K9-AIF Quick Start Guide

From Zero to a Running Agent in 15 Minutes — Using K9X Studio

Ravi Natarajan

https://k9x.ai

K9-AIF Framework · K9X Ecosystem

Last Updated: 2026-07-21

K9-AIF Quick Start Guide

From Zero to a Running Agent in 15 Minutes — Using K9X Studio

---

Author: Ravi Natarajan
Project: K9-AIF Framework, K9X Ecosystem
Last Updated: 2026-07-21
Companion to: K9-AIF Developer Guide (the 31-chapter reference manual)
Website: https://k9x.ai
Try Studio: https://studio.k9x.ai
Main Repository: https://github.com/k9aif/k9-aif-framework

---

1. What This Guide Is

The Developer Guide is a 31-chapter reference manual — architecture rationale, every ABB contract, every pattern, every gotcha. It is the right document once you're building something real. It is the wrong first document if you have never run K9-AIF before and just want to see the Router → Orchestrator → Squad → Agent hierarchy actually execute.

This guide is that first document. It uses **K9X Studio** — the visual drag-and-drop builder — instead of hand-writing YAML or running the CLI generator, because designing on a canvas makes the four-layer hierarchy visible before you have to reason about it in code. By the end you will have a real, runnable K9-AIF application on your machine, and you'll know exactly which Developer Guide chapter to open next for whatever you want to do to it.

Fast is not the same as shallow-by-design, though. K9-AIF is built on pure Object-Oriented principles — abstract contracts (ABBs) separated from concrete implementations (SBBs), Liskov substitution, open/closed — and on TOGAF's Architecture Building Block discipline. That is what makes the four boxes you drag onto the canvas in Step 3 flexible and extensible rather than a fixed template: every node maps to a real ABB contract with a stable interface, so the system you get after 15 minutes here is architecturally identical in kind to a production system with 40 agents. Nothing about going fast cuts a corner on that. This guide just doesn't dwell on *why* the architecture is shaped this way — that reasoning is Developer Guide Chapter 1 (*Why Architecture-First Agentic AI Matters*) and CLAUDE.md's *Architecture & Design Discipline* section, both worth reading once the app in front of you is running.

If a step here says "for the full picture, see Chapter N" — that's not filler. This guide deliberately stays shallow; the Developer Guide is where the depth lives.

---

2. The Ecosystem in Three Installs

Three packages, all on PyPI:

```
pip install k9x          # K9X Studio - the visual builder this guide walks through
pip install k9-aif       # the framework itself - pulled in automatically as a dependency of k9x
pip install k9x-satan    # adversarial validation - optional for now, see Section 11
```

You only need `k9x` to get through this guide. `k9-aif` comes along for the ride (Studio's backend reads the live ABB library to populate its palette). `k9x-satan` is for later — proving your Shield configuration actually blocks attacks — and isn't part of the 15-minute walkthrough.

---

3. Prerequisites

Check these off before you start the clock:

- **Python 3.10+** for `pip install k9x` (3.11 or 3.12 recommended — avoid 3.14, `venv`/`ensurepip` have known issues)
- **Ollama** (recommended, optional) — running locally with two small models pulled. Everything below works without an LLM too (Studio falls back to rule-based suggestions and templated agents), but the "AI-assisted" path in Step 3 needs this:

```
ollama pull llama3.2:1b         # "general" model
ollama pull granite3-dense:2b   # "reasoning" model
```

- **A `k9-aif-framework` git checkout** — not needed to design in Studio, but needed in Step 6 when you actually *run* the app you export. `setup.sh` in your exported scaffold will clone one for you if you don't already have it, so you can skip this until you get there.

---

4. Step 1 — Get K9X Studio Running

Fastest path — no install: open studio.k9x.ai in a browser and skip to Step 2. This is the hosted instance; `localhost`/`127.0.0.1` are blocked there for LLM endpoints (it would point at the server's own resources), so if you want AI-assisted suggestions against your own Ollama, use your machine's LAN IP instead of `localhost` when configuring the LLM endpoint (Step 3).

Local — via pip (recommended):

```
pip install k9x
k9x config       # writes .env-example - fill in your LLM endpoint, save as .env (optional)
k9x studio       # starts Studio on http://localhost:12999 and opens your browser
```

Run `k9x studio --bg` to run it in the background, `k9x studio --stop` to stop it, or `k9x studio --port <N>` to use a different port.

Local — from an ecosystem checkout (only if you're developing Studio itself, not just using it):

```
cd k9x-ecosystem/k9x_studio
./run.sh    # starts backend :8090 + frontend dev server :5173
```

`run.sh` activates the shared framework venv, installs Studio's own backend and frontend dependencies on first run, and starts two processes:

```
Backend  → http://localhost:8090
Frontend → http://localhost:5173
```

Open `http://localhost:5173`. If `run.sh` exits immediately complaining the shared venv is missing, go back and do the Prerequisites step — Studio does not create `k9-aif-framework/.venv` for you.

---

5. Step 2 — Project Setup

The landing screen asks for four things:

Field	What it's for
Name	Becomes your app's folder name and Python package name
Author	Stamped into the generated scaffold's README

Field	What it's for
Domain	One or two words — e.g. "claims processing", "support tickets"
Description	One or two sentences describing what the system should do

The description matters more than it looks — it's what the AI-assisted architecture suggestion in the next step reasons from. Something like *"Classify incoming support emails by urgency, then draft a response for agent review"* gives Studio enough to work with. A one-word description forces you into the manual canvas path instead.

---

6. Step 3 — Design the Architecture

You have two ways to get from a blank canvas to a wired Router → Orchestrator → Squad → Agent graph. Use (a) for your first run — it's faster and shows you a correct topology to learn from. Use (b) once you already know what you're building.

(a) AI-assisted — click "Generate Architecture"

If you configured an LLM (see the Studio README's *LLM Configuration* section — `.env`, environment variables, `config.yaml`, or the session-only browser panel, checked in that priority order), Studio proposes Orchestrators, Squads, and Agents tailored to your project description and drops them onto the canvas, already connected. Without an LLM configured, this button still works — it produces a generic but valid fallback topology instead of a tailored one.

Either way, review what landed on the canvas before moving on. Click through each node in the **Inspector** panel (right side) and read the role/goal it was given.

(b) Manual canvas — drag, drop, connect

The **Palette** (left side) has one entry per K9-AIF layer:

Palette Node	K9-AIF ABB	What gets generated
Router	`K9EventRouter`	`router/` — Python + config
Orchestrator	`BaseOrchestrator`	`orchestrators/` — Python + config
Squad	`BaseSquad`	`squads/yaml/<name>.yaml`
Agent	`BaseAgent`	`agents/yaml/<name>.yaml` + `agents/src/<name>.py`
Validation Loop	`K9ValidationLoopAgent`	Agent with an iterative confidence-loop scaffold
Critic-Actor	`K9CriticActorAgent`	Agent with an actor/critic refinement scaffold
Guard	`BaseGovernance`	A governance config entry

Drag one **Router**, one **Orchestrator**, one **Squad**, and one **Agent** onto the canvas, in that order. Draw a connection Router → Orchestrator → Squad → Agent. That's the minimum viable topology — one event, one path, one decision-maker at each layer. It mirrors exactly the hierarchy in Developer Guide Chapter 1.5 (*Router → Orchestrator → Squads → Agents Hierarchy*) and Chapter 3 (*Core Architecture*).

---

7. Step 4 — Configure Your Agent

Select the Agent node. In the **Inspector** panel, fill in:

- **Role** — who the agent is (its system prompt persona)
- **Goal** — what it's trying to accomplish
- **Model** — which model-catalog alias it should request (`general` or `reasoning`, if you pulled the Ollama models above)

This is the same `role` / `goal` / `model` triplet that ends up in the agent's YAML — see SKILLS.md Skill 1 if you want to see the raw file this produces. You are designing it; Studio is typing it.

---

8. Step 5 — Export Your Scaffold

Click **Generate** (or **Export**). You'll get a ZIP download containing:

```
config/           config.yaml, agents.yaml, squads.yaml
agents/            one .py per Agent node
squads/            one .py per Squad node
orchestrators/     one .py per Orchestrator node
router/            Python + config for the Router
setup.sh           one-time environment setup
run.sh             starts the app
README.md          project-specific quick start (generated for you)
```

If you're running Studio locally with `K9X_PROJECTS_ROOT` set, you can write directly to `k9_projects/<AppName>/` instead of downloading a ZIP — same output, no unzip step.

---

9. Step 6 — Run It

```
unzip <your-app-name>.zip
cd <your-app-name>

./setup.sh                      # one-time: venv, framework path, deps, .env
source .venv/bin/activate       # if setup.sh created .venv/ for you
./run.sh
```

`setup.sh` walks you through pointing at (or cloning) a `k9-aif-framework` checkout, installs both the framework's and your project's dependencies, and writes `K9_ENV` / `K9_FRAMEWORK_PATH` into a local `.env`. Run `./setup.sh --verify` any time afterward to re-check the environment without redoing setup.

If your Agent's model calls out to Ollama, make sure it's running (`ollama serve`) with the two models from the Prerequisites section pulled. You should see your Router receive an event, hand it to the Orchestrator, the Orchestrator run the Squad, and the Squad execute your Agent — printed to the console as it happens.

---

10. What You Just Built

```
Event → Router (K9EventRouter)
            Orchestrator (BaseOrchestrator)
                  Squad (BaseSquad)
                        Agent (BaseAgent) → LLM → result
```

Every box in that diagram is a concrete instance of a framework ABB contract — nothing here is a toy simplification of how a production K9-AIF system works, it's the same hierarchy at a smaller scale. Three-layer decoupling still applies: your Router knows only its Orchestrator, the Orchestrator knows only its Squad, the Squad knows only its Agent. Nothing you built violates the rules a much larger EOC-scale system also has to follow.

---

11. Where to Go Next

This guide stops here on purpose. Everything past this point is Developer Guide territory:

You want to…	Read
Add a second agent, or wire a real squad flow	Chapter 5 (Agent Development Guide), SKILLS.md Skill 1
Make an agent iterate instead of one-shot	Chapter 8 (Validation Loop Pattern)
Add actor/critic self-review to an agent	Chapter 9 (Critic-Actor Pattern)
Enforce governance before/after an agent runs	Chapter 13 (Governance and Zero Trust)
Harden the pipeline against prompt injection, PII leakage, etc.	Chapter 14 (K9X Security and Vulnerability Checks)
Prove your Shield configuration actually blocks attacks	`pip install k9x-satan` — Chapter 14, k9x.ai/k9x-security
Connect a real LLM provider (Claude, Bedrock, OpenAI) instead of Ollama	Chapter 23 (Provider Adapter Pattern), SKILLS.md Skill 13
Understand everything Studio's canvas can do	Chapter 20 (K9X Studio Integration)
Publish your SBB for reuse across projects	Chapter 21 (K9X Enterprise Continuum)
Understand *why* the architecture is shaped this way — OO principles, TOGAF ABBs, why this pays off at scale	Chapter 1 (Introduction), CLAUDE.md's Architecture & Design Discipline
See the whole framework, end to end	The full Developer Guide

---

12. Troubleshooting

Symptom	Fix
`k9x: command not found` after `pip install k9x`	Confirm the install went into the Python environment on your `PATH` — check with `pip show k9x`
`run.sh` exits: "shared venv not found" (ecosystem-checkout path only)	Create `k9-aif-framework/.venv` first — see Prerequisites
"Generate Architecture" gives a generic result	No LLM configured — run `k9x config`, fill in the LLM endpoint, save as `.env`
LLM endpoint rejected on studio.k9x.ai	`localhost`/`127.0.0.1` are blocked on the hosted instance — use your machine's IP or hostname
`Connection refused` calling the model	Ollama isn't running — `ollama serve`, then confirm the models are pulled
Port 12999 (pip path) or 8090/5173 (ecosystem-checkout path) already in use	`k9x studio --port <N>`, or stop whatever's using the port
`setup.sh` (Step 6) can't find `k9_aif_abb/` at `K9_FRAMEWORK_PATH`	Point it at (or let it clone) an actual `k9-aif-framework` checkout — a pip-installed `k9-aif` package alone isn't enough for this step

Symptom	Fix
Exported app runs but agent output looks like a stub	Confirm the model alias in the Agent's YAML matches an entry in `config.yaml`'s `model_catalog`

References

- K9-AIF Developer Guide — the full reference manual: `Developer-guide.pdf` (this folder)
- K9X Studio: studio.k9x.ai · github.com/k9aif/k9x-studio · `pip install k9x`
- K9-AIF Framework: github.com/k9aif/k9-aif-framework · `pip install k9-aif`
- K9X SATAN — adversarial validation: satan.k9x.ai · `pip install k9x-satan`
- K9X Security — full vulnerability check inventory: k9x.ai/k9x-security