CV Scanner

ARCHITECTURAL REQUIREMENTS & DESIGN

Version: 4.0.0

Date: September 28th, 2025

Status: Beta

Document Type: Architectural Requirements & design

Project: CV Scanner - Automated CV Analysis System

Team Members

Marcelo Parsotam

Unaisah Hassim

Talhah Karodia

Abdullah Pochee

Ronan Smart

Table of Contents

1 Architectural Structural Design & Requirements

1.1 Model-View-Controller (MVC)

Purpose:

Separates the application into three interconnected components: Model (data), View (user interface), and Controller (business logic).

Justification:

Facilitates organized code, enhances maintainability, and supports parallel development by decoupling user interface and business logic.

1.2 3-Tier Architecture

Purpose:

Divides the system into layers such as Presentation, Business Logic, Data Access, and Data Storage.

Justification:

Promotes separation of concerns, improves scalability, and allows independent scaling of each tier based on load.

1.3 Transaction Script Architecture

Purpose:

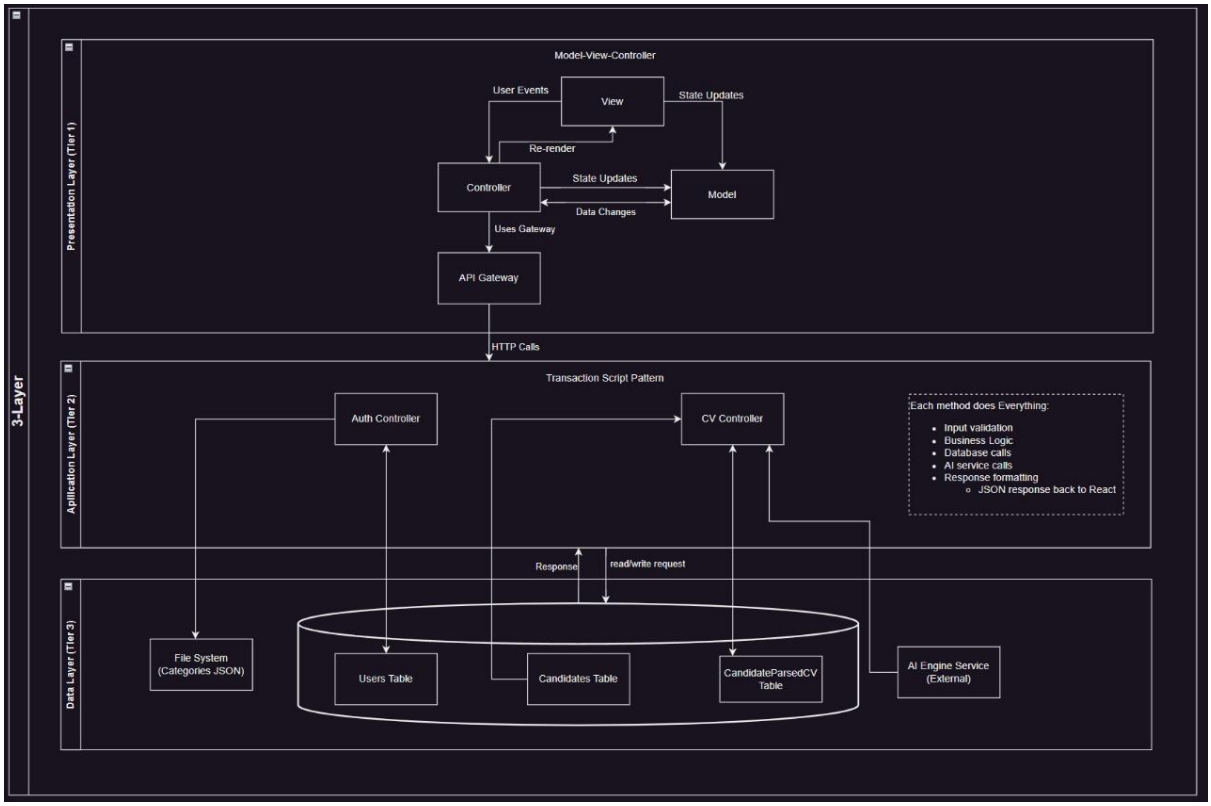
Structures the system around procedural business transactions, where each operation is handled by a single script that contains all the logic required to complete the transaction from start to finish.

Justification:

- **Simplicity:** Straightforward implementation for well-defined business processes without complex object relationships.
- **Performance:** Minimal abstraction overhead for CRUD-heavy operations commonly found in CV processing workflows.

- **Understandability:** Linear code flow mirrors business process documentation, making it easy for developers to follow and maintain.
- **Rapid Development:** Accelerates implementation of new business transactions by avoiding complex domain modelling.
- **Maintainability:** Each transaction is self-contained, making it easier to modify specific business operations without affecting others.

Architectural Diagram



2 Quality Requirements

2.1 User Friendliness (Usability)

The system must be easy to use and understand for all user roles by providing a clean, consistent, and role-specific interface with minimal training required. Tasks such as viewing CVs, uploading new documents, and managing users should require no more than 3 clicks. Tooltips and feedback messages must be available to guide users during their tasks.

Justification

A clean and intuitive interface lowers the learning curve and minimizes user errors. By keeping interactions within **3 clicks** and providing tooltips/feedback, users (Admins, Editors, and Viewers) can efficiently perform tasks with little or no training. This reduces support costs, increases adoption, and improves user satisfaction.

2.2 User Management (Security)

The system must restrict access to the User Management page so that only users with the admin role can access it. This includes Adding, Editing and Removing users from the system of Editor and Viewer Role.

The system shall securely store user passwords using the bcrypt hashing algorithm with salting. This ensures passwords are not stored in plaintext and protects against brute-force and rainbow table attacks.

Justification

Restricting access to the User Management page ensures **role-based access control**, preventing unauthorized changes to sensitive system data. Using **bcrypt with salting** guarantees secure password storage, which is critical to prevent credential theft in case of a data breach. These measures collectively uphold confidentiality, integrity, and compliance with security best practices.

2.3 Different CV document types (Interoperability)

The system must support uploading and parsing of CVs in both .pdf and .doc/.docx formats. Regardless of format, all parsed data must be normalized into a common internal structure to ensure consistent downstream processing. The parser must extract key fields such as Name, Contact Information, Work History, Skills, and Education from both file types.

Justification

Supporting multiple formats (.pdf, .doc/.docx) ensures that users can upload CVs in the most common industry-standard formats, improving accessibility and adoption. Normalizing parsed data into a common structure avoids downstream inconsistencies, allowing all processing modules (scoring, ranking, searching) to work uniformly regardless of input type. This increases interoperability and robustness.

2.4 Reusing Register Logic (Reusability)

The system must promote component and logic reusability. For example, the user registration form and backend endpoint must support both self-registration by users and admin-initiated user creation. This reduces code duplication and simplifies maintenance.

Justification

Reusing the same registration logic for both **self-registration** and **admin-initiated creation** prevents code duplication and inconsistencies in business rules (e.g., password policies, validation checks). This reduces maintenance overhead, ensures uniform behaviour across the system, and simplifies testing since only one logic flow needs to be verified.

3 Deployment Model

Target Environment

The CV Scanner system will be deployed in a cloud-hosted environment, primarily leveraging Microsoft Azure services. It must also remain hostable on a native IIS instance to support hybrid or on-premises hosting if required. The core services will be deployed on Azure App Services and will connect to a managed Azure SQL Database for persistent storage.

This hybrid deployment flexibility ensures the system can be hosted in:

- Cloud (Azure App Services + Azure SQL DB) — for scalability and high availability.
- On-premises IIS instance — for organizations requiring self-hosted environments or compliance with internal IT policies.

Deployment Topology

The system follows a multi-tier, containerized microservices topology, ensuring scalability, maintainability, and modularity. The deployment is broken down as follows:

Presentation Tier (Frontend)

- Built with React (JSX project).
- Can be hosted as:
  - Azure App Service (Web App) for cloud deployments.
  - IIS Static Web Server for on-prem hosting.
- Provides a role-specific UI for Admins, Editors, and Viewers.

Application Tier (Backend & AI Services)

- Backend API Service (Java Spring Boot)
  - Packaged as a Docker container.
  - Deployable on Azure App Services (Linux Web App) or in IIS (WAR/JAR deployment via Tomcat integration).
  - Exposes RESTful endpoints for CV management, authentication, and user operations.

- AI Parsing Service (Python + BART model)
  - Containerized service handling CV parsing and normalization.
  - Exposed via REST API within the cluster.
  - Deployable in Azure App Services (Custom Container) or in an IIS-hosted Docker runtime for hybrid deployments.
  - Scales independently to meet workload demands.
- Authentication Service
  - Responsible for login, role-based access, and bcrypt password hashing.
  - Deployable as a separate backend microservice in Azure App Services or alongside the backend API within IIS.

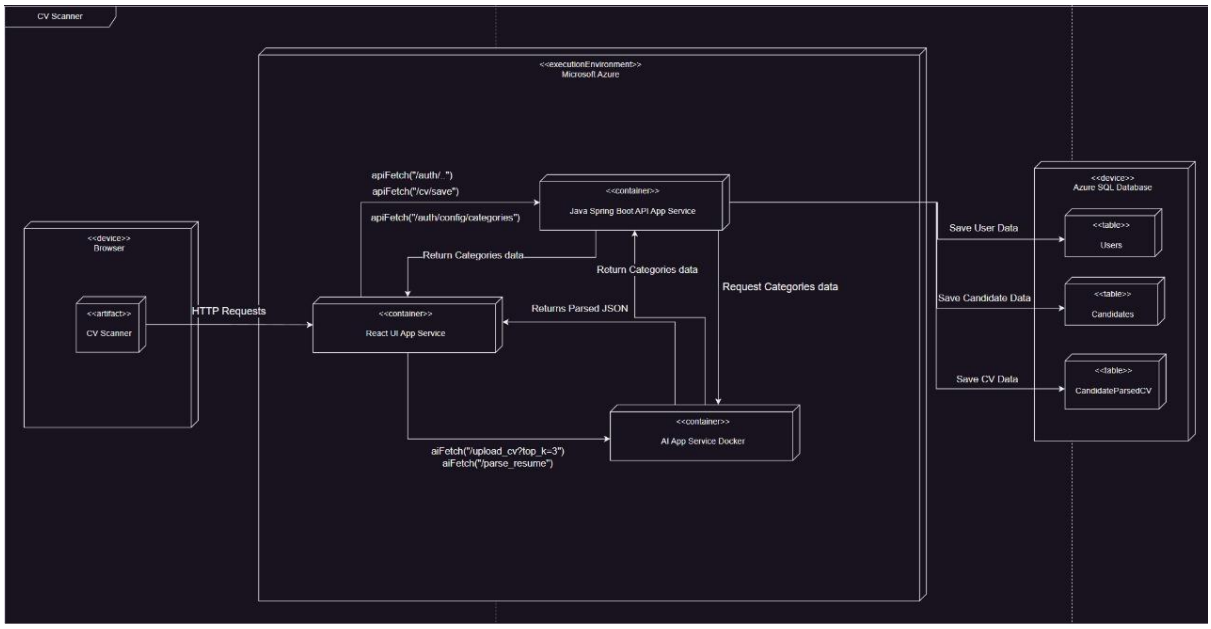
Data Tier (Database)

- Azure SQL Database as the primary data store.
- Secured with firewall rules, IP restrictions, and TLS encryption.
- Tied to the system via connection strings stored in environment variables.
- For on-prem IIS deployments, fallback to SQL Server on-premises can be configured.

Deployment Tools & Platforms

- **Docker**: Used to containerize the backend, AI parsing, and authentication services.
- **Kubernetes (AKS)**: Provides orchestration in Azure for scaling, rolling updates, and high availability.
- **Azure App Services**: Primary hosting environment for frontend, backend, and AI services.
- **IIS**: Provides on-premises hosting compatibility for organizations not fully cloud-enabled.
- **Azure SQL Database**: Fully managed relational database service.

Deployment Diagram



Key Communication Paths:

1. External Communications:

- **Browser ↔ React UI App Service** - HTTPS requests for the web application
- **Browser ↔ Java Spring Boot API** - Direct API calls (REST/HTTPS)

2. Internal Service Communications:

- **React UI ↔ Java Spring Boot API** - Frontend-to-backend API calls
- **Java Spring Boot API ↔ AI App Service** - CV parsing and AI processing requests
- **Java Spring Boot API ↔ Azure SQL Database** - Database operations

3. Authentication Flows:

- **React UI ↔ Java Spring Boot API** - Auth configuration requests (/auth/config/categories)
- **Java Spring Boot API ↔ Auth Service** - User authentication/authorization

Specific API Endpoints Shown:

- aiFetch("/upload_cv?top_k=3") - CV upload to AI service
- aiFetch("/parse_resume") - Resume parsing request
- apiFetch("/auth/config/categories") - Auth configuration
- apiFetch("/cv/save") - Save CV data
- Various auth endpoints (/auth/...)

Data Flow Directions:

- **To Database:** "Save User Data", "Save Candidate Data", "Save CV Data"
- **From Services:** "Return Categories data", "Returns Parsed JSON"

Protocol Specifications:

- **External:** HTTPS
- **Internal:** REST APIs (HTTP/HTTPS)
- **Database:** Secure TLS connection to Azure SQL

Support for Quality Requirements

Scalability

- Containerized services scale independently (e.g., AI Parsing service can scale faster than Auth service).
- Azure App Services + AKS support autoscaling.

Reliability

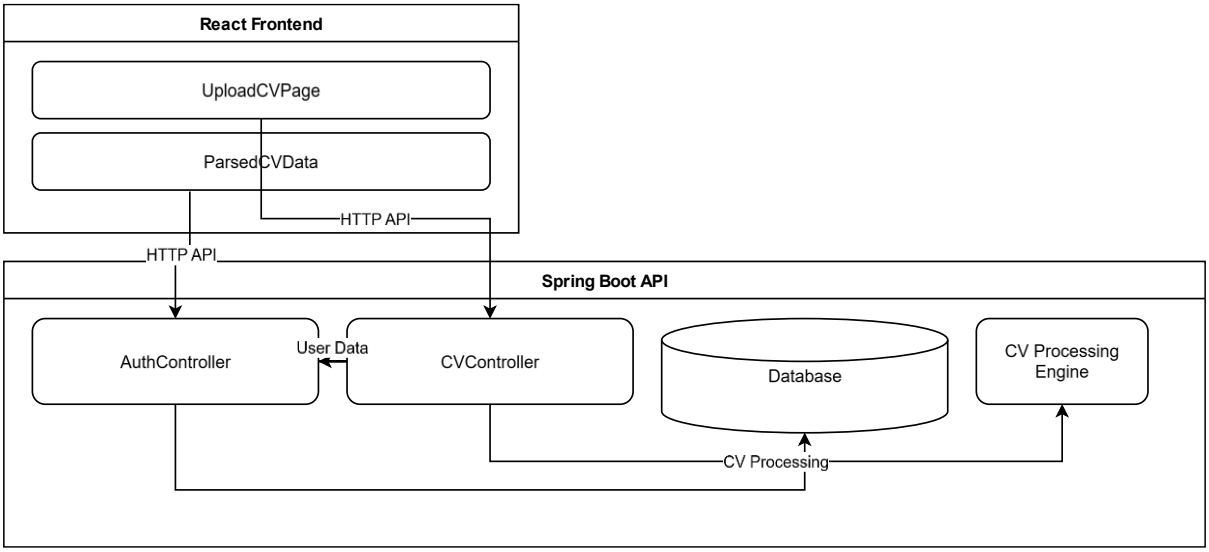
- Azure App Services provide redundancy and failover across regions.
- Azure SQL DB includes high-availability failover clusters.
- IIS deployments can use Windows Server clustering for redundancy.

Maintainability

- Dockerized microservices allow updates without redeploying the entire system.
- Clear service boundaries (backend, AI parsing, authentication) reduce regression risks.
- The Strategy Pattern in the AI parsing logic allows easy extension with new CV processing methods.

4 Service Contracts

4.1 UI To API Service Contract



Communication Channels & Protocols

Component	Protocol	Port	Description
React UI ↔ Spring API	HTTPS/HTTP	8081	REST API for all UI interactions
Spring API ↔ Database	JDBC	3306	MySQL database access
Spring API ↔ CV Engine	HTTP	5000	Python CV processing service

Key Endpoints & Request Formats

Authentication & User Management (AuthController)

POST /auth/login

Content-Type: application/json

{“email": "user@example.com", "password": "Password123"}

CV Processing (CVController)

POST /cv/uploadcv

Content-Type: multipart/form-data

--boundary
Content-Disposition: form-data; name="file"; filename="cv.pdf"
Content-Type: application/pdf
<PDF_BINARY_DATA>
--boundary--

User Profile (AuthController)

GET /auth/me?email=user@example.com

Authorization: Bearer <JWT_TOKEN>

Response Formats

Successful Login

```
{“message": "Login successful"}
```

CV Processing Result

```
{
  "status": "success",
  "processedData": {
    "skills": ["Java", "Spring Boot", "SQL"],
    "experience": "5+ years backend development",
    "education": "MSc Computer Science"
  }
}
```

User Profile

```
{
  "username": "johndoe",
  "email": "john@example.com",
  "first_name": "John",
  "last_name": "Doe",
  "role": "Recruiter"
}
```

Status / Error Codes

Code	Status	Description
200	OK	Successful operation
400	Bad Request	Invalid input / missing parameters
401	Unauthorized	Invalid credentials
403	Forbidden	Insufficient permissions
404	Not Found	Resource not found
413	Payload Too Large	File exceeds 5MB limit
415	Unsupported Media	Invalid file format (non-PDF/DOCX)
500	Internal Server Error	Server-side processing error
503	Service Unavailable	CV engine not available

Timeouts & Retry Policy

- **UI API Requests**: 30s timeout, 3 attempts (2s exponential backoff)
- **CV Processing**: 120s timeout, no retry (user must resubmit)
- **Database Queries**: 5s timeout, 2 attempts (1s backoff)

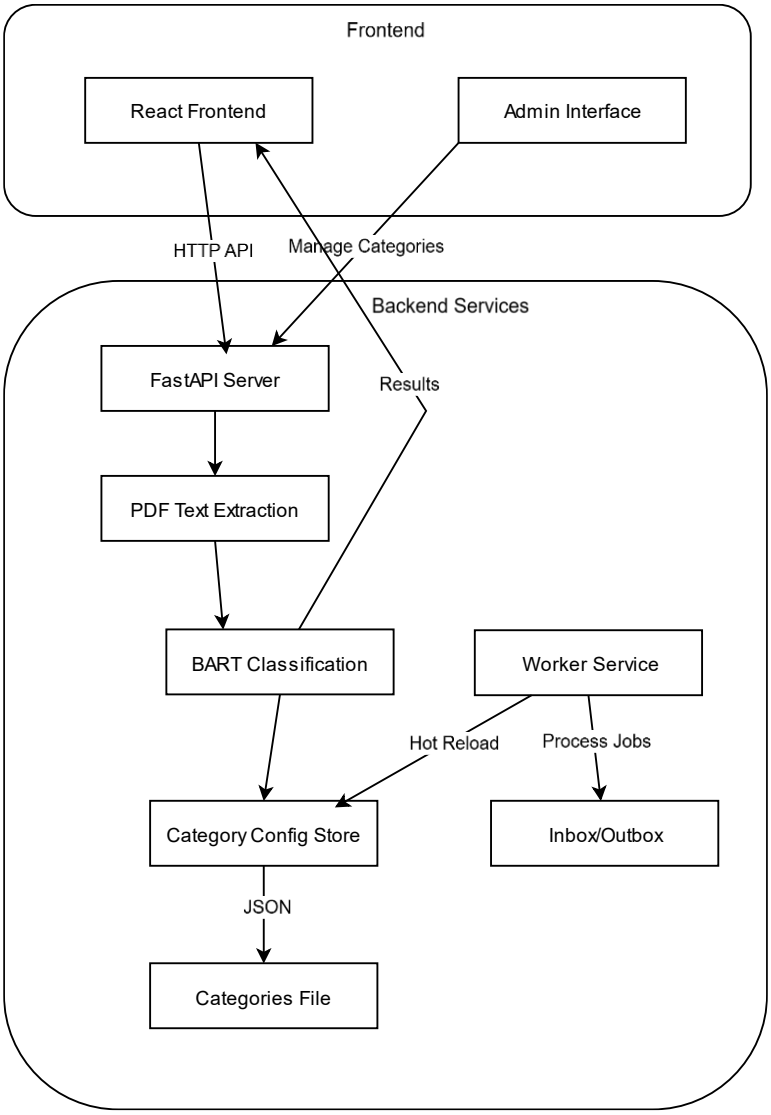
Authentication Flow

- JWT Bearer token recommended for secure endpoints.
- Role-based access control for users and admins.

Security Headers

- **Strict-Transport-Security**: max-age=31536000; includeSubDomains
- **Content-Security-Policy**: default-src 'self'
- **X-Content-Type-Options**: nosniff
- **X-Frame-Options**: DENY

4.2 UI To AI Service Contract



Communication Channels & Protocols

Component	Protocol	Port	Description
Frontend ↔ FastAPI	HTTPS/HTTP	5000	REST API for all user interactions
Worker ↔ Config Store	File I/O	-	JSON-based config file monitoring
Worker ↔ BART Model	In-process Python	-	Direct function calls
Frontend ↔ Auth Service	HTTPS	8081	User authentication/authorization

Request Formats

Upload CV Endpoint (POST /v1/upload_cv)

POST /v1/upload_cv?top_k=3 HTTP/1.1
Content-Type: multipart/form-data; boundary=boundary

--boundary
Content-Disposition: form-data; name="file"; filename="cv.pdf"
Content-Type: application/pdf

<PDF_BINARY_DATA>
--boundary--

Classify Text Endpoint (POST /v1/classify)

POST /v1/classify HTTP/1.1
Content-Type: application/json

```
{
  "text": "Candidate with 5+ years Python experience...",
  "top_k": 3
}
```

Category Management (POST /v1/admin/categories)

POST /v1/admin/categories HTTP/1.1
Content-Type: application/json

```
{
  "Programming Languages": ["Python", "Java", "Go"],
  "Cloud Tools": ["AWS", "Azure", "GCP"],
  "Roles": ["Backend Engineer", "Data Scientist"]
}
```

Response Formats

Successful CV Processing

HTTP/1.1 200 OK

Content-Type: application/json

```
{
  "status": "success",
  "top_k": 3,
  "applied": {
    "Skills": ["Python", "Django", "AWS"],
    "Roles": ["Backend Engineer"],
    "Education": ["MSc Computer Science"]
  },
  "raw": {
    "Skills": {
      "labels": ["Python", "Django", "AWS", "Java", "Go"],
      "scores": [0.92, 0.85, 0.78, 0.65, 0.42],
      "top_k": [
        {"label": "Python", "score": 0.92},
        {"label": "Django", "score": 0.85},
        {"label": "AWS", "score": 0.78}
      ]
    }
  }
}
```


Category Config Response

HTTP/1.1 200 OK
Content-Type: application/json


```
{
  "status": "saved",
  "categories": {
    "Programming Languages": ["Python", "Java", "Go"],
    "Cloud Tools": ["AWS", "Azure", "GCP"]
  }
}
```

Status / Error Codes

Code	Status	Description
200	OK	Successful operation
400	Bad Request	Invalid input/missing parameters
401	Unauthorized	Missing/invalid authentication token
403	Forbidden	Insufficient permissions
404	Not Found	Resource not found
409	Conflict	No categories configured
413	Payload Too Large	File exceeds 5MB limit
415	Unsupported Media	Invalid file format (non-PDF/DOC/DOCX)
422	Unprocessable Entity	Semantic errors in request
429	Too Many Requests	Rate limit exceeded
500	Internal Server Error	Server-side processing error
503	Service Unavailable	Worker queue full/overloaded

Timeouts & Retry Policy

Component	Timeout	Retry Policy
Frontend API Requests	30s	3 attempts with 2s exponential backoff
Worker Job Processing	120s	No retry (fail to DLQ)
Config Hot Reload	-	Continuous polling (2s interval)
Auth Service Calls	5s	2 attempts with 1s backoff

Versioning

API Versioning: URL path versioning (/v1/...)

Model Versioning:

```
# bart_model.py
MODEL_VERSION = "facebook/bart-large-mnli-v2.1"
```

Config Versioning:

```
{
  "_version": "2024.1",
  "categories": { ... }
}
```

Authentication & Authorization

Authentication Methods:

- JWT Bearer Tokens for admin endpoints
- API Keys for service-to-service communication

Role-Based Access Control:

- Admin: Full category/config access
- Recruiter: Read-only category + classify access
- System: Internal service accounts

Security Headers:

- Strict CORS policy
- HSTS enforcement
- Content-Security-Policy
- JWT in HttpOnly cookies

Key Contract Guarantees:

- Idempotency: All POST endpoints are idempotent
- Statelessness: No server-side session storage
- Forward Compatibility: Unknown fields ignored; new response fields are additive
- Backward Compatibility: v1 API maintained for 6 months after v2 release; deprecation notices via header

Error Response Example:

HTTP/1.1 409 Conflict
Content-Type: application/json


```
{
  "error": "ConfigurationConflict",
  "message": "No categories configured",
  "detail": "Please configure categories via /admin/categories first",
  "code": "CFG_001",
  "timestamp": "2024-06-15T14:30:00Z"
}
```


Monitoring Endpoints:

- GET /health: Service health check
- GET /metrics: Prometheus-formatted metrics
- GET /queue_depth: Worker queue status