaS). An overview of these services can be seen in Figure 1.

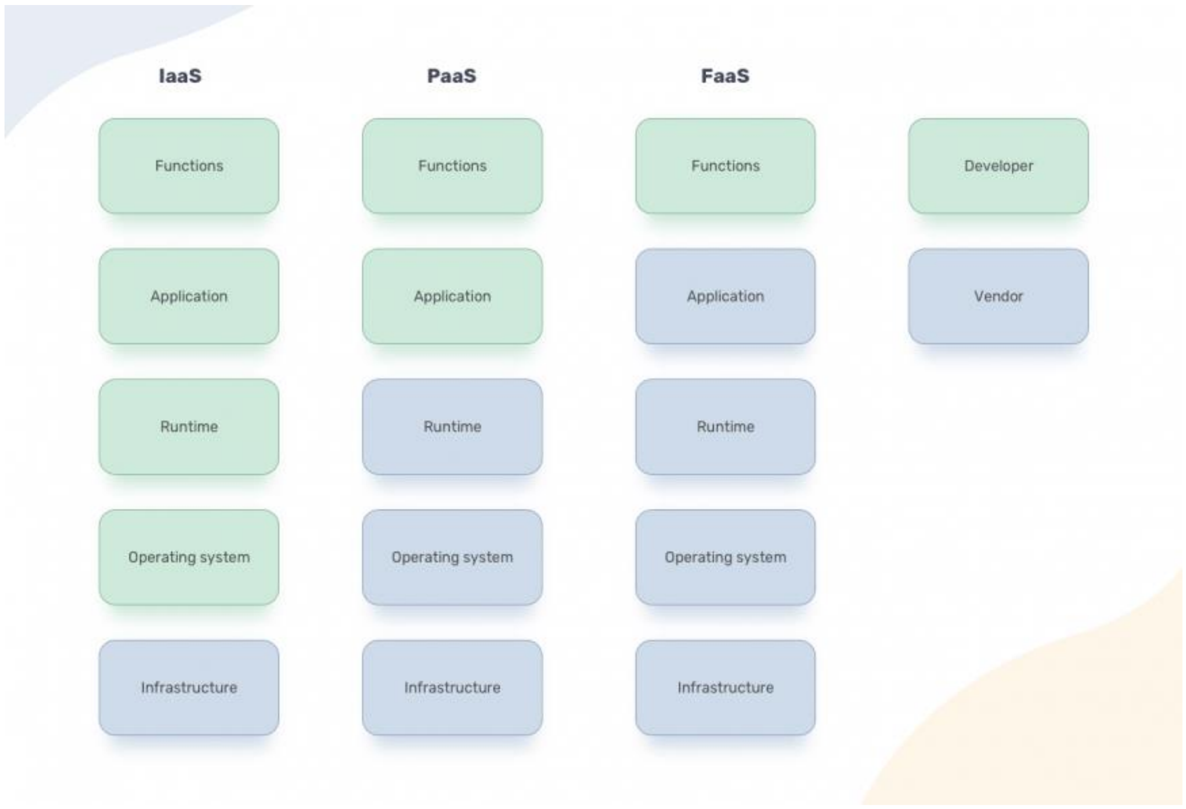


Figure 1. Types of Cloud Computing Services and What They Encapsulate

The lowest level of these is IaaS. This service provides hardware virtualization for the consumer. IaaS provides storage, networking, and servers [11]. Examples of IaaS include the Amazon Web Services (AWS) EC2, Microsoft Azure Compute, and Google Cloud Compute. In this structure, the business does not need to physically purchase or maintain hardware [12]. Businesses often choose this type of service because it allows them to specify many requirements of the system in detail. Of all these types of services, IaaS is the most flexible and easy to automate, but it is the most complex in terms of setup, configuration, and maintenance.

PaaS adds more abstraction to the concepts of IaaS, handling the server environment in which code runs [13]. In addition to storage, networking, and servers, PaaS includes middleware such as the OS and development tools [14]. The AWS Elastic Beanstalk, Google App Engine, and Oracle Cloud Platform all provide PaaS [15]. A business might choose this type of service because it is easily scalable and allows employees to produce and run applications without the overhead of managing the hardware, while still having control over the configuration of the execution environment and programming language. The business does not need to manage the infrastructure or the hardware, and therefore loses some of the flexibility found in IaaS. It is the "middle of the road" option when it comes to the sliding scale of complexity versus flexibility.

Similarly to PaaS abstracting IaaS, FaaS abstracts PaaS. In this case, FaaS contains the same abstractions as PaaS, but additionally handles the execution environment (the application / routing layer and the operating system) [16]. This service responds to the occurrence of specific events, such as cron triggers (time-based execution) or in response to HTTP requests (API requests, for example) [17]. AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, Cloudflare Workers, and Netlify Functions all provide FaaS. Businesses typically choose this service for its ease-of-use and scalability. As the business does not need to manage even the data layer, the business loses even more flexibility in exchange for simplicity. FaaS is a tool for businesses who want an API with low overhead.

2.2. Cloud Services at Fastly

Fastly is a cloud computing service provider [18]. A large part of Fastly's offerings include its expansive content delivery network (CDN), which provides quick load times for websites and keeps its users' content up to date [19]. Large companies, such as GitHub, KickStarter, TheGuardian, and Stripe, use Fastly's CDN services. Moreover, Fastly provides a wealth of products for image optimization, video streaming, cloud security, and networking. This wealth of features allows its customers to create fast, secure websites and applications with high-quality content, while continuing to innovate in new areas.

As of October 16, 2020, Fastly predicted an increase in revenue by 41% over the course of a year [20]. As Fastly continues to grow, it continues to provide additional capabilities for its edge computing offerings. One of these capabilities is becoming a FaaS provider through its serverless computing environment, C@E. Fastly is entering the serverless industry to utilize its preexisting network to allow developers to run code at the edge. One use case for this is enabling developers to run A/B experiments at the edge.

2.3. A/B Testing

A/B testing is the process of comparing two versions of a web page or application [21]. Similarly, multivariate testing is the process of comparing two or more versions of a web page or application. This paper uses A/B testing to refer to both. For a particular experiment, end-users will see one of two versions of a web page or application. Each end-user will either see a version determined by a wide range of factors including (but not limited to) location, type of device, and user ID in a process known as “bucketing.” Companies that implement this kind of testing also vary widely and include The New York Times and AirAsia [22]. Using experimentation enables the testing of a wide range of elements, such as comparing different headlines in a newspaper, shapes of buttons, or even pricing.

For example, Electronic Arts (EA) used Optimizely’s A/B testing platform on its pre-order launch of SimCity in 2013 to improve overall sales on the game. The first version of the webpage included a promotional banner, which moved the “Buy Now” button to the bottom of the page. The second version removed the banner, which moved the “Buy Now” button closer to the top of the page. EA tracked the number of clicks on the “Buy Now” button for each variation and the results eventually led them to discover that the second version drove 43.4% more purchases [23]. Another example is WallMonkeys, which used CrazyEgg’s A/B testing platform to optimize its homepage for clicks and conversions by testing the image featured on the homepage. By comparing the current featured image to a more “fun” one, WallMonkeys discovered that this produced a 27% increase in conversion rates on the site [24]. Both examples used existing A/B testing platforms to conduct their experiments and helped the companies improve the conversion rates on their websites.

Currently there are several A/B testing platforms and companies that provide hosted services. These platforms make it easy for businesses to conduct A/B experiments and the services allow businesses to create experiments without hosting their own infrastructure to power it. Frameworks include Conductrics, Planout, AB.js, Flagr, Togglz, Waffle, Gargoyle, CrazyEgg, and many more. Hosted services include Firebase (Google), and Optimizely, VWO, Zoho PageSense, HubSpot, Leadformly, Unbounce, Convertize, Kameleoon, and many more (and their associated frameworks for implementation). One framework is Conductrics, which provides both client and server-side A/B testing as well as predictive targeting using AI and machine learning to aid with analytics [25]. Another framework is PlanOut, which is an open-source testing framework geared towards researchers, businesses, and students that supports more complex experimental designs. It features extensible classes, namespaces for mutually exclusive experiments, and the PlanOut language [2].

Similar to PlanOut, Optimizely creates an easy-to-use environment for conducting experiments while also being a hosted solution that businesses do not need to host themselves. While the previously mentioned frameworks are capable of integration into the cloud, Optimizely has a “full-stack” experimentation platform capability that provides a solution for server-side experimentation [26]. Firebase, another example of a hosted service, focuses on simplifying the process of running, analyzing, and scaling experiments [27]. Furthermore, Firebase focuses on engaging users with notifications and targeting specific groups for experiments alongside other integrations within the Firebase ecosystem.

While A/B testing is a useful tool for introducing variation into a website or application, experiment designers must consider the ethics and security of A/B testing platforms [28]. For example, Optimizely exposes the entire experiment configuration to the end-user's web browser, which allows end-users to see the factors that decide which variant the experiment buckets them in. According to the Optimizely developer documentation, this is still the case [29]. One study at Northeastern University took advantage of this design to perform in-depth analytics on A/B testing on a variety of web pages [22]. This study determined which factors the experiment considered when choosing which web page to display and looked at the different variants. This may raise some concerns for websites that use Optimizely's services, but may also favor the end-user, in terms of transparency, to ensure that the experiments are ethical.

Furthermore, many businesses, including The New York Times, use A/B testing without notifying users [22]. While this may be harmless in most cases, this can facilitate the unethical, but legal, gray area of price discrimination, where experiments use end-user data to modify the price shown for a particular product. While some differentiations of end-users may be useful to determine who clicks on what, it is important that this differentiation does not become discrimination and does not extend to the prejudiced or unjust treatment of the end-users.

2.4. Languages

This subsection discusses the high-level concepts of all the programming languages that aided in the development of GOAT-AB. Different languages have advantages and disadvantages for different use cases, and they are important to recognize to understand why we used them in this project.

2.4.1. Rust

Rust is a systems programming language that focuses on memory safety, thread safety, and performance [30]. Because of Rust's emphasis on safety, the compiler enforces many rules which do not exist in other languages. For example, Rust adheres to specific ownership rules for passing data between functions. If access to the original value of the function parameter is necessary after the function terminates, then the function parameter must be a reference, indicating that the function can read the data, but not modify it. On the other hand, if access to the original value is not necessary, then the function will own that data and can directly modify it. Typically, referenced data is the input to functions and the function returns owned data. These rules, combined with Rust's other complexities, give Rust a relatively steep learning curve.

2.4.1. Go

Go is a type safe programming language developed by Google that programmers commonly use for building REST APIs, which provide the connection between the user interfaces and other infrastructure, such as SQL databases [31]. Compared to Rust, the learning curve for Go is much lower and enables REST APIs to be built quickly and safely as the language has many built-in functions for creating these kinds of applications.

2.4.1. JavaScript (JS) – React and Next.js

React is a JS framework developed by Facebook that allows programmers to easily build UIs [32]. Several companies, such as Netflix, Cloudflare, Twitter, Reddit, and GitHub, use this framework. Next.js is a further abstraction of React to make it easier to build multi-page applications without the need to implement a router to determine what pages execute what code [33]. In a Next.js application, all the pages are loaded at the beginning of the end-user's session instead of upon request. As a result, when navigating the application, Next.js will immediately swap the page content with the new page, instead of the browser needing to wait for this new content. The relationship between these three items are shown in Figure 2. Next.js and React are modules written in JavaScript (Version ES6), Next.js depends on React, and applications can depend on both Next.js and React.

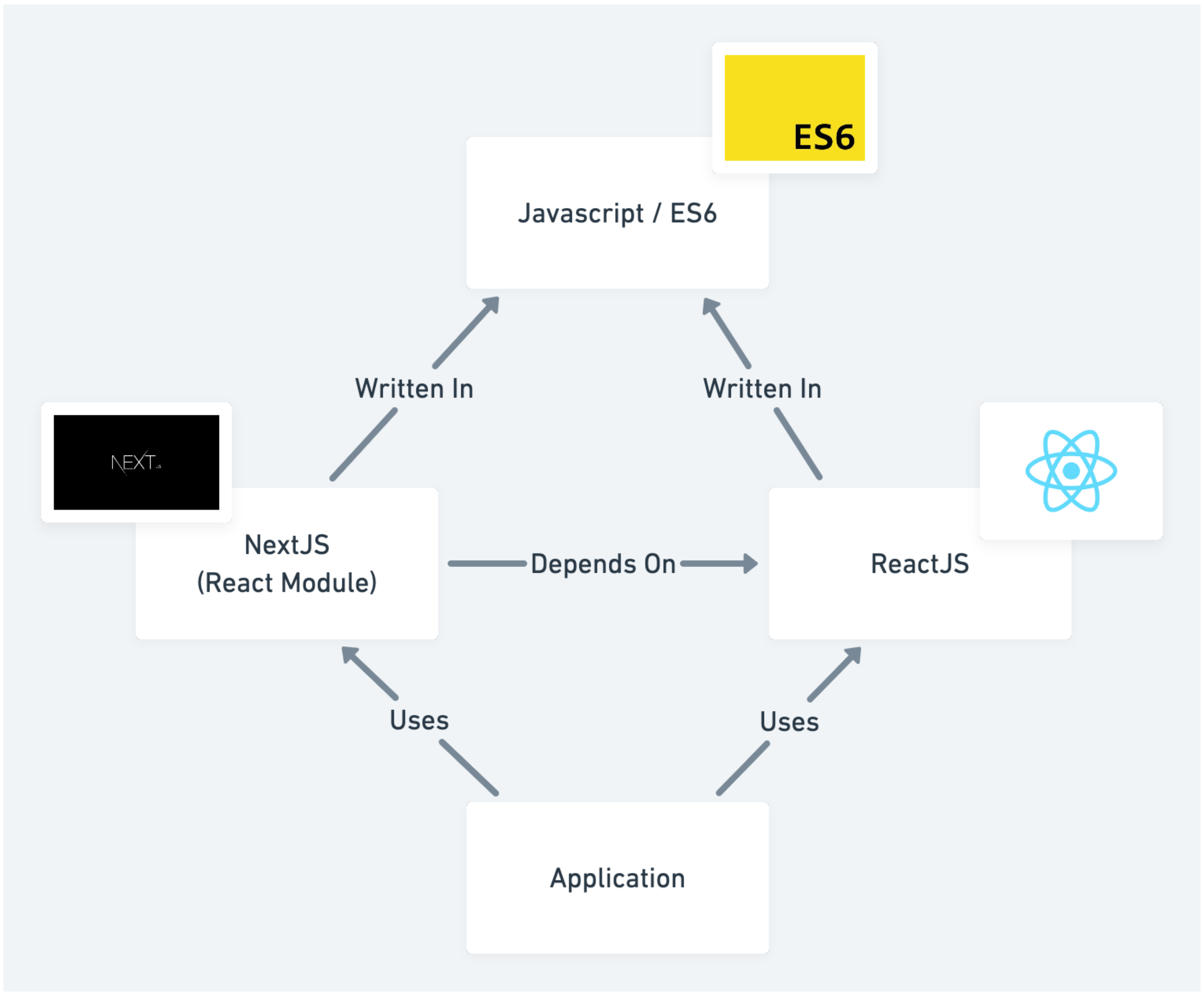


Figure 2. The Relationship between React, Next.js, and JavaScript/ES6

2.5. Agile Methodology and Scrum Framework

To develop our A/B testing platform, we utilized the Agile Methodology and Scrum Framework. The Agile Methodology is an iterative approach used in software development project teams that focuses on continuous product releases and incorporating customer feedback with every iteration [34]. Scrum is a framework for Agile that helps teams work together by encouraging communication and collaboration amongst team members as well as project management [35]. Project teams break down their projects into fixed-length iterations, called sprints, that are typically one to four weeks in length [36]. During each sprint, these teams participate in multiple meetings, as shown in Figure 3, and deliver a releasable, well-tested product.

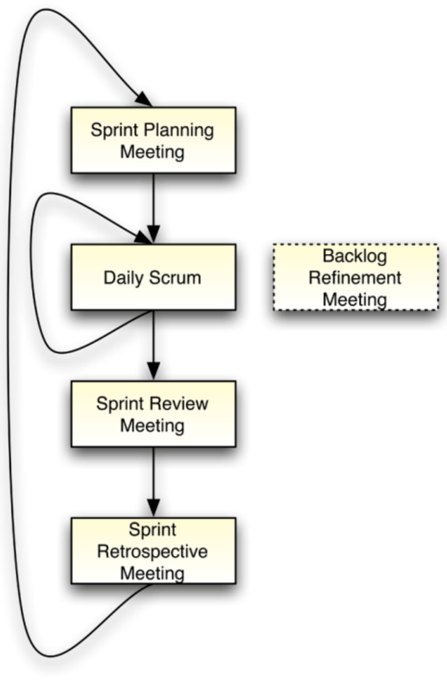


Figure 3. Scrum Flow for each Sprint [37]

As shown in Figure 4, each project team consists of a product backlog to manage all the tasks for the project. The backlog consists of user stories, which represent features that are important to the product's users and appear in the final product. At the beginning of each sprint, the team conducts a sprint planning meeting, where they choose a reasonable amount of achievable and pertinent user stories to complete by the end of the sprint. The team moves the selected user stories to the sprint backlog.

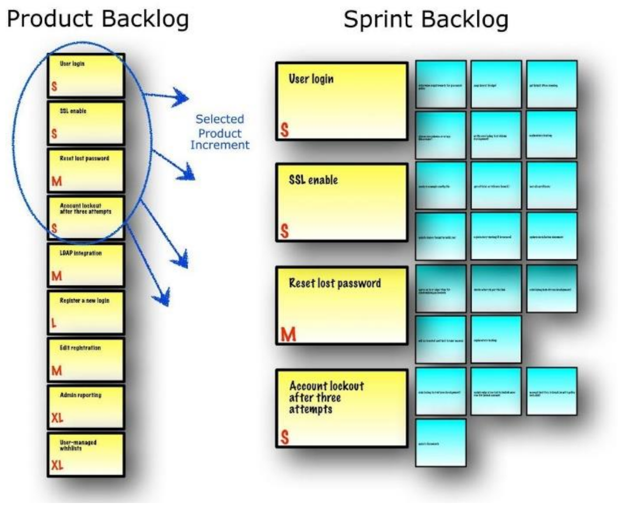


Figure 4. Agile Scrum Product and Sprint Backlog [37]

Teams participate in daily stand-up meetings, or daily scrums, where each member must speak about what tasks they completed the previous day, what tasks they will complete today, and any issues that currently block them from performing their tasks. At the end of each sprint, the team demonstrates their product to the stakeholders in the sprint review to receive crucial feedback that the team will implement in the next sprint. Each sprint ends with a sprint retrospective meeting, in which the team reflects on the current sprint by reflecting on what went well, what problems the team encountered, and how the team resolved (or did not resolve) them. Retrospectives aim to help teams improve and adapt for future sprints to maximize efficiency and minimize any potential blockers.

2.5.1. GitHub and Product Backlog

GitHub is a code hosting platform that many software programmers and technology companies utilize for its numerous benefits for projects, which include version control, code storage, and code management [38]. In addition to these benefits, GitHub has a feature called GitHub Projects that allows software programmers to integrate project management into their development cycle [39]. When creating a project board, there are several templates that programmers can select from, including Kanban boards, to best align with the needs of the project team. All changes to the boards, such as creating cards and organizing them into custom columns, occurs in real-time. Furthermore, programmers can convert the cards to issues and associate them with a pull request (PR) or assign other programmers to specific cards. Ultimately, GitHub Projects has a UI that allows teams to visualize progress and prioritize tasks for each sprint in their GitHub repository.

3. Methodology

This chapter details the design process as well as the software development tools and methodologies we utilized in order to develop an end-to-end A/B testing platform for Fastly that will run on Fastly's C@E platform.

3.1. Feature Priority List

To identify the most important and useful features for our platform, we gathered information from discussions with our sponsors and from online research on what experiment designers desire in A/B testing platforms. For this project and throughout the rest of the report, experiment designers are people who directly interact with the platform to generate experiments. From our information gathering, we developed a prioritized list of potential features that mostly dealt with integrating our platform into Fastly's ecosystem. The features are as follows, in order of importance:

1. Providing A/B testing functionality through a default selection function
2. Allowing custom selection functions for bucketing

   Giving flexibility to the experiment designers to add their own selection functions tailors the framework to the experiment they want to perform.
3. Incorporating a persistent database to store experiments and variants

   The experiment designers then create, view, modify, and delete experiments and variants at any time.
4. Enabling experiment configuration through a UI

Configuring experiments within a UI would make the A/B testing platform accessible to experiment designers and allows for cleaner and more standardized C@E codebases.

5. Developers logging their end-users' activity on C@E

    Logging end-user activity along with their variants allows companies to do their own analysis. This platform is intended to be used by customers of Fastly—Fastly does not use the platform to log/analyze user activities for internal use. For this project and throughout the rest of the report, end-users are participants in the experiments.

We also compiled a list of stretch goals that would later become contenders for future work:

1. Adding explicit bucketing characteristics to integrate with C@E

    Additional bucking characteristics include (but are not limited to) the end-user's location, browser, device, and visit count.

2. Providing configuration, results, and an analytics UI in the Fastly Dashboard

    We would provide an end-to-end solution for Fastly by providing all necessary elements of A/B testing and analytics within the Fastly serverless environment. An experiment designer could configure experiments and see the results of said experiments visually within the same dashboard, instead of relying on third-party analytics services or databases.

3. Supporting predictive designing for experiments, using statistical learning

    A statistical learning model could then analyze previous experiment data to make predictions about which changes in an experiment may cause specific changes in results.

4. Real-time suggestions for improving current experiments

    Finally, we could develop an algorithm to suggest changes mid-experiment to achieve specific goals.

These lists of features provided the foundation for our project and guided us in the development process.

3.2. System Architecture

After identifying the important components of our platform, we made an architecture diagram of how our platform would function in Fastly's ecosystem (Figure 5). The key for the diagram is as follows:

- A green box is an HTTP request.
- A pink box is a C@E action.
- A red, outlined box is an experiment designer action.
- A red, filled in box is an end-user action.
- A blue box is a Rust framework action on C@E.
- An orange box is a Fastly API/Edge Dictionary action.

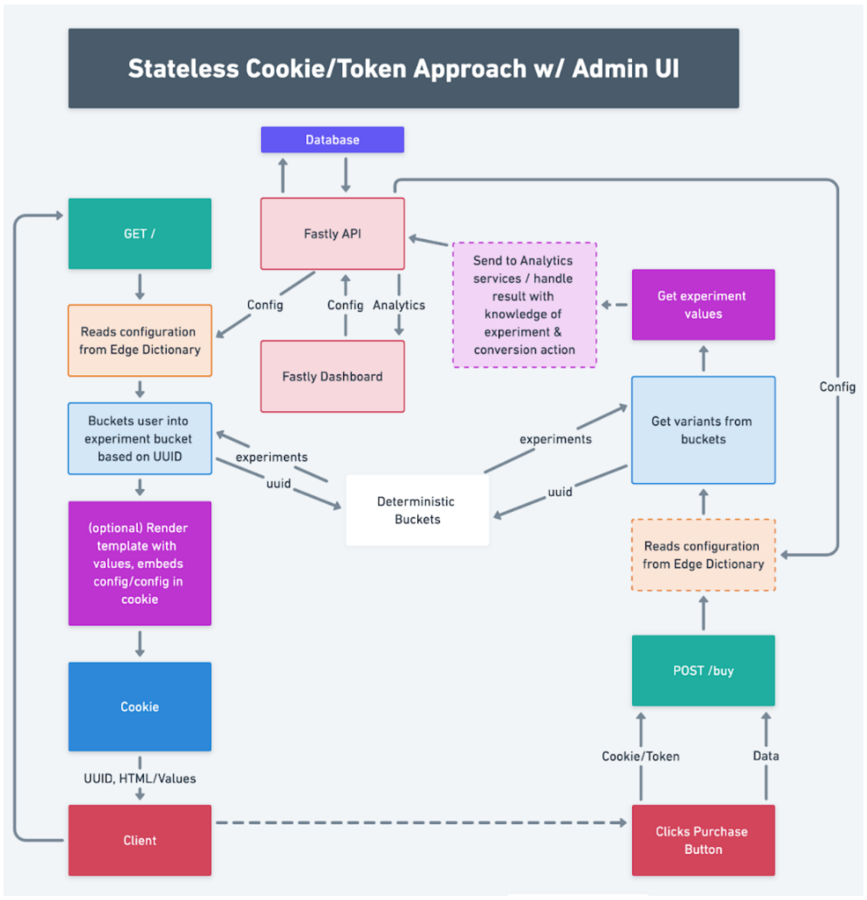

Stateless Cookie/Token Approach w/ Admin UI
Database
Fastly API
GET /
Reads configuration from Edge Dictionary
Config
Config
Analytics
Fastly Dashboard
Send to Analytics services / handle result with knowledge of experiment & conversion action
Get experiment values
Buckets user into experiment bucket based on UUID
experiments
uuid
Deterministic Buckets
experiments
uuid
Get variants from buckets
Config
(optional) Render template with values, embeds config/config in cookie
Reads configuration from Edge Dictionary
Cookie
POST /buy
UUID, HTML/Values
Cookie/Token
Data
Client
Clicks Purchase Button

Figure 5. The Relationships between C@E, Framework, Fastly, and User

To use the platform, first the experiment designers interact with the UI of our platform, to create, update or delete experiments. Then, the experiment designers publish all the experiments to an API that stores persistent data. When an experiment designer is ready, they may publish the data to Fastly's Edge Dictionaries. An Edge Dictionary is an offering from Fastly to save data at the edge, similar to how code runs at the edge with C@E [40].

Once the experiment designers publish the experiments, end-users can visit the specific website where the experiment runs. When an end-user visits the website, our framework reads the associated experiment configuration from the Edge Dictionary along with information associated with the user. Then, the framework buckets the end-user into one of the variants associated with the experiment. This bucketing is deterministic, which allows the framework to always bucket the same end-user into the same variant each time the end-user interacts with the experiment.

Once the framework buckets the end-user into a variant, the end-user can see that variant on the screen. For example, one end-user may see a red button and another may see a blue button. Oftentimes with A/B testing, end-users perform an action, sometimes called a "conversion event", where they interact with the framework in some way to provide analytics for the experiment. For our framework, if the end-user interacts with the variant, for example clicking on one of the buttons, then the framework logs the action to C@E. The C@E function reads the associated experiment from the Edge Dictionary and sends that information along with the selected variant to wherever the developer wants. From there, the experiment designer can get the results and send them to an external analytics service to gain feedback from the experiment.

3.3. Software Development

To develop our platform according to the system architecture described above, we followed the Agile Methodology and Scrum Framework and utilized several software tools. Agile and Scrum were helpful for creating an application and continually building upon previous designs to ensure that the platform is suitable for the users. Combining GitHub Projects for project management, GitHub for our code repository and Visual Studio Code (VS Code) for our Integrated Development Environment (IDE) facilitated programming and fostered collaboration and the development of well-written code.

3.3.1. Agile Methodology and Scrum Framework

Our team utilized the Agile Methodology to ensure the creation of a platform that meets the needs of our sponsor and the potential users of our platform. We had seven one-week sprints to develop our product. At the beginning of each sprint, we chose a reasonable amount of achievable and pertinent tasks to complete by the end of the sprint. We maintained all of our tasks in our product backlog on GitHub Projects. For the majority of the tasks, we all worked together to complete them.

We had daily stand-up meetings with at least one of our sponsors to discuss our progress and resolve any impediments, which often entailed walking them through the new parts of our code. We also communicated frequently with them via Slack for any issues or questions that arose during the day.

At the end of each sprint, we demonstrated our platform to our sponsors and requested a review of the changes to our code to receive crucial feedback. Each sprint ended with a sprint retrospective meeting where we reflected on the current sprint by discussing what went well, what problems we encountered, and how those problems were resolved. At the beginning of the project, the main takeaways from our retrospectives were our desires to do more pair programming and to block off time during the week to work together. As the project progressed, our ability to work as a team improved and these retrospectives shifted towards pointing out problems with our code or other issues that halted our progress. In fact, during Sprint 3, a lot of the work planned got pushed to Sprint 4, as there were a lot of code comments to address and we planned more tasks than we were able to accomplish.

3.3.2. Software Development Tools

Software development tools, such as GitHub and VS Code, played an integral role in the development of our platform.

3.3.2.1. GitHub

Our sponsors set up a private GitHub repository for us to store the code for this project. During Sprint 1, we created branches for each task that we completed, but for the rest of the sprints, per the request of our sponsors, we created one branch for each sprint. Since we worked together to complete the majority of the tasks, we did not have to deal with many merge conflicts. However, whenever we split up the work and any merge conflicts appeared, we reviewed the code differences, made appropriate changes, and noted all affected files in the merge commit comment. After completing the sprint tasks, we created a review request (or “pull request”) for our sponsors, which due to GitHub’s collaborative nature, allowed them to comment on specific lines that we could then discuss with them on the platform if they were unavailable to meet. Once we addressed all comments on the pull request, we merged it to the main branch and deleted our sprint branch.

3.3.2.2. VS Code

For all the programming done in this project, we used VS Code version 1.54 for our IDE [41]. With VS Code, we downloaded all the necessary language extensions (Rust, Go and React/Next.js) and associated linting tools that allowed us to write well-structured code for our project. Also, we handled merge conflicts on VS Code by choosing to accept the incoming or current changes and modifying where appropriate. Furthermore, we often used VS Code’s Live Share feature to do pair programming and collaborate [42].

4. Implementation

To implement the system architecture described in Chapter 3, we created a platform that is both generic and performant. This platform took the form of a minimum viable product (MVP) within the scope and time limitations of the project. While it may not be a complete product, this platform provides a starting point for future work, while still providing A/B testing functionality.

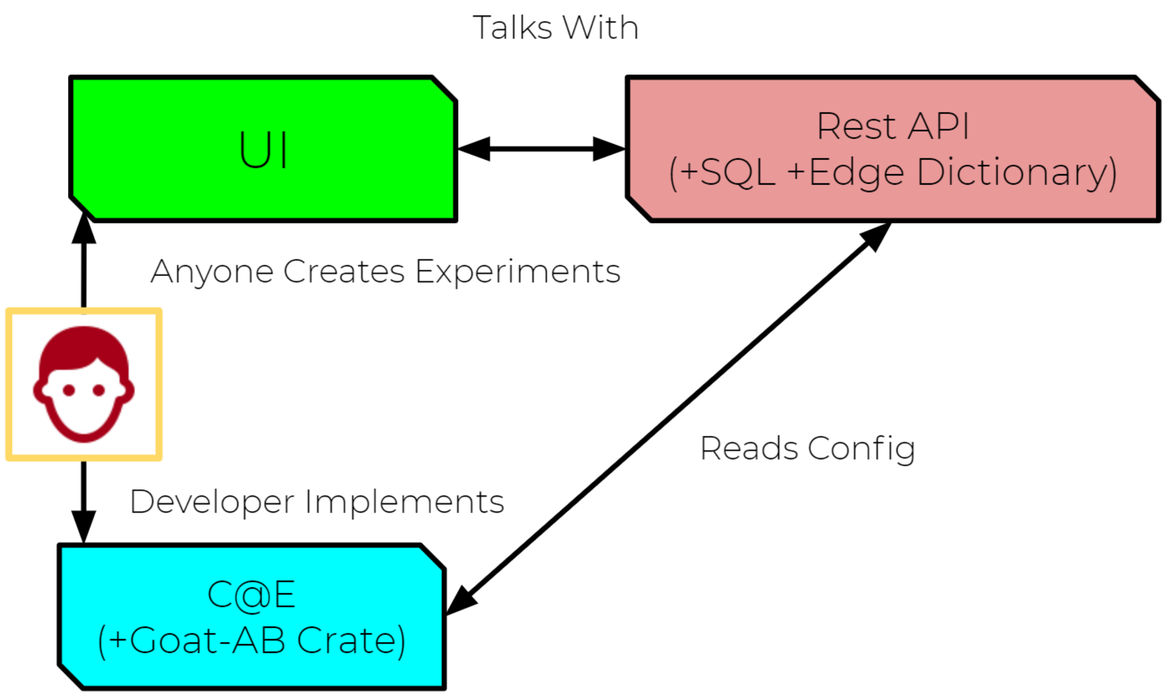


Figure 6. The Three Components of the Platform

The platform consists of three components, shown in Figure 6 above:

- The Rust Crate (Goat-AB Crate shown in blue)
- The REST API (shown in red)
- The Configuration UI (UI shown in green)

First, an experiment designer would use the UI to configure which experiments and variants they want to deploy. The UI communicates with the REST API to store that specific configuration in a database of their choosing, whether it be a PostgreSQL database or Fastly's

Edge Dictionaries. When the developer wants to test a specific experiment, they use the Rust Crate, which uses their selected data source as an input to sort an end-user into one variant per experiment. This process allows the developers and experiment designers to work together to implement A/B testing.

4.1. Rust Crate/Framework

The Rust crate provides functionality associated with A/B testing, specifically, bucketing users into the appropriate variant. This A/B testing crate can run anywhere that supports WASM, however for the scope of our project we utilized C@E. Since Rust is the primary supported language with C@E, developing the code for the A/B testing process in Rust was essential for this integration.

We developed the Rust Crate as the first step of the project because it provides the foundation for the entire platform. First, we determined which pieces of information were necessary for the different API calls to the crate. We decided that each time the crate buckets a user, we would pass in end-user-specific information, such as the end-user's IP address, and when initializing the crate, we would pass in any configuration files defining the experiments or variants. Next, we modeled the experiments and variants. Each experiment needs to hold information on its ID, name, published status, and variants. Each variant needs to hold information on its ID, name, value (which can be any valid JSON), and sample rate. When creating these objects, the crate validates them to ensure that the fields of each contain the appropriate data types. The sample rates for all of the variants in an experiment must sum exactly to 1. We also considered other validation constraints such as length, character set, and uniqueness.

The next step was developing the selection function, which buckets users into variants. We developed a default selection function, called the random selection function, that uses a hasher to convert hash input (usually something that identifies an end-user, e.g. IP address, web browser) into a random integer. Then the function converts this integer to a decimal, which corresponds to one of the variants. While we only created the random selection function, we developed the selection function to allow for multiple future selection function integrations.

One main goal during the development of the crate was to make it abstract and flexible so developers could customize A/B testing to their needs. The selection function uses Rust traits and generics to allow developers the ability to provide custom selection functions not currently programmed in the crate. This initially was difficult as there were issues with using a Hash object as a generic data type with our selection function implementation. We explored defining the selection function as an integer, enum and other data types, but ultimately, we added a "UIExperiment" struct which included an enum of the crate's selection functions and an instance of the "Experiment" model. We then removed the selection function altogether from the "Experiment" model. This solved our issues while retaining the abstraction we desired.

We were then ready to put together the public API. It consists of two functions: initialize_from_string and bucket_all. The first initializes the crate. It takes a string representing a valid JSON object and converts it into either a vector of experiments or returns a configuration error. There are various types of configuration errors that can result: parsing, validation, or runtime. A parsing error occurs when the Rust Crate cannot parse the input JSON string into a valid experiment. A validation error occurs when the JSON object does not conform to valid standards, such as string length. A runtime error occurs when there is an issue with bucketing. When initialize_from_string returns a vector of experiments, the bucket_all function takes that vector as an input along with the end-user information as a string. This function calls the selection function for each experiment in the vector. The bucket_all function returns a vector of the selected variants for each experiment for that specific user's information.

This all integrates into Fastly with C@E and Fastly's Edge Dictionaries. The design of our crate inherently supports C@E. Edge Dictionaries provide the experiment and variant definitions for the crate's initialization function.

4.2. REST API

The REST API mainly deals with storing persistent data (experiments and variants) and writing this data to Fastly's Edge Dictionaries. Currently, Edge Dictionaries do not enable data modification from C@E, so this REST API was necessary to handle that [43]. We wrote this API in Go, which allowed us to be quickly productive and efficient because of Go's strong support and built-in features for creating HTTP servers [44]. As seen in Figure 7, the REST API contains three layers: the API layer, the service layer, and the database layer.

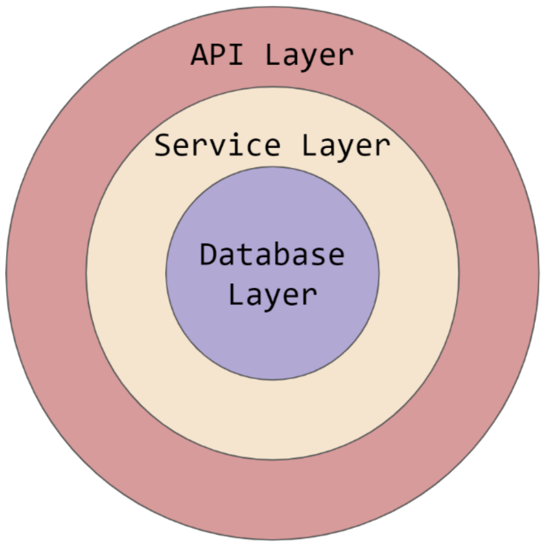


Figure 7. The Relationship between the Layers of the REST API

These layers loosely model the Open Systems Interconnection (OSI) model to achieve low coupling and high cohesion [45]. Each layer follows the Dependency Rule, which states that an inner layer should never rely on, or have access to, anything from an outer layer, but the outer layers rely on the inner layers [46]. In other words, the database layer cannot know anything about the service and API layers and the service layer cannot know about the API layer. However, the service layer can know about the database layer and the API layer can know about both the service and database layers (and therefore have access to the information in the layers that they have knowledge on). All layers utilize the logrus package [47] to provide custom error messages that contain optional fields for referencing specific experiments or variants for debugging purposes.

4.2.1. Database Layer

The database layer consists of a database that holds all the persistent data, which in our project are the experiments and variants, as seen in Figure 8.

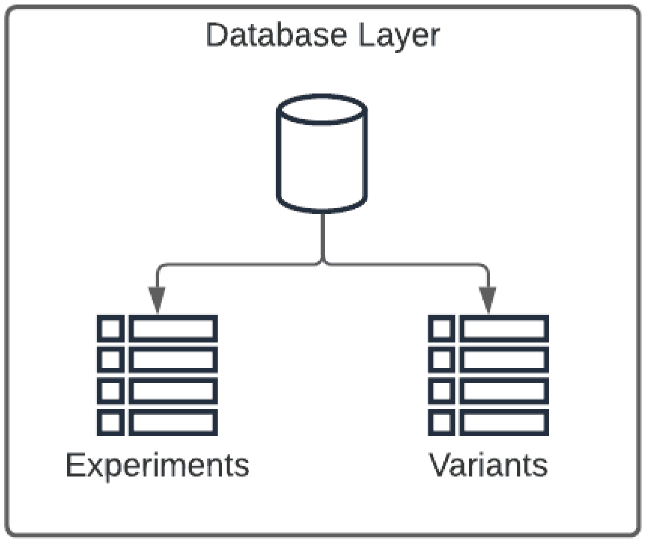
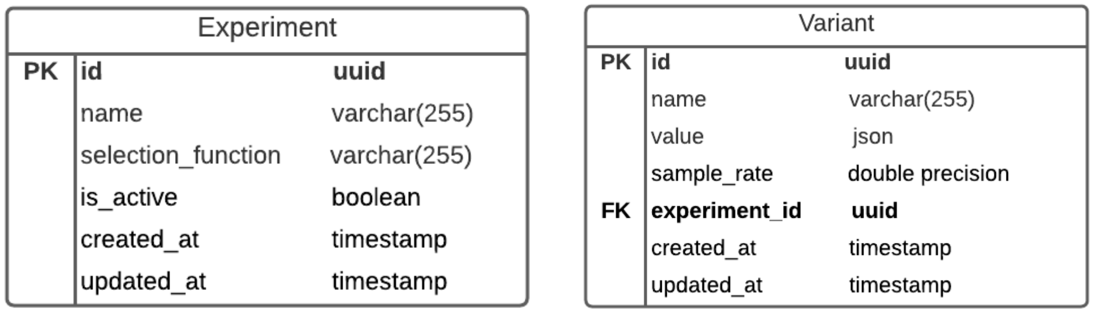


Figure 8. The Database Layer

Thus, there are two tables in the database, the experiments and variants, that both have an ID field as their primary keys, which the database generates upon creation of a given experiment and/or variant. The experiment table also contains a name, selection function, active status, and timestamps that signify when the database created or modified an experiment. Similarly, the variant table also has other fields in addition to the ID – a name, value which is a JSON object, sample rate, experiment ID and timestamps that signify when the database created or modified the variant. Variants have an experiment ID field because this acts as the foreign key that links the experiments and variants tables together. Furthermore, when there is a call to remove an experiment from the experiments table, this will also trigger the removal of all associated variants from the variants table. All the experiments and variants fields are shown in Figure 9.



Experiment		
PK	**id**	**uuid**
	name	varchar(255)
	selection_function	varchar(255)
	is_active	boolean
	created_at	timestamp
	updated_at	timestamp

Variant		
PK	**id**	**uuid**
	name	varchar(255)
	value	json
	sample_rate	double precision
FK	**experiment_id**	**uuid**
	created_at	timestamp
	updated_at	timestamp

Figure 9. Experiment Model and Variant Model

We used an external package called sqlx [48] to assist with performing all calls to the database. While we used PostgreSQL because of its native JSON support [49], this layer supports other databases as well, such as MySQL. In this layer, we instantiate the connection to the database and provide wrapper methods on the database functions, Get, Select and Query, as these appear frequently in the Service layer. Get, which we used when fetching one experiment or variant by its ID and when creating and updating an experiment or variant, returns a single query row. Select, which we used when fetching multiple experiments or variants, returns multiple query rows. Query, which we used when deleting experiments and variants from the database, returns all affected rows.

4.2.2. Service Layer

The service layer handles all the database calls for the experiments and variants and the syncing required for integration with the Edge Dictionaries, as seen in Figure 10.

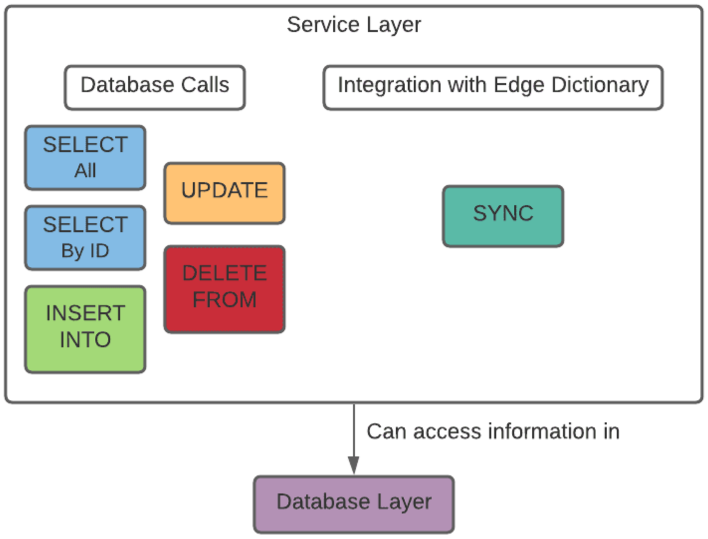


Figure 10. The Service Layer

These database calls consist of getting all experiments and variants, getting a specific experiment or variant by its ID, creating experiments and variants, updating them, and removing them from the database. In other words, the calls perform CRUD (Create Read Update Delete) operations on the data. Each database call requires the execution of an SQL statement, where the methods that get experiments and variants use SQL's SELECT, those that create use SQL's INSERT INTO, those that update use SQL's UPDATE, and those that remove use SQL's DELETE FROM.

In all the methods that get experiments, the REST API executes a query to get all the corresponding variants as well. For the create method, there is an initial check on the variant sample rates to ensure that none are negative numbers and that the sum of all the variants' sample rates for the experiment equals 100%. If this check passes, then the function creates the experiment in the experiments table and adds all its associated variants to the variants table. The create method on the variants adds the variant to the database. To update an experiment, the update method first fetches it from the database and if it exists, then the input experiment or variant will replace the corresponding one in the database. Similar to the create experiment method, the update method checks that the sample rates of the input variant are non-negative and sum to 100%. The update method on variants updates the variant in the database.

Moreover, the service layer contains the necessary support to integrate Fastly's Edge Dictionaries into the REST API. To do this, we used the Go Fastly package, which is a Go API client for interacting with the Fastly API that allows us to write to an Edge Dictionary [50]. Using the package, we instantiate a Fastly client that updates the Edge Dictionary with the experiments.

4.2.3. API Layer

The API layer contains all the HTTP handlers for the methods in the service layer and provides the connection between the REST API and the UI, as seen in Figure 11.

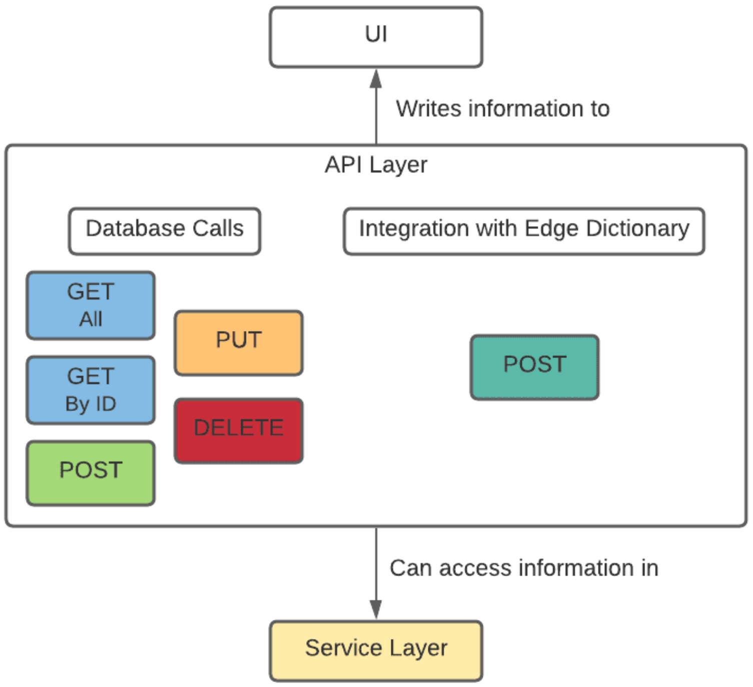


Figure 11. The API Layer

To match incoming HTTP requests to their respective handler, we used an external package called Gorilla Mux [51]. This package also provided support for middleware, such as requiring authentication with an API key, and enabling cross-origin resource sharing (CORS) to allow our UI to access the REST API.

The HTTP handlers featured in this layer handle GET, POST, PUT, and DELETE HTTP requests, which are all CRUD operations. In the GET (by ID), PUT, and DELETE requests, the corresponding handlers read the path variable to identify which specific experiment or variant the request will fetch, update, or remove from the database, respectively. For a GET request that fetches all of the experiments in the database, there are two additional parameters, limit and offset, that describe how to group and display the experiments. Limit refers to the maximum number of experiments that the handler method will return at one time, whereas offset represents the number of experiments to skip before beginning to return experiments. Both POST and PUT requests require additional data, the experiments and variants, that their handler methods read from the request body using the ioutil package [52]. This package turns the response body into a byte array, which the handler methods transform into the corresponding experiment and variants using the JSON package. If any of the requests fail, or return an HTTP status error code, the corresponding method will return an error message.

Furthermore, this layer handles the integration with the Edge Dictionary by publishing the experiments and variants through a POST request. The handler method for this performs a GET request on all experiments and then attempts to publish each one to the Edge Dictionary. If any integrations fail, the handler method returns an error message.

4.3 UI

The UI allows experiment designers to configure their experiments. Experiment designers, who may or may not have programming experience, may not want to use the REST API directly through developer networking tools such as Postman, Paw, or Insomnia. Therefore, in order to make the REST API functionality accessible to experiment designers, we developed a UI for the aforementioned CRUD methods.

4.3.1. UI Creation Process

In order to begin this process, we first designed an outline, shown in Figure 12, in the popular design prototyping tool, Sketch [53]. This application, somewhat akin to Photoshop, is a popular tool for mocking up what a website or application might look like.

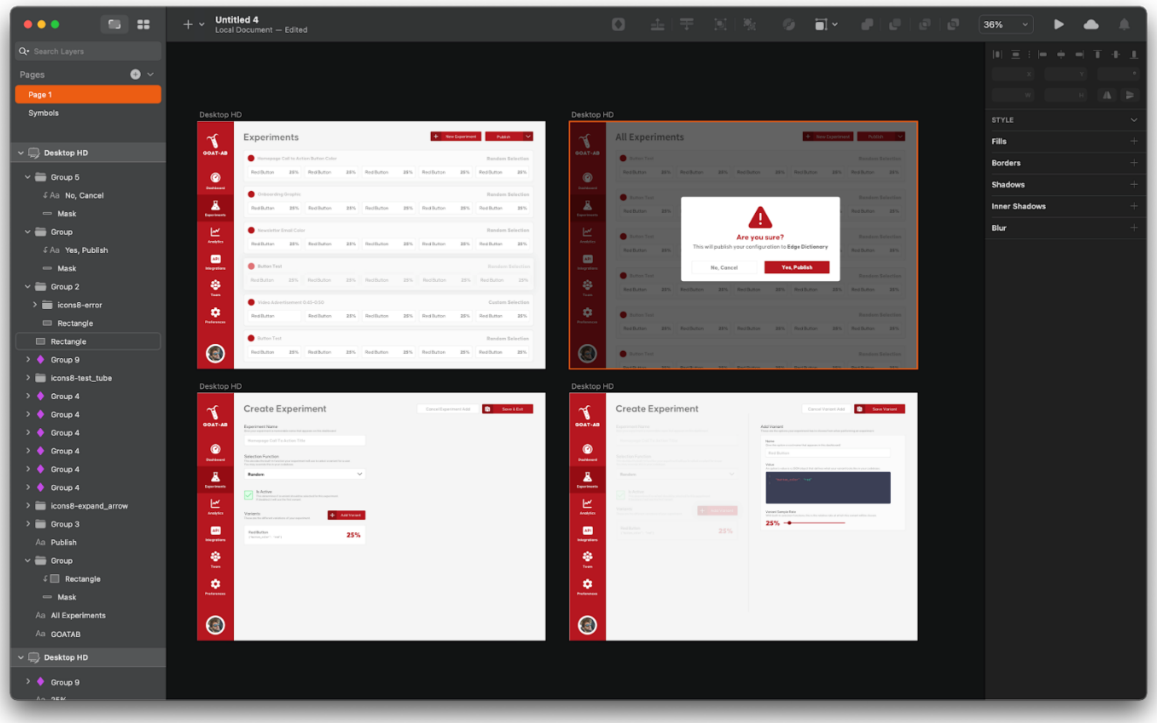
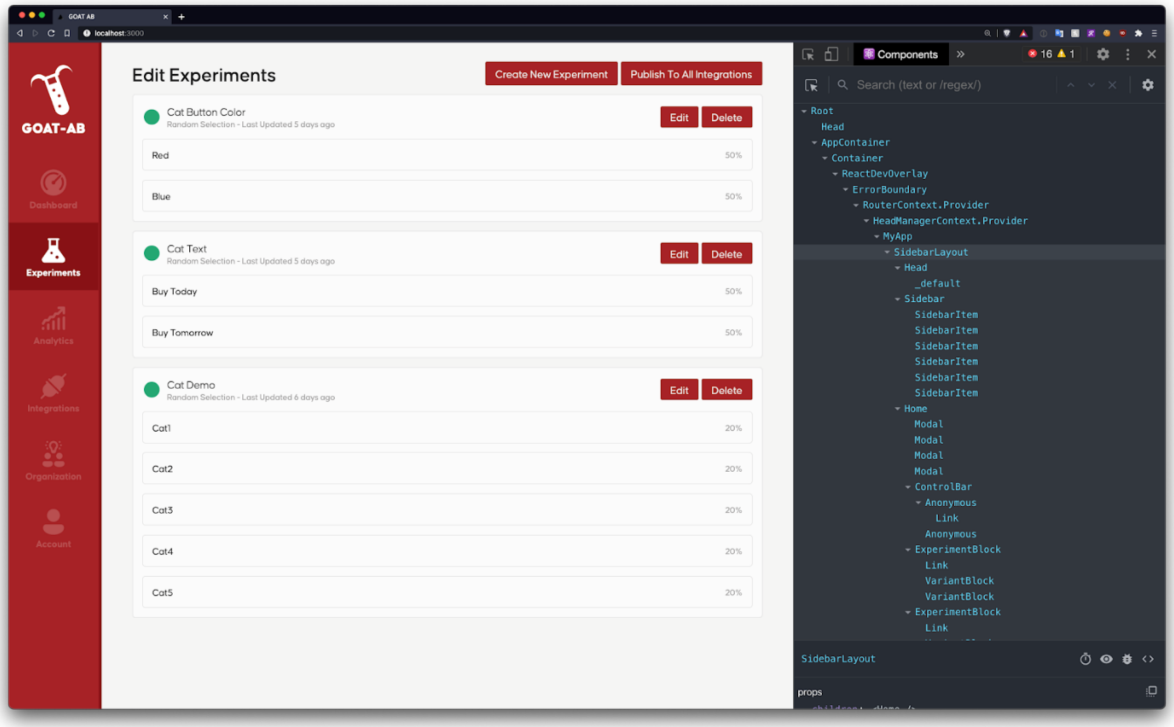


Figure 12. The Sketch Prototype

The Sketch prototype includes the three main screens of the application: the page to view experiments, create experiments, and add variants to an experiment. The first screen includes a list of all current experiments, variants that belong to them, and buttons to edit or delete experiments. This first screen also includes a button to create a new experiment and one to publish the experiments to a Fastly Edge Dictionary, where the Rust Crate would read the configuration. The second screen is the form to create and update experiments. Here experiment designers can edit the name, selection function, status, and variants of the experiment. The third page allows for the configuration of the name, value, and sample rate of variants. During the design process, we merged the experiment and variant configuration pages for the sake of simplicity, so users have an awareness that the variant is tied to the experiment.

We then moved on to implementing the designs in code. We built this user interface in JavaScript, utilizing the React framework alongside the Next.js module for React.

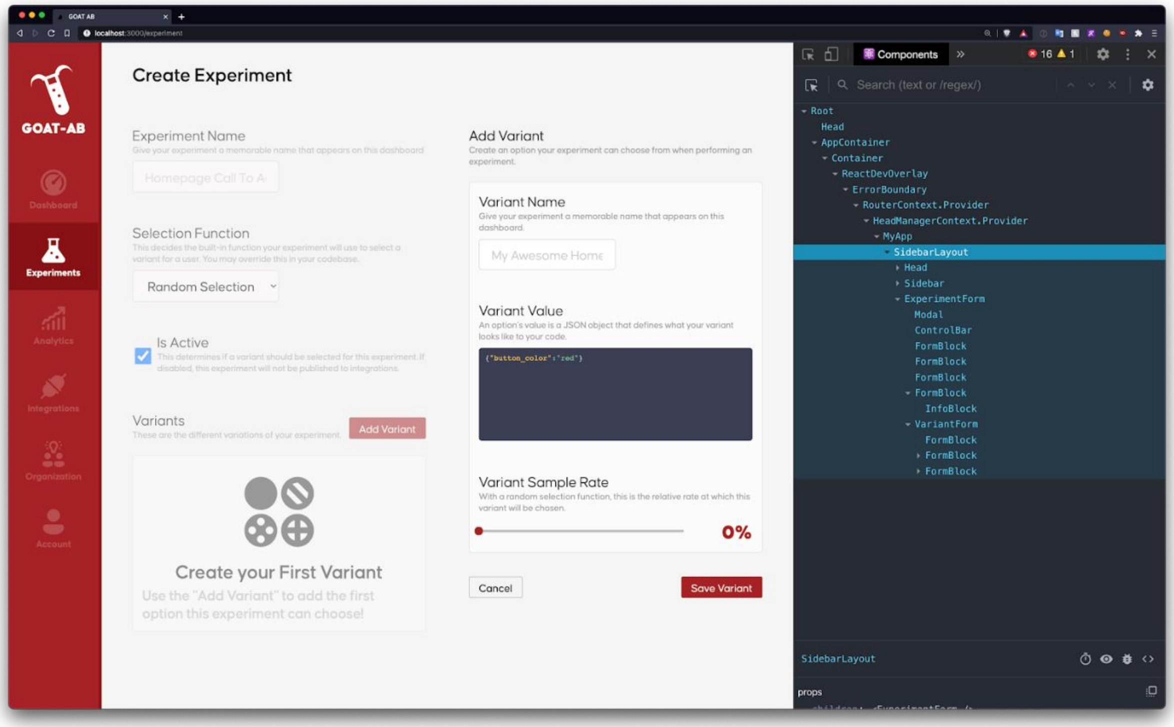
Utilizing these features, we built two main screens. These screens are near replicas of the aforementioned designs, but are created with React and Next.js components and use real data from the REST API instead of the shapes and mocked data in Sketch. In the React version, we worked to separate the different parts of the page into separate reusable components, shown in Figure 13.



```
▾ Root
    Head
  ▾ AppContainer
    ▾ Container
      ▾ ReactDevOverlay
        ▾ ErrorBoundary
          ▾ RouterContext.Provider
            ▾ HeadManagerContext.Provider
              ▾ MyApp
                ▾ SidebarLayout
                  ▾ Head
                      _default
                  ▾ Sidebar
                      SidebarItem
                      SidebarItem
                      SidebarItem
                      SidebarItem
                      SidebarItem
                      SidebarItem
                  ▾ Home
                      Modal
                      Modal
                      Modal
                      Modal
                    ▾ ControlBar
                      ▾ Anonymous
                          Link
                        Anonymous
                    ▾ ExperimentBlock
                        Link
                        VariantBlock
                        VariantBlock
                    ▾ ExperimentBlock
                        Link
```

Figure 13. The Experiment Screen with Components Shown

As shown in Figure 13, we separated the parts of the page down into “Sidebar,” “ExperimentBlock,” “VariantBlock,” “ControlBar,” and “Modal” components (amongst others) for the sake of modularity, reusability and testability, as we use them throughout the UI.



Create Experiment

Experiment Name
Give your experiment a memorable name that appears on this dashboard
Homepage Call To A

Selection Function
This decides the built-in function your experiment will use to select a variant for a user. You may override this in your codebase.
Random Selection

Is Active
This determines if a variant should be selected for this experiment. If disabled, this experiment will not be published to integrations.

Variants
These are the different variations of your experiment.

Create your First Variant
Use the "Add Variant" to add the first option this experiment can choose!

Add Variant
Create an option your experiment can choose from when performing an experiment.

Variant Name
Give your experiment a memorable name that appears on this dashboard.
My Awesome Home

Variant Value
An option's value is a JSON object that defines what your variant looks like to your code.

```
{"button_color":"red"}
```

Variant Sample Rate
With a random selection function, this is the relative rate at which this variant will be chosen.
0%

```
▾ Root
    Head
  ▾ AppContainer
    ▾ Container
      ▾ ReactDevOverlay
        ▾ ErrorBoundary
          ▾ RouterContext.Provider
            ▾ HeadManagerContext.Provider
              ▾ MyApp
                ▾ SidebarLayout
                  ▸ Head
                  ▸ Sidebar
                  ▾ ExperimentForm
                      Modal
                      ControlBar
                      FormBlock
                      FormBlock
                      FormBlock
                    ▾ FormBlock
                        InfoBlock
                    ▾ VariantForm
                        FormBlock
                      ▸ FormBlock
                      ▸ FormBlock
```

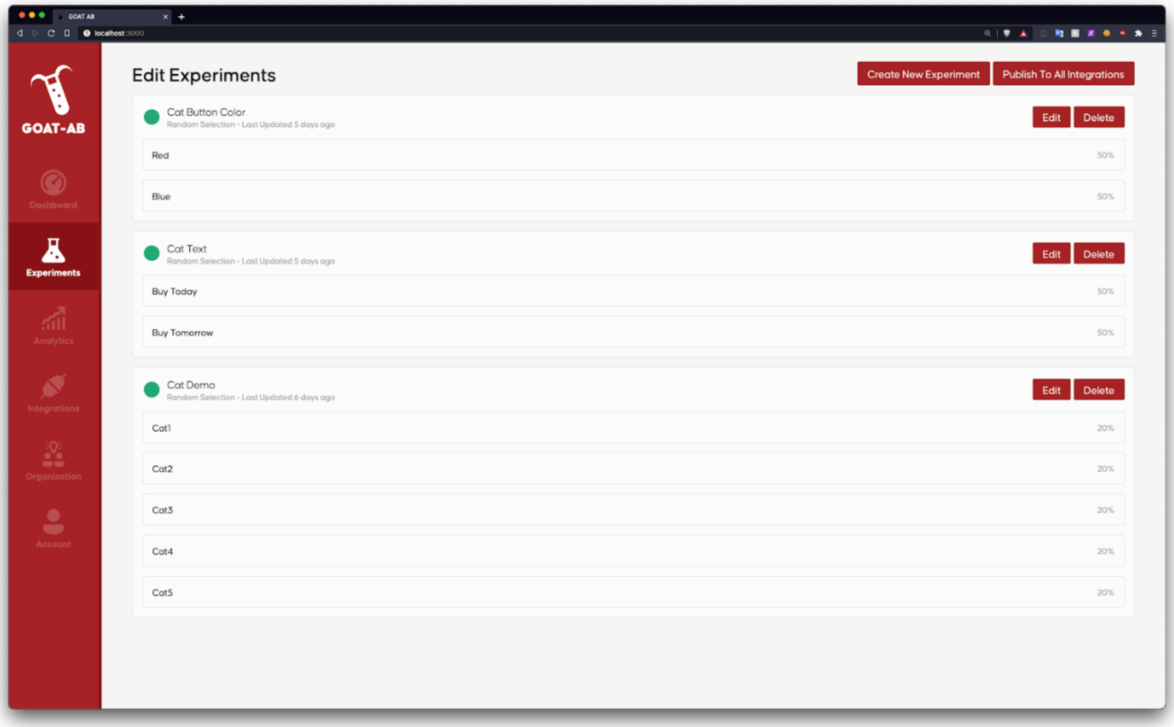
SidebarLayout
props

Figure 14. The Create Variant Screen with Components Shown

As shown in Figure 14, we see that the second screen, the form for creating and updating experiments and variants, reuses the “Sidebar,” “Modal,” and “ControlBar” components. Both screens drive the main functionality for the aforementioned REST API and are fully functional with reading, creating, updating, and deleting the experiments and variants that power the Rust Crate.

4.3.2. UI Features & Results

The UI of our end-to-end, A/B testing platform provides experiment designers the ability to run experiments in real-time. When first navigating to the UI, the home screen defaults to the experiment list, as shown in Figure 15. The sidebar, located on the left side of the screen, contains different functionalities for the platform that future iterations could implement.



Edit Experiments

Cat Button Color
Random Selection - Last Updated 5 days ago
Red 50%
Blue 50%

Cat Text
Random Selection - Last Updated 5 days ago
Buy Today 50%
Buy Tomorrow 50%

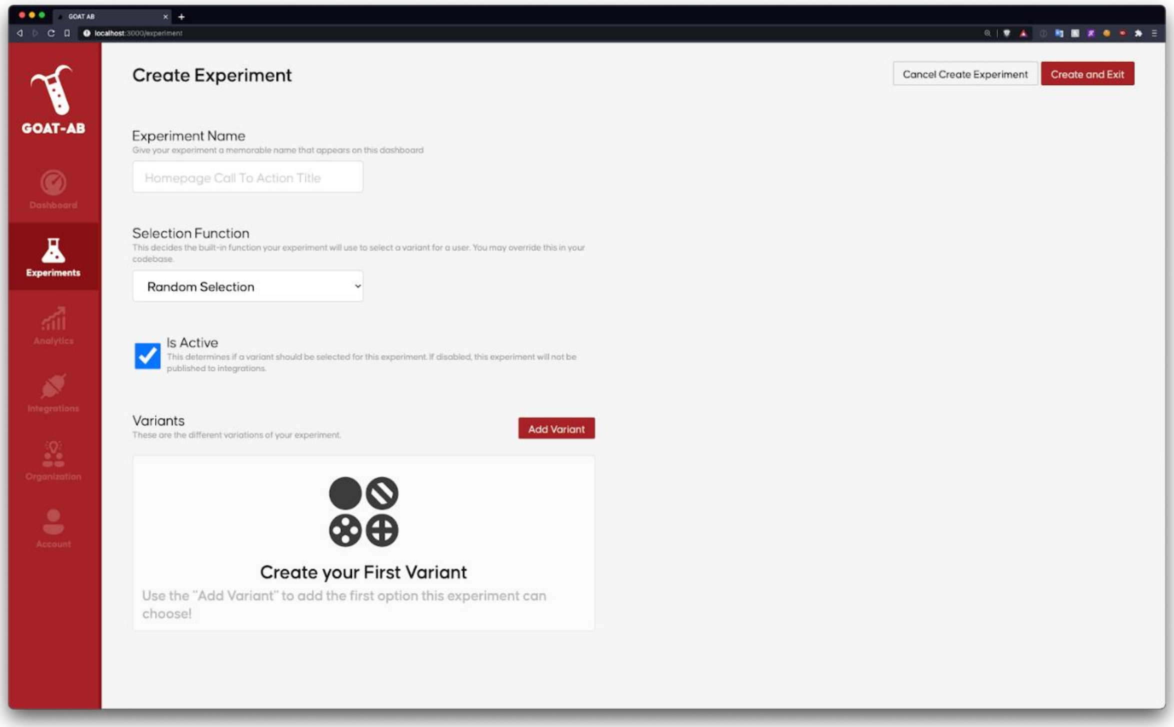
Cat Demo
Random Selection - Last Updated 6 days ago
Cat1 20%
Cat2 20%
Cat3 20%
Cat4 20%
Cat5 20%

Figure 15. GOAT-AB – Home Screen

There are five major components of the UI: create experiments, view all experiments, update experiments, delete experiments, and publish experiments. These are detailed in subsections 4.3.2.1-4.3.2.5.

4.3.2.1. Create Experiments

The Create Experiment page, as shown in Figure 16, allows users to add experiments through a form.



Create Experiment

Experiment Name
Give your experiment a memorable name that appears on this dashboard
Homepage Call To Action Title

Selection Function
This decides the built-in function your experiment will use to select a variant for a user. You may override this in your codebase.
Random Selection

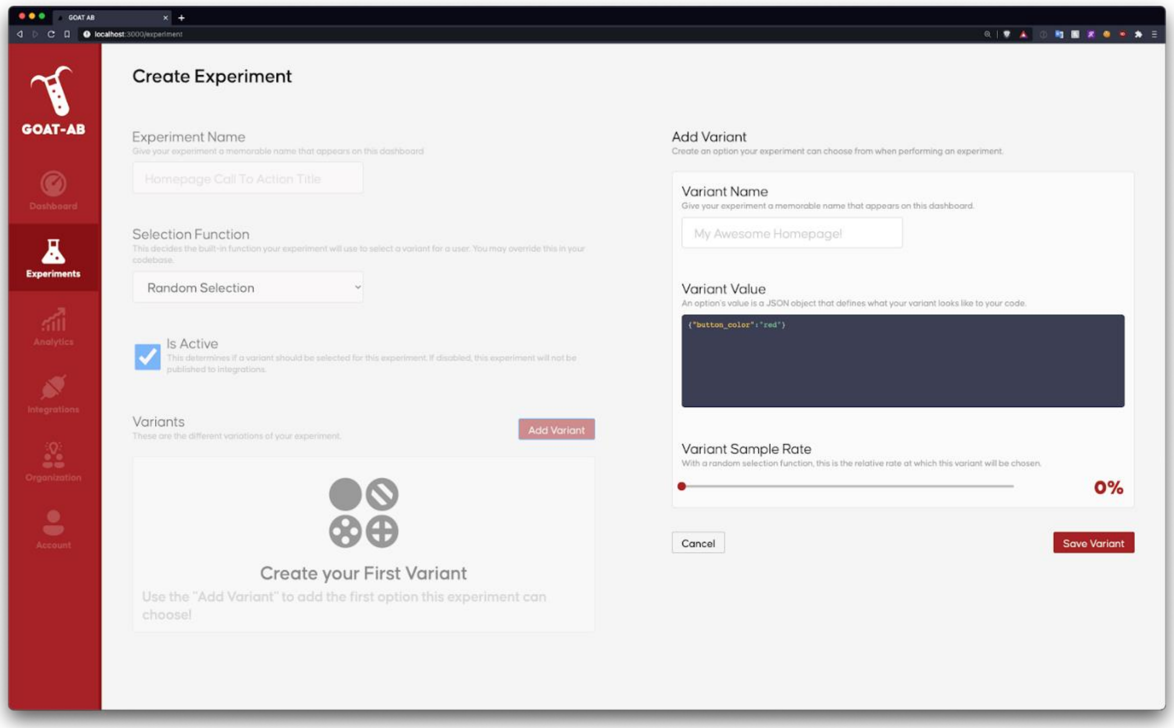
☑ Is Active
This determines if a variant should be selected for this experiment. If disabled, this experiment will not be published to integrations.

Variants
These are the different variations of your experiment.

Create your First Variant
Use the "Add Variant" to add the first option this experiment can choose!

Figure 16. GOAT-AB – Create Experiment Page

Each field has a small description to inform the developer of its purpose. The default for the selection function is “Random Selection,” yet developers can choose a different function using the dropdown or indicate that they will add a custom selection function. When adding variants to the experiment, a separate form will appear on the same screen, as shown in Figure 17.



Create Experiment

Experiment Name
Give your experiment a memorable name that appears on this dashboard
Homepage Call To Action Title

Selection Function
This decides the built-in function your experiment will use to select a variant for a user. You may override this in your codebase.
Random Selection

Is Active
This determines if a variant should be selected for this experiment. If disabled, this experiment will not be published to integrations.

Variants
These are the different variations of your experiment.
Add Variant

Create your First Variant
Use the "Add Variant" to add the first option this experiment can choose!

Add Variant
Create an option your experiment can choose from when performing an experiment.

Variant Name
Give your experiment a memorable name that appears on this dashboard.
My Awesome Homepage!

Variant Value
An option's value is a JSON object that defines what your variant looks like to your code.

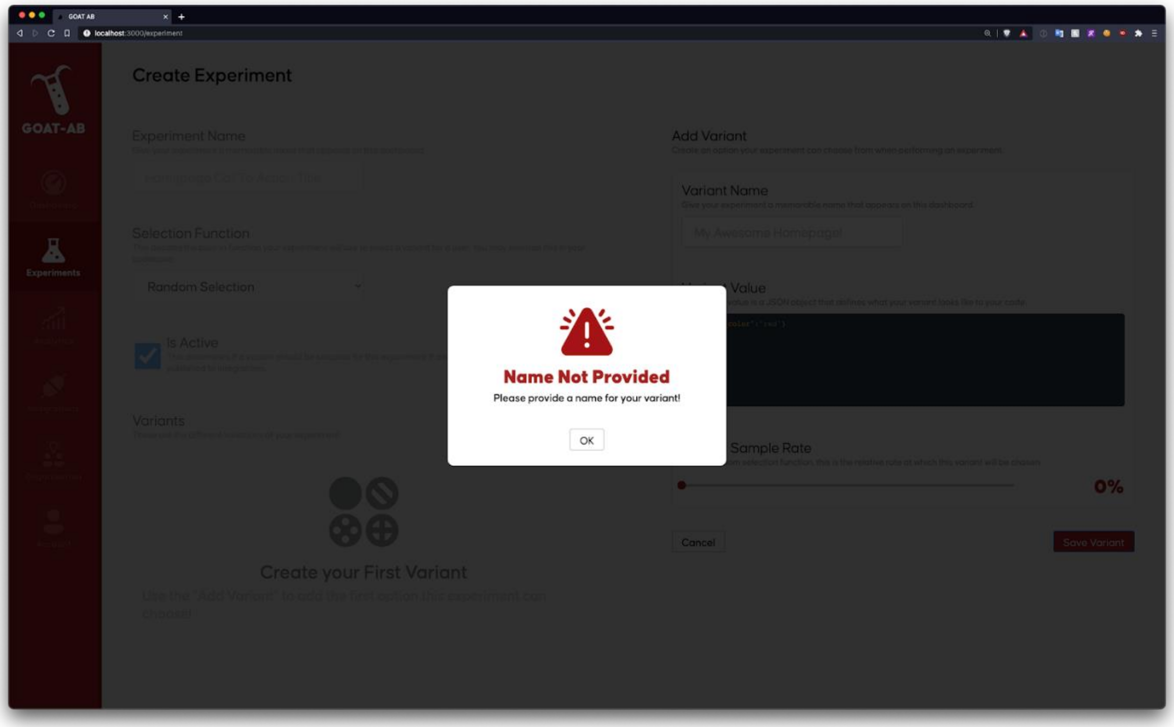
```
{"button_color":"red"}
```

Variant Sample Rate
With a random selection function, this is the relative rate at which this variant will be chosen.
0%

Cancel
Save Variant

Figure 17. GOAT-AB – Create Variants Page inside of Create Experiment Page

Similar to the Create Experiments form, each field in this Create Variants form has a small description. The value field accepts a proper JSON object, which the UI validates when developers click the “Save Variant” button. The sample rate slider ranges from 0 to 100 to ensure that developers do not input negative sample rates, and the corresponding percentage appears to the right of the slider. It is important to note that all variant fields are required and a popup, as shown in Figure 18, will appear on the screen if fields are missing. If the experiment designer decides to not create the variant, they can click on the “Cancel” button.

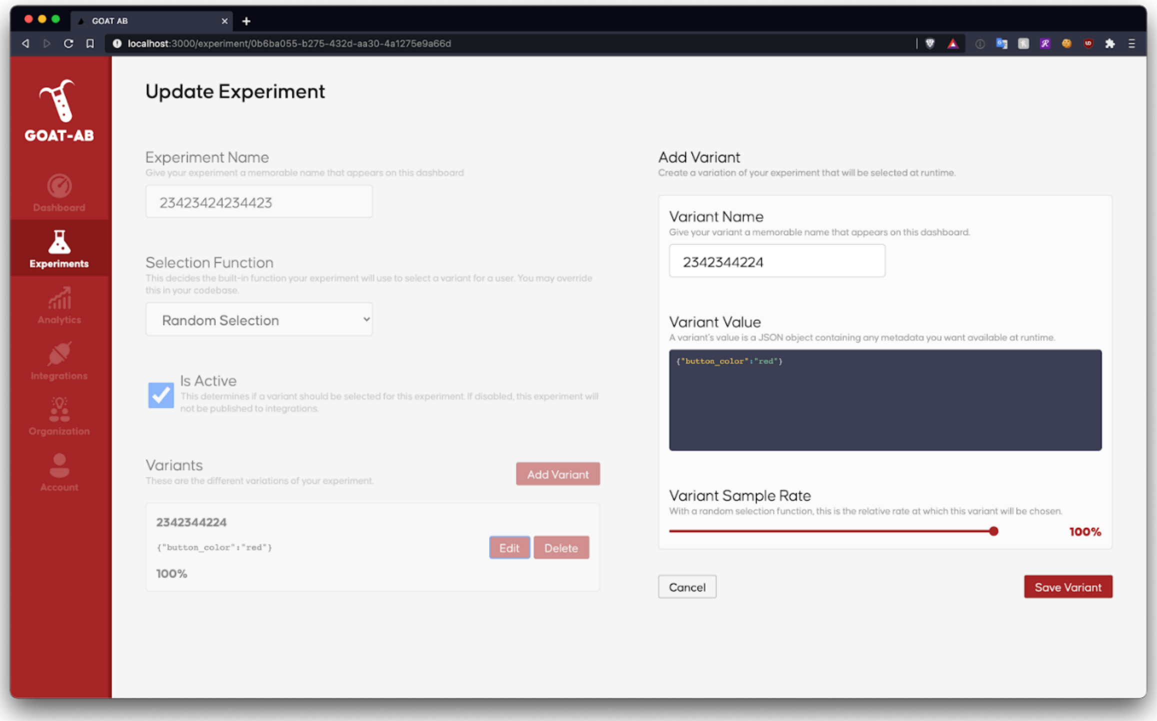


Name Not Provided

Please provide a name for your variant!

Figure 18. GOAT-AB – Missing Variant Fields Popup

If experiment designers want to modify the fields of a variant, they can click on the edit button next to the variant. The same form that appeared for creating variants shows on the screen filled with the information associated with the selected variant, as seen in Figure 19. Experiment designers are not able to successfully update the variant if any of the fields are empty or invalid.



Update Experiment

Experiment Name
Give your experiment a memorable name that appears on this dashboard
23423424234423

Selection Function
This decides the built-in function your experiment will use to select a variant for a user. You may override this in your codebase.
Random Selection

☑ Is Active
This determines if a variant should be selected for this experiment. If disabled, this experiment will not be published to integrations.

Variants
These are the different variations of your experiment.
Add Variant

2342344224
`{"button_color":"red"}`
100%
Edit Delete

Add Variant
Create a variation of your experiment that will be selected at runtime.

Variant Name
Give your variant a memorable name that appears on this dashboard.
2342344224

Variant Value
A variant's value is a JSON object containing any metadata you want available at runtime.

```
{"button_color":"red"}
```

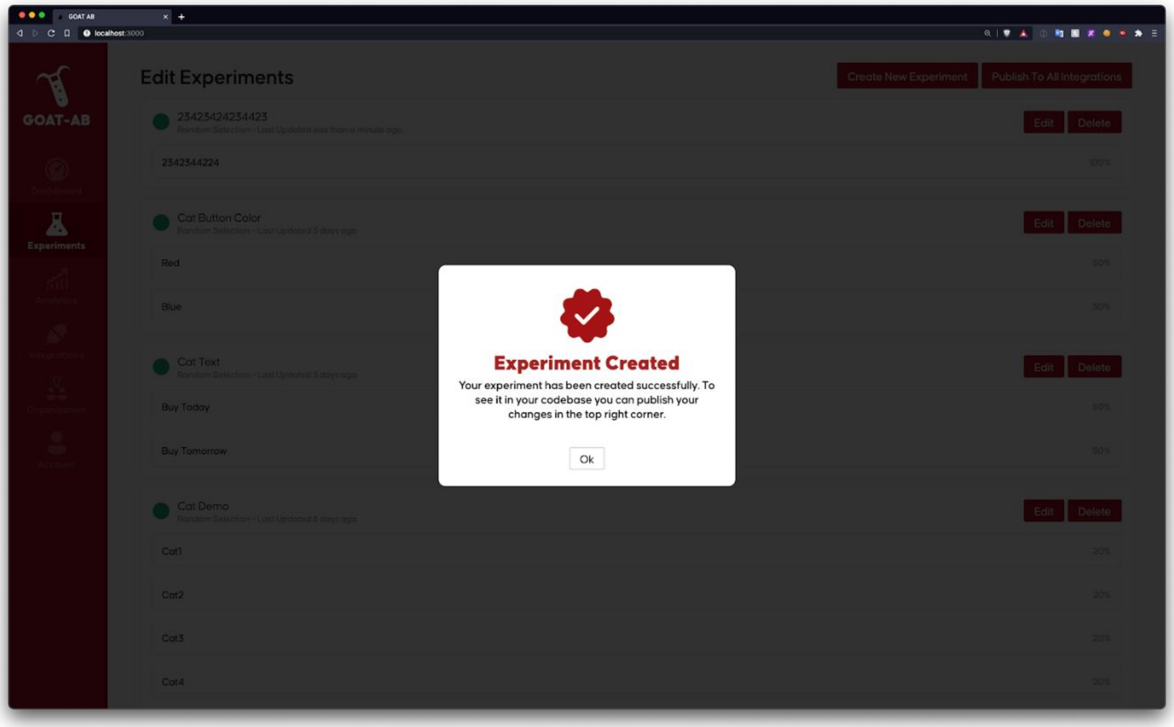
Variant Sample Rate
With a random selection function, this is the relative rate at which this variant will be chosen.
100%

Cancel Save Variant

Figure 19. GOAT-AB – Update Variant Page in Create Experiment Page

Additionally, if experiment designers create a variant, but later decide to delete it, they can click on the delete button next to the variant.

After the experiment designers finish filling out the form and adding variants, they can click on the "Create Experiment" form and will either see a success popup, as seen in Figure 20, or one of the failure popups, as seen in Figure 21 and Figure 22. It is important to note that all fields in this form are required and the sample rates of the variants must sum to 100%. If experiment designers decide to not create the experiment, they can click on the "Cancel" button.



Edit Experiments

23423424234423
Random Selection · Last Updated less than a minute ago
2342344224 100%

Cat Button Color
Random Selection · Last Updated 5 days ago
Red 50%
Blue 50%

Cat Text
Random Selection · Last Updated 5 days ago
Buy Today 50%
Buy Tomorrow 50%

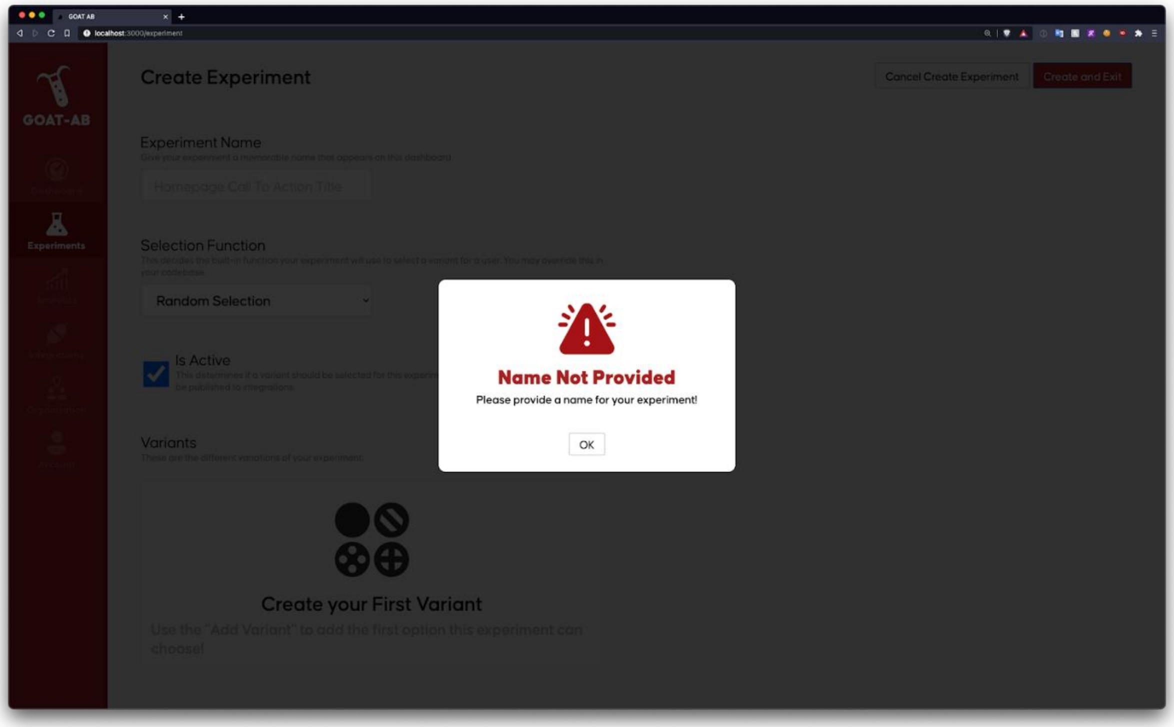
Cat Demo
Random Selection · Last Updated 6 days ago
Cat1 20%
Cat2 30%
Cat3 20%
Cat4 30%

Experiment Created

Your experiment has been created successfully. To see it in your codebase you can publish your changes in the top right corner.

Ok

Figure 20. GOAT-AB – Successful Experiment Creation Popup



Create Experiment

Experiment Name
Give your experiment a memorable name that appears on this dashboard
Homepage Call To Action Title

Selection Function
This decides the built-in function your experiment will use to select a variant for a user. You may override this in your codebase.
Random Selection

☑ Is Active
This determines if a variant should be selected for this experim... be published to integrations.

Variants
These are the different variations of your experiment.

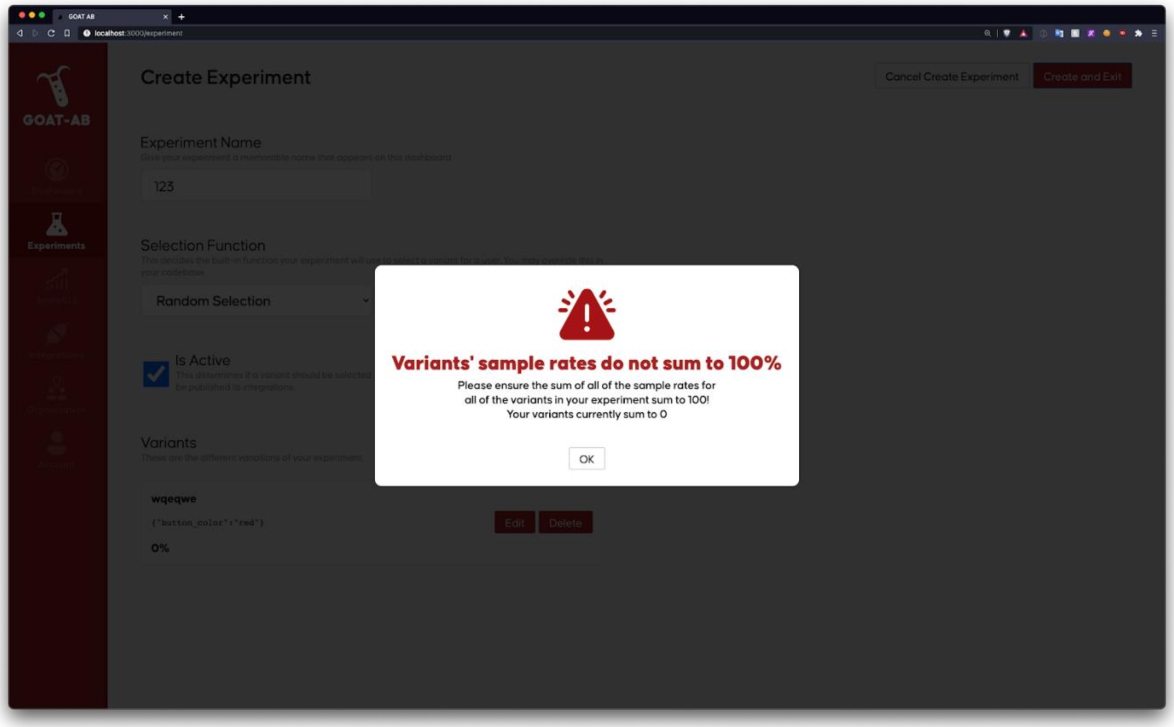
Create your First Variant
Use the "Add Variant" to add the first option this experiment can choose!

Name Not Provided

Please provide a name for your experiment!

OK

Figure 21. GOAT-AB – Missing Name Field Popup



Create Experiment

Experiment Name

Give your experiment a memorable name that appears on this dashboard.

123

Selection Function

This decides the built-in function your experiment will use to select a variant for a user. You may override this in your codebase.

Random Selection

Is Active

This determines if a variant should be selected ... be published to integrations.

Variants

These are the different variations of your experiment.

wqeqwe

{"button_color":"red"}

0%

Variants' sample rates do not sum to 100%

Please ensure the sum of all of the sample rates for all of the variants in your experiment sum to 100!
Your variants currently sum to 0

Figure 22. GOAT-AB – Sample Rate Sum Not 100% Popup

4.3.2.2. View Experiments

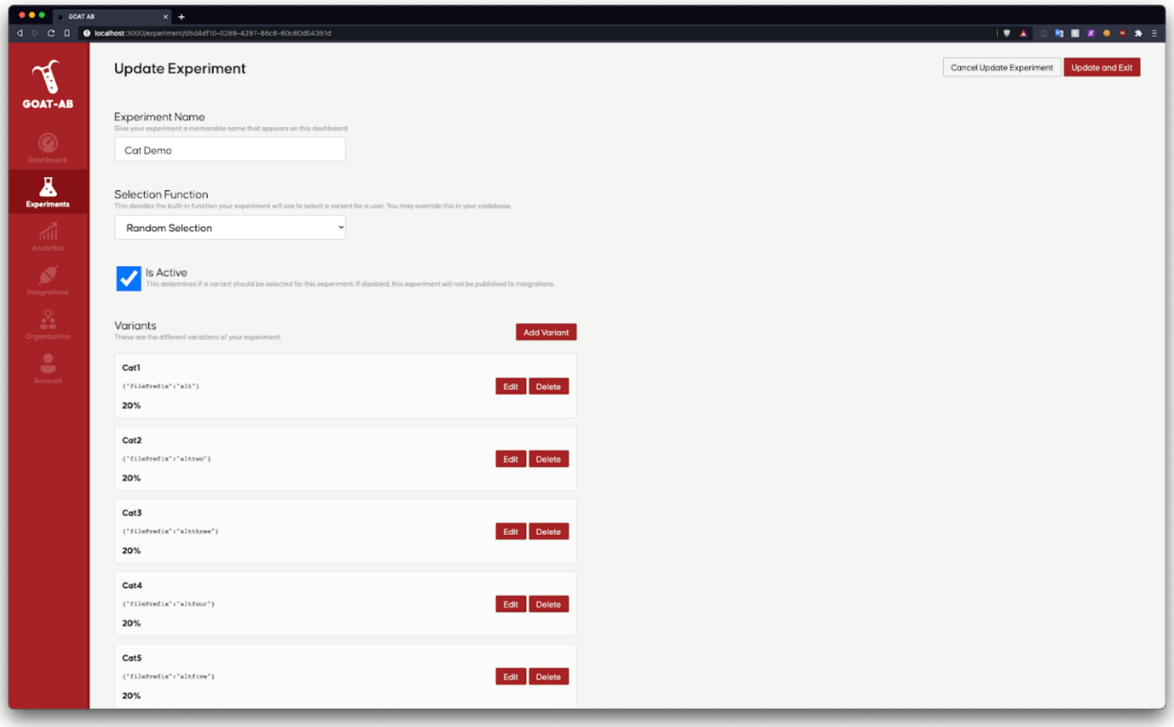
After creating an experiment, experiment designers can view their experiments on the home screen, as seen in Figure 15 above. Each experiment block contains the following fields:

- The name of the experiment
- The approximate time when the experiment was last created or modified
- A filled-in circle indicating whether the experiment is active or not (green indicates the experiment is active and red indicates it is not)
- The list of variants with their respective sample rates associated with the experiment

On the right-hand side of each experiment block, there are two buttons—clicking on the edit button indicates that the user would like to edit the experiment and clicking on the delete button indicates that the user would like to delete the experiment.

4.3.2.3. Update an Experiment

By clicking on the edit button of an experiment, experiment designers will see the Update Experiment page, as seen in Figure 23, that mimics the Create Experiment page, except that the fields contain the selected experiment information. It is important to note that in order to update the experiment, all fields in this form must be filled out and the sample rates of the variants must sum to 100%. If developers decide to not update the experiment, they can click on the "Cancel Update Experiment" button.



Update Experiment

Experiment Name
Give your experiment a memorable name that appears on this dashboard
Cat Demo

Selection Function
This decides the built-in function your experiment will use to select a variant for a user. You may override this in your codebase.
Random Selection

[✓] Is Active
This determines if a variant should be selected for this experiment. If disabled, this experiment will not be published to integrations.

Variants
These are the different variations of your experiment.

Cat1
{"filePrefix":"alt"}
20%

Cat2
{"filePrefix":"alttwo"}
20%

Cat3
{"filePrefix":"altthree"}
20%

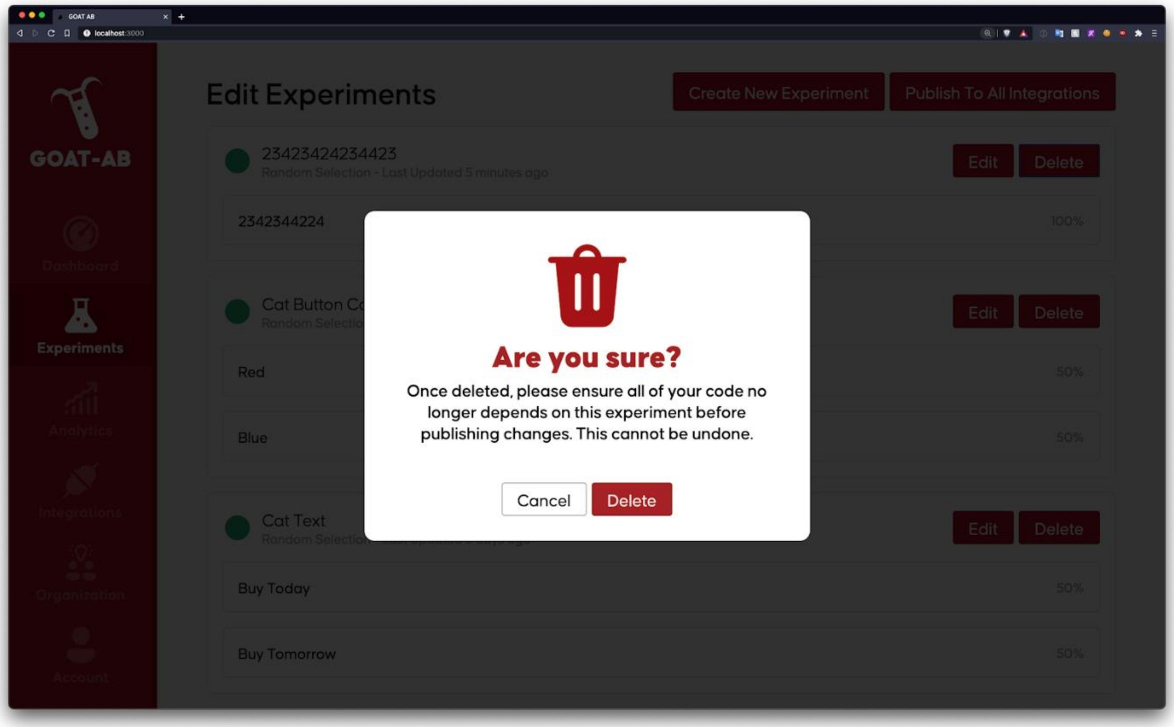
Cat4
{"filePrefix":"altfour"}
20%

Cat5
{"filePrefix":"altfive"}
20%

Figure 23. GOAT-AB – Update Experiment Page

4.3.2.4. Delete an Experiment

To delete an experiment, experiment designers can click on the delete button of an experiment. Before deleting the experiment, a popup will appear on the screen, as shown in Figure 24, to verify that the experiment designer wants to delete the selected experiment.



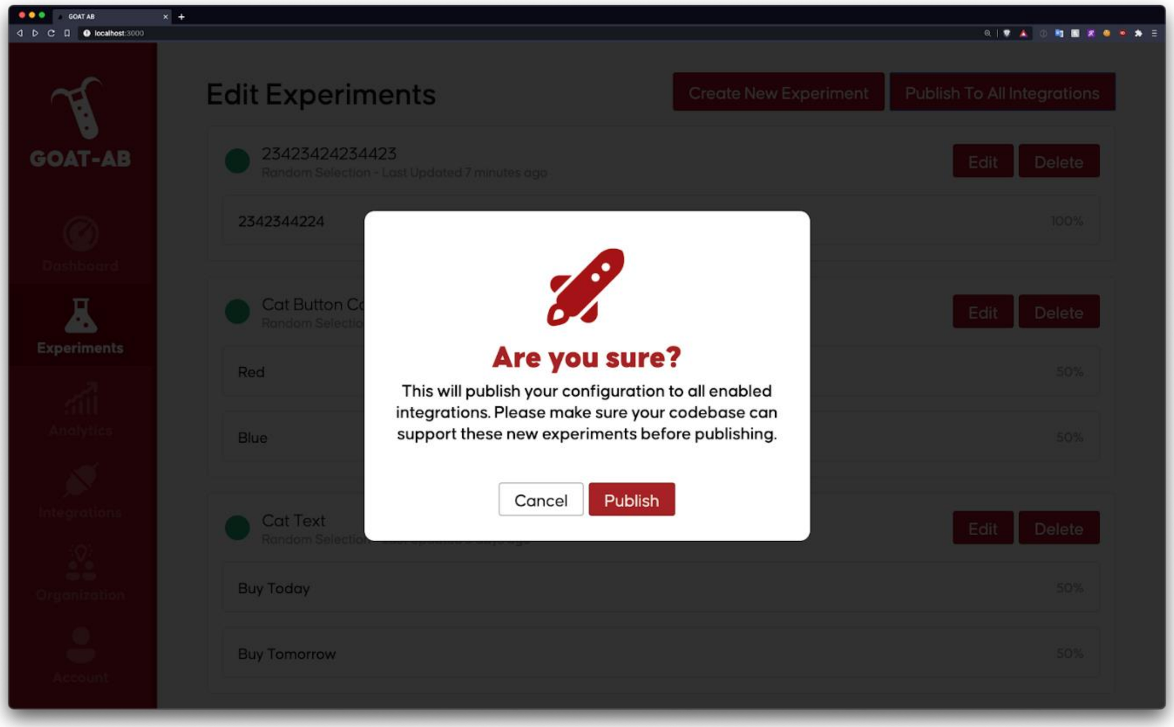
Are you sure?

Once deleted, please ensure all of your code no longer depends on this experiment before publishing changes. This cannot be undone.

Figure 24. GOAT-AB – Delete Experiment Popup

4.3.2.5. Publish Experiments

When experiment designers want to publish their experiments on C@E, they can click on the "Publish All Experiments" button at the top right-hand side of the home screen. This publishes the experiments to Fastly's Edge Dictionaries. Before doing so, however, it asks the experiment designer to confirm that the codebase attached to the experiment can handle the new values as seen in Figure 25.



Are you sure?

This will publish your configuration to all enabled integrations. Please make sure your codebase can support these new experiments before publishing.

Figure 25. GOAT-AB – Publish Experiment Popup

4.4. Results

To evaluate our platform, we performed tests on the functional components (Rust Crate, REST API and UI) and created a demo using the platform.

4.4.1. Testing

Testing was an integral part of our platform to ensure that each component was not only effective, but also responded appropriately to a variety of configurations and use cases. We determined the best ways to test our platform based on advice from our advisors and our previous knowledge and experience with testing. The testing standard we had for our code development is as follows:

1. For the Rust crate and REST API, we wrote unit tests for all the functions.
2. For the UI, we manually tested by running the service and validating that all the buttons and features worked as programmed.

3. For any instances where we utilized C@E, we used Fastly’s log tailing, which allows developers to see their function logs in near-real-time [54].
4. For the end-to-end functionality, we developed a demo that showcases a sample experiment that changed parts of a video at the edge.

4.4.1.1 Rust Crate

To test the Rust Crate, we wrote unit tests for each of the major functions: validating the configuration file, initializing the crate, the selection function, and bucketing users. For each function, we utilized the assertion crate in Rust’s standard library, which either evaluates to true if the assertion passes and panics if the assertion fails, which raises the panic! macro. Panic! is a macro that allows a program to terminate immediately when the program reaches an unrecoverable state and provides detailed feedback on the specific error.

To test the validate_configuration function, we first ensured that the configuration string contained complete information for each of the experiments and variants. Each sample configuration string is inside of a fixtures directory, rather than inline for readability. Then, we used assert! and assert_eq! to assert that the fields of the experiments and variants were valid, in terms of the fields having appropriate lengths and data types. Since we created an enum to hold all the potential errors that could result from the validation, we also checked to make sure that error returned from the test was the same as what we expected. Similar to validate_configuration, we used assert_eq! to test initialize_from_string. To test the selection function and bucket_all, we checked that each method returned one of the expected variants of an experiment and that the same variant gets returned for a specific user each time for 100 attempts. Each test for the crate passed, which provides an assurance that the crate is functional.

4.4.1.2 REST API

To evaluate the REST API, we used mocking to unit test the CRUD methods for experiments and variants. Unit testing external dependencies, such as network requests and database calls, is cumbersome, unreliable, and outside the scope of this project. Mocking provides the solution to this by replacing external dependencies with controlled objects that simulate their behavior. To apply mocking to the REST API, we created a mock interface that defined the functions necessary to make calls to the database. Then, instead of calling the actual functions that communicate with the database, we injected our mock interface into the handler functions, which handle the HTTP requests, so we could isolate them and ensure that they worked properly.

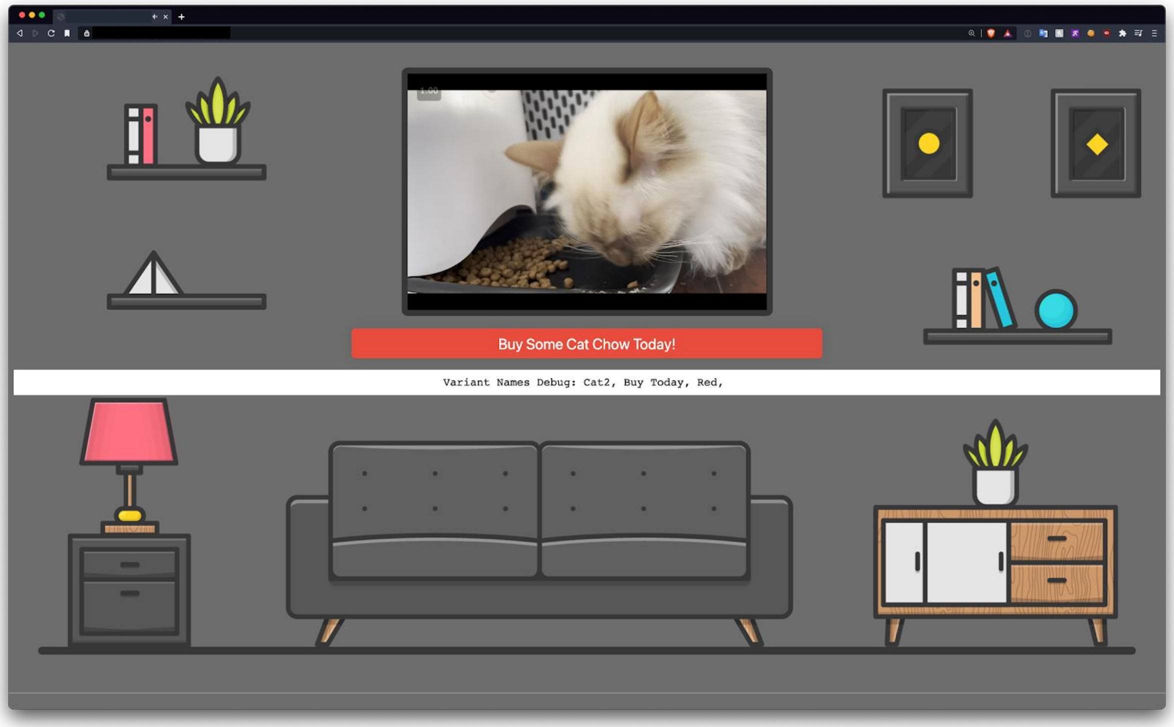
Since each handler function contained multiple test cases that resulted in repetitive code, we used table-driven tests to remove the repetition, produce tests, and provide a way to add more scenarios. In table-driven tests, the test function iterates over a table of test cases and performs the necessary evaluations on that test case. However, if one of the table-driven tests produces a fatal error, the remaining test cases never run. To resolve this, we created our table-driven tests using subtests, which allowed all tests in the table to run regardless of errors and associated a name with each test for individual testing. Each written test for the REST API passed and any that produced errors returned the expected and appropriate error message.

4.4.1.3 UI

We conducted manual tests for the UI since writing test cases for UIs is complicated. We ensured the accuracy of each input by displaying proper error messages, such as when the sample rates of the variants do not sum to 100% and if any fields had improper or no input. Each manual test of the UI passed our and our advisors' expectations.

4.4.2. Demo

While experiment designers have hands-on interactions with the platform with the UI described above, end users indirectly interact with the platform if they are a part of active experiments. We created an example experiment that inserts a different video clip into a commercial for cat food, as seen in Figure 26.

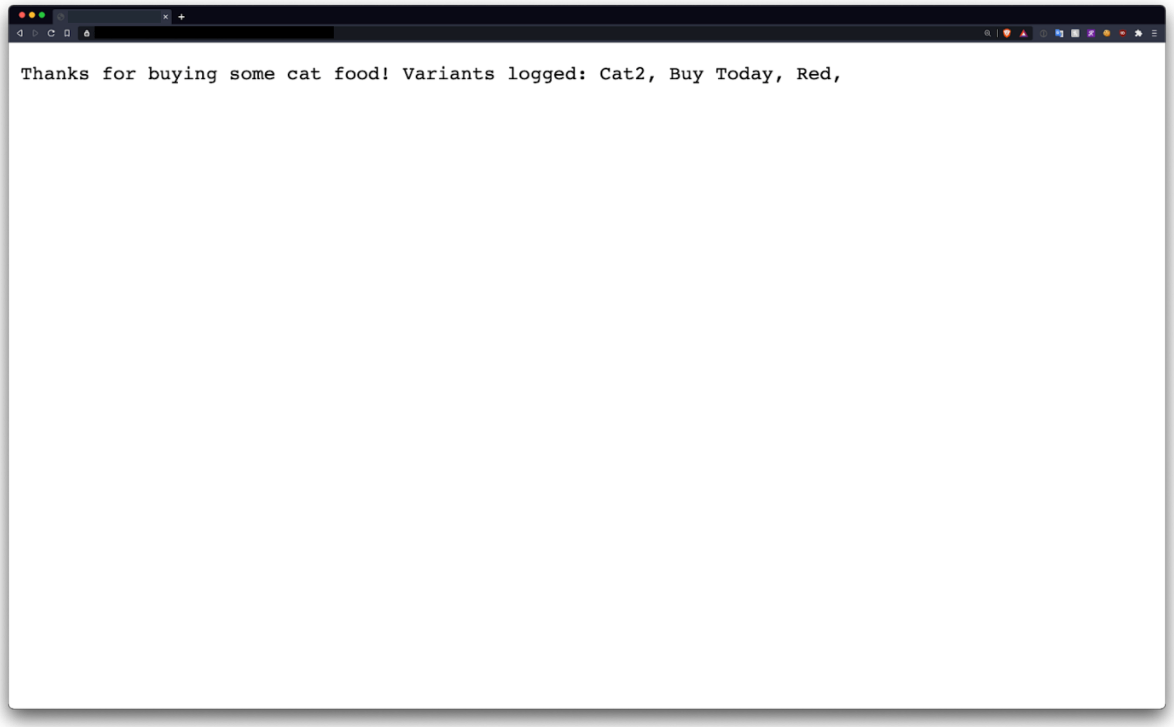


```
Variant Names Debug: Cat2, Buy Today, Red,
```

Figure 26. Example Experiment Set-up

The cat food commercial is for Purina Cat Chow and the inserted video clips are courtesy of Salman Saghafi, one of our sponsors, and his cats, Duchy and Fenny.

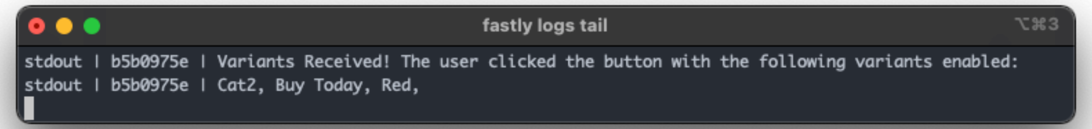
When end users go to the website that hosts this experiment, they would watch the commercial. The specific commercial relayed to the user depends on which variant the selection function bucketed the user into. Once a user clicks the button, the demo displays some “thank you” text, and lists the variants that the selection function bucketed them, as shown in Figure 27.



```
Thanks for buying some cat food! Variants logged: Cat2, Buy Today, Red,
```

Figure 27. Conversion Event Log Screen

In addition to the end-user seeing the variants, Fastly's logging service receives the variants, as shown in Figure 28, which shows that our framework can send the variants to any third-party (or future first-party) analytics services.



```
stdout | b5b0975e | Variants Received! The user clicked the button with the following variants enabled:
stdout | b5b0975e | Cat2, Buy Today, Red,
```

Figure 28. The Variants Received by Fastly Logs

The variants in Figures 26-28 show that even through three different requests and processes, the variants remain consistent, showing that the framework always buckets the end-user into the same variants as long as the bucketing uses the same end-user information.

5. Conclusion

In order to evaluate design choices and improve upon existing products and services, many companies employ A/B testing. With the recent shift towards edge computing and rapid prototyping, companies are beginning to support and integrate A/B testing platforms with the cloud to quickly iterate on experiments. While there are several A/B testing frameworks and platforms that satisfy complex experimental designs while maintaining Internet scalability and efficiency, none currently use Rust, which is the primary supported language of Fastly's serverless platform, C@E.

We developed an open-source, end-to-end A/B testing platform for Fastly, called GOAT-AB, which serves as a MVP for A/B testing on Fastly's C@E. Our platform provides a graphical UI for the creating, modifying, and deleting experiments. To store and keep track of experiments, the UI directly communicates to a REST API that handles HTTP requests and updates a database appropriately. When the developer publishes the experiments and begins testing on end-users, the Rust Crate containing the A/B testing functionality assigns the end-user to one variant per experiment. We designed each part of the platform with flexibility and abstraction in mind to provide the developer control over how they would like to design experiments.

To evaluate our platform, we individually tested each component of the platform. We wrote unit tests for the Rust Crate and REST API and manually tested the UI with error popups. To show the performance of our platform, we created a demo. The demo shows that when combined with Fastly's C@E, the platform provides working functionality for A/B testing and tracking for conversion events. Based on our tests, demonstration, and the expectations of our sponsors, the platform is a successful MVP for an A/B testing platform. Overall, GOAT-AB provides the foundation for future experimentation in a serverless space and shows a future for products that can run on Fastly’s C@E.

6. Future Work

As Fastly expands into the serverless market, there are several areas for improvement for future iterations of our platform. We divided our recommendations into two categories. The first category describes the minimum work that should be done once we leave this project. The second category includes interesting extensions of the platform that provide additional functionality for the end-user to improve their A/B testing.

6.1. Initial Work

After the completion of the project, there are several initial future implementations that directly correlate to the work already performed in this project. First, there should be some kind of maintenance for the code to ensure that we followed Fastly's code and documentation standards for Rust, Go and React/Next.js. This would also include more thorough testing for the UI using Jest, which is a testing framework to test JS and React code. Additionally, if the platform ties to the Fastly UI or Dashboard, it is crucial to conduct penetration tests, or pen tests, to identify any security weaknesses in the codebase. Adding proper authentication for databases among other infrastructure, would supplement these pen tests to ensure that unauthorized developers cannot access or modify the experiments and variants in the database. Moreover, it may be interesting to conduct performance evaluations to reduce potential overhead or discover any performance impacts. Lastly, it would be convenient for developers to have a larger range of selection functions to choose from when bucketing end-users into variants. One way to do this is to develop selection functions that include explicit bucketing characteristics, such as the end-user's location and browser, which the framework receives from C@E.

6.2. Further Functionality

Going forward, there are several interesting enhancements that we recommend for our platform. One enhancement is migrating to a scalable database, which would allow for the storage of increasing amounts of data without harming performance. Ideally, the REST API would directly store all persistent data in Fastly’s Edge Dictionaries rather than in a local database, which would further integrate our platform with C@E.

Another enhancement is publishing the experiments and variants to other integrations besides the Edge Dictionaries on C@E. Other companies could then use our platform to conduct A/B testing on the edge, which could provide some interesting results such as whether different integrations would bucket the same user into the same variant.

Furthermore, developing a full analytics suite with real-time analysis of experiments could be useful for experiment designers with little analysis experience. Combining analytics with dashboards allows experiment designers to keep track of all their experiments in an efficient way as they would no longer have to rely on third-party analytics services or databases. Fastly customers could use this platform to perform meta-analysis, possibly using deep learning, across various clients in order to make useful suggestions for experiments. With this analytics suite, Fastly will not track or analyze client activities. This proof of concept project has many interesting and viable directions in which development could continue.

7. References

[1] "What is A/B Testing? A Practical Guide With Examples," VWO. Available: https://vwo.com/ab-testing/ [Accessed: Mar. 18, 2021].

[2] "PlanOut | A Framework for Online Field Experiments," GitHub.io. Available: https://facebook.github.io/planout/ [Accessed: Mar. 15, 2021].

[3] "Optimizely," Optimizely. Available: https://www.optimizely.com/. [Accessed: Mar. 15, 2021].

[4] E. Elliott, "What is WebAssembly? The Dawn of a New Era," Medium, Jun. 18, 2015. Available: https://medium.com/javascript-scene/what-is-webassembly-the-dawn-of-a-new-era-61256ec5a8f6 [Accessed: Dec. 11 2020].

[5] "Rust on Compute@Edge," Fastly. Available: https://developer.fastly.com/learning/compute/rust/ [Accessed: Mar. 18, 2021].

[6] "Serverless Computing - The Complete Guide," Confluent. Available: https://www.confluent.io/learn/serverless-computing/ [Accessed: Mar. 17, 2021].

[7] M. Teodoro, "The power of serverless, 72 times over," Fastly, Nov. 11, 2020. Available: https://www.fastly.com/blog/the-power-of-serverless-at-the-edge [Accessed: Dec. 11 2020].

[8] M. G. Avram, "Advantages and Challenges of Adopting Cloud Computing from an Enterprise Perspective," Procedia Technology, volume 12, pp. 529-534, Dec, 2014. doi: 10.1016/j.protcy.2013.12.525.

[9] "What Is Cloud Computing? A Beginner’s Guide," Microsoft Azure. Available: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/ [Accessed: Dec. 11 2020].

[10] Cloudfare, "What is the Cloud? | Cloud Definition," Cloudfare. Available: https://www.cloudflare.com/learning/cloud/what-is-the-cloud/ [Accessed: Dec. 11 2020].

[11] S. Bhardwaj, L. Jain and S. Jain, "Cloud Computing: A Study of Infrastructure as a Service (IaaS)," International Journal of Engineering and Information Technology, vol. 2, no. 1, pp. 60-63, 2010.

[12] IBM Cloud Education, "What is IaaS (Infrastructure-as-a-Service)," IBM, Jul. 12, 2019. Available: https://www.ibm.com/cloud/learn/iaas [Accessed: Dec. 11 2020].

[13] D. Beimborn, T. Miletzki and S. Wenzel, "Platform as a Service (PaaS)," Business & Information Systems Engineering, volume 3, issue 6, pp. 381-384, 2011. doi: 10.1007/s12599-011-0183-3.

[14] "What is PaaS? Platform as a Service," Microsoft Azure. Available: https://azure.microsoft.com/en-us/overview/what-is-paas/ [Accessed: Dec. 11 2020].

[15] A. Davies, "10 Top PaaS Providers of 2021," DevTeam. Available: https://www.devteam.space/blog/10-top-paas-providers/ [Accessed: Dec. 11 2020].

[16] L. F. Albuquerque Jr., F. S. Ferraz, R. F. A. P. Oliveira and S. M. L. Galdino, "Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS," The Twelfth International Conference on Software Engineering Advances, pp. 206-212, 2017.

[17] P. Sroczkowski, "Cloud: IaaS vs PaaS vs SaaS vs DaaS vs FaaS vs DBaaS," Brainbub, Jan. 24, 2018. Available: https://brainhub.eu/blog/cloud-architecture-saas-faas-xaas/ [Accessed: Dec. 11 2020].

[18] "Fastly," Fastly. Available: https://www.fastly.com/ [Accessed: Dec. 11 2020].

[19] A. Sraders, "What is Fastly? Analysts say the best-performing tech stock during the pandemic may still have room to run," Fortune, Jun. 23, 2020. Available: https://fortune.com/2020/06/23/what-is-fastly-fsly-zoom-pandemic-tech-stocks-best-performers-stock-market-news/ [Accessed: Dec. 11 2020].

[20] T. Green, "The Fastly Growth Story Takes a Hit," Nasdaq, Oct. 16, 2020. Available: https://www.nasdaq.com/articles/the-fastly-growth-story-takes-a-hit-2020-10-16 [Accessed: Dec. 11 2020].

[21] "A/B Testing," Optimizely. Available: https://www.optimizely.com/optimization-glossary/ab-testing/ [Accessed: Dec. 11 2020].

[22] S. Jiang, J. Martin and C. Wilson, "Who's the Guinea Pig?: Investigating Online A/B/n Tests in-the-Wild," Proceedings of the Conference on Fairness, Accountability, and Transparency, pp. 201-210, Jan. 29-31, 2019, Atlanta, GA, USA. ACM, New York, NY, USA. doi: 10.1145/3287560.3287565.

[23] "Case Study: Electronic Arts," Optimizely, Jun. 2013. Available: http://pages.optimizely.com/rs/optimizely/images/Customer_Story_EA.pdf [Accessed: Mar. 15, 2021].

[24] C. E. Inc, "Crazy Egg Case Studies| The Best Website Optimization Tool," Crazy Egg. Available: https://www.crazyegg.com/case-studies [Accessed: Mar. 15, 2021].

[25] "Algorithms - Conductrics," Conductrics. Available: https://conductrics.com/machine-learning/ [Accessed: Dec. 11 2020].

[26] "Optimizely Full Stack," Optimizely. Available: https://www.optimizely.com/platform/full-stack/ [Accessed: Dec. 11 2020].

[27] "Firebase A/B Testing," Firebase. Available: https://firebase.google.com/docs/ab-testing [Accessed: Dec. 11 2020].

[28] R. Kohavi, A. Deng, B. Frasca, R. Longbotham, T. Walker and Y. Xu, "Trustworthy online controlled experiments: five puzzling outcomes explained," Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 786-794, Aug. 2012. doi: doi/10.1145/2339530.2339653.

[29] "Getting started with web - Optimizely web," Optimizely. Available: https://docs.developers.optimizely.com/web/docs/getting-started [Accessed: Dec. 11 2020].

[30] "Rust Programming Language," Rust. Available: https://www.rust-lang.org/ [Accessed: Mar. 17, 2021].

[31] "Golang," Golang. Available: https://golang.org/ [Accessed: Mar. 17, 2021].

[32] S. Aggarwal, "Modern Web-Development using ReactJS," International Journal of Recent Research Aspects, volume 5, issue 1, pp. 133-137, 2018. Available: http://ijrra.net/Vol5issue1/IJRRA-05-01-27.pdf [Accessed: Mar. 18, 2021].

[33] "Next.js by Vercel," Next.js. Available: https://nextjs.org/ [Accessed: Mar. 17, 2021].

[34] "What is Agile?," Atlassian. Available: https://www.atlassian.com/agile [Accessed: Mar. 11, 2021].

[35] A. Srivastava, S. Bhardwaj and S. Saraswat, "SCRUM model for agile methodology," International Conference on Computing, Communication and Automation, pp. 864-869, May 5-6, 2017, Greater Noida, India. doi: 10.1109/CCAA.2017.8229928.

[36] "What Is a Sprint in Agile?," Wrike. Available: https://www.wrike.com/project-management-guide/faq/what-is-a-sprint-in-agile/ [Accessed: Mar. 11, 2021].

[37] M. James and L. Walter, "Scrum Reference Card," Scrum Reference Card, 2017. Available: https://scrumreferencecard.com/scrum-reference-card/ [Accessed: Mar. 11, 2021].

[38] K. Finley, "What Exactly Is GitHub Anyway?," TechCrunch, Jul. 14, 2012. Available: https://techcrunch.com/2012/07/14/what-exactly-is-github-anyway/ [Accessed: Mar. 11, 2021].

[39] C. Wanstrath, "A whole new GitHub Universe: announcing new tools, forums, and features," GitHub Blog, Sept. 14, 2016. Available: https://github.blog/2016-09-14-a-whole-

new-github-universe-announcing-new-tools-forums-and-features/ [Accessed: Mar. 11, 2021].

[40] "Edge Dictionaries," Fastly Developer Hub. Available: https://developer.fastly.com/reference/api/dictionaries/ [Accessed: Mar. 18, 2021].

[41] "Visual Studio Code," Visual Studio Code. Available: https://code.visualstudio.com/ [Accessed: Mar. 11, 2021].

[42] A. Silver, "Introducing Visual Studio Live Share," Visual Studio Code, Nov. 15, 2017. Available: https://code.visualstudio.com/blogs/2017/11/15/live-share [Accessed: Mar. 11, 2021].

[43] Fastly, "About Edge Dictionaries," Fastly Help Guides, Jan. 18, 2019. Available: https://docs.fastly.com/en/guides/about-edge-dictionaries [Accessed: Mar. 15, 2021].

[44] K. Rogovoy, "Here are some amazing advantages of Go that you don’t hear much about," freeCodeCamp, Feb. 1, 2018. Available: https://www.freecodecamp.org/news/here-are-some-amazing-advantages-of-go-that-you-dont-hear-much-about-1af99de3b23a/ [Accessed: Mar. 18, 2021].

[45] J. Day and H. Zimmermann, "The OSI reference model," Proceedings of the IEEE, volume 71, issue 12, pp. 1334-1340, Dec. 1983. doi: 10.1109/PROC.1983.12775.

[46] K. Stemmler, "The Dependency Rule," Khalilstemmler, Jul. 30, 2019. Available: https://khalilstemmler.com/wiki/dependency-rule/ [Accessed: Mar. 16, 2021].

[47] S. Eskildsen, "Logrus," GitHub, 2014. Available: https://github.com/sirupsen/logrus [Accessed: Mar. 17, 2021].

[48] J. Moiron, "sqlx," GitHub, 2013. Available: https://github.com/jmoiron/sqlx [Accessed: Mar. 17, 2021].

[49] L. Halliday, "Unleash the Power of Storing JSON in Postgres," CloudBees, Jul. 9, 2015. Available: https://www.cloudbees.com/blog/unleash-the-power-of-storing-json-in-postgres/ [Accessed: Mar. 16, 2021].

[50] Fastly, "Go Fastly," GitHub. [Online]. Available: https://github.com/fastly/go-fastly [Accessed: Mar. 16, 2021].

[51] The Gorilla Authors, "gorilla/mux," GitHub. Available: https://github.com/gorilla/mux [Accessed: Mar. 17, 2021].

[52] The Go Authors, "Package ioutil," Golang, 2009. Available: https://golang.org/pkg/io/ioutil/ [Accessed: Mar. 17, 2021].

[53] "Prototyping," Sketch, Nov. 24, 2020. Available: https://www.sketch.com/docs/prototyping/ [Accessed: Mar. 18, 2021].

[54] E. Muller, “Introducing Compute@Edge Log Tailing for better observability and easier debugging,” Fastly, Feb. 8, 2021. Available: https://www.fastly.com/blog/introducing-compute-edge-log-tailing-for-better-observability-and-easier-debugging/ [Accessed: Mar. 18 2021].