server-side-documentation

Transcription Backend - Complete Server-Side Documentation

Table of Contents

1. System Overview
2. Architecture
3. Project Structure
4. Core Services
5. API Endpoints
6. Data Models
7. Configuration
8. Installation & Setup
9. Deployment
10. Testing
11. Monitoring & Logging
12. Troubleshooting
13. Development Guidelines

System Overview

The Transcription Backend is a comprehensive FastAPI-based service that provides three main functionalities:

1. **Video Transcription Service** - Transcribes videos from various platforms (YouTube, TikTok, X/Twitter, Instagram)

2. **AI Script Tailoring Service** - Transforms transcripts into marketing psychology-optimized scripts using GPT-4.1-mini
3. **Product Scraper Service** - Extracts product descriptions from e-commerce URLs

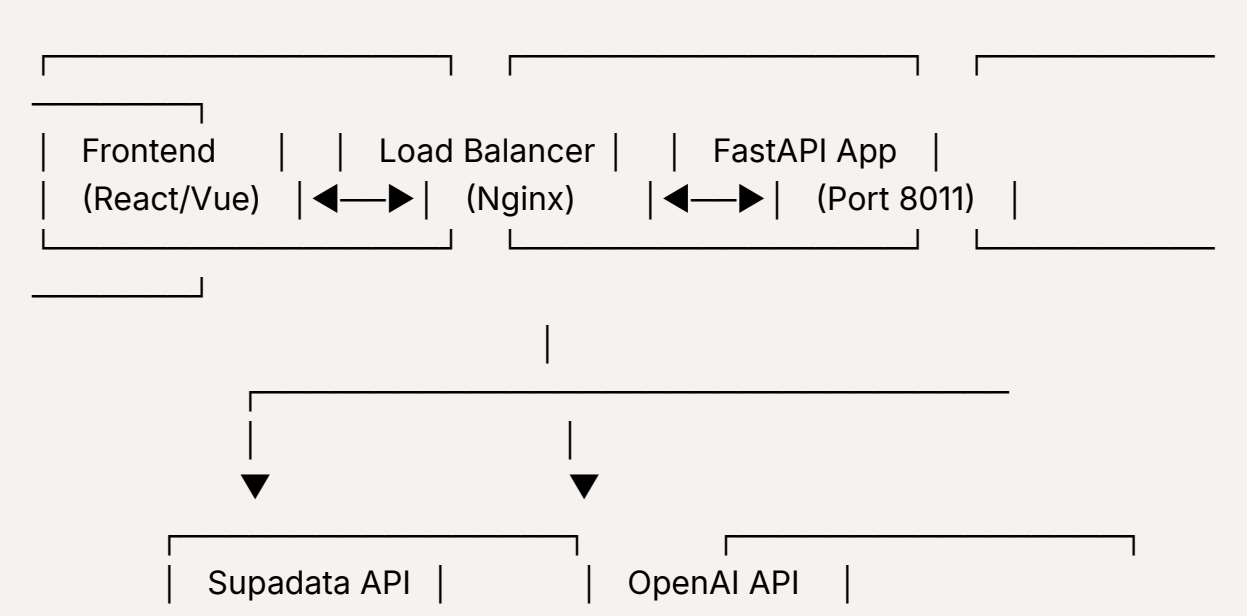
Key Features

- **Multi-Platform Transcription**: Supports YouTube, TikTok, X/Twitter, Instagram with platform-specific reliability ratings
- **AI-Powered Script Optimization**: Uses GPT-4.1-mini with marketing psychology principles (Kahneman, Cialdini, Sutherland, Hormozi)
- **Intelligent Web Scraping**: Multi-method product description extraction with fallback strategies
- **Robust Error Handling**: Comprehensive retry logic, JSON parsing recovery, and graceful degradation
- **Production Ready**: CORS support, health checks, caching, and comprehensive logging

---

Architecture

High-Level Architecture



```
        │  (Transcription) │          │  (GPT-4.1-mini)  │
        └──────────────────┘          └──────────────────┘
```

Service Architecture

```
FastAPI Application (main.py)
├── Transcription Service
│   ├── Supadata Integration
│   ├── Platform Detection
│   ├── Retry Logic
│   └── Caching System
├── AI Script Tailoring Service
│   ├── OpenAI Integration
│   ├── Psychology Prompt Engineering
│   ├── JSON Response Parsing
│   └── Error Recovery
└── Product Scraper Service
    ├── Multi-Selector Extraction
    ├── Structured Data Parsing
    ├── Meta Tag Fallback
    └── Content Cleaning
```

Project Structure

```
/root/transcription-backend/
├── main.py                 # FastAPI application entry point
├── models.py               # Pydantic data models
├── config.py               # Configuration settings
├── ai_service.py           # AI Script Tailoring service
├── scraper_service.py      # Product scraping service
├── requirements.txt        # Python dependencies
├── test_ai_endpoint.py     # AI service testing
├── test-model.py           # Model testing utilities
├── test.json               # Test data
```

```
├── app.log                # Application logs
├── nohup.out              # Process output logs
├── myenv/                 # Virtual environment
├── __pycache__/           # Python cache
│
├── Documentation/
├── FASTAPI_MIGRATION_GUIDE.md
├── FRONTEND_INTEGRATION_GUIDE.md
├── PRODUCT_SCRAPER_API.md
├── README_AI_SCRIPT.md
├── SIMPLE_INTEGRATION.md
└── server-side-documentation.md (this file)
```

Core Services

1. Transcription Service

File: `main.py` (lines 54-309)

Purpose: Transcribes videos from various social media platforms using Supadata API.

Key Features:
- Platform detection and reliability assessment
- Intelligent retry logic with exponential backoff
- In-memory caching with TTL
- Comprehensive error handling

Platform Support:
- **YouTube**: High reliability, fully supported
- **TikTok**: Low reliability, rate limiting issues
- **X/Twitter**: Low reliability, access restrictions
- **Instagram**: Medium reliability, some limitations
- **Vimeo/Twitch**: Not supported

Core Methods:

```
def _detect_platform(url: str) → dictasync def _transcribe_with_retry(url: str, la
ng: str, text: bool, mode: str, max_retries: int = 3) → dict
```

2. AI Script Tailoring Service

File: `ai_service.py`

Purpose: Transforms video transcripts into marketing psychology-optimized scripts using GPT-4.1-mini.

Key Features:
- GPT-4.1-mini integration with optimized parameters
- Marketing psychology prompt engineering
- Robust JSON parsing with error recovery
- Comprehensive response validation

Psychology Principles Applied:
- **Daniel Kahneman**: Loss aversion, cognitive biases
- **Robert Cialdini**: Authority, social proof, scarcity, reciprocity
- **Claude Hopkins**: Scientific advertising, urgency
- **Rory Sutherland**: Value reframing, identity psychology
- **Alex Hormozi**: Value stacking, grand slam offers

Core Methods:

```
async def generate_tailored_script(request: AITailoringRequest) → AITailoring
Data
async def call_openai_api(system_prompt: str, user_prompt: str) → strdef pars
e_ai_response(ai_response: str) → Dict[str, Any]
```

3. Product Scraper Service

File: `scraper_service.py`

Purpose: Extracts product descriptions from e-commerce URLs using multiple extraction methods.

Key Features:
- Multi-layered extraction strategy

- CSS selector-based extraction
- JSON-LD structured data parsing
- Meta tag fallback
- Content cleaning and normalization

Extraction Methods:
1. **CSS Selectors**: Product-specific selectors for major e-commerce platforms
2. **Structured Data**: JSON-LD schema.org markup
3. **Meta Tags**: Meta description as fallback
4. **Content Cleaning**: Text normalization and unwanted pattern removal

Core Methods:

```
async def scrape_product(request: ProductScraperRequest) → ProductScrap
erData
def _extract_text_by_selectors(soup: BeautifulSoup, selectors: list) → Optional
[str]
def _extract_structured_data_description(soup: BeautifulSoup) → Optional[str]
```

API Endpoints

Base URL

```
http://localhost:8011
```

1. Transcription Endpoint

POST `/transcribe`

Transcribes videos from supported platforms.

Request Body:

```
{  "urls": ["https://youtube.com/watch?v=example"],  "lang": "en",  "text": true,
"mode": "auto"}
```

Response:

{ "results": [{ "url": "https://youtube.com/watch?v=example", "platform": "YouTube", "language": "en", "transcript": "Transcribed content...", "status": "success" }], "summary": { "total": 1, "successful": 1, "failed": 0 }}

2. AI Script Tailoring Endpoint

POST `/api/ai-tailor-script`

Transforms transcripts into marketing-optimized scripts.

Request Body:

{ "originalTranscript": "Your video transcript here...", "productDescription": "Product description for optimization..."}

Response:

{ "success": true, "data": { "tailoredScript": "Optimized script content...", "confidence": 0.95, "processingTime": 3.2, "wordCount": 156, "estimatedReadTime": "45s", "sectionBreakdown": [{ "sectionName": "Hook", "triggerEmotionalState": "Curiosity + Authority", "originalQuote": "Original text...", "rewrittenVersion": "Rewritten text...", "sceneDescription": "Filming instructions...", "psychologicalPrinciples": ["Loss Aversion", "Authority"], "timestamp": "00:00:01 → 00:00:03" }], "sutherlandAlchemy": { "explanation": "Value reframing explanation...", "valueReframing": [...], "identityShifts": [...] }, "hormoziValueStack": { "coreOffer": "Main value proposition...", "valueElements": [...], "totalStack": {...}, "grandSlamElements": [...] } }, "metadata": { "originalLength": 89, "improvementAreas": ["product_alignment"], "apiVersion": "2.0", "timestamp": "2024-01-15T10:30:00Z", "model_used": "gpt-4.1-mini" }}

3. Product Scraper Endpoint

POST `/api/scrape-product`

Extracts product descriptions from URLs.

Request Body:

{ "url": "https://example-store.com/product/123"}

Response:

{ "success": true, "data": { "description": "Extracted product description...", "title": "Product Title (optional)" }, "metadata": { "url": "https://example-store.com/product/123", "domain": "example-store.com", "timestamp": "2024-01-15T10:30:00Z", "apiVersion": "1.0" }}

4. Platform Support Check

GET `/platform-support?url={video_url}`

Checks platform support and reliability.

Response:

{ "url": "https://youtube.com/watch?v=example", "platform_info": { "platform": "YouTube", "supported": true, "reliability": "high", "recommendation": "Fully supported and reliable" }, "timestamp": "2024-01-15T10:30:00Z"}

5. Health Check

GET `/health`

Service health and configuration status.

Response:

{ "status": "healthy", "service": "Transcription & AI Script Tailoring API", "version": "2.1", "timestamp": "2024-01-15T10:30:00Z", "openai_model": "gpt-4.1-mini", "services": ["transcription", "ai_script_tailoring", "platform_detection", "product_scraping"], "supported_platforms": { "high_reliability": ["YouTube"], "medium_reliability": ["Instagram"], "low_reliability": ["TikTok", "X (Twitter)"], "not_supported": ["Vimeo", "Twitch"] }}

Data Models

Request Models

AITailoringRequest:

```
class AITailoringRequest(BaseModel):
    originalTranscript: str = Field(..., description="Required transcript text")
    productDescription: str = Field(..., description="Required product descriptio
n")
```

ProductScraperRequest:

```
class ProductScraperRequest(BaseModel):
    url: str = Field(..., description="Product URL to scrape")
```

UrlList (Transcription):

```
class UrlList(BaseModel):
    urls: List[str]
    lang: str = "en"    text: bool = True    mode: str = "auto"
```

Response Models

AITailoringResponse:

```
class AITailoringResponse(BaseModel):
    success: bool    data: AITailoringData
    metadata: Dict[str, Any]
class AITailoringData(BaseModel):
    tailoredScript: str    confidence: float    processingTime: float    wordCount: i
nt    estimatedReadTime: str    sectionBreakdown: List[SectionBreakdown]
    sutherlandAlchemy: SutherlandAlchemy
    hormoziValueStack: HormoziValueStack
```

ProductScraperResponse:

```
class ProductScraperResponse(BaseModel):
    success: bool    data: ProductScraperData
    metadata: Dict[str, Any]
class ProductScraperData(BaseModel):
    description: str    title: Optional[str] = None
```

Complex Models

SectionBreakdown:

```
class SectionBreakdown(BaseModel):
    sectionName: str    triggerEmotionalState: str    originalQuote: str    rewritten
Version: str    sceneDescription: str    psychologicalPrinciples: List[str]
    timestamp: str
```

SutherlandAlchemy:

```
class SutherlandAlchemy(BaseModel):
    explanation: str    valueReframing: List[Dict[str, Any]]
    identityShifts: List[str]
```

HormoziValueStack:

```
class HormoziValueStack(BaseModel):
    coreOffer: str    valueElements: List[Dict[str, Any]]
    totalStack: Dict[str, Any]
    grandSlamElements: List[str]
```

Configuration

Environment Variables

File: `config.py`

```
class Settings:
    # OpenAI Configuration    OPENAI_API_KEY: str = "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"    OPENAI_MODEL: str = "gpt-4.1-mini"    # Performance Settings    MAX_TOKENS: int = 4000    TEMPERATURE: float = 0.2    REQUEST_TIMEOUT: int = 60
```

Supadata Configuration

API Key: `sd_efd97aab44fb38592803c0a90b75133b`

Supported Parameters:
- `lang`: Language code (default: “en”)
- `text`: Return text format (default: true)
- `mode`: Transcription mode (default: “auto”)

CORS Configuration

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000", "https://staging.myrefera.com", "*", "https://abdul-rafay.myrefera.com"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Caching Configuration

```
# In-memory cache with TTLCACHE = {}
CACHE_TTL = 86400  # 24 hours in seconds
```

---

nstallation & Setup

Prerequisites

- Python 3.8+
- pip package manager
- Virtual environment (recommended)

Step-by-Step Installation

1. **Clone/Download the Project**:

   ```
   cd /root/transcription-backend
   ```

2. **Create Virtual Environment**:

   ```
   python -m venv myenv
   source myenv/bin/activate  # Linux/Mac# ormyenv\Scripts\activate     # Wi
   ndows
   ```

3. **Install Dependencies**:

   ```
   pip install -r requirements.txt
   ```

4. **Verify Installation**:

   ```
   python -c "import fastapi, openai, supadata; print('All dependencies install
   ed successfully')"
   ```

Dependencies

requirements.txt:

```
fastapi==0.104.1
uvicorn==0.24.0
supadata
pydantic==2.5.0
openai>=1.40.0
```

```
python-dotenv==1.0.0
requests>=2.25.0
beautifulsoup4>=4.9.0
```

Development Setup

1. **Start Development Server**:

   ```
   uvicorn main:app --reload --host 0.0.0.0 --port 8011
   ```

2. **Test the API**:

   ```
   curl http://localhost:8011/health
   ```

3. **Run Tests**:

   ```
   python test_ai_endpoint.py
   ```

Deployment

Production Deployment

1. **Environment Setup**:

   ```
   # Set production environment variablesexport OPENAI_API_KEY="your-pr
   oduction-key"export OPENAI_MODEL="gpt-4.1-mini"
   ```

2. **Start Production Server**:

   ```
   uvicorn main:app --host 0.0.0.0 --port 8011 --workers 4
   ```

3. **Background Process**:

   ```
   nohup uvicorn main:app --host 0.0.0.0 --port 8011 --workers 4 > app.log
   ```

```
2>&1 &
```

Docker Deployment (Optional)

Dockerfile:

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 8011
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8011"]
```

Build and Run:

```
docker build -t transcription-backend .
docker run -p 8011:8011 transcription-backend
```

Nginx Configuration

```
server {
    listen 80;
    server_name your-domain.com;

    location / {
        proxy_pass http://localhost:8011;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Process Management

Using systemd (create `/etc/systemd/system/transcription-backend.service`):

```
[Unit]Description=Transcription Backend APIAfter=network.target[Service]Type=execUser=www-dataWorkingDirectory=/root/transcription-backendEnvironment=PATH=/root/transcription-backend/myenv/binExecStart=/root/transcription-backend/myenv/bin/uvicorn main:app --host 0.0.0.0 --port 8011Restart=always[Install]WantedBy=multi-user.target
```

Enable and Start:

```
sudo systemctl enable transcription-backend
sudo systemctl start transcription-backend
sudo systemctl status transcription-backend
```

Testing

Unit Tests

Test AI Endpoint (`test_ai_endpoint.py`):

```
import requests
import json
def test_ai_endpoint():
    url = "http://localhost:8011/api/ai-tailor-script"    payload = {
        "originalTranscript": "Test transcript content...",
        "productDescription": "Test product description..."    }
    response = requests.post(url, json=payload)
    assert response.status_code == 200    data = response.json()
    assert data["success"] == True    assert "tailoredScript" in data["data"]
    print("✅ AI endpoint test passed")
if __name__ == "__main__":
    test_ai_endpoint()
```

Integration Tests

Test All Endpoints:

```
import requests
def test_health_endpoint():
    response = requests.get("http://localhost:8011/health")
    assert response.status_code == 200    data = response.json()
    assert data["status"] == "healthy"    print("✅ Health check passed")
def test_transcription_endpoint():
    payload = {
        "urls": ["https://youtube.com/watch?v=dQw4w9WgXcQ"],
        "lang": "en",
        "text": True,
        "mode": "auto"    }
    response = requests.post("http://localhost:8011/transcribe", json=payload)
    assert response.status_code == 200    print("✅ Transcription endpoint pas
sed")
def test_scraper_endpoint():
    payload = {"url": "https://example.com"}
    response = requests.post("http://localhost:8011/api/scrape-product", json=
payload)
    assert response.status_code == 200    print("✅ Scraper endpoint passed")
```

Load Testing

Using Apache Bench:

```
# Test AI endpoint with 100 requests, 10 concurrentab -n 100 -c 10 -p test_pay
load.json -T application/json http://localhost:8011/api/ai-tailor-script
```

test_payload.json:

```
{  "originalTranscript": "Test transcript for load testing...",  "productDescriptio
n": "Test product for load testing..."}
```

Monitoring & Logging

Logging Configuration

Application Logs:

```
import logging
# Configure logginglogging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
# Log exampleslogger.info(f"🎯 Starting transcription for {len(data.urls)} URL
s")
logger.info(f"✅ Transcription successful for {platform_info['platform']} URL:
{url}")
logger.error(f"❌ Error scraping product description: {e}")
```

Log Files

- **app.log**: Application logs
- **nohup.out**: Process output when running in background

Health Monitoring

Health Check Endpoint:

```
curl http://localhost:8011/health
```

Response Monitoring:

```
{  "status": "healthy",  "service": "Transcription & AI Script Tailoring API",  "ver
sion": "2.1",  "timestamp": "2024-01-15T10:30:00Z",  "openai_model": "gpt-4.1
-mini",  "services": ["transcription", "ai_script_tailoring", "platform_detection",
"product_scraping"]}
```

Performance Metrics

Key Metrics to Monitor:
- Response times for each endpoint

- Success/failure rates
- OpenAI API usage and costs
- Supadata API usage
- Memory usage
- CPU utilization

Log Analysis:

```
# Check for errorsgrep "❌" app.log
# Check response timesgrep "processingTime" app.log
# Check API usagegrep "OpenAI API" app.log
```

Troubleshooting

Common Issues

1. OpenAI API Errors

Error: `OpenAI API Error: Invalid API key`

Solution:

```
# Check API key in config.pypython -c "from config import settings; print(setti
ngs.OPENAI_API_KEY[:20] + '...')"# Test API keypython -c "from openai import
OpenAI; client = OpenAI(api_key='your-key'); print('API key valid')"
```

2. Supadata Connection Issues

Error: `SupadataError: Connection failed`

Solution:

```
# Check Supadata API keypython -c "from supadata import Supadata; supada
ta = Supadata(api_key='sd_efd97aab44fb38592803c0a90b75133b'); print('Su
padata connected')"# Test with a simple URLpython -c "from supadata import
Supadatasupadata = Supadata(api_key='sd_efd97aab44fb38592803c0a90b7
5133b')result = supadata.transcript(url='https://youtube.com/watch?v=dQw4
```

```
w9WgXcQ', lang='en', text=True, mode='auto')print('Supadata test successfu
l')"
```

3. Port Already in Use

Error: `Address already in use`

Solution:

```
# Find process using port 8011lsof -i :8011
# Kill the processskill -9 <PID># Or use a different portuvicorn main:app --host
0.0.0.0 --port 8012
```

4. JSON Parsing Errors

Error: `JSONDecodeError: Expecting value`

Solution:
- The AI service includes automatic JSON recovery
- Check logs for detailed error information
- Verify OpenAI model availability

5. CORS Issues

Error: `CORS policy: No 'Access-Control-Allow-Origin' header`

Solution:

```
# Update CORS configuration in main.pyapp.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Allow all origins for development    allow_credentials
=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Debug Mode

Enable Debug Logging:

```
import logging
logging.getLogger().setLevel(logging.DEBUG)
```

Debug AI Service:

```
# Add to ai_service.pylogger.debug(f"Raw OpenAI Response: {ai_response}")
logger.debug(f"Parsed Data: {parsed_data}")
```

Performance Issues

Slow Response Times

Check:
1. OpenAI API response times
2. Network connectivity
3. Server resources (CPU, memory)
4. Concurrent request handling

Optimize:

```
# Reduce max_tokens for faster responsesMAX_TOKENS = 2000  # Instead of 4000# Lower temperature for consistencyTEMPERATURE = 0.1  # Instead of 0.2
```

Memory Issues

Monitor:

```
# Check memory usageps aux | grep uvicorn
# Monitor with htophtop
```

Optimize:

```
# Implement request limits# Add response compression# Use connection pool ing
```

Development Guidelines

Code Structure

File Organization:
- `main.py`: FastAPI app, routing, and transcription service
- `ai_service.py`: AI script tailoring logic
- `scraper_service.py`: Product scraping logic
- `models.py`: Pydantic data models
- `config.py`: Configuration settings

Coding Standards

Python Style:
- Follow PEP 8 guidelines
- Use type hints for all functions
- Document complex functions with docstrings
- Use meaningful variable names

Example:

```
async def _transcribe_with_retry(
    url: str,
    lang: str,
    text: bool,
    mode: str,
    max_retries: int = 3) → dict:
    """    Transcribe URL with retry logic and proper error handling.    Args:
url: Video URL to transcribe        lang: Language code        text: Whether to ret
urn text        mode: Transcription mode        max_retries: Maximum number of
retry attempts    Returns:        dict: Transcription result or error information
"""
```

Error Handling

Consistent Error Format:

```
try:
    # Operation    result = await some_operation()
    return result
except SpecificException as e:
    logger.error(f"❌ Specific error: {e}")
    raise HTTPException(
        status_code=500,
        detail={
            "success": False,
            "error": {
                "code": "SPECIFIC_ERROR",
                "message": f"Operation failed: {str(e)}"            }
        }
    )
```

Testing Guidelines

Test Structure:

```
def test_function_name():
    """Test description."""    # Arrange    input_data = "test input"    expected_o
utput = "expected result"    # Act    result = function_under_test(input_data)
    # Assert    assert result == expected_output
    print("✅ Test passed")
```

API Documentation

Endpoint Documentation:

```
@app.post("/api/endpoint", response_model=ResponseModel)
async def endpoint_function(request: RequestModel):
    """    Brief description of what this endpoint does.    Args:        request: Req
uest model with required fields    Returns:        ResponseModel: Structured res
ponse with data and metadata    Raises:        HTTPException: 500 if processin
g fails    """
```