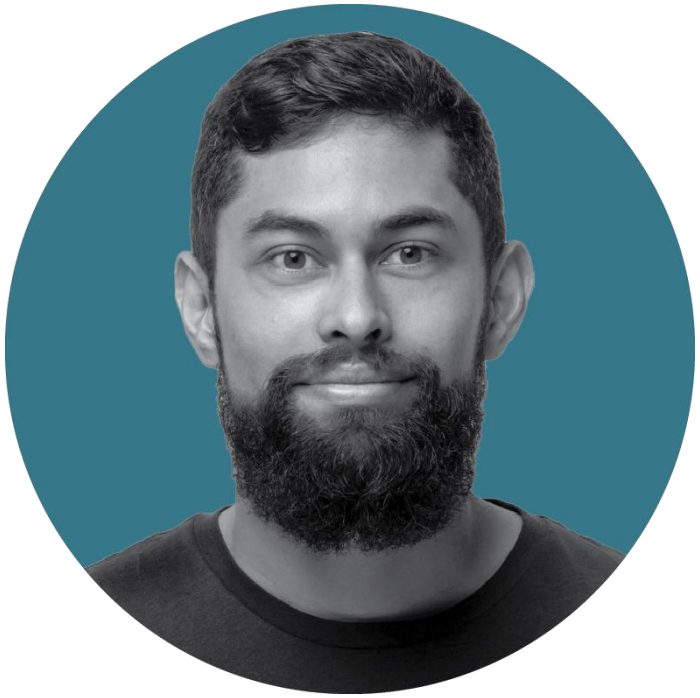


Creating a Context Aware AI Assistant in APEX

using a Private LLM

https://emacodes.dev/

Emanuel Cifuentes

Solutions Architect

VISCOSITY NORTH AMERICA

Session Goals

- Core concepts
- How to
  - Choose a model
  - Run your model in your own computer
  - Integrate with your data using Oracle APEX
- Getting ready to deploy a model in Production

Core Concepts

LLM

A **Large Language Model (LLM)** is a type of artificial intelligence system trained on massive amounts of text to understand and generate human language. At its core, it's a statistical model, usually built with transformer neural networks, that predicts the next word in a sequence based on context.

Because of the scale of its training data and parameters (often billions or more), an LLM can perform a wide range of tasks such as summarizing documents, answering questions, generating code, or reasoning about problems (debatable), even if it hasn't been explicitly programmed for them.

Retrieval-Augmented Generation (RAG)

An AI technique that combines a **language model** with an external knowledge source to improve accuracy and relevance. Instead of relying only on what the model learned during training, **RAG** retrieves relevant documents or data (for example, from a database or a search index) at query time and feeds that information into the model's prompt.

The model then uses both its internal knowledge and the retrieved context to generate the response. This approach is widely used to keep answers up-to-date, reduce hallucinations, and handle domain-specific questions without retraining the model.

OpenAI Compatible API

An **“OpenAI-compatible API”** means the server exposes the same REST endpoints, request/response formats, and headers as the official OpenAI API. For example /v1/chat/completions, /v1/completions, /v1/embeddings, etc.

Because the interface is the same, you can use standard OpenAI client libraries (like LangChain and other SDKs) and just point them to your server’s base URL and API key. This allows you to swap between OpenAI’s hosted models and your own locally-hosted models (e.g., vLLM, Ollama, llama-cpp) without changing your application code.

Parameters

In an **LLM (Large Language Model)**, **parameters** are the numerical weights (values in matrices and tensors) that the neural network learns during training.

They define how input tokens (words, sub-words) are transformed through the network's layers to produce the next token's probability.

In simpler terms: parameters are the "long-term memory" of what the model has learned from its training data. Billions of tiny knobs that collectively encode grammar, facts, reasoning patterns, and style.

Quantization

A technique used to reduce the size and computational cost of a machine-learning model by storing its numerical weights with lower-precision numbers.

For example, an LLM might be trained with 16 or 32-bit floating-point weights (FP16/FP32). Quantization converts them to smaller data types such as 8-bit integers (INT8) or even 4-bit formats (Q4, Q5, etc.).

This reduces memory footprint and bandwidth, allowing the model to run faster and fit on devices with limited RAM or GPU VRAM. The trade-off is usually a small loss in accuracy, though modern quantization methods are designed to minimize this impact.

Why the number of parameters matters

Capacity:

More parameters generally give the model more representational power. This often improves its ability to understand complex prompts, handle long-range context, and generalize across diverse tasks.

Resource cost:

Large parameter counts require more memory (RAM/VRAM) and compute power for both **training** and **inference**. A 7B-parameter model can often run on a single modern GPU or even CPU (with quantization). A 70B-parameter model may need multiple high-end GPUs to load and serve.

Latency and energy:

Bigger models take longer and use more energy per request. Quantization and optimized runtimes reduce this but don’t eliminate the trend.

Performance trade-off:

Past a certain point, increasing size yields diminishing returns for many use-cases; smaller fine-tuned or specialized models can outperform a huge general-purpose one for narrow tasks.

Resources Required

Small models (1B – 7B parameters)

Full-precision (FP16): 2–14 GB

Quantized (INT8 / 4-bit GGUF): 1–6 GB

Easily fits in a modern laptop with ≥16 GB RAM or a single consumer GPU (e.g., RTX 3060/4060). Even mobile devices.

Medium models (13B – 30B parameters)

Full-precision (FP16): 26–60 GB

Quantized (4-bit): 12–24 GB

Usually needs a high-end single GPU (24–48 GB VRAM) or 2 lower-VRAM GPUs, or a workstation/server with plenty of CPU RAM if running on CPU.

Large models (65B – 70B+ parameters)

Full-precision (FP16): 130–140 GB

Quantized (4-bit): 35–70 GB

Typically served on multi-GPU setups (e.g., 2× 80 GB A100/H100) or large-memory CPU servers; not practical for ordinary desktops.

File Formats

GGUF (GPT-Generated Unified Format)

A lightweight, binary format designed for llama.cpp and other CPU/edge-oriented runtimes.

It stores weights in quantized form (e.g., Q4_K_M, Q5_K) for small memory footprints and fast inference on CPUs or modest GPUs.

MLX

Apple's Machine Learning eXchange format used with the MLX framework for Apple Silicon (M-series) devices.

It optimizes models for Metal acceleration and low-precision math on macOS/iOS, making it ideal for local LLM inference on Apple hardware.

Safetensors

A popular, framework-agnostic tensor checkpoint format (used by Hugging Face Transformers, vLLM, etc.).

It's designed for speed and safety (no code execution on load), and usually stores full-precision weights (FP16/FP32) for server/GPU deployments.

.pth / .pt

The legacy PyTorch checkpoint formats.

They serialize both model weights and sometimes Python code; still widely found for research and older models, but gradually replaced by safetensors.

Choosing a model

Choosing a model

Intended Task

Use case: Chat/Q&A, code generation, summarization, RAG, reasoning, multimodal, etc.

Domain specialization: General-purpose vs. domain-tuned (finance, legal, medical).

Language and modality: Human languages, programming languages, or multimodal (text-image).

Model Size & Resource Needs

Parameters vs. hardware: Check whether it fits in your available GPU/CPU RAM.

Latency requirements: Smaller models respond faster.

Energy and cost: Bigger models are more expensive to host.

Performance vs. Efficiency Trade-off

Quality/accuracy: Often improves with size.

Quantization support: Can reduce memory and compute needs with minimal quality loss.

Benchmarking: Look at task-specific scores (MMLU, GSM-8K, etc.) instead of just size.

Choosing a model

Licensing & Cost

License terms: Open-source (e.g., Llama 3, Mistral) vs. restrictive/commercial.

Deployment cost: Cloud API pay-per-token vs. self-hosted hardware and energy.

Redistribution: If you build a product, confirm you can legally ship the model.

Deployment Environment

Target hardware: Cloud GPUs, on-prem GPUs, edge devices, Apple Silicon.

Runtime support: Frameworks (vLLM, llama.cpp, Ollama, PyTorch) and model formats (GGUF, MLX, safetensors).

Scaling needs: Multi-GPU, distributed inference, autoscaling.

Context Window & Features

Context length: Long-context models (e.g., 32k–200k tokens) for RAG or document QA.

Tool-use / function-calling: If you need structured outputs or external tool invocation.

Multilingual or multimodal support: If your use-case needs it.

Choosing a model

Ecosystem & Community

Availability of fine-tunes or adapters (LoRA, PEFT).

Community and vendor support.

Integration libraries: LangChain, LlamaIndex, RAG tooling, embeddings.

Loading a model

Means transferring a trained machine-learning model's **stored weights and configuration** from disk into **memory (RAM/VRAM)** so that a runtime or inference engine (like **PyTorch, vLLM, llama.cpp**, etc.) can use it to process inputs.

For an LLM, this typically involves:

- Reading the model file (e.g., GGUF, safetensors, MLX).
- Allocating memory on CPU or GPU to hold billions of parameters.
- Converting the stored weights into the runtime's tensor format.
- Initializing any supporting data structures (tokenizer, KV-cache, quantization kernels, etc.).

Live Demo

Creating an AI Assistant

- Pick a tool (ollama, LM Studio, Clara, Jan)
- Pick and load a model
- Enable the OpenAI compatible REST interface
- Make it available to the world (ngrok)

Creating an AI Assistant

- Shared Components -> AI Services
- Shared Components -> AI Configurations
- Create a Page with an empty region with a static ID
- Dynamic action: On Page Load -> Show AI Assistant

Getting ready for production

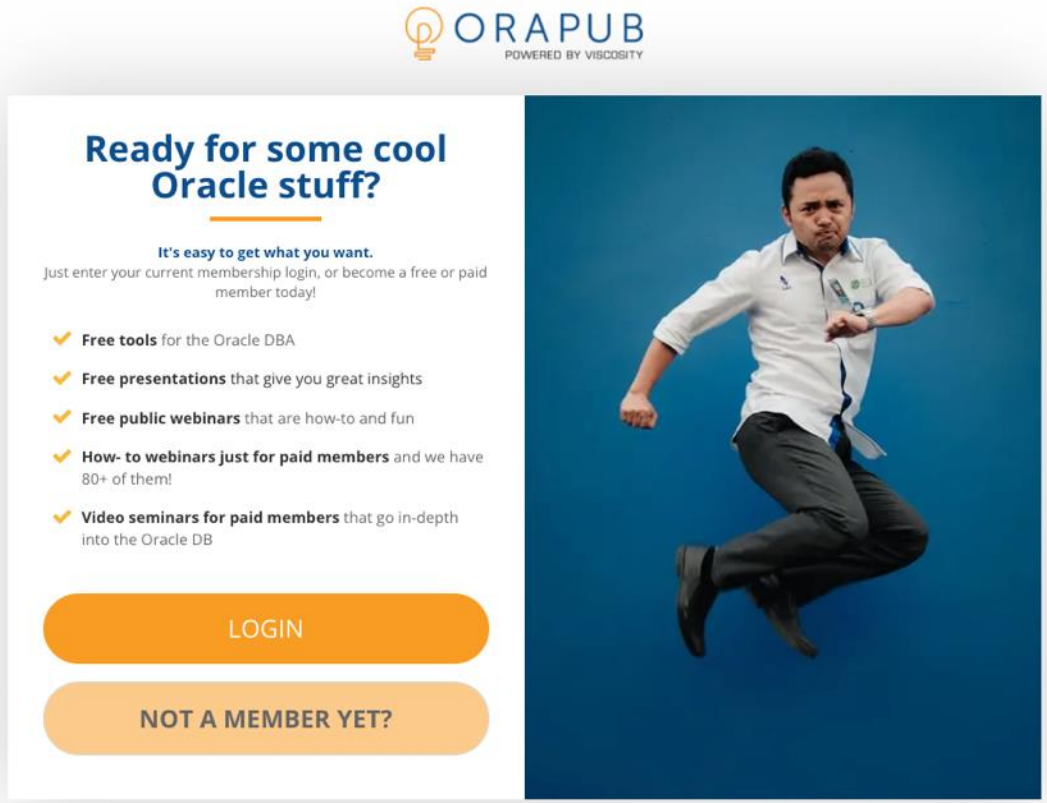
- Find good hardware
- Get familiar with python and vLLM
- Choose the right model
- Learn about the MCP server built into sqlcl

Any Questions?

✉ emanuel.cifuentes@ViscosityNA.com

VISCOSITY
NORTH AMERICA

All Viscosity hosted webinar recordings and conference sessions slide decks will be posted to www.OraPub.com for free and paid members!



Follow Us Online!

- Facebook.com/ViscosityNA
- LinkedIn.com/company/Viscosity-North-America
- @ViscosityNA
- Viscosity North America
- @Viscosity_NA