Westfälische Wilhelms-Universität Münster
Institut für Geoinformatik

Bachelorarbeit
im Fach Geoinformatik

Rich Data Interfaces for Copernicus Data

Themensteller: Prof. Dr. Albert Remke
Betreuer: Dr. Christian Knoth, Dipl.-Geoinf. Matthes Rieke
Ausgabetermin: 20.05.2022
Abgabetermin: 22.08.2022

Vorgelegt von: Alexander Nicolas Pilz
Geboren: 06.12.1995
Telefonnummer: 0176 96982246
E-Mail-Adresse: apilz@uni-muenster.de
Matrikelnummer: 512 269
Studiengang: Bachelor Geoinformatik
Fachsemester: 6. Semester

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

C3S Climate Change Service

CAMS Copernicus Atmosphere Monitoring Service

CLMS Copernicus Land Monitoring Service

CMEMS Copernicus Marine Environment Monitoring Service

CORS Cross-Origin Resource Sharing

DIAS Data and Information Access Services

EMS Emergency Management Service

ESA European Space Agency

GMES Global Monitoring for Environmental Security

GRD Ground Rage Detected

HATEOAS Hypermedia As The Engine Of Application State

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON Java Script Object Notation

KML Keyhole Markup Language

NDSI Normalized Difference Sigma-Naught Index

OGC Open Geospatial Consotium

OSW Ocean Swell Spectra

OWI Ocean Wind Field

REST Representational State Transfer

RVL Radial Surface Velocity

SAR Synthetic Aperture Radar

ScanSAR Scanning Synthetic Aperture Radar

SLC Single Look Complex

SM Stripmap Mode

SNAP Sentinel Application Platform

SNDSI Shannons Entropy of Normalized Difference Sigma-Naught Index

TOPSAR Terrain Observation with Progressive Scans SAR

URL Uniform Resource Locator

UUID Universally Unique Identifier

WSGI Web Server Gateway Interface

XML Extensible Markup Language

YAML Yet Another Markup Language

1 Einleitung

1.1 Motivation

Geodaten werden für die unterschiedlichsten Anwendungsfälle benötigt. Zu diesen Anwendungsfällen gehören unter anderem der Umwelt- und Katastrophenschutz. Dort werden flächendeckende, verlässliche und aktuelle Geodaten benötigt. Diese müssen schnell und einfach beschafft und verarbeitet werden können, um rechtzeitig Maßnahmen zu ergreifen oder mögliche Schäden zu quantifizieren. Flächendeckende Fernerkundungsdaten von diversen Plattformen werden vom europäischen Copernicus-Programm erfasst und im Internet bereitgestellt und können über Webseiten und APIs beschafft werden. Manche Daten, wie die Radardaten der Sentinel-1 Mission, bedürfen jedoch einer komplexen Vorverarbeitung bevor aus ihnen verlässliche Aussagen über die in den abgebildeten Räumen aufgezeichneten Phänomene getätigt oder andere Daten abgeleitet werden können [20]. Diese Vorverarbeitungen können nicht von allen Nutzer*innen selbst durchgeführt werden. Zum einen verfügt nicht jeder über die nötigen Fachkenntnisse und zum anderen stehen nicht jedem die entsprechenden technischen Infrastrukturen zur Verfügung. Um dieses Problem zu lösen und Nutzer*innen mit analysebereite oder interpretationsfähige Daten zu versorgen, können sogenannte Rich Data Interfaces zum Einsatz kommen. Diese versetzen Nutzer*innen in die Lage, analysebereite oder interpretationsfähige Daten direkt über eine API beziehen zu können. Um die Interoperabilität sicherzustellen, sollte diese nach den Maßgaben eines Standards wie dem OGC API - Processes - Part 1: Core implementiert werden. Beziehen Nutzer*innen analysebereite oder interpretationsfähige Daten in standardisierter Weise bietet dies eine Reihe von Vorteilen. So ist sichergestellt das die Daten in gleicher Weise und in möglichst hoher Qualität vorprozessiert wurden und so die Interoperabilität sichergestellt ist. Darüber hinaus werden Analysen und Ergebnisse leichter vergleichbar da die ihnen zugrunde liegenden Daten den selben Ansprüchen genügen [17].

1.2 Ziele

Im Rahmen dieser Arbeit soll untersucht werden, ob sich der OGC API - Processes - Part 1: Core Standards als Grundlage zur Implementierung von Rich Data Interfaces für die Daten des Copernicus-Programmes eignet. Dazu soll eine prototypische Anwendung in der Programmiersprache Python entwickelt werden, welche eine OGC API - Processes - Part 1: Core standardkonforme API anbietet. Um einen Bezug zu einem realistischen Anwendungsszenario zu schaffen, soll die Anwendung Überschwemmungsmonitoring auf Basis von Sentinel-1 Daten ermöglichen. Ergebnisse des Überschwemmungsmonitorings sollen der NDSI, also analysebereite Daten sowie aus diesem abgeleitete Überschwemmungsmasken, also interpretationsfähige Daten sein. Zusätzlich sollen die Sentinel-1 Datensätze vollständig durch die Anwendung kalibriert werden. Außerdem soll ein Konzept für den Zugriff auf Daten des Copernicus-Programmes erarbeitet und implementiert werden. Schlussendlich wird eine heuristische Evaluation der proto-

typischen Implementierung durchgeführt. Dazu werden die Nutzbarkeitsheuristiken nach Jakob Nielsen verwendet, welche die Nutzerfreundlichkeit und Anwendbarkeit in den Fokus ihrer Betrachtung rücken.

1.3 Aufbau

Im ersten Teil dieser Arbeit sollen die nötigen fachlichen Grundlagen kurz erläutert werden. Dazu zählen die Grundlagen der Radarfernerkundung mit Synthetic Aperture Radar Systemen (SAR), die Ziele sowie der Aufbau des Copernicus-Programmes, die Vorstellung eines möglichen Verfahrens zum radargestütztem Überschwemmungsmonitoring und eine kurze Vorstellung des OGC sowie dem OGC API - Processes - Part 1: Core Standard. Im Implementierungsteil wird die prototypische Implementierung des Rich Data Interface für Copernicus-Daten vorgestellt. Dabei wird der verwendete Softwarestack, die Grundstruktur der Anwendung sowie die Funktionsweisen der Endpoints der API und der angebotenen Prozesse beschrieben. Auf die prototypische Implementierung aber auch auf die durch den Standard beschränkten Möglichkeiten bezugnehmend folgt die Evaluation der Anwendung. Diese geschieht auf Basis der von Nielsen vorgeschlagenen Heuristiken, welche sich auf einzelne Aspekte der Nutzbarkeit von Anwendungen beziehen. Der Diskussionsteil schafft Raum für die Erörterung der Forschungsfrage. Im Ausblick wird kurz umrissen wie die prototypisch umgesetzte Anwendung sinnvoll weiterentwickelt werden kann. Als Abschluss der Arbeit soll ein kurzes Fazit dienen, welches die gewonnenen Erkenntnisse zusammenfasst.

2 Grundlagen

2.1 Radarfernerkundung

Bei der Radarfernerkundung werden vom Radarsystem in regelmäßigen Abständen elektromagnetische Signale ausgesandt. Nach dem Senden eines Signals folgt ein Zeitfenster, indem die Plattform auf Echos des ausgesandten Signals wartet. Trifft das ausgesandte Signal auf eine Oberfläche, zum Beispiel die Erdoberfläche, wird ein Bruchteil in Richtung Empfänger reflektiert und als Echo vom Fernerkundungssystem empfangen [21].
Die Radarfernerkundung gehört zu den aktiven Fernerkundungsmethoden, da hier im Gegensatz zur optischen Fernerkundung nicht nur von Oberflächen reflektierte Strahlung von anderen Strahlungsquellen wie der Sonne aufgenommen wird, sondern das Fernerkundungssystem selbst als Strahlungsquelle dient. Messungen können daher tageszeitnähbandgig erfolgen. Bildgebende Radarsysteme werden auf mobilen Plattformen montiert und blicken seitlich auf die zu beobachtende Oberfläche [21] (siehe Abbildung 1).
Die Eigenschaften des reflektierten Signals hängen sowohl von Parametern des Aufnahmesystems als auch von Parametern der reflektierenden Oberfläche ab. So werden in der Radarfernerkundung verschiedene Frequenzbänder verwendet, welche sich in Frequenz und Wellenlänge unterscheiden. Da sich die Wechselwirkungen zwischen Signalen unterschiedlicher Frequenzbänder und den reflektierenden Oberflächen unterscheiden, können so unterschiedliche Aspekte der beobachteten Oberflächen hervorgehoben werden. Dabei kommen in der Regel Wellenlängen von 0,75m bis 120m zum Einsatz. Mit einer größeren Wellenlänge kann ein Medium auch tiefer durchdrungen werden. Außerdem werden Wolken, Dunst und Rauch durchdrungen, was den zusätzlichen Vorteil bietet, wetterunabhängig Messungen durchführen zu können [1].
Die Rauigkeit ist eine Eigenschaft der reflektierenden Oberfläche und hat großen Einfluss auf das reflektierte Signal. Ist diese im Verhältnis zur verwandten Wellenlänge gering, so kommt es zu einer spiegelnden Reflektionen und nur ein geringer Anteil des Signals wird zum Empfänger zurückgeworfen. Zusätzlich ist die Polarisation der ausgesandten und empfangenen Signale bei der Messung ausschlaggebend. Sie können horizontal oder vertikal polarisiert sein. Dies führt zu vier möglichen Polarisationsmodi für das Senden und das Empfangen, nämlich HH, VV, HV und VH. Auch die Polarisation sorgt für eine unterschiedliche Wiedergabe von beobachteten Objekten und kann somit verwendet werden, um bestimmte Aspekte hervorzuheben [1]. Die Auflösung entlang des Azimut unterscheidet sich von der Auflösung in Blickrichtung. Die Auflösung in Azimutrichtung wird von der Antennenlänge bestimmt, da diese festlegt wie lange die Reflektionen eines Objektes empfangen werden. Die Antennenlänge kann bauartbedingt nicht beliebig gesteigert werden. Die Auflösung in Blickrichtung hängt von der Bandbreite ab, welche sich aus der Sendefrequenz und der Signaldauer zusammensetzt. Bei Radarsystemen mit einer synthetischen Apertur wird durch die Bewegung des Sensors in Azimutrichtung die wirksame Antennenlänge rechnerisch verlängert, indem die reflektierten Signale eines beobachteten Objektes von verschiedenen Standpunkten und unterschiedlichen Zeitpunkten miteinander kor-

reliert werden. So können hohe Azimutauflösungen erzielt werden. Solche Systeme eigenen sich auch für den Einsatz auf Satelliten [1].

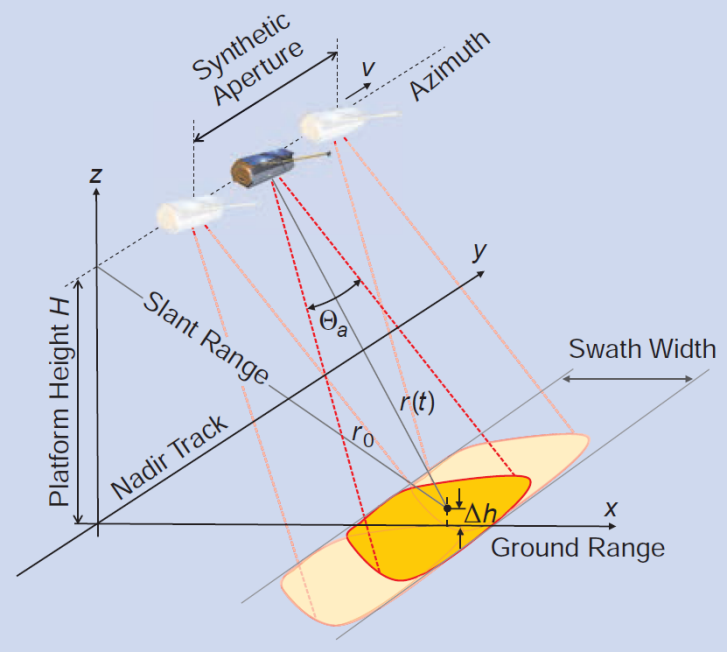

Abbildung 1: Prinzip eines SAR Fernerkundungssystems [21]

SAR-Systeme können in unterschiedlichen Aufnahmeverfahren arbeiten. Das einfachste dieser Verfahren ist das Stripmap Verfahren, bei welchem nur ein Aufnahmestreifen kontinuierlich aufgenommen wird. Breitere Aufnahmestreifen können mit dem ScanSAR Verfahren erzielt werden. Dabei werden unter verschiedenen Depressionswinkeln, in Blickrichtung und zeitversetzt mehrere Subaufnahmestreifen erzeugt. Im Vergleich zum Stripmap Verfahren ist die Auflösung jedoch geringer. Wird eine höhere Auflösung benötigt kann das Spotlight Verfahren zum Einsatz kommen, bei dem eine fixe Region über einen längeren Zeitraum hinweg beobachtet wird. Dies führt zu einer sehr langen wirksamen Antenne (siehe Abbildung 2). Angepasste Verfahren oder Mischformen können je nach Beobachtungsszenario zum Einsatz kommen [21].

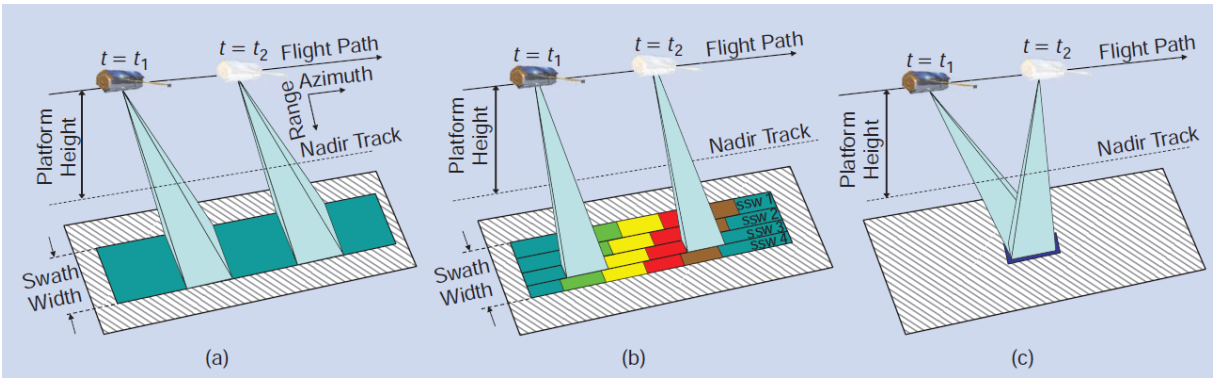

Abbildung 2: Aufnahmeverfahren SAR Systemen (a) Strimpmap (b) ScanSAR (c) Spotlight [21]

Im Gegensatz zu optischen Aufnahmeverfahren liefern die Rohdaten einer Befliegung mit Radarsensoren noch keine Bilddaten. Um Bilder zu erzeugen, bedarf es zunächst einer komplexen Verarbeitung der aus Amplitude und Phase bestehenden reflektierten Signale. Dabei werden die Daten entlang des Azimuts und der Blickrichtung gefiltert. In der Regel repräsentieren die Pixelwerte eines aus Radardaten abgeleiteten Bildes die Reflektivität des korrespondierenden Bodenelements. Mittels Geocodierung kann das so entstandene Bild verortet werden. Zusätzlich können diverse, ebenfalls rechen- und zeitintensive, Kalibrierungen vorgenommen werden. Dazu gehören Verfahren, welche Rauscheffekte minimieren, die geometrischen Eigenschaften verbessern oder die Interpretation der Bilder erleichtern [21].

2.2 Copernicus-Programm

2.2.1 Ziele

Das Copernicus-Programm ging aus dem Global Monitoring for Environmental Security Programm Programm (GMES) hervor, welches 1998 mit dem Ziel initiiert wurde, Europa zu ermöglichen, eine führende Rolle bei der Lösung von weltweiten Problemen im Kontext Umwelt und Klima zu verschaffen. Teil dieser Bestrebungen ist der Aufbau eines leistungsfähigen Programms zur Erdbeobachtung. 2012 wurde das GMES-Programm zum Copernicus-Programm umbenannt [6]. Erklärte Ziele des Copernicus-Programmes sind das Überwachen der Erde, um den Schutz der Umwelt sowie Bemühungen von Katastrophen- und Zivilschutzbehörden zu unterstützen. Gleichzeitig soll die Wirtschaft im Bereich Raumfahrt und der damit verbundenen Dienstleistungen unterstützt und Chancen für neue Unternehmungen geschaffen werden [9].

2.2.2 Aufbau

Das Copernicus-Programm besteht aus Weltraum, In-Situ- und Service-Komponente. Zur Weltraum-Komponente gehören die verschiedenen Satellitenmissionen sowie Bodenstationen, welche für den Betrieb sowie die Steuerung und Kalibrierung der Satelliten sowie der Verarbeitung und Validierung der Daten verantwortlich sind [9].
Sentinel-1 Satelliten sind mit bildgebenden Radarsystemen ausgerüstet und beobachten wetter- und tageszeitunabhängig Land-, Wasser- und Eismassen, um unter anderem das Krisenmanagement zu unterstützen. Satelliten der Sentinel-2 Mission führen hochauflösende, multispektrale Kameras mit und liefern weltweit optische Fernerkundungsdaten.
Altimetrische und radiometrische Daten von Land- und Wasserflächen werden von der Sentinel-3 Satellitenmission gesammelt, während spektrometrische Daten zur Überwachung der Luftqualität von Sentinel-4 und 5 Satelliten erfasst werden. Ozeanografische Daten sollen von den Sentinel-6 Satelliten geliefert werden [12].
Die In-Situ-Komponente sammelt Daten von see-, luft- und landbasierten Sensoren sowie geografische und geodätische Referenzdaten. Die harmonisierten Daten werden verwendet, um

die Daten der Weltraum-Komponente zu verifizieren oder zu korrigieren. Gleichzeitig können räumliche oder thematische Lücken in der Datenabdeckung gefüllt werden [9] [8].

Zur Service-Komponente gehören unterschiedliche Dienste, welche jeweils auf ein Themengebiet abgestimmt sind und Daten in hoher Qualität bereitstellen. Der Copernicus Atmosphere Monitoring Service (CAMS) soll Informationen zur Luftqualität und der chemischen Zusammensetzung der Atmosphäre liefern. Daten bezüglich des Zustands und der Dynamik der Meere und deren Ökosysteme lassen sich über den Copernicus Marine Environment Monitoring Service (CMEMS) beziehen. Informationen zur Flächennutzung und Bodenbedeckung werden vom Copernicus Land Monitoring Service (CLMS) bereitgestellt. Um eine nachhaltige Klimapolitik planen und umsetzen zu können, stellt der Copernicus Climate Change Service (C3S) aktuelle sowie historische Klimadaten bereit. Um den Zivilschutzbehörden schnelle Reaktionen auf Umweltkatastrophen zu ermöglichen, stellt der Emergency Management Service (EMS) entsprechende Fernerkundungsdaten bereit. Ähnliche Daten können von europäischen Zoll- und Grenzschutzbehörden über den Copernicus Security Service bezogen werden [9] [8].

2.2.3 Sentinel 1

Die Sentinel-1 Satellitenmission liefert wetter- und tageszeitunabhängige Radardaten der Erdoberfläche. Die Mission besteht aus zwei Satelliten, Sentinel-1 A und B, sowie einer Bodenkomponente, welche für Steuerung, Kalibrierung und Datenverarbeitung verantwortlich ist. Die Satelliten tragen als Hauptinstrument ein bildgebendes Radar mit synthetischer Apertur, welches im C-Frequenzband arbeitet. Es stehen zwei Polarisationsmodi, Single (HH, VV) oder Dual (HH+HV, VV+VH) zur Verfügung [2]. Die Erfassung von Daten kann in vier Aufnahmemodi erfolgen, welche sich in Auflösung, Streifenbreite und Anwendungsszenario unterscheiden (siehe Tabelle 1). Der Standardmodus ist der Stripmap Modus (SM) bei dem Aufnahmestreifen mit einer kontinuierlichen Folge von Signalen abgetastet werden [2]. Die Aufnahmemodi Interferometric Wide Swath Mode (IW) und Extra-Wide Swath Mode (EW) arbeiten im TOPSAR Verfahren mit drei beziehungsweise fünf Sub-Aufnahmestreifen, um ein größeres Gebiet aber in geringerer Auflösung aufnehmen zu können. TOPSAR ist eine Abwandlung des ScanSAR Verfahrens, bei welchem die Antenne zusätzlich in Azimut-Richtung vor und zurück bewegt wird, um die radiometrische Qualität der resultierenden Bilder zu verbessern. Wenn der Wave Modus (WV) zum Einsatz kommt, werden kleine, Vignetten genannte, Szenen im Stripmap Verfahren aufgenommen. Sie werden in regelmäßigen Abständen und wechselnden Depressionwinkeln aufgenommen [21] [2].

Tabelle 1: Eigenschaften der Aufnahmemodi der Sentinel-1 Mission [11]

Modus	IW	WV	SM	EW
Polarisation	Dual	Single	Dual	Dual
Azimutauflösung (m)	20	5	5	40
Rage-Auflösung (m)	5	5	5	20
Streifenbreite (km)	250	20x20	80	410

Beide Satelliten befinden sich auf einem polnahen, sonnensynchronen Orbit. Ein Zyklus dauert 12 Tage, in denen die Erde 175 Mal umrundet wird. Da es sich um ein Satellitenpaar handelt, welches als Tandem die Erde umrundet, wird ein Punkt auf der Erdoberfläche alle sechs Tage von einem der Satelliten überflogen. Das System kann eine zuverlässige globale und systematische Abdeckung liefern. Dabei können im IW Modus alle relevanten Land-, Wasser- und Eismassen alle zwölf Tage vollständig von einem Satelliten erfasst werden. In Krisensituationen können Daten nach Bedarf innerhalb von zweieinhalb bis fünf Tagen Daten erfasst werden [11]. Nach dem Erfassen der Daten und Übersenden an eine Bodenstation werden diverse Vorverarbeitungsschritte vorgenommen, in die sowohl interne also auch externe Parameter einfließen. Daraus ergeben sich diverse Produkte, welche sich durch Aufnahmemodus, Produkt-Typ sowie durch ihre Auflösung unterscheiden. SLC-Produkte sind im wesentlichen kalibrierte Rohdaten, in denen Amplitude und Phase nicht zur Reflektivität kombiniert wurden und die geometrische Auflösung sich in Azimut- und Blickrichtung unterscheidet. GRD-Produkte bilden hingegen die Reflektivität ab und haben eine annähernd quadratische geometrische Auflösung. Die Reflektivität wird in der logarithmischen Maßeinheit Dezibel (dB) angegeben. Die Korrektur der Schrägdistanz in Blickrichtung erfolgt durch Projektion auf einen Ellipsoiden. [2].

2.2.4 Datenzugang

Die Daten des Copernicus-Programmes sollen einer möglichst breiten Nutzergruppe möglichst einfach zugänglich gemacht werden. Sie sollen frei zugänglich und kostenlos angeboten werden [9]. Daten der Sentinel-1, 2, 3 und 5 können über das von der ESA betriebene Copernicus Open Access Hub bezogen werden. Datensätze können sowohl auf der Webseite als auch mithilfe einer API gesucht und heruntergeladen werden. Daten der Sentinel-3, 4, 5 und 6 sowie weiterer Satelliten können über das dem Copernicus Open Access Hub ähnlichen EUMETCast bezogen werden. In Ergänzung zu diesen Quellen werden Daten von fünf privaten, in Kooperation mit dem Copernicus-Programm stehenden Unternehmen in unterschiedlichen Formen bereitgestellt. Diese als Data and Information Access Services (DIAS) bezeichneten Plattformen stellen unverarbeitete und abgeleitete Daten sowie Werkzeuge zur Analyse zur Verfügung [7]. Da die DIAS-Plattformen kommerziell betrieben werden, müssen einige Dienste und Werkzeuge bezahlt werden, während Nutzer*innen sich lediglich am Copernicus Open Access Hub oder

EUMETCast registrieren müssen. Zu erwähnen ist, dass die DIAS-Plattformen Zugriff auf die gesamte Historie gestatten. Aus dem Copernicus Open Access Hub lassen sich nur aktuelle Datensätze synchron beziehen. In der Regel müssen Daten, welche älter als einen Monat sind, aus dem Landzeitarchiv wiederhergestellt werden. Dieser Vorgang kann einige Zeit in Anspruch nehmen [10].

2.3 Überschwemmungsmonitoring

Um Wasserflächen und damit auch überflutete Areale auf Radarbildern zu erkennen, können die Reflektionseigenschaften von Wasserflächen genutzt werden. Da Wasser eine sehr niedrige Rauigkeit besitzt, kommt es beim Aufprall eines Radarsignals zu einer spiegelnden Reflektion und nur ein sehr geringer Teil des Signals wird zum Empfänger zurückgeworfen. In den resultierenden Bildern äußert sich dieser Umstand in niedrigen Reflektivitätswerten. Um die Areale mit niedrigen Reflektionswerten zu detektieren, können Verfahren genutzt werden, welche aus den Histogrammen der Bilder einen Schwellwert ermitteln. Um die Ergebnisse einer solchen Schwellwertbestimmung zu verbessern, sollten die Radardaten zusätzlich kalibriert werden. So kann die genaue Kenntnis über die tatsächliche Flugbahn des Satelliten dazu beitragen, die geografische Genauigkeit zu verbessern. Diese kann zusätzlich durch Verfahren wie die Diffentialentzerrung gesteigert werden, die die durch das Relief entstandenen Lagefehler ausgleicht [1]. Die radiometrische Genauigkeit kann gesteigert werden, indem zum Beispiel thermisches Rauschen aus den Daten entfernt wird und die Reflektivitätswerte zum sogenannten σ_0-Wert umgerechnet werden. Dieser repräsentiert den Querschnitt der Reflektivität für eine normierte Fläche am Boden [20]. Dieses Maß erlaubt zudem das Vergleichen unterschiedlicher Radaraufnahmen. Außerdem sollte ein Speckle-Filter zum Einsatz kommen. Dieser reduziert körnige Bildstrukturen, welche auf homogenen Flächen in Radarbildern auftreten und die rechnerische Bildauswertung erschweren können. Mit unterschiedlichen Speckle-Filtern können verschiedene Bildeigenschaften in unterschiedlicher Weise erhalten oder verändert werden. Die ESA schlägt hier die Nutzung eines Lee-Sigma Filters vor [1, 18]. Auf Basis des Schwellwertes kann eine Binärisierung der Bilder durchgeführt werden. Die beiden entstehenden Werte repräsentieren überflutete beziehungsweise trocken liegende Areale [18]. Eines dieser Schwellwertverfahren wurde von Nobuyuki Otsu vorgeschlagen. Bei diesem Verfahren werden alle Werte eines Histogramms durchlaufen. Jeder dieser Werte teilt das Histogramm in zwei Gruppen und bildet so einen Schwellwert. Jener Wert, welcher die gewichtete Varianz zwischen der Klassen maximiert, wird als optimaler Grenzwert angesehen [27]. Die Binärisierung kann direkt auf Basis der Radaraufnahme der Überflutung oder auf abgeleiteten Daten erfolgen. So können zum Beispiel Radaraufnahmen vor dem Überschwemmungsereignis σ_0^f mit überflutungsfreien Referenzaufnahmen σ_0^r zum Normalized Difference Sigma-Naught Index (NDSI) kombiniert werden [30]. Dabei werden die Reflektivitätswerte von zwei unterschiedlichen Zeitpunkten zum NDSI verrechnet, welcher als Maß für die Stärke der Veränderung interpretiert werden kann.

$$NDSI = \frac{\sigma_0^f - \sigma_0^r}{\sigma_0^f + \sigma_0^r} \tag{1}$$

Dieses Maß bewegt sich zwischen -1 und 1, wobei Werte um 0 für identische Reflektionswerte an beiden Zeitpunkten und daher für geringe Veränderung stehen. Aufgrund der Reflektionseigenschaften von Wasserflächen deuten Werte nahe -1 auf überflutete Areale hin [30]. Die vielen und teilweise zeitintensiven Prozessierungsschritte können, je nach Größe des zu untersuchenden Areals, viel Zeit und Rechenleistung in Anspruch nehmen.

2.4 Application Programming Interfaces

Schnittstellen sind gemeinsame Grenzen zwischen funktionalen Einheiten, über die mittels vorgegebener Kommunikationswege Informationen ausgetauscht werden können [13]. Ein Application Programming Interface erlaubt also den Austausch von Informationen zwischen zwei unterschiedlichen Programmen. Genauer ausgedrückt erlaubt eine API einem Programm Funktionen und Ressourcen eines anderen Programms zu nutzen. Die Nutzung von APIs erlaubt die Modularisierung von Anwendungen in voneinander unabhängige und untereinander austauschbare Module [13]. Bei diesen kann es sich zum Beispiel um Web-Anwendungen handeln, welche ihre Informationen über standardisierte Protokolle, zum Beispiel HTTP, über das Internet bereitstellen. Mit Requests können diese Informationen über die API angefragt werden. Die API beantwortet einen Request mit einem Response, welcher die im Request spezifizierten Informationen enthält.

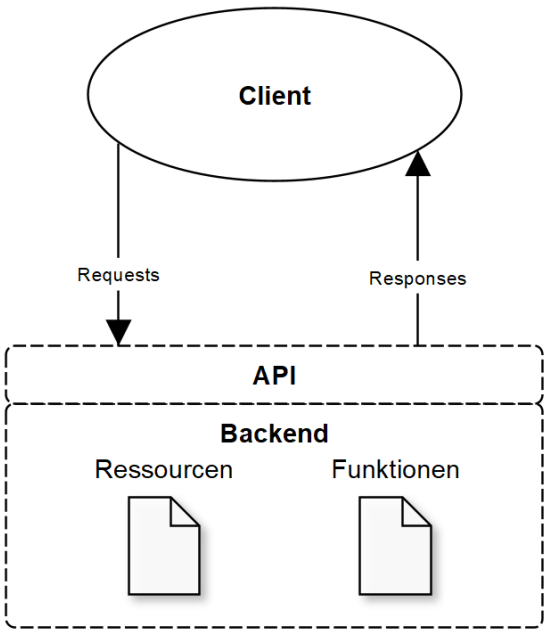

Abbildung 3: Modell einer Webanwendung mit API (eigene Darstellung)

Moderne Web-Anwendungen werden häufig nach dem REST Architekturstil entwickelt. Dieser beschreibt eine Reihe von Einschränkungen und Vereinbarungen für verteilte Hypermedia-Systeme wie das Internet [16]. Durch die Umsetzung der REST-Prinzipien sollen das HTTP und auf diesem basierende Infrastrukturen optimal genutzt sowie Webanwendungen so skalierbar, ausfallsicher, transparent und zuverlässig wie möglich werden [14, 16]. Ein zentraler Begriff ist der der Ressourcen. Eine Ressource ist eine abstrakte Datenstruktur, welche in verschiedenen konkreten Repräsentationen vorliegen kann. Typische Formate für Repräsentationen sind XML, JSON oder HTML. Ressourcen haben eine eindeutige URI, welche häufig die Gestalt eines URL annimmt. Die Interaktionen mit Ressourcen erfolgen mit den HTTP-Methoden `GET`, `POST`, `PUT` und `DELETE` [16]. Als RESTful bezeichnete Systeme sollen zudem stateless operieren. Dies bedeutet, dass alle für die Beantwortung eines Requests nötigen Informationen im Request selbst enthalten sind. Zusätzlich hängt der Erfolg oder Misserfolg eines Requests nicht von dem Ergebnis eines anderen Requests ab [14, 16]. Allerdings sollen Responses zwischengespeichert werden können, um häufig auftretende Requests schneller beantworten zu können [14]. Die REST-Prinzipien können um die des HATEOAS-Architekturstil erweitert werden. Nach diesem sollte eine API selbstbeschreibend sein. Sämtliche Interaktionsmöglichkeiten mit Ressourcen sollten mit den Repräsentationen verknüpft sein. Zudem sollte jede Ressource Verknüpfungen zu anderen Ressourcen enthalten. Ein Client benötigt also nur Zugang zu einer Ressource, um alle anderen Ressourcen zu erreichen. Dies Bedeutet auch, dass der Client nicht a-priori Kenntnis über die Details der API haben muss, um mit ihr zu interagieren [16].

2.4.1 Rich Data Interfaces

Der Begriff Rich Data Interface kann unter zwei Gesichtspunkten betrachtet werden. Einerseits kann darunter eine API verstanden werden, welche reich an Interaktionsmöglichkeiten ist, den Nutzer*innen also umfangreiche Funktionen zur Interaktion mit den durch die von der API angebotenen Ressourcen ermöglicht. Andererseits könnten die Ressourcen selbst reich an Informationen sein. Reichhaltige Ressourcen können zum Beispiel analysebereite oder interpretationsfähige Daten sein. Unter analysebereiten Daten werden verstanden welche gemäß einer möglichst kleinen Menge von Anforderungen vorprozessiert wurden und in der Folge mit minimalem Aufwand für weitere Analysen verwendet werden können. Interpretationsfähige Daten sind Daten welche durch ein gut dokumentiertes und gebräuchliches Verfahren prozessiert wurden und direkt von Menschen interpretiert werden können [17].

2.5 OGC und OGC Standards

Das Open Geospatial Consortium (OGC) widmet sich der Aufgabe die Entwicklung von internationalen Standards und unterstützenden Diensten, welche die Interoperabilität im Bereich der Geodaten verbessern, voranzutreiben. Das OGC soll dabei offene Systeme und Techniken verbreiten, welche es erlauben, Dienste und Prozesse mit Raumbezug in Kreisen der Informatik

zu verbreiten und die Nutzung von interoperabler und kommerzieller Software zu fördern [25]. Dabei wird versucht, möglichst viele Akteure aus Wissenschaft, Wirtschaft und Verwaltung zu beteiligen, um Standards zu schaffen, welche auf möglichst breitem Konsens basieren. Zudem werden Mechanismen zur Zertifizierung von standardkonformen Software-Lösungen angeboten [25].
Das OGC formuliert Standards für unterschiedliche Themenbereiche. In Standards aus dem Bereich Data Models und Encodings werden Daten- und Datenaustauschformate definiert. Standards, welche Webdienste und Schnittstellen zum Austausch von Geodaten beschreiben, werden dem Bereich Services und APIs zugeordnet. Der OGC API - Processes - Part 1: Core Standard gehört in diesen Bereich. Die Bereiche Discovery und Containers enthalten Standards für die Speicherung von Geodaten sowie die Auffindbarkeit und Durchsuchbarkeit dieser. Ein weiterer Bereich widmet sich Standards zum Thema Sensornetzwerke [26].

2.6 OGC API - Processes - Part 1: Core

Der OGC API - Processes - Part 1: Core Standard soll das Bereitstellen von aufwendigen Prozessierungsaufgaben und ausführbaren Prozessen, welche über eine Web API von anderen Programmen aufgerufen und gestartet werden können, unterstützen [29]. Der Standard ist dabei von Konzepten des OGC Web Processing Service 2.0 Interface Standards beeinflusst und bedient sich des REST Architekturstils sowie der Java Script Object Notation (JSON).
Der Aufbau des OGC API - Processes - Part 1: Core Standards orientiert sich am OGC Spezifikationsmodell. Dieses beschreibt die modularen Komponenten eines Standards und wie diese miteinander in Verbindung stehen [4]. Das Spezifikationsmodell definiert einen Standard als Teillösung eines Entwicklungsproblems. Diese Teillösung limitiert die Anzahl an möglichen Implementierungen. Ziel ist die Harmonisierung der Implementierungen und so die Interoperabilität zu steigern. Der OGC API - Processes - Part 1: Core Standard formuliert Requirements sowie Recommendations und fasst diese zu Requirements Classes zusammen, welche wiederum ein Standardisierungsziel beschreiben. Requirements beschreiben Eigenschaften oder Vorgehensweisen, die die Implementierung umsetzen muss, um standardkonform zu sein. Recommendations sind hingegen nicht verpflichtend, beschreiben aber aus Sicht der Autoren empfehlenswerte Eigenschaften oder Vorgehensweisen [4]. Jedes Requirement kann mit einem ebenfalls im Standard definierten Conformance-Test-Case überprüft werden. Diese Tests können zu Conformance-Test Classes zusammengefasst werden, welche alle Tests zum Prüfen einer Requirements Class umfassen. Alle vom Standard definierten Conformance-Test Classes werden im Conformance-Suit zusammengefasst [4]. Erfüllt eine Implementierung alle im Conformance-Suit definierten Tests, kann sie mit einem Certificate of Conformance für die implementierten Requirements Classes versehen werden. Requirements, Recommendation und Conformance-Suit bilden gemeinsam eine Spezifikation, welche nach der Anerkennung durch ein legitimes Gremium wie das OGC als Standard anerkannt werden. Der OGC API - Proces-

ses - Part 1: Core ist seit August 2021 als solcher anerkannt [4, 29]. Der OGC API - Processes - Part 1: Core Standard definiert sieben Requirements Classes, welche Endpoints sowie deren Funktionen, Responses und mögliche Fehlersituationen umfassen. Die Requirements Class Core beschreibt dabei die Kernfunktionalitäten, welche von standardkonformen Implementierungen umgesetzt werden. Die in der Requirements Class Core beschriebenen Funktionalitäten könnten auch mit dem OGC OGC Web Processing Service 2.0 umgesetzt werden. Da dem Nutzer*innen mit diesen Kernfunktionalitäten Ressourcen zugänglich gemacht werden sollen, werden in den Requirements Classes JSON und HTML Repräsentationen dieser Ressourcen in JSON und HTML definiert [29]. Die Requirements Class OpenAPI Specification 3.0 definiert, wie implementierte APIs mithilfe der OpenAPI 3.0 Spezifikation beschrieben und dokumentiert werden sollen [29]. Da der Standard primär, aber nicht ausschließlich, zum Breitstellen von Diensten aus dem Bereich der Geoinformatik genutzt werden soll, wird in der Requirements Class OGC Process Description die Nutzung des OGC Processes Description Formats zum Beschreiben von angebotenen Prozessen empfohlen [29]. Die Requirements Classes JobList, Callback und Dismiss beschreiben zusätzliche Ressourcen und Interaktionsmöglichkeiten mit den auszuführenden Instanzen eines Prozesses, welche als Jobs bezeichnet werden [29].

2.7 Evaluationskriterien

Die Evaluation einer Implementierung einer API kann unter verschiedenen Gesichtspunkten erfolgen. So können technische Aspekte wie Performanz, Skalierbarkeit, Stabilität und Wartbarkeit untersucht werden. Zu dieser technischen Bewertung einer API kann auch das Überprüfen der Standardkonformität gezählt werden. Diese kann durch einen ebenfalls zu implementierenden Unit-Test teilweise überprüft werden, da manche Testfälle des Test-Suits des Standards automatisch überprüft werden können [29]. Der Fokus der in dieser Arbeit durchgeführten Evaluation soll jedoch die Benutzbarkeit und Benutzerfreundlichkeit der Implementierung sein. Dafür können die von Jakob Nielsen 1993 aufgestellten Heuristiken verwendet werden. Dabei handelt es sich um zehn Heuristiken, unter denen eine Anwendung hinsichtlich ihrer Benutzbarkeit betrachtet werden kann. Die Heuristiken decken unter anderem die Themenfelder Verständlichkeit, Fehlerbehandlung, Dokumentation und Konsistenz ab. Da sich einige der vorgeschlagenen Heuristiken auf grafische Benutzeroberflächen beziehen, werden sie im Rahmen dieser Arbeit nicht betrachtet. Zusätzlich sollte bemerkt werden, dass es sich bei einer heuristischen Evaluation um eine subjektive Einschätzung des Evaluierenden handelt [23, 24].

3 Implementierung

3.1 Softwarestack

Die prototypische Entwicklung eines Rich Data Interface für Copernicus-Daten erfolgte im Rahmen dieser Arbeit mit der Programmiersprache Python. Diese ist nicht nur aufgrund ihrer Einfachheit vorteilhaft, sondern erlaubt auch den Zugriff auf eine große Zahl von Packages für die unterschiedlichsten Anwendungsfälle. Weite Teile des Rich Data Interface wurden mithilfe des `flask`-Frameworks umgesetzt. Dieses erlaubt das schnelle Entwickeln einer leichtgewichtigen API. Neben Python Standard-Bibliotheken finden nur wenige zusätzliche Packages Verwendung. Um zu Anfragen passende Sentinel-1 Datensätze aus dem Copernicus Open Access Hub zu finden kommt das Package `sentinelsat` zum Einsatz. Das Herunterladen der Sentinel-1 Datensätze erfolgt ebenfalls mit diesem Package. Die Vorprozessierung der Sentinel-1 Datensätze werden mit dem Package `snappy` durchgeführt. Dieses ist ein Wrapper für die Sentinel Application Platform (SNAP), welche im Rahmen des Copernicus-Programmes kostenlos zur Verfügung gestellt wird. Mithilfe von `snappy` lassen sich die Funktionen der Sentinel-1 Toolbox der SNAP in Python verwenden. Statistische Auswertungen der Radarbilder werden mit dem Package `skimage` durchgeführt. Für das Zuschneiden und mathematische Kombinieren der Radarbilder kommen das `osgeo` Package und das Modul `gdal` zum Einsatz. Die Versionierung erfolgt mit Git und der Plattform GitHub.

3.2 Struktur

Die Anwendung ist in vier Python-Scripte aufgeteilt. Im `api.py` Script ist die API der Anwendung definiert. Die API setzt die Requirements-Classes Core, OGC Process Description, JSON, HTML, OpenAPI 3.0, Job List und Dismiss um. Die geordnete Abarbeitung der angelegten Jobs wird vom Script `processing.py` nach dem Modell der Queue gesteuert. Dies bedeutet, dass zeitgleich nur ein Job ausgeführt wird. Nach dessen Beendigung wird der älteste angelegte Job gewählt und bearbeitet. Die eigentlichen Prozesse sowie Hilfsfunktionen befinden sich im `utils.py` Script. Dazu zählen auch die eigentlichen Prozesse sowie Parser für ihre Eingabedaten. Das `test.py` Script kann dazu genutzt werden, um die Stabilität und Standardkonformität der API zu testen und enthält den Test-Suit.
Diese Scripte verwalten Dateien in einem einfachen Verzeichnissystem. Templates für statische Ressourcen befinden sich im *Templates*-Verzeichnis. In den Unterverzeichnissen *HTML* und *JSON* befinden sich die Repräsentationen von Ressourcen als `.html` und `.json`-Dateien. Das *Processes*-Unterverzeichnis enthält die Beschreibungen der von der Anwendung angebotenen Prozesse. Die Anwendung erlaubt das persistente Hinterlegen von Sentinel-1 Datensätzen. Diese Datensätze können im *Data*-Verzeichnis abgelegt werden. Jeder Sentinel-1 Datensatz enthält eine `.kml`-Datei, welche Metadaten zum Datensatz enthält. Diese werden im *Coverage*-Unterverzeichnis abgelegt. Jeder angelegte Job, also jede auszuführende Instanz eines Prozes-

ses erhält ein einzigartiges Verzeichnis innerhalb des Verzeichnisses *Jobs*. In diesem Verzeichnis befinden sich eine `status.json` und eine `job.json` sowie weitere für die Bearbeitung des Jobs benötigte Dateien. Neben diesen Dateien enthält jedes *Job*-Verzeichnis ein *Results*-Unterverzeichnis in dem die Ergebnisse des jeweiligen Jobs abgelegt werden (siehe Abbildung 4) [28].

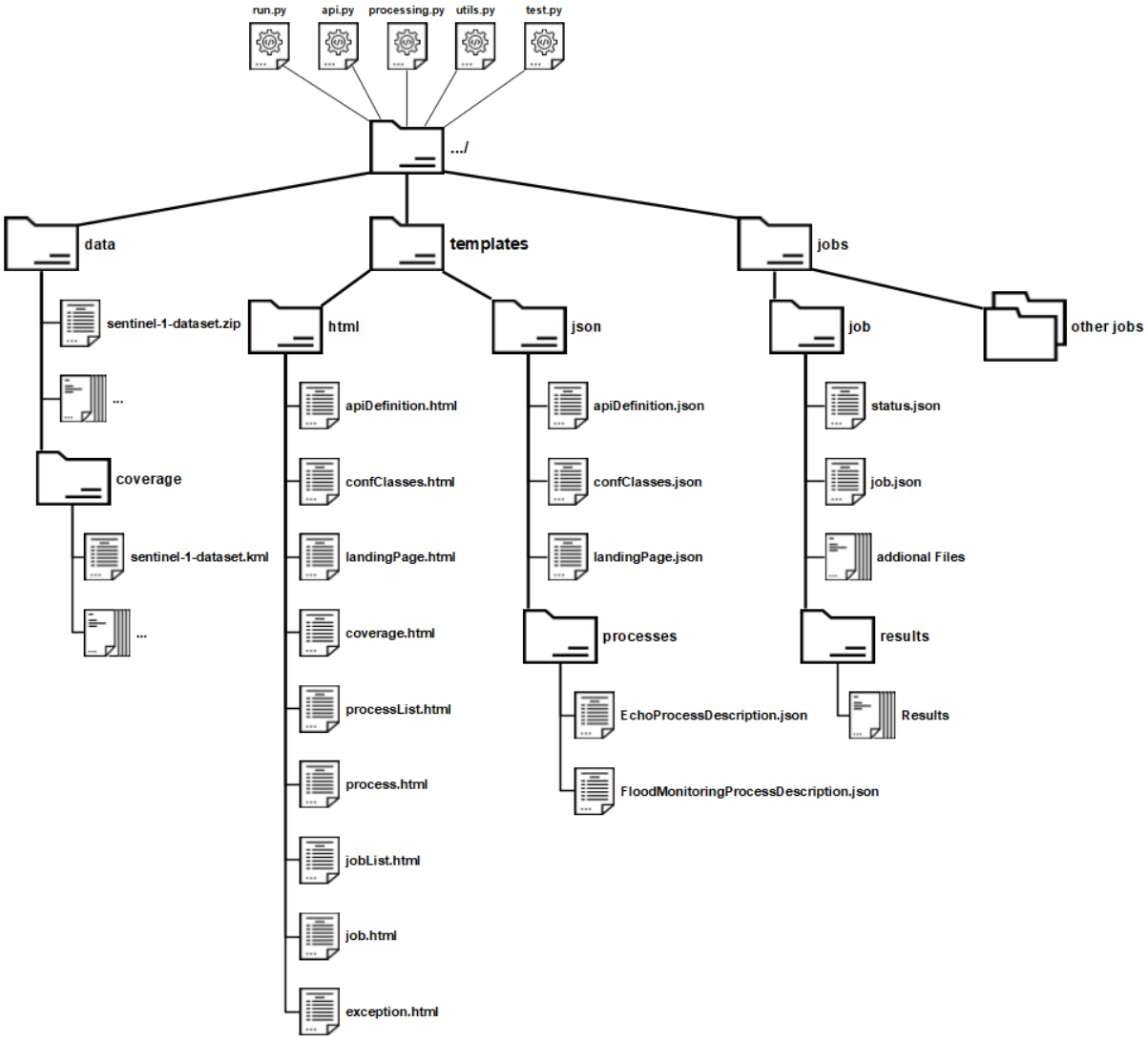

Abbildung 4: Struktur der prototypischen Implementierung [28]

3.3 Ressourcen

Ressourcen sind die über die Endpoints der API bereitgestellten Informationen und Dateien. Sie können in unterschiedlichen Repräsentationen vorliegen. Die Ressourcen werden vom OGC API - Processes - Part 1: Core Standard mittels `.yaml`-Schemadateien beschrieben. Diese definieren neben den Bezeichnungen für Attribute auch ihre Datentypen und ob diese optional sind oder nicht. Die meisten angebotenen Ressourcen können um Media-Type `text/html` oder im Media-Type `application/json` angefragt werden. Um aus diesen Dateien einen Respon-

se zu generieren, werden `.html`-Dateien zuvor mit der Methode `render_template()`, welcher auch dynamische Inhalte übergeben werden können, gerendert, während `.json`-Dateien zunächst geladen und anschließend mit der Methode `jsonify()` bearbeitet werden. Beide genannten Methoden geben ein Response-Objekt zurück, welches zusammen mit Headern und einem HTTP-Statuscode versandt werden kann. Die meisten Ressourcen enthalten zudem eine Verknüpfung zu sich selbst mit der Relation `self` und gegebenenfalls eine Verknüpfung zur Ressource im jeweils anderen Media-Type mit der Relation `alternate`.

3.4 Encodings

In der Requirements Class JSON wird definiert welche Ressourcen im `JSON`-Format, also dem Media-Type `application/json`, angefragt werden können. Dazu gehören alle Responses der Endpunkte API Landing Page, API-Definition, Conformance Deklaration, Process List, Prozess Description, Prozess Execution und Job Status, welche mit dem HTTP-Statuscode 200 versandt werden. Da die prototypische Implementierung auch die Endpunkte Job List und Coverage bereitstellt, können die korrespondierenden Ressourcen auch im Media-Type `application/json` angefragt werden.

In der Requirements Class HTML werden analog zur Requirements Class JSON jene Ressourcen definiert, welche im `HTML`-Format, also im Media-Type `text/html` angefragt werden können. Jedoch entfällt in dieser Requirements Class die Einschränkung auf bestimmte Endpunkte und alle Responses, welche mit dem HTTP-Statuscode 200 versandt werden, müssen den Media-Type `text/html` unterstützen. Stellen Endpoints ihre Ressourcen sowohl den Media-Type `application/json` als auch `text/html` zur Verfügung, so können Nutzer*innen diesen über den optionalen Parameter `f` spezifizieren. Wird kein Media-Type über diesen Parameter spezifiziert, so wird standardmäßig der Media-Type `text/html` verwendet.

3.5 Konfiguration der HTTP Version

Die Umsetzung des Requirements HTTP 1.1 aus der Requirements Class Core verlangt, dass die API exklusiv das HTTP 1.1 unterstützt. Das `flask`-Framework nutzt standardmäßig das HTTP 1.0. Teil des Frameworks ist die WSGI-Bibliothek `Werkzeug`, welche das Implementieren von Webanwendungen erlaubt. Um die verwendete HTTP-Version von 1.0 auf 1.1 umzustellen, müssen Variablen in `Werkzeug` angepasst werden. Nach dem Import der Module `WSGIRequestHandler` und `BaseWSGIServer` wurde in beiden die Version des HTTP Protokolls durch Änderung von Variablen angepasst [28]. In diesem Requirement werden zudem alle HTTP-Statuscodes gelistet, die Nutzer*innen von einer standardkonformen Implementierung mindestens erwarten können. Der Standard erlaubt die Nutzung weiterer HTTP-Statuscodes [29].

Tabelle 2: Vorgesehene HTTP-Statuscodes [29]

HTTP-Statuscode	Bedeutung
200	Ok
201	Created
204	No Content
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
410	Gone
429	Too Many Requests
500	Internal Server Error
501	Not Implemented

Zusätzlich enthalten die meisten Responses einen `link`-Header, welcher eine Verknüpfung mit der Relation `self` enthält. Zusätzlich ist ein `resource`-Header Teil eines Responses. Dieser identifiziert die angefragte Ressource zusätzlich.

3.6 Endpoints der API

3.6.1 API Landig Page Endpoint

Der API Landing Page Endpoint kann über den nachstehendem URL angefragt werden und liefert als Ressource die API Landing Page.

`http://HOST:PORT/?f=<MEDIA-TYPE>`

Der API Landing Page Endpoint ist Teil der Requirements Class Core. Die einzig zulässige HTTP-Methode für diesen Endpoint ist die HTTP-Get Methode. Die API Landing Page kann als Eintrittspunkt zu allen anderen Funktionalitäten der Anwendung bezeichnet werden. Sie enthält Verknüpfungen zu den Endpoints, API Landing Page, API-Definition, Conformance, Process List, Process Description, Job List und Coverage. Die API Landing Page kann in den Media-Types `text/html` und `application/json` abgerufen werden. Ein an den API Landing Page Endpoint gerichteter Request wird zunächst auf die verwendetet HTTP-Methode hin überprüft. Handelt es sich nicht um die HTTP-Get Methode wird der HTTP-Statuscode 405 als Response versandt. Im nächsten Schritt erfolgt die Prüfung des angefragten Media-Types. Enthält der Parameter `f` einen nicht unterstützten Media-Type wird als Response der HTTP-Statuscode 406 versandt. Wird der Media-Type nicht spezifiziert oder `text/html` angefragt wird die `landingPage.html` an die Funktion `render_template()` übergeben, um ein

Response-Objekt zu erzeugen. Spezifiziert der Request den Media-Type `application/json` wird das Response-Objekt aus der `landingPage.json` erzeugt. Dieses wird mit dem HTTP-Statuscode 200 sowie den Headern `resource` und `link` versandt. Ersterer enthält den Wert `landingPage`. Kommt es während der Ausführung zu einem Fehler im Programmcode wird mit dem HTTP-Statuscode 500 geantwortet (siehe Anhang 1) [28, 29].

3.6.2 API Definition Endpoint

Der API-Definition Endpoint kann über den nachstehenden URL angefragt werden und liefert als Ressource die API-Definition.

`http://HOST:PORT/api?f=<MEDIA-TYPE>`

Auch der API-Definition Endpoint ist Teil der Requirements Class Core. Auch für diesen Endpoint ist die einzig zulässige HTTP-Methode Get. Die API-Definition enthält detaillierte Informationen zur API. In ihr sind alle verfügbaren Endpoints mit ihren Parametern und Responses aufgeführt. Auch die API-Definition kann in den Media-Types `text/html` und `application/json` abgerufen werden.

Requests an den API-Definition Endpoint werden ebenfalls zunächst auf die verwendete HTTP-Methode hin überprüft. Wird nicht HTTP-Get verwendet, wird als Response der HTTP-Statuscode 405 versandt. Wird im Request mittels des Parameters `f` ein nicht unterstützter Media-Type angefragt, wird ein Response mit dem HTTP-Statuscode 406 versandt. Requests welche keinen oder einen unterstützten Media-Type spezifizieren, lösen die Generierung eines Response-Objektes aus entweder der `apiDefinition.html` oder `apiDefinition.json` aus. Der `resource`-Header erhält den Wert `apiDefinition`. Dieser wird als Teil des Responses mit dem HTTP-Statuscode 200 versandt. Ebenfalls Teil dieses Responses ist ein `link`-Header. Auch dieser Endpoint fängt Fehler bei der Ausführung mit dem Versenden des HTTP-Statuscodes 500 ab (siehe Anhang 2) [28, 29].

3.6.3 Conformance Declaration Endpoint

Ein weiterer zur Requirements Class Core gehörender Endpoint ist der Conformance Declaration Endpoint. Der Conformance Declaration Endpoint kann über den nachstehenden URL angefragt werden und liefert als Ressource die Conformance Declaration.

`http://HOST:PORT/conformance?f=<MEDIA-TYPE>`

Get ist ebenfalls die einzige zulässige HTTP-Methode für Requests an diesen Endpoint. Die Conformance Declaration enthält Verknüpfungen zu allen von der Anwendung implementierten Requirements Classes und steht in den Media-Types `text/html` und `application/json` zur Verfügung. Wie bereits beschrieben wird ein ankommender Request zunächst auf die von

ihm verwendete HTTP-Methode überprüft und bei nicht Zulässigkeit dieser mit dem HTTP-Statuscode 405 geantwortet. Ebenso erfolgt eine Überprüfung des mit dem Parameter `f` spezifizierten Media-Types, wobei die Anfrage eines nicht unterstützten Media-Types in einem Response mit dem HTTP-Statuscode 406 resultiert. Je nach im Request spezifizierten Media-Type wird das Response-Objekt entweder aus der `confClasses.html` oder der `confClasses.json` generiert. Der `resource`-Header enthält den Wert `conformance`. Response-Objekt, `resource`- und `link`-Header werden mit dem HTTP-Statuscode 200 versandt. Auftretende Fehler werden mit einem Response mit dem HTTP-Statuscode 500 abgefangen (siehe Anhang 3) [28, 29].

3.6.4 Process List Endpoint

Der ebenfalls zur Requirements Class Core gehörende Process List Endpoint kann über den nachstehenden URL angefragt werden.

```
http://HOST:PORT/processes?f=<MEDIA-TYPE>&limit=<INTEGER>
```

Request sind nur mit der HTTP-Methode Get zulässig. Als Ressource erhalten Nutzer*innen eine detaillierte Liste der durch die Anwendung angebotenen Prozesse. In dieser Liste finden sich die Bezeichnungen, Steueroptionen sowie die Ein- und Ausgaben der Prozesse. Die Process Liste kann in den Media-Types `text/html` und `application/json` angefragt werden. Wie bei der Beantwortung von Requests an die bereits vorgestellten Endpoints werden Requests mit unzulässigen HTTP-Methoden oder nicht unterstützten Media-Types, welche ebenfalls mit dem Parameter `f` spezifiziert werden, mit Responses, welche die HTTP-Statuscodes 405 beziehungsweise 406 enthalten, beantwortet. Zusätzlich erfolgt allerdings eine Prüfung des `limit`-Parameters. Dieser kann dazu verwendet werden, die Liste auf eine spezifizierte Anzahl von Objekten einzuschränken. Wird ein Limit kleiner 0 oder größer 10000 spezifiziert wird der Wert auf den Standardwert 10 gesetzt. Unabhängig vom gewählten Media-Type werden zunächst alle Prozess-Beschreibungen aus dem *Processes*-Verzeichnis geladen und in einem Array gespeichert. Soll der Response im Media-Type `text/html` versandt werden, wird dieses mit dem `limit`-Parameter eingeschränkt und dem Renderer zusammen mit der `processList.html` übergeben. Das entstehende Response-Objekt wird zusammen mit `resource`- und `link`-Header sowie dem HTTP-Statuscode 200 versandt. Für die Generierung eines Responses im Media-Type `application/json` wird das auf das spezifizierte Limit eingeschränkte Array zu einem JSON-Objekt hinzugefügt, welches noch um die Verknüpfungen `self` und `alternate` ergänzt wird. Aus diesem Objekt wird ein Response-Objekt generiert, welches zusammen mit den beiden Headern `link`- und `resource` und dem HTTP-Statuscode 200 versandt wird. Der `resource`-Header enthält in beiden Fällen den Wert `processes`. Treten während der beschriebenen Schritte Fehler auf, wird ein Response mit dem HTTP-Statuscode 500 versandt (siehe Anhang 4) [28, 29].

3.6.5 Process Description Endpoint

Unter dem nachstehenden URL kann der Process Description Endpoint angefragt werden.

`http://HOST:PORT/processes/<processID>?f=<MEDIA-TYPE>`

Dieser ist Teil der Requirements Class Core. Welche Prozessdetails zurückgegeben werden hängt vom `processID`-Parameter ab. Alle Prozesse werden über eine eindeutige Bezeichnung, die `processID` gekennzeichnet. Sie können der Prozess Liste entnommen werden. Um eine Prozessbeschreibung anzufragen, darf nur die HTTP-Get Methode verwendet werden. Als Ressource liefert der Endpoint eine detaillierte Beschreibung des im Request spezifizierten Prozesses und enthält Informationen zu den Steueroptionen sowie den Ein- und Ausgaben des Prozesses. Die Beschreibungen sind dabei so strukturiert wie durch die Requirements Class OGC Process Description vorgegeben. Wie die Prozessliste kann auch die Prozessbeschreibung im Media-Type `text/html` oder `application/json` angefragt werden. Jeder Prozess, der von der Anwendung angeboten wird, muss eine eigene Prozessbeschreibung haben. Wird von diesem Endpoint ein Response generiert, werden zunächst die verwendete HTTP-Methode sowie der angefragte Media-Type, welcher über die Parameter `f` spezifiziert wird, überprüft. Ungültige beziehungsweise nicht unterstützte HTTP-Methoden und Media-Types werden durch Responses mit den HTTP-Statuscodes 405 beziehungsweise 406 beantwortet. Im nächsten Schritt wird geprüft ob der über den Parameter `processID` spezifizierte Prozess in der Anwendung registriert ist. Ist dies nicht der Fall wird als Response eine `no-such-process`-Exception mit dem HTTP-Statuscode 404 versandt. Existiert die spezifizierte `processID`, wird die korrespondiere Prozessbeschreibung geladen. Um ein Response-Objekt zu generieren, wird je nach spezifiziertem Media-Type entweder dem Renderer die `process.html` zusammen mit der geladenen Prozessbeschreibung übergeben oder direkt aus dieser ein solches generiert. Das Response-Objekt wird zusammen mit dem HTTP-Statuscode 200 sowie den `resource` und `link` Headern versandt. Der `resource`-Header enthält den Wert `process - <processID>`. Fehler werden durch Responses mit dem HTTP-Statuscode 500 abgefangen (siehe Anhang 5) [28, 29].

3.6.6 Process Execution Endpoint

Die Requirements Class Core definiert auch die Funktionalität zum Ausführen der angebotenen Prozesse. Um Prozesse auszuführen, können Requests mit der HTTP-Methode Post and den nachstehenden URL gestellt werden.

`http://HOST:PORT/processes/<processID>/execution`

Dabei müssen die in der Prozessbeschreibung aufgeführten Inputs, die gewünschten Outputs sowie der Response-Typ im Request enthalten sein. Wird die Instanziierung eines Prozesses mittels eines Requests an diesen Endpoint gestartet, wird zunächst die verwendete HTTP-Methode überprüft. Wird nicht die HTTP-Methode Post verwendet, wird ein Response mit dem

HTTP-Statuscode 405 versandt. Es folgt die Überprüfung des im Request spezifizierten Prozesses. Wird dieser von der Anwendung nicht bereitgestellt, wird mit einer `no-such-process`-Exception und dem HTTP-Statuscode 404 geantwortet. Für jeden von der API angebotenen Prozess existiert ein Parser, welcher die im Request enthaltenen Inputs überprüft. Sind diese valide, kann mit der Erzeugung eines Jobs fortgefahren werden. Andernfalls wird ein Response mit dem HTTP-Statuscode 400 geantwortet. Die eigentliche Instanziierung des Prozesses beginnt mit der Erzeugung einer einzigartigen Zeichenkette, welche den Job identifiziert. Im nächsten Schritt wird das *Job*-Verzeichnis angelegt. Innerhalb des *Job*-Verzeichnisses wird ein Unterverzeichnis für die Ergebnisse angelegt. Es folgt das Anlegen der `status`- und einer `job.json`. In der `status.json` werden alle Informationen wie die `jobID`, `processID`, der Status und Fortschritt sowie die Zeitstempel des Erstellungs-, Start- und Beendigungszeitpunkt gespeichert. Lediglich der Erstellungszeitstempel erhält bereits einen Wert. Der Job erhält initial den Status `accepted`. Die `job.json` enthält ebenfalls die Job- und Prozess-ID sowie die Eingabewerte und die Spezifizierung der Ausgaben. Abschließend wird ein Response-Objekt aus der `status.json` erzeugt und mit dem HTTP-Statuscode 201 versandt. Fehler werden durch Responses mit dem HTTP-Statuscode 500 abgefangen (siehe Anhang 6) [28, 29].

3.6.7 Job List Endpoint

Ein Aufruf des nachstehenden URL liefert als Ressource eine Liste aller angelegten Jobs. Der Endpoint ist in der gleichnamigen Requirements Class definiert.

```
http://HOST:PORT/jobs?f=<MEDIA-TYPE>&limit=<INTERGER>&
type=<TYPE>&processID=<processID>&status=<STATUS>&
datetime=<DATETIME>&minDuration=<INTEGER>&maxDuration=<INTEGER>
```

Die Jobliste kann nur mit der HTTP-Get Methode abgefragt werden. Diese kann mit einer Reihe von zusätzlichen Parametern eingeschränkt werden. Analog zur Prozessliste kann die Jobliste mithilfe des Parameters `f` im Media-Type `text/html` und `application/json` angefragt werden. Request an diesen Endpoint werden zunächst auf die verwendete HTTP-Methode und den spezifizierten Media-Type hin überprüft. Nicht unterstützte HTTP-Methoden oder Media-Types werden mit Responses mit den HTTP-Statuscodes 405 beziehungsweise 406 beantwortet. Im nächsten Schritt werden die optionalen Parameter überprüft. Die Parameter `status`, `processID` und `limit` dienen der Einschränkung der Liste auf Jobs mit einem bestimmten Status oder Jobs, welche einen bestimmten Prozess ausführen oder die Anzahl der angezeigten Jobs limitieren. Zusätzlich kann die Liste mit dem Parameter `datetime` auf Jobs eingeschränkt werden, welche ab oder bis zu einem bestimmten Zeitpunkt oder innerhalb eines Zeitraumes erstellt wurden. Sollen nur gestartete oder abgeschlossene Jobs mit einer minimalen oder maximalen Laufzeit aufgeführt werden, kann die Liste mit den Parametern `minDuration` und `maxDuration` auf diese eingeschränkt werden. Die Generierung eines Responses beginnt mit dem Iterieren über alle angelegten Jobs. Entspricht ein Job den spezifizierten Eigenschaften, wird er zu einem Ar-

ray hinzugefügt. Dieses Array wird im nächsten Schritt entweder in auf das spezifizierte Limit eingeschränkter Form dem Renderer übergeben oder es wird ein Response-Objekt aus einem zuvor instanziierten JSON-Objekt erzeugt. Dieses wird anschließend mit dem HTTP-Statuscode 200, einem den Wert `jobs` enthaltenden `resource`-Header sowie einem `link`-Header versandt. Kommt es während der Generierung des Jobliste zu Fehlern, so wird dies durch einen Response mit dem HTTP-Statuscodes 500 abgefangen (siehe Anhang 7) [28, 29].

3.6.8 Job Endpoint

Die Requirements Class Core definiert einen Job Endpoint, welcher die Jobbeschreibung eines Jobs als Ressource liefert. Dieser kann unter anderem entnommen werden, ob ein und wann ein Job gestartet wurde, in welchem Bearbeitungsschritt er sich befindet oder ob der Job bereits erfolgreich abgeschlossen oder gescheitert ist. Requests an den Job Endpoint können an den nachstehenden URL gestellt werden.

`http://HOST:PORT/jobs/<jobID>?f=<MEDIA-TYPE>`

Die Jobbeschreibung kann nur mit der HTTP-Get oder HTTP-Delete Methode angefragt werden. Welche Jobbeschreibung zurückgegeben werden soll, wird über den `jobID`-Parameter spezifiziert. Der Media-Type kann mittels des Parameters `f` spezifiziert werden. Die Ressource Jobbeschreibung steht in den Media-Types `text/html` und `application/json` zur Verfügung. Geht ein Request an diesen Endpoint ein, erfolgt eine Überprüfung der verwendeten HTTP-Methode, wobei eine nicht unterstütze HTTP-Methode zu einem Response mit dem HTTP-Statuscode 405 führt. Außerdem findet eine Überprüfung des spezifizierten media-Types statt. Die Anfrage eines nicht unterstützten Media-Types führt zu einem Response mit dem HTTP-Statuscode 406. Verwendet ein Request die HTTP-Methode Get und spezifiziert einen existierenden Job, wird die entsprechende `status.json` aus dem *Job*-Verzeichnis geladen. Je nach gewähltem Media-Type wird diese zusammen mit dem `job.html` Template dem Renderer übergeben oder direkt ein Response-Objekt generiert. Dieses wird zusammen mit dem HTTP-Statuscode 200, `resource`- und `link`-Header versandt. Der `resource`-Header enthält dabei den Wert `job - <jobID>`. Verwendet ein Request die HTTP-Delete Methode wird ebenfalls zunächst die status.json aus dem *Job*-Verzeichnis geladen. Ergibt die Überprüfung des Job-Status, dass dieser noch nicht auf `dismissed` gesetzt wurde, erfolgt dies. Zusätzlich wird die im Job-Staus enthaltene Nachricht zu `job dismissed` geändert. Es folgt wie bereits beschrieben die Generierung und Versendung eines Response-Objektes aus dem Job-Status. Lediglich der `resource`-Header enthält nun den Wert `job - <jobID> dismissed`. Dies geschieht auch wenn der Job bereits den Status `dismissed` aufweist. Nur der verwendete HTTP-Statuscode ist in diesem Fall nicht 200 sondern 410. Für beide unterstützten HTTP-Methoden gilt: Existiert die im Request spezifizierte `jobID` nicht, wird mit einer `no-such-job`-Exception geantwortet. Diese wird zusammen mit dem HTTP-Statuscode 404 versandt. Wird ein Response mit dem

HTTP-Statuscode 500 versandt, ist es zu Fehlern bei der Verarbeitung des Requests gekommen (siehe Anhang 8 und 9) [28, 29].

3.6.9 Job Results Endpoint

Die Ergebnisse eines erfolgreichen Jobs können unter nachstehendem URL abgerufen werden. Wie die Ausführung der Prozesse ist auch das Beschaffen der Prozessierungsergebnisse Teil der Requirements Class Core.

`http://HOST:PORT/jobs/<jobID>/results`

Die bereitgestellten Ressourcen variieren je nach Prozess, gewähltem Output sowie dessen Transmission-Mode, wobei die prototypische Implementierung nur `value` unterstützt und Response-Type, `document` oder `raw`. Sie können mit der HTTP-Get Methode abgerufen werden. Die Verwendung einer anderen HTTP-Methode führt zu einem Response mit dem HTTP-Statuscode 405. Wird ein Request akzeptiert, wird zunächst geprüft, ob die `jobID` überhaupt existiert und der Job bereits den Job-Status `successful` aufweist. Existiert der spezifizierte Job nicht, wird mit einer `no-such-job`-Exception und dem HTTP-Statuscode 404 geantwortet. Wird ein existierender Job angefragt, werden die `job.json` und `status.json` aus dem Job-Verzeichnis geladen. Weißt der spezifizierte Job den Status `failed` auf, wird mit einer `job-failed`-Exception und dem HTTP-Statuscode 404 geantwortet. Ist der Job noch nicht gestartet oder noch nicht beendet, wird eine `results-not-ready`-Exception und dem HTTP-Statuscode 404 geantwortet. Die Ergebnisse eines erfolgreich ausgeführten Jobs können mit dem Response-Typ `raw` oder `document` versandt werden. Welcher Response-Typ zur Anwendung kommt, wird beim Erzeugen des Jobs festgelegt. Der Response-Typ `raw` löst eine Versendung der Ergebnis-Dateien aus dem *Results*-Verzeichnis des Jobs mit dem HTTP-Statuscode 200 aus. Handelt es sich bei dem Job um eine Instanz des `Echo`-Prozesses wird die `results.json` versandt. Ist der Job eine Instanz des `FloodMonitoring`-Prozesses wird je nach bei Erzeugung spezifiziertem Ergebnis die `bin.tif`, die `ndsi.tif` oder eine `.zip`-Datei, welche beide `.tif`-Dateien enthält, versandt. Wurde bei Erzeugung des Jobs der Response-Typ `document` spezifiziert, wird ein Ergebnis-Dokument erzeugt und mit dem HTTP-Statuscode 200 versandt. Instanzen des Echo-Prozesses generieren eine leicht veränderte Form der `results.json` und versenden diese mit dem HTTP-Statuscode 200. Jobs, bei denen es sich um Instanzen des `FloodMonitoring`-Prozesses handelt, encodieren die bei Erzeugung spezifizierten Ergebnisse zunächst in das Base64-Format. Die so entstehende Zeichenkette wird zusammen mit einer Verknüpfung zur korrespondierenden `.tif`-Datei zu einer `.json`-Datei hinzugefügt. Diese wird mit dem HTTP-Statuscode 200 versandt. Wie bei allen Endpoints werden etwaige Fehler durch das Versenden eines Responses mit dem HTTP-Statuscode 500 abgefangen (siehe Anhang 10) [28, 29].

3.7 Dokumentation der API

Für die Dokumentation der API soll gemäß der Requirements Class OpenAPI 3.0 der OpenAPI 3.0 Standard genutzt werden [29]. Dieser definiert eine sprachenunabhängige Schnittstellenbeschreibung für Web APIs [19]. Diese Schnittstellenbeschreibung ist dabei so formuliert, dass sie sowohl menschen- als auch maschinenlesbar ist. Ziel dieser Dokumentationsweise ist es, die API so leicht verständlich wie möglich zu machen und so die Benutzerfreundlichkeit zu steigern [19]. Die Dokumentation erläutert sämtliche Endpoints mit den möglichen Responses und zu erwartenden HTTP-Statuscodes. Zusätzlich sind die Schemata der bereitgestellten Ressourcen einsehbar. Die Dokumentation der prototypischen Implementierung erfolgte mit `Swagger` über die Webseite Swagger-Hub. Die dort gepflegte Dokumentation wurde im JSON- und HTML-Format exportiert. Diese Dateien dienen der prototypischen Implementierung als Repräsentationen der Ressource API-Definition [28, 29].

3.8 Prozesse

3.8.1 Echo Prozess

Da eine standardkonforme Anwendung mindestens einen testbaren Prozess anbieten muss, ist der `Echo`-Prozess ebenfalls Teil der prototypischen Implementierung [29]. Der `Echo`-Prozess kann nur asynchron ausgeführt werden und die Ergebnisse können nur im Transmission-Mode `value` angefragt werden. Die einzige Eingabe ist eine Zeichenkette, welche nach einer simulierten Bearbeitungszeit auch als Ausgabe dient. Wird eine Instanz des `Echo`-Prozesses gestartet, wird zunächst der Startzeitpunkt in der `status.json` aktualisiert. Es folgt eine Wartezeit von fünf Sekunden nach der die Ergebnisdatei geschrieben wird. Diese enthält die Eingabezeichenkette sowie die Information, dass es sich um ein Echo handelt. Es folgt das Aktualisieren des Beendigungszeitpunktes und das Ändern des Status zu `successful`. Schlägt einer der beschriebenen Schritte fehl, wird der Status auf `failed` gesetzt. Zudem wird vor jedem Schritt überprüft ob der Job abgebrochen wurde, also der Status zu `dismissed` geändert wurde. Ist dies der Fall, wird die Prozessierung abgebrochen (siehe Anhang 13) [28].

3.8.2 Überschwemmungsmonitoring

Die Hauptfunktionalität der prototypischen Implementierung besteht im Überschwemmungsmonitoring. Dies wird durch den `FloodMonitoring`-Prozess realisiert. Auch dieser Prozess kann nur asynchron ausgeführt werden. Die Ergebnisse lassen sich ebenfalls nur im Transmission-Mode `value` abrufen. Zu den Eingaben gehören jeweils ein Datum vor und nach dem Überschwemmungsereignis, Nutzername und Passwort eines gültigen Copernicus Open-Access-Hub Accounts sowie eine Bounding Box des zu untersuchenden Areals. Die Bearbeitung einer Instanz dieses Prozesses beginnt mit dem Aktualisieren des Startzeitpunktes in der `status.json`. Anschließend wird eine `.geojson`-Datei aus der in den Eingaben spezifizierten Bounding Box

generiert. Im nächsten Schritt werden aus den in den Eingaben spezifizierten Zeitpunkten Suchintervalle erzeugt. Nach dem Login beim Copernicus Open-Access-Hub wird zunächst ein zum Zeitpunkt vor dem Überschwemmungsereignis passender Datensatz gesucht. Konnte ein solcher gefunden werden, wird er heruntergeladen und kalibriert. Zur Kalibrierung gehört die geometrische Korrektur des Datensatzes mithilfe der Orbit-Datei. Um die nachfolgenden Kalibrierungsschritte zu beschleunigen, wird der von der Bounding Box definierte Raum ausgeschnitten. Der nächste Kalibrierungsschritt reduziert thermisches Rauschen bevor nachfolgend die σ_0-Werte berechnet werden. Um körnige Bildstrukturen zu reduzieren, wird zusätzlich ein Speckle-Filter auf das Bild angewandt. Dabei handelt es sich um einen Lee-Sigma-Filter mit einer Fenstergröße von fünf mal fünf Pixeln. Dieser reduziert körnige Bildstrukturen ausreichend gut und erhält dabei Eigenschaften wie Kontraste gut [15]. Abschließend erfolgt eine Geländekorrektur auf Basis des Ellipsoiden. Nach erfolgreicher Kalibrierung wird das VV-Band als `.tif`-Datei in das *Results*-Verzeichnis geschrieben. Das Herunterladen und Kalibrieren des Datensatzes nach dem Überschwemmungsereignis erfolgt in gleicher Weise. Aus den kalibrierten Bildern wird im nächsten Schritt der NDSI berechnet und als `.tif`-Datei im *Results*-Verzeichnis gespeichert. Die `ndsi.tif` wird mit der `footprint.geojson` zugeschnitten. Im nächsten Schritt wird mit der von Nobuyuki Otsu vorgeschlagen Methode ein Schwellwert berechnet. Auf Basis dieses Schwellwertes wird die `ndsi.tif` binärisiert und als `bin.tif` im *Results*-Verzeichnis gespeichert. Abschließend wird der Job-Status auf `successful` gesetzt und der Beendigungszeitpunkt aktualisiert. Zwischen jedem Schritt wird geprüft, ob der Job abgebrochen wurde. Ist dies der Fall, wird die Prozessierung abgebrochen. Kommt es zu Fehlern bei der Prozessierung wird der Job-Status auf `failed` gesetzt und die Prozessierung abgebrochen (siehe Abbildung 5 und Anhang 14) [28].

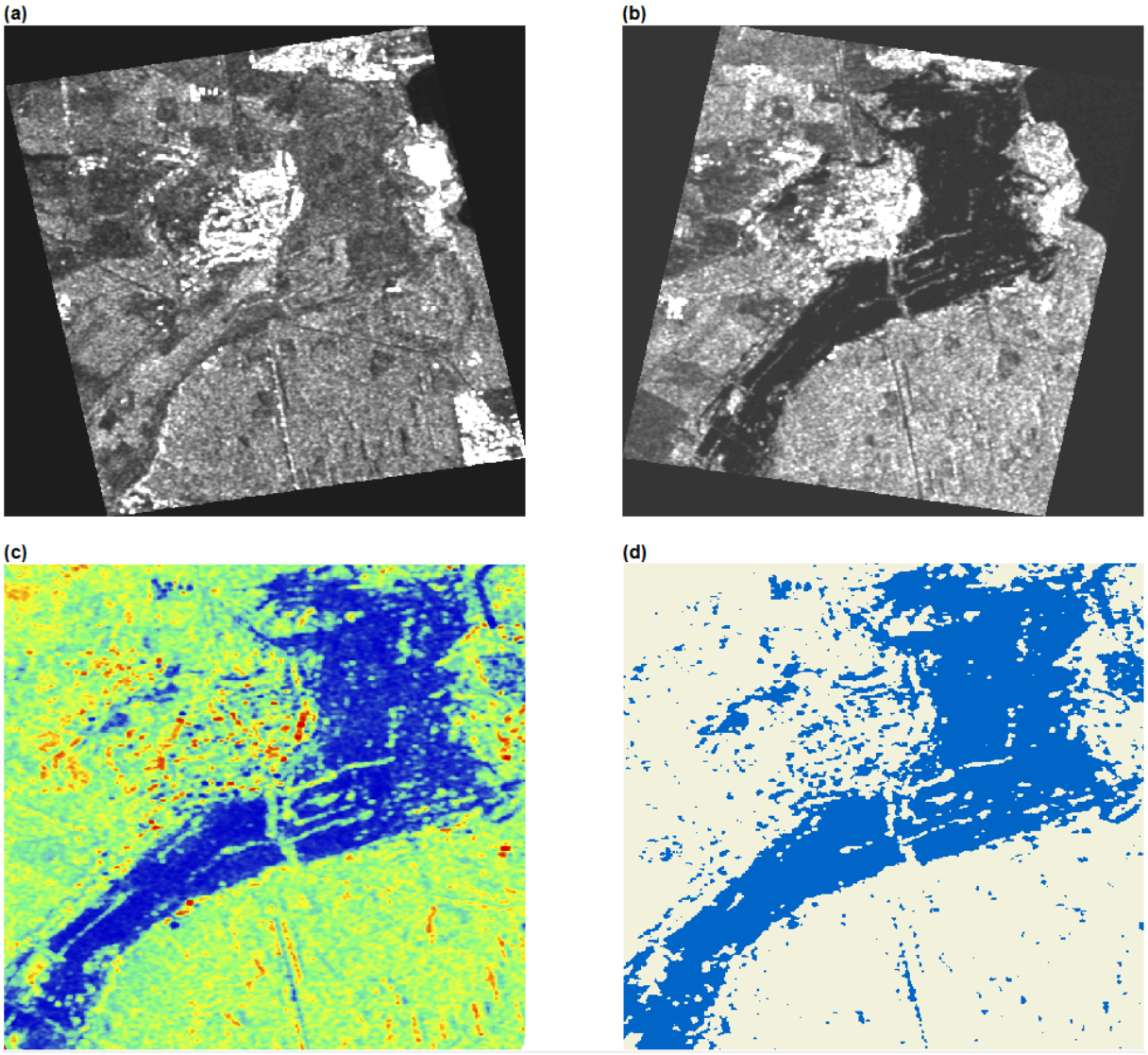

Abbildung 5: Beispielhafte Prozessierungsergebnisse (a) kalibrierte überflutungsfreie Referenzaufnahme (b) kalibrierte Aufnahme des Überschwemmungsereignisses (c) NDSI (d) binäre Überschwemmungsmaske (eigene Darstellung)

3.9 Zusätzliche Funktionalitäten

3.9.1 Coverage Endpoint

Der nicht im Standard definierte Coverage Endpoint kann über den nachstehenden URL mit der HTTP-Get Methode erreicht werden.

```
http://HOST:PORT/coverage?f=<MEDIA-TYPE>
```

Als Ressource liefert er eine Übersicht aller Sentinel-1 Datensätze, welche persistent gespeichert sind. Jobs, welche auf diese Datensätze zugreifen, können schneller abgearbeitet werden, da ein zeitaufwendiges Herunterladen entfällt. Die Coverage Ressource kann nur mit der HTTP-

Get Methode angefragt werden und steht in den Media-Types `text/html` und `application/json` zur Verfügung. Dieser kann mit dem Parameter `f` spezifiziert werden.
Eingehende Requests werden zunächst auf die verwendete HTTP-Methode überprüft. Nicht unterstützte HTTP-Methoden werden mit Responses mit dem HTTP-Statuscode 405 beantwortet. Spezifiziert der Parameter `f` nicht unterstützte Media-Types, so wird ein Response mit dem HTTP-Statuscode 406 versandt.
Nach Eingang eines gültigen Requests werden zunächst alle `.kml`-Dateien aus dem *Coverage*-Verzeichnis geladen. Aus diesen werden der Produktname, das Aufnahmedatum sowie die BBox extrahiert. Diese Informationen werden in Arrays gespeichert. Aus diesen werden entweder direkt Response-Objekte erzeugt oder zusammen mit der `coverage.html` dem Renderer übergeben. Die Response-Objekte werden mit dem HTTP-Statuscode 200 sowie dem Headern `link` und `resource` versandt. Der `resource`-Header erhält den Wert `coverage`. Fehler resultieren in Responses mit dem HTTP-Statuscode 500.
Sollen zusätzliche Datensätze einer Region für einen bestimmten Zeitraum persistent in der Anwendung gespeichert werden, so können diese über das Copernicus Open Access Hub bezogen werden. Die Datensätze müssen anschließend im *Data*-Verzeichnis abgelegt werden. Teil eines jeden Sentinel-1 Datensatzes ist eine `.kml`-Datei. Diese muss im *Coverages*-Verzeichnis abgelegt werden. Von Prozessen heruntergeladene Datensätze werden automatisch persistent und für die Anwendung nutzbar gespeichert (siehe Anhang 11) [28].

3.9.2 Download Endpoint

Um das Herunterladen von großen Bilddaten über einen URL zu erlauben, wurde die prototypische Implementierung um den Download Endpoint ergänzt. Die Verknüpfungen können auch Teil der Prozessierungergebnisse sein, wenn als Response-Typ `document` spezifiziert wurde. Bestimmte Dateien können unter dem nachstehenden URL unter Verwendung der HTTP-Methode Get heruntergeladen werden.

`http://HOST:PORT/download/<jobID>/<requestedFile>`

Nach Überprüfung der verwendeten HTTP-Methode, wobei eine nicht unterstützte HTTP-Methode in einem Response mit dem HTTP-Statuscode 405 mündet, wird die Existenz des spezifizierten Jobs sichergestellt. Existiert dieser nicht, wird eine `no-such-job`-Exception mit dem HTTP-Statuscode 404 versandt. Andernfalls werden die `status.json` und `job.json` aus dem *Job*-Verzeichnis geladen. Der Download-Endpoint stellt in der prototypischen Implementierung nur die aus dem `FloodMonitoring`-Prozess erzeugten Bilddaten zum Download bereit. Ist der spezifizierte Job eine Instanz des `Echo`-Prozesses wird ein Response mit dem HTTP-Statuscode 501 versandt. Es können die `bin.tif` oder die `ndsi.tif` zum herunterladen angefragt werden. Ist eine der beiden Dateien durch den `requestedFile`-Parameter spezifiziert, wird die entsprechende Datei mit dem HTTP-Statuscode 200 versandt. Werden andere Dateien durch

den Request spezifiziert, wird eine `no-such-file`-Exception mit dem HTTP-Statuscode 404 versandt. Responses mit dem HTTP-Statuscode 500 sind das Resultat von Fehlern im Programmablauf (siehe Anhang 12) [28].

3.9.3 Test Suit

Um die Funktionen der API und seine Standardkonformität testen zu können, wurde ein Unit-Test oder test Suit aus dem im Standard beschriebenen Abstract Test Suit abgeleitet und implementiert. Zu bemerken ist dabei, dass der prototypische implementierte Test-Suit lediglich einige maschinentestbare Testfälle abdeckt. Werden nun Änderungen am Programm vorgenommen, kann mit der Ausführung dieses Test-Suits einfach die Funktionalität und Stabilität der API überprüft werden. Bisher sind 35 der 82 Testfälle von der prototypischen Implementierung abgedeckt.

4 Evaluation

4.1 Sichtbarkeit des Systemstatus

Eine Anwendung sollte Nutzer*innen fortlaufend nicht nur über aufgetretene Fehler, sondern über möglichst viele Vorgänge informieren, sobald diese für den Nutzer*innen nützliche Informationen generieren. Dies sorgt dafür, dass Nutzer*innen die Folgen ihrer Interaktionen nachvollziehen können und ihre nächsten Interaktionen auf Basis der gewonnenen Interaktionen planen [22–24]. Konkret sollten Nutzer*innen jederzeit Informationen zum Stand der von ihnen gestarteten Prozesse erhalten. Dies beginnt beim Erzeugen eines Jobs. Hier erhalten Nutzer*innen entweder die Information, dass der Job angelegt wurde oder dass die Eingaben nicht valide waren. Informationen zu erzeugten Jobs können über die Job Liste oder den Job Status beschafft werden. Dort kann der Nutzer*innen einsehen, ob der fragliche Job bereits gestartet wurde, in welchem Bearbeitungsschritt er sich befindet oder ob bereits Resultate abrufbar sind. Zusätzlich kann dem Job-Status auch entnommen werden zu wie viel Prozent der Job bereits bearbeitet ist oder ob Fehler aufgetreten sind. In letzterem Fall kann zusätzlich ermittelt werden, welcher Prozessierungsschritt nicht erfolgreich beendet werden konnte. Allerdings erfolgt die Informationsweitergabe an den Nutzer*innen nicht automatisch, sondern nur auf dessen Initiative. Die Requirements-Class Callback des OGC API - Processes - Part 1: Core beschreibt jedoch einen Mechanismus mit dem Nutzer*innen automatisiert zum Stand der von ihnen gestarteten Prozesse informiert werden können. Dies könnte nach jedem Prozessierungsschritt oder bei einer Statusänderung erfolgen.

4.2 Nutzerkommunikation

Anwendungen sollten in möglichst einfacher und verständlicher Form mit dem Nutzer*innen kommunizieren. Befehle und Benachrichtigungen sollten möglichst in einer Sprache verfasst sein, welche Nutzer*innen direkt verstehen und interpretieren können. Die kommunizierten Konzepte sollten leicht mit der realen Welt verknüpfbar sein [22–24]. Die prototypische Anwendung verwendet Englisch als Systemsprache. Diese kann von einer großen Nutzergemeinschaft gelesen und verstanden werden. Requests werden in einer Form verfasst, welche ebenfalls direkt Kontext und Intention verstehen lässt. So wird bei Abfrage eines Jobs `/results` angehängt, um zu signalisieren, dass die Ergebnisse abgefragt werden sollen. In ähnlicher Weise wird einem Request an einen Prozess `/execute` angehängt, um den Prozess zu starten. Die Interaktion mit der Anwendung erfolgt also in quasi natürlicher Sprache.

4.3 Nutzerkontrolle und Wiedererkennbarkeit

Eine Anwendung sollte Nutzern möglichst viel Freiheit bei der Interaktion mit den bereitgestellten Ressourcen erlauben. Da Fehler passieren können, sollten Nutzer*innen zum Beispiel

in die Lage versetzt werden, durch Interaktionen gestartete Prozesse abzubrechen. Außerdem sollte beim Arbeiten mit einer Anwendung das Erinnerungsvermögen der Nutzer*in so wenig wie möglich belastet werden. Stattdessen sollten viele Inhalte wiedererkennbar und so direkt mit einem Kontext versehen werden können [22–24]. Da die prototypische Implementierung die Requirements-Class Dismiss implementiert, können Nutzer*innen von ihnen gestartete Jobs vor oder während der Prozessierung abbrechen. Haben Nutzer*innen die ID ihres Jobs und dessen Parameter vergessen, gibt es bisher kaum Möglichkeiten, den von ihnen angelegten Job schnell zu identifizieren, da diese nicht an Nutzerkonten oder Kennungen geknüpft sind. Wird jedoch auch die bereits beschriebene Requirements-Class Callback implementiert, wird beim Anlegen eines Jobs ein Subscriber-Objekt erzeugt. Dieses könnte auch dazu dienen, einen Job klar einem Nutzer*innen zuzuordnen. Diese Zuordnung könnte zum Beispiel genutzt werden, um die Job Liste nur auf Jobs einer bestimmten Nutzer*in einzuschränken oder zu verhindern das Jobs von Nutzer*innen gelöscht werden können welche diesen nicht erzeugt haben.

4.4 Flexibilität und Effiziente Nutzung

Um Nutzer*innen die Möglichkeit zu geben, bereits erlernte Interaktionen schneller durchzuführen, sollte eine Anwendung Shortcuts zu bestimmten Funktionalitäten bereitstellen [22, 24]. Die Anwendung bietet momentan keine Shortcuts zu Funktionen. Die im Rahmen des HATEOAS Architekturstils umgesetzten Verknüpfungen innerhalb der Ressourcen deuten auf die Ressource im jeweils anderen Media-Type. Die Prozessbeschreibungen enthalten darüber hinaus eine Verknüpfung mit der Relation `execution`. Eine Erweiterung der Ressourcen nach dem HATEOAS Architekturstils könnte Interaktionen mit diesen noch effizienter gestalten. So könnten die Jobbeschreibungen um Verknüpfungen mit den Relationen `results` und `dismiss` erweitert werden, um sämtliche Interaktionsmöglichkeiten direkt in der Ressource aufzuführen.

4.5 Konsistenz und Standards

Eine Anwendung sollte hinsichtlich der von ihr bereitgestellten Funktionen unbedingt konsistent sein. Dies bedeutet das Nutzer*innen identische Resultate für identische Anfragen erwarten können [22–24]. Zusätzlich kann es hilfreich sein, wenn die Anwendung weit verbreitete Standards und Konventionen einhält, da Nutzer*innen möglicherweise bereits mit diesen vertraut sind. Die Konsistenz der Interaktionen wird unter anderem durch die Anwendung des REST Architekturstils sichergestellt da hier der Zustand des Systems keinen Einfluss auf die angefragten Ressourcen nimmt. Die Verwendung des OGC API - Processes - Part 1: Core Standards sorgt zusätzlich dafür, dass auch bei Weiterentwicklung oder Erweiterung der Anwendung die Schnittstellen stabil bleiben. Dies kann zusätzlich durch den Test-Suit sichergestellt werden.

4.6 Fehlervermeidung

Nutzerfreundliche Anwendungen versorgen den Nutzer*innen nicht nur mit detaillierten Fehlermeldungen, sondern versuchen Fehler aktiv zu vermeiden. Dies erhöht die Stabilität der Anwendung, da Fehler, welche möglicherweise zum Absturz der Anwendung führen, vermieden werden können. Die Anwendung kann jedoch auch leichter verwendet werden. So können Mehrfacheingaben und lange Wartezeiten bis zum Fehlschlag eines Prozesses vermieden werden [22–24]. Wird ein Prozess von einer Nutzer*in ausgeführt, werden zunächst seine Eingaben überprüft. Sind diese nicht valide, wird kein Job angelegt, auf dessen Fehlschlag eine Nutzer*innen warten müsste. Allerdings erhält die Nutzer*in bisher keine Informationen darüber, welche Eingabeparameter nicht validiert werden konnten. Jedoch wird ein nicht valider `limit`-Parameter auf den Standardwert gesetzt und mit der Prozessierung des Requests fortgefahren. Mechanismen, welche bei fehlerhaften Eingaben einen Standardwert verwenden oder den wahrscheinlichsten Wert ermitteln, können jedoch ergänzt werden.

4.7 Fehlerbehandlung

Treten doch Fehler auf, so müssen die Fehlermeldungen einfach und verständlich sein. Im Optimalfall enthalten die Fehlermeldungen auch eine detaillierte Fehlermeldung sowie Lösungsvorschläge. Ausschließlich maschinenlesbare Fehlercodes sollten also vermieden werden [22–24]. Viele mögliche Fehler werden durch das Versenden von Responses mit einem passenden HTTP-Statuscode abgefangen. Der Zahlencode allein ist wenig aussagekräftig. Ein Response mit einem auf einen Fehler hinweisenden Statuscode enthält jedoch eine textuelle Information aus der ersichtlich wird, was der Grund für den Fehler war. Manche Fehler lösen auch das Versenden einer Exception aus. Aus dieser wird klar ersichtlich, was der Grund für den Fehler ist. Lediglich Fehler, welche während der Prozessierung eines Jobs auftreten, lassen sich mithilfe von Fehlermeldungen nur schwer nachvollziehen. Zwar kann dem Job-Status in einem solchen Fall entnommen werden, welcher Prozessierungsschritt nicht erfolgreich durchgeführt werden konnte, aber die Gründe lassen sich den Fehlermeldungen nicht entnehmen. Lösungsvorschläge werden von der Anwendung bisher nicht gemacht. Ein Abstürzen der Anwendung wird häufig unter Verwendung von `try`- und `except`-Blöcken vermieden. Tritt während der Prozessierung ein Fehler auf, wird der Status des Jobs entsprechend geändert und dessen weitere Bearbeitung abgebrochen. Um Nutzer*innen die Möglichkeit zu geben, ihre Eingaben nach einem Fehler zu korrigieren, könnten die vom Programm generierten Ausgaben Teil der Fehlermeldung sein. Versierte Nutzer*innen könnten so Rückschlüsse auf mögliche Ursachen ziehen. Manche, häufig gemachte Fehler, könnten auch explizit abgefangen werden. Dazu zählen das Verwechseln von den Zeitstempeln von vor und nach dem Überschwemmungsereignis oder die Angabe von invaliden Koordinaten der Bounding Box.

4.8 Hilfe und Dokumentation

Eine Anwendung sollte nicht nur in einer für Softwareentwickler verständlichen Art und Weise dokumentiert sein. Zusätzlich kann es von Vorteil sein, wenn eine Dokumentation interaktiv durchsucht werden kann, um zu lösende Problem schnell zu beheben. [22–24]. Die prototypische Implementierung enthält eine in die Anwendung integrierte Dokumentation. Diese beschreibt detailliert sämtliche implementierten Funktionen. Da die Dokumentation dem OpenAPI 3.0 Standard entspricht, ist sie sowohl menschen- als auch maschinenlesbar. Alle Eingabemöglichkeiten und zu erwartenden Ausgaben werden für jeden Endpoint der API aufgeführt. Zusätzlich liegen zu den Eingaben und Ressourcen Schemata vor, welche die Struktur dieser beschreiben. Die Funktionsweise der von der Anwendung angebotenen Prozesse kann den entsprechenden Prozess-Beschreibungen entnommen werden. In diesen sind die Kontrolloptionen, verfügbare Transmissions-Modi sowie mögliche Ein- und Ausgaben aufgeführt. Um die Dokumentation interaktiver zu gestalten, bieten Plattformen wie SwaggerHub Möglichkeiten zum Aufsetzten von Anwendungs-Mockups aus API-Definitionen. Die eigentlichen Prozesse sind zwar nicht Teil dieser Mockups jedoch können Nutzer*innen sich auf diese Weise vertraut mit der API machen.

5 Diskussion

Die prototypisch implementierte OGC API - Processes - Part 1: Core standardkonforme API konnte zeigen, dass die im Standard formulierten Ziele erreicht werden. Die API erlaubt Nutzer*innen mit wenigen, einfach zu bedienenden Endpoints Prozesse, welche Geodaten erzeugen oder verarbeiten, zu starten, zu überwachen und deren Ergebnisse abzurufen. Dabei werden die Architekturstile REST und HATEOAS berücksichtigt. Die angebotenen Ressourcen stehen in menschen- und maschinenlesbaren Formaten zur Verfügung und enthalten Verknüpfungen zu anderen Ressourcen. Die prototypische Anwendung konnte zeigen, dass sich das `flask`-Framework gut dazu eignet, auf wenige Funktionen beschränkte APIs zu entwickeln. Endpoints und die mit ihnen verknüpften Funktionen lassen sich mit geringem Aufwand implementieren. Dabei können die bereitgestellten Ressourcen auch dynamische Inhalte enthalten. Das Copernicus Open Access Hub ist als Datenquelle zu Daten des Copernicus-Programmes geeignet. Das `sentinelsat`-Package ermöglicht einfaches Auffinden von geeigneten Datensätzen. Sofern diese sich nicht im Langzeitarchiv befinden, können diese direkt heruntergeladen werden. Die Limitierungen des Langzeitarchivs können durch das persistente Speichern von Datensätzen umgangen werden. Die räumliche und zeitliche Abdeckung wird dann jedoch durch den zur Verfügung stehenden Speicherplatz beschränkt. Die SNAP Plattform und der Python-Wrapper `snappy` ermöglichen eine vollständige Kalibrierung von Sentinel-1 Datensätzen. Die nötige Installation und Konfiguration der Plattform wirken sich allerdings auch nachteilig auf die Eigenschaften der prototypischen Anwendung aus. So wird zum Beispiel die Python Version eingeschränkt. Auch gestaltet sich etwaige Containerisierung aufwendig. Zu bemerken ist jedoch, dass für das Kalibrieren von Sentinel-1 Datensätzen kaum alternative Lösungen oder Python-Packages zur Verfügung stehen. Allerdings stellen manche DIAS Plattformen bereits kalibrierte Datensätze zur Verfügung. Die Verwendung dieser spart zwar die Installation und Konfiguration von SNAP, nimmt Entwicklern allerdings die Möglichkeit, Einfluss auf die Kalibrierungen zu nehmen. Das implementierte Verfahren zum Überschwemmungsmonitoring auf Basis der Daten der Sentinel-1 Mission erlaubt ein Detektieren von Überschwemmungen. Die Berechnung des NDSI und dessen Binärisierung anhand eines Schwellwertes kann leicht mit den Python-Packages `skimage` und `osgeo` implementiert werden. Allerdings schwankt die Qualität der Ergebnisse stark. So liefert das gewählte Schwellwertverfahren die verlässlichsten Schwellwerte, wenn der NDSI eine stark bimodale Verteilung aufweist. Ist diese nur schwach oder gar nicht vorhanden, kann es zu einer wenig brauchbaren Binärisierung kommen. Die Ausprägung der Bimodalität hängt dabei vom gewählten Raumausschnitt sowie dem Verhältnis von überfluteten zu trockenliegenden Flächen in diesem ab. Da die Anwendung die Ausgabe von NDSI und binärer Überschwemmungsmaske erlaubt, können Nutzer*innen sowohl analysebereite als auch interpretationsfähige Daten beziehen.

Die Evaluation der Nutzbarkeit der prototypischen Anwendung offenbarte, dass diese für An-

wendungen erstrebenswerte Eigenschaften aufweist. Zum einen ist die Anwendung detailliert dokumentiert. Zum anderen werden Nutzer*innen bereits in vielfältiger Weise über den Stand seiner gestarteten Prozesse informiert. Ein wichtiger Aspekt computergestützter Verfahren ist ihre Reproduzierbarkeit. Das Committee on Reproducibility and Replicability in Science definiert ein vorgestelltes computergestütztes Verfahren als reproduzierbar, wenn mit identischen Eingaben und unter gleichen Systembedingungen identische Resultate erzielt werden können. Dies bedeutet zunächst, dass die verwendete Software quelloffen und kostenlos zur Verfügung steht, um Interessierte in die Lage zu versetzen, das vorgestellte Verfahren selbst durchzuführen. Um dies so einfach wie möglich zu machen, sollte die verwendete Software detailliert dokumentiert sein. Ein besonderes Augenmerk sollte dabei auf den zugrundeliegenden Daten, den verwendeten Methoden und der ursprünglichen Systemumgebung liegen [3]. Die API der prototypischen Implementierung ist durch die Dokumente der OGC und die API-Definition gut und ausführlich dokumentiert. Diese gewährleisten zusätzlich das Nutzer*innen die Bedienung der API schnell erlernen können. Die zum Überschwemmungsmonitoring verwendeten Methoden in dieser Arbeit und den im Rahmen dieser Arbeit zurate gezogenen Arbeiten ausführlich beschrieben. Sie stehen jedoch nicht gesammelter und aufbereiteter Form zur Verfügung. Die verwendete Systemumgebung kann aus der der Anwendung beigefügten `environment.yaml` nachvollzogen werden.

Da die prototypisch implementierte API den OGC API - Processes - Part 1: Core Standard nicht vollständig umsetzt, konnten nicht alle beschriebenen Funktionen getestet werden. Dazu zählen die Callback Funktionalität, die Ausgabe von Ergebnissen im Response-Typ `reference` und ein vollständiger Test-Suit. Im Rahmen dieser Arbeit wurden nur Daten aus dem Copernicus Open Access Hub bezogen. Zur Struktur und Qualität der von den DIAS Plattformen bereitgestellten Daten kann keine Aussage gemacht werden. Da die Evaluation der Anwendung auf heuristischer Basis erfolgte, sind die vorgestellten Ergebnisse subjektiv und vom Evaluierenden abhängig. Außerdem wurden nur Nutzbarkeitsaspekte betrachtet. Eine Evaluierung von technischen Aspekten ist nicht erfolgt. Ein Vergleich mit anderen Implementierungen welche zum Beispiel andere Frameworks verwenden oder in anderen Programmiersprachen verfasst sind ist aus Gründen des Umfangs ebenfalls nicht Teil dieser Arbeit.

6 Fazit und Ausblick

Im Rahmen dieser Arbeit konnte ein funktionsfähiges Rich Data Interface für die Daten des Copernicus-Programmes implementiert werden. Teil der Anwendung ist eine API, welche diese wesentlichen im OGC API - Processes - Part 1: Core Standard beschriebenen Funktionen anbietet. Mithilfe der API kann Überschwemmungsmonitoring auf Basis der Radardaten der Sentinel-1 Mission betrieben werden. Die heuristische Untersuchung dieser prototypischen Implementierung konnte bestätigen das sich der OGC API - Processes - Part 1: Core Standard gut dazu eignet nutzerfreundliche APIs für Abwendungen zu entwickeln, welche Nutzer*innen in die Lage versetzten komplexe Verarbeitungen von Daten des Copernicus-Programmes durchzuführen. Nutze*innen können dabei sowohl analysebereite als auch interpretationsfähige Daten zur Verfügung gestellt werden. Ebenso konnte bestätigt werden das sich solche Anwendungen mir der Programmiersprache Python sowie die zur Verfügung stehenden Packages implementieren lassen. Die Nutzung des `flask`-Frameworks erlaubt schnell und leicht APIs zu implementieren während Packages wie `osgeo` und `skimage` auch komplexe Verarbeitungen von Rasterdaten erlauben. Die Daten des Copernicus-Programmes können ebenfalls in sinnvoller Weise mit einer Anwendung gekoppelt werden. Die Beschaffung von Datensätzen ist mit dem `sentinelsat`-Package gut umsetzbar während der Python-Wrapper `snappy` die vollständige Kalibrierung der Datensätze ermöglicht. Die limitierenden Eigenschaften des Copernicus Open Access Hub können zumindest für eine begrenzte Anzahl von Datensätzen umgangen werden. Das vorgestellte Verfahren zum Detektieren von Überschwemmungen funktioniert unter optimalen Bedingungen hinreichend gut. Gleichzeitig konnte die vorgestellte Anwendung deutlich Potentiale zur Weiterentwicklung aufzeigen.

Um die vorgestellte prototypische Anwendung zu einem Softwareprodukt zu machen, welches produktiv eingesetzt werden kann, sollten einige Aspekte ausgebaut und erweitert werden. Zunächst sollte die Installation der Anwendung deutlich vereinfacht werden, da das Erzeugen eines passenden Python-Environments und die Konfiguration von SNAP sich als umständlich und zeitaufwendig erweisen können. Eine mögliche Lösung wäre die Containerisierung der Anwendung mithilfe von Docker [17].

Auch werden bisher nicht alle Aspekte des OGC API - Processes - Part 1: Core Standards umgesetzt. Die Anwendung sollte im nächsten Schritt also vervollständigt werden um zum Beispiel auch den Transmission-Mode `reference` zu unterstützen und einen Callback-Mechanismus bereitzustellen. Auch Recommendations wie die Unterstützung von HTTPS und CORS sollten erwogen werden. Zudem sollte die Anwendung durch das OGC Validierungsverfahren auf ihre Standardkonformität hin überprüft und gegebenenfalls durch die OGC zertifiziert werden.

Darüber hinaus sollte die Anwendung sollte aus Gründen der Reproduzierbarkeit um eine zusammengefasste und aufbereitete Methodenbeschreibung sowie ein ausführbares Beispiel ergänzt werden.

Da die Limitierungen der vorgestellten Kopplung zu den Daten des Copernicus-Programmes möglicherweise durch die Nutzung von DIAS Plattformen umgangen werden können, sollten diese als alternative Datenquellen erschlossen werden.
Um die Schwellwertbildung unabhängiger von den Eigenschaften der Verteilung der Reflexionswerte zu machen, sollten andere Verfahren untersucht und gegebenenfalls implementiert werden. Dies können komplexere statistische Verfahren wie der SNDSI, aber auch maschinelle Lernverfahren und Modelle sein [5, 30]. Des Weiteren sollte untersucht werden ob sich die vorgestellte Anwendung in sinnvoller Wiese um die Konzepte der CARD4L Initiative des Committee on Earth Observation Satellites (CEOS) erweitert werden kann. Dazu zählen unter anderem die Formulierung von Produkt Spezifikationen sowie die Einführung von Qualitätsmerkmalen für die bereitgestellten Daten [17].
Im Rahmen dieser Arbeit wurden nur die Nutzbarkeitsaspekte evaluiert. Zusätzlich sollten jedoch auch Aspekte wie die Wartbarkeit, Skalierbarkeit und Performanz näher mit geeigneten Heuristiken und Verfahren untersucht werden. Falls Aspekte mit Metriken bewertet werden können, sollten die heuristischen Evaluationen um diese ergänzt werden. Mit Hinblick auf eine mögliche technische Evaluationen scheint es sinnvoll Frameworks wie `django` und andere Programmiersprachen wie R und Java zu erproben da auch für diese eine Vielzahl von Packages und Bibliotheken zur Verfügung stehen um Webanwendungen und Prozesse zu entwickeln.

Literatur

[1] J. Albertz, Einführung in die Fernerkundung, 4. Auflage Darmstadt: Wissenschaftliche Buchgesellschaft, 2009

[2] M. Bourbigot, H. Johnson, R. Piantanida (2016, März 03). Sentinel-1 Product Definition [Online]. Verfügbar unter: https://sentinel.esa.int/web/sentinel/user-guides/sentinel-1-sar/document-library/-/asset_publisher/1dO7RF5fJMbd/content/sentinel-1-product-definition (Zugriff am: 13. Juni 2022).

[3] Committee on Reproducibility and Replicability in Science (2019, Mai 07). Reproducibility and Replicability in Science [Online]. Verfügbar unter: https://nap.nationalacademies.org/catalog/25303/reproducibility-and-replicability-in-science (Zugriff am: 27. Juli 2022).

[4] S. Cox, D. Danko, J. Greenwood, J.R. Herring, A. Matheus, R. Pearsall, C. Portele, B. Reff, P. Scarponcini, A. Whiteside (2009, Oktober 19). The Specification Model - A Standard for Modular specifications [Online]. Verfügbar unter: https://www.ogc.org/standards/modularspec (Zugriff am: 27. Juni 2022).

[5] S. El Amrani Abouelassad (2019, November 30). Flood Detection with a Deep Learning Approach Using Optical and SAR Satellite Data [Online]. Verfügbar unter: https://www.ipi.uni-hannover.de/de/lehre/abschlussarbeiten/abgeschlossene-masterarbeiten/ (Zugriff am: 25. Juli 2022).

[6] Europäische Kommission (2018, Oktober 06). Copernicus: 20 years of History [Online]. Verfügbar unter: https://www.copernicus.eu/en/documentation/information-material/signature-esafrance-collaborative-ground-segment (Zugriff am: 13. Juni 2022).

[7] Europäische Kommission (2018, Juni). The DIAS: User-friendly Access to Copernicus Data and Information [Online]. Verfügbar unter: https://www.copernicus.eu/en/access-data/dias (Zugriff am: 24. Juni 2022)

[8] Europäische Kommission (2019). What is Copernicus [Online]. Verfügbar unter: https://www.copernicus.eu/en/documentation/information-material/brochuresbrochures (Zugriff am: 13. Juni 2022).

[9] Europäisches Parlament und Rat der Europäischen Union (2014, April 24). Regulation (EU) No 377/2014 Establishing the Copernicus Programme and repealing Regulation (EU) No 911/2010 [Online]. Verfügbar unter: https://www.kowi.de/Portaldata/2/Resources/horizon2020/coop/Copernicus-regulation.pdf (Zugriff am: 13. Juni 2022).

[10] European Space Agency (2022). Copernicus Open Access Hub - Overview [Online]. Verfügbar unter: https://scihub.copernicus.eu/userguide/WebHome (Zugriff am: 10. Juni 2022).

[11] European Space Agency (2012, März). Sentinel-1 ESA's Radar Observatory Mission for GMES Operational Services [Online]. Verfügbar unter: https://sentinel.esa.int/web/sentinel/missions/sentinel-1/overview (Zugriff am: 13. Juni 2022).

[12] European Space Agency (2018). Sentinels - Space for Copernicus [Online]. Verfügbar unter: https://www.d-copernicus.de/daten/satelliten/daten-sentinels/ (Zugriff am: 13. Juni 2022).

[13] C. Holmes, D. WWesloh, C. Heazel, G. Gale, A. Christl, J. Lieberman, C. Reed, J. Herring, M. Desruisseaux, D. Blodgett, S. Simmons, B. de Lathower und G. Percivall (2017, Februar 23). OGC Open Geospatial APIs - White Paper [Online]. Verfügbar unter: https://docs.ogc.org/wp/16-019r4/16-019r4.html (Zugriff am: 03. Juli 2022).

[14] F. Houbie, S. Sankaran, J. Lieberman, P. Vretanos, J. Masó (2016, Januar 16). OGC Testbed 11 REST Interface Engineering Report [Online]. Verfügbar unter: https://www.ogc.org/docs/er (Zugriff am: 05. Juli 2022).

[15] J.S. Lee, I. Juekevich, P. Dewaele, P. Wambacq, A. Oosterlinck (1994, Februar). Speckle filtering of synthetic aperture radar images: A Review [Online]. Verfügbar unter: https://www.researchgate.net/publication/239659062_Speckle_filtering_of_synthetic_aperture_radar_images_A_Review (Zugriff am: 27. Juli 2022).

[16] M. Biehl, API Design, 1. Auflage Suurstoffi: API-University Press, 2016

[17] J. Maso, A. Zabala and A. Brodia (2021, Januar 01). OGC Testbed-16: Analysis Ready Data Engineering Report [Online]. Verfügbar unter: https://scihub.copernicus.eu/userguide/WebHome (Zugriff am: 11. Juni 2022).

[18] A. McVittie (2019, Februar). Sentinel-1 Flood mapping tutorial [Online]. Verfügbar unter: https://step.esa.int/main/doc/tutorials/ (Zugriff am: 15. Juni 2022).

[19] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, U. Sarid (2021, February 15). OpenAPI Specification v3.1.0 [Online]. Verfügbar unter: https://spec.openapis.org/oas/v3.0.1 (Zugriff am: 07. Juli 2022).

[20] N. Miranda und P.J. Meadows (2015, Mai 21). Radiometric Calibration of S-1 Level-1 Products Generated by the S-1 IPF [Online]. Verfügbar unter: https://sentinel.esa.int/web/sentinel/user-guides/document-library/-/asset_publisher/xlslt4309D5h/content/sentinel-1-radiometric-calibration-of-products-generated-by-the-s1-ipf (Zugriff am: 15. Juni 2022).

[21] A. Moreira, M. Younis, P. Prats-Iraola, G. Krieger, I. Hajnsek und K. P. Papathanassiou (2013, April 17). A Tutorial on Synthetic Aperture Radar [Online]. Verfügbar unter: https://www.researchgate.net/publication/257008464_A_Tutorial_on_Synthetic_Aperture_Radar (Zugriff am: 6. Juni 2022).

[22] J. Nielsen (2020, November 15). 10 Usability Heuristics for User Interface Design [Online]. Verfügbar unter: https://www.nngroup.com/articles/ten-usability-heuristics/ (Zugriff am: 22. Juli 2022).

[23] J. Nielsen (2020, November 15). 10 Usability Heuristics for User Interface Design [Online]. Verfügbar unter: https://www.nngroup.com/articles/ten-usability-heuristics/ (Zugriff am: 27. Juli 2022).

[24] J. Nielsen, Usability Engineering, Mountain View: Academic Press Inc., 1993

[25] Open Geospatial Consortium (2021, Dezember 16). Bylaws of Open Geospatial Consortium [Online]. Verfügbar unter: https://www.ogc.org/ogc/policies (Zugriff am: 27. Juni 2022).

[26] Open Geospatial Consortium (2022). OGC Standards [Online]. Verfügbar unter: https://www.ogc.org/docs/is (Zugriff am: 27. Juli 2022).

[27] N. Otsu (1976, Januar). A Threshold Selection Method from Gray-Level Histograms [Online]. Verfügbar unter: https://ieeexplore.ieee.org/document/4310076/citations#citations (Zugriff am: 14. Juni 2022).

[28] A. Pilz (2022, August 08). Prototypical Rich Data Interface for Copernicus Data [Online]. Verfügbar unter: https://zenodo.org/record/6979963 (Zugriff am: 10. August 2022).

[29] B. Pross und P. A. Vretanos. (2021, Dezember 20). OGC API – Processes – Part 1: Core [Online]. Verfügbar unter: https://docs.opengeospatial.org/is/18-062r2/18-062r2.html (Zugriff am: 24. Juni 2022).

[30] N. I. Ulloa, S.-H. Chiang und S.-H. Yun (2020, April 27). Flood Proxy Mapping with Normalized Difference Sigma-Naught Index and Shannon's Entropy [Online]. Verfügbar unter: https://doi.org/10.3390/rs12091384 (Zugriff am: 21. Juni 2022).

A Pseudocode

A.1 Ablauf eines Requests an den API Landing Page Endpoint

Programmablauf 1 Ablauf eines Requests an den API Landing Page Endpoint

```
Input: Request
if Request.HTTP-Method != GET then
    return HTTP-Statuscode 405
else
    try
        if Request.f == text/html or Request.f == None then
            Response ← render_template(templates/html/landingPage.html)
            Link-Header ← http://HOST:PORT/?f=text/html
            Resource-Header ← landingPage
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else if Request.f == application/json then
            JSON ← open(templates/json/landingPage.json)
            Response ← jsonify(JSON)
            Link-Header ← http://HOST:PORT/?f=application/json
            Resource-Header ← landingPage
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else
            return HTTP-Statuscode 406
    except
        return HTTP-Statuscode 500
```

A.2 Ablauf eines Requests an den API Definition Endpoint

Programmablauf 2 Ablauf eines Requests an den API Definition Endpoint

```
Input: Request
if Request.HTTP-Method != GET then
    return HTTP-Statuscode 405
else
    try
        if Request.f == text/html or Request.f == None then
            Response ← render_template(templates/html/apiDefinition.html)
            Link-Header ← http://HOST:PORT/api?f=text/html
            Resource-Header ← apiDefinition
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else if Request.f == application/json then
            JSON ← open(templates/json/apiDefinition.json)
            Response ← jsonify(JSON)
            Link-Header ← http://HOST:PORT/api?f=application/json
            Resource-Header ← apiDefinition
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else
            return HTTP-Statuscode 406
    except
        return HTTP-Statuscode 500
```

A.3 Ablauf eines Requests an den Conformance Endpoint

Programmablauf 3 Ablauf eines Requests an den Conformance Endpoint

```
Input: Request
if Request.HTTP-Method != GET then
    return HTTP-Statuscode 405
else
    try
      if Request.f == text/html or Request.f == None then
          Response ← render_template(templates/html/confClasses.html)
          Link-Header ← http://HOST:PORT/conformance?f=text/html
          Resource-Header ← conformance
          return Response with Link- and Resource-Header and HTTP-Statuscode 200
      else if Request.f == application/json then
          JSON ← open(templates/json/confClasses.json)
          Response ← jsonify(JSON)
          Link-Header ← http://HOST:PORT/conformance?f=application/json
          Resource-Header ← conformance
          return Response with Link- and Resource-Header and HTTP-Statuscode 200
      else
          return HTTP-Statuscode 406
    except
      return HTTP-Statuscode 500
```

A.4 Ablauf eines Requests an den Process List Endpoint

Programmablauf 4 Ablauf eines Requests an den Process List Endpoint

```
Input: Request
if Request.HTTP-Method != GET then
    return HTTP-Statuscode 405
else
    try
        if Request.limit == None or Request.limit <= 0 or Request.limit > 100000 then
            Limit ← 10
        else
            Limit ← Anfrage.limit
        if Request.f == text/html or Request.f == None then
            Process List ← []
            Processes ← Process Descriptions in templates/json/processes
            for Process in Processes do
                JSON ← open(templates/json/processes/<Process>ProcessDescription.json)
                Process List append JSON
            Response ← render_template(templates/html/processList.html, Process List[0:Limit])
            Link-Header ← http://HOST:PORT/processList?f=text/html
            Resource-Header ← processList
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else if Request.f == application/json then
            Process List ← []
            Processes ← List of Process descriptions in templates/json/processes
            for Process in Processes do
                JSON ← open(templates/json/processes/<Process>ProcessDescription.json)
                Process List append JSON
            add Links to self and alternate to Process List
            Response ← jsonify(Process List[0:Limit])
            Link-Header ← http://HOST:PORT/processList?f=application/json
            Resource-Header ← processList
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else
            return HTTP-Statuscode 406
    except
        return HTTP-Statuscode 500
```

A.5 Ablauf eines Requests an den Process Description Endpoint

Programmablauf 5 Ablauf eines Requests an den Process Description Endpoint

```
Input: Request, Process-ID
if Request.HTTP-Method != GET then
    return HTTP-Statuscode 405
else
    try
      if Request.f == text/html or Request.f == None then
          if templates/json/processes/<Process-ID>ProcessDescription.json exists then
              JSON ← open(templates/json/processes/<Process-ID>ProcessDescription.json)
              Response ← render_template(templates/html/process.html, JSON)
              Link-Header ← http://HOST:PORT/processes/<Process-ID>?f=text/html
              Resource-Header ← process - Request.<Process-ID>
              return Response with Link- and Resource-Header and HTTP-Statuscode 200
          else
              Exception ← No such process exception
              Resource-Header ← no-such-process
              return Exception with Resource-Header and HTTP-Statuscode 404
      else if Request.f == application/json then
          if templates/json/processes/<Process-ID>ProcessDescription.json exists then
              JSON ← open(templates/json/processes/<Process-ID>ProcessDescription.json)
              Response ← jsonify(JSON)
              Link-Header ← http://HOST:PORT/processes/<Process-ID>?f=application/json
              Resource-Header ← process - Request.<Process-ID>
              return Response with Link- and Resource-Header and HTTP-Statuscode 200
          else
              Exception ← No such process exception
              Resource-Header ← no-such-process
              return Exception with Resource-Header and HTTP-Statuscode 404
      else
          return HTTP-Statuscode 406
    exept
      return HTTP-Statuscode 500
```

A.6 Ablauf eines Requests an den Process Execution Endpoint

Programmablauf 6 Ablauf eines Requests an den Process Execution Endpoint

```
Input: Request, Process-ID
if Request.HTTP-Methode != POST then
    return HTTP-Statuscode 405
else
    try
        if templates/json/processes/<Process-ID>ProcessDescription.json exists then
            Input Parameters ← parse(Request.Parameters)
            if Input Parameters == False then
                return HTTP-Statuscode 400
            else
                Job-ID ← generateUUID()
                create Job Directory
                create Results Directory
                create status.json
                create job.json
                Response ← jsonify(status.json)
                Location-Header ← http://HOST:PORT/jobs/<Job-ID>?f=application/json
                Resource-Header ← job - Job-ID
                return Response with Location- and Resource-Header and HTTP-Statuscode 201
        else
            Exception ← No such process exception
            Resource-Header ← no-such-process
            return Exception with Resource-Header and HTTP-Statuscode 404
        return HTTP-Statuscode 500
```

A.7 Ablauf eines Requests an den Job List Endpoint

Programmablauf 7 Ablauf eines Requests an den Job List Endpoint

```
Input: Request
if Request.HTTP-Method != GET then
    return HTTP-Statuscode 405
else
    try
        if Request.limit == None or Request.limit <= 0 or Request.limit > 100000 then
            Limit ← 10
        else
            Limit ← Request.limit
        if Request.type == None then
            Type = [process]
        else
            Type = Request.type
        if Request.processID == None then
            Process-ID = [Echo, FloodMonitoring]
        else
            Process-ID = request.processID
        if Request.status == None then
            Status = [accepted, running, successful, failed, dismissed]
        else
            Status = request.status
        Jobs ← Jobs in jobs/
        Job List ← []
        for Job in Jobs do
            JSON ← open(jobs/Job/status.json)
            Creation ← checkCreationDate(JSON.created, Request)
            Duration ← checkDuration(JSON, Request)
            if JSON.type (in) Type and JSON.processID (in) Process-ID and JSON.status in Status and Creation == True and Duration[0]
            == True and Duration[1] == True then
                Job List append JSON
        if Request.f == text/html or Request.f == None then
            Response ← render_template(templates/html/jobList.html, Job List[0:Limit])
            Link-Header ← http://HOST:PORT/jobs?f=text/html
            Resource-Header ← jobs
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else if Request.f == application/json then
            add Links to self and alternate to Job List
            Response ← jsonify(Job Liste[0:Limit])
            Link-Header ← http://HOST:PORT/jobs?f=application/json
            Resource-Header ← jobs
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else
            return HTTP-Statuscode 406
    exept
        return HTTP-Statuscode 500
```

A.8 Ablauf eines Requests an den Job Endpoint mit HTTP-Methode Get

Programmablauf 8 Ablauf eines Requests an den Job Status Endpoint mit HTTP-Methode Get

```
Input: Request, Job-ID
if Request.HTTP-Method == GET then
    try
      if Request.f == text/html or Request.f == None then
        if jobs/<Job-ID>/status.json exists then
          JSON ← open(jobs/<Job-ID>/status.json)
          Response ← render_template(templates/html/job.html, JSON)
          Link-Header ← http://HOST:PORT/jobs/<Job-ID>?f=text/html
          Resource-Header ← job - <Job-ID>
          return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else
          Exception ← No such job exception
          Resource-Header ← no-such-job
          return Exception with Resource-Header and HTTP-Statuscode 404
      else if Request.f == application/json then
        if jobs/<Job-ID>/status.json exists then
          Job ← open(jobs/<Job-ID>/status.json)
          add Links to self and alternate to Job
          Response ← jsonify(Job)
          Link-Header ← http://HOST:PORT/jobs/<Job-ID>?f=application/json
          Resource-Header ← job - <Job-ID>
          return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else
          Exception ← No such job exception
          Resource-Header ← no-such-job
          return Exception with Resource-Header and HTTP-Statuscode 404
      else
        return HTTP-Statuscode 406
    exept
      return HTTP-Statuscode 500
else if Request.HTTP-Method == DELETE then
    Handling of Request using HTTP DELETE
else
    return HTTP-Statuscode 405
```

A.9 Ablauf eines Requests an den Job Endpoint mit HTTP-Methode Delete

Programmablauf 9 Ablauf eines Requests an den Job Status Endpoint mit HTTP-Methode Delete

```
Input: Request, Job-ID
if Request.HTTP-Method == GET then
    Handling of Request using HTTP GET
else if Request.HTTP-Method == DELETE then
    try
      if jobs/<Job-ID>/status.json exists then
          Job ← open(jobs/<Job-ID>/status.json)
          if Job.status != dismissed then
              Job ← open(jobs/<Job-ID>/status.json)
              if Encoding == text/html or Encoding == None then
                  Response ← render_template(templates/html/job.html)
                  Link-Header ← http://HOST:PORT/jobs/<Job-ID>?f=text/html
                  Resource-Header ← job-dismissed
                  return Response with Link- and Resource-Header and HTTP-Statuscode 200
              else if Encoding == application/json then
                  Response ← jsonify(Job)
                  Link-Header ← http://HOST:PORT/jobs/<Job-ID>?f=application/json
                  Resource-Header ← job-dismissed
                  return Response with Link- and Resource-Header and HTTP-Statuscode 200
              else
                  return HTTP-Statuscode 406
          else
              if Encoding == text/html or Encoding == None then
                  Response ← render_template(templates/html/job.html)
                  Link-Header ← http://HOST:PORT/jobs/<Job-ID>?f=text/html
                  Resource-Header ← job-dismissed
                  return Response with Link- and Resource-Header and HTTP-Statuscode 410
              else if Encoding == application/json then
                  Response ← jsonify(Job)
                  Link-Header ← http://HOST:PORT/jobs/<Job-ID>?f=application/json
                  Resource-Header ← job-dismissed
                  return Response with Link- and Resource-Header and HTTP-Statuscode 410
              else
                  return HTTP-Statuscode 406
      else
          Exception ← No such job exception
          Resource-Header ← no-such-job
          return Exception with Resource-Header and HTTP-Statuscode 404
    exept
      return HTTP-Statuscode 500
else
    return HTTP-Statuscode 405
```

A.10 Ablauf eines Requests an den Job Results Endpoint

Programmablauf 10 Ablauf eines Requests an den Job Results Endpoint

```
Input: Request, Job-ID
if Request.HTTP-Method != GET then
    return HTTP-Statuscode 405
else
    try
        if jobs/<Job-ID>/status.json exists then
            Job Jobfile ← open(jobs/<Job-ID/job.json)
            Job Status ← open(jobs/<Job-ID/status.json)
            if Job Jobfile.processID == Echo then
                if Job Status.status == successful then
                    if Job Jobfile.responseType == raw then
                        return jobs/<jobID>/results/result.json and HTTP-Statuscode 200
                    else
                        Result-Value ← open(jobs/<Job-ID>/results/result.json)
                        embed Results-Value into Result-Document
                        return Result-Document and HTTP-Statuscode 200
                else if Job Status.status == failed then
                    Exception ← job failed exception
                    Resource-Header ← job-failed
                    return Exception with Resource-Header and HTTP-Statuscode 404
                else
                    Exception ← results not ready exception
                    Resource-Header ← results-not-ready
                    return Exception with Resource-Header and HTTP-Statuscode 404
            else if Job Jobfile.processID == FloodMonitoring then
                if Job Status.status == successful then
                    if Job Jobfile.responseType == raw then
                        if bin.tif and ndsi.tif are requested then
                            ZIP
                            add ndsi.tif and bin.tif to ZIP
                            return ZIP and HTTP-Statuscode 200
                        else if ndsi.tif is requested then
                            return ndsi.tif and HTTP-Statuscode 200
                        else if bin.tif is requested then
                            return bin.tif and HTTP-Statuscode 200
                    else
                        if bin.tif and ndsi.tif are requested then
                            bin-Base64 ← encodeImageBase64(bin.tif)
                            ndsi-Base64 ← encodeImageBase64(ndsi.tif)
                            embed bin-Base64 and ndsi-Base64 into Result-Document
                            add Download-Links to Result-Document
                            return Result-Document and HTTP-Statuscode 200
                        else if ndsi.tif is requested then
                            ndsi-Base64 ← encodeImageBase64(ndsi.tif)
                            embed ndsi-Base64 into Result-Document
                            add Download-Link to Result-Document
                            return Result-Document and HTTP-Statuscode 200
                        else if bin.tif is requested then
                            bin-Base64 ← encodeImageBase64(bin.tif)
                            embed bin-Base64 into Result-Document
                            add Download-Links to Result-Document
                            return Result-Document and HTTP-Statuscode 200
                else if Job Status.status == failed then
                    Exception ← job failed exception
                    Resource-Header ← job-failed
                    return Exception with Resource-Header and HTTP-Statuscode 404
```

```
            else
                Exception ← results not ready exception
                Resource-Header ← results-not-ready
                return Exception with Resource-Header and HTTP-Statuscode 404
        else
            Exception ← No such job exception
            Resource-Header ← no-such-job
            return Exception with Resource-Header and HTTP-Statuscode 404
    exept
        return HTTP-Statuscode 500
```

A.11 Ablauf eines Requests an den Coverage Endpoint

Programmablauf 11 Ablauf eines Requests an den Coverage Endpoint

```
Input: Request
if Request.HTTP-Methode != GET then
    return HTTP-Statuscode 405
else
    try
        KML ← KML-Files in data/coverage/
        Coverages ← []
        BBoxes ← []
        for File in KML do
            KML ← open(File)
            Product ← KML.Name, KML.BBox, KML.Date
            BBox ← KML.BBox
            Coverages append Dataset
            BBoxes append BBox as .geojson
        if Request.f == text/html or Request.f == None then
            Response ← render_template(templates/html/coverage.html, Coverages, BBoxes)
            Link-Header ← http://HOST:PORT/coverage?f=text/html
            Resource-Header ← coverage
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else if Request.f == application/json then
            Response ← jsonify(Coverages)
            Link-Header ← http://HOST:PORT/coverage?f=application/json
            Resource-Header ← coverage
            return Response with Link- and Resource-Header and HTTP-Statuscode 200
        else
            return HTTP-Statuscode 406
    exept
        return HTTP-Statuscode 500
```

A.12 Ablauf eines Requests an den Download Endpoint

Programmablauf 12 Ablauf eines Requests an den Download Endpoint

```
Input: Request, Job-ID, requested File
if Request.HTTP-Methode != GET then
    return HTTP-Statuscode 405
else
    try
        if jobs/<Job-ID>/ exists then
            Status-JSON ← open(jobs/<Job-ID>/status.json)
            Job-JSON ← open(jobs/<Job-ID>/status.json)
            if Job-JSON.processID == FloodMonitoring then
                if <requested File> == bin then
                    return bin.tif and HTTP-Statuscode 200
                else if <requested File> == ndsi then
                    return ndsi.tif and HTTP-Statuscode 200
                else
                    Exception ← No such file exception
                    Resource-Header ← no-such-file
                    return Exception with Resource-Header and HTTP-Statuscode 404
            else
                return HTTP-Statuscode 501
        else
            Exception ← No such job exception
            Resource-Header ← no-such-job
            return Exception with Resource-Header and HTTP-Statuscode 404
    exept
```

return HTTP-Statuscode 500

A.13 Echo Prozess

Programmablauf 13 Ablauf eines Echo Prozesses

Input: Job
try
Break If Job.Status == dismissed
Job.Started = now()
Wait 5 Seconds
Break If Job.Status == dismissed
Create result.json
Job.Finished = now()
exept
Job.Status = failed

A.14 Überschwemmungmonitoring Prozess

Programmablauf 14 Ablauf eines Überschwemmungmonitoring Prozesses

Input: Job
try
Job.Started = now()
break if Job.Status == dismissed
#Setup Processing
create footprint.geojson **from** Job.Input.BBOX
Pre-Timeframe ← **from** Job.Input.preDate
Post-Timframe ← **from** Job.Input.postDate
break if Job.Status == dismissed
API ← Login at Copernicus Open Access Hub **with** Job.Input.Credentials
break if Job.Status == dismissed
#Retrieve and Calibrate Pre-Product
Pre-Product ← getProduct(API, Pre-Timeframe, Job.Input.BBOX)
break if no matching Product could be found
if Pre-Dataset **not in** Datastore **then**
Pre-Dataset ← retrieveProduct(API, Pre-Product)
break If Pre-Product **in** Long Term Archive
apply Orbit-File to Pre-Dataset
clip Pre-Dataset with footprint.geojson
remove thermal Noise from Pre-Dataset
convert VV Band **to** σ_0 Values
perform Speckle-Filtering **with** 5x5 Lee-Sigma-Filter
perform Terrain-Correction with Ellipsoid
store calibrated Pre-Dataset **as** GeoTIFF
break if Job.Status == dismissed
#Retrieve and Calibrate Post-Product
Post-Product ← getProduct(API, Post-Timeframe, Job.Input.BBOX)
break if no matching Product could be found
if Post-Product **not in** Datastore **then**
Post-Dataset ← retrieveProduct(API, Post-Product)
break if Post-Product **in** Long Term Archive
apply Orbit-File **to** Post-Dataset
clip Post-Dataset **with** footprint.geojson
remove thermal Noise **from** Post-Dataset
convert VV Band **to** σ_0 Values
perform Speckle-Filtering **with** 5x5 Lee-Sigma-Filter
perform Terrain-Correction **with** Ellipsoid
store calibrated Post-Dataset **as** GeoTIFF
break if Job.Status == dismissed
#Calculate and Threshold NDSI
NDSI ← calculateNDSI(calibrated Pre- and Post-Dataset)
store NDSI **as** ndsi.tif
clip NDSI **with** footprint.geojson
Threshold ← Threshold NDSI with Ostsus-Method
Flood-Mask ← Binarize NDSI with Threshold
store Flood-Mask **as** bin.tif
Job.Finished = now()
except
Job.Status = failed

Plagiatserklärung des Studierenden

Hiermit versichere ich, dass die vorliegende Arbeit über

Rich Data Interfaces for Copernicus Data

selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Alexander Pilz, Münster den 15. August 2022

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Alexander Pilz, Münster den 15. August 2022