Motivationsreden-Generator

Entwickelte Anwendung

Wir haben eine Web-Anwendung entwickelt, mit der man sich Motivationsreden generieren lassen kann und sich diese als Audio-Datei anhören und herunterladen kann. Die Web-Anwendung wurde mit Java Script und Express.js entwickelt.

Um sich eine Motivationsrede zu generieren, kann der Nutzer der Web-Anwendung verschiedene Informationen angeben, aus denen dann der Text für die Rede generiert wird. Der Input des Nutzers besteht dabei aus maximal 3 Wörtern sowie einer Stimmung. Diese Informationen werden in einem Prompt zusammengeführt und an ein Large Language Modell (LLM) weitergeleitet. Als LLM wird das Modell „Llama3.2:1b Instruct“ verwendet, da dieses recht klein ist und somit nicht so viele Ressourcen benötigt. Um das Modell zu verwenden, nutzen wir Ollama. Ollama stellt eine lokale API bereit, mit der wird das LLM nahtlos in unsere Web-Anwendung integrieren konnten. Die generierte Motivationsrede wird anschließend an ein Text-to-Speech (TTS) Modell weitergeleitet, um eine Audio-Datei zu erstellen. Als TTS-Modell nutzen wir Bark-Small von Suno. Bark ist ein „transformer-based“ Text-to-Audio Modell, dass sehr realistische Audio-Dateien generieren kann. Die generierte Audio-Datei kann dann von dem Nutzer angehört sowie heruntergeladen werden. Alle Informationen (Anfrage, Stimmung, Motivationsrede, Audio-Datei) werden zusätzlich in einer Datenbank gespeichert und in einer Historie auf der Website angezeigt. Dafür verwenden wir eine MariaDB.

Vor- und Nachteile für unsere Anwendung aus der Realisierung als Cloud Native

Da unsere Anwendung als Cloud Native Anwendung und somit auch als Microservices-Architektur entwickelt wurde, kann diese in einem Kubernetes Cluster installiert werden. Dadurch erreichen wir eine hohe Verfügbarkeit, sowie eine automatische Skalierung basierend auf der Auslastung. Außerdem sind alle Bestandteile der Gesamt-Applikation unabhängig voneinander und können dementsprechend auch unabhängig voneinander skaliert sowie weiterentwickelt werden. Lediglich die APIs müssen konsistent bleiben, um einen Betrieb der Anwendung zu gewährleisten, auch wenn sich einzelne Komponenten verändern oder ausgetauscht werden. Außerdem ermöglicht diese Unabhängigkeit der einzelnen Komponenten und die Kleinteiligkeit schnellere Entwicklungszyklen. Wenn man in diese Richtung weiterdenkt, und die Applikation in Zukunft nicht in einer privaten Cloud, sondern bspw. bei AWS oder Azure hosten würde, könnte man die eigenen Komponenten für LLM und TTS durch bereits fertige KI-Dienste der Cloud-Anbieter austauschen und somit den Aufwand für Hosting und Pflege der Modelle reduzieren. Ein weiterer Vorteil, wenn eine Anwendung Cloud Native entwickelt wurde und verwendet wird, ist die Bezahlung der genutzten Ressourcen mit dem Prinzip Pay-per-Use. Dadurch können viele Kosten eingespart werden, da nur für die Hardware gezahlt wird, die tatsächlich verwendet wird, was insbesondere bei ressourcenintensiven Modellen wie LLMs oder TTS sinnvoll ist. Ein weiterer positiver Aspekt ist die einfache Integration von Mehrsprachigkeit. Wenn wir unsere Applikation global bereitstellen wollen, können wir von den Cloud-Anbietern leicht zusätzliche Sprachmodelle und TTS-Dienste in verschiedenen Sprachen nutzen.

Allerdings bringt Cloud Native auch Nachteile mit sich. Dazu gehören beispielsweise die Kosten bei hoher Nutzung. Die Kosten in der Public Cloud können sehr stark steigen, wenn viele Benutzer auf die Anwendung zugreifen oder rechenintensive Vorgänge ausgeführt werden, wie bspw. die Nutzung eines LLMs oder TTS-Modells. Außerdem können die Kosten auch unvorhersehbar sein, wenn die Nutzung stark schwankt. Zusätzlich besteht eine gewisse Abhängigkeit von Cloud

Anbietern, wenn man nicht aufpasst. Es besteht die Gefahr eines Vendor Lock-In, wenn man viele Dienste eines spezifische Cloud Anbieters nutzt. Außerdem hat man eine eingeschränkte Kontrolle über die Infrastruktur. Diesen Nachteil könnte man durch eine Private Cloud relativieren. Ein weiterer Punkt, den man bedenken sollte, ist die Latenz durch externe APIs sowie eine Abhängigkeit von den Netzwerkbedingungen. Durch die Verwendung von Microservices steigt außerdem die Komplexität in der Entwicklung und Verwaltung in Cloud Native Umgebungen wie z.B. Kubernetes. Ein zusätzlicher Punkt, der betrachtet werden muss, ist, wenn man externe Dienste für bspw. TTS-Services oder LLMs nutzt, ist die Abhängigkeit von der Verfügbarkeit von diesen externen Diensten.

Alternative Realisierungsmöglichkeiten und Vergleich mit der eigenen Lösung

Es gibt verschiedene alternative Realisierungsmöglichkeiten für unsere Anwendung. Dabei kann sowohl der Aufbau der Anwendung betrachtet werden als auch die Art des Betriebs.

Unsere Anwendung wurde so entwickelt, dass sie in Kubernetes deployt werden kann und sowohl in einem lokalen Kubernetes Cluster als auch in einem Kubernetes Cluster in der Cloud ausgeführt werden kann. Außerdem wurden sie als Microservice-Architektur entwickelt.

Anstelle von Kubernetes könnte man auch **separat deployte Microservices** einsetzen. Dabei werden die Dienste unabhängig voneinander bereitgestellt und auf verschiedenen Infrastrukturen wie virtuellen Maschinen oder Cloud-Services betrieben. Dies bietet eine höhere Flexibilität und technologische Vielfalt, erfordert jedoch manuelle Verwaltung von Skalierung, Kommunikation und Sicherheit zwischen den Services. Dies ist neben dem skizzierten Aufwand auch fehleranfälliger

Als Alternative zu einer Microservice-Architektur könnte es auch als **monolithische Gesamt-Anwendung** realisiert werden. Ein Monolith ist häufig einfacher zu entwickeln, zu testen und bereitzustellen, da keine komplexe Orchestrierung von Microservices notwendig ist. Allerdings gehen durch diese Art der Entwicklung alle Vorteile der Microservice-Architektur wie bspw. die Unabhängigkeit der einzelnen Komponenten verloren. Bei einer Cloud Native Applikation sind alle Bestandteile voneinander unabhängig und können dementsprechend auch sich wiedersprechende Abhängigkeiten haben. Bei einer monolithischen Gesamt-Anwendung müssen alle Bestandteile und deren Abhängigkeiten miteinander kompatibel sein, da diese auf der gleichen Hardware oder virtuellen Maschine (VM) installiert werden. Außerdem haben Monolithen Skalierungsprobleme bzw. es kann immer nur die Gesamt-Anwendung und nicht nur einzelne Bestandteile skaliert werden. Außerdem besteht eine geringere Flexibilität, wenn einzelne Komponenten verändert werden sollen.

Was im Absatz über den Monolith bereits angedeutet wurde, ist das **On-Premise Deployment**. Bei dieser Option würde die gesamte Infrastruktur lokal betrieben und man würde die Anwendung Bare Metal oder auf einer Virtuellen Maschine (VM) installieren. Dadurch hat der Betreiber die volle Kontrolle über die Hard- und Software und die Daten, aber dadurch auch mehr Verantwortung für Wartung und Skalierung. Zusätzlich besteht keine Abhängigkeit von Cloud-Anbietern oder deren Services, da alles lokal betrieben werden kann. Außerdem sind die Kosten kalkulierbarer, da diese nur einmalig bei der Anschaffung anfallen und nicht nutzungsbasiert wie in der Cloud. Ein weiterer Vorteil ist eine bessere Datenschutzkontrolle, da die Daten nicht in externen Clouds gehostet werden - allerdings ist das für unsere Anwendung bisher nicht relevant. Dieser Punkt wird aber ausführlicher im letzten Kapitel beschrieben.

Ein On-Prem Deployment bringt allerdings auch Nachteile mit sich, wie beispielsweise ein hoher Initialaufwand für die Anschaffung und Einrichtung der Infrastruktur sowie laufende Aufwände für Wartung und Sicherstellung des Betriebs. Außerdem ist es schwieriger schnell und nachfragebasiert zu skalieren und es bestehen hohe Fixkosten.

Ein weiterer Ansatz für den Betrieb ist ein **hybrider Ansatz**. Dabei kann bspw. die Hauptanwendung On-Prem betrieben werden und spezialisierte und ressourcenintensive Aufgaben wie die LLM-Anfragen und TTS-Audio-Erstellung in die Cloud ausgelagert werden. Hier würden wir also ebenfalls wieder auf eine Microservice Architektur zurückgreifen, die aber nicht zwingend Cloud Native entwickelt ist. Vorteile sind hier die Kosteneffizienz, Flexibilität und Skalierbarkeit für rechenintensive Teile der Anwendung. Die Hauptanwendung wird kostengünstig auf eigener Hardware betrieben, während die KI-Modelle nutzungsbasiert aufgerufen und bezahlt werden. Dieser Ansatz bringt eine gewisse Komplexität in der Verwaltung mit sich, da zwei verschiedene Architekturen verwaltet werden müssen. Außerdem besteht ein Latenzrisiko und man ist von externen Diensten und deren Verfügbarkeit abhängig.

Für eine solche Web-Anwendung, die verschiedene Komponenten kombiniert, die unterschiedliche Anforderungen an die zugrundeliegende Hardware haben, hängt die Wahl der Alternativlösung stark von den erwarteten Nutzungsmustern und den zu erwartenden Kosten ab. Wenn hohe Rechenanforderungen benötigt werden, aber Cloud-Dienste vermieden werden sollen, könnte ein dedizierter Server verwendet werden. Auf diesem kann dann dennoch bspw. ein privates Kubernetes Cluster aufgesetzt werden, wenn man die Vorteile dieser Cloud Native Plattform nutzen möchte.

Cloud Native Pattern

Infrastructure & Cloud

Die Pattern zu Infrastructure & Cloud beleuchten, wie Cloud-native Infrastrukturen die Bereitstellung und Skalierung von Anwendungen revolutionieren.

In unserer Architektur verwenden wir zum einen das Pattern: **Containerized Apps**. Dieses Pattern beschreibt die Verwendung von Containern, um Anwendungen in leichtgewichtigen, isolierten Umgebungen zu betreiben. Wir haben unsere Anwendung mit verschiedenen Microservices realisiert. Diese sind containerisiert und können in Kubernetes ausgeführt werden. Dadurch können die Abhängigkeiten einzelner Bestandteile der Applikation je Container bzw. Pod installiert werden und stehen somit nicht in Konflikt mit Abhängigkeiten von anderen Anwendungen. Außerdem kann unsere Anwendung auf allen beliebigen Kubernetes-Distributionen installiert und betrieben werden. Ein anderes Pattern, das beachtet wird, ist **Private Cloud**. Dieses beschreibt die Nutzung von Cloud-Technologien innerhalb der eigenen Infrastruktur, die exklusiv für ein Unternehmen bereitgestellt wird. Aktuell wird die Applikation auf lokaler Hardware bereitgestellt und kann über den Browser erreicht werden. In Zukunft kann die Anwendung außerdem in der Open Stack Cloud der DBHW Mannheim betrieben werden und ein DNS-Eintrag erstellt werden, um diese Anwendung für alle Nutzer, die sich im VPN der DHBW Mannheim befinden zugreifbar zu machen.

Development & Design

In diesem Bereich verwenden wir mehr Pattern. Zum einen das Pattern **zu Microservices Architekturen**. Dieses Pattern empfiehlt, Anwendungen in kleine, unabhängige Service zu unterteilen, die eigenständig entwickelt, bereitgestellt und skaliert werden können. Diese Services kommunizieren dann über APIs miteinander, wodurch sich neue Funktionen schneller einführen lassen und Services einfacher bearbeitet werden können. Dies deutet also auch schon auf das nächste Pattern hin: **Communicate through APIs**. Dieses empfiehlt die Verwendung von klaren und gut definierten APIs, um die Kommunikation zwischen verschiedenen Services zu standardisieren. In unsere Anwendung haben wir diese beiden Pattern realisiert, indem wir die Gesamt-Anwendung in vier voneinander unabhängige Services aufgeteilt haben: Datenbank, LLM, TTS-Service, Web-Anwendung. Die Web-Anwendung bildet die Logik ab, um die einzelnen

Komponenten zu verbinden. Aus der Web-Anwendung werden API-Anfragen an die anderen Services gestellt.

Außerdem nutzen wir das Pattern „**Avoid Reinventing the Wheel**”. Dieses beschreibt, dass man bestehende Lösungen und bewährte Praktiken verwenden soll, anstatt alles neu zu entwickeln. In unsere Web-Anwendung wird das realisiert, indem wir für den LLM Teil auf ein bestehendes Helm Chart für Ollama zurückgreifen. Dadurch werden zum einen bestehende LLMs verwendet, zum anderen wird eine implementierte Nutzung der Modelle wiederverwendet, sodass lediglich eine Datei angepasst werden muss, um zu beschreiben, welche Modelle deployt werden sollen. Auch für das Text-to-Speech (TTS)-Modell nutzen wir ein vortrainiertes Modell, um den Entwicklungsaufwand zu verringern. Über die Transformers-Bibliothek von Hugging Face laden wir das Modell suno/bark-small in unseren Container. Dies ermöglicht es uns, ohne tiefgreifende Modellentwicklung schnell auf leistungsfähige TTS-Funktionen zuzugreifen. Den Interference-Prozess können wir über den Transformers Modelling Code trotzdem mit geringem Aufwand beeinflussen.

Operations

Die Pattern zu Operations befassen sich mit dem Betrieb von Cloud-native Anwendungen. Aus diesem Bereich nutzen wir das Pattern „**Customization Through Microservices**“. Diese beschreibt im Prinzip wieder einen Vorteil einer Microservice-Architektur - nämlich die Unabhängigkeit der einzelnen Services und die dadurch entstehende Möglichkeit Anpassungen an den Services vorzunehmen, ohne die Gesamt-Anwendung bearbeiten zu müssen. Dadurch, dass wir unsere Anwendung als Microservice implementiert haben, sind die einzelnen Bestandteile voneinander unabhängig und können unabhängig voneinander weiterentwickelt werden. Die Web-Anwendung braucht lediglich weiterhin den Zugriff auf die APIs.

Ein weiteres Pattern, dass wir umsetzen ist **GitOps**. Dieses fördert die Verwaltung von Infrastruktur und Anwendungen über Git-Repositories und ermöglicht dadurch eine deklarative, versionierte und nachvollziehbare Steuerung von Änderungen. In unserem Projekt wurde ebenfalls mit einem Git-Repository gearbeitet, das auch in Zukunft weiterverwendet werden kann, um Änderungen nachverfolgen zu können.

Datensicherheit

Die Datensicherheit unserer Anwendung ist gewährleistet, da die eingegebenen Informationen der Nutzer weder sensibel noch schützenswert sind. Die Nutzer geben lediglich bis zu drei Wörter und eine Stimmungslage an, während die restlichen Daten durch eine KI generiert werden. Aktuell läuft die Anwendung lokal auf eigener Hardware. In Zukunft könnte diese in der Open Stack Cloud der DHBW Mannheim betrieben werden. Hier wäre der Zugriff auf die Anwendung auf das Netzwerk der DHBW Mannheim beschränkt, da die Applikation in der OpenStack Cloud innerhalb dieses Netzwerks gehostet wird. Sie ist somit aktuell und auch in Zukunft nicht für das gesamte Internet zugänglich, sodass nur Studenten und Professoren der DHBW Mannheim sowie teilweise der DHBW CAS Zugriff haben.

Hinsichtlich der DSGVO spielt diese für unsere Anwendung aktuell keine Rolle, da keine personenbezogenen Daten verarbeitet werden. Denkbar wäre jedoch eine zukünftige Erweiterung, bei der Nutzer Accounts angelegt werden könnten, um eigene frühere Anfragen und Audiodateien erneut anzusehen. In diesem Fall würden personenbezogene Daten verarbeitet, die dann den Anforderungen der DSGVO entsprechend geschützt werden müssten. Hierbei könnten Anonymisierungs- und Verschlüsselungstechniken zum Einsatz kommen, um den Datenschutzanforderungen gerecht zu werden. Auch die Implementierung eines robusten Authentifizierungsmechanismus wäre essenziell, um den Zugang zu den Daten zu kontrollieren.