Berlin Simple Open Data: Erkenntnisse aus der Entwicklung von drei MCP-Servern für Open Data

Dieses Dokument fasst die wichtigsten Herausforderungen und Erkenntnisse aus der Entwicklung des “Berlin Simple Open Data”-Projekts zusammen – einer Sammlung von MCP-Servern (Model Context Protocol), die KI-Systemen den Zugang zu Berlins Open-Data-Ökosystem ermöglichen.

Projektübersicht

Das Projekt besteht aus drei MCP-Servern und einer Web-Oberfläche:

Komponente	**Funktion**
Berlin Open Data MCP	Suche und Datenabruf vom Berliner Datenportal
Datawrapper MCP	Automatische Erstellung von Diagrammen und Karten
Masterportal MCP	Generierung von Geodatenportalen zum Download
Interface Prototype	Web-basierte Chat-Oberfläche

Teil 1: Berlin Open Data MCP

Überblick

Der Berlin Open Data MCP-Server ermöglicht die Verbindung zum offiziellen Berliner Datenportal (daten.berlin.de) für natürlichsprachige Suche und Datenabruf. Was zunächst wie eine unkomplizierte API-Integration aussah, entpuppte sich als deutlich komplexer: Das Portal zeigte grundlegende Schwächen in der Suchfunktion, die Datenformate waren überraschend vielfältig, und es traten kritische Sicherheitsfragen auf, wenn KI-Systeme mit realen Daten arbeiten.

Herausforderung 1.1: Die Suche funktioniert nicht wie erwartet

Die Ausgangslage
Berlins Datenportal basiert auf CKAN, einer weit verbreiteten Open-Data-Plattform. Die Erwartung: Nutzer suchen nach „Mietpreise" oder „Miete" und finden passende Datensätze.

Die Realität
Tests zeigten schnell, dass CKAN nur exakte Textübereinstimmungen findet – ohne jede sprachliche Intelligenz:

- „Wohnung" (Singular): **0 Treffer**
- „Wohnungen" (Plural): **11 Treffer**
- „Miete": **0 Treffer**
- „Mietspiegel" (Kompositum): **39 Treffer**

Die Suchmaschine kennt weder Wortstämme noch unscharfe Suche und versteht keine deutschen Komposita – fatal für ein deutschsprachiges Portal.

Der Weg zur Lösung
Erster Versuch: Wildcard-Suchen wie „Miet*" sollten alles finden, was mit „Miet" beginnt. Das Ergebnis war ernüchternd: **CKANs Wildcard-Funktion ist defekt** – „Miet*" lieferte null Treffer, obwohl „Mietspiegel" existiert.

Zweiter Versuch: Ein professioneller Stemming-Algorithmus sollte Wörter auf ihre Stammformen reduzieren. Auch das scheiterte – der Ansatz setzte funktionierende Wildcards voraus.

Die finale Lösung ging einen völlig anderen Weg: Eine Analyse aller 2.660 Datensätze im Portal ermittelte, welche Begriffe tatsächlich gemeinsam vorkommen. Dieses „hybride Query-Expansion-System" erweitert Suchanfragen nun automatisch um das Vokabular des Portals. Wer nach „Miete" sucht, findet auch „Mietspiegel" – weil das System weiß, dass das Portal diesen Begriff verwendet.

Erkenntnisse
- Open-Data-Portale haben nicht automatisch eine gute Suchfunktion
- Gute Suche erfordert erheblichen Aufwand – selbst bei etablierten Plattformen
- Lösungen müssen auf echten Daten basieren, nicht auf Annahmen
- Die Kluft zwischen Alltagssprache und Portal-Vokabular ist eine zentrale UX-Hürde

Herausforderung 1.2: Die Vielfalt der Datenformate

Der Plan
Das Projekt startete mit Unterstützung für CSV und JSON – die gängigsten Formate.

Die Wirklichkeit
Eine Analyse des Portals zeigte: Diese Formate decken weniger als die Hälfte der Datensätze ab.

Format	**Datensätze**	**Anteil**
CSV/JSON	~1.000	~40%
Excel (XLS/XLSX)	545	20,6%
WFS (Web Feature Service)	596	22,4%
GeoJSON/KML	78	2,9%
JavaScript-generierte Downloads	182	6,9%
ZIP-Archive	diverse	—

Jedes Format brachte eigene Hürden mit sich:

Excel-Dateien erforderten eine zusätzliche Bibliothek und die Verarbeitung von Binärdaten statt Text.

WFS (Web Feature Service) ist ein Standard für geografische Daten. Die Unterstützung erforderte einen eigenen Protokoll-Client, Paginierung für große Datensätze und Koordinatentransformationen – Berlin nutzt EPSG:25833, die meisten Kartenwerkzeuge erwarten jedoch WGS84.

JavaScript-generierte Downloads von statistik-berlin-brandenburg.de bieten keine direkten Datei-URLs. Der Download wird erst durch JavaScript im Browser ausgelöst. Dafür musste ein Headless-Browser (Puppeteer) integriert werden – mit rund 300 MB zusätzlichen Abhängigkeiten.

Erkenntnisse
- „Offene Daten" heißt nicht „leicht zugängliche Daten"
- Die Formatvielfalt erzeugt einen langen Rattenschwanz an Integrationsarbeit

- Geodaten bringen unsichtbare Komplexität mit: Koordinatensysteme sind für Nutzer nicht sichtbar, aber entscheidend für korrekte Ergebnisse
- Manche Behördendaten erfordern Browser-Automatisierung – das wirft Fragen zur Maschinenlesbarkeit auf

---

Herausforderung 1.3: Wenn die KI Daten erfindet

Der Vorfall
Ein Nutzer fragte nach einer Analyse der Berliner Kitas – ein Datensatz mit 2.930 Einträgen. Der MCP-Server erkannte korrekt, dass der Datensatz zu groß war, und lieferte eine Vorschau mit 10 Zeilen sowie Download-Anweisungen.

Was dann passierte, war alarmierend: Als der automatische Download scheiterte (Proxy-Fehler), **erfand die KI kurzerhand 2.902 fiktive Kitas** – mit plausiblen Bezirksverteilungen, erfundenen Kapazitätszahlen und detaillierten Analysen. Das Ganze wurde als echte Daten präsentiert. Der Nutzer bemerkte die Fälschung nicht.

Warum das passierte
Die KI stand vor einem Dilemma:

1. Der Nutzer wollte eine Analyse
2. Die Vorschau deutete auf mehr Daten hin
3. Der Download schlug fehl
4. Statt das Scheitern einzugestehen, wollte die KI „helfen" – mit synthetischen Daten

Das ist ein grundsätzliches Risiko, wenn KI-Systeme auf reale Datenquellen zugreifen, die ausfallen können.

Gegenmaßnahmen
Die Reaktion war mehrschichtig:

1. **Explizite Warnungen** bei großen Datensätzen:
   „KRITISCH: Erstellen Sie KEINE synthetischen Daten basierend auf der Vorschau"

2. **Keine Download-Vorschläge mehr** – um fehlschlagende Automatisierungsversuche zu vermeiden

3. **Selbstprüfungs-Anweisungen** – die KI soll verifizieren, dass sie echte Daten verwendet

4. **Nutzer-Dokumentation** – Hinweise auf Warnsignale wie „Erstelle Beispieldaten basierend auf verfügbaren Informationen..."

Die Grenzen
Diese Maßnahmen reduzieren das Risiko, beseitigen es aber nicht. Das Verhalten der KI liegt außerhalb der Kontrolle des MCP-Servers – Warnungen können ignoriert werden. Nutzer müssen wachsam bleiben.

Erkenntnisse
- KI-Systeme versuchen „hilfreich" zu sein – selbst wenn Scheitern die richtige Antwort wäre
- Erfundene Daten können täuschend echt wirken, besonders wenn die KI die echte Datenstruktur kennt
- MCP-Server müssen Fehlerfälle mitdenken, nicht nur Erfolgsfälle
- Textwarnungen allein reichen möglicherweise nicht aus, um Halluzinationen bei kritischen Daten zu verhindern

---

Herausforderung 1.4: Analyse ohne Kontextüberlauf

Das Problem
KI-Systeme haben begrenzte Kontextfenster. Ein Datensatz mit 3.000 Zeilen passt nicht hinein – die Antwortqualität würde leiden. Aber Nutzer wollen Fragen stellen wie „Wie viele Einwohner hat jeder Bezirk?" – und das erfordert den kompletten Datensatz.

Die Lösung
Eine zweiteilige Architektur:
1. **Caching**: Der vollständige Datensatz wird serverseitig gespeichert, die KI erhält nur eine Vorschau (3 Zeilen)
2. **Code-Ausführung**: Eine Sandbox-Umgebung erlaubt der KI, JavaScript gegen die gecachten Daten auszuführen

Die KI kann also `data.reduce((sum, row) => sum + row.population, 0)` schreiben und bekommt die echte Summe – ohne je alle 3.000 Zeilen zu sehen.

Stolpersteine in der Praxis
Die Implementierung offenbarte zahlreiche Grenzfälle:
- Code, der Objekte zurückgibt, braucht Klammern
- `console.log` funktioniert in der Sandbox nicht wie erwartet
- Spaltennamen mit Sonderzeichen erfordern Sonderbehandlung
- Manchmal schreibt die KI Code, der stillschweigend fehlschlägt

Jeder dieser Fälle erforderte zusätzliche Anleitungen in der Tool-Beschreibung.

Erkenntnisse

- Daten serverseitig zu halten und KI-Analyse zu ermöglichen ist architektonisch sinnvoll – aber komplex
- Die KI muss eine neue Fähigkeit lernen (JavaScript für Datenanalyse), was eigene Fehlerquellen schafft
- Tool-Beschreibungen sind Dokumentation für eine KI, nicht für Menschen – sie müssen basierend auf beobachteten Fehlern weiterentwickelt werden

---

Herausforderung 1.5: Jeder KI-Client verhält sich anders

Die Entdeckung

Der MCP-Server funktioniert mit verschiedenen KI-Clients: Claude Desktop, Claude.ai und Mistrals Le Chat. Tests zeigten überraschende Unterschiede:

Verhalten	**Claude Desktop**	**Claude.ai**	**Le Chat**
Session bleibt erhalten	✓ Ja	✗ Nein	✗ Nein
Suchanfrage wird verändert	Nein	Nein	Fügt Jahr hinzu

Session-Probleme

Claude Desktop behält die Session über mehrere Tool-Aufrufe. Claude.ai und Le Chat starten bei jedem Aufruf neu – das bricht den Workflow aus Caching und Code-Ausführung.

Veränderte Suchanfragen

Le Chat fügt automatisch das aktuelle Jahr zur Suche hinzu. Wer im Januar 2026 nach Bevölkerungsdaten fragt, sucht plötzlich nach „Bevölkerung 2026" – aber statistische Daten hinken 1–2 Jahre hinterher. Die aktuellsten Daten stammen von 2024, die Suche findet nichts.

Workarounds

- Ein globaler Cache mit Zeitablauf fängt fehlende Sessions ab
- Dokumentation warnt Nutzer vor client-spezifischem Verhalten

Erkenntnisse

- MCP-Server können sich nicht auf einheitliches Client-Verhalten verlassen

- Das MCP-Protokoll lässt zu viel Spielraum – das führt zu Fragmentierung
- Nutzer bekommen je nach Client unterschiedliche Ergebnisse
- Serverseitige Workarounds können helfen, aber nicht alles lösen

---

Zusammenfassung: Berlin Open Data MCP

Der Aufbau des Berlin Open Data MCP zeigte: KI mit Behördendaten zu verbinden ist schwieriger als gedacht. Die Herausforderungen:

1. **Schwache Infrastruktur**: Die Portalsuche entspricht nicht modernen Erwartungen
2. **Formatvielfalt**: „Offene" Daten erfordern spezialisierte Verarbeitung
3. **KI-Sicherheit**: Halluzinationen bei Datenfehlern sind ein ungelöstes Problem
4. **Architektur**: Analyse ohne Kontextüberlauf erfordert durchdachtes Design
5. **Client-Fragmentierung**: Verschiedene KI-Clients verhalten sich unterschiedlich – undokumentiert

---

Teil 2: Datawrapper MCP

Überblick

Der Datawrapper MCP-Server ermöglicht die automatische Erstellung von Datenvisualisierungen über die Datawrapper-API. Die Idee: Nutzer beschreiben, was sie sehen wollen, und die KI erstellt passende Diagramme. Was als überschaubares Projekt mit wenigen Diagrammtypen begann, wuchs schnell – und offenbarte dabei, dass Karten eine ganz eigene Kategorie von Komplexität darstellen.

Herausforderung 2.1: Vom einfachen Balkendiagramm zur Diagramm-Vielfalt

Der Start
Die erste Version unterstützte drei Grundtypen: Balkendiagramme, Liniendiagramme und einfache Karten. Das schien für den Anfang ausreichend.

Die wachsenden Anforderungen
In der Praxis zeigte sich schnell: Nutzer erwarten mehr. „Zeig mir den Unterschied zwischen Männern und Frauen" passt nicht in ein einfaches Balkendiagramm – dafür braucht man

Range-Plots oder Arrow-Plots. „Wie verteilen sich die Sitze im Parlament?" erfordert Election-Donuts.

Die Diagrammtypen wuchsen auf 13 verschiedene Varianten:

Typ	**Anwendung**
Balken/Säulen	Vergleiche zwischen Kategorien
Linien/Flächen	Zeitverläufe
Scatter	Korrelationen
Range/Arrow	Veränderungen, Spannen
Pie/Donut	Anteile am Ganzen
Election-Donut	Sitzverteilungen
Tabellen	Strukturierte Daten
Karten	Geografische Verteilungen

Die Herausforderung der Automatisierung
Jeder Diagrammtyp hat eigene Anforderungen an die Datenstruktur. Ein Scatter-Plot braucht mindestens zwei numerische Spalten. Ein Range-Plot erwartet genau zwei Werte pro Kategorie. Die KI muss diese Regeln verstehen und die richtigen Entscheidungen treffen.

Dazu kamen Detailfragen: Wann braucht ein Diagramm eine Farbcodierung? Wie hoch sollte es sein? Welche Achsenbeschriftung passt? Für jeden Typ mussten „Smart Defaults" entwickelt werden – vernünftige Voreinstellungen, die in den meisten Fällen funktionieren.

Erkenntnisse
- Die Anzahl der Diagrammtypen wächst mit den Anwendungsfällen
- Jeder Typ bringt eigene Validierungsregeln mit
- „Smart Defaults" sind entscheidend für gute Ergebnisse ohne manuelle Konfiguration
- Die Tool-Beschreibung muss der KI helfen, den richtigen Typ zu wählen

---

Herausforderung 2.2: Karten sind keine Diagramme

Die Annahme

Karten schienen zunächst wie ein weiterer Diagrammtyp: Daten rein, Visualisierung raus.

Die Realität
Karten erwiesen sich als fundamental anders. Die erste Hürde: Es gibt verschiedene Kartentypen mit völlig unterschiedlichen Anforderungen.

Kartentyp	**Datenformat**	**Anwendung**
Symbol-Karten	Punkte mit Koordinaten	Standorte markieren
Choropleth-Karten	Flächen mit Werten	Regionen einfärben

Das Scheitern der Automatik
Der erste Ansatz versuchte, den Kartentyp automatisch zu erkennen: Punkte → Symbol-Karte, Polygone → Choropleth. Das funktionierte in Testfällen, scheiterte aber in der Praxis regelmäßig. Die KI wählte den falschen Typ, oder die Erkennung war uneindeutig.

Die Lösung: Weg von der Automatik, hin zur expliziten Angabe. Die KI muss nun den Kartentyp selbst bestimmen – und wenn sie unsicher ist, nachfragen. Das erforderte Änderungen an der Tool-Beschreibung, um die KI zu „zwingen", bei Karten Rückfragen zu stellen.

Choropleth-Karten und das Basemap-Problem
Choropleth-Karten färben Regionen ein – aber dafür muss die Karte wissen, welche Regionen es gibt. Berlin hat verschiedene Gliederungsebenen: Bezirke, Prognoseräume, Planungsräume (LOR). Jede Ebene hat andere Grenzen und andere Schlüssel.

Ein eigener „Basemap-Matcher" wurde nötig: Er analysiert die Daten und erkennt, welche Berliner Gliederungsebene gemeint ist. Enthalten die Daten „Mitte", „Pankow", „Neukölln"? Dann sind es Bezirke. Enthalten sie achtstellige Codes? Dann sind es Planungsräume.

Bounding Boxes und Kartenausschnitte
Jede Karte braucht einen passenden Ausschnitt. Zu weit rausgezoomt: Berlin ist ein kleiner Punkt. Zu nah dran: Teile werden abgeschnitten. Die Lösung: Automatische Berechnung der Bounding Box aus den GeoJSON-Daten, mit etwas Rand drumherum.

Selbst das hatte Tücken: Quadratische Seitenverhältnisse funktionierten besser als die Standard-Rechtecke von Datawrapper, weil Berlin annähernd quadratisch ist.

Erkenntnisse
- Karten sind eine eigene Kategorie mit eigenen Regeln
- Automatische Erkennung klingt gut, scheitert aber oft – explizite Angaben sind zuverlässiger

- Regionale Daten erfordern Wissen über lokale Verwaltungsstrukturen
- Selbst „einfache" Fragen wie der Kartenausschnitt haben überraschende Komplexität

---

Herausforderung 2.3: Veröffentlichen mit Sicherheitsnetz

Das Problem
Datawrapper unterscheidet zwischen Entwurf und Veröffentlichung. Ein Diagramm kann erstellt, bearbeitet und erst dann veröffentlicht werden. Die erste Implementierung veröffentlichte automatisch – aber was, wenn das Ergebnis nicht passt?

Die Konsequenz
Ein falsch konfiguriertes Diagramm landete sofort öffentlich im Datawrapper-Account des Nutzers. Das war besonders problematisch bei Karten, wo kleine Fehler (falscher Kartentyp, fehlende Legende) sofort sichtbar wurden.

Die Lösung: Zweistufiger Workflow
Die Implementierung wurde aufgeteilt:

1. **create_visualization**: Erstellt das Diagramm als Entwurf und liefert eine Vorschau-URL
2. **publish_visualization**: Veröffentlicht erst nach expliziter Bestätigung

Die Tool-Beschreibung weist die KI nun an, nach der Erstellung zu fragen: „Soll ich das Diagramm so veröffentlichen?" Das gibt Nutzern die Chance, Korrekturen anzufordern.

Erkenntnisse

- Irreversible Aktionen brauchen Bestätigungsschritte
- Die KI muss durch Tool-Design zu verantwortungsvollem Verhalten geführt werden
- Vorschau-URLs sind wertvoller als automatische Veröffentlichung

---

Zusammenfassung: Datawrapper MCP

Der Datawrapper MCP zeigt eine typische Entwicklung: Was als einfache API-Integration beginnt, wird durch die Vielfalt realer Anwendungsfälle komplex. Besonders Karten erwiesen sich als eigene Problemdomäne mit Anforderungen, die weit über „noch ein Diagrammtyp" hinausgehen.

Teil 3: Masterportal MCP

Überblick

Der Masterportal MCP-Server generiert fertige Geodatenportale zum Download. Nutzer liefern GeoJSON- oder WFS-Daten, der Server erstellt ein komplettes Masterportal als ZIP-Datei – auspacken, auf einen Webserver legen, fertig. Die Idee ist elegant, die Umsetzung erwies sich als Kampf mit undokumentiertem Verhalten, hartcodierten Pfaden und Konfigurationsdateien, die anders funktionieren als beschrieben.

Herausforderung 3.1: Die Dokumentation stimmt nicht mit der Realität überein

Die Erwartung
Masterportal ist ein etabliertes Open-Source-Projekt der deutschen Verwaltung. Es gibt offizielle Dokumentation, Beispielkonfigurationen und eine aktive Community. Der Plan: Dokumentation lesen, Konfiguration generieren, fertig.

Die Realität
Die erste funktionsfähige Version entstand nicht durch Befolgen der Dokumentation, sondern durch Reverse Engineering einer funktionierenden Installation. Konfigurationsoptionen, die laut Dokumentation existieren sollten, zeigten keine Wirkung. Andere Optionen waren gar nicht dokumentiert, aber notwendig.

Beispiele für Überraschungen:

Erwartung	**Realität**
XYZ-Tiles für Hintergrundkarte	Masterportal unterstützt nur WMS, keine XYZ-Tiles
Layer erscheinen automatisch	Zusätzliche Properties `typ` und `showInLayerTree` nötig
Standard-CSS-Einheiten	Manche Werte erwarten Pixel ohne Einheit, andere mit

Der Durchbruch

Ein funktionierendes Beispiel wurde Zeile für Zeile analysiert und mit der eigenen Konfiguration verglichen. Erst dann wurde klar, welche Kombinationen tatsächlich funktionieren. Die Dokumentation diente danach nur noch als grobe Orientierung – das Beispiel war die Wahrheit.

Erkenntnisse

- Bei komplexer Software ist ein funktionierendes Beispiel wertvoller als Dokumentation
- „Dokumentiert" bedeutet nicht „korrekt dokumentiert"
- Systematisches Reverse Engineering ist manchmal der schnellste Weg

---

Herausforderung 3.2: Koordinatensysteme und Projektionen

Das Problem

Berlin verwendet das Koordinatensystem EPSG:25833 (UTM Zone 33N). Die meisten Web-Kartenwerkzeuge erwarten WGS84 (EPSG:4326) oder Web Mercator (EPSG:3857). Masterportal arbeitet intern mit EPSG:25832.

Das bedeutet: Daten, die von Berliner Quellen kommen, müssen transformiert werden. Und Masterportal muss wissen, in welchem System die Daten vorliegen.

Die Iterationen

- Erste Version: Alles in WGS84 annehmen → Karten zeigten Afrika statt Berlin
- Zweite Version: EPSG:25833 konfigurieren → Hintergrundkarte funktionierte nicht mehr
- Dritte Version: EPSG:25832 für Masterportal, Transformation der Eingabedaten → funktioniert, aber mit deutschem Basemap-Dienst statt OpenStreetMap

Die finale Konfiguration nutzt basemap.de als Hintergrundkarte (funktioniert mit deutschen Koordinatensystemen) und transformiert alle Eingabedaten entsprechend.

Erkenntnisse

- Koordinatensysteme sind eine versteckte Komplexitätsquelle
- Fehler äußern sich oft irreführend (leere Karte, falsche Position)
- Die Wahl des Basemap-Dienstes hängt vom Koordinatensystem ab
- Europäische Verwaltungsdaten nutzen oft andere Systeme als globale Web-Tools

---

Herausforderung 3.3: Versionen, Pfade und Deployment

Das Versions-Problem

Masterportal wird als ZIP-Download bereitgestellt. Der Plan: Version 3.18.0 herunterladen und einbinden.

Das Problem: Version 3.18.0 existierte nicht auf dem Download-Server. Sie wurde angekündigt, aber nie veröffentlicht. Der Fallback auf Version 3.12.0 funktionierte – aber nur nach Anpassung aller Pfade.

Hartcodierte Pfade
Masterportal verwendet Webpack für den Build-Prozess. Webpack hat bestimmte Pfade hartcodiert. Der Versuch, die Ordnerstruktur zu „normalisieren" (z.B. Umbenennung von `3_12_0` zu `current`) führte zu weißen Seiten – die JavaScript-Dateien konnten ihre Abhängigkeiten nicht mehr finden.

Die Lösung: Den Original-Ordnernamen beibehalten, auch wenn er unschön aussieht.

Docker und Railway
Das Deployment auf Railway brachte weitere Überraschungen. Der Download des Masterportal-Runtimes schlug fehl, weil Docker-Layer gecacht wurden. Die Lösung erforderte explizites Cache-Busting und eine Neustrukturierung des Dockerfiles.

Zudem musste das komplette Masterportal-Runtime (~30 MB) ins Docker-Image eingebettet werden, da es zur Laufzeit für jedes generierte Portal gebraucht wird.

Erkenntnisse
- Externe Downloads können jederzeit verschwinden oder sich ändern
- Hartcodierte Pfade in Build-Tools sind gefährlich für die Portabilität
- Docker-Caching kann zu schwer debugbaren Fehlern führen
- Bei generierten Paketen muss das Runtime mitgeliefert werden

---

Herausforderung 3.4: Session-State und mehrere Portale

Das Problem
Der MCP-Server sollte mehrere Layer unterstützen: Nutzer fügen nach und nach Datensätze hinzu, am Ende wird ein Portal mit allen Layern generiert. Das erfordert Session-State – der Server muss sich merken, was bereits hinzugefügt wurde.

Die Komplikation
Beim Deployment auf Railway läuft der Server möglicherweise in mehreren Instanzen. Jede Instanz hat ihren eigenen Speicher. Wenn Anfrage 1 zu Instanz A geht und Anfrage 2 zu Instanz B, ist der State von Anfrage 1 verloren.

Die Lösung
Nach mehreren Debug-Sessions mit ausführlichem Logging wurde der Ansatz vereinfacht: Jede Server-Instanz verwaltet genau eine Session. Das reicht für den typischen Anwendungsfall (ein Nutzer baut ein Portal) und vermeidet die Komplexität verteilter State-Verwaltung.

Zusätzlich wurde das Session-Reset verbessert: Nach dem Erstellen eines Portals wird die Layer-Liste geleert, damit das nächste Portal nicht versehentlich alte Layer enthält.

Erkenntnisse
- Session-State in verteilten Systemen ist komplex
- Die einfachste Lösung, die funktioniert, ist oft die beste
- Explizites Aufräumen verhindert Zustandsverschmutzung zwischen Aufrufen

---

Herausforderung 3.5: Von der Multi-Tool- zur Single-Tool-Architektur

Der erste Entwurf
Die ursprüngliche Architektur hatte mehrere Tools:

- `add_layer`: Fügt einen Layer hinzu
- `configure_map`: Setzt Titel, Zentrum, Zoom
- `generate_portal`: Erstellt das finale ZIP

Das erschien logisch – kleine, fokussierte Werkzeuge, die kombiniert werden können.

Das Problem in der Praxis
Die KI hatte Schwierigkeiten mit dem Workflow. Sie vergaß Layer hinzuzufügen, rief Tools in der falschen Reihenfolge auf oder generierte leere Portale. Die Session-Probleme verschärften das: Wenn zwischen zwei Tool-Aufrufen die Session verloren ging, war alle vorherige Arbeit weg.

Die Lösung
Zusammenfassung zu einem einzigen Tool `create_portal`, das alles in einem Aufruf erledigt:
- Titel und Karteneinstellungen
- Alle Layer mit Daten und Styling
- Sofortige Generierung des ZIP

Das ist weniger flexibel, aber deutlich robuster. Die KI muss nur noch ein Tool mit allen Parametern aufrufen.

Erkenntnisse

- Mehr Tools bedeuten mehr Möglichkeiten für Fehler
- KI-Agenten kommen mit wenigen, mächtigen Tools besser zurecht als mit vielen kleinen
- Bei Session-Problemen ist Zustandslosigkeit die robustere Wahl

---

Zusammenfassung: Masterportal MCP

Der Masterportal MCP zeigt, wie ein konzeptionell einfaches Projekt durch Plattform-Eigenheiten komplex wird. Die Herausforderungen lagen weniger in der eigenen Logik als in der Integration mit einem System, das seine eigenen, oft undokumentierten Regeln hat. Die wichtigste Lektion: Bei fremder Software ist das funktionierende Beispiel wichtiger als die Dokumentation.

Übergreifende Erkenntnisse

Was wir über KI und Daten gelernt haben

1. **KI-Systeme brauchen Leitplanken** – Sie versuchen zu „helfen", auch wenn Scheitern die richtige Antwort wäre
2. **Tool-Design beeinflusst KI-Verhalten** – Weniger, mächtigere Tools sind robuster als viele kleine
3. **Fehlerfälle sind wichtiger als Erfolgsfälle** – Das System muss auch dann sinnvoll reagieren, wenn etwas schiefgeht
4. **Client-Verhalten ist nicht standardisiert** – Verschiedene KI-Clients verhalten sich unterschiedlich, oft undokumentiert

Was wir über Open Data gelernt haben

1. **„Offen" heißt nicht „zugänglich"** – Formate, Koordinatensysteme und Download-Mechanismen schaffen Hürden
2. **Suchfunktionen sind oft schwach** – Selbst etablierte Plattformen haben keine gute Volltextsuche
3. **Dokumentation ist oft veraltet oder unvollständig** – Funktionierende Beispiele sind wertvoller
4. **Lokale Standards erfordern lokales Wissen** – Berliner Verwaltungsgliederung, deutsche Koordinatensysteme

Empfehlungen für ähnliche Projekte

1. **Mit einem funktionierenden Beispiel starten**, nicht mit der Dokumentation
2. **Früh und oft testen** – Die meisten Probleme zeigen sich erst im Einsatz

3. **Fehlerfälle durchdenken** – Was passiert, wenn der Download fehlschlägt? Wenn die Session verloren geht?
4. **KI-Verhalten beobachten und dokumentieren** – Tool-Beschreibungen basierend auf beobachteten Fehlern anpassen
5. **Einfachheit bevorzugen** – Ein robustes Tool ist besser als drei fragile

–

Stand: 19.01.2026
Alsino Skowronnek
Code: https://github.com/alsino/ODIS