AI Authentication Implementation Guide

Overview

This guide provides comprehensive instructions for integrating real AI-powered authentication into the Genesis Provenance platform. The current implementation uses a **mock AI analysis engine** (`/lib/ai-mock.ts`) that simulates AI authentication results for demonstration and testing purposes.

Table of Contents

Current Implementation

Mock AI Analysis Engine

Location: `/lib/ai-mock.ts`

Key Features:
- Simulates 1.5-3 second processing time
- Generates realistic confidence scores (70-98%)
- Provides category-specific counterfeit indicators
- Creates authenticity markers based on item type
- Calculates fraud risk levels (low, medium, high, critical)

What It Does:
- ✅ Provides immediate, working UI for demo purposes
- ✅ Shows realistic analysis results
- ✅ Generates provenance events
- ✅ Creates educational content
- ❌ Does NOT analyze actual images
- ❌ Does NOT use real computer vision
- ❌ Does NOT perform fraud detection

Database Schema

The `AIAnalysis` model is production-ready:

```
model AIAnalysis {
  id                    String            @id @default(uuid())
  itemId                String
  requestedByUserId     String
  status                AIAnalysisStatus  @default(pending)
  confidenceScore       Int?              // 0-100
  fraudRiskLevel        FraudRiskLevel?
  findings              Json?
  counterfeitIndicators Json?
  authenticityMarkers   Json?
  analyzedImageIds      String[]
  processingTime        Int?              // milliseconds
  apiProvider           String?           // "google-vision", "aws-rekognition", "mock
"
  errorMessage          String?
  requestedAt           DateTime          @default(now())
  completedAt           DateTime?
  updatedAt             DateTime          @updatedAt
  // Relations
  item                  Item              @relation(...)
  requestedBy           User              @relation(...)
}
```

API Routes

1. **POST /api/items/[id]/ai-analysis** - Request new analysis
2. **GET /api/items/[id]/ai-analysis** - Fetch analyses for an item
3. **GET /api/admin/ai-analyses** - Admin dashboard (all analyses)

UI Components

- **AIAnalysisSection** (`/components/dashboard/ai-analysis-section.tsx`)
- **AI Authentication Tab** on item detail page
- **Admin AI Analysis Management** (`/app/(dashboard)/admin/ai-analyses/page.tsx`)

Architecture Overview

Current Flow (Mock)

```
User → Request Analysis → API Route → Create DB Record → Process Mock Analysis →
Update DB → Display Results
```

Proposed Flow (Production)

```
User → Request Analysis → API Route → Create DB Record → Queue Job →
  → Background Worker → Fetch Images from S3 → Send to CV API →
  → Process Results → Update DB → Notify User → Display Results
```

Key Differences

Aspect	Mock	Production
Processing	Synchronous	Asynchronous (queue)
Time	1-3 seconds	10-60 seconds
Cost	Free	$1.50-$5.00 per analysis
Accuracy	Simulated	Real AI
Images	Not analyzed	Actually analyzed
Scalability	Limited	Horizontal

Integration Options

Option 1: Google Cloud Vision AI (Recommended)

Pros:
- ✅ Excellent object detection and OCR
- ✅ Robust text recognition for serial numbers, date codes
- ✅ Material and texture analysis
- ✅ Extensive API documentation
- ✅ Free tier: 1,000 units/month

Cons:
- ❌ $1.50 per 1,000 units after free tier
- ❌ Requires GCP setup

Best For:
- Watches (dial text, serial numbers)
- Jewelry (hallmarks, stamps)
- Handbags (date codes, hardware stamps)
- Documents (certificates, receipts)

Option 2: AWS Rekognition

Pros:
- ✅ Excellent for luxury cars (object/scene detection)
- ✅ Integration with existing AWS infrastructure (S3)
- ✅ Custom labels for brand-specific features
- ✅ Free tier: 5,000 images/month (12 months)

Cons:
- ❌ $1-$5 per 1,000 images depending on feature
- ❌ Less robust OCR than Google Vision

Best For:
- Luxury cars (body panels, interior, VIN plates)

- Fine art (style analysis, forgery detection)
- General object detection

Option 3: Hybrid Approach

Recommended Strategy:
- Use **Google Cloud Vision** for text-heavy analysis (watches, handbags, jewelry)
- Use **AWS Rekognition** for image-heavy analysis (cars, art, collectibles)
- Route based on `item.category.slug`

Google Cloud Vision AI Integration

Step 1: Setup GCP Project

```
# Install Google Cloud SDK
curl https://sdk.cloud.google.com | bash
exec -l $SHELL

# Initialize and authenticate
gcloud init
gcloud auth application-default login

# Create new project (or use existing)
gcloud projects create genesis-provenance-ai
gcloud config set project genesis-provenance-ai

# Enable Vision API
gcloud services enable vision.googleapis.com

# Create service account
gcloud iam service-accounts create ai-vision-service \
  --display-name="AI Vision Service Account"

# Grant permissions
gcloud projects add-iam-policy-binding genesis-provenance-ai \
  --member="serviceAccount:ai-vision-service@genesis-provenance-
ai.iam.gserviceaccount.com" \
  --role="roles/cloudvision.user"

# Create and download key
gcloud iam service-accounts keys create ./gcp-vision-key.json \
  --iam-account=ai-vision-service@genesis-provenance-ai.iam.gserviceaccount.com
```

Step 2: Install Dependencies

```
cd /home/ubuntu/genesis_provenance/nextjs_space
yarn add @google-cloud/vision
```

Step 3: Configure Environment Variables

Add to `.env`:

```
# Google Cloud Vision AI
GOOGLE_CLOUD_PROJECT_ID=genesis-provenance-ai
GOOGLE_APPLICATION_CREDENTIALS=./gcp-vision-key.json
GOOGLE_VISION_API_KEY=your_api_key_here
```

Step 4: Create Google Vision Utility

Create `/lib/ai-google-vision.ts`:

```
import vision from '@google-cloud/vision';
import { FraudRiskLevel } from '@prisma/client';
import { downloadFile } from './s3';

const client = new vision.ImageAnnotatorClient();

interface VisionAnalysisResult {
  confidenceScore: number;
  fraudRiskLevel: FraudRiskLevel;
  findings: {
    summary: string;
    overallAssessment: string;
    keyObservations: string[];
  };
  counterfeitIndicators: {
    found: boolean;
    items: Array<{
      indicator: string;
      severity: 'low' | 'medium' | 'high';
      description: string;
    }>;
  };
  authenticityMarkers: {
    found: boolean;
    items: Array<{
      marker: string;
      confidence: 'low' | 'medium' | 'high';
      description: string;
    }>;
  };
  processingTime: number;
}

export async function analyzeWithGoogleVision(
  item: any,
  imageS3Keys: string[]
): Promise<VisionAnalysisResult> {
  const startTime = Date.now();
  const indicators: any[] = [];
  const markers: any[] = [];
  const observations: string[] = [];

  try {
    // Download images from S3
    const imageBuffers = await Promise.all(
      imageS3Keys.map(async key => {
        const url = await downloadFile(key);
        const response = await fetch(url);
        return Buffer.from(await response.arrayBuffer());
      })
    );

    // Analyze each image
    for (const buffer of imageBuffers) {
      // 1. Label Detection (objects, materials)
      const [labelResult] = await client.labelDetection(buffer);
      const labels = labelResult.labelAnnotations || [];

      // Check for luxury materials
      const luxuryMaterials = ['leather', 'gold', 'silver', 'diamond', 'precious'];
      const detectedMaterials = labels
        .filter(l => luxuryMaterials.some(m => l.description?.toLower-
Case().includes(m)))
        .map(l => l.description);

      if (detectedMaterials.length > 0) {
        markers.push({
          marker: 'Material Detection',
          confidence: 'high',
          description: `Detected luxury materials: ${detectedMaterials.join(', ')}`
        });
      }

      // 2. Text Detection (serial numbers, stamps, hallmarks)
      const [textResult] = await client.textDetection(buffer);
      const text = textResult.fullTextAnnotation?.text || '';

      if (text.trim()) {
        observations.push(`Text detected: ${text.substring(0, 100)}...`);

        // Check for serial number format
        const serialMatch = text.match(/\b[A-Z0-9]{6,}\b/);
        if (serialMatch) {
          markers.push({
            marker: 'Serial Number Detected',
            confidence: 'medium',
            description: `Serial/reference number found: ${serialMatch[0]}`
          });
        }
      } else {
        indicators.push({
          indicator: 'Missing Text Elements',
          severity: 'medium',
          description: 'Expected text (serial numbers, stamps) not clearly visible'
        });
      }

      // 3. Logo Detection
      const [logoResult] = await client.logoDetection(buffer);
      const logos = logoResult.logoAnnotations || [];

      if (logos.length > 0) {
        markers.push({
          marker: 'Brand Logo Detected',
          confidence: 'high',
          description: `Detected logo: ${logos[0].description}`
        });
      }

      // 4. Image Properties (color, quality)
      const [propsResult] = await client.imageProperties(buffer);
      const colors = propsResult.imagePropertiesAnnotation?.dominantColors?.colors ||
[];

      // Check image quality
      const avgPixelScore = colors.reduce((sum, c) => sum + (c.score || 0), 0) / col-
ors.length;
      if (avgPixelScore < 0.5) {
        indicators.push({
          indicator: 'Image Quality Concern',
          severity: 'low',
          description: 'Image quality may be insufficient for detailed analysis'
        });
      }
    }
    // Calculate confidence score based on markers and indicators
    const markerScore = Math.min(markers.length * 15, 60);
    const indicatorPenalty = indicators.length * 10;
    const confidenceScore = Math.max(50, Math.min(98, 70 + markerScore - indicatorPen-
alty));

    // Determine fraud risk
    let fraudRiskLevel: FraudRiskLevel;
    if (indicators.length === 0 && markers.length >= 3) {
      fraudRiskLevel = FraudRiskLevel.low;
    } else if (indicators.some(i => i.severity === 'high')) {
      fraudRiskLevel = FraudRiskLevel.critical;
    } else if (indicators.length >= 3) {
      fraudRiskLevel = FraudRiskLevel.high;
    } else if (indicators.length >= 1) {
      fraudRiskLevel = FraudRiskLevel.medium;
    } else {
      fraudRiskLevel = FraudRiskLevel.low;
    }

    const processingTime = Date.now() - startTime;

    return {
      confidenceScore,
      fraudRiskLevel,
      findings: {
        summary: `AI analysis completed with ${confidenceScore}% confidence using
Google Cloud Vision AI.`,
        overallAssessment: indicators.length === 0
          ? 'No significant irregularities detected. All analyzed features consistent
with authentic specimen.'
          : `Analysis detected ${indicators.length} area(s) requiring attention. Fur-
ther expert examination is recommended.`,
        keyObservations: [
          `Analyzed ${imageBuffers.length} high-resolution image(s)`,
          `Identified ${markers.length} positive authenticity marker(s)`,
          ...observations,
        ],
      },
      counterfeitIndicators: {
        found: indicators.length > 0,
        items: indicators,
      },
      authenticityMarkers: {
        found: markers.length > 0,
        items: markers,
      },
      processingTime,
    };
  } catch (error) {
    console.error('Google Vision API error:', error);
    throw error;
  }
}
```

Step 5: Update API Route

Modify `/app/api/items/[id]/ai-analysis/route.ts`:

```
import { analyzeWithGoogleVision } from '@/lib/ai-google-vision';
import { generateMockAnalysis } from '@/lib/ai-mock';

// In processAnalysis function, replace mock call:
async function processAnalysis(
  analysisId: string,
  item: any,
  imageIds: string[]
) {
  try {
    await prisma.aIAnalysis.update({
      where: { id: analysisId },
      data: { status: AIAnalysisStatus.processing },
    });

    // Get S3 keys for images
    const images = await prisma.mediaAsset.findMany({
      where: { id: { in: imageIds } },
      select: { cloudStoragePath: true },
    });
    const s3Keys = images.map(img => img.cloudStoragePath);

    // Use real Google Vision AI or fallback to mock
    const result = process.env.GOOGLE_CLOUD_PROJECT_ID
      ? await analyzeWithGoogleVision(item, s3Keys)
      : await generateMockAnalysis(item, imageIds);

    const apiProvider = process.env.GOOGLE_CLOUD_PROJECT_ID ? 'google-vision' :
'mock';

    // Update analysis with results
    await prisma.aIAnalysis.update({
      where: { id: analysisId },
      data: {
        status: AIAnalysisStatus.completed,
        confidenceScore: result.confidenceScore,
        fraudRiskLevel: result.fraudRiskLevel,
        findings: result.findings as any,
        counterfeitIndicators: result.counterfeitIndicators as any,
        authenticityMarkers: result.authenticityMarkers as any,
        processingTime: result.processingTime,
        apiProvider,
        completedAt: new Date(),
      },
    });

    // ... rest of the function
  } catch (error) {
    // ... error handling
  }
}
```

AWS Rekognition Integration

Step 1: Setup AWS Account and IAM

```
# Install AWS CLI
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install

# Configure credentials
aws configure
```

Create IAM policy for Rekognition:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rekognition:DetectLabels",
        "rekognition:DetectText",
        "rekognition:DetectModerationLabels",
        "rekognition:CompareFaces"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::YOUR_BUCKET_NAME/*"
    }
  ]
}
```

Step 2: Install Dependencies

```
yarn add @aws-sdk/client-rekognition
```

Step 3: Configure Environment Variables

```
# AWS Rekognition (uses existing AWS credentials)
AWS_REKOGNITION_ENABLED=true
```

Step 4: Create AWS Rekognition Utility

Create `/lib/ai-aws-rekognition.ts`:

```
import { RekognitionClient, DetectLabelsCommand, DetectTextCommand } from '@aws-sdk/
client-rekognition';
import { getBucketConfig } from './aws-config';
import { FraudRiskLevel } from '@prisma/client';

const client = new RekognitionClient({});
const { bucketName } = getBucketConfig();

export async function analyzeWithRekognition(
  item: any,
  imageS3Keys: string[]
): Promise<any> {
  const startTime = Date.now();
  const indicators: any[] = [];
  const markers: any[] = [];
  const observations: string[] = [];

  for (const s3Key of imageS3Keys) {
    // Label Detection
    const labelsCommand = new DetectLabelsCommand({
      Image: {
        S3Object: {
          Bucket: bucketName,
          Name: s3Key,
        },
      },
      MaxLabels: 20,
      MinConfidence: 70,
    });

    const labelsResponse = await client.send(labelsCommand);
    const labels = labelsResponse.Labels || [];

    // Analyze labels for category-specific markers
    if (item.category.slug === 'luxury-cars') {
      const carLabels = labels.filter(l =>
        ['Car', 'Vehicle', 'Automobile', 'Transportation'].includes(l.Name || '')
      );
      if (carLabels.length > 0) {
        markers.push({
          marker: 'Vehicle Identified',
          confidence: 'high',
          description: `Confirmed vehicle in image with ${carLabels[0].Confidence?.toF
ixed(1)}% confidence`
        });
      }
    }

    // Text Detection
    const textCommand = new DetectTextCommand({
      Image: {
        S3Object: {
          Bucket: bucketName,
          Name: s3Key,
        },
      },
    });

    const textResponse = await client.send(textCommand);
    const textDetections = textResponse.TextDetections || [];

    const detectedText = textDetections
```

```
      .filter(t => t.Type === 'LINE')
      .map(t => t.DetectedText)
      .join(' ');

    if (detectedText) {
      observations.push(`Text detected: ${detectedText.substring(0, 100)}`);
    }
  }

  // Calculate scores (similar to Google Vision)
  const confidenceScore = Math.max(50, Math.min(98, 70 + markers.length * 15 - indic-
ators.length * 10));
  const fraudRiskLevel = indicators.length === 0 ? FraudRiskLevel.low : FraudRisk-
Level.medium;

  return {
    confidenceScore,
    fraudRiskLevel,
    findings: {
      summary: `AI analysis completed with ${confidenceScore}% confidence using AWS
Rekognition.`,
      overallAssessment: indicators.length === 0
        ? 'No significant irregularities detected.'
        : `Analysis detected ${indicators.length} area(s) requiring attention.`,
      keyObservations: [
        `Analyzed ${imageS3Keys.length} image(s)`,
        `Identified ${markers.length} positive markers`,
        ...observations,
      ],
    },
    counterfeitIndicators: {
      found: indicators.length > 0,
      items: indicators,
    },
    authenticityMarkers: {
      found: markers.length > 0,
      items: markers,
    },
    processingTime: Date.now() - startTime,
  };
}
```

Cost Estimation

Google Cloud Vision AI

Feature	Free Tier (Monthly)	After Free Tier
Label Detection	1,000 units	$1.50/1,000
Text Detection	1,000 units	$1.50/1,000
Logo Detection	1,000 units	$1.50/1,000
Image Properties	1,000 units	$1.50/1,000

Average Cost per Analysis: ~$0.006 (4 features × $1.50/1,000)

Monthly Estimates:
- 100 analyses: **$0.60**
- 500 analyses: **$3.00**
- 1,000 analyses: **$6.00**
- 5,000 analyses: **$30.00**

AWS Rekognition

Feature	Free Tier (12 months)	After Free Tier
Label Detection	5,000 images/month	$1.00/1,000
Text Detection	1,000 images/month	$1.50/1,000
Face Comparison	1,000 images/month	$1.00/1,000

Average Cost per Analysis: ~$0.0025 (1 image × $2.50/1,000)

Monthly Estimates:
- 100 analyses: **$0.25**
- 500 analyses: **$1.25**
- 1,000 analyses: **$2.50**
- 5,000 analyses: **$12.50**

Recommendations

- **Small Volume (< 1,000/month):** Use free tiers
- **Medium Volume (1,000-10,000/month):** Hybrid approach
- **Large Volume (> 10,000/month):** Negotiate enterprise pricing

Testing Strategy

1. Unit Tests

```
// __tests__/ai-google-vision.test.ts
import { analyzeWithGoogleVision } from '@/lib/ai-google-vision';

describe('Google Vision AI Integration', () => {
  it('should detect text in watch dial', async () => {
    const result = await analyzeWithGoogleVision(mockWatchItem, [watchImageS3Key]);
    expect(result.authenticityMarkers.found).toBe(true);
    expect(result.confidenceScore).toBeGreaterThan(70);
  });

  it('should flag missing serial numbers', async () => {
    const result = await analyzeWithGoogleVision(mockHandbagItem, [genericIm-
ageS3Key]);
    expect(result.counterfeitIndicators.items).toContainEqual(
      expect.objectContaining({ indicator: 'Missing Text Elements' })
    );
  });
});
```

2. Integration Tests

```
# Test with real images
curl -X POST http://localhost:3000/api/items/{item-id}/ai-analysis \
  -H "Cookie: next-auth.session-token=..." \
  -H "Content-Type: application/json"
```

3. A/B Testing

- Run mock and real AI side-by-side
- Compare results for accuracy
- Validate fraud detection rate

Production Deployment

1. Background Job Queue (Recommended)

For production, use a job queue to handle async processing:

```
yarn add bullmq ioredis
```

Setup Queue:

```
// lib/queue.ts
import { Queue, Worker } from 'bullmq';
import { analyzeWithGoogleVision } from './ai-google-vision';

const connection = {
  host: process.env.REDIS_HOST,
  port: Number(process.env.REDIS_PORT),
};

export const aiAnalysisQueue = new Queue('ai-analysis', { connection });

// Worker
const worker = new Worker(
  'ai-analysis',
  async job => {
    const { analysisId, item, imageS3Keys } = job.data;
    const result = await analyzeWithGoogleVision(item, imageS3Keys);
    // Update database with result
  },
  { connection }
);
```

2. Error Handling

- Implement retry logic (3 attempts)
- Log failures to monitoring system
- Send admin alerts for critical failures

3. Rate Limiting

- Limit requests per user (e.g., 5 per hour)
- Implement queue prioritization
- Throttle API calls to CV providers

Monitoring and Optimization

Key Metrics

1. **Analysis Success Rate:** Target > 95%
2. **Average Processing Time:** Target < 30 seconds
3. **API Cost per Analysis:** Target < $0.01
4. **User Satisfaction:** Confidence score distribution

Dashboards

- Use admin page `/admin/ai-analyses` for real-time monitoring
- Track fraud detection accuracy
- Monitor API costs

Optimization Tips

1. **Image Preprocessing:**
   - Compress images before API calls
   - Use optimal resolution (1024×768)
   - Remove duplicates

2. **Caching:**
   - Cache API responses for identical images
   - Store analysis results for re-use

3. **Batch Processing:**
   - Analyze multiple items in parallel
   - Use Vision API batch endpoints

Next Steps

Phase 1: MVP (Week 1-2)

- [ ] Choose primary CV provider (Google Vision recommended)
- [ ] Setup GCP project and service account
- [ ] Implement basic integration for one category (e.g., watches)
- [ ] Test with 10-20 real items

Phase 2: Expansion (Week 3-4)

- [ ] Extend to all categories
- [ ] Add AWS Rekognition for cars/art
- [ ] Implement job queue
- [ ] Deploy to staging

Phase 3: Production (Week 5-6)

- [ ] Production deployment
- [ ] Monitor costs and performance
- [ ] Gather user feedback
- [ ] Iterate and optimize

Phase 4: Advanced Features

- [ ] Custom ML models for specific brands
- [ ] Integration with manufacturer databases
- [ ] Blockchain verification
- [ ] Expert review workflow

Support and Resources

Documentation

- Google Cloud Vision AI Docs (https://cloud.google.com/vision/docs)
- AWS Rekognition Developer Guide (https://docs.aws.amazon.com/rekognition/)
- Genesis Provenance API Reference (./API_REFERENCE.md)

Contact

For questions or issues during implementation:
- Technical Support: support@genesisprovenance.com
- Integration Help: dev@genesisprovenance.com

Last Updated: December 2024
Version: 1.0
Status: Ready for Implementation