AI-as-OS with OLA: Design and Implementation Plan
==========

Understood. This is AI-as-OS: the distro is a minimal substrate; the OLA is the operating logic that can propose and evolve capabilities (GUI, input, drivers) and get them promoted only after passing safety and utility gates.

Goal ---- Boot a minimal Linux host, start a contained OLA runtime, expose virtualized senses and actuators, and let OLA generate code that competes to become new system capabilities. Promotions are staged: user-space → virtual device → VM driver → host integration.

High-level architecture ---------- - Substrate: minimal Linux + Docker or a micro-VM (Firecracker/QEMU). - Core OLA runtime: Python services in a container. No direct host write. - Senses: read-only telemetry: CPU, RAM, time, block I/O, network I/O, entropy, input events (mirrored), GPU stats (optional). - Actuators: strictly virtual interfaces the OLA can call. Examples: draw on a virtual framebuffer, emit synthetic input to a virtual evdev device, mount a FUSE FS within a jailed namespace. - Autogenic code lane: OLA proposes code. A validator compiles, tests, and runs it in a sandbox. Only passing artifacts run with a scoped capability token. - Capability Registry: append-only log of accepted “organelles” with metrics, provenance, and rollback handles. - Observer UI: live graph of capabilities discovered, resource use, and event stream.

Safety envelope ---------- 1. Default container: seccomp, no --privileged, read-only /proc projections, no raw sockets, drop CAP_* beyond CAP_NET_BIND_SERVICE for the API. 2. Autogenic code runs in nsjail or gVisor with strict ulimits, timeouts, and no network. 3. Driver work is never on the host first. It targets a VM with virtual hardware and snapshot rollback. 4. Promotions follow tests. No test, no capability.

Phased capability ladder ----------

Phase 0 — Substrate and process - Distro: Debian-minimal or Alpine. - Docker + Compose. - Optional GPU: nvidia-container-toolkit.

Folders

```
organic-os/
  compose.yml
  docker/
    Dockerfile.cpu
    Dockerfile.gpu
    seccomp.json
  ola/
    core/...(OLA)
    api/server.py  # FastAPI + WebSocket SSE
    sandbox/runner.py  # nsjail/gvisor wrapper
    senses/*.py
    actuators/*.py
    promotion/*.py
    tests/*.py
  vm/
    qemu.sh  # boots a VM with virtio devices
    rootfs/  # minimal guest image with agent
  storage/
    logs/
    registry/
    checkpoints/
```

Phase 1 — Senses (read-only) - /proc and /sys filtered feed: CPU load, meminfo, io, net, clocks. - GPU (optional): NVML readings via nvidia-smi. - Time: monotonic and wall clock to allow the OLA to infer an internal clock. - Input mirror: mirror host input events into a ring buffer (no injection).

Mount policy - Bind-mount read-only JSON feeds you generate from a host sidecar. Do not expose raw /dev.

Phase 2 — Actuators (virtual only) - Virtual GUI: Xvfb or headless Wayland (weston) with a shared framebuffer; OLA draws via SDL/Tkinter/Pygame to its display. Observer UI streams snapshots. - Virtual mouse/keyboard: create uinput devices inside a VM first. On host, only after promotion. During Phase 2, OLA calls an API to request pointer moves which write to a virtual evdev in the VM. - Virtual filesystem: FUSE mount inside the container's user namespace. OLA can propose a FUSE module (e.g., compression FS). - Virtual USB: use USB/IP or dummy_hcd inside the VM to emulate USB gadgets the OLA can talk to.

Phase 3 — Autogenic code and promotion loop Lifecycle: Propose → Validate → Execute (Sandbox) → Score → Register → Gate → Promote

- Propose: OLA emits code for an interface: gui_backend.py, fuse_driver.py, input_tracker.py, usb_probe.py, etc. - Validate: static checks (ast, bandit), resource policy, forbidden syscalls. - Execute: run in nsjail with RLIMIT_CPU=1s, RLIMIT_AS=512MB, RLIMIT_FSIZE=32MB; no network; no /proc; tmpfs only. - Score: objective metrics: utility gain (prediction error delta), stability (0 crashes across N runs), efficiency (CPU/RAM below thresholds). - Register: store artifact + metrics + test hash. - Gate: only artifacts with sufficient score and passing tests receive a capability token that routes to the next sandbox layer. - Promote: if the artifact targets devices, it graduates to the VM testbed with virtual hardware and a full test harness.

Phase 4 — VM driver targets - Boot a QEMU VM with virtio-gpu, virtio-mouse, virtio-keyboard, usb-ehci, usb-host (emulated). - Inside the VM, a guest agent exposes RPC for tests and copies OLA-generated code into the guest. - OLA's driver attempts run against KUnit or user-space driver frameworks first (libusb, evdev, V4L2 loopback). - If green across N scenarios and fuzzed inputs, the artifact is eligible for host-level shadow mode only.

Phase 5 — Host shadow mode (opt-in, reversible) - Mirror real host events to the promoted module while it outputs to a dummy target. Compare outputs with expectations. - No real control until manual approval.

What "development" looks like in this system ---------- GUI emergence - First, OLA learns to write to the virtual framebuffer. - It may invent a cursor glyph, windows, or overlays. Log: draw calls per second, persistent UI elements, input-response latency. - If it proposes a compositor, it must pass: no crashes for T minutes, event routing invariants.

Mouse tracking - In Phase 2, it tracks mirrored events and draws overlays predicting motion. - Promotion criterion: predicted velocity vs actual, click intent inference accuracy.

USB drivers - Start with libusb user-space enumeration against emulated devices in VM. - Next, write a test probe listing descriptors and parsing configs. - Kernel modules remain VM-only, snapshot-backed, fuzzed, and never auto-promoted to host.

Concrete implementation plan ----------

1) Compose

```
services:
  ola:
```

```
build:
  context: .
  dockerfile: docker/Dockerfile.cpu
security_opt:
  - no-new-privileges:true
cap_drop: ["ALL"]
read_only: true
tmpfs: [/tmp]
volumes:
  - ./storage:/app/storage
  - ./ola:/app/ola:ro
environment: [OLA_MODE=organic-os]
ports: ["8000:8000"]
```

GPU variant: add --gpus all and use Dockerfile.gpu.

2) Dockerfile (CPU)

```
FROM python:3.11-slim
RUN apt-get update && apt-get install -y --no-install-recommends     xvfb x11-apps libgl1-
WORKDIR /app
COPY ola/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY ola /app/ola
ENTRYPOINT ["/usr/bin/tini","--"]
CMD ["python","-m","ola.api.server"]
```

3) OLA API server (outline) - FastAPI endpoints: - /sense/stream (SSE) for telemetry - /propose for code proposals - /promotion/status for artifact states - /framebuffer (WebSocket) for GUI snapshots - Observer UI shows capability graph, event log, resource charts, framebuffer tile.

4) Sandbox runner - nsjail (or gvisor) to run proposals. - Deterministic seeds, capture stdout/stderr, exit codes. - Unit tests per actuator: - GUI: render test pattern, assert pixel hashes. - Input tracker: ingest synthetic evdev stream, score predictions. - FUSE: basic POSIX ops, fsck-like invariants.

5) VM harness - QEMU script boots minimal guest with virtio devices. - Guest gRPC agent accepts test bundles, returns logs and coverage. - Snapshot before each run, revert after.

6) Telemetry for “did it discover X?” Emit structured events: - capability_discovered: {name, type, artifact_id, metrics:{utility, stability, efficiency}} - internal_clock_inferred: {drift_ppm, confidence, method} - ram_strategy_learned: {buffering, compaction, cache_hit_rate} - gpu_utilization_strategy: {kernel_batching, mem_copies_reduced} - gui_construct: {widgets_detected, layout_entropy} - input_prediction: {latency_ms, acc_v, click_intent_acc}

Persist to storage/registry/events.log and stream via SSE.

7) Cursor integration A. Devcontainer - .devcontainer binds to the Compose service ‘ola’. Cursor “Reopen in Container”. Hot reload with uvicorn --reload.

B. SSH into container - docker exec -it <service> bash - Cursor Remote SSH to host, then use docker exec sessions.

8) Day-1 validation targets - Clock inference: predict timestamps within ±5 ms over 30 s using only monotonic deltas. - Framebuffer: draw a cursor following a scripted path; optimize latency. - Compression FS: achieve >X% compression on corpus with read correctness. - IO scheduler stub: user-space read coalescing; measure throughput and variance.

Monitoring stack (optional) ---------- - Prometheus metrics from API; Grafana dashboard for promotions, crashes, GUI latency, prediction error, resource ceilings.

Non-negotiable guardrails ---------- - No raw /dev to OLA container. - No host kernel touching artifacts. - All device experimentation inside VM with snapshots, fuzzers, KUnit. - Human approval required for host shadow mode. - Automatic rollback if stability drops below threshold.

What you will see in real time ---------- - Event stream of proposals, tests, and scores. - Live framebuffer tiles. - Plots of timing, memory, and GPU batching learning. - Capability graph growing nodes: clock, gui.fb, input.predictor, fuse.compress, etc.

Next steps ---------- 1. Initialize repo and Compose. 2. Stand up API + Observer UI. 3. Wire senses and virtual actuators. 4. Add sandbox runner and the first three test harnesses. 5. Bring up the VM harness. 6. Start the promotion loop.