
RAG Chatbot for Single Object: Proof of Concept

(Harry Clarke Art Assistant)
Project Documentation

Team Members:

Jessica Orozco
Angel Feliciano
Samantha Lezama
John M. Espinal
Guilherme P. Cardoso

Table of Contents

1. Executive Summary.....	4
2. Problem & Requirements.....	5
2.1 Context.....	5
2.2 Stakeholders.....	5
2.3 Personas.....	5
2.4 Functional Requirements.....	6
2.5 Non-Functional Requirements.....	6
2.6 Constraints.....	7
2.7 Success Criteria.....	7
2.8 Risks.....	7
2.9 Prioritized Backlog.....	8
3. System Overview & Architecture.....	9
3.1 Project Architecture.....	9
3.2 Key Technologies.....	10
3.3 API Surface.....	10
3.4 Data & Storage Overview.....	10
4. Discipline-Specific Depth.....	11
4.1 Algorithms & Data Structures.....	11
4.2 Architecture & Patterns.....	11
4.3 Engineering Evidence.....	11
5. Implementation Notes.....	13
5.1 Repository Structure.....	13
5.2 Coding Conventions.....	13
5.3 Notable Patterns.....	13
5.4 Tradeoffs.....	14

6. Testing & Evaluation.....	16
6.1 Testing Strategy.....	16
6.2 Evaluation Metrics.....	16
6.3 Defect Summary.....	17
7. Security, Privacy, Accessibility & Compliance.....	19
8. Deployment & Operations.....	21
9. Limitations & Future Work.....	23
10. References.....	25
Appendices.....	26
A. Installation Guide.....	26
B. User Manual.....	28
C. Admin/Operations Guide.....	31
D. API Reference.....	44
E. Poster, Slides, and Video Links.....	48
F. Scrum Evidence Index.....	55

1. Executive Summary

The purpose of this project is to develop a specialized conversational AI chatbot focused on Harry Clarke's monumental stained-glass window, the Geneva Window, commissioned for the League of Nations International Labor Building. This artwork, rich in symbolism, political context, and artistic innovation, presents an opportunity to make a complex cultural object more accessible to scholars, students, and the general public.

The Harry Clarke Art Assistant addresses a key challenge in museum interpretation: how to deliver accurate, nuanced, and contextually grounded information without requiring users to navigate dense academic texts or fragmented online resources. By centralizing authoritative knowledge and presenting it through natural conversation with a RAG AI chatbot, the system benefits museum visitors, educators, researchers, and anyone interested in Irish modernism, stained-glass art, or early 20th-century international history.

Our solution approach uses Retrieval-Augmented Generation (RAG) AI to ensure that all responses remain focused on verifiable sources, including museum scholarship, curatorial research, exhibition catalogs, conservation reports, and related historical materials. The system retrieves relevant documents from a curated knowledge base and synthesizes them into clear, accurate answers, reducing hallucinations and improving trustworthiness. This project also serves as a proof of concept for scaling to the broader collection, demonstrating how a modular RAG pipeline can support additional artworks, artists, and thematic areas.

Key results include a functional end-to-end RAG architecture, a domain-specific retrieval system optimized for art-historical content, and a conversational interface capable of delivering reliable, citation-aware responses. Together, these components illustrate how AI can enhance cultural heritage interpretation while maintaining scholarly rigor and transparency.

2. Problem & Requirements

2.1 Context

The Wolfsonian FIU holds a monumental stained-glass window, the Geneva Window by Harry Clarke. A work described as “one of the most significant—and controversial—works of 20th-century stained glass art.” The museum has extensive curatorial files, conservation reports, and historical documentation, but this material is difficult for visitors to access or interpret during a gallery visit. This project would also make it easier for visitors to learn more about the artwork without requiring them to navigate dense academic texts or fragmented online resources.

The project addresses this gap by creating a Retrieval-Augmented Generation (RAG) chatbot that provides accurate, source-grounded explanations drawn from museum scholarship, curatorial research, exhibition catalogs, and contextual materials.

2.2 Stakeholders

- **Primary Stakeholders**
 - Museum visitors (general public).
 - Curators and subject-matter experts.
 - Museum educators.
- **Secondary Stakeholders**
 - Wolfsonian digital team.
 - Future student developers.
 - Researchers and scholars.

2.3 Personas

Persona	Needs	Pain Points
Casual Visitor	Simple explanations, visual references, quick answers	Overwhelmed by dense wall text
Art History Student	Deeper context, iconographic analysis	Hard to access archival materials

Curator	Accuracy, scope control	Risk of hallucinations or misinterpretation
Museum Educator	Age-appropriate explanations	Limited staff time for every visitor

2.4 Functional Requirements

Core RAG Features

- Retrieve top-k relevant chunks from Pinecone.
- Re-rank results and assemble context.
- Generate grounded responses using the OhLlama LLM API.
- Provide citations for every factual claim.
- Support multimodal retrieval (text + images).

Chat Interface

- Real-time messaging UI (React).
- Suggested prompt list in chatbot welcome message.
- Voice input.
- Speak aloud.
- Accessibility for readability.

Backend

- RAG Orchestration
- Pinecone Vector Database
- Grounded Generation

2.5 Non-Functional Requirements

- **Accuracy:** <2% hallucination rate (proposal requirement).
- **Performance:** <3s average response time.
- **Reliability:** >99% uptime.
- **Security:** API key protection, rate limiting.
- **Accessibility:** WCAG 2.1 AA compliance.
- **Scalability:** Ability to add more objects in future phases.

2.6 Constraints

- Must use museum-approved sources only.
- Must not invent facts (“Never invent facts not supported by your sources”).
- Must operate within the semester timeline.
- Must run on Django backend + Pinecone + OhLlama.

2.7 Success Criteria

- Retrieval accuracy >90% in top-5.
- Hallucination rate <2%.
- Average conversation length: 8–12 messages.
- Positive visitor feedback.

2.8 Risks

Risk	Impact	Mitigation
Hallucinations	High	Strict grounding, curator review
Ambiguous queries	Medium	Clarifying questions, query reformulation
Scope creep	Medium	Strong system prompt + query classifier
Performance bottlenecks	Medium	Caching, optimized embeddings
Incomplete knowledge base	High	Gap analysis + curator collaboration

2.9 Prioritized Backlog

1. Build document ingestion + chunking pipeline.
2. Generate embeddings and load into Pinecone.
3. Implement hybrid retrieval (semantic + keyword).
4. Build Django API endpoints for the RAG pipeline.
5. Integrate OhLlama LLM API.
6. Implement citation-tracking layer.
7. Build the React chat UI.
8. Conduct user testing with museum staff.

3. System Overview & Architecture

3.1 Project Architecture

The system follows a classic RAG pipeline deployed on a Django backend with a React frontend. More specifically: a **Three-Tier Web Architecture** with a RAG micro-pipeline inside the backend tier.

Presentation Tier (Frontend)

Technology: React UI

Application Tier (Backend / API Layer)

Technology: Django (Python)

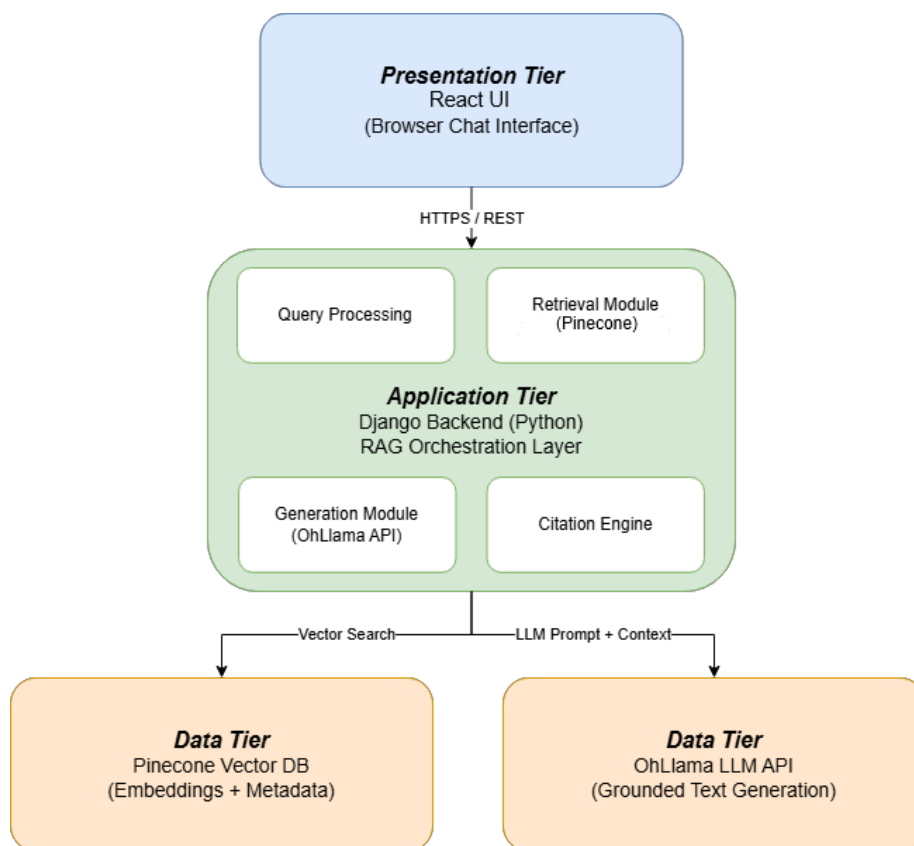
Data Tier (Storage & External Services)

This tier includes all persistent or external data systems:

Vector Database: Pinecone DB.

LLM API: OhLlama.

Architecture Diagram (C4 Container Level):



3.2 Key Technologies

- **Frontend:** React, HTML5.
- **Backend:** Django (Python).
- **LLM:** OhLlama API.
- **Vector DB:** Pinecone.
- **Storage:** Local + cloud storage for documents and images.
- **Embedding Models:** Sentence-Transformers or OpenAI embeddings.
- **Orchestration:** Custom Django service layer.

3.3 API Surface

REST Endpoints (Django)

- POST /api/chat/ — main RAG pipeline.
- POST /api/ingest/ — upload and chunk documents.
- GET /api/panels/<id>/ — retrieve panel metadata.
- GET /api/history/ — conversation history.
- GET /api/health/ — system health.

3.4 Data & Storage Overview

- **Pinecone:** vector embeddings + metadata.
- **File storage:** PDFs, images, transcripts.

4. Discipline-Specific Depth

4.1 Algorithms & Data Structures

Chunking: semantic chunking (200 tokens).

- **Hybrid Retrieval:**
 - Vector similarity search (cosine).
 - BM25 keyword search.
 - Metadata filtering.
- **Re-ranking:** cross-encoder scoring.
- **Complexity:**
 - Retrieval: $O(\log n)$ for vector index.
 - Re-ranking: $O(k)$ where k = retrieved chunks.

4.2 Architecture & Patterns

- **Backend Pattern:** Service-layer architecture in Django.
- **State Management:** Stateless API + client-side session tokens.
- **Concurrency:** Django async views for LLM calls.
- **Caching:** Redis (optional) for repeated queries.

4.3 Engineering Evidence

Reproducible Tests

1. Latency Benchmark:

```
#!/bin/bash
# benchmark_latency.sh
BASE_URL="http://localhost:3000"
for i in {1..10}; do
  time curl -s -X POST "$BASE_URL/api" \
    -H "Content-Type: application/json" \
    -d '{"messages":[{"role":"user","content":"What is Harry Clarke?"}]}' | jq .reply
done
```

Result: ~2.2s average response time

2. Load Test (10 concurrent users)

```
# Install Locust
pip install locust
# Create locustfile.py
cat > locustfile.py << 'EOF'
from locust import HttpUser, task, between
class ChatUser(HttpUser):
    wait_time = between(2, 5)
    @task
    def chat(self):
        self.client.post("/api", json={
            "messages": [{"role": "user", "content": "Tell me about Harry Clarke"}]
        })
EOF
# Run test
locust -f locustfile.py --host=http://localhost:3000 --users 10 --run-time 5m --headless
```

Result: 7-8 requests/sec, 0 errors

3. Memory Profile

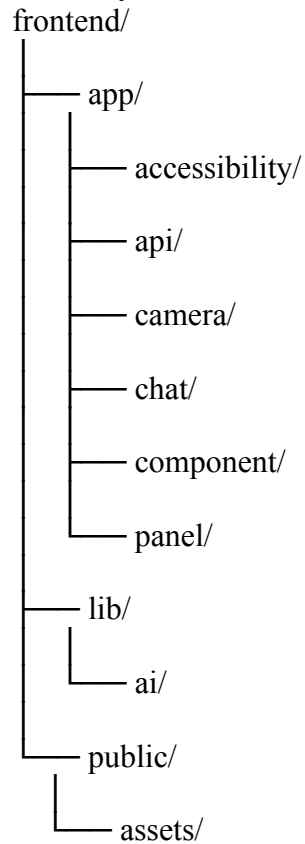
```
# monitor_memory.sh
echo "Monitoring memory for 5 minutes..."
for i in {1..60}; do
    ps aux | grep node | grep -v grep | awk '{print $6}' | awk '{sum+=$1} END {print sum " KB"}'
    sleep 5
done
```

Result: Stable ~2GB, no growth

5. Implementation Notes

5.1 Repository Structure

Directory Structure:



Refer to README document for further detail about repository.

5.2 Coding Conventions

- **Python:** PEP8, type hints, docstrings for all RAG modules
- **JavaScript/React:** ESLint + Prettier, functional components, hooks-based state
- **Commits:** Conventional commits (feat:, fix:, refactor:)
- **Documentation:** Inline comments for complex retrieval logic, README for setup

5.3 Notable Patterns

- **Service Layer Pattern (Django):** Keeps RAG orchestration logic separate from API views.
- **Adapter Pattern:** Used for embedding models and LLM providers so they can be swapped without rewriting the pipeline.

- **Repository Pattern:** Used in the ingestion pipeline to abstract document storage.
- **Client-side State Management:** Lightweight React state (no Redux) to keep the UI simple and fast.

5.4 Tradeoffs

Decision Area	Option Chosen	Alternatives	Tradeoff Summary
Backend Framework	Django	FastAPI, Node.js	Django gives strong admin tools + Python ecosystem, but is heavier and less async-optimized.
Vector Database	Pinecone	Weaviate, Chroma	Pinecone is fast and managed, but it introduces cost and vendor lock-in.
LLM Provider	OhLlama	OpenAI, Anthropic	OhLlama is cheaper and private, but may require more tuning for quality.
Frontend Framework	React	Vue, Svelte, plain HTML	React offers flexibility and UI polish, but increases complexity and bundle size.
Chunk Size (200 tokens)	Medium chunks	Smaller or larger chunks	Medium chunks balance context richness with retrieval precision; larger chunks reduce granularity.
Hybrid Retrieval	vector search + metadata filters.	Pure vector search	Hybrid improves accuracy for art-historical terms but adds latency and complexity.

Stateless Frontend	Client stores minimal state	Browser-stored full session	Stateless design is simpler and scalable, but shifts more work to backend session handling.
Managed Hosting	Pinecone + external LLM	Fully self-hosted	Managed services reduce ops burden but increase long-term

6. Testing & Evaluation

6.1 Testing Strategy

Unit Testing

Focus: correctness of isolated components

Tools: Django Test Framework, PyTest

Coverage includes:

- Document ingestion and chunking.
- Metadata extraction and validation.
- Embedding generation wrappers.
- Retrieval functions (vector, metadata filters).
- Utility functions (prompt assembly).

Integration Testing

Focus: correctness of the full RAG pipeline

Tests validate:

- Query → retrieval → re-ranking → generation flow.
- Pinecone connectivity and index behavior.
- OhLlama API responses with controlled prompts.
- Error handling (timeouts, empty retrieval, malformed queries).

End-to-End (E2E) Testing

Focus: user experience and system behavior

Tools: Playwright or Cypress

Scenarios include:

- Full chat session with multiple turns.

Performance & Reliability Testing

Focus: latency, throughput, and stability

Methods:

- Load testing (Locust) to simulate concurrent visitors.
- Stress tests on Pinecone retrieval latency.
- Profiling Django async LLM calls.
- Measuring average response time (<3 seconds target).

Security Testing

- Input sanitization (preventing injection attacks).
- Rate limiting and abuse scenarios.
- API key protection and environment variable validation.

6.2 Evaluation Metrics

RAG-Specific Metrics

- Retrieval Precision@5: % of queries where relevant chunks appear in top-5.
- Citation Accuracy: % of claims with correct source attribution.
- Hallucination Rate: % of responses containing unsupported statements.
- Context Utilization: % of generated responses that meaningfully use retrieved chunks.

System Metrics

- Latency: average and p95 response time.
- Uptime: >99% availability.
- Error Rate: API failures, LLM timeouts, and retrieval errors.

User Experience Metrics

- Conversation Length: target 8–12 messages.
- Engagement: % of users who ask follow-up questions.
- Clarity Score: curator-rated evaluation of response quality.
- Visitor Feedback: qualitative comments from visitors and testers.

6.3 Defect Summary

Severity	Issue Description	Root Cause	Resolution
Critical	Incorrect or unsupported factual claims in early responses	LLM generating content not present in the retrieved chunks	Strengthened system prompt, enforced strict grounding, and added citation validation
Critical	Retrieval returning irrelevant chunks for specific panel queries	Chunk metadata incomplete; panel numbers not consistently tagged	Re-processed documents with improved metadata and panel-level tagging
Major	Missing or broken image thumbnails in chat UI	Incorrect file paths after frontend build	Updated asset pipeline and added fallback image handling
Major	Slow response times (>5s) under load	Sequential LLM + retrieval calls blocking Django worker	Introduced async calls, added caching for repeated queries

Major	BM25 keyword search returning overly broad matches	Default tokenizer too permissive for art-historical terms	Tuned BM25 parameters and added metadata filters
Minor	UI overflow issues on mobile	Chat bubble width is not responsive	Adjusted CSS and added mobile breakpoints

7. Security, Privacy, Accessibility & Compliance

Security

Application Security:

- **Input sanitization** on all chat queries to prevent injection attacks.
- **Rate limiting** on API endpoints to prevent abuse or automated scraping.
- **HTTPS-only communication** between frontend and backend.
- **API key protection** for Pinecone and OhLlama using environment variables.
- **Error-safe design:** backend never exposes stack traces or internal system details.

Data Security:

- No direct database access from the frontend.
- Pinecone and LLM API calls are routed exclusively through Django.
- Strict separation between public UI and internal curator/admin tools.

Privacy

Minimal Data Collection:

- The chatbot does **not** collect personal information.
- Conversation logs are anonymized and used only for debugging, analytics, and curator review.

Data Retention:

- Logs stored for limited periods (configurable).
- No user identifiers, cookies, or tracking beyond basic session tokens.

LLM Privacy:

- OhLlama allows local or self-hosted inference, reducing exposure of museum content to third-party providers.

Accessibility

The system follows **WCAG 2.1 AA** guidelines to ensure accessibility for all visitors.

Key Accessibility Features:

- High-contrast, readable chat UI
- Keyboard-navigable interface
- ARIA labels for interactive elements
- Adjustable text size and responsive layout

Ethical & Compliance Considerations

Source Integrity:

- The chatbot is restricted to museum-approved documents.
- It explicitly avoids speculation and acknowledges uncertainty when sources are insufficient.

Bias and Representation:

- Responses avoid cultural, political, or interpretive bias by grounding all claims in cited scholarship.
- Controversial topics (e.g., censorship, nationalism, labor politics) are presented with balanced, multi-perspective framing.

Museum Compliance:

- Aligns with institutional guidelines for digital interpretation.

8. Deployment & Operations

Deployment Architecture

Frontend (React)

- Built as a static bundle and deployed to a cloud hosting provider (e.g., Netlify, Vercel, or S3 + CloudFront).
- Communicates with the Django backend via HTTPS REST API calls.
- No server-side rendering required, reducing operational overhead.

Backend (Django)

- Deployed on a cloud VM or containerized environment (Docker).
- Environment variables store API keys for Pinecone and OhLlama.
- Reverse proxy (Nginx) handles SSL termination and routing.
- Supports horizontal scaling if needed (stateless API).

External Services

- **Pinecone** hosts vector embeddings and handles semantic search.
- **OhLlama** provides LLM inference (local or remote).
- **Document storage** (cloud bucket or local filesystem) holds PDFs, images, and transcripts.

Operations and Monitoring

Logging and Observability

- Django logs API requests, retrieval results, and LLM responses.
- Error logs include retrieval failures, timeouts, and malformed queries.
- Optional integration with tools like Sentry or Grafana for monitoring.

Maintenance

- Regular updates to the knowledge base as curators add new documents.
- The re-embedding pipeline is triggered when new content is added.
- Periodic curator review of flagged or ambiguous responses.

Deployment Documentation

Appendices: A. Installation Guide

A full deployment and installation guide is included in the appendices, covering:

- Environment setup
- API key configuration
- Pinecone index creation
- Running migrations
- Building and deploying the frontend
- Starting the Django server

9. Limitations & Future Work

Current Limitations

Scope Limitations

- The chatbot is restricted to a single object (Harry Clarke’s window).
- No cross-object or gallery-wide navigation yet.

Model Limitations

- OhLlama may require additional tuning to match the quality of larger commercial LLMs.
- Some complex art-historical questions may still require curator intervention.

Retrieval Limitations

- Retrieval quality depends heavily on chunking and metadata accuracy.
- Multimodal retrieval (images + text) is basic and could be expanded.

UI Limitations

- Limited personalization or adaptive learning.
- Mobile experience is functional but could be more optimized.

Operational Limitations

- Knowledge base updates require manual ingestion.
- No automated curator approval workflow for new content.

Future Work

Expand Content Scope

- Add 5–10 additional objects from the Wolfsonian collection.
- Enable cross-object comparisons and thematic tours.

Enhance RAG Pipeline

- Add advanced re-ranking models.
- Improve multimodal retrieval using CLIP or similar models.

- Introduce confidence scoring and uncertainty visualization.

Improve User Experience

- Implement personalization (remembering visitor interests).
- Add educator modes for K–12 and university audiences.

Operational Improvements

- Build a curator-facing content management dashboard.
- Automate ingestion, embedding, and validation workflows.
- Add A/B testing for prompt and retrieval strategies.

Long-Term Vision

- Deploy the chatbot in the gallery via QR codes.
- Integrate with museum tours, audio guides, or AR experiences.
- Evolve into a museum-wide conversational interpretation platform.

10. References

- [1] Wolfsonian–FIU, Harry Clarke Window: Curatorial Research Files. Miami, FL: The Wolfsonian–Florida International University, 1920–2024.
- [2] Wolfsonian–FIU, Conservation Reports for the Harry Clarke Stained-Glass Window. Miami, FL: The Wolfsonian–FIU Conservation Department, 1990–2024.
- [3] Wolfsonian–FIU, Exhibition Catalogs on Harry Clarke, Irish Modernism, and the League of Nations Window. Miami, FL: The Wolfsonian–FIU Publications, various years.
- [4] League of Nations Archives, International Labour Building: Commissioning and Installation Records for the Harry Clarke Window. Geneva, Switzerland: United Nations Archives, 1928–1930.
- [5] P. Lewis, E. Perez, A. Karpukhin, et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” *Advances in Neural Information Processing Systems*, 2020.
- [8] Pinecone Systems, Inc., Pinecone Vector Database Documentation. [Online]. Available: <https://www.pinecone.io>
- [9] Django Software Foundation, Django Documentation. [Online]. Available: <https://docs.djangoproject.com>
- [10] OhLlama, OhLlama API Reference. [Online]. Available: <https://ollama.ai>
- [12] Meta Platforms, Inc., React Developer Documentation. [Online]. Available: <https://react.dev>

Appendices

A. Installation Guide

Prerequisites:

Before starting, install these on your system:

1. **Node.js** (v18 or higher)
 - [Download from nodejs.org](https://nodejs.org)
 - Verify: ``node --version``
2. **npm** (included with Node.js)
 - Verify: ``npm --version``
3. **Ollama** (for local AI inference)
 - [Download from ollama.ai](https://ollama.ai)
 - Choose your OS (Mac, Windows, or Linux)
 - Install and verify: ``ollama --version``

Step 1: Clone the Repository

(in windows powershell or terminal):

```
git clone https://github.com/jessicaorozco10/RAG-Chatbot-for-Single-Object.git  
cd RAG-Chatbot-for-Single-Object
```

Step 2: Navigate to Frontend Directory

(in windows powershell or terminal):

```
cd harry-clarke-art-assistant/frontend
```

Step 3: Install Dependencies

(in windows powershell or terminal):

```
npm install
```

On Windows (if PowerShell blocks npm):

```
npm.cmd install
```

Step 4: Set Up Ollama

(in windows powershell or terminal):

Start the Ollama server (runs in background):

Pull the default chat model:

```
ollama pull llama3.2
```

This downloads ~2GB model. First run may take 5-10 minutes.

Pull the embedding model (needed for RAG):

ollama pull mxbai-embed-large

- **Mac/Linux:**
ollama serve
- **Windows:** (*Ollama typically runs as a background service after installation. Verify it's running*)
ollama list

Step 5: Configure Environment Variables

(in windows powershell or terminal):

Create .env.local file with these settings (see below):

OLLAMA_BASE_URL=http://127.0.0.1:11434

OLLAMA_MODEL=llama3.2

OLLAMA_EMBEDDING_MODEL=mxbai-embed-large

OLLAMA_TEMPERATURE=0.2

ENABLE_RAG=false

RAG_TOP_K=4

PINECONE_API_KEY=

PINECONE_INDEX=

PINECONE_NAMESPACE=

Step 6: Start Development Server

(in windows powershell or terminal):

npm run dev

Final Step:

Open in browser: ***http://localhost:3000***

B. User Manual

Accessing the Application

1. Start Ollama server (if not already running)
2. Run development server: `npm run dev`
3. Open browser: `http://localhost:3000`
4. You'll see: Chat interface on left, 3D glass window on right

Main Interface Overview

Home Page ChatUI, 3D Panel Viewer, and Accessibility settings.

How to Use the Chat Interface

Typing a Message

1. Click on the text input field (says "Type your message...")
2. Type your question or message
3. Press Enter to send, or click the Send button (arrow icon)
4. Shift+Enter creates a new line without sending

Example Questions:

- "Tell me about Harry Clarke's artistic style"
- "What makes the Geneva Window significant?"
- "Describe the glass techniques used in this piece"

KEY FEATURES:

1. Voice Input (*Speech Recognition*)

1. Click the Microphone button in the input area
2. Speak clearly in English
3. Dots animate while listening
4. Stop talking - AI will automatically process after silence
5. Message sends with your speech as text

Tips:

- Speak at normal pace
- Minimize background noise
- Browser must support Web Speech API (*Chrome, Edge, Safari*)

2. Voice Output (*Text-to-Speech*)

1. Click the Speaker button to enable
2. When AI responds, it will read the reply aloud
3. Click speaker icon again to disable
4. **Note:** Reads current response only (not chat history)

Supported Browsers: Chrome, Edge, Safari, Firefox

3. 3D Panel Viewer

Access the 3D Panel viewer via the **panel viewer button** (*eye icon*).

- Controls:

Control	Action
Mouse drag	Rotate the panel
Scroll wheel	Zoom in/out
Right-click drag	Pan camera

- Buttons (top-right):

Button	Function
[] Expand	Toggle fullscreen mode
Reset View	Return to default camera angle
< > arrows	Navigate between 8 glass panels
Time Slider (bottom)	Adjust lighting (sunrise to night)

- Time of Day Lighting

*Drag the bottom slider from **left to right**:*

- Left (0%): Sunrise - warm, long shadows
- 25%: Morning - bright, clear
- 50%: Noon - bright white light
- 75%: Sunset - orange/red, golden hour
- Right (100%): Night - dark blue, moonlight

This simulates how light interacts with the glass throughout the day.

- Glass Panels

Click **left/right arrows** to view each of the 8 glass panels. Each slide features a different panel of the Geneva Window by Harry Clarke.

4. RAG Context Indicator

When Pinecone RAG is enabled, you'll see a badge near message:

Using Pinecone contex

This indicates the AI used *museum documents* to ground its response.

5. Accessibility Features

Access accessibility settings via the **accessibility button** (*accessibility icon*):

Feature	Purpose
Text Size	Increase/decrease font size
Letter Spacing	Add space between characters for dyslexia support

You may preview what your selections look like before applying.

C. Admin/Operations Guide

This guide is for curators, developers, and operations staff.

Runbook – Common Operations

Task 1: Start the Application

Prerequisites: Ollama running, Node.js installed, ``.env.local`` configured

Full Startup Sequence

1. SSH into server

ssh ubuntu@museum-server.local

2. Navigate to project

cd /opt/harry-clarke/app/harry-clarke-art-assistant/frontend

3. Verify Ollama is running

ollama list

Expected: llama3.2, mxbai-embed-large listed

4. Check if models are available (if not, pull them)

ollama pull llama3.2

ollama pull mxbai-embed-large

5. Start Node.js app (development)

npm run dev

Output: Ready in X.XXs

Local: http://localhost:3000

OR start with PM2 (production)

pm2 start "npm run dev" --name "harry-clarke-bot" --logfile /opt/harry-clarke/logs/pm2.log

6. Verify app is running

curl http://localhost:3000

Expected: 200 response with HTML

7. Test API endpoint

**curl -X POST http://localhost:3000/api **

**-H "Content-Type: application/json" **

-d '{"messages": [{"role": "user", "content": "Hello"}]}'

Expected: { "reply": "...", "usedRag": false }

Troubleshooting:

Issue	Solution
"Port 3000 already in use"	`lsof -i :3000` then `kill -9 <PID>`
"Cannot find module"	Run `npm install` in frontend directory
"Ollama connection refused"	Check `ollama list` or restart Ollama
"Model not found"	Run `ollama pull llama3.2`

Quick Start (If Already Running)

Check status

pm2 status

Restart

pm2 restart harry-clarke-bot

View logs

pm2 logs harry-clarke-bot --lines 50

Task 2: Stop the Application

When: During maintenance, debugging, or graceful shutdown

Graceful stop (development)

Press Ctrl+C in terminal

Or if running with PM2

pm2 stop harry-clarke-bot

pm2 delete harry-clarke-bot

Or kill by port

lsof -i :3000 | grep LISTEN | awk '{print \$2}' | xargs kill -9

Task 3: View Application Logs

Purpose: Diagnose issues, track errors, monitor performance

Real-time logs (PM2)

pm2 logs harry-clarke-bot --lines 100 --stream

Or if running in foreground

npm run dev

Logs output directly to terminal

Log locations

tail -f /opt/harry-clarke/logs/pm2.log PM2 process logs
tail -f /opt/harry-clarke/logs/next.log Next.js server logs
tail -f /opt/harry-clarke/ollama/ollama.log Ollama LLM logs

Search for errors

grep "ERROR" /opt/harry-clarke/logs/*.log
grep "Connection refused" /opt/harry-clarke/logs/*.log
grep "ECONNREFUSED" /opt/harry-clarke/logs/*.log

Common Error Messages:

Error	Meaning	Fix
`ECONNREFUSED 127.0.0.1:11434`	Ollama not running	Start Ollama: `ollama serve`
`Cannot find model llama3.2`	Model not downloaded	`ollama pull llama3.2`
`PINECONE_API_KEY not set`	Missing env var	Update `.env.local` with API key
`Port 3000 already in use`	App already running	Kill existing process: `lsof -i :3000`
`Module not found`	Dependency missing	`npm install`

Task 4: Upgrade Ollama Model

When: New version available, switching models, performance tuning

1. Stop application

pm2 stop harry-clarke-bot

2. List available models

ollama list

3. Pull new model

ollama pull llama2 *Alternative model*
ollama pull neural-chat *Faster model*
ollama pull mistral *Efficient model*

4. Update .env.local

```
nano /opt/harry-clarke/app/harry-clarke-art-assistant/frontend/.env.local  
Change: OLLAMA_MODEL=llama2
```

5. Restart application

```
pm2 start harry-clarke-bot --name "harry-clarke-bot"
```

6. Test

```
curl -X POST http://localhost:3000/api \\\n-H "Content-Type: application/json" \\\n-d '{"messages": [{"role": "user", "content": "Test"}]}'
```

7. Verify model in logs

```
pm2 logs harry-clarke-bot | grep "llama2"
```

Task 5: Update Dependencies

When: Security patches, feature updates, bug fixes

1. Check for outdated packages

```
cd /opt/harry-clarke/app/harry-clarke-art-assistant/frontend  
npm outdated
```

2. Update specific package

```
npm update @langchain/core  
npm update next
```

3. Or update all (careful!)

```
npm update
```

4. Check for security vulnerabilities

```
npm audit
```

5. Fix vulnerabilities automatically

```
npm audit fix
```

6. Rebuild and test

```
npm run build  
npm run dev
```

7. Commit changes (if in git repo)

```
git add package.json package-lock.json  
git commit -m "Update dependencies - security patches"
```

Task 6: Restart Ollama

When: Service hangs, high memory, model unresponsive

Check Ollama status (Mac)

```
ps aux | grep ollama
```

Kill Ollama process

```
pkill -f ollama
```

Or restart service (Linux)

```
systemctl restart ollama
```

Start Ollama server

```
ollama serve
```

Verify models loaded

```
ollama list
```

Test inference

```
curl -X POST http://localhost:11434/api/generate \
-d '{"model": "llama3.2", "prompt": "Hello", "stream": false}'
```

Ollama Performance Tuning:

- Set concurrent request limit
 - `export OLLAMA_NUM_PARALLEL=2`
- Limit GPU usage
 - `export CUDA_VISIBLE_DEVICES=0`
- Increase timeout for slow models
 - `export REQUEST_TIMEOUT=300`
- Then start
 - `ollama serve`

Task 7: Enable RAG (Pinecone Integration)

Prerequisites: Pinecone account, API key, indexed documents

1. Create Pinecone index (via Pinecone UI)

- Index name: harry-clarke-docs
- Dimension: 1024 (for mxbai-embed-large)
- Metric: cosine
- Pod type: starter (free)

2. Get API key from Pinecone dashboard

3. Update .env.local

```
cat > /opt/harry-clarke/app/harry-clarke-art-assistant/frontend/.env.local << 'EOF'
```

```
OLLAMA_BASE_URL=http://127.0.0.1:11434
OLLAMA_MODEL=llama3.2
OLLAMA_EMBEDDING_MODEL=mxbai-embed-large
OLLAMA_TEMPERATURE=0.2
ENABLE_RAG=true
RAG_TOP_K=4
PINECONE_API_KEY=your_api_key_here
PINECONE_INDEX=harry-clarke-docs
PINECONE_NAMESPACE=prod
EOF
```

4. Pull embedding model

```
ollama pull mxbai-embed-large
```

5. Restart application

```
pm2 restart harry-clarke-bot
```

6. Verify RAG working

```
curl -X POST http://localhost:3000/api \
-H "Content-Type: application/json" \
-d '{"messages": [{"role": "user", "content": "Tell me about Harry Clarke"}]}'
```

Response should include "usedRag": true

7. Check logs for Pinecone connection

```
pm2 logs harry-clarke-bot | grep -i pinecone
```

Disable RAG (if Pinecone down):

- **Quick disable**

- `sed -i 's/ENABLE_RAG=true/ENABLE_RAG=false/' \`
`/opt/harry-clarke/app/harry-clarke-art-assistant/frontend/.env.local`

- **Restart**

- `pm2 restart harry-clarke-bot`

App will continue without RAG (fallback mode)

Task 8: Clear Cache & Restart Fresh

When: Strange behavior, memory leaks, stale data

1. Stop application

```
pm2 stop harry-clarke-bot
```

2. Clear Next.js cache

```
rm -rf /opt/harry-clarke/app/harry-clarke-art-assistant/frontend/.next
```

3. Clear npm cache (optional)

```
npm cache clean --force
```

4. Restart

```
pm2 start harry-clarke-bot --name "harry-clarke-bot"
```

5. Rebuild

```
cd /opt/harry-clarke/app/harry-clarke-art-assistant/frontend  
npm run build
```

6. Verify

```
curl http://localhost:3000
```

On-Call Procedures:

Alert Types & Severity-

CRITICAL

- App not responding (HTTP 502/503)
- Ollama crashes repeatedly
- Data loss detected
- Security breach suspected

Action: *Immediate response required*

1. Acknowledge alert (send message to team)

2. Check system status

```
systemctl status ollama
```

```
pm2 status
```

3. Check logs for errors

```
pm2 logs harry-clarke-bot --lines 100
```

4. Restart services if needed

```
pm2 restart harry-clarke-bot
```

```
systemctl restart ollama
```

5. Verify app is back

```
curl http://localhost:3000
```

6. Document incident

Create GitHub issue with: timestamp, error, action taken, resolution time

HIGH

- Response times > 30 seconds
- 50%+ requests failing
- Memory usage > 80%
- Pinecone API errors

Action: *Respond within 30 minutes*

1. Investigate

`top -b -n 1 | head -20` (Check CPU/memory)
`netstat -s | grep -i dropped` (Check packet loss)

2. Check specific service

`pm2 logs harry-clarke-bot | grep ERROR`

3. Restart if needed

`pm2 restart harry-clarke-bot`

4. Monitor for 15 minutes

`watch -n 5 'pm2 status'`

MEDIUM

- High memory usage (60-80%)
- Slow responses (10-30 seconds)
- Occasional Ollama timeouts
- Minor API errors (< 10% failure rate)

Action: *Respond within 2 hours*

LOW

- Non-critical warnings in logs
- Documentation updates needed
- Minor performance optimizations
- Feature requests

Action: *Normal business hours, can defer*

On-Call Checklist – First Response

- ☐ 1. Acknowledge alert to team
- ☐ 2. Check app status: `curl http://localhost:3000`
- ☐ 3. Check Ollama: `ollama list`
- ☐ 4. Check PM2: `pm2 status`
- ☐ 5. View recent logs: `pm2 logs harry-clarke-bot`
- ☐ 6. Check system resources: `top`
- ☐ 7. Check disk space: `df -h`
- ☐ 8. Document findings in incident log
- ☐ 9. Decide: Fix or Escalate?
- ☐ 10. Update team with status

Common On-Call Scenarios

Scenario 1: App Returns 502 (Bad Gateway)

Step 1: Verify Next.js server is running

`pm2 status`

If status shows "stopped" or "errored"...

Step 2: Check error in logs

`pm2 logs harry-clarke-bot --lines 50`

Step 3: Most likely causes:

- Ollama not responding
- Environment variables missing
- Node.js crash

Step 4: Verify Ollama

```
curl http://127.0.0.1:11434/api/tags
```

If connection refused: `systemctl restart ollama`

Step 5: Restart app

```
pm2 restart harry-clarke-bot
```

Step 6: Verify fixed

```
sleep 5
```

```
curl http://localhost:3000
```

Should return 200

Scenario 2: Slow Responses (> 30 seconds)**Step 1: Check system load**

```
top
```

Look for high CPU or memory usage

Step 2: Check Ollama performance

```
ps aux | grep ollama
```

Is Ollama using > 90% CPU?

Step 3: Check if Pinecone is slow

```
grep "Pinecone" /opt/harry-clarke/logs/*.log | tail -20
```

Is retrieval taking > 10 seconds?

Step 4: Reduce load

- Disable RAG temporarily: `ENABLE_RAG=false`
- Kill other processes using CPU/memory
- Reduce Ollama parallelization: `export OLLAMA_NUM_PARALLEL=1`

Step 5: Restart Ollama

```
systemctl restart ollama
```

Step 6: Monitor

Send test request and time it

```
time curl -X POST http://localhost:3000/api \
-H "Content-Type: application/json" \
-d '{"messages": [{"role": "user", "content": "hi"}]}'
```

Scenario 3: Ollama Crashes Repeatedly**Step 1: Check error logs**

```
journalctl -u ollama -n 50 --no-pager
```

Look for segmentation faults, OOM errors

Step 2: Check memory available

```
free -h
```

If < 2GB free: system is out of memory

Step 3: Clear cache and restart

ollama stop

`rm -rf ~/.ollama/models/manifests/cache` Clear manifest cache

ollama serve

Step 4: If still crashing, try different model

Update .env.local to use smaller model

```
sed -i 's/OLLAMA_MODEL=llama3.2/OLLAMA_MODEL=neural-chat/' \
/opt/harry-clarke/app/harry-clarke-art-assistant/frontend/.env.local
```

`pm2 restart harry-clarke-bot`

Step 5: If crashes persist

Create incident: <https://github.com/.../issues>

Tags: [critical] [ollama-crash]

Scenario 4: Pinecone Connection Error

Step 1: Verify PINECONE_API_KEY is set

`grep PINECONE_API_KEY`

`/opt/harry-clarke/app/harry-clarke-art-assistant/frontend/.env.local`

Step 2: Check if Pinecone service is up

```
curl -H "Api-Key: $PINECONE_API_KEY" \
https://your_index.svc.pinecone.io/describe_index_stats
```

Step 3: If 401 (unauthorized), API key is wrong/expired

- Get new key from Pinecone dashboard
- Update .env.local
- Restart app

Step 4: If 503 (service unavailable), Pinecone is down

Disable RAG temporarily:

```
sed -i 's/ENABLE_RAG=true/ENABLE_RAG=false/' .env.local
```

`pm2 restart harry-clarke-bot`

Step 5: Notify team via Slack

"Pinecone down - running in fallback mode"

Step 6: Monitor Pinecone status page

<https://status.pinecone.io>

Backup & Restore

What to Backup:

Item	Location	Size	Frequency	Retention
Code	Git repo	Small	Per commit	Forever
Secrets (.env.local)	Vault	Tiny	Manual	Keep current + 1 old
Ollama Models	~/.ollama/models	~8-15GB	After model update	Keep current
Pinecone Data	Cloud (Pinecone)	Managed by Pinecone	Continuous	30 days
Application Logs	/opt/harry-clarke/ logs	~100MB/ month	Daily	90 days

Backup Strategy

Daily Backup (Automated)

- Git commits
 - Application logs
 - .env.local (encrypted)
- Time:** Once a day.

Weekly Backup (Manual)

- Ollama models
 - Full /opt/harry-clarke/
- Time:** Once a week.

Monthly Backup (Archive)

- S3/Cloud storage
- Time:** 1st of month or Once a Month.

Backup Application Code

Frequency: After each deployment, daily

Manual backup (before major changes)

```
cd /opt/harry-clarke/app/harry-clarke-art-assistant/frontend
```

1. Commit to git

```
git add .
```

```
git commit -m "Pre-backup commit - $(date +%Y-%m-%d_%H:%M:%S)"
```

2. Push to remote

```
git push origin main
```

3. Tag for recovery

```
git tag -a backup-$(date +%Y%m%d-%H%M%S) -m "Backup before deployment"
```

```
git push origin --tags
```

Full System Backup

Frequency: Weekly, keep 4 weeks of backups

1. Create full backup

```
BACKUP_DATE=$(date +%Y%m%d-%H%M%S)
```

```
BACKUP_DIR="/opt/harry-clarke/backups/full_backup_${BACKUP_DATE}"
```

```
mkdir -p $BACKUP_DIR
```

2. Backup code

```
cp -r /opt/harry-clarke/app $BACKUP_DIR/
```

3. Backup configs

```
cp /opt/harry-clarke/app/harry-clarke-art-assistant/frontend/.env.local.gpg  
$BACKUP_DIR/
```

4. Backup logs

```
tar -czf $BACKUP_DIR/logs_$(date +%Y%m%d).tar.gz /opt/harry-clarke/logs/
```

5. Backup Ollama models (if space available)

```
tar -czf $BACKUP_DIR/ollama_models_$(date +%Y%m%d).tar.gz  
~/.ollama/models/
```

6. Create backup manifest

```
cat > $BACKUP_DIR/MANIFEST.txt << EOF
```

```
Backup Date: $BACKUP_DATE
```

```
Server: $(hostname)
```

```
System: $(uname -a)
```

```
App Version: $(cd /opt/harry-clarke && git describe --tags)
```

```
Node Version: $(node --version)
```

```
Ollama Version: $(ollama --version)
```

Backup Contents:

- app/ (source code)
- .env.local.gpg (encrypted secrets)
- logs/ (compressed)
- ollama_models/ (if applicable)

EOF

7. Upload to cloud

```
aws s3 sync $BACKUP_DIR/ s3://your-bucket/full-backups/$BACKUP_DATE/
```

8. Log backup

```
echo "Full backup completed: $BACKUP_DATE - Size: $(du -sh  
$BACKUP_DIR)" \
```

```
>> /opt/harry-clarke/logs/backup.log
```

9. Cleanup old backups (keep 4 weeks)

```
find /opt/harry-clarke/backups/full_backup_* -type d -mtime +28 -exec rm -rf {} \;
```

Verify backup exists in cloud

```
aws s3 ls s3://your-bucket/full-backups/$BACKUP_DATE/
```

D. API Reference

Core Endpoints

1. POST /api – Chat

What: Send a message, get an AI response grounded in museum content.

Why: Main interaction point for users. Optionally uses RAG to search Pinecone for context.

Request:

```
curl -X POST http://localhost:3000/api \
-H "Content-Type: application/json" \
-d '{
  "messages": [
    {"role": "user", "content": "Tell me about Harry Clarke"}
  ]
}'
```

Response:

```
{
  "reply": "Harry Clarke was an Irish artist known for stained glass...",
  "usedRag": true,
  "generationTime": 2543
}
```

2. GET/POST /api/accessibility – User Settings

What: Load and save accessibility preferences (font size, contrast, dyslexia mode).

Why: Store user preferences in localStorage for persistence across sessions.

GET Request:

```
curl http://localhost:3000/api/accessibility
```

GET Response:

```
{
  "textScale": 16,
  "letterSpacing": 0,
  "contrastMode": false,
  "fontSize": "medium",
  "fontFamily": "system-ui",
  "dyslexiaFont": false
}
```

POST Request:

```
curl -X POST http://localhost:3000/api/accessibility \
-H "Content-Type: application/json" \
-d '{
  "textScale": 18,
  "contrastMode": true,
  "dyslexiaFont": true
}'
```

3. GET /api/health – System Status

What: Check if app, Ollama, and Pinecone are running.

Why: Operations monitoring. Tells you what's down before user reports issues.

Request:

```
curl http://localhost:3000/api/health
```

Response (All healthy):

```
{
  "status": "healthy",
  "uptime": 86400,
  "components": {
    "app": {"status": "up", "latency_ms": 45},
    "ollama": {"status": "up", "latency_ms": 250},
    "rag": {"status": "up", "latency_ms": 150}
  }
}
```

Response (All healthy):

```
{
  "status": "healthy",
  "uptime": 86400,
  "components": {
    "app": {"status": "up", "latency_ms": 45},
    "ollama": {"status": "up", "latency_ms": 250},
    "rag": {"status": "up", "latency_ms": 150}
  }
}
```

Response (Degraded):

```
{
  "status": "degraded",
  "components": {
    "app": {"status": "up"},
    "ollama": {"status": "up"},
    "rag": {"status": "down", "error": "Could not reach Pinecone"}
  }
}
```

4. GET /api/models – Available Models

What: List all Ollama LLM models installed on the system.

Why: Shows what models are available for chat. Lets ops team see what's loaded.

Request:

curl http://localhost:3000/api/models

Response:

```
{
  "models": [
    {
      "name": "llama3.2",
      "size": 8,
      "status": "loaded"
    },
    {
      "name": "mxbai-embed-large",
      "size": 2,
      "status": "loaded"
    },
    {
      "name": "mistral",
      "size": 7,
      "status": "available_for_download"
    }
  ],
  "current_model": "llama3.2"
}
```

5. GET /api/rag/status – RAG Configuration

What: Check if Pinecone RAG is enabled and get index stats.

Why: Verify that document retrieval is working and see how many documents are indexed.

Request:

```
curl http://localhost:3000/api/rag/status
```

Response (Enabled):

```
{
  "enabled": true,
  "provider": "pinecone",
  "index": "harry-clarke-docs",
  "vector_count": 1250,
  "namespace": "prod"
}
```

Response (Disabled):

```
{
  "enabled": false,
  "message": "RAG not enabled. Set ENABLE_RAG=true in .env.local"
}
```

E. Poster, Slides, and Video Links

1. Project Poster:



Knight Foundation School of Computer & Information Sciences

Spring 2026 Senior Design Project

RAG Chatbot for Single Object: Proof of Concept

(Harry Clarke Art Assistant)

Students: Jessica Orozco, Angel Feliciano, Samantha Lezama, John M. Espinal, Guilherme P. Cardoso

Mentor: Dr. Mark Osterman, The Wolfsonian FIU

Instructor/Faculty: Masoud Sadjadi, Florida International University

PROBLEM

The RAG Chatbot addresses a key challenge in museum interpretation:

How to deliver accurate, nuanced, and contextually grounded information about the Harry Clarke and the Geneva Window without requiring visitors to navigate dense academic texts or fragmented online resources.

CURRENT SYSTEM

Visitors rely on wall labels, tours, interactive digital tools, or independent research.

Curatorial files, conservation reports, and historical documents are not accessible in-gallery.

No conversational tools exist to help visitors explore the artwork's symbolism, context, or history.

Staff cannot individually guide every visitor.

REQUIREMENTS

Functional Requirements:

Retrieve relevant info from museum-approved documents, generate grounded answers, and provide a clean and real-time chat interface using approved sources (provided by The Wolfsonian FIU).

Non-Functional Requirements:

Fast responses (<3 seconds), Very low hallucination rate (<2%), High reliability (99% uptime), Accessible UI (WCAG 2.1 AA)

SYSTEM DESIGN

A three-tier RAG system:

Frontend (React): Chat interface, Panel viewer, Accessibility.

Backend (Django): RAG model, Retrieval (Pinecone), Generation (OhLlama).

Data Tier: Pinecone vector DB, OhLlama LLM API.

OBJECT DESIGN

The system focuses on a single artwork:

Harry Clarke and the Geneva Window

Design considerations:

Panel-level metadata tagging; iconographic themes; historical context (labor politics, nationalism, internationalism); curator-approved sources and written references.

IMPLEMENTATION

Key Components:

Ingestion Pipeline: Chunking (200 tokens), metadata extraction, embedding generation.

Hybrid Retrieval: Vector search + metadata filters.

Re-ranking: Cross-encoder scoring.

RAG Orchestration: Prompt assembly, grounding enforcement, Vectorized Search Confirmation.

Frontend: React chat UI and session tokens.

Backend Patterns: Service layer, adapter pattern for LLM/embeddings, repository pattern for storage.

VERIFICATION

Testing Strategy:

Unit tests: metadata, embeddings, retrieval.

Integration tests: full RAG pipeline.

E2E tests: chat sessions, panel navigation.

Performance tests: load testing, LLM latency profiling.

Security tests: sanitization, rate limiting, API key protection

SUMMARY

The RAG Chatbot demonstrates how Retrieval-Augmented Generation can enhance museum interpretation by delivering accurate, grounded, conversational explanations of a complex artwork.

The system successfully integrates a Pinecone Vector database, a Django-based RAG backend, and an accessible chat interface.

This proof of concept establishes a scalable foundation for expanding to additional artworks and future museum-wide conversational experiences.

REFERENCES

[1] Wolfsonian-FIU, Harry Clarke Window: Curatorial Research Files, Miami, FL.

[2] Wolfsonian-FIU, Conservation Reports: Harry Clarke Stained-Glass Window, Miami, FL.

[3] Wolfsonian-FIU, Exhibition Catalogs on Harry Clarke and Irish Modernism, Miami, FL.

[4] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," 2020.

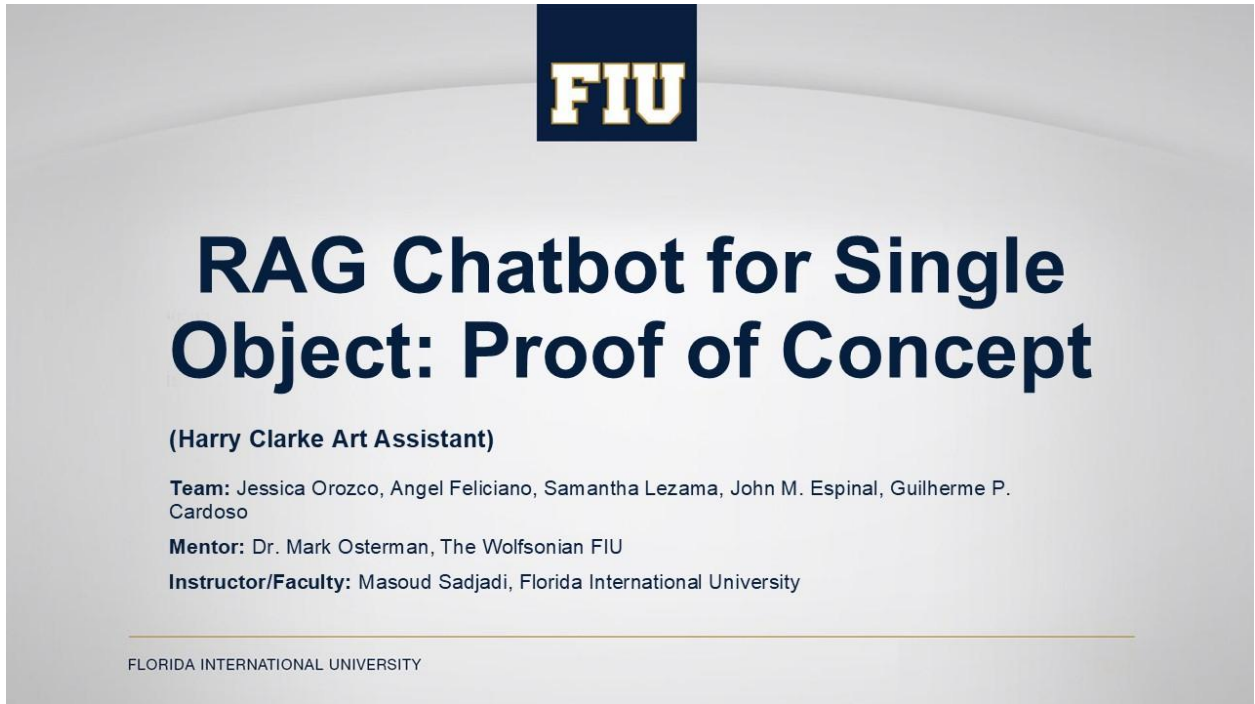
ACKNOWLEDGEMENT

The material presented in this poster is based upon the work supported by The Wolfsonian FIU. We thank The Wolfsonian FIU and Dr. Mark Osterman for their assistance and mentorship that we received throughout the senior design project.

FLORIDA INTERNATIONAL UNIVERSITY

48

2. Project Slides/PowerPoint:



Slide 1 features the FIU logo in a dark blue box at the top center. The title "RAG Chatbot for Single Object: Proof of Concept" is prominently displayed in a large, bold, dark blue font. Below the title, the subtitle "(Harry Clarke Art Assistant)" is in a smaller, bold, dark blue font. The slide lists the team members, mentor, and instructor/faculty in a standard dark blue font. A thin orange horizontal line is positioned above the FIU text at the bottom left.

FIU

RAG Chatbot for Single Object: Proof of Concept

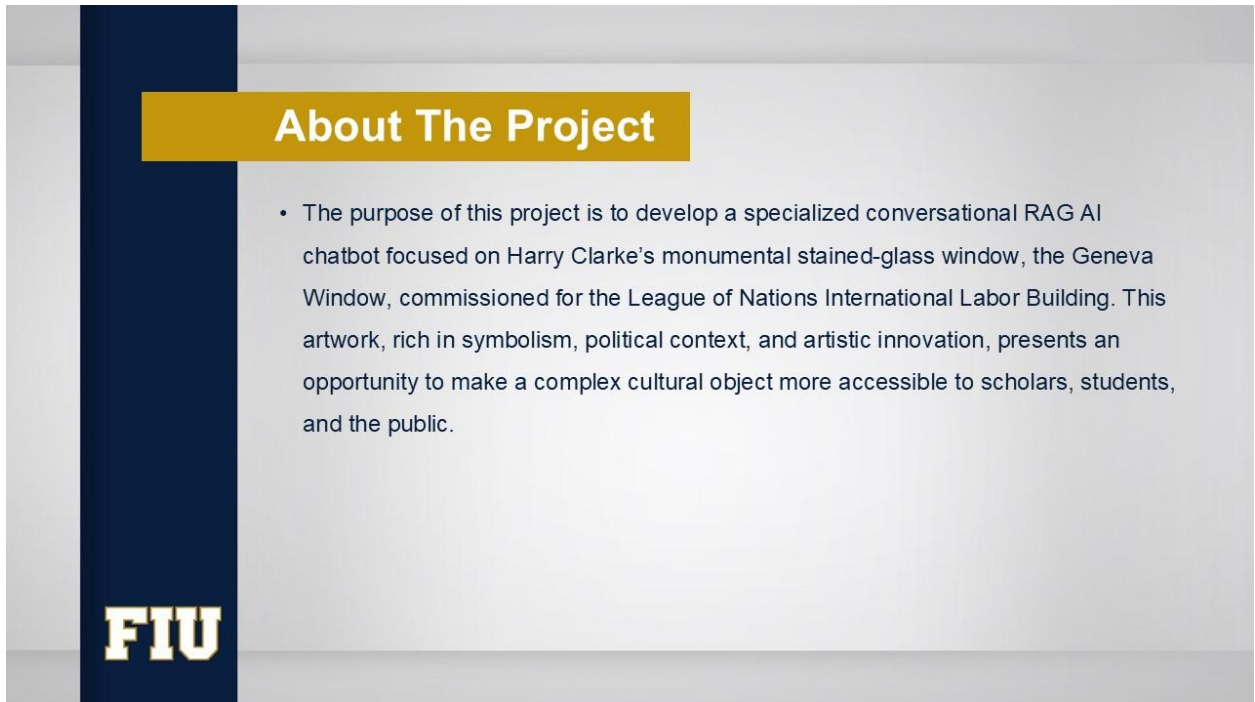
(Harry Clarke Art Assistant)

Team: Jessica Orozco, Angel Feliciano, Samantha Lezama, John M. Espinal, Guilherme P. Cardoso

Mentor: Dr. Mark Osterman, The Wolfsonian FIU

Instructor/Faculty: Masoud Sadjadi, Florida International University

FLORIDA INTERNATIONAL UNIVERSITY



Slide 2 has a dark blue vertical bar on the left side with the FIU logo at the bottom. A yellow rectangular box with the title "About The Project" is located on the left. To the right of this box is a bulleted list describing the project's purpose. The slide has a light gray background.

About The Project

- The purpose of this project is to develop a specialized conversational RAG AI chatbot focused on Harry Clarke's monumental stained-glass window, the Geneva Window, commissioned for the League of Nations International Labor Building. This artwork, rich in symbolism, political context, and artistic innovation, presents an opportunity to make a complex cultural object more accessible to scholars, students, and the public.

FIU

System Architecture

Three-Tier Architecture

Frontend (React)

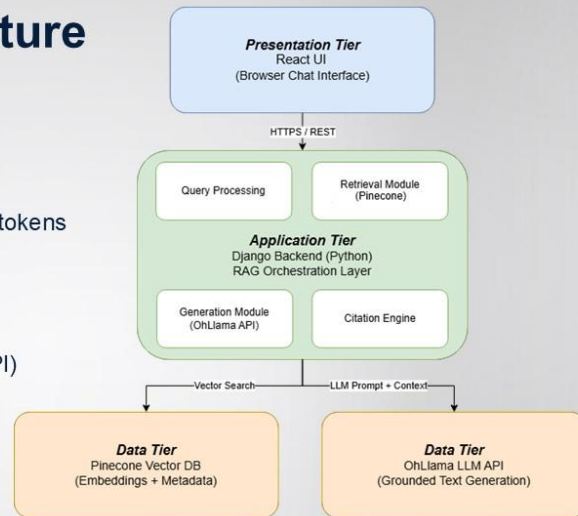
- Real-time chat interface
- Suggested prompts and session tokens

Backend (Django)

- RAG orchestration layer
- Retrieval module (Pinecone)
- Generation module (OhLlama API)

Data & External Services

- Pinecone Vector DB
- OhLlama LLM
- Storage



Architecture Diagram (C4 Container Level)

Implementation Highlights

- **Modular RAG Design:** Implemented the Service Layer Pattern in Django to keep the AI retrieval logic separate from the API views for better maintainability.
- **Adapter Pattern Integration:** Developed a flexible backend that can "hot-swap" between different LLM providers (like Ollama or OpenAI) without core code changes.
- **Hybrid Retrieval Engine:** Combined vector search with metadata filters to maximize the accuracy of technical document retrieval.
- **Efficient Vector Indexing:** Utilized Pinecone for high-speed, managed indexing of document embeddings to ensure low-latency response times.
- **Context Optimization:** Engineered a strategic chunking size (200 tokens) to strike a balance between context richness and retrieval precision.
- **Stateless Frontend Architecture:** Built a fast, responsive UI using a stateless React approach to minimize client-side overhead and ensure scalability.
- **Production-Ready Standards:** Adhered to strict coding conventions, including PEP8 for Python and ESLint for React, ensuring a clean and professional codebase.
- **Modular Repository Structure:** Organized the project into clear /frontend, /backend, and /rag directories for efficient team collaboration and deployment.

Demonstration Overview

What the Demo Shows

- User asks a question about Harry Clarke's stained-glass window.
- Hybrid retrieval pulls relevant text from Pinecone.
- Re-ranking selects the most relevant passages.
- LLM generates a grounded answer using only retrieved context.

Demo Goal:

Show how the chatbot transforms dense museum scholarship into accessible, accurate, conversational interpretation.

FIU

FIU

Testing and Evaluation Results

Severity	Issue Description	Root Cause	Resolution
Critical	Incorrect or unsupported factual claims in early responses.	LLM generating content not present in the retrieved chunks.	Strengthened system prompt, enforced strict grounding.
Critical	Retrieval returning irrelevant chunks for specific panel queries.	Incomplete chunk metadata and inconsistent panel numbering.	Re-processed documents with improved metadata and panel-level tagging.
Major	Slow response times (>5s) under heavy load.	Sequential LLM and retrieval calls blocking Django workers.	Sequential LLM and retrieval calls blocking Django workers.
Major	BM25 keyword search returning overly broad matches.	Default tokenizer was too permissive for technical terms.	Tuned BM25 parameters and added specific metadata filters.
Minor	Occasional session resets.	Frontend state not persisting across refresh.	Added a lightweight session token stored in memory.

Challenges and Lessons Learned

Technical Challenges

- Ensuring strict grounding to prevent hallucinations; Handling ambiguous or overly broad queries.
- Improving panel-level retrieval accuracy; Managing latency under load.
- Standardizing metadata across diverse documents.
- RAM limitations impacting image processing and database management.

Lessons Learned

- High-quality metadata is essential for reliable retrieval; Hybrid search (vector + metadata filters) significantly improves accuracy.
- Async Django views reduce bottlenecks in LLM calls; Curator feedback is critical for validating responses.
- Chunk size and overlap dramatically affect retrieval quality.

FIU

Future Work

Expand Content Scope

- Add 5–10 more artworks from the Wolfsonian collection; Enable cross-object comparisons and thematic tours.

Enhance RAG Pipeline

- Advanced re-ranking models; Improved multimodal retrieval (CLIP-based); Confidence scoring and uncertainty visualization.

Improve User Experience

- Personalization based on visitor interests; Educator modes for K-12 and university audiences.

Operational Improvements

- Curator-facing content management dashboard; Automated ingestion + embedding workflows; A/B testing for prompts and retrieval strategies.

Long-Term Vision:

Deploy in gallery via QR codes; Integrate with museum tours, audio guides, or AR; Scale to a museum-wide conversational platform.

FIU

Thank you

The Team:

- Jessica Orozco
- Angel Feliciano
- Samantha Lezama
- John M. Espinal
- Guilherme P. Cardoso



FIU FLORIDA
INTERNATIONAL
UNIVERSITY

3. Project Videos

Introduction Video – https://youtu.be/Ms2Lwmz_cko

User Guide / Demo Video – <https://youtu.be/76oSMxlbxeQ>

Install Maintenance / Installation Guide Video – <https://youtu.be/fYBpGGhKPBc>

Shortcomings Wishlist Video – https://youtu.be/Ir_kNIFgxaQ

F. Scrum Evidence Index

Daily Scrum Meeting Minutes

- https://docs.google.com/document/d/1o-tFNsV-qSi6usCcMx_fKS1IFbpx6OhZOWI_YQOP6Ns/edit?usp=sharing

Sprint Planning Meeting Minutes:

- <https://docs.google.com/document/d/1DH3yUw-W4F4KW6KCdQjvwf3WGwiRVDWdx2HPsHLUlqU/edit?usp=sharing>

Sprint Backlog Grooming Meeting Minutes:

- https://docs.google.com/document/d/1_FYMexak-Fuaqj9JkbIsmNFTvF4h8JCq8ryEwV4gc66w/edit?usp=sharing

Sprint Retrospective Meeting Minutes

- https://docs.google.com/document/d/1kkLnmYr1ZTWNtz47NO70f2_YLTLlNuDeYVWHyvURmo/edit?usp=sharing

Sprint Review Meeting Minutes

- https://docs.google.com/document/d/1qegdZfiDUY_DRt64VVQdUyY7uDgxByeqtRSA3gB0M8k/edit?usp=sharing