Media Processor Project Architecture

🏷️ Project Overview

A **distributed media processing system** that handles image uploads, applies watermarks, and stores processed files in a centralized location. The system uses Redis-based task queuing (ARQ) for asynchronous job processing and Docker containerization for scalability.

🏗️ System Components

1. **Fast Server** (Port 8000)

Type: API Gateway / File Upload Handler
Framework: FastAPI + Uvicorn
Location: `fast-server/main.py`

Responsibilities:

- Accepts file uploads from clients via POST `/media` endpoint
- Saves uploaded files to shared `uploads/` directory
- Enqueues watermarking tasks to Redis queue
- Manages Redis connection lifecycle (lifespan)

Key Features:

- Async file handling
- Redis connection pooling
- Environment-based configuration (Redis host via `REDIS_HOST`)

Dependencies:

- FastAPI, Uvicorn (web server)
- ARQ (task queue client)
- Redis (connection management)

---

2. **Worker Server** (Background Process)

Type: Async Task Processor
Language: Python (runs as worker via ARQ)
Location: `worker-server/worker.py`

Two-Stage Processing Pipeline:

Stage 1: Image Watermarking (`watermark_image_task`)

- Downloads logo (Google logo by default from URL)
- Opens original image file

- Resizes logo to 20% of image dimensions
- Pastes logo in top-right corner (with 10px padding)
- Converts to RGBA for transparency support
- Saves watermarked image as `watermarked_[original_filename]`
- Enqueues Stage 2 automatically
- **Processing Time:** 3-8 seconds (simulated latency)

Stage 2: Central Upload (`upload_to_central_task`)

- Sends watermarked image to Central Server (Port 8003)
- Includes worker hostname in metadata
- **Retry Logic:** Up to 5 retries with exponential backoff (10s, 20s, 30s...)
- **Timeout:** 15 seconds per upload attempt

Key Features:

- CPU-intensive image processing using Pillow
- Network-intensive file shipping with resilience
- Automatic pipeline orchestration via Redis
- Worker identification via `HOSTNAME` environment variable

Dependencies:

- Pillow (image processing)
- ARQ, Redis (task queue)
- httpx (async HTTP client)

---

3. **Central Server** (Port 8003)

Type: Final Storage & Logging Service
Framework: FastAPI + Uvicorn
Location: `central-server/main.py`

Responsibilities:

- Receives watermarked images from workers
- Stores files in `central_storage/` directory
- Logs all uploads to `worker_activity.log`
- Tracks worker metadata and timestamps

Endpoint: POST `/upload-final`

- Accepts multipart file upload
- Accepts worker_name in form data
- Records timestamp of storage

Output:

```
```

```
[2026-01-15 14:30:45] WORKER: worker-1 | IMAGE: watermarked_photo.jpg
[2026-01-15 14:30:52] WORKER: worker-2 | IMAGE: watermarked_landscape.jpg
```

---

4. **Redis** (Backend Message Broker)

Type: Task Queue & Message Broker
Default Host: `localhost` (configurable via `REDIS_HOST`)

Queue Topics:

- `watermark_image_task` - Stage 1 watermarking jobs
- `upload_to_central_task` - Stage 2 central upload jobs

🔄 Request Flow Diagram

```
┌─────────────────┐
│   CLIENT        │
│  (File Upload)  │
└────────┬────────┘
         │ POST /media
         ▼
┌─────────────────────────────┐
│   FAST SERVER (Port 8000)   │
│  - Save to uploads/         │
│  - Enqueue Stage 1 Job      │
└────────┬────────────────────┘
         │
         ▼
┌──────────────────────────┐
│  REDIS QUEUE (ARQ)       │
│  watermark_image_task    │
└────────┬─────────────────┘
         │
         ▼
┌───────────────────────────────────┐
│  WORKER SERVER                    │
│  STAGE 1: watermark_image_task    │
│  - Download logo                  │
│  - Process image                  │
│  - Save watermarked file          │
│  - Enqueue Stage 2                │
└────────┬──────────────────────────┘
         │
         ▼
┌──────────────────────────┐
│  REDIS QUEUE (ARQ)       │
│  upload_to_central_task  │
└────────┬─────────────────┘
```

```
         │
         ▼
┌──────────────────────────────────┐
│  WORKER SERVER                   │
│  STAGE 2: upload_to_central_task │
│  - Send to Central Server (8003) │
│  - Retry logic if failure        │
└────────┬─────────────────────────┘
         │
         ▼
┌──────────────────────────────┐
│  CENTRAL SERVER (Port 8003)  │
│  - Store in central_storage/ │
│  - Log worker activity       │
└──────────────────────────────┘
```

📁 Directory Structure & File Storage

```
media-processor/
├── fast-server/
│   ├── main.py
│   ├── Dockerfile
│   ├── requirements.txt
│   └── uploads/               ← Stage 1: Raw uploads
│
├── worker-server/
│   ├── worker.py
│   ├── Dockerfile
│   ├── requirements.txt
│   └── (processed files)      ← Stage 1 → Stage 2 transition
│
├── central-server/
│   ├── main.py
│   ├── central_storage/       ← Stage 2: Final storage
│   └── worker_activity.log    ← Activity logging
│
└── testfiles/
    └── test.py
```

🔧 Configuration & Environment Variables

Variable	Default	Purpose
`REDIS_HOST`	`localhost`	Redis server hostname/IP
`HOSTNAME`	System hostname	Worker identifier

Variable	Default	Purpose
`GOOGLE_LOGO`	Google logo URL	Watermark logo URL
`CENTRAL_SERVER_URL`	`http://host.docker.internal:8003/upload-final`	Central server endpoint

🐳 Docker Deployment

Fast Server Dockerfile:

- Builds FastAPI web server
- Exposes port 8000
- Connects to Redis queue

Worker Server Dockerfile:

- Builds ARQ worker process
- Processes queue jobs asynchronously
- Connects to Redis and Central Server

Central Server:

- Runs on port 8003
- No containerization mentioned yet

⚙ Key Design Patterns

1. **Two-Stage Pipeline**
   - Decouples image processing from file shipping
   - Allows independent retry logic per stage
   - Enables horizontal scaling of workers
2. **Async Task Queue (ARQ)**
   - Non-blocking request handling
   - Distributed job processing
   - Automatic retry with exponential backoff
3. **Shared Volume Architecture**
   - `uploads/` directory: Intermediate file exchange
   - `central_storage/`: Final destination
   - Reduces network overhead within pipeline
4. **Worker Identification**
   - Hostname-based worker tracking
   - Audit trail in activity logs
   - Load distribution visibility

📊 Performance Characteristics

Component	Latency	Throughput	Bottleneck
Fast Server	~100ms	Limited by disk I/O	File save speed
Worker Stage 1	3-8s	CPU-bound	Image processing
Worker Stage 2	1-15s	Network-bound	Central server response
Central Server	~100ms	Limited by disk I/O	File save + logging

🔍 Monitoring & Logging

- **Worker Activity Log:** `worker_activity.log` tracks all completed uploads
- **Console Logs:** Emoji-enhanced logs for each stage
- **Redis Monitoring:** Queue depth visible via Redis client
- **Docker Logs:** Container output for debugging

🚀 Scalability Considerations

✅ **Scalable:**

- Multiple worker instances (horizontal scaling)
- Redis handles distributed queue
- Stateless fast/central servers

⚠ **Potential Bottlenecks:**

- Single Redis instance (recommend Redis cluster)
- Shared volume storage (recommend distributed storage like S3)
- Central server disk I/O

🧪 Testing

Test Files Location: `testfiles/test.py`

- Upload test images
- Validate watermarking
- Verify central storage