

# PREN - Puzzle Simulator

## Inhaltsverzeichnis

1. Einführung und Ziele .....	2
1.1. Aufgabenstellung .....	2
2. Randbedingungen .....	3
2.1. Technische Randbedingungen .....	3
2.2. Randbedingungen zur Puzzle-Geometrie .....	3
2.3. Organisatorische Randbedingungen .....	3
3. Kontext und Abgrenzung .....	4
3.1. Fachlicher Kontext .....	4
3.2. Technischer Kontext (Simulator) .....	4
3.3. Technischer Kontext (phyysischer Roboter - zukünftig) .....	5
3.4. Abgrenzung .....	5
4. Lösungsstrategie .....	6
4.1. Strategien pro Subsystem .....	6
5. Bausteinsicht .....	8
5.1. Gesamtsystem (Ebene 1) .....	8
5.2. Puzzle Generator (Ebene 2) .....	9
5.3. Puzzle Solver (Ebene 2) .....	10
6. Laufzeitsicht .....	11
6.1. Szenario 1: Puzzle-Generierung .....	11
6.2. Szenario 2: Puzzle-Lösung .....	13
7. Verteilungssicht .....	15
8. Querschnittliche Konzepte .....	16
8.1. Konfigurationsverwaltung .....	16
8.2. Fehlerbehandlung .....	16
8.3. Logging und Debugging .....	17
8.4. Reproduzierbarkeit .....	17
9. Architekturentscheidungen .....	18
9.1. 1. Strikte Trennung Generator / Solver .....	18
9.2. 2. Klassische Computer Vision statt Machine Learning .....	18
9.3. 3. Pipeline-Lösungsprozess der Puzzles .....	18
9.4. 4. Generierte Puzzles müssen vielseitig und reproduzierbar sein .....	19
10. Qualitätsanforderungen .....	20
11. Risiken und technische Schulden .....	21
11.1. Technische Schulden .....	21
11.2. Risiken .....	21
11.3. Mögliche Massnahmen .....	21

# 1. Einführung und Ziele

## 1.1. Aufgabenstellung

Das Projekt "Puzzle-Roboter-Simulator" entsteht im Rahmen eines Wettbewerbs, bei dem ein physischer Puzzle-Roboter per Knopfdruck ein Puzzle lösen muss. Der **Simulator** dient der Entwicklung und dem Test der Software-Komponenten in einer kontrollierten, realitätsnahen Umgebung, bevor diese auf die physische Hardware übertragen werden.

Der Simulator umfasst zwei Hauptkomponenten:

1. **Puzzle Generator:** Erzeugt realistische Puzzle-Bilder mit konfigurierbaren Kamera-Effekten (Barrel Distortion, Vignettierung, Rauschen, etc.), um die Aufnahmebedingungen einer RaspberryPi Kamera zu simulieren.
2. **Puzzle Solver:** Analysiert die generierten Bilder und löst das Puzzle durch Bilderkennung, Kantenerkennung, Edge-Matching und Layout-Rekonstruktion.

## 2. Randbedingungen

Dieses Kapitel definiert die technischen, fachlichen und organisatorischen Rahmenbedingungen, innerhalb derer die Architektur des Puzzle-Simulators entwickelt wurde.

### 2.1. Technische Randbedingungen

Randbedingung	Beschreibung
<b>Python 3.11</b>	Programmiersprache für alle Komponenten
<b>Flask REST API</b>	Kommunikation zwischen Generator, Solver und (zukünftig) physischer Hardware
<b>OpenCV</b>	Kernbibliothek für Bildverarbeitung
<b>Raspberry Pi Camera Module 3 Wide</b>	Kamera welche über eine Auflösung von 4608×2592 Pixel verfügt.
<b>Raspberry Pi 5 Model B 8GB</b>	Ziel-Hardware

### 2.2. Randbedingungen zur Puzzle-Geometrie

Randbedingung	Erläuterung
<b>Exakte Anzahl Teile</b>	Das Puzzle besteht aus exakt 6 Teilen
<b>Alle Teile haben Randberührungen</b>	Jedes Teil grenzt mindestens mit einer Kante an den Puzzle-Rand; keine "innenliegenden" Teile ohne Randkante
<b>Nur Rotation, kein Umdrehen</b>	Teile müssen nicht "gespiegelt" werden; Rotation im Bereich 0–360° ausreichend
<b>Schwarze Teile</b>	Einheitliche Farbe für alle Teile; erleichtert Segmentierung durch Kontrast zum weissen Hintergrund
<b>Toleranz zwischen Teilen</b>	Im gelösten Zustand: ~1mm Abstand zwischen Teilen

### 2.3. Organisatorische Randbedingungen

Randbedingung	Beschreibung
<b>Zeitrahmen</b>	Entwicklung im Rahmen eines Semesterprojekts (ca. 14 Wochen)
<b>Wettbewerbsformat</b>	6 Puzzle-Teile welche in gelöstem Zustand einen A5 grossen Rahmen füllen
<b>Knopfdruck-Start</b>	Physischer Roboter muss vollautomatisch arbeiten – Simulator simuliert Teilprozesse davon
<b>Dokumentationspflicht</b>	arc42-Dokumentation als Teil der Projektabgabe

# 3. Kontext und Abgrenzung

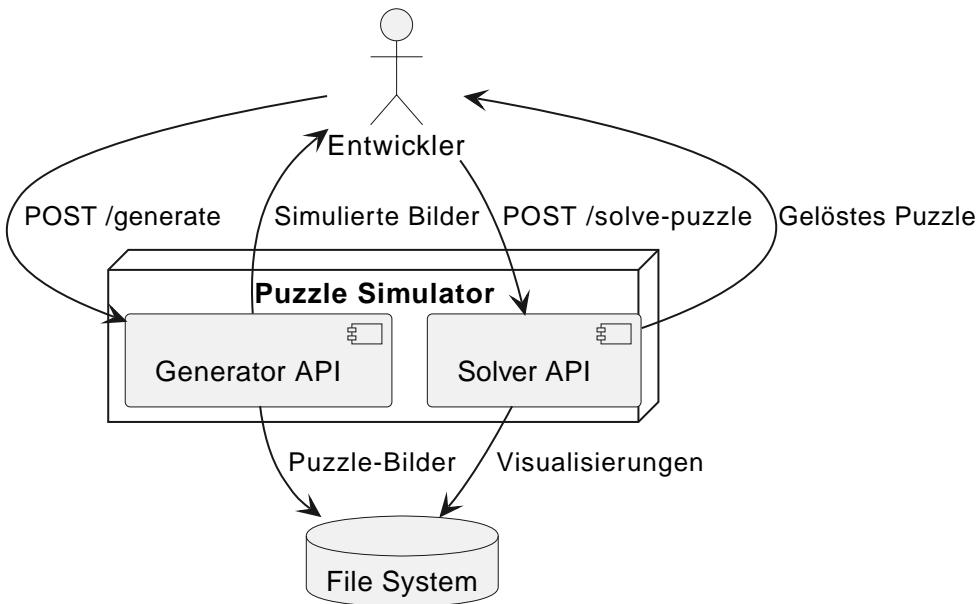
Dieses Kapitel beschreibt das Umfeld des Puzzle-Simulators und grenzt ihn gegenüber Nachbarsystemen ab. Es zeigt die fachlichen und technischen Schnittstellen zu externen Akteuren sowie die klare Definition, was der Simulator leistet und was nicht.

## 3.1. Fachlicher Kontext

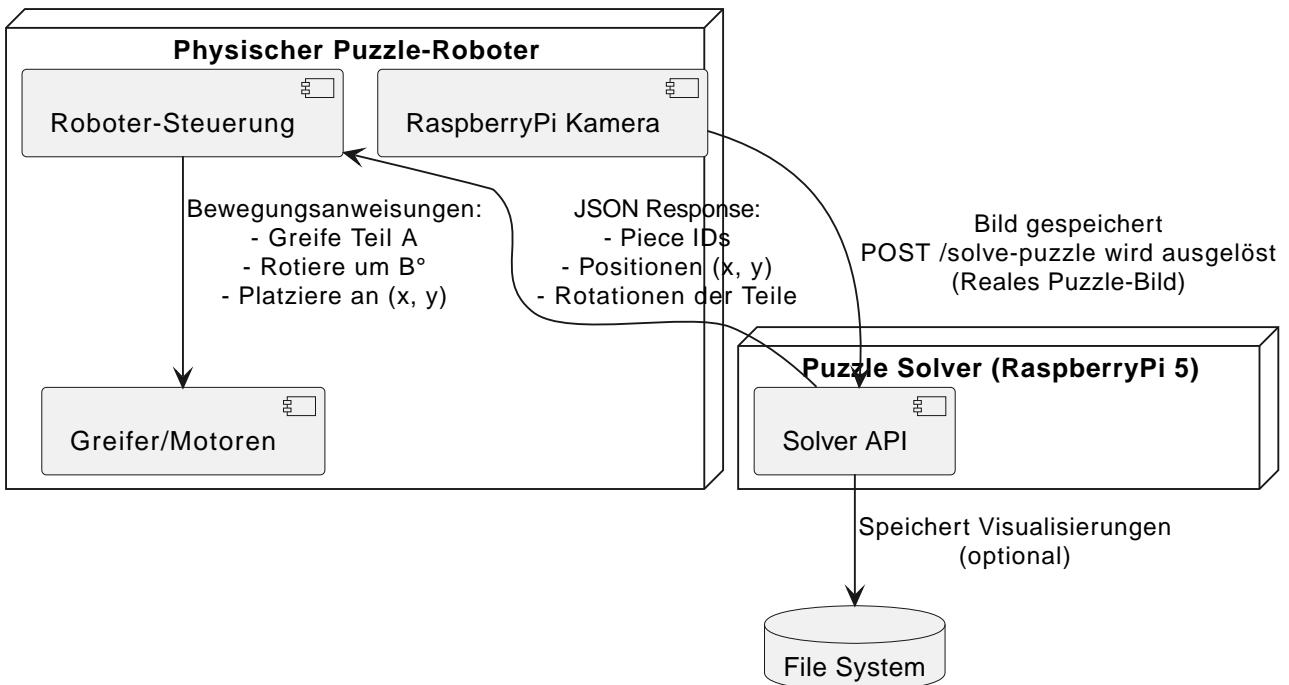
Nachbarsystem	Schnittstelle
<b>Entwickler</b>	REST-API ( <a href="#">/generate</a> , <a href="#">/solve-puzzle</a> , <a href="#">/calibrate</a> ) Steuert Generator und Solver über HTTP-Requests und GUI-Eingaben
<b>Puzzle Generator</b>	Erzeugt synthetische Puzzle-Bilder (4608×2592 px) mit simulierten Kamera-Imperfektionen für realitätsnahe Tests der Lösung des Puzzles
<b>Physischer Puzzle-Roboter</b> (zukünftig)	Übernimmt Solver-Algorithmen, wendet sie auf reale Kamera-Bilder an und berechnet Lage sowie Versatz. Diese Daten werden an die Steuerung der Motoren und des Greifers übergeben.

## 3.2. Technischer Kontext (Simulator)

Entwicklungs-/Test-Kontext mit selbst generierten Puzzles:



### 3.3. Technischer Kontext (phyysischer Roboter - zukünftig)



### 3.4. Abgrenzung

#### 3.4.1. Was der Simulator IST

Funktion	Beschreibung
<b>Software-Entwicklungsumgebung</b>	Ermöglicht frühe Entwicklung und Test von Puzzle-Solver-Algorithmen ohne physische Hardware
<b>Generator realistischer Test-Daten</b>	Simuliert Kamera-Imperfektionen (Verzerrung, Schatten, etc.) für realitätsnahe Tests
<b>Visualisierungs-Tool</b>	Zeigt jeden Pipeline-Schritt (Segmentierung → Edge Detection → Matching → Solving) visuell
<b>Reproduzierbares Test-System</b>	Seed-basierte Generierung garantiert identische Puzzles

#### 3.4.2. Was der Simulator NICHT IST

Kein	Begründung
<b>Ersatz für physische Tests</b>	Generierte Bilder können reale Bedingungen nur approximieren (Reflexionen, Schatten)
<b>Roboter-Steuerungssoftware</b>	Keine Anbindung an Motoren, Greifer oder Hardware-Controller
<b>Produktionsreife Lösung</b>	Proof-of-Concept, nicht Wettbewerbsreif

# 4. Lösungsstrategie

Die Gesamtlösung besteht aus **zwei unabhängigen, aber komplementären Subsystemen**, die über eine gemeinsame Flask-REST-API kommunizieren:

Subsystem	Hauptaufgabe	Technologischer Ansatz
Puzzle Generator	Erzeugt Test-Puzzles mit Kamera-Simulation	Parametrische Geometrie + physikbasierte Kamera Effekte
Puzzle Solver	Löst Puzzle-Bilder durch Computer Vision	Clean Up → Segmentation → Edge Detection → Matching → Solving

## 4.1. Strategien pro Subsystem

### 4.1.1. Puzzle Generator

Wichtigste Entscheidungen:

Aspekt	Lösung	Begründung
Pasende Puzzle-Teile	ReversedCut wrapper invertiert Punktreihenfolge (male/female)	Garantiert passende Puzzle-Teile
Piece Placement	Grid-basiert mit Safe Zones	Verhindert Überlappung trotz Rotation
Shadow Rendering	Nutzt skalierte Boundingboxes für die Performance	Simuliert realistische Bilder des finalen Roboters
Camera Effects	Effekte basierend auf OpenCV und NumPy Bibliotheken	Simuliert tatsächliche Verzerrungen durch die Kamera-Aufnahme

### 4.1.2. Puzzle Solver

Wichtigste Entscheidungen:

Problem	Lösung	Begründung
Bild-Entzerrung	OpenCV <code>undistort()</code> mit calibration_params.json	Muss VOR Segmentierung erfolgen, sonst sind Konturen gekrümmmt
Teile erkennen (Schwarz auf weissem Grund)	Otsu-Verfahren um Puzzleteil zu erkennen	Robuster Algorithmus um Schwellwerte zwischen Vordergrund und Hintergrund zu definieren.
Eck-Erkennung (Tabs/Slots verfälschen Ecken)	Rechteckige Bounding Box mit Snap zum nächsten Konturpunkt	Suboptimale Lösung da es nicht nur Rechteckige Puzzles gibt. Für Rechteckige Puzzles dafür sehr zuverlässig

<b>Problem</b>	<b>Lösung</b>	<b>Begründung</b>
<b>Edge Matching</b> (Passende Kanten finden)	Kurvenkorrelation + filter auf Tab-Slot matches	Tab-Slot filter verbessert Performance
<b>Puzzle-Rekonstruktion</b>	Start in Ecke + Greedy Algorithmus	Einfache Lösung und nicht zu performance intensiv, da alle Teile $\geq 1$ flache Kante haben

### 4.1.3. Visualisierung

**Strategie:** Jede Pipeline-Stufe erzeugt Outputs in `app/static/output/`

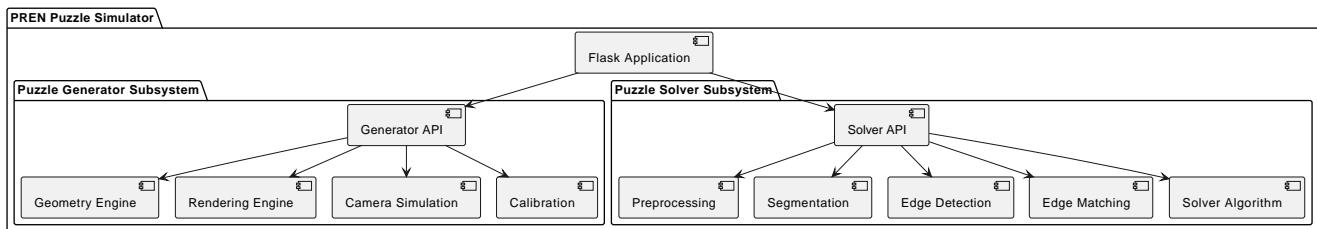
<b>Stufe</b>	<b>Visualisierung</b>	<b>Nutzen</b>
Segmentation	Teile mit transparentem Hintergrund	Kontur-Qualität, Überlappung prüfen
Edge Detection	Konturen + Ecken + Kanten mit Labels	Gewählte Eckpunkte und Kantenklassifikation prüfen
Matching	Side-by-Side Kantenpaare mit Bewertung	False-Positive Identifikation
Solution	Grid-Layout + Rendered Assembly	End-to-End Plausibilität

# 5. Bausteinsicht

Die Bausteinsicht des Systems zeigt auf der ersten Ebene die Hauptkomponenten des Simulators. Die zweite Ebene verfeinert die beiden Subsysteme (Generator und Solver) in ihre internen Bausteine mit deren Verantwortlichkeiten und Abhängigkeiten.

## 5.1. Gesamtsystem (Ebene 1)

### 5.1.1. Übersichtsdiagramm



### 5.1.2. Enthaltene Bausteine

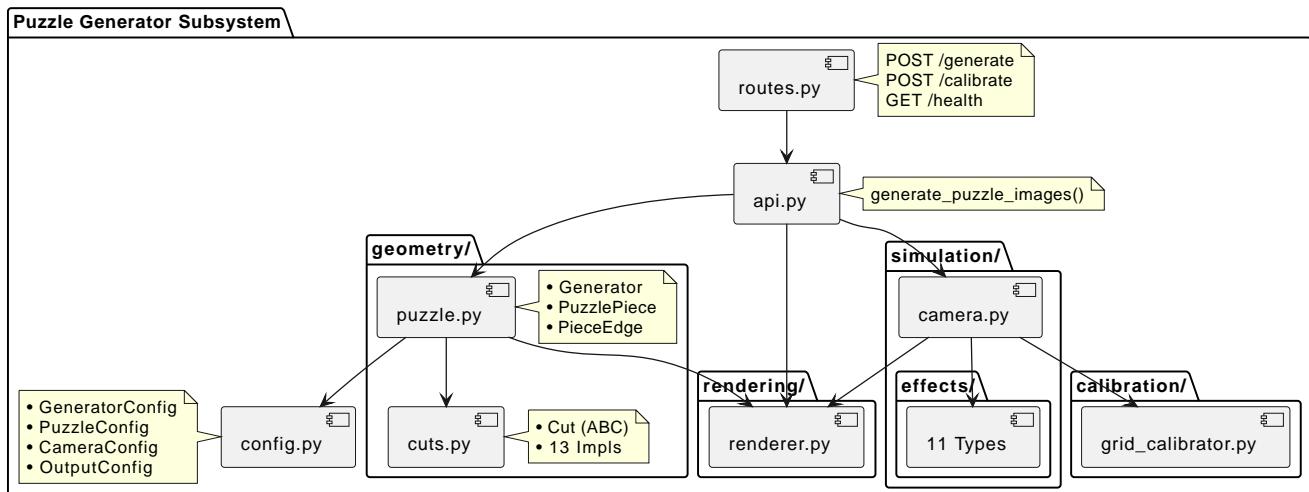
Baustein	Verantwortlichkeit
Flask Application	REST-API Server; Routing; Request/Response Handling; Session Management
Puzzle Generator Subsystem	Erzeugt realistische Puzzle-Bilder mit Kamera-Simulation
Puzzle Solver Subsystem	Analysiert Puzzle-Bilder und rekonstruiert die Lösung

### 5.1.3. Wichtige Schnittstellen

Schnittstelle	Protokoll	Beschreibung
REST-API	HTTP/JSON	<code>/generate</code> , <code>/solve-puzzle/&lt;filename&gt;</code> , <code>/calibrate</code>
File System	Local FS	<code>app/static/output/</code> für Visualisierungen, <code>app/main/img/</code> für Uploads
Configuration	Python Dataclasses	Typsichere Parameter-Objekte (GeneratorConfig, PuzzleConfig, etc.)

## 5.2. Puzzle Generator (Ebene 2)

### 5.2.1. Übersichtsdiagramm

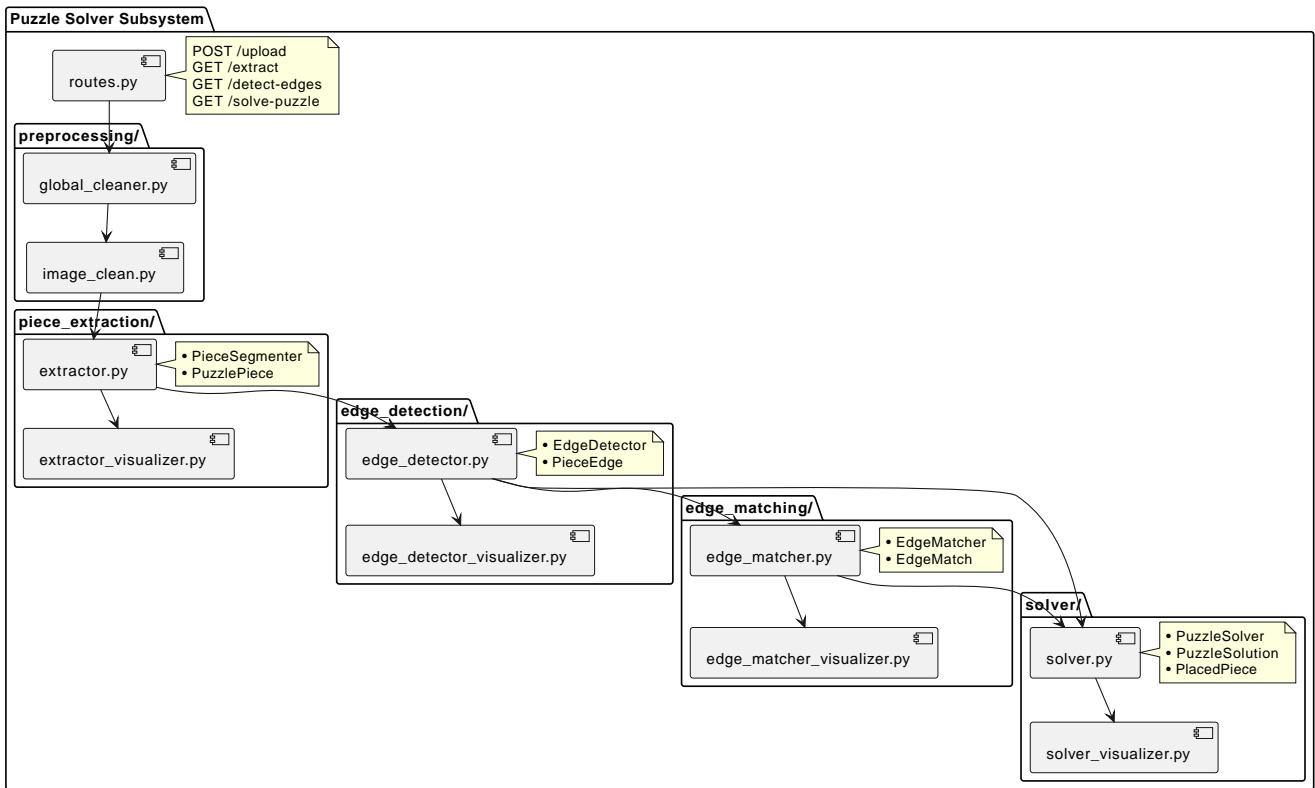


### 5.2.2. Enthaltene Bausteine

Baustein	Verantwortlichkeit
<b>routes.py</b>	Flask-Routen, Request Parsing, Response Serialization
<b>api.py</b>	Business Logic, Orchestrierung der Pipeline
<b>geometry/</b>	Parametrische Puzzle-Generierung, Cut-Algorithmen
<b>rendering/</b>	Bild-Rendering, Schatten
<b>simulation/</b>	Kamera-Effekt-Pipeline
<b>calibration/</b>	Barrel Distortion Kalibrierung
<b>config.py</b>	Typsichere Konfiguration

## 5.3. Puzzle Solver (Ebene 2)

### 5.3.1. Übersichtsdiagramm



### 5.3.2. Enthaltene Bausteine

Baustein	Verantwortlichkeit
<code>preprocessing/</code>	Globale (Kamera) und lokale (Kontur) Korrekturen
<code>piece_extraction/</code>	Segmentierung, Otsu Verfahren
<code>edge_detection/</code>	Eck-Erkennung, Kanten-Extraktion und Kanten-Klassifikation
<code>edge_matching/</code>	Kurvenkorrelation, Tab-Slot-Kompatibilität
<code>solver/</code>	Corner-Start, Greedy Expansion
<b>Visualizers</b>	PNG-Outputs für alle Pipeline-Stufen

# 6. Laufzeitsicht

Dieses Kapitel beschreibt die wichtigsten Ablaufszenarien des Systems zur Laufzeit. Es zeigt, wie die in Kapitel 5 beschriebenen Bausteine zusammenarbeiten, um die Hauptfunktionalitäten zu realisieren.

## **Abgedeckte Szenarien:**

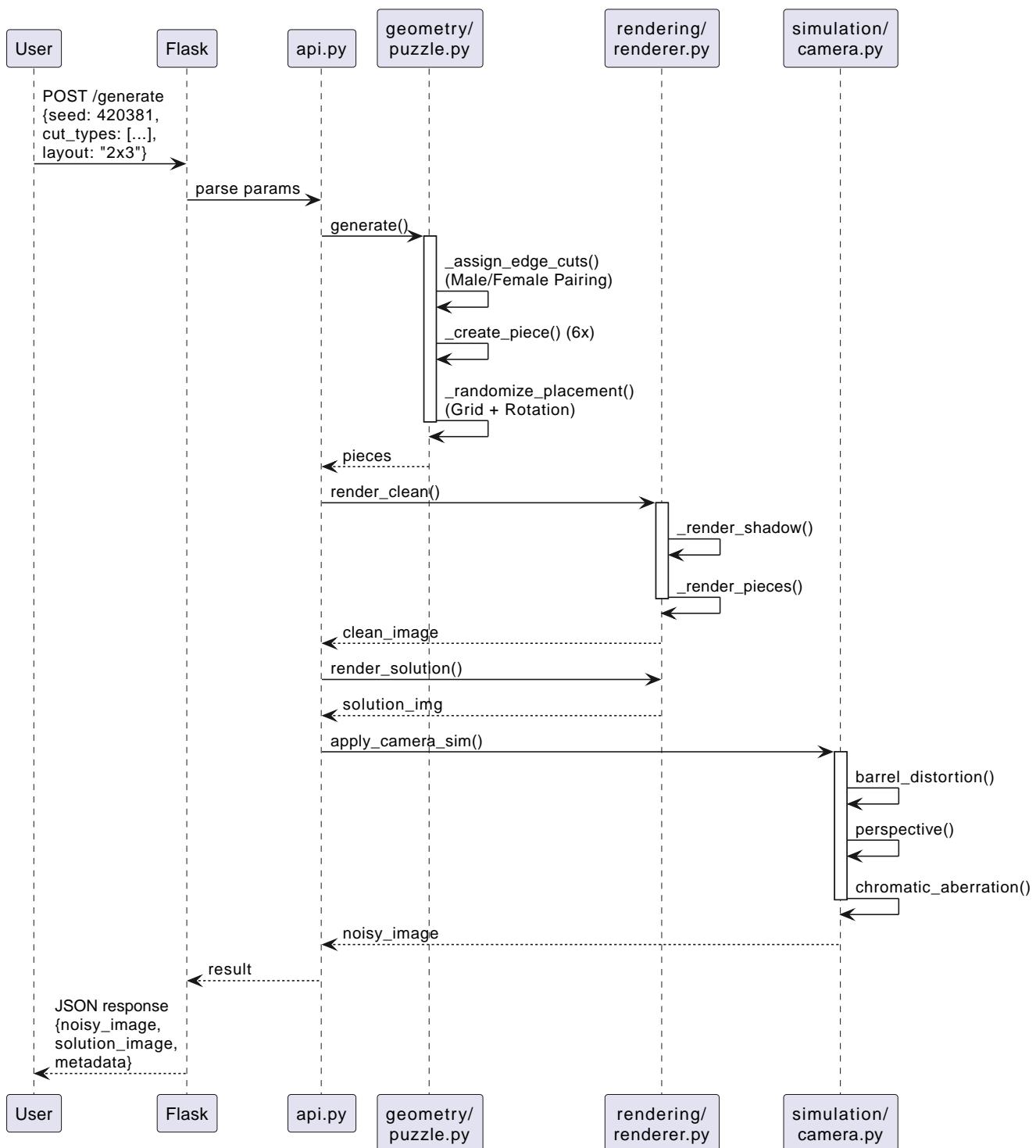
1. Puzzle-Generierung (Generator)
2. Puzzle-Lösung (Solver)

## 6.1. Szenario 1: Puzzle-Generierung

### 6.1.1. Motivation

Ein Entwickler möchte ein neues Test-Puzzle mit realistischen Kamera-Effekten generieren, um den Solver zu testen.

## 6.1.2. Ablauf



## 6.1.3. Wichtige Schritte

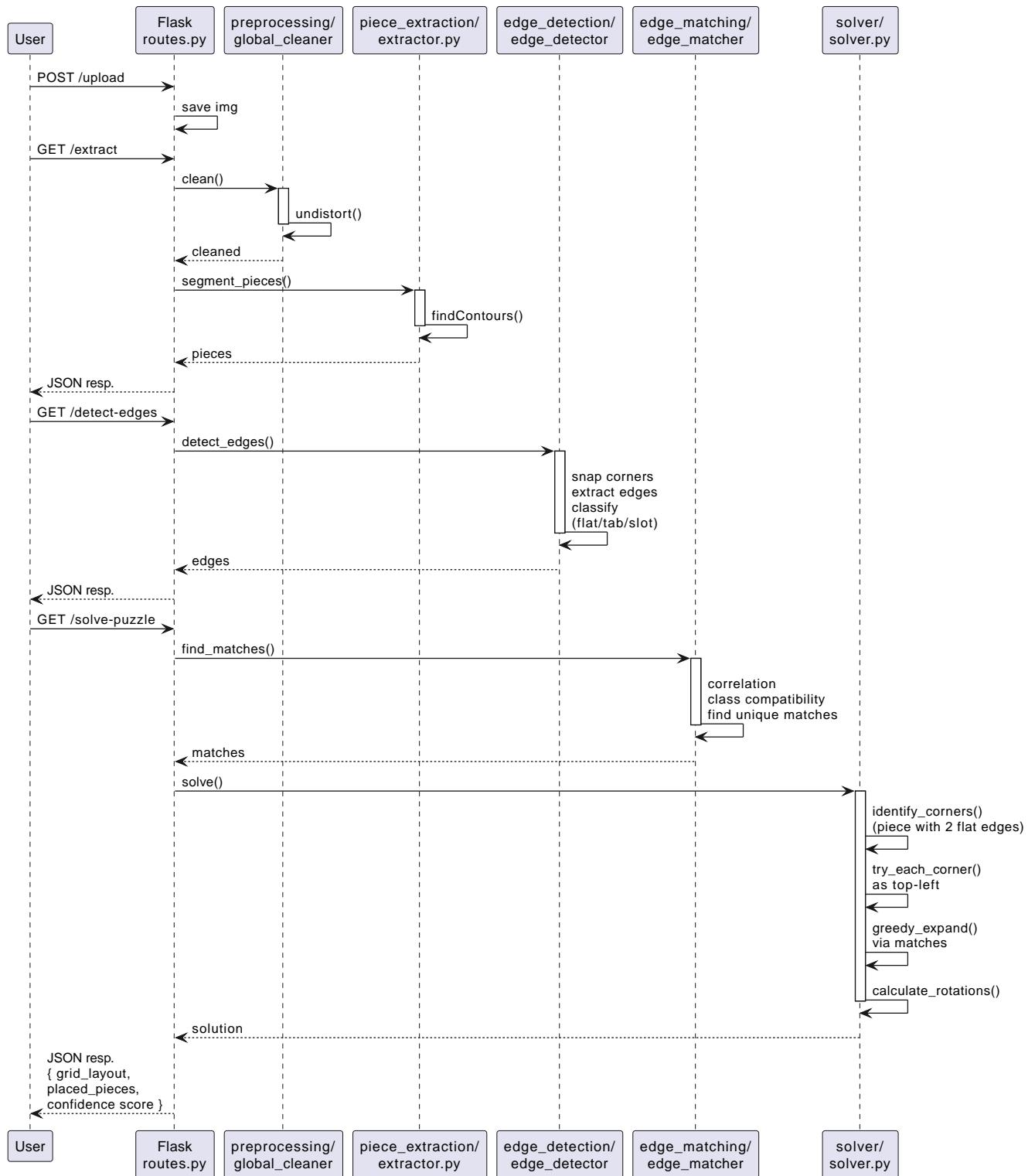
1. Edge Cuts zuteilen
2. Teile generieren
3. Zufälliges platzieren
4. Sauberes Rendering
5. Rendering der Lösung
6. Verzerrtes Rendering

## 6.2. Szenario 2: Puzzle-Lösung

### 6.2.1. Motivation

Ein Entwickler lädt ein vom Generator erzeugtes Puzzle-Bild und möchte die automatische Lösung testen.

### 6.2.2. Ablauf



### **6.2.3. Wichtige Schritte**

1. Global Cleaning
2. Segmentierung
3. Kantenerkennung
4. Kanten-Matching
5. Puzzle lösen

# 7. Verteilungssicht

Dieses Kapitel beschreibt die technische Infrastruktur und Deployment-Architektur des Puzzle-Simulators. Der Simulator läuft vollständig lokal als monolithische Flask-Anwendung ohne externe Abhängigkeiten.

Randbedingung	Erläuterung
<b>Virtualenv</b>	Isolierte Python-Umgebung; es sind keine System-weiten Installationen nötig
<b>Lokaler Betrieb</b>	Keine Abhängigkeiten zur Cloud oder dem Internet; vollständig offline lauffähig.
<b>File-based Storage</b>	Keine Datenbank; Zwischenergebnisse und Visualisierungen lokal in <code>app/static/output</code> als Dateien gespeichert

# 8. Querschnittliche Konzepte

Dieses Kapitel beschreibt übergreifende Konzepte und Lösungsansätze, die in mehreren Bausteinen verwendet werden.

## 8.1. Konfigurationsverwaltung

### 8.1.1. Konzept

Sofern sinnvoll werden alle Parameter über **Python Dataclasses** verwaltet keine und Dictionaries verwendet.

### 8.1.2. Implementierung

Komponente	Dataclasses	Zweck
<b>Generator</b>	<code>GeneratorConfig, PuzzleConfig, RenderConfig, CameraConfig, OutputConfig</code>	Parameter für Geometrie, Rendering, Kamera-Effekte
<b>Solver</b>	Parameter als Query-Strings <code>(min_score, blur_kernel)</code>	Keine komplexen Configs nötig (Pipeline-basiert)

### 8.1.3. Vorteile

- **IDE-Support:** Autocomplete, Type Checking
- **Validation:** Fehler vor/beim Import statt während der Ausführung
- **Dokumentation:** Types sind Self-Documenting

## 8.2. Fehlerbehandlung

### 8.2.1. Strategie

Einheitliche JSON-Error-Responses auf allen REST-Endpunkten.

### 8.2.2. Standard-Format

```
{  
    "success": false,  
    "error": "Human-readable error message",  
    "details": {  
        "code": "ERROR_CODE",  
        "context": {...}  
    }  
}
```

### 8.2.3. Fehlertypen

Typ	Beispiel	HTTP Code
Validation Error	"Invalid parameter: blur_kernel must be odd"	400
Not Found	"File not found: puzzle.jpg"	404
Processing Error	"Could not solve puzzle - insufficient matches"	400
Internal Error	"Unexpected error in edge detection"	500

## 8.3. Logging und Debugging

### 8.3.1. Visualisierungs-Pipeline

**Strategie:** Jede Pipeline-Stufe erzeugt "Debug"-PNG in `app/static/output/` zur Validierung der Outputs des Puzzle-Lösungsprozesses

Stufe	Output-Dateien
Segmentation	<code>pieces_&lt;filename&gt;_&lt;id&gt;.png</code> (RGBA mit transparentem Hintergrund)
Edge Detection	<code>edges_&lt;filename&gt;_&lt;id&gt;.png</code> (Konturen + Ecken + Labels)
Matching	<code>matches_&lt;filename&gt;_&lt;id&gt;.png</code> (Side-by-Side Kantenpaare mit Scores)
Solution	<code>solution_grid_&lt;filename&gt;.png, solution_render_&lt;filename&gt;.png</code>

### 8.3.2. Abruf

Via REST-API: `GET /output/<filename>.png`

## 8.4. Reproduzierbarkeit

### 8.4.1. Seed-basierte Generierung

**Generator** nutzt Seeds um reproduzierbare Puzzles zu ermöglichen. Seed wird random gewählt ausser dieser wird spezifisch angegeben

```
if seed is not None:  
    np.random.seed(seed)
```

**Vorteil:** Exakte Reproduzierbarkeit von Test-Puzzles

# 9. Architekturentscheidungen

Die folgenden Entscheidungen haben die Architektur des Puzzle-Simulators massgeblich geprägt. Sie werden jeweils mit dem zugrundeliegenden Problem, der gewählten Lösung und deren Begründung dokumentiert.

## 9.1. 1. Strikte Trennung Generator / Solver

**Problem:** Entwicklung und Test der Solver-Algorithmen benötigt reproduzierbare, realitätsnahe Puzzle-Bilder.

**Lösung:**

- Generator erstellt Puzzles mit konfigurierbaren Parametern (Seed, Cut-Types, Kamera-Effekte)
- Solver arbeitet ausschliesslich mit Bildern

## 9.2. 2. Klassische Computer Vision statt Machine Learning

**Problem:** Puzzle-Solving könnte mit Machine Learning Ansätzen wie z.B. Deep Learning gelöst werden

**Lösung:**

- Bewusster Einsatz von CV-Algorithmen

**Begründung:**

- **Interpretierbarkeit:** Jeder Schritt ist nachvollziehbar und debuggbar
- **Keine Trainingsdaten:** Keine grossen Puzzle-Datensätze verfügbar
- **Erfahrung:** Wenig Grundwissen und Erfahrung im Deep Learning Bereich

## 9.3. 3. Pipeline-Lösungsprozess der Puzzles

**Problem:** Der Puzzle-Simulator könnte das Puzzle per Knopfdruck in "einem Schritt" lösen, aber dadurch wäre die manuelle testbarkeit der einzelnen Schritte sehr eingeschränkt.

**Lösung:**

- Lösung des Puzzles als Pipeline in einzelnen Teilschritten.

**Begründung:**

- **Visualisierung:** Jeder Schritt erzeugt visualisierte Outputs
- **Fehlerdiagnose:** Problem lokalisierbar in spezifischem Pipeline-Schritt

## 9.4. 4. Generierte Puzzles müssen vielseitig und reproduzierbar sein

**Problem:** Puzzle-Schnittformen müssen vielfältig und passend sein.

**Lösung:** Abstract Base Class `Cut` mit 13 Implementierungen und einer seed basierten Generierung

- **Male/Female Pairing:** `ReversedCut` Wrapper macht es einfach Gegenstücke zu generieren
- **Reproduzierbar:** Seed-basierte Randomisierung

# 10. Qualitätsanforderungen

Die nachfolgende Tabelle definiert die zentralen Qualitätsziele des Puzzle-Simulators in priorisierter Reihenfolge. Sie dienen als Leitplanken für Architektur- und Implementierungsentscheidungen.

Priorität	Qualitätsziel	Beschreibung
1	<b>Realitätsnähe</b>	Die generierten Puzzle-Bilder müssen die tatsächlichen Kamerabedingungen (Verzerrung, Rauschen, Schatten) möglichst exakt simulieren, um realistische Testszenarien zu schaffen.
2	<b>Robustheit</b>	Der Solver muss zuverlässig Puzzles unter verschiedenen Bedingungen lösen können (unterschiedliche Schnittformen, Rotationen, Beleuchtungsverhältnisse).
3	<b>Nachvollziehbarkeit</b>	Alle Pipeline-Schritte (von der Vorverarbeitung bis zur Lösung) müssen visualisiert und nachvollziehbar sein, um Fehleranalyse und Debugging möglichst einfach zu gestalten.
4	<b>Konfigurierbarkeit</b>	Parameter für Puzzle-Generierung und Solver-Algorithmen müssen über REST-API steuerbar sein.
5	<b>Performance</b>	Die komplette Pipeline des Solving sollte in <30 Sekunden ablaufen, damit der physische Roboter im Wettbewerb genügend Zeit hat das Puzzle zu lösen

# 11. Risiken und technische Schulden

## 11.1. Technische Schulden

Technische Schuld	Auswirkung	Priorität
<b>Nur rechteckige Puzzles</b> Edge Detection nutzt <code>minAreaRect</code> → funktioniert nur für 4-Eck-Teile	Im Wettbewerb können andere Formen vorkommen → Solver versagt	Hoch
<b>Kantenlängen-Abhängigkeit</b> Edge Matching vergleicht Kurvenkorrelation ohne Normalisierung	Kanten mit stark unterschiedlichen Längen matchen schlecht → False Negatives	Mittel
<b>Keine Real-Welt-Validierung</b> Solver nur mit generierten Bildern getestet	Unbekannte Fehlerquellen in realen Bedingungen (Reflexionen, Schatten, Farbvariationen)	Mittel

## 11.2. Risiken

Risiko	Mitigation	Wahrscheinlichkeit
<b>Nicht-rechteckige Puzzles im Wettbewerb</b>	Fallback: Zeit für Anpassung einplanen	Hoch
<b>Generator simuliert Kamera ungenau</b>	Reale Testbilder mit RaspberryPi Kamera aufnehmen und damit optimieren	Mittel
<b>Performance auf RaspberryPi 5 ungenügend</b>	Algorithmen optimieren	Niedrig

## 11.3. Mögliche Massnahmen

### 1. Erweitern der Edge Detection

- Implementierung einer form-agnostischen Eckenermittlung (z.B. Douglas-Peucker Simplification)

### 2. Real-Welt-Tests

- Echte Puzzle-Bilder mit RaspberryPi Kamera testen

### 3. Kantenlängen-Normalisierung

- Edge Matching auf relative Krümmung statt absolute Werte umstellen

### 4. Performance-Profilng

- Laufzeitmessungen auf RaspberryPi 5 durchführen und Bottlenecks identifizieren