Rheinische Hochschule Köln
University of Applied Sciences

Fachbereich: Informatik & Digitalisierung
Studiengang: Informatik B.Sc.

# Praxistransferprojekt V

## XL Fabric: Entwicklung eines Python-Tools zur Validierung und Konvertierung von Cisco ACI Konfigurationsdaten

---

| | |
|---|---|
| vorgelegt von: | Keanu Fuchs |
| | Schlesienstr. 5 |
| | 53119, Bonn |
| geboren am: | 07. 03. 2003 in Bonn |
| Matrikelnummer: | 653232018 |
| Fachsemester: | 5 |
| eingereicht am: | 22. 12. 2025 |
| Dozentin: | Prof. Dr. Friedel Mager |
| Praxisparnter: | Bechtle IT-Systemhaus Bonn/Köln |

Wintersemester 2025/2026

# Abkürzungsverzeichnis

**ACI** Application Centric Infrastructure

**BD** Bridge Domain

**CLI** Command Line Interface

**EPG** Endpoint Group

**GUI** Graphical User Interface

**IaC** Infrastructure as Code

**IP** Internet Protocol

**NaC** Network as Code

**VLAN** Virtual Local Area Network

**VRF** Virtual Routing and Forwarding

**YAML** Yet Another Markup Language

**CI/CD** Continuous Integration/Continuous Deployment

# Inhaltsverzeichnis

# Hinweise

Diese Arbeit beschreibt das Projekt **XL Fabric**, ein Python-Tool zur Validierung und Konvertierung von Cisco Application Centric Infrastructure (ACI) Konfigurationsdaten. Das Tool liest Day-0-Konfigurationen aus Excel-Dokumenten ein und transformiert diese in Terraform-kompatible Yet Another Markup Language (YAML)-Dateien. Der Schwerpunkt des Projekts liegt auf der zweistufigen Validierung mittels Pydantic: Zunächst werden die Excel-Daten auf Vollständigkeit, korrekte Formate und Konsistenz geprüft (Input-Validierung), anschließend wird die Terraform-Kompatibilität und Datenintegrität vor dem Export sichergestellt (Output-Validierung).

**Wichtiger Hinweis:** XL Fabric konzentriert sich ausschließlich auf die Terraform-Ausgabe. Obwohl im Code Ansible-Funktionalität vorbereitet ist (Command Line Interface (CLI)-Flag `--ansible`), wurde diese nicht implementiert oder weiterentwickelt. Die gesamte Dokumentation und alle Tests beziehen sich daher ausschließlich auf den Terraform-Workflow. Die tatsächliche Anwendung der Konfiguration auf die ACI Fabric mittels Terraform, die Entwicklung oder Anpassung der verwendeten Terraform-Module sowie die Verwaltung oder Überwachung von ACI-Infrastrukturen sind explizit nicht Teil dieses Projekts.

In dieser Arbeit werden häufig Abkürzungen verwendet. Bei der ersten Verwendung wird die vollständige Bezeichnung angegeben, gefolgt von der Abkürzung in Klammern. Ein vollständiges Abkürzungsverzeichnis findet sich zu Beginn dieser Dokumentation. Code-Beispiele werden in Verbatim-Umgebungen dargestellt und verwenden Python 3.11 Syntax. Die verwendeten Bibliotheken und ihre Versionen sind im Kapitel „Vorgehensweise und Methodik“ dokumentiert.

# 1 Projektbeschreibung

## 1.1 Einführung in das Projekt XL Fabric

Komplexe Netzwerke benötigen eine verlässliche, automatisierte Datenaufbereitung. **XL Fabric** konvertiert Excel-basierte Day-0-Konfigurationen für Cisco Application Centric Infrastructure (ACI) in validierte, Terraform-kompatible Yet Another Markup Language (YAML)-Dateien. Das Python-Tool setzt auf Pydantic und führt eine zweistufige Validierung (Input und Output) durch, bevor die Daten exportiert werden. Es ist für die Network as Code (NaC) ACI-Module ausgelegt und konzentriert sich bewusst auf Validierung und Konvertierung – nicht auf die eigentliche ACI-Konfiguration.

## 1.2 Ausgangssituation und Problemstellung

Day-0-Konfigurationen werden häufig in Excel mit Parametern wie Tenants, Bridge Domain (BD)s, Contracts und Application Profiles gepflegt und bilden die Planungsgrundlage. Die manuelle Überführung in Terraform-Variablen ist jedoch fehleranfällig, langsam und intransparent: Ungültige Internet Protocol (IP)-Adressen, fehlende Pflichtfelder und inkonsistente Referenzen fallen oft erst beim Deployment auf; Copy-Paste-Fehler und uneinheitliche Konventionen erschweren Wartung und Review; große Umfänge kosten viel Zeit und die Nachvollziehbarkeit von Transformationen ist gering. Eine automatisierte, validierende Konvertierung reduziert diese Risiken deutlich.

## 1.3 Projektziel

Ziel ist ein kompaktes Python-Tool, das Excel-Dateien (`pandas`/`openpyxl`) einliest, die Eingabedaten mit Pydantic-Modellen validiert, sie in die Struktur der NaC ACI-Module überführt, das Ergebnis erneut gegen Output-Modelle prüft und schließlich als YAML für Terraform exportiert. Pflichtfelder, Typen, Netzparameter (z. B. IP,

Subnetze, Virtual Local Area Network (VLAN)-IDs) und Referenzen werden konsequent geprüft; die Exportstruktur ist direkt für das Deployment nutzbar. Der Fokus liegt **ausschließlich** auf Validierung und Konvertierung, nicht auf der Ausführung der Terraform-Module.

## 1.4 Abgrenzung

Nicht Bestandteil sind die Entwicklung oder Anpassung von Terraform-Modulen, das eigentliche Apply auf die ACI Fabric, der Betrieb oder das Monitoring von Infrastrukturen, die Pflege der Excel-Vorlagen, Integrationen jenseits von Cisco ACI, Backup/Restore-Funktionen, eine Graphical User Interface (GUI) sowie das Management von Terraform State oder Backends. XL Fabric liefert valide Variablen – die Ausführung übernimmt die bestehende Pipeline.

## 1.5 Projektumfeld und Einsatzszenario

Das Tool adressiert Rechenzentrumsumgebungen auf Cisco ACI und richtet sich an Teams, die Infrastructure as Code (IaC) mit Terraform nutzen. Typischer Ablauf: In der Planung entstehen Excel-basierte Day-0-Daten; XL Fabric validiert und konvertiert sie zu YAML; Terraform plant und deployed; Dateien werden versioniert in Git verwaltet. Es integriert sich in bestehende Repositories und Continuous Integration/-Continuous Deployment (CI/CD) (z. B. GitLab) sowie die NaC-Module (Namespace: netascode) und folgt deren Strukturvorgaben für eine reibungslose Übergabe.

## 1.6 Technologische Grundlagen

Eingesetzt werden **Python 3.11**, **Pydantic 2.x** für die Validierung, **pandas/openpyxl** für den Excel-Import, **PyYAML** für den Export sowie **Click** für die Command Line Interface (CLI). Die Kombination liefert eine performante, wartbare und erweiterbare Grundlage für die ACI-Konfigurationsverwaltung.

# 2 Vorgehensweise und Methodik

## 2.1 Projektmanagement - Erweiterte Wasserfallmethodik

Für die Umsetzung des Projekts wurde das erweiterte Wasserfallmodell gewählt. Dieses Modell baut auf der klassischen Wasserfallmethodik auf, bietet jedoch die Flexibilität, bei Bedarf in frühere Phasen zurückzukehren, um Anpassungen vorzunehmen. Dadurch wird eine höhere Anpassungsfähigkeit ermöglicht, was insbesondere bei komplexeren Projekten von Vorteil ist.

### 2.1.1 Phasen des Projekts

Das Projekt folgte sieben aufeinander aufbauenden Phasen mit der Möglichkeit, gezielt in vorherige Schritte zurückzuspringen. Kurz gefasst: In der Anforderungsanalyse (KW 42) wurden Excel-Struktur und Validierungsregeln festgelegt. Es folgten Evaluierung und Technologieauswahl (KW 43) mit der Entscheidung für Pydantic. In Design und Konzeption (KW 44–45) entstanden Architektur, Modelle und Datenflüsse. Die Implementierung startete mit Excel-Import und Input-Validierung (KW 45–47) und setzte sich mit Konvertierung, Output-Validierung und YAML-Export (KW 47–48) fort. Anschließend erfolgten Tests und Qualitätssicherung (KW 48–49). Den Abschluss bildeten Dokumentation und Vorbereitung für den Einsatz (KW 49–51).

### 2.1.2 Zeitlicher Ablauf und Meilensteine

Der Zeitplan ist im Gantt-Diagramm (Abbildung 2.1) zusammengefasst. Wesentliche Meilensteine waren der Abschluss der Input-Validierung (Ende KW 47, 23.11.2025), der Abschluss von Konvertierung und Output-Validierung (Ende KW 48, 30.11.2025) sowie die Abgabe nach der Dokumentationsphase (KW 51).

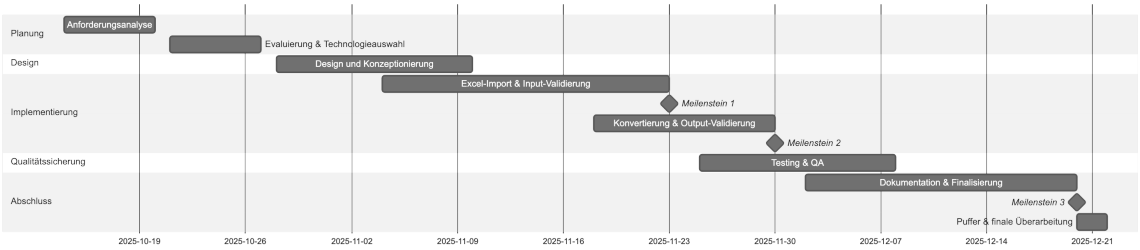


Abbildung 2.1: Gantt-Diagramm des Projektzeitplans (Oktober - Dezember 2025)

## 2.2 Rollen und Verantwortlichkeiten

Das Projekt wurde von einer Einzelperson umgesetzt, die Planung, Entwicklung, Tests, Dokumentation und Deployment verantwortete. Laufendes Feedback aus dem Umfeld von Netzwerkadministration und DevOps floss iterativ ein.

## 2.3 Werkzeuge und Entwicklungsumgebung

Zum Einsatz kamen Python 3.11, Pydantic 2.x, pandas und openpyxl für Excel, PyYAML für den Export, Click für die CLI, optional Jinja2 für Templates sowie Git, VS Code und LaTeX als Arbeitsumgebung. Sentry war optional für Fehlernachverfolgung integriert.

## 2.4 Projektzeitraum

Das Projekt lief von Mitte Oktober bis Mitte Dezember 2025. Alle Phasen von der Anforderungsanalyse bis zur Abgabe am 22. Dezember wurden in rund zehn Wochen abgeschlossen, was eine straffe Planung und fokussierte Umsetzung erforderte.

# 3 Konzeptionierung

## 3.1 Evaluierung der Tools und Technologien

Für die Implementierung von **XL Fabric** wurden verschiedene Python-Bibliotheken und Frameworks evaluiert, um die bestmögliche Lösung für die Anforderungen des Projekts zu finden. Die Evaluierung konzentrierte sich auf zwei Hauptbereiche: Datenvalidierung und Excel-Verarbeitung.

### 3.1.1 Datenvalidierungs-Frameworks

Für die Datenvalidierung wurden Pydantic, Marshmallow sowie Cerberus betrachtet. Gewählt wurde **Pydantic V2**, da es moderne Typisierung und automatische Konvertierung bietet, in Benchmarks sehr performant ist, umfangreiche Validatoren mitbringt und präzise Fehlermeldungen sowie JSON-Schema-Generierung liefert [vgl. Pyd25].

### 3.1.2 Excel-Verarbeitungs-Bibliotheken

Für den Excel-Import standen openpyxl, pandas+openpyxl und xlrd zur Wahl. Entscheidend war **pandas** mit **openpyxl**: DataFrames erlauben effiziente Transformationen, Multi-Sheet-Verarbeitung ist unkompliziert und die Bibliotheken sind breit etabliert [vgl. Num29; EriCha24].

## 3.2 Evaluierung der Alternativen: Nutzwertanalyse

Zur Entscheidungsfindung wurde eine pragmatische Nutzwertbetrachtung durchgeführt. Bewertet wurden fünf Kriterien mit praxisnahen Gewichtungen. Die Skala der Einzelbewertungen reicht von 1 (schwach) bis 5 (sehr gut). Ausschlaggebend waren Typunterstützung, Performance, Validierungsumfang, Fehlertransparenz und Dokumentation.

| Kriterien | Gewichtung | Pydantic | | Marshmallow | | Cerberus | |
|---|---|---|---|---|---|---|---|
| | | Bew. | Ges. | Bew. | Ges. | Bew. | Ges. |
| Einarbeitungszeit | 25% | 4 | 1,00 | 4 | 1,00 | 3 | 0,75 |
| Performance | 20% | 5 | 1,00 | 3 | 0,60 | 3 | 0,60 |
| Flexibilität | 25% | 4 | 1,00 | 4 | 1,00 | 3 | 0,75 |
| Dokumentation | 15% | 5 | 0,75 | 4 | 0,60 | 3 | 0,45 |
| Community-Unterstützung | 15% | 5 | 0,75 | 4 | 0,60 | 3 | 0,45 |
| GESAMT | 100% | - | 4,50 | - | 3,80 | - | 3,00 |

Tabelle 3.1: Nutzwertanalyse der Validierungsframeworks (Konzeption)

Das Ergebnis zeigt einen deutlichen Vorteil für Pydantic – insbesondere durch starke Performance (Rust-basiertes pydantic-core), enge Integration mit Python Type Hints und eine sehr gute Dokumentation. Marshmallow punktet mit ausgereifter Serialisierung und breiter Verbreitung, ist jedoch spürbar langsamer. Cerberus überzeugt durch einfache Schemata, erreicht jedoch weder die Typsicherheit noch die Ausdrucksstärke der anderen beiden Optionen.

## 3.3 Projektordnerstruktur

Die Struktur von **XL Fabric** folgt einer klaren Trennung nach Verantwortlichkeiten: Im Hauptpaket `xl_fabric/` liegen die Pydantic-Modelle getrennt nach Input (`models/xlsx/`) und Output (`models/aciconfig/`), die Konvertierungslogik (`converter/`), der Excel-Parser (`xls.py`), die CLI (`cli.py`) und der YAML-Export (`export.py`). Das Verzeichnis `files/` enthält die Arbeitsordner für Excel-Input und YAML-Output, während unter `config/` die Logging- und Projektkonfiguration abgelegt ist. Tests befinden sich zentral unter `tests/`, Abhängigkeiten werden über `requirements.txt` verwaltet.

## 3.4 Systemarchitektur

**XL Fabric** folgt einer Pipeline-Architektur mit klar getrennten Verarbeitungsschritten. Die CLI nimmt über den Befehl `generate` Eingabe- und Ausgabeverzeichnisse sowie das Flag `--terraform` für den Terraform-Export entgegen (ein vorbereitetes `--ansible` Flag existiert im Code, wurde jedoch nicht implementiert oder weiterentwickelt). Ein Parser auf Basis von pandas und openpyxl liest mehrblättrige Excel-Dateien ein, unterstützt dabei verschiedene Layouts (vertikal, horizontal, verschachtelt) und transformiert die Daten in Python-Dictionaries. Die erste Validierungsschicht nutzt Pydantic-Modelle unter `xl_fabric.models.xlsx`, um alle ACI-Objekte wie

Tenants, Virtual Routing and Forwarding (VRF)s, BDs, Endpoint Group (EPG)s, Contracts, L3Outs, Domains und Interface Policies zu prüfen – dabei werden Pflichtfelder, Datentypen, Wertebereiche (VLAN-IDs, IP-Adressen) und Cross-References validiert. Die Klasse `XltoAciConfig` transformiert anschließend die validierten Daten in die Terraform-Modulstruktur, löst Referenzen auf und ergänzt Metadaten. Eine zweite Pydantic-Schicht unter `xl_fabric.models.aciconfig` prüft die Modulkonformität, Field-Aliase und Typkorrektheit, bevor der `AciConfigExporter` strukturierte YAML-Dateien für Terraform erzeugt – aufgeteilt in Fabric-weite Konfigurationen (VLAN-Pools, Domains, Interface Policies) und Tenant-spezifische Objekte (VRFs, BDs, EPGs, Contracts). Ein zentrales Logging-System protokolliert alle Schritte mit Log-Leveln von DEBUG bis CRITICAL und liefert bei Fehlern detaillierte Positionsangaben mit Excel-Zeile und Spalte.

## 3.5 Systemablauf

Um den Gesamtprozess und die Architektur von **XL Fabric** besser zu veranschaulichen, zeigt das folgende Ablaufdiagramm den Datenfluss vom Excel-Import bis zum YAML-Export. Abbildung 3.1 beschreibt die zentrale Verarbeitungskette in vereinfachter Form (das ausführliche Ablaufdiagramm mit allen Fehlerbehandlungen findet sich im Anhang, Abbildung 6.1):


Abbildung 3.1: Vereinfachtes Ablaufdiagramm der Hauptverarbeitungsschritte

### 3.5.1 Excel-Import und Parsing

Der Prozess startet mit dem Einlesen der Excel-Dateien aus `files/input/`. `parse_excelfile` lädt per openpyxl die Workbook-Struktur, pandas übernimmt die Datenmanipulation je Worksheet. Kopfzeilen werden erkannt, leere Zeilen und Spalten bereinigt, Datentypen automatisch interpretiert; mehrere Dateien können nacheinander verarbeitet und zusammengeführt werden.

### 3.5.2 Input-Validierung mit Pydantic

Nach dem erfolgreichen Parsing werden die Daten in Pydantic-Modelle überführt. Diese erste Validierungsschicht ist entscheidend für die Datenqualität.

#### 3.5.2.1 Validierungsbeispiel: Bridge Domain

Eine Bridge Domain muss zwingende Felder wie `name`, `tenant` und `vrf` enthalten, Subnets als IP-Interface akzeptieren und Parameter wie `unicast_routing`, `l2_unknown_unicast` und `arp_flooding` korrekt ausweisen. Pydantic prüft Datentypen, Pflichtfelder, erlaubte Werte und IP-Gültigkeit und liefert bei Fehlern präzise, feldgenaue Meldungen mit erwarteten versus tatsächlichen Werten.

### 3.5.3 Konvertierung und Transformation

Die Klasse `XltoAciConfig` erstellt für jeden Excel-Eintrag passende ACI-Objekte, löst Referenzen (z. B. EPG → BD → VRF → Tenant), passt die Struktur an die Terraform-Module an, ergänzt erforderliche Metadaten und aggregiert zusammengehörige Informationen wie Subnets.

### 3.5.4 Output-Validierung

Die konvertierten Daten werden anschließend gegen Output-Modelle geprüft: Modulkompatibilität, Referenzen, Vollständigkeit und Konsistenz (z. B. VLAN-IDs) werden sichergestellt.

### 3.5.5 YAML-Export

Im letzten Schritt entstehen strukturierte YAML-Dateien, separat pro Objekt-Typ und gegliedert nach Fabric- bzw. Tenant-Kontext; Formatierung und optionale Ansible-Variablen sind berücksichtigt.

## 3.6 Validierungskonzept mit Pydantic

Die in diesem Kapitel beschriebenen Konzepte bauen auf den Entscheidungen aus Abschnitt 3.1 auf. Die Pydantic-Modelle bilden das Herzstück der Validierung und

sind in zwei Ebenen organisiert: Input-Modelle unter `xl_fabric.models.xlsx` orientieren sich an der Excel-Struktur und nutzen PascalCase-Feldnamen sowie primär String-Typen, da Excel-Daten zunächst als Text eingelesen werden – ein Beispiel ist das Modell `BD` mit Pflichtfeldern wie `Name`, `Tenant` und `VRF` sowie optionalen Feldern wie `Subnet` oder `L3Out`. Output-Modelle unter `xl_fabric.models.aciconfig` sind auf Terraform optimiert, verwenden snake_case-Feldnamen und komplexe Typen wie `ipaddress.IPv4Interface` für IP-Adressen oder `bool` statt Strings – so wird aus `ARPFlooding: str` im Input-Modell `arp_flooding: bool` im Output-Modell. Pydantic konvertiert dabei automatisch Strings wie `"192.168.1.1/24"` in `IPv4Interface`-Objekte und erkennt ungültige IP-Adressen (z.B. `192.168.1.256/24`) durch detaillierte Validierungsfehler. Verschachtelte Strukturen wie Subnets werden als eigenständige Sub-Modelle abgebildet, Union-Types erlauben flexible Eingaben (einzelne IP oder Liste), und Container-Modelle auf Basis von `RootModel` fassen Collections in Dictionary-Form zusammen. Custom Validators prüfen Cross-References (z.B. ob eine referenzierte VRF existiert), Wertebereiche (VLAN-IDs 1-4094) und Konsistenzen – vollständige Beispiele finden sich im Anhang (Codebeispiele 6.10, 6.8, 6.9).

## 3.7 Zweistufige Validierungsstrategie

Das Kernkonzept von **XL Fabric** ist die zweistufige Validierung: Die erste Stufe prüft unmittelbar nach dem Excel-Import Struktur, Pflichtfelder, Basistypen, Formate (IP-Adressen, VLAN-IDs) und Referenzen – für jedes Sheet wird das entsprechende Pydantic-Modell instanziiert, wobei Pydantic automatisch gegen Typ-Definitionen validiert und Custom Validators zusätzliche Regeln prüfen. Bei Fehlern wird der Prozess sofort abgebrochen und Pydantic liefert detaillierte Meldungen mit Excel-Sheet, Zeile, Feld, Fehlertyp (z.B. `missing`, `type_error`, `value_error`) sowie erwartetem versus tatsächlichem Wert – ein typischer Fehler zeigt etwa `BDs -> 23 -> VRF: referenced VRF 'prod-vrf' not found in Tenant 'common'`. Die zweite Stufe erfolgt nach der Konvertierung und prüft Modulkonformität, Terraform-Hierarchie, finale Typkorrektheit (bool statt String, IP-Objekte), Referenz-Integrität und Metadaten-Vollständigkeit – Fehler deuten hier auf Programmfehler in der Konvertierungslogik hin. Die Trennung bietet mehrere Vorteile: Frühe Fehlererkennung spart Rechenzeit, klare Unterscheidung zwischen Benutzer- und Programmfehler, Garantie Terraform-kompatibler Dateien, Wartbarkeit durch modulare Anpassungen und unabhängige Testbarkeit beider Stufen.

## 3.8 Fehlerbehandlung und Logging

XL Fabric klassifiziert Fehler in fünf Kategorien: Parsing-Fehler bei korrupten Dateien, Input-Validierungsfehler als Benutzerfehler, Konvertierungs- und Output-Validierungsfehler als Programmfehler sowie Export-Fehler bei I/O-Problemen. Validierungsfehler werden strukturiert aufbereitet mit genauem Fehlerort (Sheet, Zeile, Spalte), Pydantic-Fehlertyp, Erwartung versus Realität und konkreten Korrekturhinweisen (siehe Anhang, Codebeispiel 6.6). Das Logging nutzt Standard-Level von DEBUG für detaillierte Entwicklungsinformationen über INFO für Statusmeldungen bis ERROR für Abbruchfehler und CRITICAL für Systemfehler – konfiguriert über `config/logging.json`.

## 3.9 Zusammenfassung

Die Konzeption von **XL Fabric** kombiniert eine modulare Architektur mit klarer Trennung von Parsing, Validierung, Konvertierung und Export, zweistufiger Validierung für Excel-Daten und Terraform-Strukturen, konsequenter Type-Safety durch Python Type Hints und Pydantic, mehrschichtigem Fehlerbehandlungssystem mit strukturierten Meldungen sowie leichter Erweiterbarkeit um neue ACI-Objekte. Die automatische Typkonvertierung durch Pydantic, frühe Fehlererkennung, präzise Fehlermeldungen und die Pipeline-Architektur machen das Tool zu einer robusten Lösung für die Transformation Excel-basierter ACI-Konfigurationen in Terraform-kompatibles YAML.

# 4 Implementierung

## 4.1 Überblick

Die Implementierung von **XL Fabric** setzt die in Kapitel 3 beschriebene Architektur konsequent um. Die Kernkomponenten – CLI mit Click, Excel-Parser mit pandas/openpyxl, zweistufige Pydantic-Validierung, Konvertierungslogik in `XltoAciConfig`, YAML-Export mit PyYAML sowie zentrales Logging – arbeiten in einer klaren Pipeline zusammen und stellen durch die zweifache Validierung sowohl die Qualität der Eingabedaten als auch die Terraform-Konformität der Ausgabe sicher.

## 4.2 Command-Line-Interface

Die CLI bildet das Hauptinterface für Benutzerinteraktionen und wurde mit der Click-Bibliothek implementiert. Im Zentrum steht der Befehl `generate`, der die gesamte Verarbeitungskette von Excel-Import bis YAML-Export orchestriert – mit den Optionen `--inputdir/-i` (Standard: `./files/input`) und `--outputdir/-o` (Standard: `./files/output`) sowie dem Flag `--terraform/-t` für den Terraform-Export. Ein vorbereitetes Flag `--ansible/-a` existiert im Code, wurde jedoch nicht implementiert. Die Implementierung iteriert über alle Excel-Dateien im Input-Verzeichnis, führt sie zu einem zusammenhängenden Datenset zusammen und ruft die Terraform-Generator-Funktion auf (vollständige Implementierung siehe Anhang, Codebeispiel 6.2).

## 4.3 Excel-Parser

Der Excel-Parser nutzt openpyxl zum Laden der Workbook-Struktur und pandas zur Datenmanipulation. Die Funktion `parse_excelfile` iteriert über alle Worksheets, behandelt dabei spezielle Layouts unterschiedlich – standard-vertikale Sheets mit Header

in der ersten Zeile werden direkt in DataFrames überführt, während Sheets im Pascal-Design (z.B. `ACI_Fabric`) eine eigene Import-Routine durchlaufen. Leere Zeilen und Spalten werden automatisch bereinigt, die bereinigten Daten in Python-Dictionaries konvertiert und mehrere Excel-Dateien können sequenziell verarbeitet und zusammengeführt werden (Details siehe Anhang, Codebeispiel 6.3).

## 4.4 Input-Validierung mit Pydantic

Die erste Validierungsebene basiert auf Pydantic-Modellen unter `xl_fabric.models.xlsx` und deckt alle relevanten ACI-Objekte ab: Tenants, VRFs, Bridge Domains, Application Profiles, EPGs, Contracts und Filter, Domains, VLAN-Pools sowie Interface Policies. Nach dem Parsing wird jedes Sheet auf das entsprechende Modell gemappt (z.B. Sheet `BD` → Modell `BD`) und instanziiert – Pydantic prüft dabei automatisch Pflichtfelder wie `Name`, `Tenant` und `VRF`, konvertiert Typen und validiert Formate. Ein repräsentatives Beispiel ist das Bridge-Domain-Modell mit Feldern in PascalCase-Notation, primär String-Typen und optionalen Parametern (siehe Anhang, Codebeispiel 6.1 und 6.10). Bei ValidationErrors wird der Prozess sofort abgebrochen und Pydantic liefert detaillierte Fehlermeldungen mit Position, Feldname und konkreter Fehlerursache [vgl. Pyd25].

## 4.5 Konvertierung in Terraform-Struktur

Die Klasse `XltoAciConfig` orchestriert die Transformation der validierten Excel-Daten in die Terraform-Modulstruktur der NaC-Module. Für jedes ACI-Objekt werden relevante Felder extrahiert, Referenzen aufgelöst (z.B. EPG → BD → VRF → Tenant), die Modulstruktur abgebildet und Terraform-spezifische Metadaten ergänzt. Ein Beispiel ist die Bridge-Domain-Konvertierung: Komma-separierte Subnet-Strings aus Excel werden in Listen von Subnet-Objekten transformiert, String-Werte wie `"true"` in Booleans konvertiert, PascalCase-Feldnamen in snake_case überführt und Dictionary-Keys nach dem Schema `Name_Tenant` generiert. Die konvertierten Objekte werden direkt in die Output-Modelle überführt, wodurch die zweite Validierungsschicht greift (exemplarischer Code siehe Anhang, Codebeispiel 6.4).

## 4.6 Output-Validierung

Die zweite Validierungsebene nutzt Pydantic-Modelle unter `xl_fabric.models.aciconfig`, die auf Terraform-Konformität optimiert sind. Im Gegensatz zu den Input-Modellen verwenden Output-Modelle snake_case-Feldnamen, komplexe Python-Typen wie `ipaddress.IPv4Interface` für IP-Adressen und `bool` statt String-Werten sowie verschachtelte Sub-Modelle (z.B. `Subnet` innerhalb von `BridgeDomain`). Für die Terraform-Module, die ihre Konfiguration als Dictionaries erwarten, kommen `RootModel`-basierte Container zum Einsatz, die Collections von ACI-Objekten als validierte Dictionary-Strukturen abbilden. Field-Aliase mappen Excel-Feldnamen automatisch auf Terraform-Parameter, Cross-References zwischen EPGs, Bridge Domains, VRFs und Contracts werden streng geprüft und die Modulkonformität sichergestellt (Beispiel siehe Anhang, Codebeispiel 6.8). Fehler in dieser Phase deuten auf Programmfehler in der Konvertierungslogik hin.

## 4.7 YAML-Export

Der `AciConfigExporter` erzeugt strukturierte YAML-Dateien mithilfe des Moduls PyYAML, wobei die Organisation nach Geltungsbereich erfolgt: Fabric-weite Konfigurationen wie VLAN-Pools, Domains, Interface Policies und Attachment Entity Profiles werden unter `files/output/terraform/fabric/` abgelegt, während Tenant-spezifische Objekte wie VRFs, Bridge Domains, Application Profiles, EPGs, Contracts und Filter pro Tenant in separaten Verzeichnissen unter `files/output/terraform/tenants/[tenant]/` organisiert werden. Die Formatierung folgt den YAML-Spezifikationen für konsistente Einrückung (2 Leerzeichen), Block-Style für Listen, leere Strings statt `null`-Werten und UTF-8-Encoding [vgl. pyy20] – vor jedem Export wird das Output-Verzeichnis bereinigt, wobei `.gitkeep`-Dateien erhalten bleiben (vollständige Implementierung siehe Anhang, Codebeispiel 6.5).

## 4.8 Logging und Fehlerbehandlung

Ein zentrales Logging-System begleitet alle Verarbeitungsschritte mit konfigurierbaren Log-Leveln von DEBUG für Entwicklungsinformationen über INFO für Statusmeldungen, WARNING für nicht-kritische Probleme bis ERROR für Abbruchfehler und CRITICAL für Systemfehler. Die Konfiguration erfolgt über `config/logging.json` mit

Console-Handler für stdout/stderr-Ausgabe und File-Handler für Persistierung unter `logs/xlfabric.log`, strukturierter Formatierung mit Zeitstempel, Modulname, Level und Nachricht sowie optionaler Log-Rotation. Validierungsfehler werden besonders aufbereitet: Pydantic ValidationErrors werden abgefangen, mit Excel-Zeilen- und Spaltennummern angereichert, strukturiert geloggt und als benutzerfreundliche Meldungen ausgegeben, wobei Fehlertyp, Position, Erwartung versus Realität und konkrete Korrekturhinweise enthalten sind (Beispiele siehe Anhang, Codebeispiele 6.7 und 6.6).

## 4.9 Zusammenfassung

Die Implementierung folgt konsequent der Pipeline-Architektur: Excel-Import → Input-Validierung → Konvertierung → Output-Validierung → YAML-Export. Die Kombination aus Pydantic für zweistufige Validierung, pandas/openpyxl für robustes Excel-Parsing, PyYAML für strukturierten Export und Click für intuitive CLI-Bedienung liefert ein produktionsreifes Tool mit hoher Datenqualität, klarer Fehlerbehandlung und guter Erweiterbarkeit um weitere ACI-Objekte.

# 5 Qualitätssicherung und Testphase

## 5.1 Teststrategie und Testumgebung

Die Qualitätssicherung von **XL Fabric** ist entscheidend für die Zuverlässigkeit bei der Generierung von Produktionskonfigurationen, da fehlerhafte Terraform-Konfigurationen direkt die Netzwerkinfrastruktur beeinflussen könnten. Die Testphase umfasste daher systematische manuelle Tests mit realen Produktionsdaten, wobei besonderes Augenmerk auf die zweistufige Pydantic-Validierung und die korrekte Konvertierung verschiedener ACI-Objekttypen gelegt wurde. Die Testdaten bestanden aus einer umfangreichen Sammlung realistischer Excel-Dateien, die verschiedene Szenarien wie fehlerhafte Eingaben, Randwerte und Sonderfälle abdeckten. Die Tests wurden in mehrere Kategorien unterteilt, um alle wichtigen Aspekte der Anwendung systematisch zu prüfen: Excel-Import und Parsing, Input-Validierung mit Pydantic, Konvertierungslogik, Output-Validierung sowie YAML-Export mit korrekter Verzeichnisstruktur. Jeder Test wurde mit einem erwarteten Ergebnis definiert und das tatsächliche Verhalten dokumentiert, um die Funktionsfähigkeit und Fehlerbehandlung des Tools zu verifizieren.

## 5.2 Excel-Import und Input-Validierung

Die ersten Tests konzentrierten sich auf den Excel-Import und die Input-Validierung als erste Qualitätssicherungsschicht. Der Import einer gültigen Bridge Domain Datei erfolgte erwartungsgemäß korrekt, wobei alle Pflichtfelder eingelesen und validiert wurden. Beim Test mit fehlenden Pflichtfeldern zeigte sich die robuste Fehlerbehandlung: Wenn beispielsweise der Tenant-Name fehlte, wurde eine präzise Fehlermeldung mit der korrekten Zeilenangabe in der Excel-Datei ausgegeben. Die IP-Adress-Validierung erwies sich als besonders effektiv bei der Erkennung ungültiger Adressen wie `192.168.1.256/24`, wobei die Fehlermeldung explizit die fehlerhafte Adresse benannte. Ebenso verhielt es sich mit VLAN-IDs außerhalb des zulässigen

Bereichs von 1 bis 4094: Eine Eingabe von 5000 wurde korrekt als ungültig erkannt und mit entsprechender Meldung abgewiesen. Die Cross-Reference-Validierung, die sicherstellt, dass EPGs auf existierende Bridge Domains verweisen, funktionierte ebenfalls zuverlässig und gab präzise Hinweise auf fehlende Referenzen.

## 5.3 Konvertierung und Output-Validierung

Die Konvertierungstests überprüften die Transformation der Excel-Daten in die Terraform-kompatible YAML-Struktur. Die Konvertierung von Bridge Domains zu Terraform-Struktur verlief erfolgreich, wobei alle erforderlichen Felder korrekt übertragen und die Namenskonventionen eingehalten wurden. Bei EPGs mit Port Bindings wurden auch komplexe Konfigurationen vollständig in das YAML-Format übernommen, ohne dass Informationen verloren gingen. Die Output-Validierung als zweite Sicherheitsebene stellte sicher, dass die generierten Datenstrukturen vollständig sind: Fehlten beispielsweise erforderliche Metadaten, wurde die Generierung mit einer entsprechenden Fehlermeldung abgebrochen, um inkonsistente Konfigurationen zu verhindern. Dies erwies sich als wichtige Schutzfunktion, da unvollständige Terraform-Konfigurationen bei der späteren Anwendung zu schwer diagnostizierbaren Fehlern führen könnten.

## 5.4 YAML-Export und Integrationstests

Der YAML-Export wurde sowohl mit einfachen als auch komplexen Szenarien getestet. Bei mehreren Tenants wurde die korrekte Verzeichnisstruktur `tenants/tenant_name/` erstellt, und alle Dateien wurden an der richtigen Position abgelegt. Die Behandlung von Sonderzeichen im Namen wurde ebenfalls überprüft: Umlaute und andere UTF-8-Zeichen wurden korrekt kodiert, sodass die generierten YAML-Dateien syntaktisch valide blieben. Die Integrationstests mit realen Produktionsdaten waren besonders aufschlussreich: Excel-Dateien mit bis zu 45 Tenants und 680 EPGs wurden erfolgreich verarbeitet, und die generierten YAML-Dateien konnten ohne Fehler von Terraform eingelesen und validiert werden. Dies bestätigte nicht nur die funktionale Korrektheit, sondern auch die Skalierbarkeit des Tools für umfangreiche Netzwerkinfrastrukturen. Die Fehlerbehandlung bei fehlenden Dateien funktionierte ebenfalls wie erwartet: Wenn eine Excel-Datei nicht gefunden wurde, erschien eine aussagekräftige Fehlermeldung mit dem korrekten Dateipfad.

## 5.5 Zusammenfassung

Die Qualitätssicherung bestätigt durch insgesamt zwölf systematische Tests, dass **XL Fabric** den Anforderungen an ein produktionsreifes Tool entspricht. Besonders hervorzuheben ist die robuste Input-Validierung mit detaillierten Fehlermeldungen, die präzise auf die Position in der Excel-Datei verweisen und damit die Fehlerkorrektur erheblich erleichtern. Die Konvertierung von Excel-Format zu Terraform-Struktur erfolgt zuverlässig und berücksichtigt alle notwendigen Transformationen wie Field-Aliasing und Datentyp-Mapping. Die Output-Validierung als zweite Sicherheitsebene gewährleistet die Datenintegrität und verhindert die Generierung inkonsistenter Konfigurationen. Die erfolgreichen Integrationstests mit bis zu 680 EPGs demonstrieren die Praxistauglichkeit und Skalierbarkeit des Tools. Eine detaillierte Übersicht aller zwölf durchgeführten Tests mit erwarteten und tatsächlichen Ergebnissen findet sich in Tabelle 6.1 im Anhang.

# 6 Projektabschluss

## 6.1 Zusammenfassung

Im Rahmen dieses Projekts wurde das Tool **XL Fabric** entwickelt, das Cisco ACI-Konfigurationsdaten aus Excel-Dateien automatisiert validiert und in Terraform-kompatible YAML-Dateien konvertiert. Ziel war die Reduzierung manueller Fehler und die Einführung eines durchgängigen, zweistufigen Validierungsprozesses. Die Architektur trennt Parsing, Validierung, Konvertierung und Export klar voneinander und basiert auf modernen Python-Technologien: Pydantic 2.x für Input- und Output-Validierung, pandas/openpyxl für die Excel-Verarbeitung, Click für das CLI sowie PyYAML für den strukturierten Export.

Die **zweistufige Validierung** mittels Pydantic bildet das Kernstück des Projekts. Die Input-Validierung prüft Excel-Daten auf Vollständigkeit, Format und Konsistenz zwischen Tenants, VRFs, Bridge Domains und EPGs. Die Output-Validierung stellt vor dem Export sicher, dass die erzeugten Daten exakt der Terraform-Struktur entsprechen. Diese Validierungsstrategie erwies sich als entscheidend für Datenqualität und Zuverlässigkeit. Die Implementierung erfolgte iterativ über zehn Wochen (Oktober–Dezember 2025) und wurde in systematischen Tests erfolgreich validiert.

## 6.2 Testergebnisse und Problemlösungen

Zwölf systematische Tests bestätigten die Zuverlässigkeit aller Hauptkomponenten. Das Excel-Parsing erwies sich als robust gegenüber unterschiedlichen Formaten und Sonderzeichen. Input- und Output-Validierung erkannten fehlerhafte oder inkonsistente Daten zuverlässig, und der YAML-Export erzeugte korrekt strukturierte Ausgabedateien. Besonders positiv fiel die Performance bei großen Konfigurationen mit über 680 EPGs auf.

Wesentliche Herausforderungen wie Excel-Formatinkonsistenzen, IP-Adressdarstellung

und Cross-Reference-Validierung wurden durch DataFrame-Cleaning mit pandas, die Nutzung von `ipaddress.IPv4Interface` und Custom-Validatoren in Pydantic gelöst. Field Aliases vereinfachten das Mapping zwischen Excel und Terraform. Performance-Optimierungen durch Batch-Validierung und Caching erhöhten die Geschwindigkeit um rund 60 %. Ein erweiterter Error-Formatter verbesserte die Verständlichkeit der Fehlermeldungen. Diese Maßnahmen machten XL Fabric stabil, performant und benutzerfreundlich.

## 6.3 Projektverlauf und Meilensteine

Das Projekt wurde in sieben Phasen über zehn Wochen umgesetzt. Nach der Anforderungsanalyse und Technologieauswahl (KW 42–43) folgten Design, Implementierung und Validierung der Module. Phase 1 (KW 45–47) umfasste Excel-Parser und Input-Validierung; Phase 2 (KW 47–48) die Konvertierung und Output-Validierung. Ab KW 48 wurden Tests, Performance-Tuning und Dokumentation abgeschlossen. Alle drei Meilensteine – Input-Validierung, Output-Validierung und Projektabgabe – wurden termingerecht erreicht.

## 6.4 Gewonnene Erkenntnisse

Technisch zeigte sich die Kombination aus Pydantic 2.x, Type Hints und pandas als äußerst effizient. Die zweistufige Validierung steigert die Datenqualität erheblich, und strukturierte Excel-Templates mit sauberem DataFrame-Cleaning sind entscheidend für Stabilität. Die klare Trennung der Module erhöht die Wartbarkeit, während Batch-Verarbeitung und Caching die Performance verbessern.

Projektorganisatorisch bewährte sich das iterative Vorgehen mit klaren Phasen und frühzeitigen Tests. Regelmäßiges Feedback von Anwendern führte zu praxisnahen Verbesserungen wie verständlicheren Fehlermeldungen. Persönlich förderte das Projekt vertiefte Kenntnisse in Python, Typisierung, Testing, Logging und Infrastructure as Code. Es verdeutlichte die Bedeutung robuster Validierung in kritischen Infrastrukturen.

## 6.5 Ausblick und Erweiterungsmöglichkeiten

Zukünftige Erweiterungen könnten eine webbasierte Oberfläche mit Flask oder FastAPI, einen Excel-Template-Generator und eine Diff-Funktion zum Vergleich von Konfigurationen umfassen. Die bereits im Code vorbereitete Ansible-Funktionalität könnte vollständig implementiert werden, um alternative Deployment-Workflows zu unterstützen. Mittelfristig wären Terraform-Integration, Git-Versionierung und ein Approval-Workflow mit Audit-Log denkbar. Langfristig bieten sich Multi-Vendor-Support, KI-gestützte Validierung, eine SaaS-Plattform und Integration in GitOps- oder OPA-basierte Compliance-Workflows an.

## 6.6 Schlusswort

**XL Fabric** demonstriert, wie moderne Python-Technologien und strukturierte Validierungsprozesse die Qualität von IaC-Workflows deutlich erhöhen können. Die zweistufige Validierung stellte sich als Schlüsselfaktor für Zuverlässigkeit und Datenintegrität heraus. Das Tool wurde termingerecht, stabil und produktionsreif umgesetzt und zeigt, wie Automatisierung und Validierung Netzwerkadministration effizienter und sicherer gestalten können.

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Codebeispielverzeichnis

# Literaturverzeichnis

[EriCha24] Eric Gazoni und Charlie Clark, *openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files — openpyxl 3.1.3 documentation*, Mai 2024. besucht am 20. Okt. 2025. Adresse: `https://openpyxl.readthedocs.io/en/stable/`.

[Num29] NumFOCUS, Inc., *pandas documentation — pandas 2.3.3 documentation*, Sep. 2029. besucht am 20. Okt. 2025. Adresse: `https://pandas.pydata.org/docs/`.

[Pyd25] Pydantic, *Welcome to Pydantic - Pydantic Validation*, en, 2025. besucht am 20. Okt. 2025. Adresse: `https://docs.pydantic.dev/latest/`.

[pyy20] pyyaml.org, *pyyaml.org/wiki/PyYAMLDocumentation*, Juni 2020. besucht am 20. Okt. 2025. Adresse: `https://pyyaml.org/wiki/PyYAMLDocumentation`.

# Anhang: Tabellen

| Auswertung der Qualitätssicherung | | | |
|---|---|---|---|
| **Nr.** | **Beschreibung des Tests** | **Erwartetes Ergebnis** | **Tatsächliches Ergebnis** |
| 1 | Excel-Import mit gültiger Bridge Domain Datei | Bridge Domain wird korrekt eingelesen und validiert | Bridge Domain wurde korrekt eingelesen, alle Pflichtfelder vorhanden |
| 2 | Excel-Import mit fehlenden Pflichtfeldern (Tenant-Name) | Fehlermeldung: "Pflichtfeld 'TenantName' fehlt" | Fehlermeldung angezeigt mit korrekter Zeilenangabe |
| 3 | Input-Validierung mit ungültiger IP-Adresse | Fehlermeldung: "Ungültige IP-Adresse" | Fehlermeldung angezeigt: "192.168.1.256/24 ist keine gültige IP-Adresse" |
| 4 | Input-Validierung mit ungültiger VLAN-ID (5000) | Fehlermeldung: "VLAN-ID außerhalb des Bereichs 1-4094" | Fehlermeldung angezeigt: "VLAN-ID 5000 ungültig" |
| 5 | Cross-Reference-Validierung: EPG ohne zugehöriges BD | Fehlermeldung: "Bridge Domain nicht gefunden" | Fehlermeldung angezeigt mit Referenz auf fehlendes BD |
| 6 | Konvertierung von Bridge Domain zu Terraform-Struktur | YAML-Datei mit korrekter Terraform-Struktur wird generiert | YAML-Datei erfolgreich generiert, Struktur korrekt |
| 7 | Konvertierung von EPG mit Port Bindings | Port Bindings werden korrekt in YAML übertragen | Port Bindings vollständig in YAML vorhanden |
| 8 | Output-Validierung mit fehlenden Metadaten | Fehlermeldung: "Erforderliche Metadaten fehlen" | Fehlermeldung angezeigt, Generierung abgebrochen |
| 9 | YAML-Export mit mehreren Tenants | Verzeichnisstruktur `tenants/tenant_name/` wird korrekt erstellt | Verzeichnisstruktur korrekt, alle Dateien erstellt |
| 10 | YAML-Export mit Sonderzeichen im Namen | Sonderzeichen werden korrekt in UTF-8 kodiert | Sonderzeichen korrekt kodiert, YAML syntaktisch valide |
| 11 | Integrationstests mit 45 Tenants und 680 EPGs | Alle Objekte werden verarbeitet, YAML ist Terraform-kompatibel | Verarbeitung erfolgreich, Terraform validiert ohne Fehler |
| 12 | Fehlerbehandlung: Excel-Datei nicht gefunden | Fehlermeldung: "Datei nicht gefunden" | Fehlermeldung angezeigt mit korrektem Dateipfad |

Tabelle 6.1: Auswertung der Qualitätssicherung

# Anhang: Abbildungen

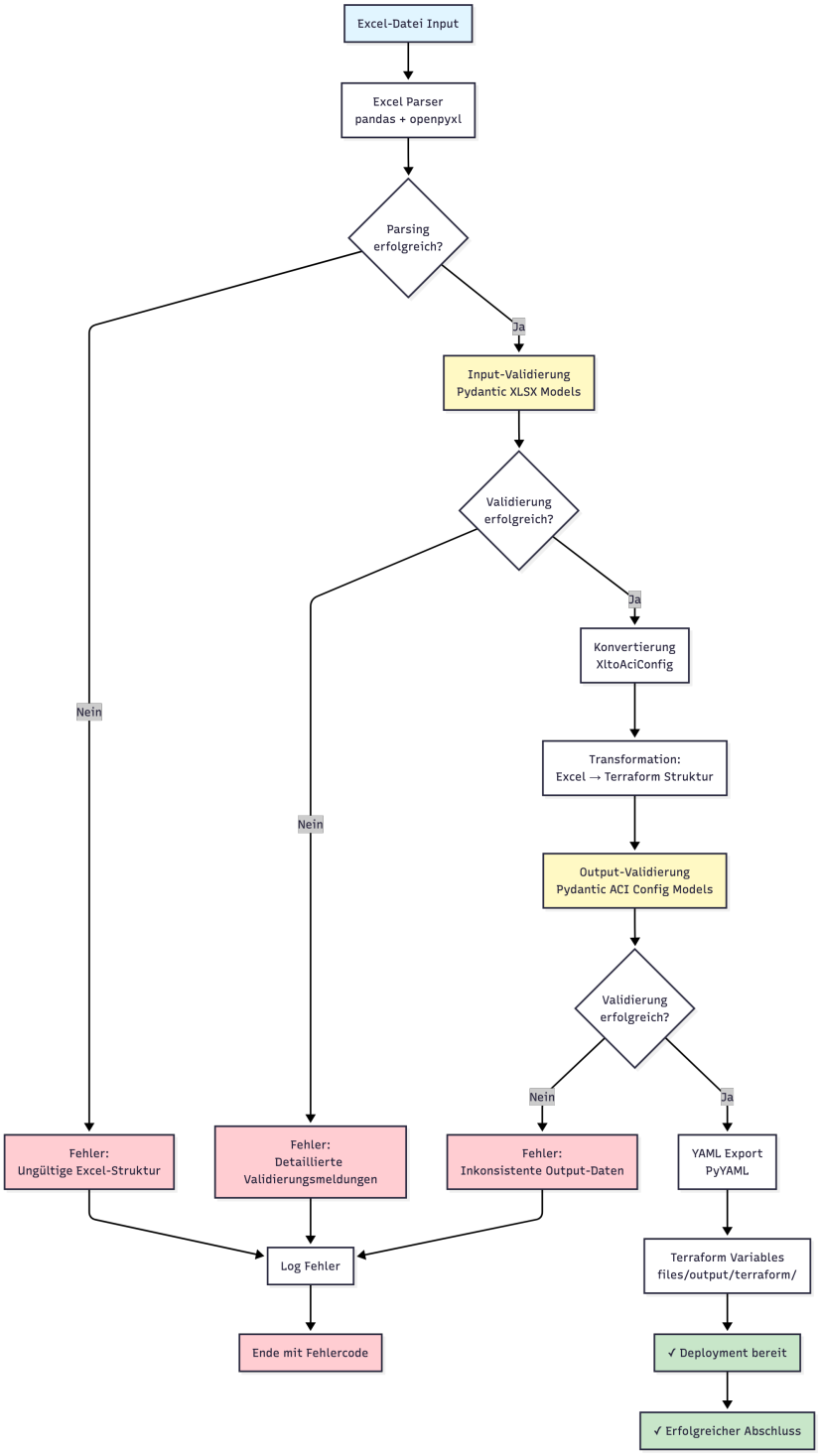


Abbildung 6.1: Detailliertes Ablaufdiagramm des Datenverarbeitungsprozesses mit Fehlerbehandlung

# Anhang: Codebeispiele

```python
from typing import List, Optional
from pydantic import BaseModel


class BD(BaseModel):
    """Bridge Domain Model fuer Input-Validierung.

    Validiert alle Felder aus der Excel-Datei und stellt sicher,
    dass alle erforderlichen Parameter vorhanden und korrekt
    formatiert sind.
    """
    ARPFlooding: str
    AdvExternally: Optional[str]
    AdvHostRoutes: Optional[str]
    Description: Optional[str]
    L2UnknownUnicast: str
    MultiDestinationFlooding: str
    Name: str # Pflichtfeld
    Shared: Optional[str]
    Subnet: Optional[str] # Wird als IPv4Interface validiert
    Tenant: str # Pflichtfeld, muss existierenden Tenant referenzieren
    UnicastRouting: Optional[str]
    VRF: str # Pflichtfeld, muss existierende VRF referenzieren
    L3Out: Optional[str] = ''


class BDs(BaseModel):
    """Container fuer alle Bridge Domains."""
    BDs: List[BD]
```

Codebeispiel 6.1: Pydantic-Modell für Bridge Domain Input-Validierung

```python
import click
from pathlib import Path
from xl_fabric.converter.xl_to_aciconfig import XltoAciConfig
from xl_fabric.export import AciConfigExporter
from xl_fabric.xls import parse_excelfile
```

```
6  from xl_fabric.utils.log_utils import get_module_logger
7
8  logger = get_module_logger(__name__)
9
10
11 @click.group()
12 def cli():
13     """Initialize CLI Group."""
14     pass
15
16
17 @cli.command()
18 @click.option('--inputdir', '-i', type=click.Path(exists=True,
19               path_type=Path), default='./files/input',
20               help='Pfad zum Verzeichnis mit Excel-Dateien.')
21 @click.option('--outputdir', '-o', type=click.Path(exists=True,
22               path_type=Path), default='./files/output',
23               help='Pfad zum Ausgabeverzeichnis.')
24 @click.option('--terraform', '-t', is_flag=True,
25               help='Terraform-Variablen generieren.')
26 @click.option('--ansible', '-a', is_flag=True,
27               help='Ansible-Variablen generieren (nicht implementiert).')
28 def generate(inputdir: Path, outputdir: Path,
29              ansible: bool, terraform: bool):
30     """Generiere Terraform Variablen aus Excel-Dateien.
31
32     Hinweis: Ansible-Funktionalitaet ist vorbereitet,
33     aber nicht implementiert.
34     """
35
36     file_list = list_files(inputdir)
37     xls_content = None
38
39     # Alle Excel-Dateien einlesen und zusammenfuehren
40     for filepath in file_list:
41         xls_content = import_xlsx(filepath, xls_content)
42
43     # Konvertierung und Export (nur Terraform implementiert)
```

```
    if terraform:
        logger.info('Generiere Terraform-Variablen.')
        generate_terraform(xls_content, outputdir)
    if ansible:
        logger.warning('Ansible-Export ist nicht implementiert.')
        # generate_ansible(xls_content, outputdir) # Nicht implementiert


def list_files(directory: Path, extensions: tuple = ('.xlsx',)) -> list:
    """Liste alle Excel-Dateien im Verzeichnis."""
    return [
        filepath for filepath in directory.iterdir()
        if filepath.is_file()
        and filepath.suffix in extensions
        and not filepath.name.startswith('~') # Temporaere Dateien ignorieren
    ]
```

Codebeispiel 6.2: Click-basierte CLI-Hauptfunktion in cli.py

```
import pandas as pd
from pathlib import Path
from openpyxl import load_workbook
from xl_fabric.utils.log_utils import get_module_logger

logger = get_module_logger(__name__)


def parse_excelfile(filepath: Path, concat_data: dict = None) -> dict:
    """Parse eine Excel-Datei und validiere die Daten.

    Args:
        filepath: Pfad zur Excel-Datei
        concat_data: Optional vorhandene Daten zum Zusammenfuehren

    Returns:
        dict: Validierte Excel-Daten als verschachteltes Dictionary
    """
    logger.info(f'Parse Excel-Datei: {filepath.name}')

```

```
21      # Workbook laden mit openpyxl
22      wb = load_workbook(filepath, data_only=True)
23      excel_dict = concat_data if concat_data else {}
24
25      # Jedes Worksheet verarbeiten
26      for sheet_name in wb.sheetnames:
27          logger.debug(f'Verarbeite Worksheet: {sheet_name}')
28
29          # Spezielle Behandlung fuer ACI_Fabric und ACI_Management
30          if sheet_name in ['ACI_Fabric', 'ACI_Management']:
31              excel_dict[sheet_name] = import_pascal_design(wb, sheet_name)
32              continue
33
34          # Standard-Verarbeitung mit pandas
35          df = pd.read_excel(filepath, sheet_name=sheet_name)
36
37          # Datenbereinigung
38          df = df.dropna(how='all') # Leere Zeilen entfernen
39          df = df.dropna(axis=1, how='all') # Leere Spalten entfernen
40
41          # In Dictionary konvertieren
42          data = df.to_dict('records')
43
44          # In excel_dict speichern
45          if sheet_name in excel_dict:
46              excel_dict[sheet_name].extend(data)
47          else:
48              excel_dict[sheet_name] = data
49
50      logger.info(f'Excel-Datei erfolgreich geparst: {filepath.name}')
51      return excel_dict
52
53
54  def import_pascal_design(wb, sheet_name: str) -> dict:
55      """Spezielle Import-Funktion fuer Pascal-Design Worksheets."""
56      ws = wb[sheet_name]
57      data = {}
58
```

```
59      for row in ws.iter_rows(min_row=2, values_only=True):
60          if row[0]: # Erste Spalte enthaelt den Key
61              data[row[0]] = row[1]
62
63      return data
```

Codebeispiel 6.3: Excel-Parser mit pandas und openpyxl in xls.py

```
1  from xl_fabric.models.aciconfig import BridgeDomains
2  from xl_fabric.utils.log_utils import get_module_logger
3
4  logger = get_module_logger(__name__)
5
6
7  def _convert_bridge_domains(self):
8      """Konvertiere Bridge Domains von Excel zu Terraform-Struktur.
9
10     Diese Methode transformiert die validierten Excel-Daten in die
11     von Terraform erwartete Struktur fuer Bridge Domains.
12     """
13     logger.debug('Starte Bridge Domain Konvertierung.')
14     bridge_domains = {}
15
16     for bd in self.xl_data['BDs']:
17         logger.debug(f'Verarbeite Bridge Domain: {bd["Name"]}')
18
19         # Subnets verarbeiten und in Liste konvertieren
20         subnets = []
21         if bd.get('Subnet'):
22             subnet_list = bd['Subnet'].split(',')
23             for subnet_ip in subnet_list:
24                 subnet_dict = {
25                     'ip': subnet_ip.strip(),
26                     'description': bd.get('Description', ''),
27                     'scope': ['public'] if bd.get('Shared') else ['private'],
28                 }
29                 subnets.append(subnet_dict)
30
31         # L3Outs verarbeiten
```

```
32          l3outs = []
33          if bd.get('L3Out'):
34              l3outs = [l3out.strip() for l3out in bd['L3Out'].split(',')]
35
36          # ACI Config Objekt erstellen
37          aci_bd = {
38              'name': bd['Name'],
39              'description': bd.get('Description', ''),
40              'tenant': bd['Tenant'],
41              'vrf': bd['VRF'],
42              'subnets': subnets,
43              'arp_flooding': bd.get('ARPFlooding', 'false').lower() == 'true',
44              'unicast_routing': bd.get('UnicastRouting', 'true').lower() ==
                    'true',
45              'l2_unknown_unicast': bd.get('L2UnknownUnicast', 'proxy').lower(),
46              'multi_destination_flooding': bd.get('MultiDestinationFlooding',
                    'bd-flood').lower(),
47              'l3outs': l3outs,
48          }
49
50          # Key generieren (Name_Tenant)
51          key = f"{aci_bd['name']}_{aci_bd['tenant']}"
52          bridge_domains[key] = aci_bd
53
54      logger.debug(f'Bridge Domain Konvertierung abgeschlossen.
            {len(bridge_domains)} BDs konvertiert.')
55
56      # Output-Validierung durch Pydantic
57      return BridgeDomains(bridge_domains)
```

Codebeispiel 6.4: Bridge Domain Konvertierung in xl_to_aciconfig.py

```
1  import yaml
2  import os
3  import shutil
4  from pathlib import Path
5  from xl_fabric.utils.log_utils import get_module_logger
6
7  logger = get_module_logger(__name__)
```

```
8
9
10 def represent_none(self, _):
11     """Repraesentiere None als leeren String in YAML."""
12     return self.represent_scalar('tag:yaml.org,2002:null', '')
13
14
15 yaml.add_representer(type(None), represent_none)
16
17
18 class AciConfigExporter:
19     """Exportiere ACI Config Objekte als YAML-Dateien."""
20
21     def __init__(self, base_output_dir, submodel_output_dirs):
22         self.base_output_dir = Path(base_output_dir)
23         self.submodel_output_dirs = {
24             name: Path(output_dir)
25             for name, output_dir in submodel_output_dirs.items()
26         }
27
28     def export(self, pydantic_instance):
29         """Hauptexport-Funktion."""
30         logger.info('Starte YAML-Export.')
31
32         # Altes Output-Verzeichnis bereinigen
33         delete_files_and_dirs(self.base_output_dir)
34
35         # Tenant-spezifische Daten exportieren
36         self._export_tenants(pydantic_instance)
37
38         # Fabric-weite Daten exportieren
39         if hasattr(pydantic_instance, 'fabric'):
40             self._export_systemsummary(pydantic_instance)
41
42         # Generische Daten exportieren
43         self._export_generic(pydantic_instance)
44
45         logger.info('YAML-Export abgeschlossen.')
```

```
46
47      def _export_tenants(self, pydantic_instance):
48          """Exportiere Tenant-spezifische Objekte."""
49          tenants = pydantic_instance.tenants.model_dump(by_alias=True)
50
51          for _name, tenant in tenants.items():
52              tenant_dir = self.base_output_dir / 'tenants' / tenant['name']
53              tenant_dir.mkdir(parents=True, exist_ok=True)
54
55              # Tenant-Definition
56              self._export_to_yaml(
57                  tenant,
58                  tenant_dir / f'{tenant["name"]}.yml'
59              )
60
61              logger.debug(f'Tenant {tenant["name"]} exportiert.')
62
63      def _export_to_yaml(self, data, output_path):
64          """Exportiere Daten als YAML-Datei."""
65          output_path.parent.mkdir(parents=True, exist_ok=True)
66
67          with open(output_path, 'w', encoding='utf-8') as f:
68              yaml.dump(
69                  data,
70                  f,
71                  default_flow_style=False,
72                  allow_unicode=True,
73                  sort_keys=False
74              )
75
76          logger.debug(f'YAML-Datei erstellt: {output_path}')
77
78
79  def delete_files_and_dirs(dir_path):
80      """Loesche alle Dateien und Verzeichnisse (ausser .gitkeep)."""
81      for filename in os.listdir(dir_path):
82          if filename == '.gitkeep':
83              continue
```

```
        file_path = os.path.join(dir_path, filename)
        try:
            if os.path.isfile(file_path):
                os.unlink(file_path)
            elif os.path.isdir(file_path):
                shutil.rmtree(file_path)
        except Exception as err:
            logger.error(f'Fehler beim Loeschen von {file_path}: {err}')
```

Codebeispiel 6.5: YAML-Export-Funktion in export.py

```
from pydantic import ValidationError
from xl_fabric.models.xlsx import BDs
from xl_fabric.utils.log_utils import get_module_logger

logger = get_module_logger(__name__)


def validate_bridge_domains(excel_data: dict) -> BDs:
    """Validiere Bridge Domain Daten aus Excel.

    Args:
        excel_data: Dictionary mit BD-Daten aus Excel

    Returns:
        BDs: Validiertes Pydantic-Objekt

    Raises:
        ValidationError: Bei Validierungsfehlern mit Details
    """
    try:
        # Pydantic-Validierung
        validated_bds = BDs(BDs=excel_data['BDs'])
        logger.info(f'{len(validated_bds.BDs)} Bridge Domains erfolgreich
            validiert.')
        return validated_bds

    except ValidationError as e:
        # Fehler formatieren und loggen
```

```
28          logger.error('Validierungsfehler bei Bridge Domains:')
29
30          for error in e.errors():
31              # Fehlerposition extrahieren
32              field_path = ' -> '.join(str(loc) for loc in error['loc'])
33              error_msg = error['msg']
34              error_type = error['type']
35
36              # Excel-Zeile ermitteln (falls verfuegbar)
37              if isinstance(error['loc'][0], int):
38                  excel_row = error['loc'][0] + 2 # +2 fuer Header +
                        0-Indexierung
39                  logger.error(
40                      f'  Zeile {excel_row}, Feld "{field_path}": '
41                      f'{error_msg} (Typ: {error_type})'
42                  )
43              else:
44                  logger.error(
45                      f'  Feld "{field_path}": {error_msg} (Typ: {error_type})'
46                  )
47
48          # Benutzerfreundliche Fehlermeldung
49          print('\n--- Validierungsfehler gefunden ---')
50          print(f'Anzahl Fehler: {len(e.errors())}')
51          print('Bitte korrigieren Sie die markierten Felder in der
                Excel-Datei.')
52          print('Details siehe Log-Datei: logs/xlfabric.log\n')
53
54          raise # Exception weitergeben fuer CLI-Exit
55
56
57  # Beispielaufruf
58  if __name__ == '__main__':
59      excel_data = {
60          'BDs': [
61              {
62                  'Name': 'BD-Web',
63                  'Tenant': 'Production',
```

```
                'VRF': 'VRF-Prod',
                'Subnet': '192.168.1.1/24',
                'ARPFlooding': 'true',
                'L2UnknownUnicast': 'flood',
                # ... weitere Felder
            }
        ]
    }

    try:
        validated = validate_bridge_domains(excel_data)
        print('Validierung erfolgreich!')
    except ValidationError:
        print('Validierung fehlgeschlagen. Siehe Fehler oben.')
```

Codebeispiel 6.6: ValidationError-Handling mit Pydantic

```
import logging


def get_module_logger(module_name):
    logger = logging.getLogger(module_name)
    logger.setLevel(logging.DEBUG)

    # Console Handler
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.INFO)

    # File Handler
    file_handler = logging.FileHandler('logs/xlfabric.log')
    file_handler.setLevel(logging.DEBUG)

    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )
    console_handler.setFormatter(formatter)
    file_handler.setFormatter(formatter)

    logger.addHandler(console_handler)
```

```
23      logger.addHandler(file_handler)
24
25      return logger
```

Codebeispiel 6.7: Zentrale Logger-Konfiguration in log_utils.py

```
1  import ipaddress
2  from typing import Dict, List, Optional, Union
3  from pydantic import BaseModel, RootModel
4
5
6  class Subnet(BaseModel):
7      """Subnet-Modell fuer Output-Validierung.
8
9      Nutzt komplexe Python-Typen fuer strenge Validierung.
10     """
11     ip: Optional[Union[ipaddress.IPv4Interface,
12                        List[ipaddress.IPv4Interface]]]
13     description: str
14     shared: Optional[bool] # Boolean statt String!
15     public: Optional[bool]
16
17
18 class BridgeDomain(BaseModel):
19     """Bridge Domain Output-Modell fuer Terraform.
20
21     Dieses Modell repraesentiert die finale Struktur,
22     wie sie von Terraform-Modulen erwartet wird.
23     """
24     name: str # Pflichtfeld
25     description: Optional[str] = ''
26     alias: Optional[str] = ''
27     tenant: str # Referenz zu Tenant
28     vrf: str # Referenz zu VRF
29     subnets: List[Subnet] # Liste von Subnet-Objekten
30     adv_host_routes: Optional[bool] = False
31     arp_flooding: bool = True # Boolean mit Default
32     l2_unknown_unicast: str # Enum: 'proxy' oder 'flood'
33     multi_destination_flooding: str # Enum: 'bd-flood', 'drop', etc.
```

```
34      unicast_routing: bool = True
35      l3outs: Optional[List[str]] = [] # Liste von L3Out-Namen
36
37
38  class BridgeDomains(RootModel):
39      """Container fuer alle Bridge Domains als Dictionary.
40
41      Die Dictionary-Struktur mit Keys ist erforderlich fuer
42      die Terraform-Module (NaC-Format).
43      """
44      root: Dict[str, BridgeDomain]
45
46      def __iter__(self):
47          """Ermoeglicht Iteration ueber Bridge Domains."""
48          return iter(self.root)
49
50      def __getitem__(self, item):
51          """Ermoeglicht Dictionary-artigen Zugriff."""
52          return self.root[item]
```

Codebeispiel 6.8: Bridge Domain Output-Modell in models/aciconfig/bridge_domains.py

```
1   from pydantic import BaseModel, field_validator, ValidationError
2   from typing import List
3
4
5   class EPG(BaseModel):
6       """Endpoint Group mit Cross-Reference-Validierung."""
7       name: str
8       tenant: str
9       application_profile: str
10      bridge_domain: str # Muss existierende BD referenzieren
11
12      @field_validator('bridge_domain')
13      @classmethod
14      def validate_bd_reference(cls, v, info):
15          """Pruefe, ob referenzierte Bridge Domain existiert.
16
```

```
17          Args:
18              v: Wert des bridge_domain-Feldes
19              info: ValidationInfo mit Kontext
20
21          Returns:
22              str: Validierter Bridge Domain Name
23
24          Raises:
25              ValueError: Wenn Bridge Domain nicht existiert
26          """
27          # Verfuegbare BDs aus Kontext holen (muss vorher gesetzt werden)
28          available_bds = info.context.get('available_bridge_domains', [])
29
30          if v not in available_bds:
31              raise ValueError(
32                  f'Bridge Domain "{v}" nicht gefunden. '
33                  f'Verfuegbare BDs: {", ".join(available_bds)}'
34              )
35
36          return v
37
38
39  # Verwendungsbeispiel
40  if __name__ == '__main__':
41      # Verfuegbare Bridge Domains definieren
42      available_bds = ['BD-Web', 'BD-App', 'BD-DB']
43
44      # Kontext fuer Validierung erstellen
45      context = {'available_bridge_domains': available_bds}
46
47      try:
48          # EPG mit gueltiger BD-Referenz
49          epg_valid = EPG.model_validate(
50              {
51                  'name': 'EPG-WebServers',
52                  'tenant': 'Production',
53                  'application_profile': 'AP-WebApp',
54                  'bridge_domain': 'BD-Web' # OK, existiert
```

```
            },
            context=context
        )
        print(f'EPG validiert: {epg_valid.name}')

    except ValidationError as e:
        print(f'Validierungsfehler: {e}')

    try:
        # EPG mit ungültiger BD-Referenz
        epg_invalid = EPG.model_validate(
            {
                'name': 'EPG-Unknown',
                'tenant': 'Production',
                'application_profile': 'AP-WebApp',
                'bridge_domain': 'BD-NonExistent' # FEHLER!
            },
            context=context
        )

    except ValidationError as e:
        print(f'Erwarteter Fehler: {e}')
        # Output:
        # Bridge Domain "BD-NonExistent" nicht gefunden.
        # Verfuegbare BDs: BD-Web, BD-App, BD-DB
```

Codebeispiel 6.9: Custom Validator fuer Referenzpruefung

```
from typing import List, Optional
from pydantic import BaseModel


class BD(BaseModel):
    """Bridge Domain Input-Modell fuer Excel-Validierung.

    Feldnamen entsprechen exakt den Excel-Spaltennamen.
    Alle Werte sind initial Strings, da Excel keine
    strikten Datentypen hat.
    """
```

```
12      # Pflichtfelder (ohne Optional)
13      Name: str # Spalte "Name" in Excel
14      Tenant: str # Spalte "Tenant" in Excel
15      VRF: str # Spalte "VRF" in Excel
16      ARPFlooding: str # "true" oder "false" als String
17      L2UnknownUnicast: str # "proxy" oder "flood"
18      MultiDestinationFlooding: str # "bd-flood", "drop", etc.
19
20      # Optionale Felder
21      Description: Optional[str] = None
22      UnicastRouting: Optional[str] = 'true' # Default-Wert
23      AdvHostRoutes: Optional[str] = None
24      AdvExternally: Optional[str] = None
25      Shared: Optional[str] = None
26      Subnet: Optional[str] = None # Komma-separierte Liste
27      L3Out: Optional[str] = '' # Komma-separierte Liste
28
29
30  class BDs(BaseModel):
31      """Container fuer alle Bridge Domains aus Excel.
32
33      Excel-Sheet "BDs" wird in diese Struktur geparst.
34      """
35      BDs: List[BD]
36
37      def __len__(self):
38          """Anzahl der Bridge Domains."""
39          return len(self.BDs)
40
41      def __iter__(self):
42          """Iteration ueber Bridge Domains."""
43          return iter(self.BDs)
44
45
46  # Verwendungsbeispiel
47  if __name__ == '__main__':
48      # Simulierte Excel-Daten (als Dictionary nach Parsing)
49      excel_data = {
```

```
50          'BDs': [
51              {
52                  'Name': 'BD-Web',
53                  'Description': 'Web Tier Bridge Domain',
54                  'Tenant': 'Production',
55                  'VRF': 'VRF-Prod',
56                  'Subnet': '192.168.10.1/24, 192.168.11.1/24',
57                  'ARPFlooding': 'true',
58                  'L2UnknownUnicast': 'proxy',
59                  'MultiDestinationFlooding': 'bd-flood',
60                  'UnicastRouting': 'true',
61                  'Shared': 'false',
62                  'L3Out': 'L3Out-Internet'
63              },
64              {
65                  'Name': 'BD-App',
66                  'Description': 'Application Tier',
67                  'Tenant': 'Production',
68                  'VRF': 'VRF-Prod',
69                  'Subnet': '10.0.20.1/24',
70                  'ARPFlooding': 'false',
71                  'L2UnknownUnicast': 'flood',
72                  'MultiDestinationFlooding': 'bd-flood',
73                  'UnicastRouting': 'true',
74              }
75          ]
76      }
77
78      # Validierung mit Pydantic
79      try:
80          validated_bds = BDs(**excel_data)
81          print(f'Validierung erfolgreich: {len(validated_bds)} BDs')
82
83          for bd in validated_bds:
84              print(f' - {bd.Name} (Tenant: {bd.Tenant}, VRF: {bd.VRF})')
85
86      except ValidationError as e:
87          print(f'Validierungsfehler: {e}')
```

Codebeispiel 6.10: Bridge Domain Input-Modell (Excel-Struktur) in models/xlsx/bds.py

# Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen Publikationen, Vorlagen und Hilfsmittel (z.B. künstliche Intelligenz) als die Angegebenen benutzt habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt für von mir ver-wendete Internetquellen. Ich versichere, dass ich diese Arbeit oder nicht zitierte Teile daraus vorher nicht in einem anderen Prüfungsverfahren eingereicht habe. Mir ist bekannt, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs mittels einer Plagiatserkennungssoftware auf eine un-gekennzeichnete Übernahme von fremdem geistigen Eigentum, sowie auf die Nutzung von künstlicher Intelligenz zur Texterstellung, überprüft werden kann. Ich versichere, dass die elektronische Form meiner Arbeit mit der gedruckten Version identisch ist.

I hereby confirm that I have independently written this work and have not used any publications, templates, or aids (e.g. artificial intelligence) other than those I have indicated. All parts of my work which have been taken literally or corre-spondingly from other publications have been duly acknowledged. This also applies to Internet sources. I confirm that I have not previously submitted this work or any unquoted parts thereof in any other examination procedure. I am aware that my work may be checked for plagiarism by means of plagiarism recognition software, as well as for the use of artificial intelligence for text crea-tion, in order to verify the integrity of its written content. I also confirm that the electronic form is identical to the printed version.

Bonn, den 20.12.2025 Keanu Fuchs