Building an Enterprise API Integration Assistant: Complete Technical Blueprint

An individual developer can build a production-ready AI-powered API assistant for **under $50/month** using open-source tools, with a clear path from MVP to scale. This comprehensive guide synthesizes the best approaches across architecture, LLMs, RAG, UI, and deployment for maximum cost-effectiveness.

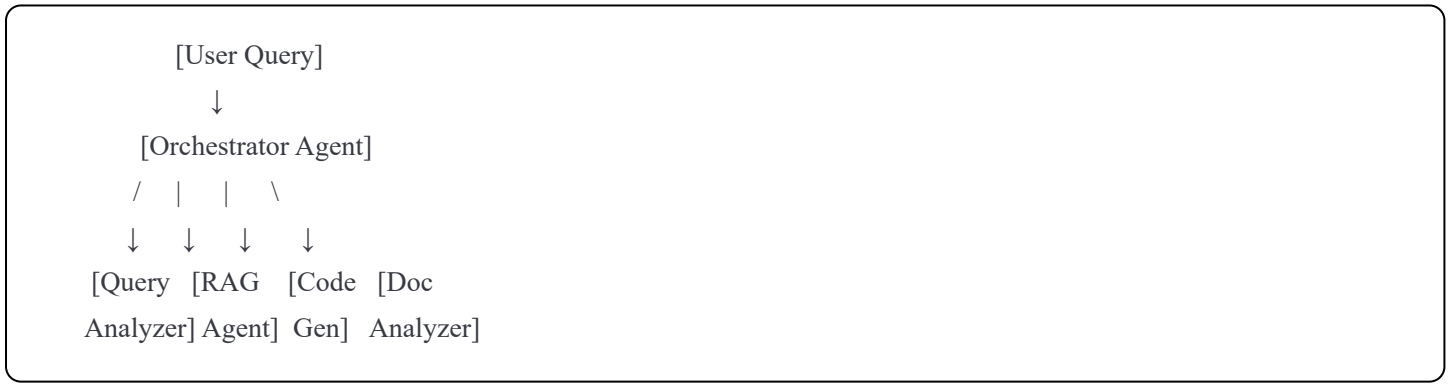
The optimal stack: LangGraph + Ollama + ChromaDB + Streamlit

The recommended architecture combines **LangGraph** for agent orchestration (production-ready, excellent RAG integration), **DeepSeek-Coder-6.7B via Ollama** for free local inference, **ChromaDB** for vector storage, and **Streamlit** for rapid UI development. This zero-to-low-cost stack handles hundreds of API documents while providing a clear upgrade path. Start with a **modular monolith** architecture—simpler deployment, faster iteration, lower operational overhead—then decompose into services only when specific components need independent scaling.

Architecture patterns that actually work

Supervisor/worker agent pattern

The most practical multi-agent architecture for an API assistant uses a central orchestrator delegating to specialized agents:



The **Query Analyzer** classifies intent (explain, generate_code, debug), the **RAG Agent** retrieves relevant documentation with source citations, the **Code Generator** produces integration code using templates plus LLM, and the **Doc Analyzer** parses OpenAPI specs and extracts endpoint metadata. This separation enables targeted improvements to each capability without destabilizing others.

Framework recommendation: LangGraph wins

Framework	Learning Curve	RAG Integration	Production-Ready	Best For
LangGraph	Medium-High	⭐⭐⭐⭐⭐	Yes	Your MVP—complex workflows with cycles

Framework	Learning Curve	RAG Integration	Production-Ready	Best For
CrewAI	Low	⭐⭐⭐	Prototype	Quick demos, role-based teams
AutoGen	High	⭐⭐⭐	Developing	Conversation-driven agents

LangGraph provides native state management with checkpointing, graph-based workflows ideal for cyclical RAG retrieval, and seamless LangChain ecosystem integration. (Zapier) CrewAI's lower learning curve is tempting, but LangGraph's flexibility prevents painful framework migrations later.

MCP (Model Context Protocol) for tool integration

Anthropic's Model Context Protocol (November 2024) standardizes how AI applications connect to external tools—think "USB-C for AI." (Red Hat) Pre-built MCP servers exist for GitHub, PostgreSQL, filesystem operations, and web search. Adopting MCP from day one future-proofs your architecture and reduces custom connector maintenance.

RAG implementation: chunking matters more than model choice

Retrieval strategy progression

Phase 1 (MVP): Basic vector similarity with ChromaDB and (all-MiniLM-L6-v2) embeddings (**384 dimensions, ~80MB**). Sufficient for hundreds of documents with ~150MB total storage.

Phase 2: Add BM25 keyword search via LangChain's (EnsembleRetriever) for **hybrid search**, improving exact API endpoint name matching by **15-30%**. Implement re-ranking with (BAAI/bge-reranker-base) (free, ~300MB) for significant precision gains.

Phase 3: Deploy **Corrective RAG (CRAG)** pattern—evaluate retrieval quality before generation, transform queries if results are poor, fall back to web search when needed.

Chunking strategies for API documentation

Content Type	Recommended Approach	Chunk Size
OpenAPI specs	Per-endpoint with linked schemas	Variable
Prose documentation	Recursive character splitting	500-1000 tokens
Code examples	Keep functions intact	Full function
JSON schemas	Semantic by object	Include parent refs

Critical insight: Poor chunks defeat excellent embedding models. Structure-aware chunking for OpenAPI specs—extracting endpoints, parameters, response schemas as enriched metadata—dramatically outperforms

naive text splitting.

```python
# Endpoint metadata enrichment
chunk_metadata = {
    "endpoint": "/users/{id}",
    "method": "GET",
    "tags": ["users"],
    "parameters": ["id: string (path)"],
    "response_schema": "User",
    "source_file": "openapi.yaml"
}
```

Vector database selection

Database	Setup	Best For	Limitations
ChromaDB	`pip install chromadb`	MVP, <100K docs	No hybrid search
Qdrant	Docker or embedded	Production, complex filtering	More setup
pgvector	PostgreSQL extension	Teams using PostgreSQL	~10M vector ceiling
FAISS	Library only	Research, max performance	No persistence/filtering

Start with ChromaDB (literally one line to install), migrate to **Qdrant** when needing hybrid search or exceeding 100K documents.

LLM strategy: free first, scale smart

Local inference with Ollama (zero cost)

DeepSeek-Coder-6.7B outperforms CodeLlama-13B for Python while requiring only **4-5GB VRAM** with Q4 quantization. Setup is trivial:

```bash
curl -fsSL https://ollama.com/install.sh | sh
ollama pull deepseek-coder:6.7b
```

Hardware requirements:

- **16GB RAM laptop:** DeepSeek-Coder-6.7B-Q4 (~9-12 tok/s)

- **32GB RAM + RTX 3060 12GB:** DeepSeek-Coder-33B-Q4 (~6-8 tok/s)
- **RTX 3090/4090:** Full precision models

Free API tiers worth using

Provider	Best Model	Daily Limit	Speed	Use Case
Groq	Llama 3.3 70B	100K tokens	⚡⚡⚡ Blazing	Backup for local
Google AI Studio	Gemini 2.5 Pro	100 requests	⚡⚡	Long API docs (1M context)
Together AI	DeepSeek-Coder	~$5 credits	⚡⚡	Code-specific tasks

Cost comparison at scale

Usage	Free Tier	Hybrid (Local + API)	Full Cloud
Hobby (50 req/day)	$0	$0	~$5/mo
MVP (200 req/day)	$0*	$10-20/mo	$30-50/mo
Growth (1K req/day)	N/A	$50-100/mo	$150-300/mo

*With local model + free API tier combination

Caching reduces costs 50-80%

Implement **semantic caching** with GPTCache to serve similar queries from cache:

```python
from gptcache import Cache
from gptcache.adapter.api import init_similar_cache

cache = Cache()
init_similar_cache(cache_obj=cache, data_dir="cache_dir")
# "Write API call" and "Create API request" hit same cache
```

Document storage and processing

SQLite for MVP, PostgreSQL for growth

Phase 1: SQLite with `aiosqlite` for async support—zero setup, file-based, sufficient for hundreds of documents. Use FTS5 for full-text search.

Phase 2: Migrate to PostgreSQL + pgvector when needing concurrent writes, team collaboration, or exceeding ~100GB.

```sql
-- Core schema for API documentation
CREATE TABLE endpoints (
    id INTEGER PRIMARY KEY,
    doc_id INTEGER REFERENCES documents(id),
    path TEXT NOT NULL,
    method TEXT NOT NULL,
    description TEXT,
    parameters JSON,
    completeness_score REAL,
    UNIQUE(doc_id, path, method)
);

-- FTS5 for text search
CREATE VIRTUAL TABLE chunks_fts USING fts5(content);
```

Parsing different API formats

Format	Parser	Notes
OpenAPI/Swagger	`prance.ResolvingParser`	Resolves $ref automatically
GraphQL	`graphql-core`	`build_schema()` for SDL
Postman	`postmanparser`	Collection → requests extraction
General docs	`unstructured`	25+ formats, LLM-optimized

Document processing pipeline:

1. Format detection by extension/content
2. Specialized parser selection
3. Endpoint/schema extraction
4. Chunking with metadata enrichment
5. Embedding generation (batch, normalized)
6. Storage with content hashing for incremental updates

Streamlit UI implementation

Chat interface with streaming

```python
import streamlit as st

st.title("API Integration Assistant")

if "messages" not in st.session_state:
    st.session_state.messages = []

for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

if prompt := st.chat_input("Ask about any API..."):
    st.session_state.messages.append({"role": "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    with st.chat_message("assistant"):
        response = st.write_stream(generate_response(prompt))  # Streaming!
    st.session_state.messages.append({"role": "assistant", "content": response})
```

Essential UI components

Sidebar configuration:

```python
with st.sidebar:
    model = st.selectbox("Model", ["gpt-4", "deepseek-coder", "local"])
    temperature = st.slider("Temperature", 0.0, 2.0, 0.7)

    with st.expander("Advanced"):
        max_tokens = st.number_input("Max Tokens", 100, 4096, 1024)
```

File upload with processing:

```python
```

```
uploaded_files = st.file_uploader(
    "Upload API specs",
    accept_multiple_files=True,
    type=["json", "yaml", "yml", "py", "md"]
)
```

Code display with syntax highlighting:

```
python

st.code(generated_code, language="python", line_numbers=True)
```

Alternatives if Streamlit limits you

Framework	Best For	Dev Speed
Chainlit	Chat-only apps	⭐⭐⭐⭐⭐
Gradio	Quick ML demos	⭐⭐⭐⭐⭐
React/Next.js	Full production apps	⭐⭐

Chainlit offers superior chat UX (built-in chain-of-thought visualization) if your app is purely conversational.

Deployment path: local to cloud

Docker Compose for local development

```
yaml
```

```yaml
services:
  app:
    build: .
    ports: ["8501:8501"]
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - VECTOR_DB_HOST=qdrant
    depends_on: [qdrant]

  qdrant:
    image: qdrant/qdrant:latest
    ports: ["6333:6333"]
    volumes: [qdrant_storage:/qdrant/storage]

  ollama:  # Optional local LLM
    image: ollama/ollama:latest
    ports: ["11434:11434"]
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              capabilities: [gpu]
```

Cloud hosting comparison

Platform	Free Tier	Starting Price	GPU	Recommendation
HuggingFace Spaces	Yes + ZeroGPU	$9/mo PRO	Free H200 shared	**Best for AI apps**
Railway	None	$5/mo + usage	No	Quick MVPs
Render	750hrs (sleeps)	$7/mo	No	Prototypes
Fly.io	None	~$5/mo	Limited	Global edge

Hidden costs to watch:

- **Egress:** Render charges $30/100GB after free tier; Fly.io only $0.02/GB Judoscale
- **Database:** Often 2-3x compute costs
- **Build minutes:** Some platforms charge for CI time

Recommended progression

1. **Weeks 1-4:** Local Docker with Ollama + ChromaDB
2. **Month 1-2:** HuggingFace Spaces (free GPU) or Railway ($5-20/mo)
3. **Month 3-6:** Render Professional or Fly.io ($30-80/mo)
4. **6+ months:** Evaluate AWS/GCP when exceeding $150-200/mo on PaaS

Error handling and monitoring

Retry with exponential backoff

```python
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(
    stop=stop_after_attempt(5),
    wait=wait_exponential(multiplier=1, min=1, max=60)
)
async def call_llm(prompt: str) -> str:
    return await client.chat.completions.create(...)
```

Circuit breaker for graceful degradation

```python
from pybreaker import CircuitBreaker

llm_circuit = CircuitBreaker(fail_max=5, reset_timeout=120)

@llm_circuit
def get_response(prompt):
    return call_llm_api(prompt)

# Fallback when circuit opens
def with_fallback(prompt):
    try:
        return get_response(prompt)
    except CircuitBreaker.Error:
        return "I'm temporarily unavailable. Please try again."
```

Free monitoring stack

- **Langfuse** (self-hosted): Full LLM observability, prompt management, evaluations
- **Prometheus + Grafana**: Metrics collection and dashboards
- **structlog**: Structured JSON logging for LLM requests

Key metrics to track: Token usage, response latency (P50/P95/P99), error rates by type, requests per user, and cache hit rates.

Advanced features roadmap

Phase 2 quick wins (1-2 weeks each)

Feature	Tool	Complexity	Value
Mermaid diagrams	streamlit-mermaid	LOW	HIGH
CLI tool	Typer	LOW	HIGH
Mock server generation	Prism	LOW	HIGH
Basic OCR	Tesseract	LOW	MEDIUM

Phase 3 enhancements (2-4 weeks each)

Feature	Tool	Notes
Code analysis	Tree-sitter	40+ languages, incremental parsing
Auto documentation	LLM + Sphinx	Generate from code structure
Test generation	OpenAPI → pytest	Schema-based test cases

Defer to Phase 4+

- **Voice interface:** Whisper is straightforward, but UX design for voice is challenging for code-heavy workflows
- **Multi-user collaboration:** Requires significant infrastructure investment
- **VS Code extension:** Medium-high complexity, build after core product validated

---

Implementation timeline

Week 1-2: RAG foundation

- ChromaDB setup with `all-MiniLM-L6-v2` embeddings
- OpenAPI parser with endpoint extraction
- Basic retrieval Q&A in Streamlit

Week 3-4: Agent layer

- LangGraph supervisor with query routing
- Code generation agent with templates
- Langfuse monitoring integration

Week 5-6: Production hardening

- Docker Compose with Ollama + Qdrant
- Error handling with circuit breakers
- Deploy to HuggingFace Spaces or Railway

Month 2+: Iterate based on usage

- Add hybrid search when precision issues arise
- Upgrade to `bge-base-en-v1.5` embeddings
- Implement semantic caching

Estimated monthly costs

Phase	Compute	LLM	Storage	Monitoring	Total
MVP (Local)	$0	$0 (Ollama)	$0	$0	**$0**
MVP (Cloud)	$5-20	$20-50	$0-10	$0	**$25-80**
Growth	$30-50	$50-100	$15-25	$0-20	**$95-195**

The path from zero-cost local development to production-ready cloud deployment is well-defined. Start with the free stack, validate with users, then invest in infrastructure as usage grows. The modular architecture ensures any component can be upgraded independently without rebuilding the entire system.