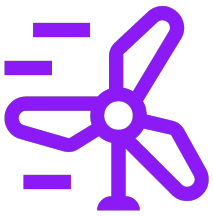

Entwicklung der Cloud Native App Green Grid Guide

Wind & Solar Potential Explorer

Laborarbeit in Cloud Infrastructures and Cloud Native Applications

Im Master Studiengang Informatik an der

DHBW-CAS

von

Linus Baumann, Alexander Dixon, Cosima Dotzauer, Dennis Hilgert, Jonas Werner

28. September 2025

Matrikelnummern	xxxxxxx, xxxxxxx, 3750882, xxxxxxx, xxxxxxx
Kurs	W3M20035
Semester	SoSe 2025
Begutachter	Prof. Dr. Christoph Sturm

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Vorstellung der Cloud-Native-Anwendung

Die in der Laborarbeit entwickelte Cloud-Native-Anwendung Green Grid GGuide hat das Ziel, Nutzern wie Energieunternehmen oder Grundstückseigentümern bei der Suche nach geeigneten Standorten für erneuerbare Energien zu unterstützen. Hierfür werden sowohl historische als auch aktuelle Wetterdaten sowie geographische Gegebenheiten analysiert, um potenzielle Standorte für Wind- oder Solaranlagen zu identifizieren und umfassend zu bewerten.

Die Bewertung dieser Standorte erfolgt anhand verschiedener Kriterien. Neben geeigneten Wetterbedingungen wird auch die Entfernung des Standorts zu bestehenden Stromleitungen berücksichtigt, da dies entscheidend für die wirtschaftliche Rentabilität einer möglichen Anlage ist. Zudem wird geprüft, ob sich der Standort in einem Naturschutzgebiet, im Wald oder in einem Wohngebiet befindet, da diese Faktoren Einfluss auf die Genehmigungschancen für den Bau einer Anlage haben.

Die Anwendung bietet den Nutzern eine intuitive Benutzeroberfläche inform einer interaktiven Karte und einstellbaren Parametern wie den Suchradius zur Visualisierung dieser Informationen, die bei der Entscheidungsfindung unterstützt und somit zur Planung von Wind- und Solaranlagen beiträgt.

2 Vorteile und Nachteile der Cloud-Native-Realisation

In diesem Kapitel werden die Vorteile und Nachteile der Cloud-Native-Realisierung der Anwendung zur Unterstützung der Standortfindung für erneuerbare Energien diskutiert. Zudem werden mögliche alternative Realisierungsmöglichkeiten aufgezeigt.

2.1 Vorteile und Nachteile der Cloud-Native-Realisierung

Die Realisierung der Anwendung als Cloud-Native-Anwendung bietet mehrere Vorteile:

Skalierbarkeit: Bei steigender Nachfrage, beispielsweise wenn viele Nutzer gleichzeitig potenzielle Standorte für Wind- oder Solaranlagen abfragen, können zusätzliche Instanzen der Microservices schnell bereitgestellt werden, um die Last zu bewältigen. Dies wird durch die Containerisierung und die Orchestrierung mit Kubernetes ermöglicht. Unbenötigte Instanzen können bei geringerer Nachfrage effizient heruntergefahren werden, was Ressourcen spart.

Unabhängige Entwicklung von Microservices: Die Nutzung von Microservices ermöglicht es verschiedenen Teammitgliedern, parallel an spezifischen Komponenten der Anwendung zu arbeiten. So kann ein Teammitglied beispielsweise an der Verarbeitung von Wetterdaten arbeiten, während ein anderes Teammitglied gleichzeitig den Geo-Microservice weiterentwickelt. Dies beschleunigt die Einführung neuer Funktionen, wie die Integration zusätzlicher Wetterdatenquellen, ohne die gesamte Anwendung zu beeinträchtigen.

Resilienz und Verfügbarkeit: Der Einsatz von Kubernetes-Primitiven wie Liveness- und Readiness-Probes gewährleistet, dass fehlerhafte Instanzen automatisch erkannt und neu gestartet werden. Wenn beispielsweise Fehler in der Wetterdatenabfrage in der Datenbank auftreten, führt dies nicht zu einem Ausfall der gesamten Anwendung.

Automatisierung in der Bereitstellung und einfacher Betrieb: Die Implementierung von CI/CD-Pipelines zur Automatisierung des Entwicklungs- und Bereitstellungsprozesses reduziert menschliche Fehler und beschleunigt die Zeit von der Entwicklung bis zur Produktion. Dies ist entscheidend, um zeitnah auf Änderungen in den regulatorischen Anforderungen für erneuerbare Energien zu reagieren und neue Datenquellen schnell zu integrieren. Hier eingesetzte Tools wie Ansible erleichtern das Infrastrukturmanagement und steigern die Effizienz der Anwendung.

Trotz dieser Vorteile gibt es auch einige **Nachteile:**

Erhöhter Overhead durch Containerisierung: Jeder Microservice läuft in einem eigenen Container, was zusätzliche Ressourcen benötigt und die Infrastrukturverwaltung komplizierter macht.

Herausforderungen bei der Netzwerkkommunikation: Die Sicherheit und Effizienz der Datenübertragung zwischen dem Weather-Microservice und der PostgreSQL-Datenbank können Schwierigkeiten bereiten. Auch die Kommunikation des Frontends mit den Microservices über ein Netzwerk kann potenzielle Latenzen und eine komplexere Architektur mit sich bringen. Wenn beispielsweise der Geo-Microservice aufgrund von Netzwerkproblemen nicht erreichbar ist, kann dies die gesamte Anwendung beeinträchtigen und den Zugriff auf wichtige Informationen für die Standortbewertung verzögern.

2.2 Alternative Realisierungsmöglichkeiten

Alternativen zur Cloud-Native-Architektur sind die monolithische Architektur und On-Premise-Lösungen. Jede dieser Alternativen bietet spezifische Vor- und Nachteile für die vorliegende Anwendung.

2.2.1 Monolithische Architektur

Eine monolithische Architektur integriert alle Funktionen der Anwendung in einer einzigen, großen Codebasis. Dies kann die Komplexität reduzieren und die Verwaltung erleichtern, da alle Komponenten als eine Einheit entwickelt, getestet und bereitgestellt werden.

Im Kontext der vorliegenden Anwendung würde dies bedeuten, dass sowohl die Verarbeitung der Wetterdaten als auch die geospatialen Analysen in einer einzigen Anwendung zusammengefasst wären. Dies könnte von Vorteil sein, da alle Informationen zur Bewertung eines Standorts auf einmal gesammelt werden, was die Integrationsphase vereinfacht.

Allerdings bringt eine monolithische Architektur erhebliche Nachteile mit sich. Die Flexibilität und Skalierbarkeit wären stark eingeschränkt, da die gesamte Anwendung als Einheit skaliert werden müsste. Bei einem Anstieg der Nutzerzahlen, beispielsweise wenn viele Grundstückseigentümer gleichzeitig potenzielle Standorte abfragen, wäre es nicht möglich, nur die wetterbezogenen Funktionen zu skalieren. Zudem würde die Entwicklung und Wartung der Anwendung erschwert, da keine einzelnen Teams unabhängig an spezifischen Services arbeiten könnten. Änderungen an einer Funktion könnten unbeabsichtigte Auswirkungen auf andere Teile der Anwendung haben.

2.2.2 On-Premise-Lösungen

On-Premise-Lösungen bieten der vorliegenden Anwendung mehrere Vorteile, insbesondere in Bezug auf Souveränität und Sicherheit.

Kontrolle über die Infrastruktur: On-Premise-Lösungen ermöglichen eine vollständige Kontrolle über die Infrastruktur und die Daten. Im Kontext der Anwendung könnte eine traditionelle serverbasierte Lösung alle Funktionen auf einem zentralen Server bündeln. Dies würde den initialen Aufwand für das Hosting und die Verwaltung der Anwendung verringern, da alle Wetter- und Geoinformationen direkt von einem Server abgerufen werden könnten.

Souveränität und Sicherheit: Durch den Betrieb der Anwendung auf eigenen Servern wird die Sicherheit der Daten erhöht, da sensible Informationen innerhalb der eigenen Räumlichkeiten gespeichert werden. Dies verringert die Risiken, die mit externen Cloud-Diensten verbunden sind, und ist besonders wichtig, wenn es um den Schutz kritischer Daten geht. In der aktuellen Anwendung hat die DSGVO zwar keine hohe Relevanz, da keine personenbezogenen Daten verarbeitet werden, jedoch könnten zukünftige Erweiterungen, die ein Pricing-Modell implementieren und somit sensible Informationen wie Zahlungsdaten erfordern, die Notwendigkeit für On-Premise-Lösungen verstärken.

Dennoch bringt eine monolithische Architektur, wie sie in On-Premise-Lösungen häufig anzutreffen ist, einige Herausforderungen mit sich. Die Agilität und Flexibilität der Anwendung wären eingeschränkt, da alle Funktionen als Einheit betrieben werden müssten. Im Falle eines Anstiegs der Nutzerzahlen könnte die Anwendung nicht effizient skaliert werden, was die Reaktionsfähigkeit und Leistung der Anwendung beeinträchtigen würde.

3 Cloud-Native Patterns in der Anwendungsarchitektur

In der entwickelten Cloud-Native-Anwendung zur Beratung für nachhaltige Energiegewinnungsstandorte finden sich verschiedene Cloud-Native Patterns. Nachfolgend werden die verwendeten Pattern "Microservices Architectureünd "Communicate Through APIsäus dem Bereich "Development & Designßowie die Pattern "Continuous Deliveryünd "Continuous Deploymentäus dem Bereich Ìnfrastructure & Cloud"vorgestellt, die beide jeweils eng miteinander verknüpft sind.

3.0.1 1. Microservices Architecture & Communicate Through APIs (Development & Design)

Die Anwendung folgt dem Pattern der **Microservices-Architektur**, bei dem die Gesamtanwendung in mehrere unabhängige, spezialisierte Services unterteilt wird, um Kosten für die Koordination zwischen Teams zu senken. [1]

Eng damit verbunden ist die Kommunikation mit den entsprechenden Microservices. Diese läuft über stabile und voneinander getrennte APIs, was dem Pattern **Communicate Through APIs** entspricht. [2].

In der vorliegenden Anwendung gibt es beispielsweise den **Weather-Microservice**, der für die Verarbeitung und Bereitstellung von Wetterdaten zuständig ist, und den **Geo-Microservice**, der Informationen zur geographischen Infrastruktur bereitstellt. **Weather-Microservice:** Dieser Service ist verantwortlich für die Verarbeitung und Bereitstellung von Wetterdaten. Er bezieht seine Informationen über die OpenMeteo-APIs, die sowohl historische als auch aktuelle Wetterbedingungen bereitstellen. Die Wetterdaten werden in einer PostgreSQL-Datenbank gespeichert, um eine effiziente Abfrage und Analyse zu ermöglichen. Zudem beinhaltet der Weather-Microservice ein Modell zur Wettervorhersage, das auf historischen Wetterdaten basiert und Prognosen für zukünftige Wetterbedingungen erstellt.

Geo-Microservice: Der Geo-Microservice liefert Informationen zur vorherrschenden Infrastruktur, einschließlich der Überprüfung der Entfernung zu bestehenden Stromleitungen und der Lage in Bezug auf Naturschutzgebiete, Wald oder Wohngebiete. Die Daten für diesen Service werden aus der Overpass-API von OpenStreetMap bezogen.

Beide Microservices sind über REST-APIs erreichbar und in Containern isoliert, was eine flexible und skalierbare Architektur gewährleistet.

Vorteile des Einsatzes: Die Microservices-Architektur ermöglicht eine modulare Entwicklung, bei der unterschiedliche Teammitglieder an den Komponenten der Anwendung parallel arbeiten können. Die Entwicklung des Weather-Microservice erfolgte beispielsweise separat von der des Geo-Microservices. Dies ermöglicht eine höhere Agilität der Entwicklung und erhöht die Entwicklungsgeschwindigkeit. Zudem kann jeder Microservice die für ihn am besten geeigneten Technologien verwenden, was die Effizienz und Leistung der jeweiligen Anwendung verbessert. Der Weather-Microservice nutzt beispielsweise eine PostgreSQL-Datenbank zur Speicherung der Wetterdaten. Würde der Geo-Service auch eine Datenbank benötigen, könnte er eine PostgreSQL-Datenbank mit PostGIS-Erweiterung nutzen, welche speziell für räumliche Datentypen gedacht ist und eine einfachere Suche nach Orten innerhalb des Suchradius ermöglicht. Diese Erweiterung wäre für den Weather-Microservice nicht benötigt.

Eng mit der Verwendung der Microservices wird bei der hier vorliegenden Anwendung auch das Pattern **Communicate Through APIs** genutzt. Alle Interaktionen zwischen dem Frontend und dem Weather- sowie dem Geo-Microservice erfolgen über klar definierte REST-APIs. Dies trägt ebenso positiv zur Wartbarkeit und Flexibilität der Anwendung bei. Wenn beispielsweise der Geo-Microservice überearbeitet werden muss, bleibt das Frontend unberührt, solange die API-Schnittstelle beibehalten wird.

3.0.2 2. Continuous Delivery und Continuous Deployment (Infrastructure & Cloud)

Ein zentrales Pattern in der Anwendungsarchitektur ist **ContinuousDelivery**, das einen schnellen Programmier-, Test- und Auslieferungszyklus ermöglicht. Dies erlaubt es, neue Funktionen zügig zu veröffentlichen und schnell auf Benutzerfeedback zu reagieren [3]. Eng verknüpft mit diesem Pattern ist das **ContinuousDeployment**, das sicherstellt, dass jede Änderung, die den Testprozess besteht, automatisch in die Produktionsumgebung überführt wird. Dies sorgt dafür, dass neue Funktionen schnell hinzugefügt und Änderungen bei Bedarf rückgängig werden können.[4]

In der Anwendung wird Continuous Delivery und Deployment durch den Einsatz von GitHub Actions umgesetzt. Hier werden CI/CD-Pipelines erstellt, die bei Pull-Requests auf den Main-Branch automatisch ausgeführt werden. Jede Änderung am Weather-Microservice und Geo-Microservice wird mithilfe von Unit-Tests automatisch geprüft, um sicherzustellen, dass keine bestehenden Funktionen beeinträchtigt werden.
Nach erfolgreichem Testen wird ein Docker-Image erstellt und in einer Container-Registry gespeichert.

Anschließend erfolgt die automatische Bereitstellung des neuen Images in der Produktionsumgebung, wodurch die Anwendung stets auf dem neuesten Stand gehalten wird. Dieser Workflow wird in Abbildung 3.1 graphisch dargestellt.

Abbildung 3.1: CI/CD-Pipeline mit GitHub Actions [Eigene Abbildung]

Die weitere Umsetzung, wie das Pattern Continuous Deployment und Delivery in der Anwendung umgesetzt wurde, ist in Abbildung 3.2 dargestellt. Im Folgenden wird die Umsetzung kurz erläutert:

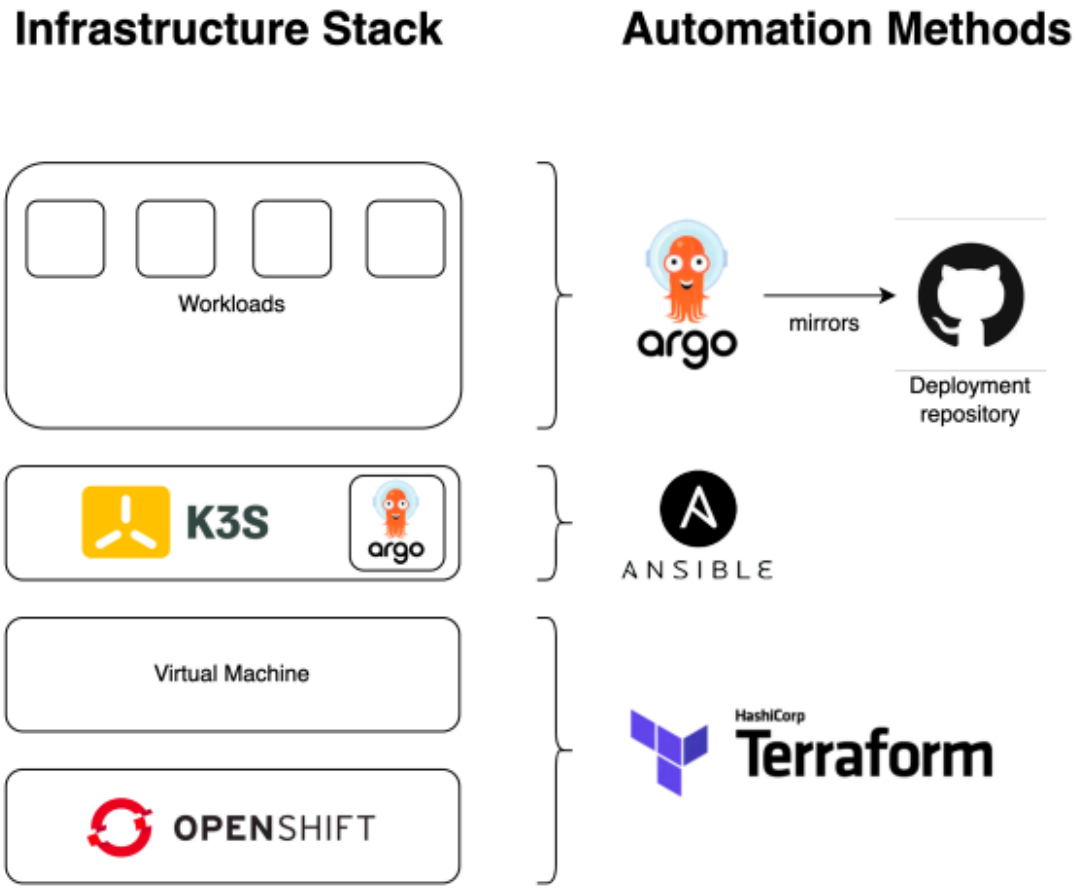

Abbildung 3.2: Infrastruktur der Anwendung und zugehörige Automatisierungsmethoden [Eigene Abbildung]

Auf der obersten Ebene befinden sich die "Workloads", die die eigentlichen Anwendungen darstellen, wie zum Beispiel das Frontend oder die einzelnen Microservices. Änderungen der Workloads, die durch eine CI-Pipeline getestet wurden, werden in ein "Deployment Repositoryäuf GitHub geschoben.

Argo CD überwacht bzw. spiegelt dieses Repository.
Unter dieser Ebene befindet sich K3S und, eine leichtgewichtige Kubernetes-Distribution und Argo CD sowie Ansible. K3S stellt hier die Laufzeitumgebung für containerisierte Anwendungen dar. Sobald eine Änderung im Deployment Repository erkannt wird, pullt Argo CD diese Änderungen automatisch und wendet sie auf dem K3S Ziel-Cluster an. Ansible dient zur Provisionierung und zu Konfiguration der Virtual Machine, auf denen K3S läuft. Zum Beispiel wird Ansible genutzt, um neue VMs zu erstellen oder Abhängigkeiten zu konfigurieren.
Für die bereitstellung der Virtual Machine wird Terraform genutzt. Terraform ermöglicht es, die gesamte Infrastruktur als Code zu definieren und zu verwalten, einschließlich der Virtual Machines, auf denen OpenShift installiert wird. Dies wird ebenfalls von Terraform durchgeführt. So wird eine konsistente und reproduzierbare Infrastruktur bereitsgestellt.

Zusammengefasst ist die gesamte Pipeline, beginnend mit der Entwicklung, über die automatisierten Tests, bis hin zur Möglichkeit, die Anwendung und Infrastruktur in eine automatisiert zu deployen, auf Continuous Deployment und Continuous Delivery ausgelegt. Dadurch wird die Qualität der Anwendung sichergestellt, da Änderungen kontinuierlich getestet werden, und eine schnelle Reaktion auf Feedback von Nutzern sowie auf sich ändernde Anforderungen ermöglicht werden.

4 Datensicherheit und Datenschutz in Cloud-Native-Anwendungen

Im Kontext der vorliegenden Anwendung hat die Datenschutz-Grundverordnung (DSGVO) derzeit eine geringe Relevanz, da keine personenbezogenen Daten verarbeitet werden. Nutzer benötigen keinen Account, um die Anwendung zur Suche nach Standorten innerhalb Deutschlands zu verwenden, weshalb auch keine entsprechenden Datenschutzmechanismen implementiert wurden.

Für zukünftige Versionen der Anwendung ist jedoch ein Pricing-Modell vorgesehen, welches den Zugriff auf Standorte außerhalb Europas ermöglicht. Dieses Modell würde die Erfassung personenbezogener Daten, wie Name und Zahlungsinformationen, erfordern sowie die Einwilligung der Nutzer zur Datenverarbeitung.

Die Sicherheit dieser Daten muss während der Erzeugung, Übertragung, Nutzung, Speicherung, Archivierung und Löschung gewährleistet sein. Entsprechende Sicherheitsmaßnahmen müssen sowohl auf Netzwerk- als auch auf Anwendungsebene implementiert werden. Dabei sind wichtige Aspekte zu berücksichtigen, wie beispielsweise der Standort der Server, auf denen die Nutzerdaten gespeichert werden.

Literaturverzeichnis

[1] *Pattern: Microservices Architecture*. https://www.cnpatterns.org/development-design/microservices-architecture. Accessed: 2024-09-19. URL: https://www.cnpatterns.org/development-design/microservices-architecture.

[2] *Pattern: Communicate Through APIs*. https://www.cnpatterns.org/development-design/communicate-through-apis. Accessed: 2025-09-19.

[3] *Pattern: Continuous Delivery*. https://www.cnpatterns.org/infrastructure-cloud/continuous-delivery. Accessed: 2024-09-19. URL: https://www.cnpatterns.org/infrastructure-cloud/continuous-delivery.

[4] *Pattern: Continuous Deployment*. https://www.cnpatterns.org/infrastructure-cloud/continuous-deployment. Accessed: 2024-09-19. URL: https://www.cnpatterns.org/infrastructure-cloud/continuous-deployment.