

The goal of this document is to share what information would be **MOST** useful for the development of our AI research project

Given

Resource: <https://www.learngrantwriting.org/blog/smart-way-to-use-ai-grant-writing/>

Resource: <https://codewave.com/insights/python-ai-machine-learning-libraries/>

Step 1: Base Framework

Includes core sections all grants require such as...

- Project overview
- Need statement
- Objectives
- Outcomes, and
- Budget summary

Adaptable for different grants

Step 2: Gather Grant-Specific Context

- Use AI to summarize grant guidelines and eligibility criteria
- Creates context for AI to work without building a fixed mold (easier to process different grants)

Technical Frameworks:

Pandas and Numpy:

- Used for data manipulation and numerical operations
- Crucial for cleaning and structuring grant data / analyzing past proposal performance

Sckit-learn:

- Used for predictive analysis
- Could create a model that predicts likelihood of success for a given grant opportunity based on organization's history

Beautiful Soup or Scrapy:

- Web scraping libraries are necessary for extracting data from online grant databases, foundation websites, or PDFs of RPFs

System components and workflow

A robust AI grant applier is more than just a writing bot; it's a multi-step workflow.

1. Grant research and opportunity analysis:

1. Scrape grant-funding databases and funder websites for new opportunities.
2. Use NLP tools (via Hugging Face) to analyze the scraped Request for Proposals (RFPs) and extract key requirements, deadlines, and evaluation criteria.
3. Employ a predictive model (built with Scikit-learn) to score each opportunity based on your organization's past success and the grant's requirements.

2. Context gathering and information retrieval:

1. Develop a vector database (using Pinecone or ChromaDB) populated with your organization's mission statement, project information, and previously submitted grants.
2. Allow users to upload project-specific documents through the web interface (built with Streamlit).


3. Proposal drafting and generation:

1. Use an orchestration framework (like LangChain) to manage the drafting process.
2. For each section of the grant, the system would retrieve relevant information from the vector database and generate a draft using an LLM (via OpenAI API).
3. Refine the content by matching the tone and style of previous successful proposals stored in the vector database.

4. Review, editing, and customization:

1. Use the web interface to present the AI-generated draft to the user for review.
2. The user can then provide feedback or make edits, ensuring the final document has a human touch.

5. Submission and tracking:

1. Automate the submission process using web automation tools like Selenium to fill out and submit online forms.
2. Create a system to track the application's status and report on outcomes. 

Fatemah:

Resource:

<https://med.stanford.edu/medicine/news/current-news/standard-news/ai-for-grant-writing.html>

Potential risks we need to account for:

- Plagiarism
- Monotonous writing across all grants
- Repetition/redundancies in “why” the grant is deserved

Ways we can avoid these issues:

- Ensemble learning
- Defining a structure for the AI model to adhere to
- Testing the AI in different applications to root out any clear, recurring issues

Some ideas to organize and clean data:

- Excel sheet

Victorya Devero:

FastAPI: <https://fastapi.tiangolo.com/>

- Built in request/response validation—easier for beginner level API integrations, fast, and works well with Python: <https://fastapi.tiangolo.com/>

HuggingFace: <https://huggingface.co/docs>

- can use for embeddings for grant similarity matching and organizational data retrieval; free inference for models and reduces API costs

LangChain: <https://python.langchain.com/docs/introduction/>

- Better for Cambio Labs document retrieval—direct API calls — more controlled text generation

LLM Considerations:

- OpenAI API: <https://openai.com/api/pricing/>
- DEEPSEEK R1: https://api-docs.deepseek.com/quick_start/pricing Not allowed usage, but overall the cheapest model per input token.
- Anthropic: <https://www.anthropic.com/pricing#api>
 - Claude Sonnet 4: Best for development/testing
 - Input \leq 200K tokens \$3/ MTok
 - Output \leq 200K tokens \$15/ MTok
 - Claude Haiku 3.5
 - Input : \$0.80/MTok
 - Output: \$4/MTok

Final Consideration: OpenAI API —> cheapest overall per input token.

For document processing:

- python-docx: Word document generation <https://python-docx.readthedocs.io/en/latest/>

Data Storage:

- PostgreSQL : <https://www.postgresql.org/docs/>

For the workflow, take our time processing one grant at a time, and validate it first per API call. Afterward, more robust grant validation before integrating AI.

Nuzhat:

Notes

- This methodology is an example workflow and is open for ideas or changes

Methodology

1. Set up environment
 - a. Import llama index
 - b. .env file for open ai api key
2. Import tools
 - a. DuckDuckGoSearchToolSpec - good for general web searches
 - b. Open ai - LLM
 - c. Function tool
 - d. Open ai - agent
 - e. Environment variables
 - f. guardrails
3. Initialize
 - a. Environment variables
 - b. Specific model from open ai for LLM
 - i. Set up evals to establish a performance baseline
 - ii. Meet accuracy target with best models available
 - iii. Optimize for cost and latency by downsizing models where possible
 - c. Duckduck go tool and adjust tool so ai agent will be able to use it
 - d. (optional) add research keyword function to add keywords that will aide the agent to find grants that cambio labs will be eligible for
 - i. Create research tool to make keyword function a tool for the agent
 - ii. Combine duckduck go tool and keyword function tool into a general tool variable
 - e. Create guardrail variable (ex. GuardrailsOutputParser(guard=guard))
 - i. Optional step (May need additional to tools for this)
4. Create agent
 - a. Use tools
 - b. LLM imported
 - c. verbose=True (see agent thinking)

- d. system_prompt(for guidelines)
 - e. Guardrail (optional)
- 5. Output
 - a. The prompt given
 - b. The agent response
- 6. Integration to notion as table (needs more research to actually use)
 - a. Use pydantic programs and output parsers to enable structured output
 - b. Create notion integration with read and write permissions to a specific database made in notion
 - c. Use package in llama index (llama-index-tools-notion)
 - d. Example workflow:
 - i. agent processing(agent finds grants)
 - ii. structured extraction (structured LLM and program extract grant urls and grant requirements)
 - iii. Notion tool (agent passes this structured data to the notion tool)
 - iv. Database creation (tool creates new rows in table and populates table properties)
- 7. Idea for filling out easy/ redundant question - open ai CUA ("computer-using agent")
 - a. use a multiple agent linear "swarm" pattern (simple)
 - i. First agent is research where the grants are found
 - ii. Second cua agent to fill out preliminary responses to the questions and responses are saved
 - iii. The forms are then saved on notion
 - b. Use llama agents framework (production deployments)
 - i. Control plane for orchestration
 - ii. Deploys multi-agent RAG systems for real-time interaction and monitoring

Tools

- Open AI Agent SDK.
- Open ai CUA
- Llama index
- Notion

Pros	Cons
<ul style="list-style-type: none"> ● Llama index is free ● Methodology mainly uses known and familiar tools 	<ul style="list-style-type: none"> ● Learning curve may be steep ● Still need to find a way for the agent to fill out the applications with agent

<ul style="list-style-type: none"> • Agent can list grants and their attributes such as completion status in table format in notion • Open ai api -> cheapest overall per input token according to victorya 	
--	--

Sources

<https://cdn.openai.com/business-guides-and-resources/a-practical-guide-to-building-agents.pdf>

<https://www.anthropic.com/engineering/building-effective-agents>

<https://openai.github.io/openai-agents-python/agents/>

https://docs.llamaindex.ai/en/stable/examples/agent/agent_workflow_research_assistant/

Finalized Decisions:

Base Framework:

- Llama Index: <https://www.llamaindex.ai/>
- Langchain [considering: preferably use Llama Index]
- HuggingFace

Technical Frameworks:

- Pandas: processing grants
- Numpy: numerical grant processing

Database:

- PostgreSQL

Finalized Research (Basir)

Local-First with Cloud Fallback

After researching both local and cloud-based AI implementations, I'm proposing a hybrid architecture that prioritizes learning while maintaining delivery flexibility. We'll start with local models to understand RAG fundamentals, but design our system to switch seamlessly to cloud APIs when we need better performance or hit development roadblocks.

Such an approach acknowledges that we're just students focused on learning and helping a nonprofit, not building enterprise software. Our timeline should front-load local experimentation through week 4, then pivot strategically based on what we discover.

Phase 1: Local Development Foundation (Weeks 1-4)

LlamaIndex + Ollama Configuration

LlamaIndex provides great flexibility for both local and cloud implementations, letting us choose based on project priorities rather than cost constraints.

Local Setup Documentation:

- LlamaIndex can be configured to use Ollama through custom LLM settings:
<https://docs.llamaindex.ai/en/stable/examples/llm/ollama/>
- Michael Ruminer's detailed tutorial shows exactly our setup: LlamaIndex + local HuggingFace embeddings + Ollama Llama 3.1 8B:
<https://m-ruminer.medium.com/ai-rag-with-llamaindex-local-embedding-and-ollama-llama-3-1-8b-b0620116a715>
- Working code example: `Settings.llm = Ollama(model="llama3.1")` and `Settings.embed_model = HuggingFaceEmbedding(model_name="BAAI/bge-base-en-v1.5")`
- DataCamp validation: <https://www.datacamp.com/tutorial/llama-3-1-rag> shows production-ready RAG with Ollama and local models

Free Local Embeddings

Zero-cost embedding options:

- Official LlamaIndex HuggingFace guide: <https://docs.llamaindex.ai/en/stable/examples/embeddings/huggingface/>
- BAAI/bge-small-en-v1.5 benchmarked at 23.32 embeddings/second on CPU hardware
- Alternative models: sentence-transformers/all-mpnet-base-v2 and nomic-embed-text
- OpenVINO optimization available for CPU performance gains: https://docs.llamaindex.ai/en/stable/module_guides/models/embeddings/

Proven Working Examples

Real implementations we can learn from:

- Complete GitHub implementation: <https://github.com/Otman404/local-rag-llamaindex> (FastAPI + LlamaIndex + Ollama + Qdrant)
- Medium tutorial with evidence: <https://medium.com/rahasak/build-rag-application-using-a-llm-running-on-local-computer-with-ollama-and-llamaindex-97703153db20>
- Official LlamaIndex local tutorial: https://docs.llamaindex.ai/en/stable/getting_started/starter_example_local/

Phase 2: Architecture Flexibility Design

Switchable Model Configuration

Building our system with **model-agnostic architecture** from day one. This means using LlamaIndex's settings system to make LLM and embedding providers configurable:

```
python
# Local configuration
Settings.llm = Ollama(model="llama3.1")
Settings.embed_model = HuggingFaceEmbedding(model_name="BAAI/bge-base-en-v1.5")

# Cloud configuration (when needed)
Settings.llm = OpenAI(model="gpt-4o-mini")
Settings.embed_model = OpenAIEmbedding()
```

Key insight from hybrid systems research:

<https://towardsdatascience.com/building-production-ready-rag-applications-with-llamaindex-8b064c0cf5d7> shows that successful RAG systems often combine local and cloud components strategically, using local models for development and cloud models for production demos.

Cost-Aware Implementation

Local approach eliminates API costs entirely:

- No OpenAI, Anthropic, or external API dependencies
- Process unlimited documents using only electricity
- Perfect for FERPA compliance since data never leaves local machines

Cloud fallback pricing analysis:

- Current pricing: GPT-4o Mini at \$0.15 per million input tokens, \$0.60 per million output tokens (<https://llmpricecheck.com/openai/gpt-4o-mini/>)
- Costs depend on actual usage patterns, document sizes, and testing frequency, but token-based pricing makes small-scale projects like ours affordable
- Faster development cycle, higher quality outputs, immediate deployment capability

Phase 3: Practical Implementation Stack

Local-First Technical Components

Core infrastructure (all free):

- **SQLite** for development database (no server setup required)
- **Streamlit** for rapid prototyping UI
- **Chroma or FAISS** for local vector storage
- **HuggingFace sentence-transformers** for embeddings
- **Ollama** running Llama 3.1 8B for text generation

Hardware Reality Check

System requirements:

- Llama 3.1 8B needs 16GB+ RAM for smooth operation
- Will run on most modern laptops but may be slow

OTHERWISE, remember:

- Cloud deployment options ready as backup where OpenAPI & LangChain could be used

Learning vs. Delivery Balance

Why Start Local (Educational Benefits)

1. **Deep understanding:** Learn how RAG actually works rather than just calling APIs

2. **No dependency anxiety:** Experiment freely without per-token costs
3. **Data privacy:** Cambio's documents never leave our machines
4. **Debugging skills:** Understand the full pipeline when things break

Why Keep Cloud Option (Practical Benefits)

1. **Quality assurance:** Better results for final demos and presentations
2. **Time management:** Pivot if local setup consumes too much development time
3. **Deployment flexibility:** Cloud APIs work better for team collaboration
4. **Professional experience:** Learn industry-standard integration patterns

Timeline

Weeks 1-2: Local Foundation (makes sense primarily for BTT final presentation)

- Ollama installation and model testing across team
- Basic LlamaIndex + local embeddings pipeline
- Simple document ingestion from Google Drive

Weeks 3-4: Evaluation Period

- Test local system with real Cambio documents
- Measure quality, speed, and reliability
- Make go/no-go decision on continuing local-only

Weeks 5-8: Enhanced Development

- Continue with chosen approach (local or hybrid)
- Add grant discovery and application generation
- Build switchable architecture regardless of primary approach

Weeks 9-12: Production Polish

- Deploy best-performing version for Cambio demos
- Document both approaches (local & cloud) for future development
- Prepare technical presentation showing learning outcomes