ChatGPT

Ark-OS-NOA Build Instruction Manual

Summary

Ark-OS-NOA is a **local-first agentic operating system** designed to run on Windows 11 Professional hardware without relying on external services. It treats your machine as an autonomous ecosystem: a hive-mind of specialized micro-agents orchestrated by an executive controller that plans, acts, learns, adapts and self-upgrades your software and hardware environment. It offers offline privacy, near-zero latency and full customization because all models, data and tools are stored locally [1]. The system's layered architecture combines a **sandbox/research plane** for digesting new capabilities, a **coordinator/control plane** that evaluates and promotes those capabilities, and a **deployed application plane** that runs the live agents. An internal **Trifecta-Court** governs promotions—ensuring that no new model, tool or micro-agent enters production without passing constitutional checks for safety, cost and compliance. This manual describes how to build, configure and package Ark-OS-NOA as a Windows 11 Professional application (.exe) with fully automated command-line operations.

1 Background and Philosophy

Modern enterprise AI stacks are rapidly moving away from monolithic cloud models toward **agentic operating systems** that coordinate multiple AI workers and tools. Running agents locally provides several advantages: your data never leaves your machine, performance is far faster than cloud APIs, the system can run completely offline, there are no per-token API fees and you can customize every component [1]. An **AI agent operating system** manages resources, orchestrates tasks and provides a framework for multiple agents to work together [2].

Ark-OS-NOA builds on these principles with some key innovations:

- **Micro-agent stacks:** inspired by micro-services, micro-agents are tiny specialized AI components that handle one task extremely well [3]. They use minimal compute, run locally and can be composed into larger workflows. Benefits include higher speed, lower cost, specialization and privacy [4].
- **Dynamic UI and data centrality:** the interface adapts automatically to the underlying data structures and user permissions [5]. Real-time dashboards update as agents ingest data and run tasks.
- **Three-plane architecture:** a sandbox/research cluster digests and tests new capabilities, a coordinator cluster decides promotions using constitutional policies, and a deployed cluster runs the live agents.
- **Trifecta-Court governance:** a three-branch governor—executive (NOA executive), legislative (Board/ModelSelector agents) and judicial (Court engine)—ensures that only safe, cost-effective and policy-compliant capabilities reach production. Emergency overrides require quorum and all decisions are auditable.

- **Offline model management:** models are downloaded directly from Hugging Face repositories and cached locally. The environment variable `HF_HUB_OFFLINE=1` prevents any network calls; only the cached files are used [6].

2 High-Level Architecture

Ark-OS-NOA divides the system into three cooperating planes:

1. **Sandbox/Research Plane** — An isolated environment (for example, a Hyper-V virtual machine or WSL2 instance) where new models, tools and agents are ingested. In this plane the system:
2. Clones or downloads candidate models, code and data.
3. Builds them in an **ephemeral sandbox** so they cannot harm the host.
4. Runs unit, integration and soak tests to generate scorecards, SBOMs (software bills of materials) and risk profiles.
5. Uses built-in micro-agents to digest functions of the host OS—fingerprinting hardware (CPU/GPU/TPU, RAM/VRAM, storage), OS primitives (Windows services, drivers, WSL/Hyper-V, registry keys), network topology and security posture. This environment graph anticipates mismatches and pre-configures caches, toolkits or rollback points.
6. Produces evidence bundles for the coordinator.
7. **Coordinator/Control Plane** — A control server (running on the same PC or a dedicated service) that:
8. Maintains a **capability registry** and compatibility matrix.
9. Evaluates sandbox scorecards for performance, cost and safety.
10. Runs the **Trifecta-Court**: the executive commander proposes promotions; the legislative board sets policy thresholds; the judicial court runs static and dynamic checks (licenses, supply-chain provenance, jailbreak red-team prompts) and issues verdicts (approve, deny, conditional). If court vetoes, promotions stop.
11. Manages feature flags, canary cohorts, budget enforcement and rollback triggers. It monitors performance and error budgets; breaches cause automatic rollbacks. This plane also selects optimal models and tools based on real-time telemetry.
12. **Deployed Application Plane** — A packaged Windows application (.exe) that hosts the live agents. It includes:
13. The NOA **ExecutiveCommanderChiefAgent** (the master orchestrator).
14. A **Board** of domain-specific agents and **ModelSelector** agents that choose which micro-agents and models to use.
15. Multiple **micro-agent stacks**, each comprising a CommandChief agent and one or more specialized micro-agents (for searching, coding, summarizing, planning, etc.). Micro-agents are created or shut down on demand.
16. **Storage** subsystem: local databases (PostgreSQL with pgvector, or Qdrant/Chroma) for embeddings, MinIO or S3-compatible object storage for binaries and datasets, and a vector memory layer for retrieval-augmented generation.
17. **Dynamic UI** built with Electron or a cross-platform framework that renders dashboards, logs and tool outputs in real time.

Together, these planes enable new capabilities to be **ingested, tested, vetted and promoted** automatically without disrupting the live system.

3 Core Capabilities

The following features differentiate Ark-OS-NOA from simple agent frameworks:

3.1 Local-First Operation

Running models and agents locally gives you total control over data, no network latency and no API costs. The Arsturn guide notes that a local AI agent operating system provides total privacy, low latency, offline operation and unlimited customization [1]. All models and data stay on your PC; nothing is sent to the cloud unless you explicitly permit it.

3.2 Agentic Operating System Functions

An AI agent OS manages resources and orchestrates tasks for multiple agents [2]. Ark-OS-NOA implements:

- **Goal & Task Management:** The ExecutiveCommander decomposes user goals into actionable tasks. High-level goals (e.g., "digest all engineering docs and generate a micro-agent plan") are broken into smaller tasks for micro-agents to execute.
- **Perception & Input:** Agents can ingest data from local files, websites (via offline cached content) and user input. The perception module tokenizes, classifies and embeds data into the memory layer for later retrieval.
- **Memory & Knowledge:** A vector database stores both short-term conversation history and long-term documents [7]. Agents use retrieval-augmented generation (RAG) to query relevant embeddings when reasoning.
- **Reasoning & Planning:** The LLM kernel (see Section 4.6) enables agents to reason, plan and decide next steps. It manages context switching and concurrency. AIOS-inspired features such as intelligent resource management, context switching and parallel execution improve efficiency [8].
- **Action & Tool Use:** Agents can run code, call APIs, manipulate files and interact with the OS. Built-in tool managers handle external API calls, while micro-agents perform specialized tasks like generating Python code or summarizing documents.
- **Orchestration & Scheduling:** A scheduler assigns tasks based on resource availability and priority. The context manager saves and restores agent state [9]. The coordinator allows multiple agents to run in parallel [10] and ensures smooth context switching [11].
- **Granular Permissions:** Every agent runs under a role-based permission model. Access to files, tools and network functions is controlled by policies in the Board's constitution [12].

3.3 Dynamic UI

Ark-OS-NOA provides a **dynamic user interface** that adapts to the current data structures and user permissions. The Orbitype analysis of agentic cloud OS notes that dynamic UI layers eliminate manual configuration by adapting interfaces to data and roles [5]. The NOA UI engine binds dashboard components (charts, logs, forms) to the underlying memory schema; when new vector tables or agent outputs appear, UI widgets are created automatically. Users can pin widgets, create custom layouts and monitor agent progress live.

3.4 Offline Model Management

Models are downloaded directly from Hugging Face and stored locally. To ensure offline operation you should:

1. Download the model repository on a machine with internet access using `git clone https://huggingface.co/<model-id>` or the `huggingface-cli` tool.
2. Copy the repository into the NOA cache directory on the target machine (e.g., `%USERPROFILE%\.cache\huggingface\hub`).
3. Set environment variables:

```
$env:HF_HOME = "$env:USERPROFILE\.cache\huggingface"
$env:HF_HUB_CACHE = "$env:HF_HOME\hub"
$env:HF_HUB_OFFLINE = "1"
```

   The `HF_HUB_OFFLINE` variable prevents any HTTP calls and forces the library to read only cached files [6].
4. In your Python code, load models using `from_pretrained` with the local path:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
model = AutoModelForCausalLM.from_pretrained("C:\\models\\llama-3");
tokenizer = AutoTokenizer.from_pretrained("C:\\models\\llama-3");
```

By caching models and setting `HF_HUB_OFFLINE=1`, you can run inference and fine-tuning without any network connectivity [6].

3.5 Micro-Agent Stacks

Micro-agents are tiny models or scripts that perform one specific task extremely well—extracting entities, summarizing a document, or classifying an email. A micro-agent:

- Focuses on a single task, uses minimal compute, runs locally and integrates easily into workflows [13].
- Provides speed, cost-efficiency, specialization, scalability and privacy benefits [4].

Ark-OS-NOA organizes micro-agents into **stacks**: each stack has a **CommanderChiefAgent** that orchestrates one or more micro-agents to accomplish a goal. For example, a “Research Stack” may include a web-search micro-agent, a summarization micro-agent and a citation manager. Stacks can be spawned or removed dynamically by the ExecutiveCommander; the coordinator monitors their performance and kills under-performing stacks.

3.6 Resource Management and Parallelism

Borrowing from AIOS, the coordinator implements **intelligent resource management** to allocate CPU/GPU/IO resources based on real-time demand [14]. Smooth context switching allows agents to transition between tasks without losing state [11], and parallel execution ensures that multiple micro-agents can run concurrently [10]. Built-in tools and utilities simplify agent development [15], while granular permission systems enforce security [12].

3.7 7-Layer Stack Reference

Although Ark-OS-NOA uses its own architecture, it draws inspiration from the open-source seven-layer agent stack described by FutureAGI: infrastructure, language model engine, agent framework, memory & context, tools & integrations, orchestration & workflows, and interfaces [16]. This modular stack highlights the importance of separating concerns so components can be replaced without rewriting the entire system. It also emphasizes vendor independence, scalability and cost control [17].

4 Installation Prerequisites

4.1 Hardware Requirements

Component	Recommendation
Operating System	Windows 11 Professional (64-bit) with latest updates and optional Hyper-V/WSL2 features enabled.
CPU	Multi-core (8+ cores preferred) to run micro-agents in parallel.
GPU	NVIDIA GPU with ≥6 GB VRAM for LLM inference; optional but recommended.
Memory	32 GB RAM or higher for running multiple models and vector databases.
Storage	≥1 TB SSD/NVMe; models and caches may consume hundreds of gigabytes.
Network	Only required for initial downloads; offline operation thereafter.

4.2 Software Dependencies

1. **Python 3.10+** — core language for agent logic. Install via the official Windows installer and add Python to your PATH.
2. **Git** — used to clone model repositories and manage code.
3. **Node.js (optional)** — required if you build the dynamic UI with Electron or Tauri.
4. **PostgreSQL** — install the Windows distribution and enable the pgvector extension, or choose an alternative vector store like Qdrant or Chroma.
5. **MinIO** — S3-compatible object storage server for caching models, datasets and SBOMs.
6. **PyTorch** — for running transformer models; install the CUDA-enabled version if you have a GPU.
7. **Transformers** and **huggingface_hub** libraries — for model loading; install with pip.
8. **LangChain** or **LlamaIndex** — for building agent flows and retrieval-augmented generation.
9. **PyInstaller** — to package the Python application into a standalone Windows executable. It can be installed via pip: `pip install pyinstaller` [18].

   Note: When using PyInstaller, ensure you also install the matching `pyinstaller-hooks-contrib` package. After installation, verify the command is available with `pyinstaller --version` [18]. The final build step will package your Python scripts, dependencies and data into a single `.exe` file.

5 Preparing Your Environment

5.1 Set Up Python Environment

1. Install Python 3.10+ and ensure `pip` is available.
2. Create a virtual environment:

```
python -m venv C:\arkos\venv
.\C:\arkos\venv\Scripts\Activate.ps1
```

3. Install core packages:

```
pip install torch transformers huggingface_hub langchain llama-index
pgvector psycopg2-binary minio uvicorn fastapi
pip install pyinstaller pyinstaller-hooks-contrib
```

5.2 Download and Cache Models

To run models offline:

1. On a machine with internet access, clone the desired models and tokenizers:

```
git lfs install
git clone https://huggingface.co/meta-llama/Llama-3-8B-instruct C:
\models\llama-3-8b
```

2. Copy the directory to the target PC (e.g., `C:\models`).
3. Set environment variables as described in Section 3.4 to force offline mode [6].
4. Verify offline loading by running a quick script:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
model_path = "C:\\models\\llama-3-8b"
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModelForCausalLM.from_pretrained(model_path)
print(tokenizer.decode(model.generate(tokenizer.encode("Hello",
return_tensors="pt"), max_length=20)[0]))
```

The script should run without internet. If it attempts to fetch remote files, ensure that `HF_HUB_OFFLINE=1` is set.

5.3 Install Vector Database

Option A – PostgreSQL with pgvector

1. Download and install PostgreSQL 15+ for Windows.
2. Create a database (e.g., `noa_memory`) and enable the `pgvector` extension:

   ```
   CREATE DATABASE noa_memory;
   \c noa_memory
   CREATE EXTENSION IF NOT EXISTS vector;
   ```

3. Define tables for embeddings (id, doc_id, vector, metadata). Use LangChain or LlamaIndex connectors to store and retrieve embeddings.

Option B – Qdrant or Chroma

1. Download prebuilt Qdrant binary (a single executable) or run via Docker for Windows. Qdrant is optimized for high-dimensional vectors and can run offline.
2. Configure persistent storage path in the configuration file. Use the Python `qdrant-client` to upsert and search vectors.

5.4 Set Up Object Storage

Run MinIO as a Windows service or inside a container. MinIO will store large files (models, datasets, SBOMs) in an S3-compatible API. Configure a bucket called `arkos` with versioning. Set environment variables for access key and secret key in your micro-agent scripts.

5.5 Install and Configure LangChain/LlamaIndex

Use LangChain or LlamaIndex to build agentic workflows. Define a **tool** for each micro-agent (e.g., `search_tool`, `code_executor`, `summarizer`) and register them in a central `AgentToolkit`. Create a custom **planner** (ReAct, tree-of-thought, or T-ultra) that sequences tools. LangChain's `MultiActionAgent` or LlamaIndex's `ServiceContext` can orchestrate micro-agents to fulfil a goal.

6 Building Ark-OS-NOA

6.1 Project Structure

Organize your repository as follows (example):

```
arkos/
├── noa_app/                # Python package for the NOA application
│   ├── __main__.py         # Entry point – runs the ExecutiveCommander
│   ├── commander.py        # ExecutiveCommanderChiefAgent implementation
│   ├── board_agents/
│   │   ├── board.py        # Board orchestrator and policy definitions
```

```
│   │   ├── digest_agent.py  # Research/Digest agent associated with the board
│   │   └── model_selectors.py # ModelSelector agents
│   ├── stacks/
│   │   ├── __init__.py
│   │   ├── research_stack.py # Example micro-agent stack
│   │   └── ...
│   ├── tools/
│   │   └── search.py        # Micro-agent tools (search, code execution, etc.)
│   ├── sandbox/
│   │   └── sandbox_runner.py # Functions to create and manage sandbox VMs/
containers
│   ├── coordinator/
│   │   ├── registry.py      # Capability registry and metadata store
│   │   ├── court.py         # Trifecta-Court implementation
│   │   └── promotion.py     # Promotion controller (canary and rollback logic)
│   ├── ui/
│   │   ├── main.tsx         # React/Electron UI entry
│   │   └── components/      # UI components and dashboards
│   └── config.py            # Configuration (paths, environment variables)
├── models/                  # Cached Hugging Face models
└── build_scripts/
    ├── package.py           # PyInstaller build script
    └── generate_sbom.py     # Script to generate SBOMs
```

This structure separates concerns: core logic, agent definitions, sandbox functions, coordinator services and UI. You can adjust names to match your preferences.

6.2 Implementing the Executive Commander

The **ExecutiveCommanderChiefAgent** is the heart of the deployed plane. It:

1. Loads configuration and environment variables.
2. Connects to the vector database and object storage.
3. Initializes the Board agents and ModelSelector agents, passing them memory and tool references.
4. Listens for user goals (entered through the UI or read from a file) and decomposes them into tasks.
5. Delegates tasks to micro-agent stacks. If a stack does not exist for a task, it asks the board to deploy one.
6. Monitors stack outputs and resolves dependencies. When all sub-tasks are complete, it returns the result to the user.
7. Records telemetry (latencies, errors, resource use) and sends it to the coordinator for analysis.

The commander can be implemented as an asynchronous Python class using `asyncio`. The board and model-selector agents can run in their own threads or processes. Each micro-agent should be stateless where possible so that the commander can shut it down or respawn it as needed.

6.3 Building Micro-Agent Stacks

Define micro-agents as Python functions or classes with a consistent interface (e.g., `run(input: dict) -> dict`). Example micro-agents might include:

- **IngestAgent** – reads files and embeds them in the vector database.
- **SearchAgent** – queries the vector database or offline web cache using semantic search.
- **SummarizeAgent** – summarizes text using a local summarization model.
- **CodeExecutorAgent** – runs code in a secure sandbox and returns results or errors.
- **PlanAgent** – uses the LLM to plan sequences of actions.

Group related micro-agents into a **stack**. Each stack has a **CommanderChief** that:

1. Initializes micro-agents.
2. Accepts tasks from the ExecutiveCommander.
3. Schedules micro-agents in the appropriate order.
4. Aggregates outputs and returns a combined result.

Design stacks to be composable—stacks can call other stacks as subroutines. For instance, a “Digest Stack” may use the SearchAgent to fetch data and then the SummarizeAgent to digest it.

6.4 Implementing the Coordinator and Court

The coordinator runs as a separate service (could be a FastAPI app) that:

1. **Registers capabilities**: Each new model, tool or micro-agent is represented as a capability document (YAML or JSON) with metadata about purpose, inputs/outputs, dependencies, risks and tests.
2. **Runs promotions**: The **promotion controller** selects candidate capabilities from the sandbox, reads their scorecards and decides whether to canary them in production. It uses thresholds defined by the legislative board (latency increase $\leq$10 %, failure rate $\leq$0.5 %, cost delta $\leq$5 %, zero safety events). Breaches trigger auto-rollback.
3. **Operates the Trifecta-Court**: The court enforces policy by performing static checks (license compliance, supply-chain integrity, RBAC) and dynamic checks (prompt injection tests, privacy analysis). It can veto, approve or approve with conditions. Courts can also revoke capabilities if telemetry later shows they violate policy.
4. **Maintains the capability registry**: A database mapping capability IDs to versions, statuses (sandbox, canary, production), SBOMs and risk profiles.

Implementing the court requires writing rules in a domain-specific language or using a policy engine like [Open Policy Agent]. The board defines the constitution as code, and the court runs that code automatically. Court decisions are logged and auditable.

6.5 Building the Dynamic UI

The UI can be built with Electron, Tauri or a .NET UI toolkit. Key features:

- **Dashboard**: shows active agents, stacks, tasks, and their status. Data visualizations update automatically using websockets or an event bus. The UI must support dynamic creation of widgets when new micro-agents or vector tables appear [5].
- **Goal Input**: a text box for entering high-level goals. When submitted, the ExecutiveCommander receives the goal via a local API (FastAPI/Flask) and begins planning.
- **Logs & Telemetry**: real-time feed of logs from agents, sandbox runs, promotions and court verdicts. Users can drill down into SBOMs and test results.
- **Settings**: configuration panel for environment variables, resource budgets, offline caches and role permissions.

When packaging the application, include the UI bundle within the `noa_app\ui` folder and serve it using a local HTTP server or directly from Electron.

6.6 Packaging into a Windows Executable

1. Ensure all Python modules are installed within the virtual environment.
2. Use PyInstaller to build the application. Create a **spec file** if your application requires data files, models and external binaries:

```python
# build_scripts/package.py
import PyInstaller.__main__
PyInstaller.__main__.run([
    'noa_app/__main__.py',
    '--onefile',
    '--name', 'arkos-noa',
    '--add-data', 'models;models',
    '--add-data', 'noa_app/ui/dist;ui',
    '--collect-all', 'transformers',
    '--hidden-import', 'pkg_resources.py2_warn',
])
```

3. Run the build script:

```
python build_scripts/package.py
```

PyInstaller will create `dist\arkos-noa.exe`. When launched, this executable starts the ExecutiveCommander, spins up the coordinator (if configured to run locally), launches the UI and initializes micro-agent stacks. Terminal commands (setting environment variables, starting MinIO, launching Postgres) should be executed automatically in Python via `subprocess` or `os.system`. Provide clear log messages and handle errors gracefully.

4. Test the `.exe` on a clean Windows 11 VM to verify that all dependencies are bundled. If missing DLLs or libraries are reported, add them to the spec file.

5. Sign the executable with a code-signing certificate if distributing to others.

7 Operational Workflow

7.1 Starting the System

1. Launch `arkos-noa.exe`. The application should display the UI and log messages indicating initialization.
2. On first run, the application will fingerprint the host environment (OS version, CPU/GPU, memory, storage, network) and build an **Environment & Function Graph**. This information influences model selection and sandbox preparation (e.g., pre-staging CUDA libraries on systems with GPUs).
3. The ExecutiveCommander loads existing micro-agent stacks and capabilities from the registry. It also starts watchers for drift detection (SBOM deltas, version updates, resource anomalies).
4. The UI shows the system state: stacks, agents, resource usage and pending promotions.

7.2 Running a Goal

1. Enter a goal in the UI (e.g., “Summarize the last quarter’s sales data”).
2. The ExecutiveCommander decomposes the goal into tasks and selects appropriate stacks. For example, it might deploy a `DataIngestStack` to ingest spreadsheets, a `AnalysisStack` to compute metrics and a `SummaryStack` to generate natural-language summaries.
3. Each stack runs its micro-agents. The SearchAgent queries the vector DB; the SummarizeAgent uses a local summarization model; the CodeExecutor runs Python scripts if needed.
4. When all tasks are complete, the commander aggregates results and returns them to the UI.
5. Telemetry is sent to the coordinator for monitoring and learning. If tasks breached budgets (e.g., high latency), the coordinator logs the event and may adjust model selection or trigger a rollback of the offending capability.

7.3 Adding New Capabilities

1. Copy or clone the new model/tool/agent into the sandbox input directory. Create a capability specification (YAML) describing its purpose, inputs/outputs, dependencies, risks and tests.
2. The sandbox runner detects the new capability and spins up an isolated VM/container. It installs dependencies, runs unit and integration tests, collects SBOMs and risk metrics.
3. The sandbox sends the scorecard and SBOM to the coordinator. The Board evaluates the results against policy thresholds and passes the request to the Court.
4. The Court applies constitutional checks (license compliance, supply chain integrity, dynamic red-teaming) and returns a verdict (approve, approve with conditions or reject). Conditional approvals may require enabling kill-switches or limiting resource usage.
5. If approved, the promotion controller runs a canary deployment: enabling the capability for a small percentage of tasks or limited time. It monitors latency, error rates, cost and safety. If metrics are within thresholds, the capability is promoted to production; otherwise it is rolled back automatically.

7.4 Updating and Self-Upgrading

Ark-OS-NOA can self-upgrade by ingesting new versions of existing models or agents. When drift watchers detect that a model has an available upgrade, they trigger the sandbox pipeline. The Court ensures that upgrades do not introduce regressions or violate policy. Promotions follow the same canary process. All upgrade decisions are logged for audit.

8 Security and Compliance

Ark-OS-NOA is designed with privacy and security as first principles. Key practices include:

- **Local storage** — all data, models and logs remain on the host. Internet access is optional and disabled by default.
- **Role-based access control** — Board agents define which micro-agents can access particular files, networks or devices. Sensitive operations require human approval.
- **SBOMs and provenance** — The sandbox generates SBOMs for all capabilities, enabling supply-chain auditing and license compliance. The Trifecta-Court rejects capabilities that violate open-source licenses or originate from untrusted sources.
- **Environment isolation** — Sandbox VMs/containers ensure that new code cannot impact the deployed environment until it passes testing.
- **Secure secrets** — Secrets (API keys, credentials) are sealed to the Windows Data Protection API (DPAPI) and never exposed to agents directly.
- **Audit logging** — All actions (goal submissions, promotions, court verdicts, rollbacks) are logged and tamper-evident.

9 Extending Ark-OS-NOA

- **Create new micro-agents** by adding Python modules implementing the `run()` interface. Update the stack configurations and capability specifications accordingly. Use the sandbox to test them.
- **Add new models** by downloading them offline and adding them to the cache. Update the model selector configuration to include the new model and its supported tasks.
- **Customize UI dashboards** by editing the `ui/components` directory. Use dynamic data binding to display new metrics or logs.
- **Integrate hardware sensors** by writing micro-agents that query Windows APIs (e.g., WMI for system health) and ingest results into the environment graph.

10 Conclusion

Ark-OS-NOA combines the privacy and performance advantages of local-first AI with the power of agentic architecture. Its micro-agent stacks, dynamic UI and Trifecta-Court governance provide a robust framework for building autonomous, self-upgrading applications that can run offline on Windows 11 Professional. By following this manual—setting up your environment, caching models, implementing the agent framework, configuring the coordinator and packaging the system as an executable—you can build a comprehensive AI operating layer that digests everything, adapts to its host and orchestrates complex workflows without external dependencies.

---

[1] [2] [7] Build a Local AI Agent Operating System: A Complete Guide
https://www.arsturn.com/blog/building-a-local-first-ai-agent-operating-system-a-guide

[3] [4] [13] The Rise of AI Micro-Agents: Tiny Models Automating Big Tasks - DEV Community
https://dev.to/koolkamalkishor/the-rise-of-ai-micro-agents-tiny-models-automating-big-tasks-386m

[5] Orchestrate Apps with Orbitype's Agentic Cloud OS
https://www.orbitype.com/posts/nMGYCZ/orchestrate-apps-with-orbitypes-agentic-cloud-os

[6] Environment variables
https://huggingface.co/docs/huggingface_hub/en/package_reference/environment_variables

[8] [9] [10] [11] [12] [14] [15] Understanding AI Agent Operating Systems: A Comprehensive Guide
https://www.ema.co/additional-blogs/addition-blogs/ai-agent-operating-systems-guide

[16] [17] Open-Source AI Agent Stack 2025: Complete Enterprise Guide
https://futureagi.com/blogs/open-source-stack-ai-agents-2025

[18] How to Install PyInstaller — PyInstaller 6.15.0 documentation
https://pyinstaller.org/en/stable/installation.html