MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

User interface for BricksLLM proxy

Master's Thesis

DANIEL OLEARČIN

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

User interface for BricksLLM proxy

Master's Thesis

DANIEL OLEARČIN

Advisor: RNDr. Lukáš Hejtmánek, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2025

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during the elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools: Grammarly for grammar checks, GitHub copilot for code writing, and ChatGPT to improve my writing style. I declare that I used these tools by the principles of academic integrity I verified the contest against credible sources, and I take full responsibility for it.

Daniel Olearčin

Advisor: RNDr. Lukáš Hejtmánek, Ph.D.

Acknowledgements

I would like to thank my advisor, RNDr. Lukáš Hejmánek, Ph.D., for his constant guidance, insightful advice, and support throughout the work on this thesis.

Abstract

This thesis explores the design and development of a user-friendly web interface for BricksLLM's cloud-native AI gateway, which acts as a proxy for popular large language model APIs. BricksLLM makes it easy to securely resell and monitor the usage of these APIs, providing features such as usage tracking and access control.

The main focus is to understand how BricksLLM works under the hood, especially how it works with PostgreSQL and Redis databases, and use this knowledge to build a complete, easy-to-use web interface. As part of this project, BricksLLM has been integrated into the JupyterHub service at the CERIT-SC center, allowing users to access LLM features directly from their Jupyter notebooks.

The new web interface allows users to login with OIDC, generate their own BricksLLM API key, view their usage statistics, and ensures that each user has only one API key. Administrators can monitor user activity and API usage. Everything runs in a Kubernetes environment, managed with a Helm chart for easy deployment.

Keywords

Artificial intelligence, BricksLLM, Go, HTMX, Templ, OpenAI, Web Application, User Interface, Development, Deployment

Contents

Introduction **1**

1 Theoretical part **3**
1.1 Web Development Technologies 3
1.2 BricksLLM Component 8
1.3 Kubernetes 10
1.4 Helm 14

2 Design **16**
2.1 Application Architecture 16
2.2 Authentication and Authorization 17
2.3 User Interface 18
2.4 Application Functionality 21

3 Implementation **24**
3.1 Challenges 25
3.2 Development Tools 26
3.3 Jupyter Hub Integration 29
3.4 Application Management 31
3.5 API Endpoints 33
3.6 Authentication Service 35
3.7 Session Service 36
3.8 BricksLLM Service 36
3.9 User Interface 42

4 Deployment **48**
4.1 Deployment in Kubernetes 48
4.2 Helm Chart 50

5 Conclusion **51**

A Contents of the Archive **53**

Bibliography **54**

List of Tables

3.1 API Routes 34

List of Figures

2.1 User interface design 18
2.2 Application functionality from user perspective. 21
3.1 Core services of the application and their functions. . . . 24
3.2 Navigation bar 43
3.3 Home page 43
3.4 OpenAI page 44
3.5 Token page 45
3.6 User token detailed information 46
3.7 Generic error page 47

Introduction

In recent years, artificial intelligence has experienced a surge in popularity and adoption across industries [1]. The rapid development of large-scale language models, such as those provided by OpenAI, has created new opportunities for both research and commercial applications. This growing relevance of artificial intelligence technologies inspired me to delve deeper into the field and, as part of this thesis, to design and implement practical application related to artificial intelligence.

This thesis focuses on designing and implementing a user interface for the BricksLLM component. Component acts as a proxy for various AI vendors and enabling their resale, while providing essential features such as resource consumption accounting and usage limitation. As BricksLLM itself does not include a front-end for user interaction, this interface has been developed to bridge the gap, making the system accessible and manageable for end users and administrators.
The user interface must accomplish several key objectives:

- **User Authentication:** Support secure user authentication via OpenID Connect, ensuring that only authorized users can access the system.
- **API Key Management:** Allow users to create their API tokens for BricksLLM, empowering them to interact with the OpenAI.
- **Usage Monitoring:** Provide users with clear visibility into their usage statistics, helping them track resource consumption and manage their API usage effectively.
- **Time-Limited Usage Enforcement:** Ensure that each user is subject to strict time-based limits on using the OpenAI API, regardless of how many API keys they generate. This prevents circumvention of usage restrictions through multiple key creation.

- **Administrative Oversight:** Enable system administrators to view detailed user information, particularly focusing on API usage statistics, to facilitate monitoring and management of system resources.
- **Deployment Requirements:** The entire system, including the user interface and BricksLLM component, must be deployable within a Kubernetes environment using a Helm chart package, ensuring scalability, portability, and ease of management.

The thesis is divided into five chapters:

- The first chapter presents the theoretical background, summarizing the knowledge and technologies used during the development of the thesis.
- The second chapter details the design of the application, including architectural decisions and user interface considerations.
- The third chapter describes the implementation process, outlining the development steps and technical challenges encountered.
- The fourth chapter focuses on the deployment of the application, covering the steps necessary to get the solution up and running.
- The fifth and final chapter provides a conclusion, reflecting on the results and possible future improvements.

1 Theoretical part

This chapter overviews the basic technologies and concepts for developing modern web applications. It provides detailed information about technologies such as the Go programming language, the HTMX library, the Templ templating language, and the Echo server, as well as BricksLLM, which serves as an artificial intelligence gateway for deploying large language models in production. In addition, an overview of Kubernetes and Helm is included, highlighting their role in containerizing and managing applications in the cloud. This theoretical foundation sets the stage for the practical implementation discussed in later chapters.

1.1 Web Development Technologies

Web development technologies include various tools and frameworks for creating and maintaining web applications. They include basic technologies such as HTML for structuring content, CSS for styling, and JavaScript for interactivity. Server-side languages such as Go, Python, and Ruby, and frameworks such as Echo and Rails handle back-end logic. Front-end frameworks like React, Angular, and Vue.js enable the creation of dynamic user interfaces. Database systems, API integrations, and cloud services further extend the functionality and scalability of web applications.

Go

As Kolade Chris wrote [2], Golang is an open-source, compiled, statically typed programming language developed by Google. It is designed to be simple, powerful, readable, and efficient. The programming language is officially called Go, but many refer to it as Golang because of its original domain name, `golang.org`. This alternative name improves search-ability on the Internet, although Go remains the correct name.

Go was developed by Robert Griesemer, Rob Pike, and Ken Thompson at Google in 2007 to address the growing complexity of the company's code bases and their shared dissatisfaction with the C++ lan-

guage. The language was publicly announced in 2009 and became open source in 2012 with version 1.0. Known for its simplicity, readability, efficiency, and support for concurrency, Go quickly gained popularity. Its widespread use spans areas such as back-end development, cloud computing, game development, and command-line tools. Prominent technology companies such as Google, Netflix, Kubernetes, and Docker have widely adopted Go, underscoring its importance in the field.

As stated in the Go documentation [3], the Go programming language is characterized by its simplicity, ease of learning, and suitability for team projects. It is built with concurrency and a robust standard library, supported by an extensive ecosystem. The language's strengths lie in cloud and network services, command line interfaces, web development, DevOps, and site reliability engineering. It emphasizes developer happiness through fast compilation and reduced boilerplate code and provides excellent tools. Its characteristics make it ideal for microservices, thanks to its small memory footprint and ease of containerization. Many developers who have used Go prefer not to return to other languages.

Go was designed with several key principles in mind that distinguish it from other programming languages:

- **Performance and Efficiency:** Go achieves high performance by compiling to machine code rather than using an intermediate virtual machine. This approach provides performance that is reasonably close to that of the C programming language for a garbage-collected language.

- **Simplicity and Readability:** Go was designed with simplicity as a core principle. The language incorporates elements of the C family into its syntax, with declarations and packages inspired by Pascal. In particular, the Go language is remarkably compact, with only 25 keywords, making it significantly smaller than most popular programming languages.

- **Concurrency Model:** A distinguishing feature of Go is its approach to concurrency, which differs from that of other languages that rely on traditional threading models. In contrast, Go implements communicating sequential processes. This implementation is realized through goroutines and channels.

HTMX

As stated in the HTMX documentation [4], HTMX is an open-source front-end JavaScript library that extends HTML with custom attributes, allowing developers to implement dynamic web functionality directly through HTML markup rather than writing extensive JavaScript code. Created by Carson Gross, HTMX was released on November 24, 2020, as an evolution of his earlier project intercooler.js [5].

The goal of HTMX is to complete HTML as hypertext by allowing HTML elements to make HTTP requests, process responses, and update the Document Object Model directly through declarative attributes. This approach introduces a fundamental shift in the way developers can build interactive web applications.

As noted in the Refine blog [6], HTMX significantly reduces the complexity typically associated with developing dynamic Web applications:

- **Reduced Learning Curve:** Developers work with HTML attributes rather than learning complex frameworks.

- **Improved Performance:** Without the overhead of heavy frameworks, pages load faster and perform more efficiently.

- **Enhanced Maintainability:** The declarative nature of HTMX results in more readable, self-documenting code that is easier to maintain.

- **Server Side Focus:** By shifting more logic to the server side, HTMX aligns with traditional Web development patterns while still delivering modern user experiences.

Templ

As stated in Templ documentation [7], Templ is an HTML templating language that allows developers to create components that render fragments of HTML and assemble them into complete screens, pages, documents, or applications. It was created by Adrian Hesketh and has gained considerable popularity among Go developers looking for efficient templating solutions.

Templ offers several key features that make it stand out as a templating solution:

- **Server Side Rendering:** Templ provides robust server-side rendering capabilities, allowing it to be deployed as serverless functions, Docker containers, or standard Go programs. This flexibility makes it suitable for various deployment scenarios.

- **Static Rendering:** The framework supports static rendering, allowing developers to create static HTML files that can be deployed through any preferred method. This feature is particularly useful for content-heavy sites or when optimizing for performance.

- **Compiled Code:** Components created with Templ are compiled into high-performance Go code. The compilation process transforms the templating syntax into optimized Go, eliminating runtime interpretation overhead.

- **Native Go Integration:** One of the strongest benefits of Templ is its seamless integration with Go. Developers can call any Go code and use standard control structures such as `if`, `switch`, and `for` statements directly within Templ. This allows powerful logic to be implemented without leaving the template context.

- **No JavaScript required:** Unlike many modern templating solutions, Templ does not require any client-side or server-side JavaScript to function. This results in lightweight, efficient templates that work well in environments with limited JavaScript support.

- **Developer Experience:** Templ ships with integrated development environment auto-completion support, providing an excellent developer experience. Official IDE support is available for Visual Studio Code, Helix, and Emacs.

Echo

As stated in Echo documentation [8], Echo is a high-performance, extensible, and minimalist Go web framework for building scalable web applications and RESTful APIs. It emphasizes speed and ease of use with an optimized HTTP router, middle-ware support, data binding, and flexible rendering capabilities.

Key uses of Echo include::

- **Building RESTful APIs:** Echo simplifies the process of creating scalable and well-organized API endpoints.
- **Web Application Development:** It provides features such as template rendering and data binding, making it suitable for full-featured web applications.
- **High-Performance Applications:** Echo's optimized HTTP router operates without dynamic memory allocation, ensuring efficient request handling.
- **Secure Applications:** With automatic TLS support, Echo streamlines the process of implementing secure communications.
- **middle-ware Integration:** Echo provides extensive middle-ware support for implementing common concerns such as logging, authentication, and error handling.
- **HTTP/2 Applications:** Echo supports HTTP/2, enabling developers to build faster, more responsive Web applications.
- **Custom Web Servers:** Developers can use Echo to run applications using either the standard HTTP server or the FastHTTP server.
- **API Grouping:** Echo allows for the logical grouping of APIs, which is particularly useful for managing complex applications.

- **Template-based applications:** Echo supports template rendering with any template engine, providing flexibility in creating dynamic content.
- **Extensible Projects:** Echo's design allows for easy extensibility through custom middle-ware, plugins, and components.

Echo's combination of performance, simplicity, and extensibility makes it a popular choice among Go developers for a wide range of web development projects.

1.2 BricksLLM Component

As stated in the BricksLLM GitHub repository [9], BricksLLM is a cloud-native AI gateway designed to provide an enterprise-level infrastructure for production LLM use cases. Written in Go, this open-source tool streamlines the management, monitoring, and control of large language model interactions, providing a robust solution for organizations deploying large language model-based applications.

Initially, the system focused on providing rate limiting for OpenAI spending via API keys. However, as the project evolved, the BricksLLM team identified performance bottlenecks, particularly with increasing query and response lengths. This challenge led to a significant architectural update, moving to an event-driven approach where HTTP requests generate events that are processed asynchronously, improving scalability and performance.

BricksLLM is built on a lightweight but powerful technical foundation. At its core is a Go web server coupled with PostgreSQL for database management and Redis for caching. This architecture was chosen specifically for the performance, type safety, and robust error-handling capabilities inherent in the Go language.

BricksLLM offers several practical use cases for organizations implementing large-scale language model-based solutions.

Implementing Tiered Pricing

BricksLLM allows organizations to create custom API keys with specific usage limits, enabling tiered pricing models. For example, a company might offer:

- Basic Tier: 10,000 requests/month with a $100 spend limit.
- Pro Tier: 50,000 inquiries/month with a $500 spend limit.
- Enterprise Tier: Custom limits based on client needs.

This flexibility helps capture more value from power users while providing affordable options for smaller customers.

Usage Tracking

The tool provides detailed usage metrics for each API key, including model usage and custom user IDs. This granular data allows organizations to:

- Analyze individual customer behavior.
- Identify high-value users.
- Optimize pricing strategies based on actual usage patterns.

Cost Control

By setting spending limits for API keys, companies can prevent unexpected expenses and ensure profitability for each pricing tier. This is especially valuable for startups with limited budgets for AI experimentation.

Access Control

BricksLLM provides fine-grained access control not provided natively by OpenAI. This allows enterprises to:

- Create role-based access for different teams (e.g. development, production).
- Implement temporary API keys for contractors or temporary projects.
- Enforce tighter controls on access to sensitive data.

Controlled Distribution

BricksLLM allows organizations to manage API keys with appropriate restrictions for different environments. This allows for:

- Separate keys for development, staging, and production environments.
- Limited access keys for educational or demo purposes.
- Gradual rollout of new features by controlling access at the API key level.

By addressing these critical challenges, BricksLLM provides a comprehensive solution for organizations looking to efficiently deploy, manage, and monetize large language model-based applications.

1.3 Kubernetes

As stated in the documentation [10], Kubernetes is a portable, extensible, open-source platform that facilitates both declarative configuration and automation for containerized workloads and services. Developed by Google and open-sourced in 2014, Kubernetes is built on Google's 15 years of experience running production workloads at scale, combined with the best ideas and practices from the community. The name Kubernetes comes from Greek, meaning helmsman or pilot, with K8s as an abbreviation derived from counting the eight letters between the K and s.

In production environments, managing containers while ensuring continuous availability is a significant challenge. Kubernetes addresses these challenges by providing a framework for running distributed systems in a resilient manner, handling application scaling and failover, and offering sophisticated deployment patterns. Its widespread adoption in industries such as robotics, healthcare, retail, education, gaming, and financial services demonstrates its versatility and effectiveness.

Architecture

As stated in the Kubernetes documentation [10], Understanding Kubernetes architecture is essential for effectively implementing and managing the platform. A Kubernetes cluster consists of a control plane and one or more worker nodes, each with specific components and responsibilities.

Control Plane Components

The control plane manages the overall state of the cluster and makes global decisions:

- **Kube API server:** The core component server exposes the Kubernetes HTTP API and serves as the front-end for the Kubernetes control plane.
- **ETCD:** A consistent and highly available key-value store that maintains all API server data, serving as the primary database for Kubernetes.
- **Kube Scheduler:** Examines newly created Pods that have not yet been assigned to nodes and selects nodes on which to run them based on resource requirements, constraints, and other factors.
- **Kube Controller Manager:** Runs controller processes that regulate the state of the cluster, handle node failures, maintain correct pod counts, and create default accounts and API access tokens.

- **Cloud Controller Manager (optional):** Embeds cloud-specific control logic, allowing Kubernetes to integrate with underlying cloud providers.

Node Components

Node components run on every node in the cluster, maintaining running Pods and providing the Kubernetes runtime environment:

- **Kubelet:** The primary agent that ensures containers are running in a Pod, taking a set of PodSpecs and ensuring that the described containers are running and healthy.
- **Kube Proxy:** Maintains network rules on nodes that implement services, allowing network communication to Pods from inside or outside the cluster.
- **Container Runtime:** The software responsible for running containers, such as Docker or containerd.

Additional components may be required on each node, such as systemd on Linux nodes to oversee local components.

Addons

Addons extend the functionality of Kubernetes with resources that implement clustering features:

- **DNS:** Provides cluster-wide DNS resolution, essential for service discovery.
- **Web User Interface (Dashboard):** Provides a graphical interface for cluster management.
- **Container Resource Monitoring:** Collects and stores container metrics.
- **Cluster Level Logging:** Stores container logs in a central log store.

Core Concepts

As stated in the Kubernetes documentation [10], Kubernetes uses several abstractions to represent the state of the system, which form the basis of its functionality and operations.

Pods

Pods are the smallest deployable units in Kubernetes, representing a single instance of a running process in the cluster. They can contain one or more containers that share storage, network resources, and a specification for how to run the containers. Understanding pods is fundamental to working effectively with Kubernetes.

Workloads

Workloads are applications that run on Kubernetes. The platform provides several built-in workload resources such as deployments, stateful sets, daemon sets, and jobs to manage different types of applications. These abstractions help manage the life-cycle of containerized applications at scale.

Services, Load Balancing, and Networking

Services define a logical set of Pods and a policy for accessing them. This abstraction allows for loose coupling between dependent Pods and manages networking and communication between them. Kubernetes networking concepts enable complex microservice architectures while maintaining simplicity of implementation.

Storage and Configuration

Kubernetes provides several mechanisms for both long-term and temporary storage for Pods in the cluster. ConfigMaps and Secrets provide ways to configure applications without rebuilding container images, increasing flexibility and security.

Kubernetes has established itself as the industry standard for container orchestration, providing powerful capabilities for deploying, scaling, and managing containerized applications. Its rich feature set,

extensible architecture, and strong community support continue to drive innovation in cloud-native application development and deployment.

1.4 Helm

As stated in Helm documentation [11], Helm was created to address the challenges of managing complex Kubernetes applications. It acts as a specialized package manager that bundles all the necessary Kubernetes resources into reusable packages, called charts. These charts are used to define, install, and update even the most complex Kubernetes applications. Originally started as a joint project between Google and Deis (later acquired by Microsoft), Helm has evolved into a graduate project of the Cloud Native Computing Foundation.

Helm's core functionality revolves around three primary concepts: charts (packages containing Kubernetes resources), repositories (collections of charts), and releases (instances of charts running in a cluster). This architecture allows Helm to streamline application deployment while providing robust version control and configuration management capabilities.

Helm works by defining and managing charts, which serve as templates or blueprints for Kubernetes resources. When a chart is deployed, Helm communicates directly with the Kubernetes API to create and manage the defined resources within the cluster. This templating approach allows users to customize deployments for different environments without rewriting complex YAML files, significantly reducing deployment complexity and potential errors.

Helm follows a modular architecture, built around two primary components: the Helm client and the Helm library. This design allows for streamlined interaction between users and Kubernetes clusters.

The Helm Client

The Helm Client serves as the end-user command line interface and is the primary point of interaction for all Helm operations.

Client performs many critical functions:

- Local diagram development and management.
- Repository management (adding, updating, listing).
- Release life-cycle management.
- Interface to the Helm library.
- Diagram installations and upgrades.
- Release rollbacks.

This client-centric design provides users with a consistent and intuitive interface for managing the entire application life-cycle on Kubernetes.

The Helm Library

The Helm Library contains the core logic that powers Helm's functionality. It performs several essential tasks:

- **Template rendering:** Process diagram templates using provided values to generate Kubernetes manifests.
- **Release management:** Tracking the state of each deployment to enable upgrades and rollbacks.
- **Kubernetes API Interaction:** Communicate with the Kubernetes API server to provision and manage resources.

This library based approach allows Helm to maintain a lightweight client, while still providing robust functionality for managing complex Kubernetes deployments.

2 Design

This chapter provides a comprehensive overview of the application architecture, user interface design, security mechanisms, and user features design. The Application Architecture section outlines the overall structure, including layers and relationships. The User Interface section details design principles focusing on usability, accessibility, and user experience, including layout, navigation flow, and visual elements for effective interaction. The Authentication and Authorization section addresses security mechanisms for verifying user identities and controlling access to system resources based on roles and permissions. The Application Functionality section details specific features designed to meet user needs and requirements.

2.1 Application Architecture

The application architecture is built around Go as the primary programming language, using the Echo framework for server-side operations, complemented by a modern front-end stack of HTMX, Templ, and Tailwind CSS. This combination creates a powerful yet lightweight architecture that allows for server-side rendering while maintaining an interactive user experience. The project follows a typical microservices-oriented structure.

App Layer

The application layer manages application configuration, startup, and shutdown procedures. It handles environment variables propagation and graceful service termination.

API Layer

This layer defines all HTTP routes, middle-wares, and controllers exposed by the Echo server. It handles request validation and authentication and delegates business operations to the core layer.

Core Layer

The core layer contains the core business logic of the application, including service implementations, client implementations, domain models, and data access layers.

Static Layer

Contains compiled Tailwind CSS, JavaScript utilities, and other static assets served by the application.

View Layer

The view layer contains implementation of front-end pages.

Cmd Layer

The entry point for the application, contains the main.go file that bootstraps the entire system.

2.2 Authentication and Authorization

Application authentication is handled by the OpenID Connect protocol, which extends OAuth 2.0 to provide user authentication and single sign-on functionality, while OAuth 2.0 handles API security via scoped access tokens.

The application requests the following OpenID Connect scopes:

- **openid:** Required for OpenID Connect authentication, returns the sub-claim that uniquely identifies the user.
- **profile:** Provides access to the user's profile information.
- **email:** Returns the users email address.
- **eduperson_entitlement:** A custom scope that contains the role-based access control information for the user.

The application uses profile and email scopes to associate users with their authentication tokens and display personalized information in the user interface.

Role-Based Access Control

The application uses role-based access control to efficiently and securely manage user permissions. The model assigns permissions to users based on their roles within the organization, providing a manageable approach to access control.

2.3 User Interface

The user interface follows a clean, modern design paradigm with a focus on simplicity and functionality. The system uses a minimalist aesthetic with ample white space, clear typography, and intuitive navigation to ensure that users can efficiently interact with the application.

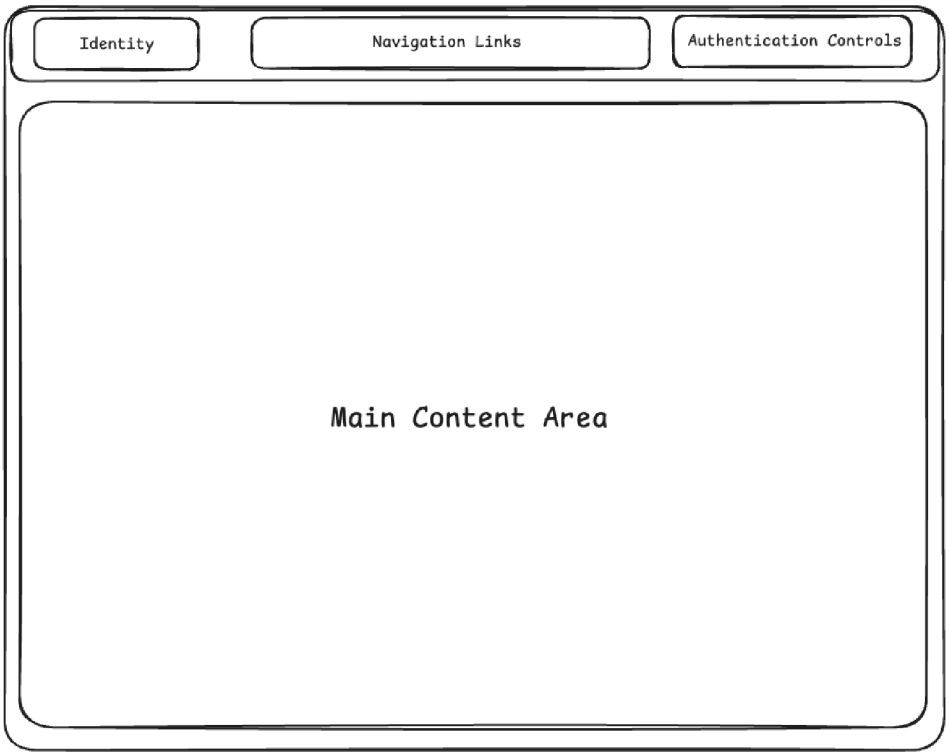

Figure 2.1: User interface design

The diagram 2.1 shows the user interface design, with components described in section 2.3.

Navigation Bar

Located at the top of the interface, the navigation bar contains the following elements:

- **Identity:** The Bricks LLM Dashboard name appears on the left side to establish website recognition.
- **Navigation Links:** Central area contains primary navigation tabs including Token, OpenAI, and Management. Management is only visible to Admin users.
- **Authentication Controls:** A Sign In button is located on the far right, providing clear access to the authentication functionality. After logging in, the button changes to Logout with a welcome message.

The navigation bar uses a subtle border to separate it from the main content area while maintaining visual cohesion with the rest of the interface.

Main Content Area

The main content area uses a responsive layout that adapts to different screen sizes and contains different content for each page.

Home Page

The page presents three prominent feature cards arranged horizontally:

- **Login Card:** Displays a login icon, Sign In heading, and descriptive text about secure authentication.
- **Create Token Card:** Displays a document icon, the heading Create Token, and an explanation of the token generation functionality.

- **Call OpenAI Card:** Displays a checkmark icon, the heading Call OpenAI, and a description of the API call functionality.

Token Page

Contains the components:

- **Token Display:** Displays the currently active token.
- **Token Statistics:** Displays usage metrics in a tabular format with columns for stat name, actual usage, limits, and units.

OpenAI Page

Contains the components:

- **Header:** OpenAI Chat title.
- **Request Label:** Text indicating where to enter requests.
- **Text Inputs:** Text boxes for entering requests and getting the response.
- **Send Button:** Action button for submitting requests.
- **Response Label:** Text indicating where response is shown.

Management Page

Contains the component:

- **User Tokens Grid:** Displays multiple tokens in a responsive grid layout, each with its unique identifier and a View Details action button.

Responsive Behavior

The interface is designed to fluidly adjust to different viewport sizes:

- On larger screens, content is arranged in multi-column layouts.
- On smaller screens, elements stack vertically to maintain readability and usability.

- Typography scales proportionally to ensure legibility across devices.

2.4 Application Functionality

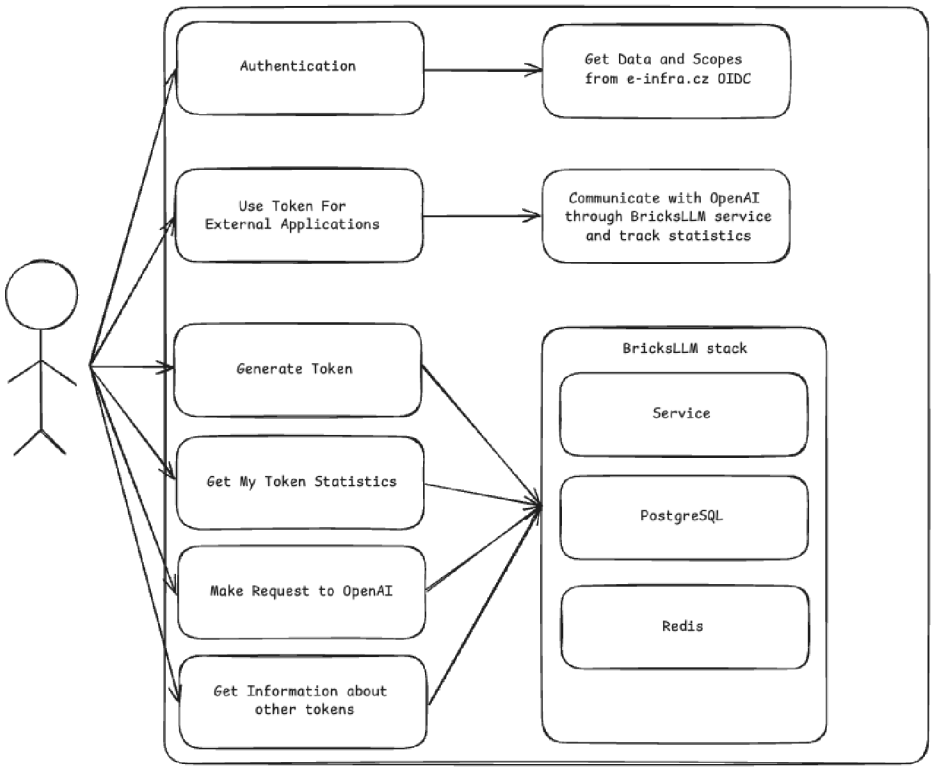

Figure 2.2: Application functionality from user perspective.

The diagram 2.2, shows the possible use cases from the users perspective. Each use case is described in the section 2.4.

The application provides a comprehensive interface for managing OpenAI API tokens and monitoring usage statistics. Users can create personal token, track detailed usage metrics, and interact directly with OpenAI services. Administrators have additional capabilities to oversee all tokens in the system.

All functionality within the application requires user authentication. This authentication requirement ensures proper access control and maintains the security of personal tokens and usage data.

Token Generation

Users can create their own token for OpenAI API communication, providing a personalized access method to OpenAI services. This functionality requires user authentication to ensure that token creation is user-specific.

Token Statistics

The application provides detailed monitoring of API usage with the following metrics:

- Request frequency over specified periods.
- Total cost of all requests.
- Cost breakdown for specified time intervals.

Each usage metric has an associated limit. When a user reaches the defined limit for any metric, the token is automatically unusable, preventing further API calls until the limit is reset or adjusted. This feature helps users control their API usage and avoid unexpected costs.

This tracking feature allows users to maintain awareness of their API usage and associated costs while providing safeguards against excessive usage.

OpenAI Communication

The application provides a streamlined interface for interacting with OpenAI services directly within the application environment. This eliminates the need for external tools or interfaces when working with OpenAI APIs.

Administrative Privileges

Users with administrative privileges have access to advanced features:

- View a list of all tokens in the system.
- Access detailed information about each token.

Using token in external applications

Users can use a generated token together with the BricksLLM component URL to make calls to OpenAI, allowing them to manage access, monitor usage, and set limits for different users or applications. Many applications support this configuration by letting you set a custom API base URL and token.

3 Implementation

This chapter covers the technical aspects of the application architecture and implementation. It covers problems that occurred, integration into JupyterHub, the development tools used to build and manage the project, the runtime management strategies employed, the handling of environment variables for configuration, the implementation of API endpoints, the authentication and session management, the communication with BricksLLM component, and the development of the user interface. Each section provides insight into the specific technologies and methodologies.

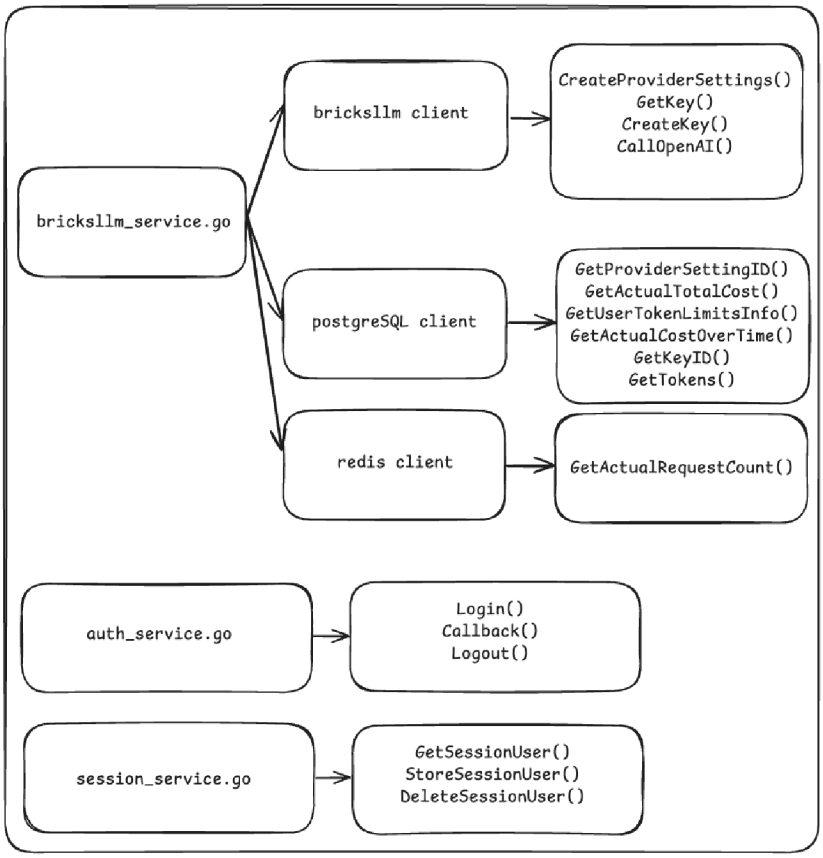

Figure 3.1: Core services of the application and their functions.

The Diagram 3.1, shows the core services of the application with their functions. The functionality of the services is explained in detail in the chapter 3.

3.1 Challenges

The development of this project presented a variety of technical and practical challenges that needed to be addressed to ensure efficient progress and a robust final product. Throughout the implementation phase, I encountered several obstacles related to the development workflow, integration with BricksLLM component, and front-end developement. In this section, I describe the major problems I encountered, along with the solutions and tools I used to overcome them.

Automatic reloading during development

During the development of the Go application, which includes both back-end and front-end components, I encountered a significant challenge: the need to rebuild and manually restart the application after each code change. This manual process would have dramatically increased development time and hindered productivity. To solve this problem, I looked for ways to automate the reload process. I discovered the Air tool for Go, which enabled automatic rebuilding and reloading of the application whenever changes were detected in the source code. This greatly streamlined the development workflow. More details about the Air tool can be found in the section 3.2.

Limitations of the BricksLLM component API endpoints

Initially, I assumed that all the information needed for this thesis project could be obtained from the exposed API endpoints of the BricksLLM component. However, as I delved deeper into the structure and architecture of BricksLLM, I realized that only a limited number of API endpoints were useful for the project. To access the rest of the required data, I had to familiarize myself with the BricksLLM PostgreSQL database tables and Redis instance, and then develop separate clients for PostgreSQL, Redis, and the API. More information about these clients can be found in the section 3.8.

Front-end Development Challenges

Developing an intuitive and user-friendly front-end within the Go application was another challenge, especially given my limited experience with front-end development outside of some exposure to the TypeScript framework. To overcome this, I researched modern approaches that would simplify the process and allow for a well-structured front-end. Ultimately, I adopted a combination of Tailwind-CSS, Templ, and HTMX that provided an efficient and straightforward way to implement a clean and usable front-end interface. More details on this approach are discussed in the 3.9 section.

3.2 Development Tools

When building this application, I used several development tools that greatly streamlined the process. These tools proved invaluable as I often needed to make changes to both the front-end and back-end components of the system simultaneously.

Being able to work on multiple aspects of the application at the same time not only accelerated the development timeline but also ensured consistency throughout the code base. By using these specialized tools, I was able to maintain a smooth workflow, quickly identify and resolve issues, and implement features more efficiently than would have been possible with a more traditional approach.

Docker Compose Setup

The development environment for this project was set up using Docker Compose, which orchestrated several services, including the BricksLLM component along with its Redis and PostgreSQL database dependencies. All services were configured with exposed ports to facilitate communication between components and provide local development access.

In the code block 3.1 is the Docker Compose configuration used for development. Orchestration was done with the command:

```
docker compose up
```

```
services:
  redis:
    container_name: redis
    image: redis:alpine3.20
    restart: always
    ports:
      - '6379:6379'
    command: redis-server --save 20 1 --loglevel
        warning
    volumes:
      - redis:/data
  postgresql:
    container_name: postgres
    image: postgres:12.20-alpine3.20
    restart: always
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_DATABASE=postgres
      - POSTGRES_PASSWORD=postgres
    ports:
      - '5432:5432'
    volumes:
      - postgresql:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U
         postgres"]
      interval: 1s
      timeout: 1s
      retries: 5
  bricksllm:
    container_name: bricksllm
    depends_on:
      postgresql:
        condition: service_healthy
      redis:
        condition: service_started
    image: luyuanxin1995/bricksllm
```

```
    environment:
      - POSTGRESQL_USERNAME=postgres
      - POSTGRESQL_PASSWORD=postgres
      - POSTGRESQL_HOSTS=postgresql
      - REDIS_HOSTS=redis
    ports:
      - '8001:8001'
      - '8002:8002'
    command:
      - '-m=dev'
```

Listing 3.1: Docker compose configuration for development

Live Reload with Air

As stated in the GitHub repository [12], Air greatly improves the development experience for Go applications by eliminating the need to manually stop, rebuild, and restart the application after code changes. It works by monitoring file changes in the project directory and automatically triggering the build process when changes are detected. The tool supports customization through configuration files, allowing developers to specify which directories to watch, which file extensions to monitor, and which commands to run during the rebuild process.

In my specific configuration, Air was set up with commands that included:

- Process and minify Tailwind CSS files.
- Generate template files.

After these steps, Air would then build the Go application from the main entry point.

In the code block 3.2 is the air configuration used for development. Live Reload is started with the command:

```
air
```

```
bin = "./bin/main"
pre_cmd = ["tailwindcss -i ./tailwind.css -o ./
   static/tailwind.css --minify", "templ generate
   "]
cmd = "go build -o ./bin/main ./cmd/main.go"
delay = 20
exclude_dir = ["static", "node_modules"]
exclude_regex = [".*_templ.go"]
exclude_unchanged = false
follow_symlink = false
include_ext = ["go", "tpl", "tmpl", "templ", "
   html"]
kill_delay = "0s"
log = "build-errors.log"
send_interrupt = false
stop_on_error = true
```

Listing 3.2: Air setup for development

3.3 Jupyter Hub Integration

As part of the familiarization with the BricksLLM component, I have designed and implemented the integration of the BricksLLM component into the JupyterHub service at the CERIT-SC center. This integration enables users to utilize the OpenAI API directly within Jupyter Notebooks. The process consisted of several key steps, which are outlined below.

Create a Provider

The first step was to configure BricksLLM with the necessary OpenAI API credentials. This was accomplished by sending a PUT request to the BricksLLM component API, specifying the OpenAI provider and the associated API key. The response from this request contains an `ID` that is used in next step.

In the code block below is the command to send a PUT request:

```
curl -X PUT http://bricksllm:8001/api/provider-settings
    -H "Content-Type: application/json"
    -d '{
          "provider":"openai",
          "setting": {
             "apikey": "api-key"
          }
        }'
```

Create an API Key

Using the `ID` from the previous step, an API key is created. This is necessary for authenticating requests through the BricksLLM component. The key is associated with specific tags to manage user access:

```
curl -X PUT http://bricksllm:8001/api/key-management/keys
    -H "Content-Type: application/json"
    -d '{
         "name": "My Secret Key",
         "key": "my-secret-key",
         "tags": ["team"],
         "settingIds": [ID]
        }'
```

User Management and Proxy Usage in Jupyter Notebook

Within the Jupyter Notebook, the integration workflow proceeds as follows:

- **User Retrieval or Creation:** The notebook first checks whether a user with a specific name already exists. If the user exists, their unique token is retrieved. If not, a new user is created with defined usage limits, such as cost and rate limits, allowed API paths, and permitted models.
- **Authentication and Request Handling:** Once the user is identified or created, the notebook prepares a request to the OpenAI API endpoint but routes it through the BricksLLM component.

The request includes the API key for authentication and the user token for identification and usage tracking.

- **Response Processing:** The response from the OpenAI API is then processed and displayed within the notebook. This allows users to interact with OpenAI models securely and by the access policies defined in BricksLLM component.

The used API endpoints of the BricksLLM component are:

- **PUT /api/provider-settings**: Registers the OpenAI provider with the required API key.
- **PUT /api/key-management/keys**: Creates a API key associated with the OpenAI provider for authentication.
- **GET /api/users**: Retrieves existing users with specific tags.
- **POST /api/users**: Creates a new user with defined access limits and permissions.
- **POST /api/providers/openai/v1/chat/completions**: Sends a request to the OpenAI API, authenticated with the generated key and user token.

Tokens created through the Jupyter Notebook are also compatible with the user interface. Any tokens generated through Jupyter Notebook will also be visible and accessible in the BricksLLM User Interface.

3.4 Application Management

The application management strategy focuses on ensuring robust runtime behavior and configuration handling. This approach centralizes critical operations such as application life-cycle management, environment variable handling, and service initialization to promote maintainability and reliability throughout the application life-cycle.

Gracious Shutdown Mechanism

Application life-cycle management is implemented through a dedicated `goroutine` that listens for shutdown signals. This `goroutine` waits for either system termination signals or specific shutdown messages sent through a designated channel. When such a signal is received, the application initiates a graceful shutdown sequence.

This approach ensures that

- Ongoing operations can be completed properly.
- Data integrity is maintained during shutdown.

By implementing this pattern, the application can handle unexpected termination requests while minimizing the risk of data corruption or resource leakage.

Environment Variables

Environment variables management is centralized using the Viper library. As stated in the GitHub repository [13] Viper provides a secure and flexible approach to configuration handling.

Centralized Service Initialization

The application's main initialization file serves as the orchestration point where all components are instantiated and configured. This file:

- Retrieves environment variables from Viper.
- Instantiates all services, clients, and database connections.
- Initializes the Echo server to handle HTTP requests.
- Starts the graceful shutdown `goroutine`.

This centralized approach to application setup creates a clear initialization flow and dependency injection pattern, making the code base more maintainable and easier to reason about. With a single point of initialization, the application startup sequence is well-defined and predictable, making it easier to debug and modify as needed.

3.5 API Endpoints

The implementation follows a controller-based pattern where related functionality is grouped and registered with the Echo web framework.

Endpoint Registration Architecture

API registration is implemented through a dedicated `APIRegister` structure that encapsulates the Echo server instance and core services. This structure provides a clean separation of concerns, where the routing logic is isolated from the actual request handling:

- The `NewAPIRegister` constructor initializes the register component.
- The `RegisterAPIEPs` function attaches all handlers to their respective routes.
- Static assets are served from a static directory.
- Controllers are instantiated with their required dependencies.

Endpoint Categories

The application's endpoints are organized into logical categories:

- **Basic Routes:** Includes the Health Check endpoint and the Home page.
- **Application:** Routes for interacting within application services.
- **Authentication:** Provider-based authentication flows, including login, callback, and logout.

Table 3.1: API Routes

HTTP Method	Route	Controller
GET	/health/check	HealthCheck
GET	/home	Home
GET	/token	Token
GET	/openai	OpenAI
GET	/management	Management
GET	/auth/:provider	Auth
GET	/auth/:provider/callback	Auth
GET	/auth/logout/:provider	Auth
GET	/bricksllm/createusertoken	BricksLLM
GET	/bricksllm/callopenai	BricksLLM

In the table 3.1 are displayed API routes implemented in the application.

Authentication middle-ware

The API routes implement authentication middle-ware to ensure proper access control:

- `RequireAuth()` ensures that the user is authenticated before accessing protected endpoints.
- `RequireManagementAccess()` provides an additional layer of authorization for management.

This structured approach to API endpoint registration creates a maintainable and scalable routing system. The controller-based architecture provides a clear separation between routing logic and business logic, while middle-ware based authentication ensures that access control is applied consistently throughout the application.

3.6 Authentication Service

Authentication is implemented using the Goth package. As stated in the GitHub repository [14] Goth package provides a unified interface for handling authentication with different providers supporting the OAuth2 and OIDC protocols. This implementation uses an OIDC (OpenID Connect) provider. The provider is instantiated by:

```
openidConnect.New(
openIDCfg.ID,
openIDCfg.Secret,
openIDCfg.CallbackURL,
openIDCfg.AutoDiscoveryURL,
strings.Split(openIDCfg.Scopes, ",")...
)
```

All configuration values are sourced from environment variables, providing flexibility and security in deployment.

The authentication workflow is encapsulated into three primary functions:

- **Login():** Initiates the authentication process by redirecting the user to the OIDC provider login page. This function constructs the appropriate authorization URL and handles the initial handshake.
- **Callback():** Handles the redirect from the OIDC provider after the user has authenticated. This function processes the authorization code, exchanges it for tokens, and retrieves user information. It is responsible for establishing the user session within the application.
- **Logout():** Manages the logout process by deleting the user session.

3.7 Session Service

Session management is implemented using the Gorilla sessions package [15]. Session service handles operations on the `CookieStore` and stores user information such as username, email, userID, and management access status after a successful login. The management access status is a boolean value that indicates whether the user can access the management page.

The service has three primary functions:

- StoreSessionUser(), RemoveSessionUser(), GetSessionUser()

These functions handle basic create, read, update, and delete operations on the user session. The session data is encapsulated in a structure called `SessionUser`, which contains fields for username, email, userID, provider (set to "openid-connect" in this context), and management access status. This structured approach ensures that all relevant user information is consistently stored and retrieved during session management.

To integrate the `CookieStore` with the session management system, the gothic package is used. The `CookieStore` is assigned as the underlying form of session storage, with the session name defined as `auth-session`. This approach allows for efficient session management while maintaining scalability and security by leveraging the features of the Gorilla sessions package, such as secure cookie settings and encryption capabilities.

3.8 BricksLLM Service

Communication with BricksLLM component is facilitated by its API endpoints and by querying the PostgreSQL database and Redis, which are integral components of the BricksLLM infrastructure. Effective communication with these components is critical to taking full advantage of BricksLLM's capabilities.

The BricksLLM service implements OpenAI access, token management, and usage tracking by integrating with PostgreSQL, Redis, and the BricksLLM client.

Below is an overview of the service functions and their roles:

- **GetProviderSettingID**() utilizes the PostgreSQL client to retrieve the unique provider settings ID associated with the OpenAI API key. This ensures that all operations reference the correct provider configuration.
- **CreateProviderSettings**() leverages the BricksLLM client to create provider settings for a given provider (in this case, OpenAI). It specifies which OpenAI models are allowed and stores the API key for OpenAI access.
- **GetUserToken**() uses the BricksLLM client to retrieve the token assigned to a user. This function ensures users have the credentials to interact with the LLM provider.
- **CreateUserToken**() firstly checks if a user already has an existing token if not, it generates a new token represented by a randomly generated 32-character string. During creation, it also sets various limits for the token, including cost limit in USD, cost limit in USD over time, rate limit over time, rate limit unit, and cost limit unit.
- **CallOpenAI**() uses the BricksLLM client to send a request to OpenAI, specifying the desired OpenAI model and handling the response.
- **GetActualRequestCount**() utilizes the Redis client to fetch the actual request count for a given token, supporting rate limiting and usage analytics.
- **GetActualTotalCost**() calls the PostgreSQL client to retrieve the total cost incurred by a user token, enabling cost tracking.
- **GetActualCostOverTime**() uses the PostgreSQL client to obtain cost data for a user token over a specified period, supporting detailed usage analytics and reporting.
- **GetUserTokenLimitsInfo**() queries the PostgreSQL client to fetch the usage limits associated with a users token.

- **GetTokens()** combines data from both the PostgreSQL client (for token information) and Redis (for associating actual request counts with tokens), providing a comprehensive view of all tokens under specific provider settings along with their current usage statistics.

BricksLLM Client

The BricksLLM client is designed to interact with the BricksLLM component API. It is responsible for managing connections to the BricksLLM, which exposes two ports: one for admin operations and another for proxy operations. Each port is associated with a Resty client instance that facilitates communication with the respective API endpoints.

The Resty package is utilized in the BricksLLM client to create and manage HTTP clients. The use of the Resty package simplifies the process of making HTTP requests to the BricksLLM API, allowing for efficient and structured communication.

Resty package is used for:

- **Client Creation:** Resty clients are created using the `resty.New()` function. Each client is then configured with a base URL that includes the port number for either admin or proxy operations.

- **Header Configuration:** The clients are set to use JSON for both content type and accept headers, ensuring that all interactions with the BricksLLM API are in JSON format.

- **API Calls:** Through these clients, the service can define HTTP functions, bodies, headers, and query parameters for API calls, providing a flexible way to interact with the BricksLLM API endpoints.

In the code block 3.3 is the initialization of Proxy Client and Admin Client to call BricksLLM component exposed API endpoints.

```
&clientImpl{
    adminClient: resty.New().
        SetBaseURL(bricksLLMCfg.BaseURL+":"+
           bricksLLMCfg.AdminPort).
        SetHeader("Content-Type", "application/
           json").
        SetHeader("Accept", "application/json"),
    proxyClient: resty.New().
        SetBaseURL(bricksLLMCfg.BaseURL+":"+
           bricksLLMCfg.ProxyPort).
        SetHeader("Content-Type", "application/
           json").
        SetHeader("Accept", "application/json"),
}
```

Listing 3.3: Initialization of Admin Client and Proxy Client

The BricksLLM client encapsulates the logic for interacting with the BricksLLM, ensuring that all necessary operations are performed efficiently and securely.

This approach ensures that the BricksLLM client can effectively manage and utilize the capabilities of the BricksLLM component while maintaining a clean and modular code structure.

BricksLLM client contains three primary functions:

- **CreateProviderSettings()** is called at the start of the application. Its purpose is to create the provider settings using the OpenAI API key (sourced from an environment variable), but only if such settings do not already exist. This ensures that the application is properly configured to communicate with the OpenAI API and avoids redundant re-creation of provider settings.

- **GetKey()** is responsible for retrieving an API token for a user. This function checks and returns the appropriate key that allows the user to interact with the LLM provider through BricksLLM component, enforcing any associated usage limits or permissions.

- **CreateKey**() handles the creation of a new API key for a user. This function associates the key with the relevant provider settings and can apply rate limits or cost restrictions, ensuring secure and controlled access for each user.

- **CallOpenAI**() is responsible for sending requests to the OpenAI API through BricksLLM component and handling the responses.

PostgreSQL Client

The PostgreSQL client is implemented using the GORM library. As stated in the documentation [16], GORM provides an object-relational mapping framework for interacting with the SQL databases. The client is created by connecting to the database using a connection string that includes the host, port, user name, password, and database name.

This client is used to perform operations that are not included in the BricksLLM service API endpoints. In particular, it retrieves data from the database to support front-end functionality.

In the code block 3.4 is the initialization of connection to the PostgreSQL database using the GORM library.

```
func NewClient(dbConfig config.DBConfig) Client
   {
     dsn := fmt.Sprintf("host=%s␣port=%d␣user=%s␣
        password=%s␣dbname=%s",
         dbConfig.Host, dbConfig.Port, dbConfig.
            User, dbConfig.Password, dbConfig.
            Database)
     db, err := gorm.Open(postgres.Open(dsn), &
        gorm.Config{})
     if err != nil {
         panic("failed␣to␣connect␣database")
     }
     return &clientImpl{db: db}
}
```

Listing 3.4: Setting up connection to PostgreSQL database

PostgreSQL client contains five primary functions with one supporting function:

- **GetProviderSettingID**() retrieves the identifier for provider settings that are already associated with the OpenAI API key. This function ensures that operations referencing provider settings use the correct record, preventing duplication and maintaining consistency.
- **GetActualTotalCost**() calculates the actual total cost incurred by a user token over a specified period. This is essential for tracking usage and providing transparency to users regarding their consumption.
- **GetUserTokenLimitsInfo**() fetches the usage limits associated with a user token.
- **GetActualCostOverTime**() returns the cost data for a user token within a set time range.
- **GetTokens**() retrieves all tokens that are present, and that are associated with provider settings. This function enables administrators to look over all tokens present and check usage.
- **GetKeyID**() is a supporting function that queries the database for the internal key identifier-based token. This approach improves query performance and reliability.

Redis client

The Redis client is implemented using the Go-redis package. As stated in the GitHub repository [17], Go-redis provides a convenient interface for interacting with Redis databases. The client is created by specifying connection options such as the Redis server address, password, and database index.

The BricksLLM component utilizes Redis for storing user requests count to OpenAI. This allows for efficient tracking and management of user requests.

In the code block 3.5 is the initialization of the Redis client used to query the Redis database.

```go
func NewClient(redisCfg config.RedisConfig)
   Client {
    client := redis.NewClient(&redis.Options{
        Addr:     redisCfg.URL,
        Password: redisCfg.Password,
        DB:       redisCfg.DatabaseIndex,
    })
    return &clientImpl{
        client: client,
        ctx:    context.Background(),
    }
}
```

Listing 3.5: Setting up connection to Redis

Redis client contains one function:

- **GetActualRequestCount**() retrieves the actual request count for the user token in a set period.

3.9 User Interface

The user interface is built using the Templ package for templating, Tailwind CSS for styling and HTMX library that extends HTML with custom attributes. This combination allows for a responsive and visually appealing design across all pages.

Base Page

The base page serves as the foundation for all other pages. It contains essential elements such as links to static resources like Tailwind CSS, `favicon.ico`, and `CDN.min.js`. It also defines the title, charset, and language of the document. The navigation bar is a key component of the base page, and its content is dynamically generated based on information from the user's session. This ensures that navigation remains consistent and personalized across all pages.

Figure 3.2: Navigation bar

The image 3.2 shows the navigation bar component. On the right is the application name, which acts as a link to the home page. In the middle are the navigation buttons for the different pages. On the left side is a personalized greeting with an option to log out or log in, if the user is not already logged in.

Home Page

The home page is designed to guide users toward specific actions. It features three prominent cards:

- **Sign In:** Redirects users to the login page.
- **Create Token:** Allows users to create a new token.
- **Call OpenAI:** Directs users to the OpenAI request submission page.

These cards provide a clear and intuitive starting point for users.

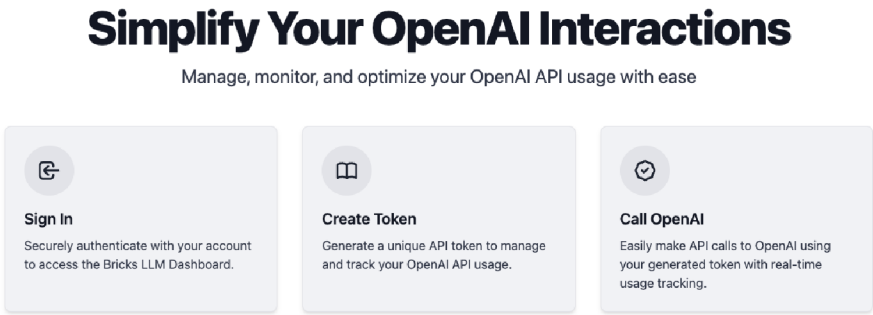

Figure 3.3: Home page

The image 3.3 shows the main content area of the Home page.

OpenAI Page

On the OpenAI page, users can submit requests to OpenAI using a text area. After sending a request, another `textarea` is displayed to show the response from OpenAI. This interactive feature allows users to engage with OpenAI models directly within the application.

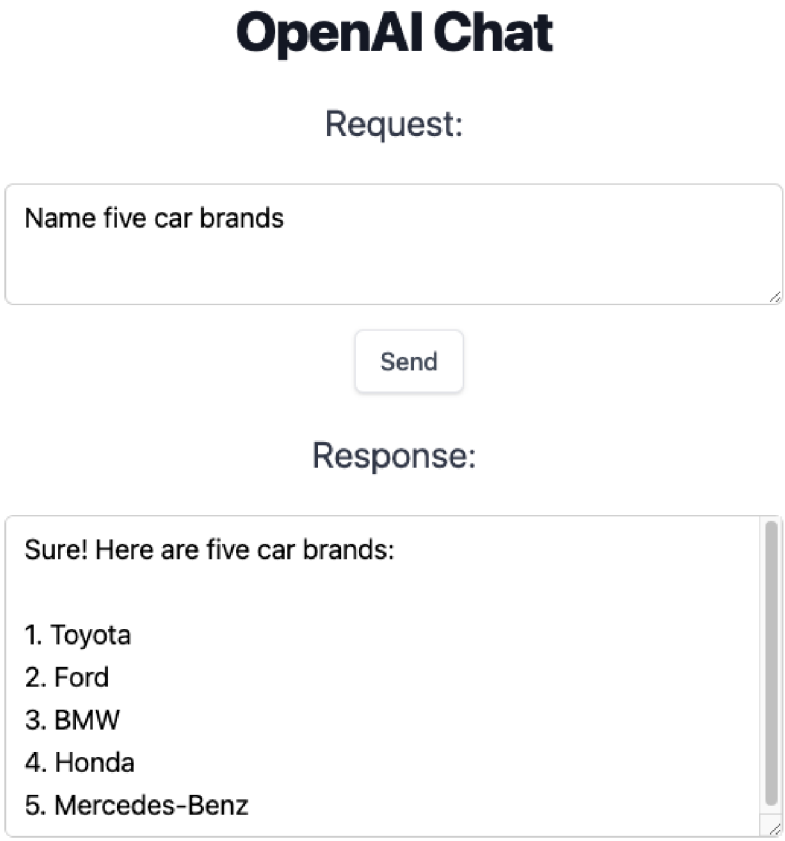

Figure 3.4: OpenAI page

The image 3.4 shows the main content area of the OpenAI page.

Token Page

The token page is dedicated to displaying information about the user token. It includes a table that lists the details of existing token. If a user has not created a token yet, a button is available to initiate this process. BricksLLM component collects usage statistics by tracking each request made to OpenAI, including the exact cost of that request as reported in the API response. This page provides users with a centralized view of their token.

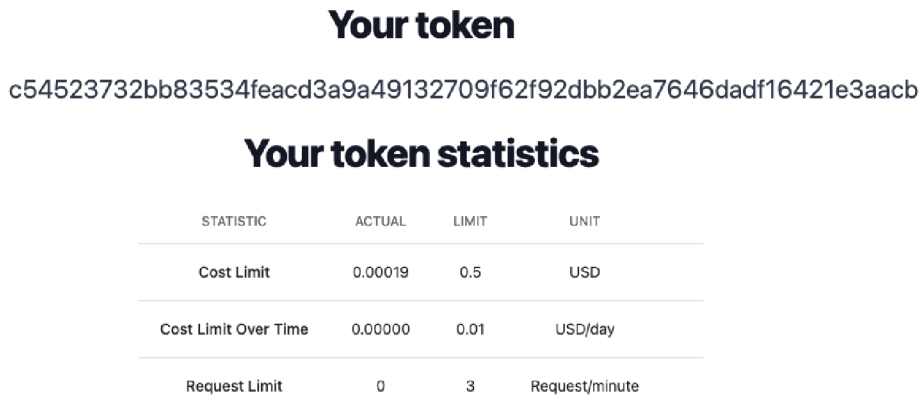

Figure 3.5: Token page

The image 3.5 shows the main content area of the Token page.

Management Page

The management page is designed for administrators and provides a comprehensive overview of all generated user tokens. It includes a table listing these tokens, with the option to open a modal window for each token to view detailed information. This functionality enables efficient management and monitoring of user tokens.

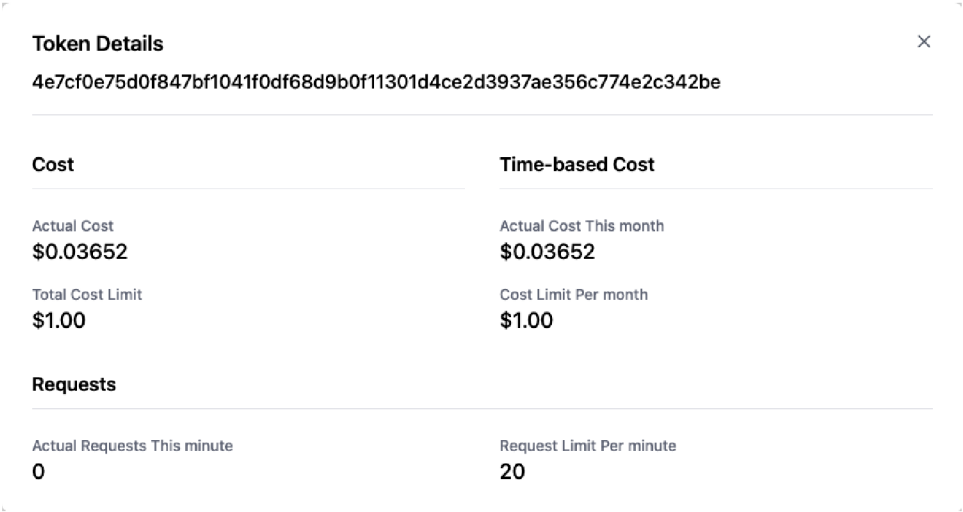

Figure 3.6: User token detailed information

The image 3.6 shows the modal window with detailed information for the user token.

Error Pages

Two types of error pages are implemented:

- **Generic Error Page:** This page is displayed when the application encounters an error and propagates the error message to the view. It provides a standardized way to handle and display error messages.
- **Token Limit Error Page:** This is a specialized error page that informs users when they have reached the request limit for their token. It communicates the issue and prevents further requests until the limit is reset.

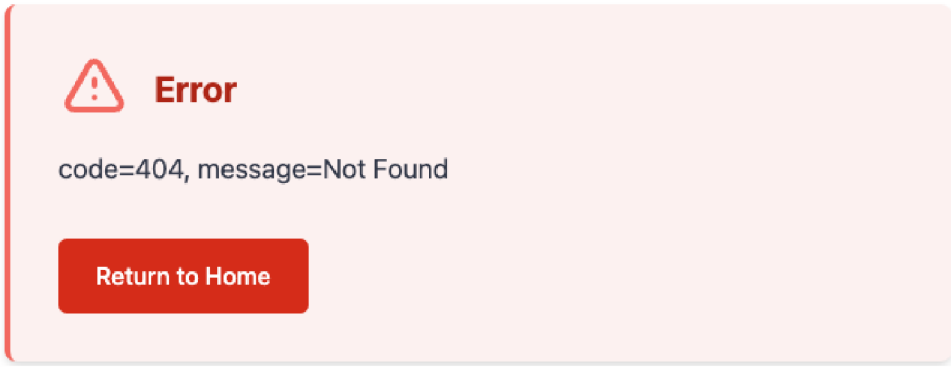

Figure 3.7: Generic error page

The image 3.7 shows the Error page when a user tries to reach a nonexistent URL.

4 Deployment

This chapter describes the deployment process of the web user interface together with BricksLLM component. The deployment is divided into two main parts: the deployment in Kubernetes and the creation of a Helm chart for easier deployment and management of the application.

4.1 Deployment in Kubernetes

The web user interface together with the BricksLLM component is deployed in a dedicated namespace called `bricksllm-ns` within the Rancher-managed Kubernetes cluster. The deployment architecture consists of several components that work together to provide the applications full functionality.

Deployments

The application has two deployments:

- **BricksLLM Component:** The core of the application that includes BricksLLM component, Redis database, and PostgreSQL database.
- **BricksLLM Dashboard:** The web user interface component that provides the front-end functionality for BricksLLM component.

Services

The following Kubernetes services are configured to enable communication between the various components and to expose functionality to users:

- **bricksllm-api:** Service for the BricksLLM component API.
- **bricksllm-redis:** Service for the Redis database.
- **bricksllm-ui:** Service for the BricksLLM dashboard.

- **bricksllm-ro:** Read-only service for the PostgreSQL database.
- **bricksllm-rw:** Read-write service for PostgreSQL database.

This separation of read and write services for the PostgreSQL database allows better performance optimization and security control by limiting write access to specific components that need it.

Ingress Configuration

To expose the application to external users, two ingress resources are configured:

- **API Ingress:** Exposes the BricksLLM component API.
- **Dashboard Ingress:** Exposes the BricksLLM dashboard.

These ingress resources are configured with appropriate TLS certificates to ensure secure communication.

Secret Management

All sensitive information required for the application to function properly is stored in external secrets, which are securely managed by the Kubernetes cluster. The following secrets have been configured:

- **bricksllm-ca:** Contains the certificate authority files.
- **bricksllm-db-app:** Contains credentials and configuration to access the PostgreSQL database.
- **bricksllm-redis:** Contains credentials and configuration to access Redis.
- **bricksllm-ui:** Contains all necessary secrets for the BricksLLM dashboard.

Using external secrets provides a secure way to manage sensitive information without exposing it in the deployment configuration files, which improves the overall security posture of the application.

4.2 Helm Chart

Initially, I deployed components manually in Rancher. To simplify my workflow and eventually create a Helm chart just for my work, I used Helmify to extract all the necessary components.

Using Helmify for Chart Generation

A tool called Helmify was used to generate the Helm chart from `bricksllm-ns` namespace. As stated in the GitHub repository [18], Helmify is a utility that converts existing Kubernetes resource definitions into a structured Helm chart, simplifying the process of generating a chart from an existing deployment.

The process of creating the Helm chart consisted of the following steps:

- First, all Kubernetes resources from the `bricksllm-ns` namespace were exported to a YAML file using the following command:

  ```
  kubectl --kubeconfig ../kuba-cluster.yaml
     get all -n bricksllm-ns -o yaml > resources.yaml
  ```

  where kuba-cluster.yaml contained the necessary connection information for the Kubernetes cluster.

- Next, the Helmify tool was used to convert the exported resources into a Helm chart:

  ```
  cat resources.yaml | helmify bricksllm-ns
  ```

The generated Helm chart encapsulates all components of the BricksLLM application, including deployments, services, ingresses, and secrets.

I filtered out only the components I had created myself and organized them into a clear, easy-to-read structure. To make the configuration more flexible, I moved all customizable values into the values.yaml file. This approach made my deployments easier to manage, while also laying a solid foundation for future automation and collaboration.

5 Conclusion

This thesis details the design and implementation of a user interface for the BricksLLM component, focusing on technical depth and practical deployment. The project began with a thorough theoretical overview of the selected web development technologies, including Go, HTMX, Templ, and the Echo web framework. These technologies were chosen for their modern approach to building robust, efficient, and maintainable web applications.

Go was chosen for its simplicity, performance, and strong support for concurrent programming, making it ideal for scalable back-end services. Adhering to Go best practices, such as maintaining clear code structure, effectively handling errors, and using meaningful naming conventions, ensured the application's reliability and maintainability. The Echo framework provided a high-performance, minimalist foundation for building HTTP servers. The Templ engine enabled type-safe, expressive templates that compile directly into Go code. This reduces runtime errors and improves developer productivity. HTMX was used to create a dynamic, responsive front end without the overhead of heavy JavaScript frameworks. This allowed for seamless, server-driven UI updates and a more maintainable code base.

A notable achievement of this thesis was successfully integrating the BricksLLM component into the JupyterHub service at the CERIT-SC center. This integration required an in-depth understanding of the BricksLLM component's architecture and capabilities, which are explored and documented in detail in section 1.2. The resulting web interface exposes BricksLLM's capabilities in a user-friendly environment and supports essential functionalities, such as user authentication, session management, tracking, and interaction with the external AI model provider, OpenAI.

A modern, cloud-native approach was used for deployment. The application was containerized and orchestrated with Kubernetes to ensure high availability, scalability, and ease of maintenance. Helm was used to manage the Kubernetes deployment's complexity, providing version-controlled, reusable charts and enabling automated rollouts, upgrades, and rollbacks as needed. Secrets management for sensitive

data and the /health/check API endpoint were used to secure the deployment and monitor application health.

Looking ahead, there are several ways to improve and expand this work. For instance, the interface could be updated to support additional AI model providers, and the user experience could be enhanced by incorporating feedback from actual users. Further optimization can also make the deployment process more reliable and efficient. It will be important to regularly gather feedback from users to ensure the tool continues to meet their evolving needs.

In summary, this thesis presents a practical solution for interacting with the BricksLLM component through user interface, combining web technologies, cloud-native deployment strategies, and a focus on usability.

A Contents of the Archive

The archive contains:

- A bricks-llm-dashboard.zip archive containing the full source code and configuration files from the associated GitHub repository.
- A bricksllm-ui-helm-chart.zip archive containing helm chart.

The bricks-llm-dashboard.zip contains:

- Application layers (api, app, cmd, core, static, view).
- .dev-env - Environment variables for development.
- compose.yaml - Docker Compose configuration for orchestrating multi-container setup.
- Dockerfile - Instructions for building the application Docker image.
- .air.toml - Live reloading during development.
- README.md - Information about environment variables, API routes, and how to run the application.
- Makefile - Automates the running of the application.
- jupyter.py - Example integration into JupyterHub.

Bibliography

1. *The State of AI: Global survey* [online]. McKinsey, 2025 [visited on 2025-03-28]. Available from: https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai.
2. *What is Go? Golang Programming Language Meaning Explained* [online]. Kolade Chris, 2021 [visited on 2025-02-11]. Available from: https://www.freecodecamp.org/news/what-is-go-programming-language/.
3. *The Go Programming Language* [online]. The Go Authors, 2025 [visited on 2025-02-13]. Available from: https://golang.org.
4. *HTMX Documentation* [online]. HTMX, 2025 [visited on 2025-02-16]. Available from: https://htmx.org/docs/.
5. *htmx* [online]. Wikipedia contributors, 2025 [visited on 2025-01-29]. Available from: https://en.wikipedia.org/wiki/Htmx.
6. *Introduction to HTMX* [online]. Refine, 2024 [visited on 2025-02-19]. Available from: https://refine.dev/blog/what-is-htmx/.
7. *Templ Documentation* [online]. Adrian Hesketh, 2025 [visited on 2025-02-21]. Available from: https://templ.guide.
8. *Echo Web Framework Documentation* [online]. LabStack, 2025 [visited on 2025-02-22]. Available from: https://echo.labstack.com/.
9. *BricksLLM: Enterprise-grade API gateway for LLMs* [online]. bricks-cloud, 2025 [visited on 2025-02-22]. Available from: https://github.com/bricks-cloud/BricksLLM.
10. *Kubernetes Documentation* [online]. The Kubernetes Authors, 2025 [visited on 2025-02-23]. Available from: https://kubernetes.io/docs/.
11. *Helm Documentation* [online]. The Kubernetes Authors, 2025 [visited on 2025-02-23]. Available from: https://helm.sh/docs/.
12. *Live reload for Go apps* [online]. Air, 2025 [visited on 2025-02-04]. Available from: https://github.com/air-verse/air.

13. *Go configuration with fangs* [online]. Viper, 2025 [visited on 2025-02-14]. Available from: https://github.com/spf13/viper.

14. *Multi-Provider Authentication for Go* [online]. Mark Bates, 2025 [visited on 2025-02-08]. Available from: https://github.com/markbates/goth.

15. *Cookie and filesystem sessions and infrastructure for custom session backends* [online]. Gorilla Sessions, 2025 [visited on 2025-02-09]. Available from: https://github.com/gorilla/sessions.

16. *Comprehensive Documentation for the GORM ORM Library* [online]. GORM, 2025 [visited on 2025-02-01]. Available from: https://gorm.io/docs.

17. *Redis Go client* [online]. Go Redis, 2025 [visited on 2025-02-16]. Available from: https://github.com/redis/go-redis.

18. *Creates Helm chart from Kubernetes yaml* [online]. Helmify, 2025 [visited on 2025-04-20]. Available from: https://github.com/arttor/helmify.