

# System Architecture & Implementation Guide

Project: Virtual Mechanical Engineer | Target Hardware: Dell GB10 / NVIDIA DGX Spark (128GB)

## ATTN AI CODING AGENT (openclaw / speckit / Claude):

This document serves as the foundational architectural specification for building a multi-agent, local-first mechanical engineering API appliance. Read the constraints and "Why" clauses carefully before generating the monorepo structure.

## 1. The Modular Monorepo Organization

To support multiple LLM frameworks (Hermes, openclaw) and a constantly expanding set of engineering tools while avoiding spaghetti code, the repository must use a strict modular structure. We will use a modern Python workspace manager (like uv or Poetry).

```
virtual-engineer-os/  
├── apps/  
│   ├── api/           # FastAPI application (Zero business logic, routing only)  
│   └── web/           # Frontend UI  
├── packages/  
│   ├── core/         # Shared domain models, structured JSON logging  
│   ├── agent_adapters/ # Interfaces for Hermes, openclaw, PI  
│   ├── tool_registry/ # Dynamic loading of engineering tools (BaseTool pattern)  
│   └── data_vault/    # RAG connectors, SQLite clients, filesystem managers  
├── docker-compose.yml  
└── pyproject.toml
```

### Why this structure?

- **Separation of Concerns:** The API should not know how Hermes manages its context window. The API only receives a request and invokes the appropriate adapter in `agent_adapters`.
- **Pluggable Tooling:** As new open-source libraries are found on GitHub, they can be wrapped in the `tool_registry` without touching the core routing or agent logic.

## 2. Data & Memory Architecture (100% Embedded)

---

The system must run completely offline without relying on heavy background database servers. Do not use containerized PostgreSQL or Qdrant.

- **State & Agent Memory (SQLite):** Use local SQLite for relational data mapping (Agent Sessions → Sub-tasks → Tool Invocations). **Why:** Requires zero background processes, handles concurrent local connections via WAL mode, and allows the node to remain highly portable.
- **Engineering RAG (LanceDB):** Use embedded LanceDB for vector search (material spec sheets, hardware catalogs). **Why:** It runs entirely in-process using Apache Arrow, providing blistering semantic search for local tooling without a separate microservice.
- **Physical Artifact Vault (File System):** Store all generated STEP, STL, and G-code files directly on the local NVMe drive (e.g., /opt/virtual-engineer/vault/). **Why:** Large CAD files do not belong in a database. If local storage fills up, this directory structure can easily be mounted to a local NAS (like an Unraid array on a Lenovo H430).

## 3. Docker Strategy: Thick Base / Thin Code

---

To enable rapid iteration by coding agents without waiting for slow container rebuilds, implement a hybrid volume-mount architecture.

### Why this strategy?

Engineering dependencies (vLLM, PyTorch, FreeCAD, CalculiX, CadQuery) take massive amounts of time to compile. If you use a standard Docker pipeline, every Python script change triggers a rebuild. By building a "Thick Base" image once and mapping the /apps and /packages directories via volumes in docker-compose.yml, coding agents can modify Python tool wrappers and see the changes execute instantly inside the container.

## 4. The Offline-Fallback Tooling Pattern

---

The GB10 node will be online most of the time but must function seamlessly if disconnected. All tools interacting with external data must follow the **Template Method Pattern** to enable a write-through cache.

```
class BaseTool(ABC):
    def execute(self, **kwargs):
        if self.is_online():
            try:
                # 1. Fetch fresh data from API
                # 2. Write data to local LanceDB / SQLite vault
                return self._run_online(**kwargs)
            except Exception:
                pass

        # Fallback to local Data Vault
        return self._run_offline(**kwargs)
```

## Why this pattern?

- **Agent Agnosticism:** The LLM (Qwen via vLLM) never has to write logic to handle HTTP 500 errors or check network connectivity. It just receives the engineering data.
- **Organic Growth:** Every time the system operates online, it caches the responses, permanently enriching the node's offline capabilities.

## 5. The Engineering Toolchain

---

Agents cannot click UIs. All installed tools must be completely accessible via CLI or Python.

- **Geometry:** CadQuery and Build123d. The agent writes Python scripts to generate parametric STEP/STL files.
- **Simulation:** CalculiX (via pycalculix) for offline finite element analysis.
- **Manufacturing:** PrusaSlicer CLI or CuraEngine to automate the pipeline from generated 3D geometry directly to printable G-code.